# Table of Contents

## Preface .................................................................................................................. 12
- Revision History ................................................................................................... 12
- Target Audience .................................................................................................. 12
- Obtaining Technical Assistance .......................................................................... 12
- Disclaimer ........................................................................................................... 13
- Reusing Code Samples ......................................................................................... 13
- Available Resources ............................................................................................ 14
- Accela Community ............................................................................................... 14
- Product Documentation ......................................................................................... 14
- Documentation Feedback ....................................................................................... 15

## Chapter 1
**Introduction ................................................................................................. 16**
- Understanding Events ......................................................................................... 17
- Example Use Cases ............................................................................................... 19
- Understanding Master Scripts .............................................................................. 23
  - Triggering Scripts .............................................................................................. 23
- Understanding Standard Choice Script Controls ................................................. 26
- Understanding Expression Builder Scripting ....................................................... 27

## Chapter 2
**Event and Script Setup ............................................................................ 30**
- Listing of Events and Master Scripts .................................................................... 30
- Working with Events ............................................................................................ 44
  - Searching for an Active Event .......................................................................... 44
  - Viewing the Full List of Accela Automation Events ........................................ 46
  - Enabling an Event ............................................................................................. 48
  - Disabling an Event ........................................................................................... 49
- Triggering Events ................................................................................................ 50
  - Triggering Meeting Agenda Events .................................................................. 50
  - Triggering Meeting Schedule Events .............................................................. 51
- Working with Scripts ........................................................................................... 52
  - Adding a Script ................................................................................................. 52
  - Searching for a Script ....................................................................................... 53
  - Editing a Script ................................................................................................. 54
  - Deleting a Script ............................................................................................... 54
# Table of Contents

## Chapter 3
**Master Scripts**  
56  
- Viewing Master Scripts ................................................................. 58  
- Understanding the EMSE Execution Path ........................................ 58  
- Creating a New Script ...................................................................... 60  
- Configuring the Universal Script .................................................... 60  
- Configuring Global Variables .......................................................... 62  
- Adding Custom Functions ............................................................... 63

## Chapter 4
**Script Controls**  
65  
- Understanding Script Controls .......................................................... 65  
- Understanding Script Control Syntax ................................................ 66  
- Understanding Case Sensitivity ....................................................... 66  
- Understanding Variable and Function Names .................................... 67  
- Understanding Curly Brackets ......................................................... 67  
- Understanding Argument Types ....................................................... 67  
- Understanding Criteria (the If Clause) ............................................... 67  
  - Understanding Criteria with Multiple Conditional Statements ....... 69  
- Understanding Actions (the Then Clause) .......................................... 70  
- Specifying Script Controls as Standard Choices ............................... 71  
- Understanding Script Control Branching ......................................... 73  
  - Using Branching to Implement a For Loop .................................... 75  
  - Using Pop-Up Messages ............................................................... 75  
  - Using Data Validation .................................................................... 77  
  - Using Variable Branching ............................................................. 78  
  - Branching to the Same Standard Choice from Different Events ...... 81  
- Naming Inspection Result Events ..................................................... 81  
- Exploring an Object ......................................................................... 82

## Chapter 5
**Accela Citizen Access Page Flow Scripts**  
84  
- Understanding Accela Citizen Access Page Flow Scripts ................. 84  
- Using Model Objects ....................................................................... 85  
- Creating a Page Flow Master Script .................................................. 85

## Chapter 6
**Script Testing**  
87  
- Understanding the Script Test Tool ................................................... 87  
- Testing an Event and Script Association ............................................ 90  
  - Associating the script to an event .................................................. 90  
  - Testing the event ........................................................................... 91  
- Running a Script Test ...................................................................... 91  
  - Using ScriptTester ......................................................................... 92  
- Troubleshooting .............................................................................. 93
### Table of Contents

- Launching the EMSE Debug Tool .............................................................. 93
- Understanding the Script Output Window .................................................. 95
- Setting the showDebug Script Control ....................................................... 97
- Using the aa.print Function ..................................................................... 98
- Using Biz Server Logs .............................................................................. 98

### Chapter 7

#### Accela Automation Object Model .......................................................... 100

- Discussing the Accela Automation Object Model ........................................ 100
- Understanding Script Return Values .......................................................... 112
  - ScriptReturnCode ..................................................................................... 112
  - ScriptReturnMessage ............................................................................... 113
  - ScriptReturnRedirection ........................................................................... 113

### Appendix A

#### Master Script Function List ................................................................. 114

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>activateTask</td>
<td>115</td>
</tr>
<tr>
<td>addAddressCondition</td>
<td>115</td>
</tr>
<tr>
<td>addAddressStdCondition</td>
<td>116</td>
</tr>
<tr>
<td>addAllFees</td>
<td>116</td>
</tr>
<tr>
<td>addAppCondition</td>
<td>117</td>
</tr>
<tr>
<td>addASITable</td>
<td>117</td>
</tr>
<tr>
<td>addASITable4ACAPageFlow</td>
<td>118</td>
</tr>
<tr>
<td>addContactStdCondition</td>
<td>118</td>
</tr>
<tr>
<td>addCustomFee</td>
<td>119</td>
</tr>
<tr>
<td>addFee</td>
<td>120</td>
</tr>
<tr>
<td>addFeeWithExtraData</td>
<td>120</td>
</tr>
<tr>
<td>addLicenseCondition</td>
<td>121</td>
</tr>
<tr>
<td>addLicenseStdCondition</td>
<td>122</td>
</tr>
<tr>
<td>addLookup</td>
<td>122</td>
</tr>
<tr>
<td>addParcelAndOwnerFromRefAddress</td>
<td>123</td>
</tr>
<tr>
<td>addParcelCondition</td>
<td>123</td>
</tr>
<tr>
<td>addParcelDistrict</td>
<td>124</td>
</tr>
<tr>
<td>addParent</td>
<td>124</td>
</tr>
<tr>
<td>addrAddCondition</td>
<td>124</td>
</tr>
<tr>
<td>addReferenceContactByName</td>
<td>125</td>
</tr>
<tr>
<td>addressExistsOnCap</td>
<td>125</td>
</tr>
<tr>
<td>addStdCondition</td>
<td>126</td>
</tr>
<tr>
<td>addTask</td>
<td>126</td>
</tr>
<tr>
<td>addTimeAccountingRecord</td>
<td>127</td>
</tr>
<tr>
<td>addTimeAccountingRecordToWorkflow</td>
<td>127</td>
</tr>
<tr>
<td>addToASITable</td>
<td>128</td>
</tr>
<tr>
<td>allTasksComplete</td>
<td>129</td>
</tr>
<tr>
<td>appHasCondition</td>
<td>129</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>applyPayments</td>
<td>130</td>
</tr>
<tr>
<td>appMatch</td>
<td>130</td>
</tr>
<tr>
<td>appNameIsUnique</td>
<td>131</td>
</tr>
<tr>
<td>assignCap</td>
<td>131</td>
</tr>
<tr>
<td>assignInspection</td>
<td>132</td>
</tr>
<tr>
<td>assignTask</td>
<td>132</td>
</tr>
<tr>
<td>autoAssignInspection</td>
<td>133</td>
</tr>
<tr>
<td>branch</td>
<td>133</td>
</tr>
<tr>
<td>capHasExpiredLicProf</td>
<td>134</td>
</tr>
<tr>
<td>capIdsFilterByFileDate</td>
<td>135</td>
</tr>
<tr>
<td>capIdsGetByAddr</td>
<td>135</td>
</tr>
<tr>
<td>capIdsGetByParcel</td>
<td>136</td>
</tr>
<tr>
<td>capSet</td>
<td>137</td>
</tr>
<tr>
<td>checkCapForLicensedProfessionalType</td>
<td>137</td>
</tr>
<tr>
<td>checkInspectionResult</td>
<td>138</td>
</tr>
<tr>
<td>childGetByCapType</td>
<td>138</td>
</tr>
<tr>
<td>closeCap</td>
<td>139</td>
</tr>
<tr>
<td>closeSubWorkflow</td>
<td>139</td>
</tr>
<tr>
<td>closeTask</td>
<td>140</td>
</tr>
<tr>
<td>comment</td>
<td>140</td>
</tr>
<tr>
<td>comparePeopleGeneric</td>
<td>141</td>
</tr>
<tr>
<td>completeCAP</td>
<td>142</td>
</tr>
<tr>
<td>contactAddFromUser</td>
<td>142</td>
</tr>
<tr>
<td>contactSetPrimary</td>
<td>142</td>
</tr>
<tr>
<td>contactSetRelation</td>
<td>143</td>
</tr>
<tr>
<td>convertDate</td>
<td>143</td>
</tr>
<tr>
<td>convertStringToPhone</td>
<td>143</td>
</tr>
<tr>
<td>copyAddresses</td>
<td>143</td>
</tr>
<tr>
<td>copyAppSpecific</td>
<td>144</td>
</tr>
<tr>
<td>copyASIFields</td>
<td>144</td>
</tr>
<tr>
<td>copyASITables</td>
<td>145</td>
</tr>
<tr>
<td>copyCalcVal</td>
<td>145</td>
</tr>
<tr>
<td>copyConditions</td>
<td>145</td>
</tr>
<tr>
<td>copyConditionsFromParcel</td>
<td>146</td>
</tr>
<tr>
<td>copyContacts</td>
<td>146</td>
</tr>
<tr>
<td>copyContactsByType</td>
<td>146</td>
</tr>
<tr>
<td>copyFees</td>
<td>147</td>
</tr>
<tr>
<td>copyLicensedProf</td>
<td>147</td>
</tr>
<tr>
<td>copyOwner</td>
<td>148</td>
</tr>
<tr>
<td>copyOwnersByParcel</td>
<td>148</td>
</tr>
<tr>
<td>copyParcelGisObjects</td>
<td>148</td>
</tr>
<tr>
<td>copyParcels</td>
<td>148</td>
</tr>
<tr>
<td>copySchedInspections</td>
<td>149</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>feeCopyByDateRange</td>
<td>169</td>
</tr>
<tr>
<td>feeExists</td>
<td>170</td>
</tr>
<tr>
<td>feeGetTotByDateRange</td>
<td>171</td>
</tr>
<tr>
<td>feeQty</td>
<td>171</td>
</tr>
<tr>
<td>getAddressConditions</td>
<td>171</td>
</tr>
<tr>
<td>getAppldByASI</td>
<td>172</td>
</tr>
<tr>
<td>getAppldByName</td>
<td>172</td>
</tr>
<tr>
<td>getApplication</td>
<td>173</td>
</tr>
<tr>
<td>getAppSpecific</td>
<td>173</td>
</tr>
<tr>
<td>getCapByAddress</td>
<td>174</td>
</tr>
<tr>
<td>getCAPConditions</td>
<td>174</td>
</tr>
<tr>
<td>getCapId</td>
<td>175</td>
</tr>
<tr>
<td>getCapsWithConditionsRelatedByRefContact</td>
<td>175</td>
</tr>
<tr>
<td>getChilden</td>
<td>175</td>
</tr>
<tr>
<td>getChildTasks</td>
<td>176</td>
</tr>
<tr>
<td>getConditions</td>
<td>176</td>
</tr>
<tr>
<td>getContactArray</td>
<td>177</td>
</tr>
<tr>
<td>getContactConditions</td>
<td>178</td>
</tr>
<tr>
<td>getCSLBInfo</td>
<td>178</td>
</tr>
<tr>
<td>getDepartmentName</td>
<td>179</td>
</tr>
<tr>
<td>getGISBufferInfo</td>
<td>180</td>
</tr>
<tr>
<td>getGISInfo</td>
<td>180</td>
</tr>
<tr>
<td>getGISInfoArray</td>
<td>181</td>
</tr>
<tr>
<td>getGuideSheetObjects</td>
<td>181</td>
</tr>
<tr>
<td>getInspector</td>
<td>182</td>
</tr>
<tr>
<td>getLastInspector</td>
<td>182</td>
</tr>
<tr>
<td>getLastScheduledInspector</td>
<td>183</td>
</tr>
<tr>
<td>getLicenseConditions</td>
<td>183</td>
</tr>
<tr>
<td>getLicenseProfessional</td>
<td>183</td>
</tr>
<tr>
<td>getParcelConditions</td>
<td>184</td>
</tr>
<tr>
<td>getParent</td>
<td>184</td>
</tr>
<tr>
<td>getParents</td>
<td>184</td>
</tr>
<tr>
<td>getRefLicenseProf</td>
<td>185</td>
</tr>
<tr>
<td>getRelatedCapsByAddress</td>
<td>185</td>
</tr>
<tr>
<td>getRelatedCapsByParcel</td>
<td>186</td>
</tr>
<tr>
<td>getReportedChannel</td>
<td>186</td>
</tr>
<tr>
<td>getScheduledInspId</td>
<td>187</td>
</tr>
<tr>
<td>getShortNotes</td>
<td>187</td>
</tr>
<tr>
<td>getTaskDueDate</td>
<td>187</td>
</tr>
<tr>
<td>getTaskStatusForEmail</td>
<td>188</td>
</tr>
<tr>
<td>hasPrimaryAddressInCap</td>
<td>188</td>
</tr>
<tr>
<td>insertSubProcess</td>
<td>188</td>
</tr>
<tr>
<td>inspCancelAll</td>
<td>189</td>
</tr>
<tr>
<td>invoiceFee</td>
<td>189</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>isScheduled</td>
<td>190</td>
</tr>
<tr>
<td>isTaskActive</td>
<td>190</td>
</tr>
<tr>
<td>isTaskComplete</td>
<td>191</td>
</tr>
<tr>
<td>isTaskStatus</td>
<td>191</td>
</tr>
<tr>
<td>jsDateToASIDate</td>
<td>192</td>
</tr>
<tr>
<td>jsDateToMMDDYYYY</td>
<td>192</td>
</tr>
<tr>
<td>licEditExpInfo</td>
<td>192</td>
</tr>
<tr>
<td>loadAddressAttributes</td>
<td>193</td>
</tr>
<tr>
<td>loadAppSpecific[4ACA]</td>
<td>193</td>
</tr>
<tr>
<td>loadASITable</td>
<td>194</td>
</tr>
<tr>
<td>loadASITables[4ACA][Before]</td>
<td>195</td>
</tr>
<tr>
<td>loadFees</td>
<td>195</td>
</tr>
<tr>
<td>loadGuideSheetItems</td>
<td>196</td>
</tr>
<tr>
<td>loadParcelAttributes</td>
<td>197</td>
</tr>
<tr>
<td>loadTasks</td>
<td>198</td>
</tr>
<tr>
<td>loadTaskSpecific</td>
<td>198</td>
</tr>
<tr>
<td>logDebug</td>
<td>198</td>
</tr>
<tr>
<td>lookup</td>
<td>199</td>
</tr>
<tr>
<td>lookupDateRange</td>
<td>199</td>
</tr>
<tr>
<td>lookupFeesByValuation</td>
<td>201</td>
</tr>
<tr>
<td>lookupFeesByValuationSlidingScale</td>
<td>202</td>
</tr>
<tr>
<td>loopTask</td>
<td>203</td>
</tr>
<tr>
<td>matches</td>
<td>204</td>
</tr>
<tr>
<td>nextWorkDay</td>
<td>204</td>
</tr>
<tr>
<td>openUrlInNewWindow</td>
<td>204</td>
</tr>
<tr>
<td>parcelConditionExists</td>
<td>205</td>
</tr>
<tr>
<td>parcelExistsOnCap</td>
<td>205</td>
</tr>
<tr>
<td>paymentByTrustAccount</td>
<td>205</td>
</tr>
<tr>
<td>paymentGetNotAppliedTot</td>
<td>206</td>
</tr>
<tr>
<td>proximity</td>
<td>206</td>
</tr>
<tr>
<td>proximityToAttribute</td>
<td>207</td>
</tr>
<tr>
<td>refLicProfGetAttribute</td>
<td>208</td>
</tr>
<tr>
<td>refLicProfGetDate</td>
<td>208</td>
</tr>
<tr>
<td>removeAllFees</td>
<td>209</td>
</tr>
<tr>
<td>removeASITable</td>
<td>209</td>
</tr>
<tr>
<td>removeCapCondition</td>
<td>209</td>
</tr>
<tr>
<td>removeFee</td>
<td>210</td>
</tr>
<tr>
<td>removeParcelCondition</td>
<td>210</td>
</tr>
<tr>
<td>removeTask</td>
<td>210</td>
</tr>
<tr>
<td>replaceMessageTokens</td>
<td>211</td>
</tr>
<tr>
<td>resultInspection</td>
<td>211</td>
</tr>
<tr>
<td>scheduleInspectDate</td>
<td>212</td>
</tr>
<tr>
<td>scheduleInspection</td>
<td>212</td>
</tr>
<tr>
<td>searchProject</td>
<td>213</td>
</tr>
</tbody>
</table>
setIVR ........................................................................................................... 213
setTask ......................................................................................................... 214
stripNN .......................................................................................................... 214
taskCloseAllExcept ....................................................................................... 215
taskStatus ..................................................................................................... 215
taskStatusDate .............................................................................................. 216
transferFunds .............................................................................................. 216
updateAddresses .......................................................................................... 217
updateAppStatus .......................................................................................... 217
updateFee .................................................................................................... 217
updateRefParcelToCap ................................................................................ 218
updateShortNotes ........................................................................................ 219
updateTask ................................................................................................... 219
updateTaskAssignedDate ............................................................................. 220
updateTaskDepartment ................................................................................ 220
updateWorkDesc .......................................................................................... 221
validateGisObjects ........................................................................................ 221
workDescGet ................................................................................................ 222
zeroPad ......................................................................................................... 222

Appendix B
Master Script
Object List..............................................................................................223
Fee................................................................................................................ 223
genericTemplateObject ................................................................................. 223
guideSheetObject .......................................................................................... 225
licenseProfObject .......................................................................................... 226
licenseObject ................................................................................................ 234
Task .............................................................................................................. 236

Appendix C
Example Expression Script..................................................................238

Appendix D
JavaScript Primer..................................................................................244
Understanding Scripts............................................................................... 244
Our First Example....................................................................................... 244
Writing And Testing Our First Script ......................................................... 246
Using Jext To Make Writing Scripts Easier........................................... 247
Using Variables .......................................................................................... 248
Numbers ................................................................................................... 250
Strings ...................................................................................................... 251
True and False ............................................................................................ 252
Arrays ....................................................................................................... 252
The Special Value "null" ............................................................................. 253
Objects ..................................................................................................... 254
# Table of Contents

Using Expressions ........................................................................................................... 254  
Mathematical Expressions ................................................................................................... 255  
String Expressions ............................................................................................................. 256  
Boolean Expressions ......................................................................................................... 257  
Relational Operators ......................................................................................................... 258  
Special Operators ............................................................................................................... 260  
Operator Precedence ........................................................................................................... 260  
Controlling What Happens Next ...................................................................................... 260  
  if … else ................................................................................................................................... 261  
  for ........................................................................................................................................... 262  
  while ....................................................................................................................................... 263  
  do … while .............................................................................................................................. 263  
Using Functions .................................................................................................................. 264  
Using Objects, Properties, and Methods ............................................................................. 265  
The Array Object .................................................................................................................. 265  
The Math Object ................................................................................................................... 266  
The String Object ................................................................................................................ 266  

## Appendix E

### Release Notes and Migration

- Execution FrameWork Changes ....................................................................................... 267  
- Script Control Sequencing Changes ................................................................................ 267  
- Upgrading from 1.x to 2.x ............................................................................................... 268  
  - Configuring the Global Variables .................................................................................... 268  
  - Migrating Custom Functions .......................................................................................... 269  
  - Installing Master Scripts ............................................................................................... 270  
- Updating Script Control Sequences .................................................................................. 270  
- Reinstating 1.x Script Control Sequencing ....................................................................... 271  
- Resolved Issues and Edits to Existing Scripts ................................................................. 271  
- New Master Scripts ........................................................................................................... 273  
- New Functions .................................................................................................................... 273
This document provides a consolidated source of information related to the Accela Automation master script framework.

Revision History

Table 1: Revision History provides a revision history of this document. This revision history summarizes changes made during each release of this document for the stated version of Accela Automation.

Table 1: Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 2014</td>
<td>Initial document release</td>
</tr>
</tbody>
</table>

Target Audience

This guide assumes the reader has a basic understanding of Accela Automation, a general understanding of programming concepts, and an understanding of the JavaScript programming language.

Generally, Accela Professional Services develops the necessary scripts as part of the system configuration and implementation effort. However, in some cases it may be necessary for an agency administrator to write some scripts. This individual should receive training from the agency’s Accela Project Manager or Accela Implementation Specialist before attempting any script writing. Improperly written scripts can seriously damage your system by incorrectly altering or deleting data for many records.

Obtaining Technical Assistance

As a starting point for all technical assistance, go to the Accela Customer Resource Center (CRC) website at www.accela.com/services/support-login. At this site you can search the knowledge base to find answers to commonly asked questions about our products and register at the Accela Forum to join in an information exchange with other Accela users.

If you still have questions after visiting Accela’s CRC site, or if you encounter any problems as you use the product, contact your system administrator. If you determine that you need professional technical assistance, have your agency’s designated contact call the CRC at (888) 7-ACCELA, ext. 5 or (888) 722-2352 ext. 5. The Accela CRC is available Monday through Friday from 4:00 AM to 6:00 PM Pacific Daylight/Standard Time.
Before you call please have this information available for the CRC representative:

- The Accela product name and version number
- Steps to reproduce the issue, including any error message or error number
- Screenshots, if possible
- Whether the problem is specific to a machine or to a user
- Exactly when the problem began
- Anything that changed on your computer or network (for example, did you install any new software?)
- A copy of your configuration file, if appropriate

Disclaimer

Your environment might look and function differently than the environment described in this guide. The feature set, portlet names, toolbar options, and the display settings described in this guide reflect default settings delivered with most new installations. The settings on your system can be different from these defaults depending on the implementation package for your agency, your user permissions, and the way that your system administrator sets up your system. Your system administrator can customize forms, drop-down lists, and also field labels throughout Accela Automation. Further, if your agency has installed any Accela add-on products, you might work with features or entire screens not explained in this guide. For information and instructions on how to use these additional features, see the documentation that came with the Accela add-on product.

Caution: Only experienced programmers or agency administrators should use the scripting feature. Improperly written scripts can adversely affect your system by incorrectly altering or deleting data. Make sure you write custom scripts carefully and test them before you implement them. You should receive some training, preferably from your Accela Project Manager or Implementation Specialist, before attempting to write scripts.

Reusing Code Samples

This document provides numerous code examples. Accela Automation does not guarantee that these code examples will work in your environment nor does Accela guarantee that these examples will produce the results you expect. Always make sure you fully test your scripts in a development environment before using them to alter production data.

Cutting and pasting code examples from this PDF document may introduce extraneous proprietary formatting elements into your script source and cause your script to produce unexpected results. Retype examples to ensure that you do not introduce any of these proprietary formatting elements into your script.
Available Resources

Accela Community

Accela hosts numerous discussions and forums on Accela Community that related to scripting. The following comprise a couple links:

- Master Script Distributions and Documentation ([http://community.accela.com/accela_automation/m/aascripts/default.aspx](http://community.accela.com/accela_automation/m/aascripts/default.aspx))
- Community Script Library ([http://community.accela.com/accela_automation/m/aascripts/default.aspx](http://community.accela.com/accela_automation/m/aascripts/default.aspx))

Product Documentation

*Accela Automation Release Notes.* This guide provides new features, related to the current release, for Accela Automation and add-on products. It also provides other types of release notes information such as known and fixed bugs, supported environments, and documentation corrections and clarifications.

The *Accela Automation Release Notes* also provides a listing of add-on products that work with Accela Automation and the documentation sets associated with those products.

*Accela Automation Installation Guide.* This guide provides instructions for installing Accela Automation, for upgrading Accela Automation from an earlier version, and for performing post installation configuration tasks such as setting up browsers, printers for a point of sale cashiering system, and so forth.

*Accela Automation Administrator Guide.* This guide instructs agency system administrators on how to set up and manage all the basic features of the Accela Automation application.

*Accela Automation User Guide.* This document provides instructions for using Accela Automation to perform daily tasks in Accela Automation. Daily tasks may include: managing applications, maintaining models, tracking fees and inspections, managing property and projects, running reports, managing business licensing, or managing code enforcement.


*Accela Automation On-premise Administrator Supplement.* This guide provides supplemental administrative tasks for agencies hosting their own deployments. If you use an Accela-hosted environment, you do not need to use this guide.

*Accela Automation Migration Guide.* This guide provides instructions for migrating an Accela Automation deployment from one environment to another. It provides instructions for performing a complete and incremental migrations.
Accela Automation Scripting Guide

Preface

This guide provides a high level overview of the main Accela Automation concepts.

Documentation Feedback

Accela’s technical publications team wants to provide you with the most accurate and useful documentation possible. We welcome your feedback in helping us improve future versions of this guide. If you have feedback and want to assist in improving the documentation, please send an email message to documentation@accela.com. Please include the product name and the version number, the title of the printed manual or online help, the specific topic (copy/paste the section you are referring to), and a detailed description of your suggestion.
Accela Automation provides a set of master script files that you configure to perform activities before or after an Accela Automation or Accela Citizen Access event (such as submitting an invoice). Accela Automation provides a separate master script file for each scriptable before and after event. Each master script file contains a global set of Accela Automation functions, that you configure through a Standard Choice script control, that ties the function to a specific before event or after event. Accela Automation controls the master script functions included within the master script files, and you should not change these functions.

Accela Automation also provides the Expression Builder interface to script form based interactions (auto-populating data fields based on user-selected values, for example) that occur before you trigger an event and master script activity.

The Event Manager Scripting Engine (EMSE) comprises the Accela Automation scripting platform. Accela Automation stores the master script files, written in JavaScript, in the Accela Automation database. Accela Automation uses the Rhino open source JavaScript engine to convert scripts into Java classes that Accela Automation executes through the EMSE API.

Figure 1: Accela Automation Scripting

Accela Automation launches scripts when the events that you associate with the script occur. You can use these event-triggered scripts to:

- Automate business processes
- Help save mouse-clicks
- Assess fees
- Update workflow
Enforce business rules
Custom data validation
Confirm event pre-requisites
Communicate
Send event driven email
Support Event / Batch driven data collection
Communicate/access web services, email, and interact with the file system

Figure 2: Master Script Flow of Execution shows the flow when you trigger an event with an associated master script.

Figure 2: Master Script Flow of Execution

Topics:
- Understanding Events
- Understanding Master Scripts
- Understanding Standard Choice Script Controls
- Understanding Expression Builder Scripting

Understanding Events

An action that a user performs through the Accela Automation user interface, clicking the Submit button to create a new record for example, constitutes an event (Figure 3: Launching an Accela Automation Event). These events initiate some sort of reaction that may affect other parts of your system. For example, when you create a new record and save it, Accela Automation updates information on your system, as required.
Other possible events include finding a record, assessing a fee, scheduling an inspection, and so forth. Accela Automation provides more than 200 events with which you can associate scripts. You cannot create new events, but you can choose the events to set up for your agency and disable the events that you do not use.

Each of the events includes a predefined set of variables that contain values about the event trigger. The associated master script can access these values.

You can trigger events from Accela Automation clients, such as Accela Mobile Office, Accela IVR, and Accela Citizen Access, or from integrated third-party products.

Accela Automation provides before and after event types (Figure 5: Triggered event process flow).

Figure 5: Triggered event process flow
A before event occurs before you save any data to the database. Scripts associated with before event types typically validate data to ensure the process saves clean and accurate data to the database. Accela Automation provides the word “before” in the suffix of before event names.

An after event occurs directly after Accela Automation saves submitted data to the database. Scripts associated with an after event implement automation of an action for the user. Accela Automation provides the word “after” in the suffix of after event names.

Example Use Cases

Scheduling
The following provides some example use cases that relate to scheduling. You can:

- Automatically update the task status in an inspection workflow when you schedule an inspection.
- Schedule an investigation inspection for the next business day after filing of a complaint.
- Check to ensure that all required inspections have passed, before scheduling a final inspection.

Assessing Fees
The following provides some example use cases for assessing fees. You can:

- Assess and invoice standard fees or assess and invoice application dependent fees.
- Check to ensure that the balance due for a record (permit or license, for example) is less than or equal to zero before issuance

Processing Documents
The following provides some example use cases that relate to document processing. You can:

- Email a PDF copy of a license, to the license holder, upon issuance or renewal.
- Check to ensure submission of all required documents, before processing an application.

Initiating and Configuring Communications
The following provides some example use cases that relate to initiating communications and configuring communications. You can:

- Initiate any kind of communication with events and scripts.
- Configure the title/subject and content of communication, by the event or action that initiates the communication.
- Configure each communication with a different title/subject and with different content according to the event or action that initiate the communication.
- Configure communications to have a different title/subject and content according to the recipients.
- Configure communications to have a different title/subject and content according to the type of communication (i.e. email, text message, or AA/ACA announcement).
Configure communications to have a different title/subject and content according to a custom set of agency-defined criteria. Examples include:

- Record type (4 level hierarchy or alias)
- Standard fields
- Template fields
- ASI
- Property information

**Attaching Communications to Records**

Accela Automation automatically attaches any outgoing emails to the license or case from which it originated. Accela Automation tracks the date and user that sent the correspondence along with the comments.

You can define whether the message subject, message body, bcc field, cc field, etc. dictates which emails Accela Automation retrieves, stores in the database, and attaches to corresponding records.

**Configuring Communication Recipients**

You can configure who receives communications, based on any of the following:

- Configuring communication recipients based on initiating event
  You can configure communications to have different recipients according to the event or action that initiates the communication.

- Configuring recipients based on type of communication
  You can configure communications to have different recipients according to the type of communication (i.e. email, text message, or AA/ACA announcement).

- Configuring communication recipients based on Accela Automation user profiles
  You configure scripts to send communications to one or more recipients based on their user profile in Accela Automation, including:
  - Agency
  - Organization (agency > bureau > division > section > group > office (department alias))
  - User Group
  - Individual users
  - Users with inspector status enabled (vs. disabled)

- Configuring communication recipients based on APO owners
  You can configure scripts to send communications to one or more recipients based on being in the reference APO database as an owner.

- Configuring communication recipients based on APO owners on a record
  You can configure scripts to send communications to one or more recipients based on being a property owner on a record (including a Work Order).

- Configuring communication recipients based on reference contacts
You can configure scripts to send communications to one or more recipients based on their contact type as a reference contact.

- Configuring communication recipients based on transaction contacts
  You can configure scripts to send communications to one or more recipients based on their contact type as a contact on a record (including a Work Order).
  Accela Automation prompts the user to email recipient(s) from a list of contacts associated with the license or case record.

- Configuring communication recipients based on reference licensed professionals
  You can configure scripts to send communications to one or more recipients based on their professional license:
  - All licensed professionals
  - Licensed professionals of one or more licensed professional types

- Configuring communication recipients based on licensed professionals associated with a record
  You can configure scripts to send communications to one or more recipients based on being a licensed professional on a record (including a Work Order).

- Configuring communication recipients based on Accela Citizen Access public user permissions
  You can configure scripts to send communications to one or more recipients based on their Accela Citizen Access public user permissions:
  - All public users
  - Anonymous public users
  - Registered public users
  - Record creator
  - Contact
  - Owner
  - Licensed Professional (any or specific)

- Configuring communication recipients based on their association with an inspection
  You can configure scripts to send communications to one or more recipients based on their association with an inspection:
  - Requestor
  - Contact
  - Inspector

- Configuring recipients based on their association with a workflow task
  You can configure scripts to send communications to one or more recipients based on their association to a workflow task:
  - Action By Department
  - Action By User
• Assigned to Department
• Assigned to User

- Configuring recipients based on their association with a condition assessment
  You can configure scripts to send communications to one or more recipients based on their association with a condition assessment:
  • Department
  • Inspector

- Configuring recipients based on their assignment to an activity
  You can configure communications to be send to one or more recipients assigned to an activity.

- Configuring recipients of the communication by agency-defined criteria (i.e. set)
  You can configure scripts to send communications to a set of recipients according to agency-defined criteria.
  Examples of criteria that you can use to create a set include:
  • All contacts on records of a designated record type (4 level hierarchy or alias)
  • All licensed professionals associated with records that contain designated values in standard fields, template fields or ASI fields
  • All owners of property according to some selection criteria such as range of addresses or proximity to a location
  • Any other set of recipients as defined by the agency

**Preventing Duplicate Communications**

When you properly configure the communication event script, for each type of communication with the same subject and same content, a single person can receive only one of each type of communication, even if they are members of more than one group of recipients.

For example: A person may be part of an agency organization (agency > bureau > division > section > group > office) and also part of an agency group (building clerk).

- If you configure an email to announce scheduled maintenance to members of this organization and also this group, the person only receives one email.
- If you configure an email and a text message for members of this organization and also this group, the person receives one email and one text message.

**Configuring Email and Text Message Sent-from Values**

You can configure emails and text messages to have different “from” values, according to the initiating event/action, type of communication, or other agency-defined criteria.
Understanding Master Scripts

Accela Automation uses scripts to perform the custom activities that extend standard event processing. When run, a script produces an effect on the objects defined in your system, such as records, parcels, addresses, and so forth.

Accela Automation provides a set of master script files that extend functionality for events. For some events, Accela Automation provides a master script file unique to that event. For the other events, Accela Automation provides a universal master script that you can use as a template for development of an event-specific script.

Accela Automation provides the following three global master script files that each event-specific master script includes during runtime.

- INCLUDES_ACCELA_FUNCTIONS
- INCLUDES_ACCELA_FUNCTIONS_ASB
- INCLUDES_ACCELA_GLOBALS

These global master script files contain the set of functions Accela Automation uses during execution of each of the event-specific scripts.

Note: Accela Automation does not support changes to or overrides of master script files, especially the functions that three global master scripts include.

Triggering Scripts

You can trigger a script from an event (Understanding Events on page 17), a batch job, a set script or a script test.

Topics:
- Batch Jobs
- Set Scripts
- Script Test

Batch Jobs

Batch jobs trigger scripts through a scheduled job in contrast to a user-invoked action. For example, you can schedule a nightly batch job, with an associated script, that looks for expired permits or licenses and updates them to an expired application and/or expiration status. At a high level batch scripts contain instructions to query records based on a specified filter, evaluate each returned record and take action for each record according to certain criteria. Accela Automation provides the Batch Job portlet (Figure 6: Batch Jobs Portlet) from where you can use UI controls to set parameters for the associated batch job script.
In addition, Accela Automation provides a batch job transaction manager for you to control transactions by scripts. The batch job transaction manager uses the following three methods to begin, commit, and roll back transactions separately.

```java
aa.batchJob.beginTransaction(int seconds)
aa.batchJob.commitTransaction()
aa.batchJob.rollbackTransaction()
```

For more information about transaction manager, see [http://docs.oracle.com/javaee/6/api/javax/transaction/TransactionManager.html](http://docs.oracle.com/javaee/6/api/javax/transaction/TransactionManager.html).

**Note:** There are some limitations when using the batch job transaction manager.

- Every time before invoking `commitTransaction()` or `rollbackTransaction()`, invoke `beginTransaction(int seconds)` first.
- The batch job transaction manager does not support event triggering scripts.
- The batch job transaction manager does not support nested transactions. For more information about nested transaction, see [http://en.wikipedia.org/wiki/Nested_transaction](http://en.wikipedia.org/wiki/Nested_transaction).
Set Scripts

You can associate a set script to the “Execute Script” button on the Set portlet (Figure 7: Set Portlet). The script contains instructions to evaluate each member (record) of the selected set and take action if the member falls into the specified criteria.

Figure 7: Set Portlet

Example Use Case

Manage an invoicing process.

- Run a batch script to evaluate records and determine if you require an invoice. If so, add the record to a set.
- Review the generated set for accuracy; add or remove records as required.
- Execute the script from the set portlet.
- The script evaluates each record; if it meets specified criteria take the appropriate action (eg. update the record, send an email or generate invoices reports).

Script Test

Accela Automation provides the Script Test tool for EMSE script writers. This tool enables you to enter and execute EMSE scripts with no affect on the Accela Automation database. The
script writer can evaluate the output of the script to determine further development effort and testing. You can use the Script Test tool to:

- Develop and test batch scripts.
- Develop and test custom functions.
- Troubleshoot and debug EMSE scripts.

**Figure 8: Script Test Tool**

Understanding Standard Choice Script Controls

You connect an event to a script through a comparably named Standard Choice script control. The script control calls functions from the global master script files, included in each script file, and passes parameters to these functions to control how the script interacts with the event (**Figure 9: Standard Choice Script Control**).
Understanding Expression Builder Scripting

**Note:** Accela Automation uses the Event Manager and Scripting Engine (EMSE) to handle default form and portlet data fields.

Expression Builder provides an interface to script client side interactions (or expressions) before triggering an event type, submitting a form for example, handled by the master script framework. You can use Expression Builder to define expressions that trigger when a form loads, or when a user selects or enters a value in an individual form field. You can use expressions to perform calculations, provide drop-down lists, or auto-populate data fields based on user-selected values.

**Example Use Case**

A user selects a value from a drop-down list in ASI. You create a script for an expression that makes the selected value affect other fields in the form to:

- Mark fields as required.
- Mark fields as read-only or hidden as they are no longer required.
- Pre-populate them based on a calculation or lookup table.
- Trigger an alert pop-up window or alert message next to other fields.

**Example Use Case**

A user enters a permit number or license number in an ASI text field. Accela Automation provides a message about the validity of the permit or license number before the user submits the form.

Expressions implement business rules that require users to receive immediate feedback in the user interface before they submit a form. You can use “before” events in the master script...
framework to perform a similar type of form validation. However, with the master script framework, the user must complete the entire form and submit it before receiving feedback. With expressions, the user receives feedback immediately upon completing an individual field on the form.

Expression Builder provides a wizard to create expressions (see *Accela Automation Administrator’s Guide*) and Accela Automation generates scripts to implement the expressions that you create through the wizard. You can view and edit the generated expression scripts in an Expression Builder window when you toggle Expression Builder from wizard mode to script mode (Figure 10: Expression Builder Portlet). You can select whether to execute the expressions for Accela Automation only, Accela Citizen Access only, or both.

Figure 10: Expression Builder Portlet

You can use a combination of field level and form level data validation, depending on your business needs. You can trigger an master script from an expression. See the discussion thread on Accela Community for more information about Expression Builder ([http://community.accela.com/search/SearchResults.aspx?q=expression+builder](http://community.accela.com/search/SearchResults.aspx?q=expression+builder)).

You can configure an expression script to populate form data from an external data source, through an external web services. When connected to an external web service, administrators can generate expressions that use data elements, from an external web service, as variables or data items.
Example Use Case

An agency administrator uses Expression Builder to build and execute an expression for the License Professional portlet. The script interacts with an external web service, such as the State Licensing Board, to check for the current status of a license and whether the Licensed Professional selected in a new application is valid.

Appendix C: Example Expression Script on page 238 provides a detailed example of an expression script.
Listing of Events and Master Scripts

Accela Automation provides an event manager interface, consisting of a collection of web pages, to identify events and their associated script (Figure 11: Events and Associated Scripts).

Figure 11: Events and Associated Scripts

Table 2: Event and Master Script List provides the list of scriptable Accela Automation events and whether an Out-Of-The-Box (OOTB) master script (Working with Scripts on page 52) is provided to associate with the event. If a master script is not provided for an event, you can easily create your own.

Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event</th>
<th>Associated Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressUpdateAfter</td>
<td>AddressUpdateAfter</td>
</tr>
<tr>
<td>AddressUpdateBefore</td>
<td>AddressUpdateBefore</td>
</tr>
<tr>
<td>ActivityDeleteAfter</td>
<td>ActivityDeleteAfter</td>
</tr>
<tr>
<td>ActivityDeleteBefore</td>
<td>ActivityDeleteBefore</td>
</tr>
<tr>
<td>ActivityInsertAfter</td>
<td>ActivityInsertAfter</td>
</tr>
<tr>
<td>ActivityInsertBefore</td>
<td>ActivityInsertBefore</td>
</tr>
<tr>
<td>ActivityUpdateAfter</td>
<td>ActivityUpdateAfter</td>
</tr>
<tr>
<td>AddressLookupAfter</td>
<td>AddressLookupAfter</td>
</tr>
<tr>
<td>AddressLookupBefore</td>
<td>AddressLookupBefore</td>
</tr>
<tr>
<td>ApplicationConditionAddAfter</td>
<td>ApplicationConditionAddAfter</td>
</tr>
<tr>
<td>ApplicationConditionAddBefore</td>
<td>ApplicationConditionAddBefore</td>
</tr>
<tr>
<td>ApplicationConditionDeleteAfter</td>
<td>ApplicationConditionDeleteAfter</td>
</tr>
<tr>
<td>ApplicationConditionDeleteBefore</td>
<td>ApplicationConditionDeleteBefore</td>
</tr>
<tr>
<td>ApplicationConditionUpdateAfter</td>
<td>ApplicationConditionUpdateAfter</td>
</tr>
</tbody>
</table>

In most cases, you can understand the nature of events by the event name and Accela Automation provides before and after events with the same trigger. For example, Accela Automation triggers the AAAddressUpdateAfter and the AAAddressUpdateBefore when the user submits an address update.
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAddressUpdateAfter</td>
<td>The after event for when a user updates a daily address.</td>
<td></td>
</tr>
<tr>
<td>AAAddressUpdateBefore</td>
<td>The before event for when a user updates a daily address.</td>
<td></td>
</tr>
<tr>
<td>AddContractLicenseAfter</td>
<td>The after event for when agency administrators associate a licensed professional with the public user account for an external inspector.</td>
<td></td>
</tr>
<tr>
<td>AddContractLicenseBefore</td>
<td>The before event for when agency administrators associate a licensed professional with the public user account for an external inspector.</td>
<td></td>
</tr>
<tr>
<td>AAOwnerUpdateAfter</td>
<td>The after event for when a user updates a daily owner.</td>
<td></td>
</tr>
<tr>
<td>AAOwnerUpdateBefore</td>
<td>The before event for when a user updates a daily owner.</td>
<td></td>
</tr>
<tr>
<td>ActivityDeleteAfter</td>
<td>The after event for when a user deletes an activity.</td>
<td></td>
</tr>
<tr>
<td>ActivityDeleteBefore</td>
<td>The before event for when a user deletes an activity.</td>
<td></td>
</tr>
<tr>
<td>ActivityInsertAfter</td>
<td>The after event for when a user inserts an activity.</td>
<td></td>
</tr>
<tr>
<td>ActivityInsertBefore</td>
<td>The before event for when a user inserts an activity.</td>
<td></td>
</tr>
<tr>
<td>ActivityUpdateAfter</td>
<td>The after event for when a user updates an activity.</td>
<td></td>
</tr>
<tr>
<td>ActivityUpdateBefore</td>
<td>The before event for when a user updates an activity.</td>
<td></td>
</tr>
<tr>
<td>AddAgendaAfter</td>
<td>The after event for when a user adds an agenda.</td>
<td></td>
</tr>
<tr>
<td>AddAgendaBefore</td>
<td>The before event for when a user adds an agenda.</td>
<td></td>
</tr>
<tr>
<td>AdditionalInfoUpdateAfter</td>
<td>The after event for when a user updates additional information.</td>
<td>✓</td>
</tr>
<tr>
<td>AdditionalInfoUpdateBefore</td>
<td>The before event for when a user updates additional information.</td>
<td>✓</td>
</tr>
<tr>
<td>AddLicenseToPublicUserAfter</td>
<td>Accela Citizen Access - The after event for when a user adds a license to a public user.</td>
<td></td>
</tr>
<tr>
<td>AddLicenseValidation4ACA</td>
<td>Accela Citizen Access - The after event for when a user adds a license to a user account.</td>
<td></td>
</tr>
<tr>
<td>AddressAddAfter</td>
<td>The after event for when a user creates an address.</td>
<td></td>
</tr>
<tr>
<td>AddressAddBefore</td>
<td>The before event for when a user creates an address.</td>
<td></td>
</tr>
<tr>
<td>AddressConditionAddAfter</td>
<td>The after event for when a user adds a condition to an address.</td>
<td></td>
</tr>
<tr>
<td>AddressLookUpAfter</td>
<td>The after event for when a user creates a reference address after looking up an address from reference.</td>
<td></td>
</tr>
<tr>
<td>AddressLookUpBefore</td>
<td>The before event for when a user creates a reference address after looking up an address from reference.</td>
<td></td>
</tr>
<tr>
<td>AddressRemoveAfter</td>
<td>The after event for when a user removes an address from the daily side.</td>
<td></td>
</tr>
<tr>
<td>AddressRemoveBefore</td>
<td>The before event for when a user removes an address from the daily side.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressSelectOnSpearFormAfter</td>
<td>The after event for when a user attaches selected addresses on the ref addresses look up result list portlet.</td>
<td></td>
</tr>
<tr>
<td>AddressSelectOnSpearFormBefore</td>
<td>The before event for when a user attaches selected addresses on the ref addresses look up result list portlet.</td>
<td></td>
</tr>
<tr>
<td>AddressSetDetailUserExecuteAfter</td>
<td>The after event for when a user executes an address set script.</td>
<td></td>
</tr>
<tr>
<td>AddressUpdateAfter</td>
<td>The after event for when a user updates an address.</td>
<td></td>
</tr>
<tr>
<td>AddressUpdateBefore</td>
<td>The before event for when a user updates an address.</td>
<td></td>
</tr>
<tr>
<td>AppHierarchyAddAfter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AppHierarchyAddBefore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AppHierarchyDeleteAfter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AppHierarchyDeleteBefore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApplicationConditionAddAfter</td>
<td>The after event for when a user adds an application condition task.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationConditionAddBefore</td>
<td>The before event for when a user adds an application condition task.</td>
<td></td>
</tr>
<tr>
<td>ApplicationConditionBatchUpdateAfter</td>
<td>The after event for when a user updates conditions of approvals.</td>
<td></td>
</tr>
<tr>
<td>ApplicationConditionDeleteAfter</td>
<td>The after event for when a user deletes an application condition task.</td>
<td></td>
</tr>
<tr>
<td>ApplicationConditionDeleteBefore</td>
<td>The before event for when a user deletes an application condition task.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationConditionOfApprovalUpdateAfter</td>
<td>The after event for when a user updates a condition of approval.</td>
<td></td>
</tr>
<tr>
<td>ApplicationConditionOfApprovalUpdateBefore</td>
<td>The before event for when a user updates a condition of approval.</td>
<td></td>
</tr>
<tr>
<td>ApplicationConditionUpdateAfter</td>
<td>The after event for when a user updates an application condition task.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationConditionUpdateBefore</td>
<td>The before event for when a user updates an application condition task.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationDeleteAfter</td>
<td>The after event for when a user deletes a record.</td>
<td></td>
</tr>
<tr>
<td>ApplicationDeleteBefore</td>
<td>The before event for when a user deletes a record.</td>
<td></td>
</tr>
<tr>
<td>ApplicationDetailNewAfter</td>
<td>The after event for when a user creates an application detail.</td>
<td></td>
</tr>
<tr>
<td>ApplicationDetailNewBefore</td>
<td>The before event for when a user creates an application detail.</td>
<td></td>
</tr>
<tr>
<td>ApplicationDetailUpdateAfter</td>
<td>The after event for when a user updates an application detail.</td>
<td></td>
</tr>
<tr>
<td>ApplicationDetailUpdateBefore</td>
<td>The before event for when a user updates an application detail.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationGISGovXMLSubmitAfter</td>
<td>The after event for when a user creates an application through Accela GIS GovXML.</td>
<td></td>
</tr>
<tr>
<td>ApplicationSelectAfter</td>
<td>The after event for when a user selects an application.</td>
<td></td>
</tr>
<tr>
<td>ApplicationSelectBefore</td>
<td>The before event for when a user selects an application.</td>
<td></td>
</tr>
<tr>
<td>ApplicationSpecificInfoUpdateAfter</td>
<td>The after event for when a user updates application specific information.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationSpecificInfoUpdateBefore</td>
<td>The before event for when a user updates application specific information.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationStatusUpdateAfter</td>
<td>The after event, that adds a history record, when a user updates application status.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationStatusUpdateBefore</td>
<td>The before event for when a user updates application status.</td>
<td>✓</td>
</tr>
<tr>
<td>ApplicationSubmitAfter</td>
<td>The after event for when a user creates a record according to the following scenarios:</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>• The createCap web service operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The initiateCAP GovXML operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The user interface of Accela Automation, Accela Citizen Access, Accela Mobile Office, Accela Wireless, and Accela IVR</td>
<td></td>
</tr>
<tr>
<td>ApplicationSubmitBefore</td>
<td>The before event for when a user creates a record in the following scenarios:</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>• The createCap web service operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The initiateCAP GovXML operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The user interface of Accela Automation, Accela Citizen Access, Accela Mobile Office, Accela Wireless, and Accela IVR</td>
<td></td>
</tr>
<tr>
<td>ApproveContactAssociationforPublicAfter</td>
<td>The after event for when a user approves a contact association for a public user.</td>
<td></td>
</tr>
<tr>
<td>AssetSubmitAfter</td>
<td>The after event for when a user creates an asset.</td>
<td></td>
</tr>
<tr>
<td>AssetSubmitBefore</td>
<td>The before event for when a user updates an asset.</td>
<td></td>
</tr>
<tr>
<td>AssetUpdateAfter</td>
<td>The after event for when a user updates an asset.</td>
<td></td>
</tr>
<tr>
<td>AssetUpdateBefore</td>
<td>The before event for when a user updates an asset.</td>
<td></td>
</tr>
<tr>
<td>AssociateAssetToWorkOrderAfter</td>
<td>The after event for when a user associates an asset to a work order.</td>
<td></td>
</tr>
<tr>
<td>AssociateAssetToWorkOrderBefore</td>
<td>The before event for when a user associates an asset to a work order.</td>
<td></td>
</tr>
<tr>
<td>AuditSetDetailUserExecuteAfter</td>
<td>The after event for when a user executes a script on a random audit set.</td>
<td></td>
</tr>
<tr>
<td>AutoPayAfter</td>
<td>The after event for when a user submits an auto payment.</td>
<td></td>
</tr>
<tr>
<td>AutoPayBefore</td>
<td>The before event for when a user submits an auto payment.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchResultInspectionByCSVAfter</td>
<td>The after event for when external inspectors upload CSV files that contain inspection results or when agency users update a batch of inspection results according to the inspection result CSV file that a contract inspector or a self-certified inspector submits. This event triggers in both Accela Automation and Accela Citizen Access.</td>
<td></td>
</tr>
<tr>
<td>BatchResultInspectionByCSVBefore</td>
<td>The before event for when external inspectors upload CSV files that contain inspection results or when agency users update a batch of inspection results according to the inspection result CSV file that a contract inspector or a self-certified inspector submits.</td>
<td></td>
</tr>
<tr>
<td>CAEConditionAddAfter</td>
<td>The after event for when a user adds a condition to a CAE.</td>
<td></td>
</tr>
<tr>
<td>CapSetDetailUserExecuteAfter</td>
<td>Occurs after the record set script executes.</td>
<td></td>
</tr>
<tr>
<td>CommunicationReceivingEmailBefore</td>
<td>The before event for when Accela Automation receives an email from the email server.</td>
<td></td>
</tr>
<tr>
<td>CommunicationReceivingEmailAfter</td>
<td>The after event for when Accela Automation receives an email from the email server.</td>
<td></td>
</tr>
<tr>
<td>CommunicationSendingEmailBefore</td>
<td>The before event for when Accela Automation sends an email.</td>
<td></td>
</tr>
<tr>
<td>CommunicationSendingEmailAfter</td>
<td>The after event for when Accela Automation sends an email.</td>
<td></td>
</tr>
<tr>
<td>ConditionAssessmentSubmitAfter</td>
<td>The after event for when a user creates a condition assessment.</td>
<td></td>
</tr>
<tr>
<td>ConditionAssessmentSubmitBefore</td>
<td>The before event for when a user creates a condition assessment.</td>
<td></td>
</tr>
<tr>
<td>ConditionAssessmentUpdateAfter</td>
<td>The after event for when a user updates a condition assessment.</td>
<td></td>
</tr>
<tr>
<td>ConditionAssessmentUpdateBefore</td>
<td>The before event for when a user updates a condition assessment.</td>
<td></td>
</tr>
<tr>
<td>ContactAddAfter</td>
<td>The after event for when a user adds a contact.</td>
<td>✓</td>
</tr>
<tr>
<td>ContactAddBefore</td>
<td>The before event for when a user adds a contact.</td>
<td>✓</td>
</tr>
<tr>
<td>ContactAddressDeactivateAfter</td>
<td>The after event for when a user deactivates a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactAddressDeactivateBefore</td>
<td>The before event for when a user deactivates a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactAddressEditAfter</td>
<td>The after event for when a user edits a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactAddressEditBefore</td>
<td>The before event for when a user edits a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactAddressLookUpAfter</td>
<td>The after event for when a user looks up a contact address.</td>
<td></td>
</tr>
<tr>
<td>Event Name</td>
<td>Description</td>
<td>Master Script?</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ContactAddressLookUpBefore</td>
<td>The before event for when a user looks up a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactAddressNewAfter</td>
<td>The after event for when a user adds a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactAddressNewBefore</td>
<td>The before event for when a user adds a contact address.</td>
<td></td>
</tr>
<tr>
<td>ContactEditAfter</td>
<td>The after event for when a user edits a contact.</td>
<td>✓</td>
</tr>
<tr>
<td>ContactEditBefore</td>
<td>The before event for when a user edits a contact.</td>
<td>✓</td>
</tr>
<tr>
<td>ContactLookUpAfter</td>
<td>The after event for when a user adds a reference contact to a record.</td>
<td></td>
</tr>
<tr>
<td>ContactLookUpBefore</td>
<td>The before event for when a user adds a reference contact to a record.</td>
<td></td>
</tr>
<tr>
<td>ContactRelatedToPublicUserAfter</td>
<td>Executes after users associate a reference contact with the public user account in Accela Automation or Accela Citizen Access.</td>
<td></td>
</tr>
<tr>
<td>ContactRelatedToPublicUserBefore</td>
<td>Executes before users associate a reference contact with the public user account in Accela Automation or Accela Citizen Access.</td>
<td></td>
</tr>
<tr>
<td>ContactRemoveAfter</td>
<td>The after event for when a user removes a contact.</td>
<td>✓</td>
</tr>
<tr>
<td>ContactRemoveBefore</td>
<td>The before event for when a user removes a contact.</td>
<td>✓</td>
</tr>
<tr>
<td>ContactUpdateAfter</td>
<td>The after event for when a user updates a contact.</td>
<td></td>
</tr>
<tr>
<td>ContactUpdateBefore</td>
<td>The before event for when a user updates a contact.</td>
<td></td>
</tr>
<tr>
<td>ContinuingEducationUpdateAfter</td>
<td>The after event for when a user commits continuing education.</td>
<td></td>
</tr>
<tr>
<td>ConvertToRealCAPAfter</td>
<td>Accela Citizen Access - The after event for converting a partial record ID to a real record ID.</td>
<td>✓</td>
</tr>
<tr>
<td>ConvertToRealCAPBefore</td>
<td>Accela Citizen Access - The before event for converting a partial record ID to a real record ID.</td>
<td></td>
</tr>
<tr>
<td>DailyActivityDeleteAfter</td>
<td>The after event for when a user deletes a daily activity.</td>
<td></td>
</tr>
<tr>
<td>DailyActivityDeleteBefore</td>
<td>The before event for when a user deletes a daily activity.</td>
<td></td>
</tr>
<tr>
<td>DailyActivityNewAfter</td>
<td>The after event for when a user creates a new daily activity.</td>
<td></td>
</tr>
<tr>
<td>DailyActivityNewBefore</td>
<td>The before event for when a user creates a new daily activity.</td>
<td></td>
</tr>
<tr>
<td>DailyActivityUpdateAfter</td>
<td>The after event for when a user updates a daily activity.</td>
<td></td>
</tr>
<tr>
<td>DailyActivityUpdateBefore</td>
<td>The before event for when a user updates a daily activity.</td>
<td></td>
</tr>
<tr>
<td>DeleteContractLicenseAfter</td>
<td>The after event for when agency administrators disassociate a licensed professional with the public user account for an external inspector.</td>
<td></td>
</tr>
<tr>
<td>DeleteContractLicenseBefore</td>
<td>The before event for when agency administrators disassociate a licensed professional with the public user account for an external inspector.</td>
<td></td>
</tr>
<tr>
<td>Event Name</td>
<td>Description</td>
<td>Master Script?</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>DocumentDeleteAfter</td>
<td>The after event for when a user deletes one or more documents.</td>
<td></td>
</tr>
<tr>
<td>DocumentDeleteBefore</td>
<td>The before event for when a user deletes one or more documents.</td>
<td></td>
</tr>
<tr>
<td>DocumentReviewAddAfter</td>
<td>The after event for when a user assigns one or more document reviewers.</td>
<td></td>
</tr>
<tr>
<td>DocumentReviewAddBefore</td>
<td>The before event for when a user assigns one or more document reviewers.</td>
<td></td>
</tr>
<tr>
<td>DocumentReviewDeleteAfter</td>
<td>The after event for when a user deletes one or more document reviewers.</td>
<td></td>
</tr>
<tr>
<td>DocumentReviewDeleteBefore</td>
<td>The before event for when a user deletes one or more document reviewers.</td>
<td></td>
</tr>
<tr>
<td>DocumentReviewUpdateAfter</td>
<td>The after event for when a user updates a document reviewer.</td>
<td></td>
</tr>
<tr>
<td>DocumentReviewUpdateBefore</td>
<td>The before event for when a user updates a document reviewer.</td>
<td></td>
</tr>
<tr>
<td>DocumentUpdateAfter</td>
<td>The after event for when a user updates document information.</td>
<td>✓</td>
</tr>
<tr>
<td>DocumentUpdateBefore</td>
<td>The before event for when a user updates document information.</td>
<td>✓</td>
</tr>
<tr>
<td>DocumentUploadAfter</td>
<td>Accela Citizen Access - The after event for when a user uploads a document or when an external inspector uploads a CSV file containing inspection results.</td>
<td></td>
</tr>
<tr>
<td>DocumentUploadBefore</td>
<td>Accela Citizen Access - The before event for when a user uploads a document.</td>
<td></td>
</tr>
<tr>
<td>EducationUpdateAfter</td>
<td>The after event for when a user updates education.</td>
<td></td>
</tr>
<tr>
<td>EstablishmentAddAfter</td>
<td>The after event for when a user adds an establishment.</td>
<td></td>
</tr>
<tr>
<td>EstablishmentAddBefore</td>
<td>The before event for when a user adds an establishment.</td>
<td></td>
</tr>
<tr>
<td>EstablishmentUpdateAfter</td>
<td>The after event for when a user updates an establishment.</td>
<td></td>
</tr>
<tr>
<td>EstablishmentUpdateBefore</td>
<td>The after event for when a user updates an establishment.</td>
<td></td>
</tr>
<tr>
<td>EventAddAfter</td>
<td>The after event for when calendar event information is created.</td>
<td></td>
</tr>
<tr>
<td>EventAddBefore</td>
<td>The before event for when calendar event information is created.</td>
<td></td>
</tr>
<tr>
<td>EventRemoveAfter</td>
<td>The after event for when calendar event information is removed.</td>
<td></td>
</tr>
<tr>
<td>EventRescheduleAfter</td>
<td>The after event for when calendar event datetime is changed.</td>
<td></td>
</tr>
<tr>
<td>EventRescheduleBefore</td>
<td>The before event for when calendar event datetime is changed.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventUpdateAfter</td>
<td>The after event for when calendar event information is updated.</td>
<td></td>
</tr>
<tr>
<td>EventUpdateBefore</td>
<td>The before event for when calendar event information is updated.</td>
<td></td>
</tr>
<tr>
<td>ExaminationBatchUpdateByCSVAfter</td>
<td>Accela Citizen Access - The after event for when a user uploads a CSV file to batch update examination.</td>
<td></td>
</tr>
<tr>
<td>ExaminationRosterUpdateAfter</td>
<td>The after event for when a user updates the examination roster (register roster, reschedule roster, delete roster, update score).</td>
<td></td>
</tr>
<tr>
<td>ExaminationSiteUpdateAfter</td>
<td>The after event for when a user updates an examination site.</td>
<td></td>
</tr>
<tr>
<td>ExaminationUpdateAfter</td>
<td>The after event for when a user updates an examination.</td>
<td></td>
</tr>
<tr>
<td>ExaminationUpdateBefore</td>
<td>The before event for when a user updates an examination.</td>
<td></td>
</tr>
<tr>
<td>ExternalDocReviewCompleted</td>
<td>The after event for when a user checks in a record document.</td>
<td></td>
</tr>
<tr>
<td>ExternalPermitStatusChange</td>
<td>The after event for when a user updates record status.</td>
<td></td>
</tr>
<tr>
<td>FeeAssessAfter</td>
<td>The after event for when a user assesses an application fee.</td>
<td>✓</td>
</tr>
<tr>
<td>FeeAssessBefore</td>
<td>The before event for when a user assesses an application fee.</td>
<td>✓</td>
</tr>
<tr>
<td>FeeEstimate4PlanReviewBefore</td>
<td>The before event for when a user creates a fee estimate for plan review.</td>
<td></td>
</tr>
<tr>
<td>FeeEstimateAfter</td>
<td>The after event for when a user creates a fee estimate in an application intake form.</td>
<td></td>
</tr>
<tr>
<td>FeeEstimateAfter4ACA</td>
<td>Accela Citizen Access - The after event for when a user creates a fee estimate in the fee item list page.</td>
<td>✓</td>
</tr>
<tr>
<td>FundTransferAfter</td>
<td>The after event for when a user transfers a fund.</td>
<td></td>
</tr>
<tr>
<td>FundTransferBefore</td>
<td>The before event for when a user transfers a fund.</td>
<td></td>
</tr>
<tr>
<td>GuidesheetUpdateAfter</td>
<td>The after event for when a user updates a guidesheet.</td>
<td></td>
</tr>
<tr>
<td>GuidesheetUpdateBefore</td>
<td>The before event for when a user updates a guidesheet.</td>
<td></td>
</tr>
<tr>
<td>InspectionAssignAfter</td>
<td>The after event for when a user assigns an inspection.</td>
<td></td>
</tr>
<tr>
<td>InspectionAssignBefore</td>
<td>The before event for when a user assigns an inspection.</td>
<td></td>
</tr>
<tr>
<td>InspectionCancelAfter</td>
<td>The after event for when a user cancels one or more inspections.</td>
<td></td>
</tr>
<tr>
<td>InspectionCancelBefore</td>
<td>The before event for when a user cancels one or more inspections.</td>
<td></td>
</tr>
<tr>
<td>InspectionMultipleScheduleAfter</td>
<td>The after event for when a user schedules one or more inspections for manage inspection.</td>
<td>✓</td>
</tr>
<tr>
<td>InspectionMultipleScheduleBefore</td>
<td>The before event for when a user schedules one or more inspections for manage inspection.</td>
<td>✓</td>
</tr>
<tr>
<td>InspectionResultModifyAfter</td>
<td>The after event for when a user modifies an inspection result.</td>
<td>✓</td>
</tr>
</tbody>
</table>
Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>InspectionResultModifyBefore</td>
<td>The after event for when a user modifies an inspection result.</td>
<td>√</td>
</tr>
<tr>
<td>InspectionResultSubmitAfter</td>
<td>The after event for when a user submits an inspection result.</td>
<td>√</td>
</tr>
<tr>
<td>InspectionResultSubmitBefore</td>
<td>The before event for when a user submits an inspection result.</td>
<td>√</td>
</tr>
<tr>
<td>InspectionScheduleAfter</td>
<td>Accela Citizen Access - The after event for when a user schedules one or more inspections.</td>
<td>√</td>
</tr>
<tr>
<td>InspectionScheduleBefore</td>
<td>Accela Citizen Access - The before event for when a user schedules one or more inspections.</td>
<td>√</td>
</tr>
<tr>
<td>InvoiceFeeAfter</td>
<td>The after event for when a user invoices a fee (manually or automatically)</td>
<td>√</td>
</tr>
<tr>
<td>LicenseProfessionalRemoveAfter</td>
<td>The after event for when a user removes a licensed professional.</td>
<td></td>
</tr>
<tr>
<td>LicenseProfessionalRemoveBefore</td>
<td>The before event for when a user removes a licensed professional.</td>
<td></td>
</tr>
<tr>
<td>LicProfAddAfter</td>
<td>The after event for when a user adds a licensed professional.</td>
<td></td>
</tr>
<tr>
<td>LicProfAddBefore</td>
<td>The before event for when a user adds a licensed professional.</td>
<td></td>
</tr>
<tr>
<td>LicProfLookUpSubmitAfter</td>
<td>The after event for when a user adds a reference license to a record.</td>
<td>√</td>
</tr>
<tr>
<td>LicProfLookUpSubmitBefore</td>
<td>The before event for when a user adds a reference license to a record.</td>
<td>√</td>
</tr>
<tr>
<td>LicProfUpdateAfter</td>
<td>The after event for when a user updates a licensed professional.</td>
<td>√</td>
</tr>
<tr>
<td>LicProfUpdateBefore</td>
<td>The before event for when a user updates a licensed professional.</td>
<td>√</td>
</tr>
<tr>
<td>MeetingAddAfter</td>
<td>The after event for when a user creates a meeting. Replaces EventAddAfter.</td>
<td></td>
</tr>
<tr>
<td>MeetingAddBefore</td>
<td>The before event for when a user creates a meeting. Replaces EventAddBefore.</td>
<td></td>
</tr>
<tr>
<td>MeetingCancelAfter</td>
<td>The after event for when a user cancels a meeting.</td>
<td></td>
</tr>
<tr>
<td>MeetingCancelBefore</td>
<td>The before event for when a user cancels a meeting.</td>
<td></td>
</tr>
<tr>
<td>MeetingRemoveAfter</td>
<td>The after event for when a user removes a meeting. Replaces EventRemoveAfter.</td>
<td></td>
</tr>
<tr>
<td>MeetingRemoveBefore</td>
<td>The before event for when a user removes a meeting. Replaces EventRemoveBefore.</td>
<td></td>
</tr>
<tr>
<td>MeetingRescheduleAfter</td>
<td>The after event for when a user reschedules a meeting. Replaces EventRescheduleAfter.</td>
<td></td>
</tr>
<tr>
<td>MeetingRescheduleBefore</td>
<td>The before event for when a user reschedules a meeting. Replaces EventRescheduleBefore.</td>
<td></td>
</tr>
<tr>
<td>Event Name</td>
<td>Description</td>
<td>Master Script?</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>MeetingScheduleAfter</td>
<td>The after event for when a user schedules a meeting.</td>
<td></td>
</tr>
<tr>
<td>MeetingScheduleBefore</td>
<td>The before event for when a user schedules a meeting.</td>
<td></td>
</tr>
<tr>
<td>MeetingUpdateAfter</td>
<td>The after event for when a user updates calendar event information.</td>
<td></td>
</tr>
<tr>
<td>MeetingUpdateBefore</td>
<td>The before event for when a user updates a meeting. Replaces EventUpdateBefore.</td>
<td></td>
</tr>
<tr>
<td>MoveAgendaAfter</td>
<td>The after event for when a user moves an agenda to another meeting.</td>
<td></td>
</tr>
<tr>
<td>MoveAgendaBefore</td>
<td>The before event for when a user moves an agenda to another meeting.</td>
<td></td>
</tr>
<tr>
<td>OnlinePaymentPost</td>
<td>Accela Citizen Access - Occurs after a user posts an online payment.</td>
<td></td>
</tr>
<tr>
<td>OnlinePaymentRegister</td>
<td>Accela Citizen Access - Occurs after a user submits a payment.</td>
<td></td>
</tr>
<tr>
<td>OnLoginEventAfter4ACA</td>
<td>Accela Citizen Access - Occurs after the login validation.</td>
<td></td>
</tr>
<tr>
<td>OnLoginEventBefore4ACA</td>
<td>Accela Citizen Access - Occurs before the login validation.</td>
<td></td>
</tr>
<tr>
<td>OwnerLookUpAfter</td>
<td>The after event for when a user creates a reference owner after looking up the owner from reference.</td>
<td></td>
</tr>
<tr>
<td>OwnerLookUpBefore</td>
<td>The before event for when a user creates a reference owner after looking up the owner from reference.</td>
<td></td>
</tr>
<tr>
<td>OwnerRemoveAfter</td>
<td>The after event for when a daily user removes an owner.</td>
<td></td>
</tr>
<tr>
<td>OwnerRemoveBefore</td>
<td>The before event for when a daily user removes an owner.</td>
<td></td>
</tr>
<tr>
<td>OwnerSelectOnSpearFormAfter</td>
<td>The after event for when a user attaches selected addresses on the reference addresses look up result list portlet.</td>
<td></td>
</tr>
<tr>
<td>OwnerSelectOnSpearFormBefore</td>
<td>The before event for when a user attaches selected addresses on the reference addresses look up result list portlet.</td>
<td></td>
</tr>
<tr>
<td>ParcelAddAfter</td>
<td>The after event for when a user creates a parcel.</td>
<td>✓</td>
</tr>
<tr>
<td>ParcelAddBefore</td>
<td>The before event for when a user creates a parcel.</td>
<td>✓</td>
</tr>
<tr>
<td>ParcelConditionAddAfter</td>
<td>The after event for when a user adds a condition to a parcel.</td>
<td></td>
</tr>
<tr>
<td>ParcelLookUpBefore</td>
<td>The before event for when a user creates a reference parcel after looking up the parcel from reference.</td>
<td></td>
</tr>
<tr>
<td>ParcelMergeAfter</td>
<td>The after event for when a user merges parcels.</td>
<td></td>
</tr>
<tr>
<td>ParcelMergeBefore</td>
<td>The before event for when a user merges parcels.</td>
<td></td>
</tr>
<tr>
<td>ParcelRemoveAfter</td>
<td>The after event for when a daily user removes a parcel.</td>
<td></td>
</tr>
<tr>
<td>ParcelRemoveBefore</td>
<td>The before event for when a daily user removes a parcel.</td>
<td></td>
</tr>
<tr>
<td>Event Name</td>
<td>Description</td>
<td>Master Script?</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ParcelSelectOnSpearFormAfter</td>
<td>The after event for when a user attaches selected parcels on the reference parcels look up result list portlet.</td>
<td></td>
</tr>
<tr>
<td>ParcelSelectOnSpearFormBefore</td>
<td>The before event for when a user attaches selected parcels on the reference parcels look up result list portlet.</td>
<td></td>
</tr>
<tr>
<td>ParcelSetDetailUserExecuteAfter</td>
<td>The after event for the parcel set execute script.</td>
<td></td>
</tr>
<tr>
<td>ParcelSplitAfter</td>
<td>The after event for when a user splits parcels.</td>
<td></td>
</tr>
<tr>
<td>ParcelSplitBefore</td>
<td>The before event for when a user splits parcels.</td>
<td></td>
</tr>
<tr>
<td>ParcelUpdateAfter</td>
<td>The after event for when a user updates a parcel.</td>
<td>✅</td>
</tr>
<tr>
<td>ParcelUpdateBefore</td>
<td>The before event for when a user updates a parcel.</td>
<td></td>
</tr>
<tr>
<td>PartTransactionSubmitAfter</td>
<td>The after event for when a user creates a part transaction.</td>
<td></td>
</tr>
<tr>
<td>PartTransactionSubmitBefore</td>
<td>The before event for when a user creates a part transaction.</td>
<td></td>
</tr>
<tr>
<td>PartTransactionUpdateAfter</td>
<td>The after event for when a user updates a part transaction.</td>
<td></td>
</tr>
<tr>
<td>PartTransactionUpdateBefore</td>
<td>The before event for when a user updates a part transaction.</td>
<td></td>
</tr>
<tr>
<td>PaymentApplyAfter</td>
<td>The after event for when a user clicks the Submit button on the payment apply page.</td>
<td>✅</td>
</tr>
<tr>
<td>PaymentApplyBefore</td>
<td>The before event for when a user clicks the Submit button on the payment apply page.</td>
<td>✅</td>
</tr>
<tr>
<td>PaymentProcessingAfter</td>
<td>The after event for when a user indicates “do pay” in the payment processing portlet.</td>
<td>✅</td>
</tr>
<tr>
<td>PaymentProcessingBefore</td>
<td>The before event for when a user indicates “do pay” in the payment processing portlet.</td>
<td>✅</td>
</tr>
<tr>
<td>PaymentReceiveAfter</td>
<td>Accela Citizen Access - The after event for when Accela Citizen Access records payment allocation.</td>
<td>✅</td>
</tr>
<tr>
<td>PaymentReceiveBefore</td>
<td>Accela Citizen Access - The before event for when Accela Citizen Access records payment allocation.</td>
<td>✅</td>
</tr>
<tr>
<td>PaymentRefundAfter</td>
<td>The after event for a payment processing/POS/record payment refund.</td>
<td></td>
</tr>
<tr>
<td>PaymentRefundBefore</td>
<td>The before event for a payment processing/POS/record payment refund.</td>
<td></td>
</tr>
<tr>
<td>PaymentRefundSubmitBefore</td>
<td>The before event for when a user submit the request to refund a payment.</td>
<td></td>
</tr>
<tr>
<td>PermitIssueAfter</td>
<td>The after event for when a user creates a permit printout.</td>
<td></td>
</tr>
<tr>
<td>PermitIssueBefore</td>
<td>The before event for when a user creates a permit printout.</td>
<td></td>
</tr>
<tr>
<td>ProctorAssignedAfter</td>
<td>The after even when proctors receive an assignment to one or more examination sessions</td>
<td></td>
</tr>
<tr>
<td>ProctorAssignedBefore</td>
<td>The before event for when a user assigns multiple proctors.</td>
<td></td>
</tr>
<tr>
<td>ProctorUnassignedAfter</td>
<td>The after event when a proctor is unassigned (deleted) from an examination session.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProctorUnassignedBefore</td>
<td>The before event for when a user removes a proctor from an examination roster.</td>
<td></td>
</tr>
<tr>
<td>ProfessionalSetDetailUserExecuteAfter</td>
<td>The after event for the professional set execute script.</td>
<td></td>
</tr>
<tr>
<td>ProximityAlertBefore</td>
<td>The before event for when a user issues a workflow proximity alert.</td>
<td></td>
</tr>
<tr>
<td>PublicUserEditAfter</td>
<td>The after event for when a user updates public user information.</td>
<td></td>
</tr>
<tr>
<td>PublicUserEditBefore</td>
<td>The before event for when a user updates public user information.</td>
<td></td>
</tr>
<tr>
<td>PublishCommentsAfter</td>
<td>The after event for when a user publishes comments.</td>
<td></td>
</tr>
<tr>
<td>PublishCommentsBefore</td>
<td>The before event for when a user publishes comments.</td>
<td></td>
</tr>
<tr>
<td>RefContactEditAfter</td>
<td>The after event for when a user updates a contact in reference side.</td>
<td></td>
</tr>
<tr>
<td>RefContactEditBefore</td>
<td>The before event for when a user updates a contact in reference side.</td>
<td></td>
</tr>
<tr>
<td>RefContactNewAfter</td>
<td>The after event for when a user creates a contact in reference side.</td>
<td></td>
</tr>
<tr>
<td>RefContactNewBefore</td>
<td>The before event for when a user creates a contact in reference side.</td>
<td></td>
</tr>
<tr>
<td>RefExamUpdateAfter</td>
<td>The after event for when a user updates a reference examination.</td>
<td></td>
</tr>
<tr>
<td>RefExamUpdateBefore</td>
<td>The before event for when a user updates a reference examination.</td>
<td></td>
</tr>
<tr>
<td>RefLicProfAddAfter</td>
<td>The after event for when a user adds a reference licensed professional.</td>
<td></td>
</tr>
<tr>
<td>RefLicProfAddBefore</td>
<td>The before event for when a user adds a reference licensed professional.</td>
<td></td>
</tr>
<tr>
<td>RefLicProfUpdateAfter</td>
<td>The after event for when a user updates a reference licensed professional.</td>
<td></td>
</tr>
<tr>
<td>RefLicProfUpdateBefore</td>
<td>The before event for when a user updates a reference licensed professional.</td>
<td></td>
</tr>
<tr>
<td>RegistrationSubmitAfter</td>
<td>Accela Citizen Access - The after event for when a public user submits a registration or when a public user associates a licensed professional with his user account.</td>
<td></td>
</tr>
<tr>
<td>RegistrationSubmitBefore</td>
<td>Accela Citizen Access - The before event for when a user submits a registration.</td>
<td></td>
</tr>
<tr>
<td>RejectContactAssociationforPublicAfter</td>
<td>The after event for when a user rejects a contact association for a public user.</td>
<td></td>
</tr>
<tr>
<td>RelatedCapUpdateAfter</td>
<td>The after event for when a user updates related records.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelatedCapUpdateBefore</td>
<td>The before event for when a user updates related records.</td>
<td></td>
</tr>
<tr>
<td>RemoveAgendaAfter</td>
<td>The after event for when a user removes an agenda from a meeting.</td>
<td></td>
</tr>
<tr>
<td>RemoveAgendaBefore</td>
<td>The before event for when a user removes an agenda from a meeting.</td>
<td></td>
</tr>
<tr>
<td>RenewalInfoUpdateAfter</td>
<td>The after event for when a user creates a permit printout.</td>
<td>√</td>
</tr>
<tr>
<td>ReportRunAfterInACA</td>
<td>The after event for when a report is run in ACA.</td>
<td></td>
</tr>
<tr>
<td>ReportRunBeforeInACA</td>
<td>The before event for when a report is run in ACA.</td>
<td></td>
</tr>
<tr>
<td>ReportServiceRunAfter</td>
<td>The after event for when a user runs a report service.</td>
<td></td>
</tr>
<tr>
<td>ReportServiceRunBefore</td>
<td>The before event for when a user runs a report service.</td>
<td></td>
</tr>
<tr>
<td>SaveAndResumeAfter4ACA</td>
<td>Accela Citizen Access - The after event for when a user saves and resume.</td>
<td></td>
</tr>
<tr>
<td>SearchMultiServiceAfter</td>
<td>The after event for when a user searches a service.</td>
<td></td>
</tr>
<tr>
<td>SelectLicenseValidation4ACA</td>
<td>Accela Citizen Access - Occurs when the user selects a license by the license drop-down list.</td>
<td></td>
</tr>
<tr>
<td>ShoppingCartCheckOutBefore</td>
<td>The before event for when a user checks out their shopping cart in ACA.</td>
<td></td>
</tr>
<tr>
<td>StrucEstLookUpAfter</td>
<td>The after event for when a user creates a reference structure or establishment, after looking up the owner from reference.</td>
<td></td>
</tr>
<tr>
<td>StrucEstLookUpBefore</td>
<td>The before event for when a user creates a reference structure or establishment after looking up the owner from reference.</td>
<td></td>
</tr>
<tr>
<td>StrucEstRemoveAfter</td>
<td>The after event for when a daily user removes a structure or establishment.</td>
<td></td>
</tr>
<tr>
<td>StrucEstRemoveBefore</td>
<td>The before event for when a daily user removes a structure or establishment.</td>
<td></td>
</tr>
<tr>
<td>StructureAddAfter</td>
<td>The after event for when a user adds a structure.</td>
<td></td>
</tr>
<tr>
<td>StructureAddBefore</td>
<td>The before event for when a user adds a structure.</td>
<td></td>
</tr>
<tr>
<td>StructureUpdateAfter</td>
<td>The after event for when a user updates a structure.</td>
<td></td>
</tr>
<tr>
<td>StructureUpdateBefore</td>
<td>The before event for when a user updates a structure.</td>
<td></td>
</tr>
<tr>
<td>taskEditActionFormBefore</td>
<td>The before event for when a user updates a workflow task.</td>
<td></td>
</tr>
<tr>
<td>TimeAccountingAddAfter</td>
<td>Executes when time accounting entries are about to be added.</td>
<td>√</td>
</tr>
<tr>
<td>TimeAccountingAddBefore</td>
<td>Executes when time accounting entries are about to be added.</td>
<td></td>
</tr>
<tr>
<td>TimeAccountingDeleteAfter</td>
<td>Executes when time accounting entries are about to be removed.</td>
<td>√</td>
</tr>
</tbody>
</table>
### Table 2: Event and Master Script List

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
<th>Master Script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeAccountingDeleteBefore</td>
<td>EXECUTES WHEN TIME ACCOUNTING ENTRIES ARE ABOUT TO BE REMOVED.</td>
<td></td>
</tr>
<tr>
<td>TimeAccountingUpdateAfter</td>
<td>EXECUTES WHEN TIME ACCOUNTING ENTRIES ARE ABOUT TO BE UPDATED.</td>
<td>✓</td>
</tr>
<tr>
<td>TimeAccountingUpdateBefore</td>
<td>EXECUTES WHEN TIME ACCOUNTING ENTRIES ARE ABOUT TO BE UPDATED.</td>
<td></td>
</tr>
<tr>
<td>UpdateContactTypeBefore</td>
<td>EXECUTES WHEN A USER ATTEMPTS TO CHANGE THE CONTACT TYPE IN A RECORD CONTACT LIST (WHICH MAY BE ON THE APPLICATION INTAKE FORM, OR IN THE CONTACT TAB OF THE RECORD PORTLET OR THE RECORDS SET PORTLET). YOU CAN ADD SCRIPTS TO THE EVENT TO VALIDATE THE CONTACT TYPE CHANGE.</td>
<td></td>
</tr>
<tr>
<td>UpdateContactTypeAfter</td>
<td>EXECUTES AFTER THE USER UPDATES THE CONTACT TYPE SUCCESSFULLY IN A RECORD CONTACT LIST.</td>
<td></td>
</tr>
<tr>
<td>V360InspectionResultSubmitAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER RESULTS AN INSPECTION.</td>
<td>✓</td>
</tr>
<tr>
<td>V360InspectionResultSubmitBefore</td>
<td>THE BEFORE EVENT FOR WHEN A USER RESULTS AN INSPECTION</td>
<td>✓</td>
</tr>
<tr>
<td>V360ParcelAddAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER ADDS A PARCEL.</td>
<td>✓</td>
</tr>
<tr>
<td>VoidFeeAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER VOIDS A (MANUALLY OR AUTOMATICALLY)</td>
<td></td>
</tr>
<tr>
<td>VoidFeeBefore</td>
<td>THE BEFORE EVENT FOR WHEN A USER VOIDS A FEE (MANUALLY OR AUTOMATICALLY)</td>
<td></td>
</tr>
<tr>
<td>VoidPaymentAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER VOIDS A PAYMENT.</td>
<td>✓</td>
</tr>
<tr>
<td>VoidPaymentBefore</td>
<td>THE BEFORE EVENT FOR WHEN A USER VOIDS A PAYMENT.</td>
<td>✓</td>
</tr>
<tr>
<td>WorkflowAdhocTaskAddAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER ADDS A WORKFLOW TASK.</td>
<td></td>
</tr>
<tr>
<td>WorkflowAdhocTaskAddBefore</td>
<td>THE BEFORE EVENT FOR WHEN A USER ADDS A WORKFLOW TASK.</td>
<td></td>
</tr>
<tr>
<td>WorkflowAdhocTaskUpdateAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER UPDATES AN ADHOC WORKFLOW TASK.</td>
<td></td>
</tr>
<tr>
<td>WorkflowAdhocTaskUpdateBefore</td>
<td>THE BEFORE EVENT FOR WHEN A USER UPDATES AN ADHOC WORKFLOW TASK.</td>
<td></td>
</tr>
<tr>
<td>WorkflowTaskUpdateAfter</td>
<td>ACCELA CITIZEN ACCESS - THE AFTER EVENT FOR WHEN A USER UPDATES A WORKFLOW TASK.</td>
<td>✓</td>
</tr>
<tr>
<td>WorkflowTaskUpdateBefore</td>
<td>ACCELA CITIZEN ACCESS - THE BEFORE EVENT FOR WHEN A USER UPDATES A WORKFLOW TASK.</td>
<td>✓</td>
</tr>
<tr>
<td>XRefContactAddressEditAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER EDITS THE CROSS REFERENCE CONTACT ADDRESS.</td>
<td></td>
</tr>
<tr>
<td>XRefContactAddressEditBefore</td>
<td>THE BEFORE EVENT FOR WHEN A USER EDITS THE CROSS REFERENCE CONTACT ADDRESS.</td>
<td></td>
</tr>
<tr>
<td>XRefContactAddressLookUpAfter</td>
<td>THE AFTER EVENT FOR WHEN A USER LOOKS UP A CROSS REFERENCE CONTACT ADDRESS.</td>
<td></td>
</tr>
</tbody>
</table>
Working with Events

Topics:
- Searching for an Active Event
- Viewing the Full List of Accela Automation Events
- Enabling an Event
- Disabling an Event

Searching for an Active Event

Before you can view or edit an active event, you must first locate it. You must also search for an event to associate a script with it. You can search for any enabled event.

To search for an event

Accela Automation displays the Event Search window.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRefContactAddressLookUpBefore</td>
<td>The before event for when a user looks up a cross reference contact address.</td>
</tr>
<tr>
<td>XRefContactAddressNewAfter</td>
<td>The after event for when a user creates a cross reference contact address.</td>
</tr>
<tr>
<td>XRefContactAddressNewBefore</td>
<td>The before event for when a user creates a cross reference contact address.</td>
</tr>
<tr>
<td>XRefContactAddressRemoveAfter</td>
<td>The after event for when a user removes a cross reference contact address.</td>
</tr>
<tr>
<td>XRefContactAddressRemoveBefore</td>
<td>The before event for when a user removes a cross reference contact address.</td>
</tr>
</tbody>
</table>
2. Complete these fields:

- **Event Name**: Enter the name of the event that you want to find.
- **Script Title**: Enter the name or title of the script associated with the event that you want to find.

To see a list of all the enabled events, leave the fields blank.

3. Click **Submit**.

**Accela Automation displays the Event List window.**

<table>
<thead>
<tr>
<th>Events - Event List</th>
<th>Associated Script</th>
<th>Action</th>
<th>Last Modified Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApplicationDetailsUpdateBefore</td>
<td>ApplicationDetailsUpdateBefore.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ApplicationDeleteAfter</td>
<td>ApplicationDeleteAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ApplicationSelectBefore</td>
<td>ApplicationSelectBefore.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ApplicationSpecificInfoUpdateAfter</td>
<td>ApplicationSpecificInfoUpdateAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ApplicationSpecificInfoUpdateBefore</td>
<td>ApplicationSpecificInfoUpdateBefore.xmlscript</td>
<td>10/16/2005 by GORDON</td>
<td></td>
</tr>
<tr>
<td>ApplicationStatusUpdateAfter</td>
<td>ApplicationStatusUpdateAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ApplicationSubmitAfter</td>
<td>ApplicationSubmitAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>CMTConditionAddAfter</td>
<td>CMTConditionAddAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ContactAddAfter</td>
<td>ContactAddAfter.xmlscript</td>
<td>10/28/2005 by GORDON</td>
<td></td>
</tr>
<tr>
<td>ContactAddBefore</td>
<td>ContactAddBefore.xmlscript</td>
<td>10/28/2005 by GORDON</td>
<td></td>
</tr>
<tr>
<td>ContactUpdateAfter</td>
<td>ContactUpdateAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ContactDeleteAfter</td>
<td>ContactDeleteAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ContactRemoveBefore</td>
<td>ContactRemoveBefore.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
<tr>
<td>ContactRemoveAfter</td>
<td>ContactRemoveAfter.xmlscript</td>
<td>11/3/2005 by ADMIN</td>
<td></td>
</tr>
</tbody>
</table>

4. Click the red dot that appears to the left of the event that you want.

**Accela Automation displays the Event Detail window.**
Viewing the Full List of Accela Automation Events

You can view the entire list of available events, including active events and disabled events.

To view the full list of Accela Automation events


   *Accela Automation displays the Event Search window.*

2. Leave the fields blank and click **Submit**.

   *Accela Automation displays the Event List window.*
3. Click **Add**.

**Accela Automation displays the Add New Event window.**

4. Click the drop-down menu to expand the list. This list contains all of the possible events.
Enabling an Event

Accela provides a list of standard events for your agency. Before you can use an event, you must enable it for your agency. Depending on the events that you choose to enable and the script that you associate with each event, you can customize Accela Automation to automatically perform various transactions.

To enable an event

1. **Choose AA Classic > Admin Tools > Events > Events.**
   
   *Accela Automation displays the Event Search window.*

   ![Event Search Window](image)

2. **Click Submit** to see a list of events enabled for your agency.
   
   *Accela Automation displays the Event List window.*

   ![Event List Window](image)

3. **Click Add.**
   
   *Accela Automation displays the Add New Event window.*

   ![Add New Event Window](image)
4. Use the drop-down list to choose from the available events.

5. Click Add.

6. Associate a script with the event. For details, see Associating Events with Scripts on page 55.

Disabling an Event

Accela provides a list of standard events for your agency. You can disable any currently enabled event. When you disable an event, Event Manager no longer tracks the event or executes any script associated with it.

To disable an event for your agency


   Accela Automation displays the Event Search window.

2. Search for the event that you want or click the Submit button to see a list of events enabled for your agency.

   Accela Automation displays the Event List.

3. Click the red dot that appears next to the event you want to disable.

   Accela Automation displays the Event Detail window.
4. Click Delete.

5. Click OK to confirm your choice.

*Accela Automation disables the event.*

**Triggering Events**

This section provides details on before and after event triggers.

**Topics**

- Triggering Meeting Agenda Events
- Triggering Meeting Schedule Events

**Triggering Meeting Agenda Events**

Accela Automation provides six events related to meeting agendas (records).

- AddAgendaBefore
- AddAgendaAfter
- MoveAgendaBefore
- MoveAgendaAfter
- RemoveAgendaBefore
- RemoveAgendaAfter

The same user action triggers the before and after version of an event.

- Click **Select** to trigger the AddAgendaBefore and AddAgendaAfter events.
- Click **Submit** to trigger the MoveAgendaBefore and MoveAgendaAfter events.
- Click **Remove** to trigger the RemoveAgendaBefore and RemoveAgendaAfter events.

To trigger a before or after agenda-related event
1. Use one of the five Accela Automation portlets to access meeting details
   - Select a meeting calendar as an administrator (Admin > Setup > Calendars > Calendar > Calendar by Type > Meeting > select a meeting calendar > select a meeting).
   - Select a meeting calendar as a daily user (Calendars > Calendar by Type > Meeting > select a meeting calendar > select a meeting).
   - Select a meeting calendar from the MyTasks portlet (My Tasks > Meetings > select a meeting).
   - Select a meeting calendar from the Task Management portlet (Task Management > select a record of the meeting task type).
   - Select a meeting calendar from the Record portlet (Record > Calendar tab > select a meeting in calendar view).

2. Click the Agenda & Vote tab.

3. Trigger a before or after AddAgenda event.
   a. Click Add.
   b. Enter search criteria for the record(s) to add and click Submit.
   c. Select the record(s) you want to add and click Select () to trigger the event.

4. Trigger a before or after MoveAgenda event.
   a. Select one or more records to move.
   b. Click Move.
   c. Enter search criteria for the meeting to which you want to move the records and click Submit.
   d. Select the meeting to which you want to move the agenda and click Submit to trigger the event.

5. Trigger a before or after add remove event.
   a. Select one or more records to remove and click Remove to trigger the event.

**Triggering Meeting Schedule Events**

Accela Automation provides four events related to meeting schedules.

- MeetingScheduleBefore
- MeetingScheduleAfter
- MeetingCancelBefore
- MeetingCancelAfter

The same user action triggers the before and after version of an event.

- Click Submit to trigger the MeetingScheduleBefore and MeetingScheduleAfter events.
- Click Cancel to trigger the MeetingCancelBefore and MeetingCancelAfter events.
To trigger a before or after schedule-related event
1. Access the Records portlet.
2. Select a record for which you want to trigger the schedule-related event.
3. Click the Meetings tab.
4. Select the meeting you want to schedule or cancel.
   - Trigger a before or after MeetingCancel event by clicking the Manage Meeting > Cancel Meeting submenu.
   - Trigger a before or after MeetingSchedule event.
     a. Click the Manage Meeting > Schedule Meeting submenu.
     b. Enter search criteria for the meeting to which you want to schedule the record and click Submit.
     c. Select the meeting for which you want to schedule the record and click Submit to trigger the event.

Working with Scripts

Topics:
- Adding a Script
- Searching for a Script
- Editing a Script
- Deleting a Script

Adding a Script

Scripts allow you to make specific changes to your database based on the event that occurs. For each pre-defined and enabled event, you can determine the script that you want to run for that event. In addition to associating standard scripts with standard events, you can write custom scripts that you want to assign to certain events.

To add a new script
   Accela Automation displays the Scripts search window.
2. Click Submit to see a list of scripts enabled for your agency.
3. Click Add.
   Accela Automation displays the Add a new script page.
4. Complete the necessary fields as described in Table 3: Script Details on page 53.

5. Click Add.

Table 3: Script Details

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script Code</td>
<td>Enter the code or abbreviation that identifies the script.</td>
</tr>
<tr>
<td>Script Title</td>
<td>Enter the name or title of the script.</td>
</tr>
<tr>
<td>Script Initializer</td>
<td>If the script requires an initializer, enter it here. The initializer may be necessary to start certain scripts and contain certain input parameters.</td>
</tr>
<tr>
<td>Script Content/Text</td>
<td>Enter the script text here. You can also copy and paste the script into this text area.</td>
</tr>
</tbody>
</table>

**Searching for a Script**

You can search for a script to view or edit it.

**To search for a script**


2. Complete the necessary fields as described in Table 3: Script Details on page 53.

3. Click Submit.

4. Click the red dot that appears to the left of the script that you want.
Editing a Script

For each pre-defined and enabled event, you can determine the script that you want to run for that event. Accela provides several standard scripts. In addition to writing original scripts, you can modify standard scripts. You can make changes to any existing script that is currently on your system.

To edit a script
1. Choose Administrator Tools > Events > Script.
2. Search for the script that you want.
3. Complete the necessary fields as described in Table 3: Script Details on page 53.
4. If you want to test the script, click the Script Test button.
5. Click Save.

Deleting a Script

You can delete any script.

To delete a script
1. Choose Administrator Tools > Events > Script.
2. Search for the script that you want.
3. Click Delete.
4. Click OK to confirm your choice.
Associating Events with Scripts

After you enable an event and add a script to your system, you can associate a script with an event. Associating a script with an event allows Accela Automation to execute or run the script when the event occurs.

To associate an event with a script, the script must already exist. For information on adding a script to your system, see Working with Scripts on page 52.

Example Use Case

Someone applies for a permit and you want Accela Automation to check the license expiration date to confirm that the license has not expired. You select an event such as ApplicationSubmitBefore and then associate a script that compares license expiration dates with the current date.

To associate an event with a script

1. Choose Administrator Tools > Events > Script.

2. Search for the event that you want. For details, see Searching for an Active Event on page 44.

3. Use the Script Name drop-down list to choose the script that you want to associate with this event.

4. Click Save.
Accela Automation provides some Out-Of-The-Box master scripts. Accela Automation defines a 1-1 relationship between the master script and the event which triggers master script execution. Accela Automation uses the same base name for the master script and the associated trigger event. The following lists these master scripts.

<table>
<thead>
<tr>
<th>Master Script</th>
<th>Master Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdditionalInfoUpdateAfter</td>
<td>AdditionalInfoUpdateBefore</td>
</tr>
<tr>
<td>ApplicationConditionAddAfter</td>
<td>ApplicationConditionDeleteBefore</td>
</tr>
<tr>
<td>ApplicationConditionUpdateAfter</td>
<td>ApplicationConditionUpdateBefore</td>
</tr>
<tr>
<td>ApplicationSpecificInfoUpdateAfter</td>
<td>ApplicationSpecificInfoUpdateBefore</td>
</tr>
<tr>
<td>ApplicationStatusUpdateAfter</td>
<td>ApplicationStatusUpdateBefore</td>
</tr>
<tr>
<td>ApplicationSubmitAfter</td>
<td>ApplicationSubmitBefore</td>
</tr>
<tr>
<td>CapSetProcessing</td>
<td>ContactAddAfter</td>
</tr>
<tr>
<td>ContactAddBefore</td>
<td>ContactEditAfter</td>
</tr>
<tr>
<td>ContactEditBefore</td>
<td>ContactRemoveAfter</td>
</tr>
<tr>
<td>ContactRemoveBefore</td>
<td>ConvertToRealCapAfter</td>
</tr>
<tr>
<td>DocumentUploadAfter</td>
<td>DocumentUploadBefore</td>
</tr>
<tr>
<td>FeeAssessAfter</td>
<td>FeeAssessBefore</td>
</tr>
<tr>
<td>FeeEstimateAfter4ACA</td>
<td>InspectionMultipleScheduleAfter</td>
</tr>
<tr>
<td>InspectionMultipleScheduleBefore</td>
<td>InspectionResultModifyAfter</td>
</tr>
<tr>
<td>InspectionResultModifyBefore</td>
<td>InspectionResultSubmitBefore</td>
</tr>
<tr>
<td>InspectionResultSubmitBefore</td>
<td>InspectionScheduleAfter</td>
</tr>
<tr>
<td>InspectionScheduleBefore</td>
<td>InvoiceFeeAfter</td>
</tr>
<tr>
<td>LicProfLookupSubmitAfter</td>
<td>LicProfLookupSubmitBefore</td>
</tr>
<tr>
<td>LicProfUpdateAfter</td>
<td>LicProfUpdateBefore</td>
</tr>
<tr>
<td>ParcelAddAfter</td>
<td>ParcelAddBefore</td>
</tr>
<tr>
<td>ParcelUpdateAfter</td>
<td>PaymentApplyAfter</td>
</tr>
<tr>
<td>PaymentApplyBefore</td>
<td>PaymentProcessingAfter</td>
</tr>
</tbody>
</table>
In addition to event-specific master scripts, Accela Automation provides the following additional master script files:

- **UniversalMasterScript**: Provides a template for creating additional event-specific master scripts.
- **ScriptTester**: Enables you to test script controls without triggering an event from the user interface.
- **INCLUDES_ACCELA_FUNCTIONS**: Included by each master script during runtime. Contains all standard master script functions provided by Accela. A copy of each of these standard functions, in previous framework versions, had to be present in each of the individual master script files. Do not modify this file outside of official Accela master script releases.
- **INCLUDES_ACCELA_FUNCTIONS_ASB**: Included by each master script during runtime. Similar to INCLUDES_ACCELA_FUNCTIONS but contains Accela provided functions specific to the ApplicationSubmitBefore event.
- **INCLUDES_ACCELA_GLOBALS**: Included by each master script during runtime. Contains global flags that are responsible for the setup the EMSE master script environment. Each master script file, from previous framework versions, set these flags individually in the master script file. Some examples of these global settings are enableVariableBranching, showDebug, showMessage, and useAppSpecificGroupName.
- **INCLUDES_CUSTOM**: Contains customizations made to the master script framework. Every executed master script evaluates the script code in this file. Segregation of customizations in this file enables you to upgrade and maintain the EMSE master script framework without an impact to your customizations.

**Topics:**
- Understanding the EMSE Execution Path
- Creating a New Script
- Configuring the Universal Script
- Configuring Global Variables
- Adding Custom Functions
Viewing Master Scripts

Accela Automation provides the Master Scripts and Custom Script administration tools as part of the Event administration tools. These tools enable you to view available master scripts, and to view or edit the custom script.

To view a master script

   
   *Accela Automation displays the Master Scripts search window.*

2. In the **Master Script Version** drop-down list, select the Master Script Version you want to view.

   *Note:* You can upgrade the master script version when you upgrade Accela Automation. Accela Automation makes all versions of the master scripts available at the same time. Administrators can set the Standard Choice MASTER_SCRIPT_DEFAULT_VERSION to continuously apply a specific version of master scripts, regardless of the master script upgrades.

3. Click the **Submit** button to see the list of master scripts provided in the version.
   
   *Accela Automation displays the master script list.*

   For the complete master script list, see Table 2: Event and Master Script List on page 31.

4. Click the red dot that appears next to the master script you want to view.
   
   *Accela Automation displays the master script detail.*

To view and edit a custom script

   
   *Accela Automation displays the custom script detail.*

   The script name of the custom script is INCLUDES_CUSTOM.

2. Edit the script code of the custom script in the **Master Script Text** field.
   
   For more information on editing custom script, see Adding Custom Functions on page 63.

Understanding the EMSE Execution Path

*Figure 12: EMSE Execution Path* shows that the master script execution process leverages four script include files.
Event Start

Event Global Parameters

Is ASB?
No

INCLUDES ACCELA_FUNCTIONS

INCLUDES ACCELA_FUNCTIONS_ASB

INCLUDES ACCELA_GLOBALS

INCLUDES CUSTOM

Event Specific Parameters

Pre-Execute
Standard Choices

Master Script Control String
Standard Choices

Event End

Event level global that control pre-execute and event entry points

ApplicationSubmitBefore utilizes a special set of functions due to not having data available in the database yet.

Master Script Functions are now loaded and available for use once the include is completed.

Master Script globals are now loaded and logged via debugging. Used to setup defaults for debugging, version checks, variable branching, and other master script flags.

Custom Include file is executed and included for use in scripting

Event Specific Parameters are loaded and available for use

Script controls are invoked for the preExecute Standard choices associated to either before or after events

Script Controls are Invoked for event level Standard choice
Creating a New Script

Accela Automation provides master scripts for many of the events. Use the UniversalMasterScript as a template to create scripts for the remaining events.

Accela Automation requires a separate script per event to:

- Identify the entry point Standard Choice that contains the script controls for that event (desired actions when triggered)
- To create and populate event-specific variables needed for each specific event (eg. wfTask, inspType)

To create a new script

1. Copy the contents of the UniversalMasterScript file and paste the contents into your script development environment (text editor or IDE).
2. Save the new script file with the same base name as the event to which you plan to associate the new script.
3. Create a new standard choice with the same name as the event. This standard choice becomes the entry point standard choice for this event (Chapter 4: Script Controls on page 65).
4. Modify the new script file as required (Configuring the Universal Script on page 60).
5. Install the script file (Chapter 2: Event and Script Setup on page 30).

Configuring the Universal Script

When you create a new script file, you copy the contents of the UniversalMasterScript file into your new script file (Creating a New Script on page 60). You then need to modify this copied content to configure it for your particular application.

To configure the universal script

1. Locate the START Configurable Parameters section of the master script.
2. Set the value of the controlString variable to the name of the Standard Choice. The Standard Choice name must match the name of the event for which the Standard Choice contains the script controls.
   
   \[
   \text{var controlString} = "<\text{Standard Choice}>"
   \]

3. Set the value of the preExecute variable to indicate whether to trigger the script before or after the event.
   
   \[
   \text{var preExecute} = "<\text{before or after}>"
   \]

   where:

   <before or after> is PreExecuteForBeforeEvents for before events and PreExecuteForAfterEvents for after events.
4. Set the `documentOnly` variable to specify whether or not to display the hierarchy of standard choice steps.

   ```javascript
   var documentOnly = false
   ```

5. Configure the remaining sections as required.

   - The following section of the master script configures the internal version of the master script file and the global master scripts to include during runtime.
     ```javascript
     var SCRIPT_VERSION = 2.0
     eval(getScriptText("INCLUDES_ACCELA_FUNCTIONS"));
     eval(getScriptText("INCLUDES_ACCELA_GLOBALS"));
     eval(getScriptText("INCLUDES_CUSTOM"));
     ```

   - This section includes the scripting to evaluate the value of the `documentOnly` variable configured in the previous section.
     ```javascript
     if (documentOnly) {
       doStandardChoiceActions(controlString,false,0);
       aa.env.setValue("ScriptReturnCode", "0");
       aa.env.setValue("ScriptReturnMessage", "Documentation Successful. No actions executed.");
       aa.abortScript();
     }
     ```

   - The `BEGIN Event Specific Variables` section loads the values for the variables of the associated event (Figure 13: AdditionalInfoUpdateAfter Variables).
Figure 13: AdditionalInfoUpdateAfter Variables

For example, Accela Automation uses the following variables for the AdditionalInfoUpdateAfter event.

```javascript
var aiBuildingCount = aa.env.getValue("AdditionalInfoBuildingCount");
var aiConstructionTypeCode = aa.env.getValue("AdditionalInfoConstructionTypeCode");
var aiHouseCount = aa.env.getValue("AdditionalInfoHouseCount");
var aiPublicOwnedFlag = aa.env.getValue("AdditionalInfoPublicOwnedFlag");
var aiValuation = aa.env.getValue("AdditionalInfoValuation");
```

This variable list corresponds to the default set of variables defined for the event.

**Note:** The INCLUDES_ACCELA_FUNCTIONS master scripts resolves the CurrentUserID, PermitId1, PermitId2, and PermitId3 global variables.

- After logging event specific variable, the master script executes the Main Loop by performing the actions prescribed by the applicable Standard Choice script controls.

```
doStandardChoiceActions(controlString,true,0);
```

Configuring Global Variables

Table 4: Configurable Global Parameters provides parameters you can configure in the INCLUDES_ACCELA_GLOBALS file.
Adding Custom Functions

Accela Automation master scripts provide a placeholder to include the INCLUDES_CUSTOM master script file.

```javascript
eval(getScriptText("INCLUDES_CUSTOM"));
```

If you need to create new functions, save your customizations in a file named INCLUDES_CUSTOM.

**Note:** To prevent the Accela Automation installer from overwriting existing customizations during a product upgrade, Accela Automation does not provide the INCLUDES_CUSTOM master script file as part of the Accela Automation installer.

If the INCLUDES_ACCELA_FUNCTIONS and INCLUDES_CUSTOM contain a function with the same name, the function in the INCLUDES_CUSTOM file overwrites the function in the INCLUDES_ACCELA_FUNCTIONS file.

Do not modify functions in the INCLUDES_ACCELA_FUNCTIONS file. If you want to modify a function from the INCLUDES_ACCELA_FUNCTIONS file, create a same named function with the different functionality in the INCLUDES_CUSTOM file.

As a best practice, use a commenting structure in your INCLUDES_CUSTOM file to keep it organized and easy to interpret. The following provides an example.

```plaintext
/**************** Modified Accela Standard Functions
(Start) **************/
```
//All modified Accela standard functions will live here
/******************** Modified Accela Standard Functions
(End) ********************/
/******************** Custom Building Functions (Start)
**************/
//All custom building functions will live here
/******************** Custom Building Functions (End)
*******************/
/******************** Custom Licensing Functions (Start)
**************/
//All custom licensing functions will live here
/******************** Custom Licensing Functions (End)
*******************/
/******************** Custom Planning Functions (Start)
**************/
//All custom planning functions will live here
/******************** Custom Planning Functions (End)
*******************/
/* Start a new section for each logical group */
Understanding Script Controls

Accela Automation uses Standard Choice script controls to instruct Accela Automation how to perform before and after event activities. Each script control provides parameters to master script functions (Appendix A: Master Script Function List on page 114) within a framework of conditional (if-then-else) expressions. A single Standard Choice can contain multiple script controls. The master script evaluates the script controls in the order that the Standard Choice numbering specifies.

Script controls use the caret (^) symbol to delimit the if clause (predicate) from the then clause (consequent) and the else clause (alternative) in a single conditional expression. Accela Automation interprets the first clause as the if clause, the second clause as the then clause, and the third (optional) clause as the else clause.

Each clause in a script control calls a master script function and provides parameter values required by that function. The variables associated with the scriptable event () determine the scope of possible variables that the script control provides to the master script function.

Enclose master script function parameters in parenthesis. Use a comma to delimit master script function parameters. Enclose string parameters in double straight quotes. Appendix D: JavaScript Primer on page 244 provides additional Javascript syntax elements you can use in script controls.

Example Use Case

Figure 14: Script Control Syntax shows a single script control. This script control says, "If the current record type is not a Building/Reroof type, then assess but do not invoice all of the fees
from the fee schedule called BLDCR05. The master script function that the script control calls, appMatch for example, provides a return value, in this case true or false, to determine whether to perform the function in the then clause.

Figure 14: Script Control Syntax

Example Use Case

Figure 15: Script Control Structure (if/then/else) says, “If Acres Disturbed is less than 5 then assess but do not invoice the SMALLACRE fee, else assess but do not invoice the BIGACRE fee.”

Figure 15: Script Control Structure (if/then/else)

Understanding Script Control Syntax

Topics:

- Understanding Case Sensitivity
- Understanding Variable and Function Names
- Understanding Curly Brackets
- Understanding Argument Types

Understanding Case Sensitivity

The master scripts and underlying JavaScript require case sensitivity for function calls or when referring to a variable. For example in Figure 15: Script Control Structure (if/then/else) you see the function addFee called in both the then and else action. If you write the same script control but call the function AddFee, the script returns an error that the function AddFee does not exist. The script considers addFee and AddFee two different function names.
Understanding Variable and Function Names

Variables and function names in the master scripts follow the camelCase practice. For example totalSquareFeet, taxiNumber, addFee(), etc. Always be aware of case sensitivity as it many times could be the culprit of causing script errors.

Understanding Curly Brackets

We already saw the usage of the caret (^) to form conditional statements, another master script specific syntax is the usage of curly brackets { }. When a user triggers an event, Accela Automation calls the associated master script. Before EMSE evaluates the first line of script controls, the master script does some pre-work to initialize and set the value of several global and event-specific variables that the script controls can reference. Some of this pre-work loads application information, task information, and parcel attributes into individual variables. The script control encloses each of these variables between two curly brackets (Figure 15: Script Control Structure (if/then/else)). For example, {Acres Disturbed} in the script control condition indicates an application specific Information field.

Understanding Argument Types

Always enclose strings in double quotes. For example:
- the criteria -- {Land Use} == "Farming"
- setting the value of a variable -- layerName = “Zoning”
- a function call that accepts string parameters addFee("AppFee","BLD_11","FINAL",1,"Y")

Do not enclose numeric fields in double quotes.

Script controls must be valid JavaScript. If a script control deviates from JavaScript syntax, outside of that which is unique to the master scripts, syntax errors occur.

Understanding Criteria (the If Clause)

Topics:
- Understanding Criteria with Multiple Conditional Statements

Criteria must always evaluate to either true or false. A criteria statement can contain logical operators, such as ==, >, >=, <, <=, or != to evaluate if a statement is true or false, and can also call functions that return true or false (Table 5: Criteria Examples with Single Operators).
Table 5: Criteria Examples with Single Operators

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>If you use a true as the if clause, the specified action always executes.</td>
</tr>
<tr>
<td>appMatch(&quot;Building/Commercial/<em>/</em>&quot;)</td>
<td>The <code>appMatch</code> function returns true or false depending on whether the current record type matches the record type in the function parameter. The asterisks (*) indicate a wildcard. In this example, any record types that start with Building/Commercial return a true.</td>
</tr>
<tr>
<td>!appMatch(&quot;Building/Commercial/<em>/</em>&quot;) appMatch(&quot;Building/Commercial/<em>/</em>&quot;) != true</td>
<td>These two examples mean the same thing, with different syntax. Both say, if the current record type is not under Building/Commercial do the action.</td>
</tr>
<tr>
<td><code>inspType</code> == &quot;Final Inspection&quot;</td>
<td>Use double equals (==) check whether a value equals another variable or a string. In the example, if the value for the <code>inspType</code> variable of the triggered event equals “Final Inspection” then execute the associated action.</td>
</tr>
<tr>
<td><code>{STRUCTURE DETAILS.Total Square Feet} &gt;= 2000</code></td>
<td>You can use criteria to test the value of an Application Specific Information. In the example, if the value of the ASI field name Total Square Feet within the ASI subgroup STRUCTURE DETAILS equals or is greater than 2000, then execute the action. A period delimits the ASI subgroup name which precedes the ASI field name. <strong>Note:</strong> You can configure a global variable to precede all ASI field names with the ASI subgroup name (Configuring the Global Variables on page 268).</td>
</tr>
<tr>
<td><code>{ParcelAttribute.Neighborhood} == “Downtown Area”</code></td>
<td>Similar to ASI fields, enclose a parcel attribute in curly brackets, and prepend it with ParcelAttribute and a period separator. In the example, if the parcel attribute Neighborhood equals Downtown Area then execute the associated action.</td>
</tr>
</tbody>
</table>
Understanding Criteria with Multiple Conditional Statements

Criteria (the if clause) can contain multiple conditional statements separated by the logical “and” operator (&&) and/or the logical “or” operator (||). All “and” conditions must be true in order for the criteria to be true. Only one “or” condition needs to be true in order for the criteria to be true.

You can use as many logical operators in your criteria as you need to satisfy your business rules. You use parenthesis to specify the evaluation order of criteria with multiple conditions and multiple operators (Table 6: Criteria Examples with Multiple Operators).

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proximity(&quot;GIS&quot;,&quot;Schools&quot;,parseInt({Number of feet}));</td>
<td>Similar to the appMatch function example, the function proximity returns true or false. The function checks to see if the parcel for the current record falls within a buffered distance on a layer within GIS. The example checks whether the current record’s parcel is within a certain number of feet (a value specified in an ASI field).</td>
</tr>
<tr>
<td>!taskStatus(&quot;Permit Issuance&quot;,&quot;Issued&quot;);</td>
<td>The taskStatus checks to see if a workflow task currently has a particular status. The example checks to see if the status of the permit issuance task updated to issued. You can use this type of check to prevent inspection scheduling before permit issuance.</td>
</tr>
</tbody>
</table>

Table 6: Criteria Examples with Multiple Operators

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspType == “Final Inspection” &amp;&amp; !isScheduled(&quot;Electrical&quot;)</td>
<td>This condition occurs during an inspection event. The criteria checks whether the inspection type that triggered the event is a final inspection and whether Accela Automation scheduled an electrical inspection. You can use this criteria during an InspectionScheduledBefore event to prevent a final inspection before an electrical inspection.</td>
</tr>
</tbody>
</table>
Understanding Actions (the Then Clause)

The right side of the script control (to the right of the caret) tells Accela Automation what to do if the criteria evaluates to true. In most cases the action portion calls a master script function to perform an action (Appendix A: Master Script Function List on page 114).

To execute multiple actions, you can write your script controls two ways; 1) list each action separated by a semicolon (;) on the same line (Table 7: Multiple Actions on Same Line), or 2) put each action on a different line (Table 8: Multiple Actions on Different Lines).

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>{Review Required} == “Yes” ^ addFee(&quot;REVIEWFEE&quot;,&quot;FEESCHEDULE&quot;,&quot;FINAL&quot;,1,&quot;Y&quot;); scheduleInspection(&quot;Special Review Inspection&quot;,1); //any additional action…</td>
</tr>
</tbody>
</table>

When you put multiple actions on different lines, start each new line with a caret (^).

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>{Review Required} == “Yes” ^ addFee(&quot;REVIEWFEE&quot;,&quot;FEESCHEDULE&quot;,&quot;FINAL&quot;,1,&quot;Y&quot;);</td>
</tr>
<tr>
<td>02</td>
<td>^ scheduleInspection(&quot;Special Review Inspection&quot;,1);</td>
</tr>
<tr>
<td>03</td>
<td>^ //any additional needs for the action…</td>
</tr>
</tbody>
</table>

Best practice recommends the multiple line approach due to width limitations for Standard Choice item entries.
To maintain consistency, best practice recommends the use of semicolons at the end of each line, even for single action script controls (Table 9: Single Action Script Controls with Semicolons).

Table 9: Single Action Script Controls with Semicolons

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>activateTask(&quot;Plan Review&quot;);</code></td>
<td>Activates the workflow task on the current record specified by the first parameter.</td>
</tr>
<tr>
<td><code>addAppCondition(&quot;Permit&quot;,&quot;Applied&quot;,&quot;Re-Inspection Fee&quot;,&quot;Re-Inspection Fee&quot;,&quot;Hold&quot;);</code></td>
<td>Applies a hold condition to the current record with the specific type, name, and description.</td>
</tr>
<tr>
<td><code>childApp = createChild(&quot;Building&quot;,&quot;Commercial&quot;,&quot;Plumbing&quot;,&quot;NA&quot;,&quot;New Walmart&quot;);</code></td>
<td>Creates a child record for the Building/Commercial/Plumbing/NA record type instance named New Walmart.</td>
</tr>
<tr>
<td><code>editAppSpecific(&quot;Total Value&quot;,parseInt({Sq Ft}) * parseInt({Price per Sq Ft}));</code></td>
<td>Updates the ASI field Total Value to the product of the ASI fields {Sq Ft} and {Price per Sq Ft}.</td>
</tr>
</tbody>
</table>

### Specifying Script Controls as Standard Choices

Figure 16: Standard Choice Annotated shows a Standard Choice script control named ApplicationSubmitAfter and Table 10: Standard Choice Script Controls defines the components on the Standard Choice UI.

Figure 16: Standard Choice Annotated
Some additional standard choice details to be remember:

- Standard Choices do not have an auto-save feature. Update your Standard Choice often to ensure you do not lose your work.
- You cannot lock a Standard Choice. An update someone else makes to a Standard Choice refreshes the Standard Choice with their changes and wipes out any changes you might have made, but not yet committed.

Table 10: Standard Choice Script Controls

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Name</td>
<td>Standard Choice name. The master script for each event designates the name of the standard choice that is the entry point for script execution. A script control can implement the branch function to refer to other script controls.</td>
</tr>
<tr>
<td>B</td>
<td>Description</td>
<td>Text area used to describe the purpose of the script controls that the Standard Choice contains. You can use this area to maintain a script control change log.</td>
</tr>
<tr>
<td>C</td>
<td>Status</td>
<td>You can designate a Standard Choice as Enabled or Disabled. When disabled, Accela Automation does not execute the script controls in the Standard Choice and does not return an error if a master script calls the Standard Choice.</td>
</tr>
<tr>
<td>D</td>
<td>Type</td>
<td>Specifies the type of Standard Choice. Use EMSE for script controls. The EMSE type designation does not affect any Accela Automation functions.</td>
</tr>
<tr>
<td>E</td>
<td>Value</td>
<td>Best practice recommends that you increment script controls by ten (eg. 10, 20, 30) to leave room for inserted script controls in the future. As of version 2.0 of the master script framework does not require sequential script control numbering.</td>
</tr>
<tr>
<td>F</td>
<td>Value Desc</td>
<td>Contains the script controls.</td>
</tr>
<tr>
<td>G</td>
<td>Debug Options</td>
<td>showMessage – when set to true, this option presents a pop-up window to the user with a custom message about script execution. showDebug – when set to true, 1, 2 or 3, this option present a pop-up window that displays debug information including variable values and script control results.</td>
</tr>
<tr>
<td>H</td>
<td>Script Controls Example</td>
<td>Lines 20-70 contain script control examples. The master script evaluates script controls in the order the Standard Choice specifies.</td>
</tr>
<tr>
<td>I</td>
<td>Active</td>
<td>You can set a script control to Active or Inactive. Select Update to enable a change. Accela Automation skips over script controls set to Inactive.</td>
</tr>
<tr>
<td>J</td>
<td>Delete</td>
<td>You can delete a script control. After confirming a deletion, Accela Automation permanently removes the item. You cannot undo a delete operation.</td>
</tr>
<tr>
<td>K</td>
<td>Update</td>
<td>Use to commit changes. This includes updating the description, status, type, value, value desc, and active flag.</td>
</tr>
<tr>
<td>L</td>
<td>Add</td>
<td>Enables the addition of a new Standard Choice.</td>
</tr>
<tr>
<td>M</td>
<td>Cancel</td>
<td>Enables you to navigate back to the page from which you came without committing changes.</td>
</tr>
</tbody>
</table>
Understanding Script Control Branching

Topics:
- Using Branching to Implement a For Loop
- Using Pop-Up Messages
- Using Data Validation
- Using Variable Branching
- Branching to the Same Standard Choice from Different Events

Each individual master script specifies the Standard Choice that provides the script controls for processing an event (Figure 17: Setting the controlString).

Figure 17: Setting the controlString

```javascript
var controlString = "ApplicationSubmitAfter";
```

The master script represented in Figure 17: Setting the controlString is for the event ApplicationSubmitAfter event. The value of the controlString variable name of the Standard Choice. For most master scripts the controlString value matches the event name for which the Standard Choice contains the script controls.

Master script evaluation of script controls begins with the first line in the Standard Choice and ends with the last line in the Standard Choice.

You can branch a script control process from one Standard Choice to another Standard Choice. The branch script control action functions like a sub-routine in traditional programming.

When a master script encounters a branch script control action, the master script stops evaluation of the current standard choice and begins evaluation of the script controls in the branched to Standard Choice. Use the following syntax to specify a branch action:

```
branch("<Standard Choice Name>")
```

where: `<Standard Choice Name>` is the name of the Standard Choice containing the branched to script controls.

In the example branch action (Table 11: Branch Action Example) the master script branches to the “Calculate Permitting Application Fees” Standard Choice when a workflow approves an application for processing. The master script then evaluates all the script controls in the “Calculate Permitting Application Fees” Standard Choice implement the application fees’ business rules.

Table 11: Branch Action Example

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td><code>wfTask == &quot;Application Acceptance&quot; &amp;&amp; wfStatus == &quot;Approve for Processing&quot; ^ branch(&quot;Calculate Permitting Application Fees&quot;);</code></td>
</tr>
</tbody>
</table>

After the master script evaluates all script controls in the branched to Standard Choice, the master script returns to the place that contains the originating branch action, evaluates anymore actions that remain on the same line, then moves onto the next line in that Standard Choice (Figure 18: Branching Flow).
You can branch to as many levels as required. The same rules that apply single level branching apply to multiple level branching. The master script completes evaluation of all script controls in the lowest level Standard Choice to which you branch first, and completes evaluation of all the script controls in the highest level Standard Choice, the Standard Choice that served as the entry point for the master script, last (Figure 19: Multiple Level Branching).

The flow of script control shown in Figure 19: Multiple Level Branching is as follows:

- Begin script control evaluation with line 01 of “WorkflowTaskUpdateAfter”
- Branch line 01 of “WorkflowTaskUpdateAfter” to “Calculate Permitting Application Fees”
- Continue script control evaluation with line 01 of “Calculate Permitting Application Fees”
- Branch line 02 of “Calculate Permitting Application Fees” to “Send Email Notifications”
- Continue script control evaluation with line 01 of “Send Email Notifications”, and continue script control evaluation through line 05
- Return to “Calculate Permitting Application Fees” and continue to evaluate script controls that follow the branch action, on line 02 through line 05
- Return to “WorkflowTaskUpdateAfter” and continue to evaluate script controls that follow the branch action, on line 01 through line 02
End script control evaluation after evaluating line 02 of “WorkflowTaskUpdateAfter”

**Using Branching to Implement a For Loop**

By default, JavaScript uses curly brackets { } to indicate the start and end of a unit of code for conditional statements or loops. In master script syntax, curly brackets indicate retrieval of a value (Understanding Curly Brackets on page 67) not the start and end of a unit of code. As a workaround, use branching to implement body of code functionality and loop functionality.

**Table 12: Incorrect Loop Using Curly Brackets** provides an incorrect example of a loop implemented with curly brackets.

**Table 12: Incorrect Loop Using Curly Brackets**

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>contactArray.length &gt; 0 ^ for (ca in contactArray) { thisContact = contactArray[ca];</td>
</tr>
<tr>
<td>02</td>
<td>^ if (thisContact[“email”] !=&quot;”) email(“<a href="mailto:noreply@accela.com">noreply@accela.com</a>”,thisContact[“email”],“Permit Update”,”Your permit has been issued.”); }</td>
</tr>
</tbody>
</table>

The master script returns several errors for these script controls due to incorrect use of curly brackets:

- Line 01 opens a curly bracket but does not close the curly bracket on the same line
- Line 02 closes a curly bracket but does not open the curly bracket on the same line

To workaround the syntax issue, you can use a branch action to designate a body of code for a loop (**Table 13: Branch Implementation for Body of Code Loop** and **Table 14: Contact Email Loop**).

**Table 13: Branch Implementation for Body of Code Loop**

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>contactArray.length &gt; 0 ^ for (ca in contactArray) branch(“Contact Email Loop”);</td>
</tr>
</tbody>
</table>

**Table 14: Contact Email Loop**

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>true ^ thisContact = contactArray[ca];</td>
</tr>
<tr>
<td>02</td>
<td>^ if (thisContact[“email”] != “”) email(“<a href="mailto:noreply@accela.com">noreply@accela.com</a>”,thisContact[“email”],“Permit Update”,”Your permit has been issued.”);</td>
</tr>
</tbody>
</table>

When using the branch action for a body of code loop, best practice prescribes appending the word “loop” to the end of the Standard Choice name.

**Using Pop-Up Messages**

Master scripts use two variables to specify whether or not to complete the transaction and the message contents to display in a pop-up window. The ScriptReturnCode variable specifies whether or not to complete the transaction.

```
aa.env.setValue(“ScriptReturnCode”, “<value>”);
```
where: `<value>` can be 0 or 1 and:
- 0 – indicates to proceed as normal
- 1 – stop the user action and return to the previous page.

The `ScriptReturnMessage` variable specifies the content of a message to display in a pop-up window.

```
aa.env.setValue("ScriptReturnMessage", "<myMessage>");
```

where: `<myMessage>` contains the content of the message to display.

You can use the `ScriptReturnMessage` to:
- notify users of an additional required activity
- notify users of a completed an activity (sent an email and added a condition, for example)
- notify users of useful information (the current location of the application, for example).

Note: Accela Automation does not display an empty message.

Figure 20: Pop-Up Message Example shows an example of a pop-up message and the accompanying variables in the master script.

Figure 20: Pop-Up Message Example

You can call the comment function for different script control actions to generate message text specific to evaluation of particular master script functions. Each message returned from the comment function displays on a new line in the pop-up window.

To display a pop-up message after evaluation of the last script control, set the `showMessage` function to true. If you do not set the `showMessage` function to true, no message displays, regardless of the number of times you call the comment function.

Table 15: Script Controls for Displaying Pop-up Messages shows how to call the comment and `showMessage` functions from a script control.

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>true ^ <code>showMessage</code> = true;</td>
</tr>
</tbody>
</table>
Figure 21: Message Window shows the resulting pop-up window generated by the script controls in Table 15: Script Controls for Displaying Pop-up Messages resulting from submission of a service request in Accela Automation.

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>true ^ comment(&quot;The service request submission is successful&quot;);</td>
</tr>
<tr>
<td>30</td>
<td>true ^ comment(&quot;Please remind the citizen to sign up on Accela Citizen Access to submit future requests and receive email status updates.&quot;);</td>
</tr>
</tbody>
</table>

You can use HTML tags in the strings submitted to the comment function to add additional formatting (bold, underlined, additional blank lines, for example).

The EMSE_DISABLE_MESSAGES Standard Choice controls display of messages to internal and public users. If you set the entry for either InternalUsers or PublicUsers to “Yes”, no pop-up messages display to the user.

### Using Data Validation

You can use a ‘before’ event type to validate submitted data, before saving to the database (Understanding Events on page 17), and cancel the transaction if the submitted data does not meet the data validation business rules that your scripts prescribe.

If a data submission attempt fails data validation, provide a message to the user as to why you cancelled the transaction (Using Pop-Up Messages on page 75). To stop the transaction, set the cancel variable in the script control to true.

```javascript
cancel = true
```

Table 16: Script Control for Data Validation provides script control example that cancels a transaction and tells the user why Accela Automation cancelled the transaction. Make sure that
the message indicates the reason for cancelling the transaction so the user can correct the situation.

Table 16: Script Control for Data Validation

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

01 (Project Name) == "" ^ showMessage = true; comment("You must designate a 'Project Name' in the ASI prior to processing the application"); cancel = true;

Figure 22: Cancelled Transaction Message shows the message displayed to the user.

Figure 22: Cancelled Transaction Message

Data validation can be especially helpful for many events, including the following:

- ApplicationSubmitBefore
- WorkflowTaskUpdateBefore
- InspectionScheduleBefore
- InspectionResultSubmitBefore
- PaymentReceiveBefore
- ApplicationStatusUpdateBefore
- UpdateContactTypeBefore

Using Variable Branching

To enable variable branching for all master scripts, set the enableVariablebranching variable in the "User Configurable Parameters" section of the INCLUDES_ACCELA_GLOBALS script to true (Configuring the Global Variables on page 268).

```java
enableVariablebranching = true;
```

Note: When you set variable branching to true, the documentOnly functionality does not work.

Figure 23: ApplicationSubmitAfter – Without Variable Branching shows an example of how the ApplicationSubmitAfter Standard Choice uses branching to organize scripts.
Without variable branching, you provide a separate script control branch action for each four level record type specification. If you have many unique record types in your implementation that require scripting, this approach involves many lines of script controls.

With variable branching, you use variables to specify the argument of the branch function instead of a literal string value. The master scripts resolve these variables and the branch function calls the appropriate Standard Choice.

Variable branching enables the branch function to accept string variables, in addition to hard coded strings concatenated together, as a single parameter. For example, with variable branching you can write the following:

```
true ^ branch("Assess Fees");
```

like the following:

```
true ^ variable1 = "Assess";
true ^ variable2 = "Fees";
true ^ branch(variable1 + " " + variable2);
```

You can use this principle to represent all possible four level record type specifications (Group/Type/Subtype/Category) with the following six variables:

```
branch(appTypeArray[0] + "//*/*/"");
branch(appTypeArray[0] + "/" + appTypeArray[1] + "/\*\*\*/" + appTypeArray[2] + "/\*\*/" + appTypeArray[3]);
branch(appTypeArray[0] + "//\*\*/" + appTypeArray[1] + "/\*/" + appArray[2] + "/\*/" + appArray[3]);
branch(appTypeArray[0] + "/\*/" + appTypeArray[1] + "//\*/" + appArray[2] + "//\*/" + appArray[3]);
branch(appTypeArray[0] + "//\*/" + appTypeArray[1] + "//\*/" + appArray[2] + "//\*/" + appArray[3]);
branch(appTypeArray[0] + "//\*/" + appTypeArray[1] + "//\*/" + appArray[2] + "//\*/" + appArray[3]);
branch(appTypeArray[0] + "//\*/" + appTypeArray[1] + "//\*/" + appArray[2] + "//\*/" + appArray[3]);
```

where the appTypeArray number in square brackets represents the level, of the four-level record type specification, contained in the array. When an event triggers, the master script resolves these variables based on the record type specification for the selected record.

The following provides an example resolution for an instance of the Licenses/Business/Taxi/Application record type:

```
branch(Licenses/\*\*/\*/)
branch(Licenses/Business/\*\*/)
branch(Licenses/Business/Taxi/\*/)
branch(Licenses/\*\*/Application)
branch(Licenses/Business/\*\*/Application)
branch(Licenses/Business/Taxi/Application)
```
Accela Automation Scripting Guide

4: Script Controls 80

The branched to Standard Choices contain the script controls for all records in the record type
hierarchy level indicated in the branch argument. For example, the script controls in the
“Licenses/*/*/*” Standard Choice apply to all license record types, including the (Licenses/
Business/Taxi/Application) record type, whereas the script controls in the “Licenses/Business/
Taxi/Application” Standard Choice only apply to instances of the Licenses/Business/Taxi/
Application record type.
The preceding example branches to the same Standard Choice, regardless of the event trigger.
To branch to a different Standard Choice for each event trigger, manually add an event
specification into the variable.
branch("<my_event>:" + appTypeArray[0]
branch("<my_event>:" + appTypeArray[0]
branch("<my_event>:" + appTypeArray[0]
appTypeArray[2] + "/*");
branch("<my_event>:" + appTypeArray[0]
branch("<my_event>:" + appTypeArray[0]
appTypeArray[3]);
branch("<my_event>:" + appTypeString);

+ "/*/*/*");
+ "/" + appTypeArray[1] + "/*/*");
+ "/" + appTypeArray[1] + "/" +
+ "/*/*/" + appTypeArray[3]);
+ "/" + appTypeArray[1] + "/*/" +

where <my_event> is the three to five character abbreviation that represents the event (Table
17: Scriptable Event Abbreviations). For example, you can use the ASA abbreviation to
represent the ApplicationSubmitAfter/Before event in the branch variable.
branch("ASA:"
branch("ASA:"
branch("ASA:"
*");
branch("ASA:"
branch("ASA:"
branch("ASA:"

+ appTypeArray[0] + "/*/*/*");
+ appTypeArray[0] + "/" + appTypeArray[1] + "/*/*");
+ appTypeArray[0] + "/" + appTypeArray[1] + "/" + appTypeArray[2] + "/
+ appTypeArray[0] + "/*/*/" + appTypeArray[3]);
+ appTypeArray[0] + "/" + appTypeArray[1] + "/*/" + appTypeArray[3]);
+ appTypeString);

which resolves to the following:
branch(ASA:Licenses/*/*/*)
branch(ASA:Licenses/Business/*/*)
branch(ASA:Licenses/Business/Taxi/*)
branch(ASA:Licenses/*/*/Application)
branch(ASA:Licenses/Business/*/Application)
branch(ASA:Licenses/Business/Taxi/Application)

You must create a Standard Choice with the same name as each possible evaluation outcome
of the branch argument variables. Use event acronyms and record type variables, in your
branch arguments, to ensure a standard naming convention for your branched to Standard
Choices, and to facilitate the organization and reuse of branched to script controls in the
Standard Choices for group level record types (Licenses/*/*/*). When you apply this standard
naming convention for your Standard Choices, you can use wildcard searches to return an
inventory of Standard Choices setup for a specific record type. For example:


%Licenses/Business/Taxi/% - returns all Standard Choices for taxi business licenses across
all events



%ASA:Licenses/Business/% - returns all Standard Choices for business licenses
application submittal



%/Application/% - returns all standard choices for application record types.


Branching to the Same Standard Choice from Different Events

If you branch to the same Standard Choice from different events:

- Prefix the name of the branched to Standard Choice with the letters “CMN” (common).
- Followed the prefix with the record type scope.
- Append the end of the script control with a short description of its function.

```
wfTask == "Final Review" && wfStatus == "Ready to Issue" ^
branch("CMN:Permits/*/*/*:INVOICE_ALL_FEES");
```

Naming Inspection Result Events

The following three events, that occur after an inspection result, violate the rule that the entry point Standard Choice (controlString value in the master script) match the event name:

- InspectionResultSubmitAfter (inspection result from AA Classic, GovXML, AMO, AW)
- V360InspectionResultSubmitAfter (inspection result from AA)
- InspectionResultModifyAfter (inspection updated from AA - FID 8400 disabled)
Best practice prescribes use of the same InspectionResultSubmitAfter Standard Choice for each of these events. Update the master script variable controlString in each event’s master script to “InspectionResultSubmitAfter”. Use the IRSA acronym to refer to this event in your branch variable (Using Variable Branching on page 78).

Exploring an Object

When working with an object while writing scripts you can reference the Javadocs documentation (http://community.accela.com/p/doc_interfaces.aspx) to explore the class it belongs to including its properties and methods. Use the getClass() function to determine the class from which EMSE instantiated an object.

You can use Script Test to create an object and use a for loop to explore the methods and properties available to the object (Figure 24: Show all methods of an object and Figure 25: Show all properties and their values for an object).

Figure 24: Show all methods of an object

Script Text:

```java
캡id = aa.cap.getCapID("C03-000059").getOutput();
capDetail = aa.cap.getCapDetail(capId).getOutput();

aa.print("capDetail is a " + capDetail.getClass());

for (x in capDetail) 
    if (typeof(capDetail[x]) == "function")
        aa.print(" " + x);
```

Script Output (script debug output will appear in this box when you submit this function)

- hasHashCode
- getBuildingCount
- getGAAgencyCode
- notifyAll
- setCreatedBy
- getGAAgencyCode
- getGAAgencyName
- getEndSignatureFlag
- equals
- getEndAssignDate
Figure 25: Show all properties and their values for an object

**Script Text:**

```javascript
capId = aa.cap.getCapID("C03-600659").getOutput();
capDetail = aa.cap.getCapDetail(capId).getOutput();

as.print("capDetail is a "+ capDetail.getClassName());

for (x in capDetail )
    if (typeof(capDetail[x]) != "function")
        as.print(" "+ x + ": "+ capDetail[x]);
```

**Script Output** (script debug output will appear in this box when you submit this file)

```javascript
capDetail is a class com.accela.as.emce.dom.CapDetailScriptModel
infractionFlag = null
referenceType =
assignStaff =
assignDept = null
balance = 23.5
percentComplete = 0
anonymousFlag = null
offmWitnessedFlag = null
ID3 = 00000
ID2 = 00000
ID1 = 00000
```
CHAPTER 5: ACCELA CITIZEN ACCESS PAGE FLOW SCRIPTS

Topics:
- Understanding Accela Citizen Access Page Flow Scripts
- Using Model Objects
- Creating a Page Flow Master Script

Understanding Accela Citizen Access Page Flow Scripts

When a citizen uses Accela Citizen Access to create an application, Accela Citizen Access creates a temporary record in the Accela Automation database and Accela Citizen Access stores application information in a capModel object. Accela Citizen Access stores capModel object data in memory for the duration of a user’s session.

The capModel object contains all the details about the application. As the user progresses through the forms on each Accela Citizen Access page, Accela Citizen Access updates the capModel object data in memory. Upon completion and submittal of the application, Accela Citizen Access passes the capModel object data to Accela Automation and Accela Automation creates a new record in the Accela Automation database.

You can use Expression Builder of page flow scripts to apply advanced business rules for Accela Citizen Access applications. If you need to populate data on an Accela Citizen Access form, try to use Expression Builder. If you need to populate data not displayed in Accela Citizen Access, use page flow scripts.

Unlike Accela Automation scripts, page flow scripts associate with events from the Accela Citizen Access Page Flow Admin tool. The user triggers page flow scripts when they navigate through different Accela Citizen Access pages. You can associate a script to the following three page flow events:
- Onload – triggers when the page loads
- Before – triggers when the user clicks the continue button, it can prevent the user from progressing if data validation fails
- After – triggers when the user clicks the continue button, can implement automation in the application process

Accela Automation master scripts interact with record data that Accela Automation stores in the database. The Accela Automation master scripts do not work on Accela Citizen Access page flow data.
For example, the `editAppSpecific` master script function updates an ASI field on the database record, but does not update the Accela Citizen Access capModel object data stored in memory. The Accela Citizen Access capModel object data overwrites the ASI field on the database record when Accela Citizen Access submits a completed application.

### Using Model Objects

The master script functions use `get` and `set` functions, with the “cap” variable, to retrieve and update information about the current record. Table 18: Retrieving the capModel Object Value shows how to get information, from the application specific information table, from a script control.

<table>
<thead>
<tr>
<th>#</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>true ^</td>
<td>asit = cap.getAppSpecificTableGroupModel ();</td>
</tr>
</tbody>
</table>

Table 19: Updating the capModel Object Value shows how to update information, from the application specific information table, from a script control.

<table>
<thead>
<tr>
<th>#</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>true ^</td>
<td>cap.setAppSpecificTableGroupModel(asit) ;</td>
</tr>
</tbody>
</table>

Table 20: Updating the capModel Object in Accela Citizen Access shows how to pass capModel updates to Accela Citizen Access at the end of your script.

<table>
<thead>
<tr>
<th>#</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>true ^</td>
<td>aa.env.setValue(&quot;CapModel&quot;, cap) ;</td>
</tr>
</tbody>
</table>

### Creating a Page Flow Master Script

You must customize each page flow script to the associated page flow.

**To create a custom page flow script**

1. Make a copy of the universal script.
2. Create a new name for the copy that accurately identifies. For example, ACA TN ASI Before, where:
   - ACA – indicates where the script runs
   - TN – is the page flow
   - ASI – is the page flow step
   - Before – is the event type (eg. Onload, After, Before)
3. Open the script in a text editor.
4. Update the controlString variable to match the name (Figure 26: Updating the controlString).

Figure 26: Updating the controlString

<table>
<thead>
<tr>
<th>Usage</th>
<th>Master Script by Accela. See ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>N/A</td>
</tr>
<tr>
<td>Action</td>
<td>N/A</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
</tr>
</tbody>
</table>

START User Configurable Parameters

Only variables in the following section will no longer be considered a “Master” changes are made, please add notes abc

```javascript
var showMessage = false;
var showDebug = true;
var preexecute = "PreexecuteBeforeEvents";
var controlString = "ACA 7N ASI Before";
var documentOnly = false;
var disableTokens = false;
var useAppSpecificGroupName = false;
```

5. Install the script (Working with Scripts on page 52). Save the script with the same name as the controlString variable set in the previous step.


7. Navigate to the proper page flow and page flow step.

8. Associate the newly added script to the proper event (Figure 27: Associating a script to an Accela Citizen Access Page Flow event).

Figure 27: Associating a script to an Accela Citizen Access Page Flow event

9. Create the standard choice entry point with the same name as the controlString variable.

10. Write appropriate script controls that interact with information stored in memory for the capModel object.

CHAPTER 6: 

SCRIPT TESTING

Topics
- Understanding the Script Test Tool
- Testing an Event and Script Association
- Running a Script Test
- Troubleshooting

Understanding the Script Test Tool

Accela Automation provides the Script Test tool to test EMSE scripts. The Script Text tool simulates script execution by running scripts and displaying the resulting output. However, scripts run in the Script Text tool do not change any values in Accela Automation, Accela Citizen Access, or the Accela Automation database.

You can use the Script Test tool to:
- Develop and test batch scripts.
- Develop and test custom functions.
- Troubleshoot and debug EMSE scripts.

To access the Script Test tool
1. From the Classic Admin portlet, click Admin Tools > Events > Script Test. 
   Accela Automation displays the Script Test interface.
Figure 28: Script Test

Warning: Improperly written scripts may incorrectly alter data for many records. Always be careful when writing and testing scripts.

Table 21: Script Test Field Definitions provides information on the fields in the Script Test interface.
Table 21: Script Test Field Definitions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| Script Transaction | A drop-down list that provides two options:  
  **Always Rollback**  
The script outputs results to the Script Output window, but does not commit actions in Accela Automation. For example, if the script updates a workflow task for each license that meets certain criteria, the Script Output window indicates an updated workflow task while the status of the record in Accela Automation remains unchanged.  
The Always Rollback selection does allow scripts to affect autonumbers for Accela Automation objects. For example, if the script assesses and invoices a fee item, the sequence numbers for fees and invoices increments even though Accela Automation did not create the fee as a part of the script test.  
  **Commit if Successful**  
The script commits requested actions in Accela Automation and displays a result message in the Script Output window. |
| Script Initializer | Contains initialization requirements for testing the script. For example, when testing a batch script, you can set batch script parameter values, like specifying an email address, to set the scope of the batch script to a record type designation. You can also set script initialization values in Script Text. |
| Script Text | Contains the contents of the script. In general, you should create scripts in a text editor, then copy and paste them into the Script Test section. |
| Script Output | Contains the returned output upon the completion of the script execution. The script only displays text strings it sends to the aa.print method ("This should be sent to the Script Output window"). If your script contains the logMessage or logDebug function, make sure you send the variable that contains the debug or message output to the aa.print method (Using the aa.print Function on page 98). |

**Note:** Scripts that run longer than the specified EMSE time-out do not exit as gracefully as they do from a batch job or a set script execution.  
Script errors display in the next encountered error pop-up window as well as the Script Output section.  
The first couple lines of an error message often indicate an undefined variable or a non-existent function.  
The error message typically the line in the script where the error occurred.
Testing an Event and Script Association

Topics

- Associating the script to an event
- Testing the event

In this scenario we install a test script and we associate the test script with an event. When the event triggers, the script displays the “EVENT TEST” pop-up message.

You can associate the test script with any event. When the “EVENT TEST” message appears in a pop-up window, you know the event that triggered the test script.

During your script development, create the association between the test script and the event you want to script first, then replace the test script with your developed script. This way, you know that you associated the developed script with the correct event.

To create the test script
1. From the Classic Admin portlet, click **Admin Tools > Events > Scripts**.
2. Click Submit to return a list of existing scripts.
3. Click Add.
4. Enter the following two script lines in the Script Content section:
   
   ```
   aa.env.setValue("ScriptReturnCode","1");
   aa.env.setValue("ScriptReturnMessage", "EVENT TEST");
   ```
5. Enter the name “EVENT_TEST” in the Script Code and Script Title sections.
6. Click Add.

Associating the script to an event

To associate an event to the test script
1. From the Classic Admin portlet, click **Admin Tools > Events > Events**.
2. Click Submit to return a list of existing events.
3. If required, add a new event.
   a. Click Add.
   b. Select the event name from the Events drop-down list, then click Add.
4. Select an existing event by clicking on the red dot to the left of its name.
5. From the Event Detail screen select the script “EVENT_TEST” from the Script Name drop-down list.
6. Click Save.
Testing the event

Test the event to script association by triggering the event with a test case. For example, to test the “ApplicationSubmitAfter” event, create a new record of any type and click Submit, the “EVENT TEST” message appears in a pop-up window.

To disassociate the script from the event
1. From the Classic Admin portlet, click Admin Tools > Events > Events.
2. Click Submit to return a list of existing events.
3. Select the event by clicking on the red dot to the left of its name.
4. From the Script Name drop-down list, scroll to the top of the list and select the blank entry.
5. Click Save.

To delete the event
1. From the Classic Admin portlet, click Admin Tools > Events > Events.
2. Click Submit to return a list of existing events.
3. Select the event by clicking on the red dot to the left of its name.
4. Click Delete to disable the event.

Figure 29: Script and Event Detail pages

Running a Script Test

Topics
- Using ScriptTester

Incorrect scripts can permanently alter or erase data on your system. Always test your scripts before you implement them.
Use the Accela Automation script test utility to run your script in a test situation and to view the effects of the scripts or any errors that it generates.

When you run a script in the script test utility, Accela Automation runs the script and changes your system accordingly. You configure runtime parameters for the script test tool that instructs Accela Automation whether to rollback all changes resulting from the script or commit the changes resulting from the script.

**Using ScriptTester**

Accela Automation provides the ScriptTester.js master script file for you to test script controls without triggering an event from the user interface. You can copy and paste the content of the ScriptTester.js file into the Script Test tool.

ScriptTester is a file that allows you to copy and paste its contents into Figure 28: Script Test on page 88 and test script controls without having to trigger an event from the user interface.

**To use the Script Test tool**

1. Copy and paste the ScriptTester.js contents into the Script Text area of the Script Test tool.
2. Edit the myCapId variable to the tested altId.
3. Edit the myUserId to the tested user.
4. Update controlString to the standard choice entry point.

**Note:** The control string can be a standard choice entry point for the event (eg. ApplicationSubmitAfter) to test an events standard choices or a specific standard choice to test specific functionality.

The ScriptTester.js master script inherits the native functionality of the Script Test tool to Always Rollback or Commit if Successful. You can use Always Rollback to repeatedly test a script and not commit the results to the database. You can select Commit if Successful after you troubleshoot your script and want to update the database.

**Note:** Always Rollback is the default.
Launching the EMSE Debug Tool

Accela Automation provides an EMSE debug tool so that users can debug scripts conveniently. Follow the instructions to launch the EMSE debug tool.

1. Log in to Accela Automation Classic.
2. Navigate to Admin Tools > Events > Script Test.
3. Enter // @Open sesame in the Script Text field.
4. Click the Submit button.

The Rhino JavaScript Debugger appears on Accela Automation Application Server.

5. Debug the scripts that populate the JavaScript Console of the debug tool.

Several kinds of scripts, such as batch job scripts and scripts that EMSE events trigger, can populate the debug tool. For example, when submit an application, you trigger the
ApplicationSubmitAfter event and the associated script automatically populates the debug tool. The debugger window looks like the following.

Understanding the Script Output Window

You can use the script output pop-up window to provide additional details about an event and associated script to help locate problems or areas in the script on which you want to focus.

The showDebug variable controls script output according to the following settings:

- 0 or false – no output to the screen or server logs
- 1 or true – generates screen output only
- 2 – outputs to the server log only
- 3 – outputs to the screen and the server log
The script output window displays the script flow, which includes evaluated criteria and executed actions (yellow section in Figure 32: Script Output with Action). An action only appears in the script output window if the criterion allows it. To locate an action in the script output window, use your browser to search for the word Action.

The script tester writes more and different information to the server.log file than the script output window. If the script output window does not provide enough information to troubleshoot your problem, check the server.log file.
Figure 33: Script Output Error Message shows an error message resulting from a misspelled function, but it does not show where it occurred. Figure 34: Server Logs Error Detail shows the server.log file with the debug output. The server.log file shows the error and where in the standard choice the script stopped executing.

If you set the showDebug variable correctly and the script output window does not display, check for the following two situations:

- an error prevents the display
- a script control later in the flow sets the showDebug variable to false.

Setting the showDebug Script Control

You can set the showDebug script control, with true as the criteria (Table 22: Common showDebug Implementation). In this case, the script output window displays to any user that triggers the associated event, which can confuse users not familiar with the testing process and EMSE.

Table 22: Common showDebug Implementation

<table>
<thead>
<tr>
<th>#</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>true ^ showDebug = 3;</td>
</tr>
<tr>
<td>20</td>
<td>// several more lines of script controls</td>
</tr>
</tbody>
</table>

To limit the showDebug function to a specific user, add a criteria to specify that the showDebug script control only applies to a specific user (Table 23: showDebug for Specified User).
Using the aa.print Function

The aa.print function outputs text to the Script Output window. A script test or a script control can call the aa.print function. When called from a script control, the aa.print function output displays at the bottom of the script output window (Figure 35: aa.print()).

Figure 35: aa.print()

```
Finished: ApplicationSpecificInfoUpdateAfter, Elapsed Time: 0.047 Seconds

Script APPLICATIONSPECIFICINFOUPDATEA

here is aa.print. My capID is 12CAP-00000-0001F
```

Using Biz Server Logs

You can write your script results to the server.log file on the Accela Automation Biz server.

To write script results to the Biz server

Set the DEBUG Standard Choice with the following settings (Figure 36: DEBUG standard choice):

<table>
<thead>
<tr>
<th>Standard Choice Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>DEBUG</td>
</tr>
<tr>
<td>Status</td>
<td>Enable</td>
</tr>
<tr>
<td>Type</td>
<td>System Switch</td>
</tr>
<tr>
<td>Standard Choice Value</td>
<td>ENABLE_DEBUG</td>
</tr>
<tr>
<td>Value Desc</td>
<td>YES</td>
</tr>
</tbody>
</table>
You can locate the log file on the server running the Accela Automation Biz server in the following location:

```
\Biz_Server\C$\Accela\av.biz\log\server.log
```

where C$ represents the root drive specification for the Accela Automation installation.

The server.log file represents the log for the current day. Each day at 12:00AM server time the server appends the date to the name of the previous day's log file. The daily history of log files remains on the server until an administrator decides to purge historical log files to free up space on the server hard disk.

If you do not have access to the server, you can request the server administrator deploy a log monitor. See the Accela Community for details of deploying a log monitor: [http://community.accela.com/accela_automation/m/aascripts/384.aspx](http://community.accela.com/accela_automation/m/aascripts/384.aspx)

A log viewing application such as BareTailPro ([http://www.baremetalsoft.com/baretailpro/index.php](http://www.baremetalsoft.com/baretailpro/index.php)) can enhance the log review process and speed up research and troubleshooting with the biz server logs.

To write specific messages to the biz server log, use the aa.debug function. The `aa.debug(string,string)` function writes the strings captured in the function call to the server.log file (Figure 37: `aa.debug(string,string)`).
This chapter provides a tutorial-like discussion of the Accela Automation object model. The functions that the Accela Automation master scripts provide reference these objects (Appendix A: Master Script Function List on page 114).

Topics:
- Discussing the Accela Automation Object Model
- Understanding Script Return Values

**Discussing the Accela Automation Object Model**

The Accela Automation object model comprises a hierarchy of objects that organizes access to different parts of Accela Automaton. At the root of the tree is the \textit{aa} object.

Figure 38: Object Model shows that the \textit{aa} object is at the root, with a few of the objects underneath the \textit{aa} object listed. Each of the objects listed beneath the \textit{aa} object is a property of the \textit{aa} object.

From earlier examples, we know that the \textit{aa} object also has methods. Figure 39: Object Model Root Methods shows two of the methods of the \textit{aa} object. You can find documentation for all the methods of the \textit{aa} object in the EMSE Javadocs.
Notice that each of the objects under the \textit{aa} object has a name that corresponds to a piece of Accela Automation. Each of these objects beneath \textit{aa} also provides methods for interacting with Accela Automation (Figure 40: Object Model Object Methods).

In this diagram, we can see that the inspection object has some methods that we can use. If we look at the documentation for the \textit{getInspections} method we find the method definition:

\texttt{getInspections(CapID capID) returns Result}

The method syntax tells us that the name of the method is \textit{getInspections} and that this method takes one parameter. Two words describe each parameter. The first word tells us the kind of parameter. The second tells us the parameter named. The name helps us to understand how to use the parameter inside the method. The type is “String”, “Number”, or perhaps the name of an object. In the case of this parameter the type is \textit{CapID}. We can look at the documentation for the method’s parameters and see:

\texttt{capID – The CapID for the record from which you want to get the array of inspections.}
We can also look up the CapID object, and read its description to find out more about it. So now, we know that we need to pass in a CapID object that identifies the record for which we want to get an array of inspections.

If we look at the end of the method definition we see “returns Result”. This last part of the method tells us that, when we call this method, we get back a result object. The result object provides an indicator of whether the method succeeded, an error type and error message if the method failed, and the output if the method succeeded.

The getSuccess method returns a Boolean value that is true if the method succeeded and false if it did not. If the getSuccess method returns false you can check the getErrorType and getErrorMessage methods return values to retrieve some information about the error. The getSuccess method returns true you can retrieve the actual output of the method by calling getOutput.

When we look at the documentation for the getInspections method we can see that the getOutput method of the result object returned by the getInspections method returns InspectionItem. The double brackets [ ] tell us that it is an array of InspectionItem objects and not just one InspectionItem object.

Return Value:

Result – Object, see description in this document. The getOutput method of the result object returns InspectionItem[], an array of InspectionItem object. See InspectionItem object description in this document.

At this point, we still have three questions. First, how do we create a CapID object that identifies the record we want. Second, how do we call the method? Third, how do we work with the array of InspectionItem objects returned to us by calling this method?

For the first question, we notice that there is a cap object beneath the aa object. We go to the reference documentation and see that the cap object has this method:

gcapID(String id1, String id2, String id3) returns Result

Return Value:

Result - Object, see description in this document. The getOutput method of the result object returns a CapID object. The getErrorMessage method returns CapNotFound if the method does not find a record that matches the three five digit ids.

This method returns a result object that has the CapID object we need. The method takes three strings that are the three ids for the permit. You can set up your Accela Automation instance to use a custom id, rather than a fifteen digit id, and a method of the cap object allows you to retrieve a CapID object using a custom id. Now we need to know how to call this method. Here is how:

```java
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
    myCap = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
    aa.abortScript();
}
```

We can see from this line that we access the cap object as a property of the aa object and then call the getCapID method of the cap object. We pass the three strings, that identify the record
we want, to the method . The method returns the result object, which we use to see if the
getCapID method succeeded. The retrieved CapID object, by calling the getOutput method of
the result object, identifies our record and we assign that object as the value of the myCap
variable.

Now we have the value we need as the parameter to pass in to the getInspections method. We
can see, from the example of calling a method of the cap object, how to call the getInspections
method of the inspection object. We know that we get back an array object from the
getInspections method call, so now we know how to write a two line script that retrieves an
array of inspections for a particular record. Here is our script so far:

```
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
    myCap = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
    aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);

if(myResult.getSuccess()) {
    myInspections = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
    aa.abortScript();
}
```

This script does not display any output yet. At this point, we should note two things. First, use a
record id that exists in the Accela Automation instance for which you are writing scripts. Second,
use a record that has scheduled, resulted, or cancelled inspections.

What happens if you choose a record that does not exists? If the method does not find a record
that matches the id, the result object’s getSuccess method returns false, and you need to check
the error type and error message.

What happens if the record does not include any scheduled, resulted, or cancelled inspections?
You get back a zero length array that means there are no inspections for the record you
selected. We come back to these possibilities a bit later, but for now let us assume that we have
the right record id, and that the record includes inspections.

Now we need to do something with the array of InspectionItem objects we got back from calling
the getInspections. What can we do? Well, let us start by trying to print out some information
about the inspections scheduled. We want to print out a few important pieces of information
about each inspection. We go to each element in the array and call some methods on the object
stored at that position. This example reminds us of the example use of a ‘while’ loop. Here is the
example again for your review:

```
myArray = new Array();
myArray[0] = "Oranges";
myArray[1] = "Bagels";
myArray[2] = "Spinach";
i = 0;
while(i < myArray.length) {
    aa.print(myArray[i]);
}
```
This example approximates what we need. We have an array and we want to loop over its elements. So let's try adding a rough version of this to our script:

```javascript
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
  myCap = myResult.getOutput();
} else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
  myInspections = myResult.getOutput();
} else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}
i = 0;
while(i < myInspections.length) {
  // At this point we need to get the inspection and do something with it to print.
  i = i + 1;
}
```

Now we have added five more lines to our script that execute a while loop one time for each element in the array. If we have three items in our array, the loop counter has the values 0, 1, and 2. When the loop counter reaches three, the loop stops repeating. Inside the loop, we have a comment as a placeholder for a print function.

Inside the loop, we retrieve the InspectionItem object from the array, that is at the position specified by the loop counter, and we use that object to print out the information. Add the line to retrieve the object:

```javascript
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
  myCap = myResult.getOutput();
} else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
  myInspections = myResult.getOutput();
} else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}
i = 0;
```
while(i < myInspections.length) { 
    theItem = myInspections[i];
    // At this point we need to use the object to print some stuff.
    i = i + 1;
}

After adding a line to the beginning of the loop we now have a variable that contains the InspectionItem object at the current position in the array. Now we just use that object to print out the inspection id number, type, and status. Here is the script:

    myResult = aa.cap.getCapID("01BLD", "00000", "00027");
    if(myResult.getSuccess()) {
        myCap = myResult.getOutput();
    } else {
        aa.print(myResult.getErrorMessage());
        aa.abortScript();
    }
    myResult = aa.inspection.getInspections(myCap);
    if(myResult.getSuccess()) {
        myInspections = myResult.getOutput();
    } else {
        aa.print(myResult.getErrorMessage());
        aa.abortScript();
    }
    i=0;
    while(i < myInspections.length) {
        theItem = myInspections[i];
        aa.print(theItem.getIdNumber());
        aa.print(theItem.getInspectionType());
        aa.print(theItem.getInspectionStatus() + "\n");
        i = i + 1;
    }

If you run this script with the right record id you receive an output that, depending on the inspections for the record and their status, looks something like this:

    4238
    Trenches
    Scheduled
    4257
    Reinforcing
    Approved
    4293
    Foundation Wall
    Approved

Now we have a useful script that retrieves some important information about a record. In our script we used three methods of the InspectionItem object: getIdNumber, getInspectionType, and getInspectionStatus. These methods do not take any arguments because their purpose is only to return information about the inspection to your script.
The tenth line of the script shows that we added the special character “\n” at the end of the value that the `getInspection` status method returned and passed the resulting string to the `print` method. This special character added an extra blank line in between each inspection’s printed values.

Up until this point, we have always used the `print` method to produce output from our script that we can see, but many of the scripts that you write for Accela Automation do not produce output in this way. You may want to modify a record’s workflow or automatically assess a fee when you schedule a new inspection. In other words, the output of your script may modify some data in Accela Automation.

As an example, we are going to check for a problem with the statuses of the inspections of our record, and if a problem exists, we are going to create a smart notice to let staff know. Let us suppose that we do not want to approve a Foundation Wall inspection before there is an approved Trenches inspection. If this scenario happens we want to create a smart notice that informs staff that the record with the record id we were checking has this problem.

So how do we approach this scenario? Well, first we need to know if there is an approved Foundation Wall inspection. If there is, then we need to know if there is an approved Trenches inspection. We already have a script that loops over all the inspections for a record. We can start from there, but instead of printing out information about the inspection, we want to see if the inspection is an approved Foundation Wall inspection. Let us add this check to the script:

```java
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
    myCap = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
    aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
    myInspections = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
    aa.abortScript();
}
i=0;
while(i < myInspections.length) {
    theItem = myInspections[i];
    if(theItem.getInspectionType() == "Foundation Wall" &&
        theItem.getInspectionStatus() == "Approved") {
        //Check to see if there is an approved Trenches inspection.
        i = i + 1;
    }
}
```

When this script executes let us suppose that it finds an inspection that is a Foundation Wall inspection with a status of Approved. When this scenario happens we need to check to see if there is an approved Trenches inspection.

The check for a Trenches inspection requires that we use a second loop inside our main loop, but we can simplify things by using a function. Let us add a function that checks to see if there is
a Trenches inspection that is Approved and a condition that uses our new function to do the checking:

Function
checkForApprovedTrenchesInspection(inspectionItemArray) {
  j = 0;
  while(j < myInspections.length) {
    currentInspection = myInspections[j];
    if(currentInspection.getInspectionType() == "Trenches" &&
      currentInspection.getInspectionStatus() == "Approved") {
      return true;
    }
    j = j + 1;
  }
  return false;
}

myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
  myCap = myResult.getOutput();
} else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
  myInspections = myResult.getOutput();
} else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}

i=0;
while(i < myInspections.length) {
  theItem = myInspections[i];
  if(theItem.getInspectionType() == "Foundation Wall" &&
    theItem.getInspectionStatus() == "Approved") {
    if(checkForApprovedTrenchesInspection(myInspections) ==
      false) {
      // If we reach this line we have confirmed that the record has a problem.
    }
  }
  i = i + 1;
}

Our function looks very similar to the previously written part of our script. We have placed the function at the top of the script, although you could also put it at the bottom as a matter of preference. We have tried to give our function a meaningful name that tells us what it does. We could also put a comment before the function to explain to other people reading our script what the function does.
The function accepts one parameter, an inspection array. We named the parameter `inspectionItemArray` to remind ourselves of what type of value we need to pass in when calling the function. The function loops over the array passed in and checks to see if each of the inspections meets the condition that it is a Trenches inspection with Approved status. As soon as the function finds an inspection it returns the value true.

When a return statement appears in the middle of a function like this, the function stops what it is doing and immediately returns the specified value; it does not wait for the loop to finish. If the loop goes all the way through the inspections for the record and does not find a matching inspection, the loop exits and the next command after the loop executes. The command after the loop is "return false", so if the function gets through the loop without finding a matching inspection, the function returns false.

The condition we added to the middle of the previously written loop uses the new function to check for an Approved Trenches inspection. If the function does not find an inspection, we know that the record has inspection statuses inconsistent with how we want to run our agency and we need to do something.

Let us suppose that the inspection matches the conditions we have set up so far. In this case, we need to replace the comment line in our script with a command to insert a smart notice with the information that we need. The smartNotice object has this method:

```javascript
addNotice(String id1, String id2, String id3, String activityType, String activityComment) returns
null
```

So when we confirm that the record has the problem, we need to call this method to create the new smart notice. After adding this method call to the script here is what we get:

```javascript
function checkForApprovedTrenchesInspection(inspectionItemArray) {
    j = 0;
    while(j < myInspections.length) {
        currentInspection = myInspections[j];
        if(currentInspection.getInspectionType() == "Trenches" &&
            currentInspection.getInspectionStatus() == "Approved") {
            return true;
        }
        j = j + 1;
    }
    return false;
}
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
    myCap = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
    aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
    myInspections = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
}
```
aa.abortScript();
}
i=0;
while(i < myInspections.length) {
    theItem = myInspections[i];
    if(theItem.getInspectionType() == "Foundation Wall" &&
        theItem.getInspectionStatus() == "Approved") {
        if(checkForApprovedTrenchesInspection(myInspections) ==
            false) {
            aa.smartNotice.addNotice("01BLD", "00000", "00027",
                "Inspection Problem",
                "Application 01BLD 00000 00027 has an approved
                Foundation Wall inspection" +
                "but no approved Trenches inspection.");
        }
    }
    i = i + 1;
}

So now, we have a script that checks a record to see if it meets a certain criteria, and takes
action by inserting a new smart notice if the record does meet the criteria.

The list of Accela Automation events includes an event called InspectionResultSubmitAfter that
is a good place to run our script and check the application. However, as our current script only
checks the record 01BLD-00000-00027. We need to modify our script to that it uses the input
parameters from the event to dynamically determine which application to check.

The documentation for the InspectionResultSubmitAfter event shows three parameters that can
tell us which record to check:

    IN: PermitId1
    IN: PermitId2
    IN: PermitId3

We need to use the getValue method of the env object to retrieve parameters passed in to our
script from an event. We add the following three lines at the top of our script to retrieve the
permit id values:

    myId1 = aa.env.getValue("PermitId1");
    myId2 = aa.env.getValue("PermitId2");
    myId3 = aa.env.getValue("PermitId3");

After adding these lines at the beginning of the script, we use the three values we retrieved in
place of the unchanging strings we passed as parameters to getCapID. We also use these
values to dynamically create the message for the smart notice. Here is the script:

    function
    checkForApprovedTrenchesInspection(inspectionItemArray) {
        j = 0;
        while(j < myInspections.length) {
            currentInspection = myInspections[j];
            if(currentInspection.getInspectionType() == "Trenches" &&
                currentInspection.getInspectionStatus() == "Approved") {
                return true;
j = j + 1;
}
return false;
}

myId1 = aa.env.getValue("PermitId1");
myId2 = aa.env.getValue("PermitId2");
myId3 = aa.env.getValue("PermitId3");
myResult = aa.cap.getCapID(myId1, myId2, myId3);
if(myResult.getSuccess()) {
    myCap = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
aa.abortScript();
}

myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
    myInspections = myResult.getOutput();
} else {
    aa.print(myResult.getErrorMessage());
aa.abortScript();
}

i=0;
while(i < myInspections.length) {
    theItem = myInspections[i];
    if(theItem.getInspectionType() == "Foundation Wall" &&
        theItem.getInspectionStatus() == "Approved") {
        if(checkForApprovedTrenchesInspection(myInspections) ==
            false) {
            aa.smartNotice.addNotice(myId1, myId2, myId3,
                "Inspection Problem",
                "Application “ + myId1 + “” + myId2 + “ “ + myId3 +
                " has an approved Foundation Wall inspection" +
                “ but no approved Trenches inspection.”);
        }
    }
    i = i + 1;
}

You can now associate the script with the InspectionResultSubmitAfter event. However, there is one more thing to do.

Currently, we use aa.print to send messages to the user when something goes wrong. While aa.print works with the Script Test page, use the environment object to send messages back to the user when you attach the script to an event. Here is the final script with the aa.print statements, replaced with the appropriate statements for informing the user with a message:

```javascript
function
checkForApprovedTrenchesInspection(inspectionItemArray) {
    j = 0;
    while(j < myInspections.length) {
        currentInspection = myInspections[j];
```
if (currentInspection.getInspectionType() == “Trenches” 
&& currentInspection.getInspectionStatus() == “Approved”) {
    return true;
}

j = j + 1;

return false;

myId1 = aa.env.getValue(“PermitId1”);
myId2 = aa.env.getValue(“PermitId2”);
myId3 = aa.env.getValue(“PermitId3”);
myResult = aa.cap.getCapID(myId1, myId2, myId3);
if (myResult.getSuccess()) {
    myCap = myResult.getOutput();
} else {
    aa.env.setValue(“ScriptReturnMessage”,
                    myResult.getErrorMessage());
    aa.abortScript();
}

myResult = aa.inspection.getInspections(myCap);
if (myResult.getSuccess()) {
    myInspections = myResult.getOutput();
} else {
    aa.env.setValue(“ScriptReturnMessage”,
                    myResult.getErrorMessage());
    aa.abortScript();
}

i=0;
while(i < myInspections.length) {
    theItem = myInspections[i];
    if (theItem.getInspectionType() == “Foundation Wall” 
        && theItem.getInspectionStatus() == “Approved”)
        if (checkForApprovedTrenchesInspection(myInspections) == false) {
            aa.smartNotice.addNotice(myId1, myId2, myId3,
            “Inspection Problem”,
            “Application “ + myId1 + “ “ + myId2 + “ “ + myId3 + 
            “ has an approved Foundation Wall inspection“ + 
            “ but no approved Trenches inspection.”);
        }
    i = i + 1;
}

Before you deploy a script like this to an environment where real users use Accela Automation,
test the script thoroughly to make sure that it works as expected. Use a test environment where
any mistakes do not affect your production data.

To test a script, go in to the Event Manager pages and associate the script with the event, then
try to exercise different parts of the script. For example, try doing several different kinds of
inspection results that do not insert a smart notice. Then try some inspection results that do
insert a smart notice. With proper testing you can be much more sure that the script works,
before you deploy it in your production environment.

Understanding Script Return Values

When execution of a script completes, the script sends the values stored in the scripts
environment object back to Accela Automation. Some events have documented special values
in the environment, which you can set to send information to the Accela Automation interface.
The event documentation provides the names for these values and the expected input from
your script.

In addition to the environment return values related specifically to the event there are some
return values that are always available for you to set. You do not need to provide these
parameters a value unless you want something specific to happen.

Note that there are both before and after events in Accela Automation. An event with a name
that ends in “Before” means the event takes place before the user action updates the database.
An event with a name that ends in “After” means the event takes place after the user action
updates the database.

Because script return values can stop Accela Automation from continuing a user action, if you
associate a script with an after event, the user action completes before the script has a chance
to stop it. If you want to be able to cancel a user action, use a before event.

Topics:

- ScriptReturnCode
- ScriptReturnMessage
- ScriptReturnRedirection

ScriptReturnCode

This return value is a numeric value (Table 24: ScriptReturnCode Values) which allows the
script to choose one of several actions to occur after the script finishes.

Table 24: ScriptReturnCode Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Proceed as normal.</td>
</tr>
<tr>
<td>1</td>
<td>Request Accela Automation to stop the user action and return to the previous page.</td>
</tr>
<tr>
<td>2</td>
<td>Request Accela Automation to stop the user action and return to the main menu.</td>
</tr>
<tr>
<td>3</td>
<td>Request Accela Automation to stop the user action and proceed to the page designated by the ScriptReturnRedirection value.</td>
</tr>
<tr>
<td>4</td>
<td>Request Accela Automation to stop the user action and log user out.</td>
</tr>
</tbody>
</table>
**ScriptReturnMessage**

When this return parameter has a value, it displays to the user as a message when the Accela Automation user request, of which the script is a part, completes. You can use this function to send informative messages, or explanations of why an error occurred.

You can use HTML characters to format the message. The text of the returned message displays to the user in a popup window.

When you set the ScriptReturnCode parameter to something other than zero, you can set a value for ScriptReturnMessage to explain to the user why they are redirected from the normal flow of the Accela Automation interface.

**ScriptReturnRedirection**

When you set the ScriptReturnCode parameter to three, and you set a value for ScriptReturnCode, the script sets the URL of the browser to the value of the ScriptReturnRedirection parameter.
Conventions

- Unless otherwise stated, all function names and parameter values are case sensitive.
- Enter function parameters in the order listed.

**Note:** The name of a function parameter is for descriptive purposes only. The name of the function parameters in this chapter can differ from the name of the corresponding function parameter in the UniversalMasterScript file.

- For the string data type, enclose the parameter value in double-quotes.
- Subscripts 1 and n in parameter names (e.g., `wfTask1`, … `wfTaskn`) indicate that you can add between one and any number of such parameters, each in double-quotes and separated by commas.
- This reference shows Boolean values as true or false.
- This reference does not document internal functions in the master script files.

CapIDModel Type

The master script functions use the CapIDModel type for the capID parameter. Chapter 7: Accela Automation Object Model on page 100 provides additional details on the capID parameter. The EMSE Javadocs contain details on the CapIDModel class. The `com.accela.aa.aamain.cap` package in the EMSE Javadocs contains the CapIDModel class and defines the constructor for this class as follows.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>serviceProviderCode</td>
<td>string</td>
</tr>
<tr>
<td>ID1</td>
<td>string</td>
</tr>
<tr>
<td>ID2</td>
<td>string</td>
</tr>
<tr>
<td>ID3</td>
<td>string</td>
</tr>
<tr>
<td>customID</td>
<td>string</td>
</tr>
<tr>
<td>trackingID</td>
<td>long</td>
</tr>
</tbody>
</table>
activateTask

Makes workflow task \textit{wfstr} active and not completed, so that users can edit \textit{wfstr}.

\textbf{Version}

1.3

\textbf{Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{wfstr}</td>
<td>string</td>
<td>Name of task to activate.</td>
</tr>
<tr>
<td>\textit{wfRelationSeqId} (optional)</td>
<td>number: long</td>
<td>Relation sequence ID of workflow process to which \textit{wfstr} belongs.</td>
</tr>
</tbody>
</table>

\textbf{Notes}

If workflow uses sub-processes that contains duplicate workflow task names, use parameter \textit{wfRelationSeqId} to specify the process or subprocess whose \textit{wfstr} you want to activate. You can find the value of \textit{wfRelationSeqId} by querying workflow tables (e.g. GPROCESS.RELATION_SEQ_ID)

addAddressCondition

Adds a condition to the specified reference address. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

\textbf{Version}

2.0

\textbf{Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{addNum}</td>
<td>long</td>
<td>Reference address number or null.</td>
</tr>
<tr>
<td>\textit{cType}</td>
<td>string</td>
<td>Type of condition (from admin-&gt;condition-&gt;condition type).</td>
</tr>
<tr>
<td>\textit{cStatus}</td>
<td>string</td>
<td>Status (from admin-&gt;condition-&gt;condition status).</td>
</tr>
<tr>
<td>\textit{cDesc}</td>
<td>string</td>
<td>Description of the condition.</td>
</tr>
<tr>
<td>\textit{cComment}</td>
<td>string</td>
<td>Condition comment.</td>
</tr>
<tr>
<td>\textit{cImpact}</td>
<td>string</td>
<td>Must be Lock, Hold, Notice, Required, or \textit{&quot;}&quot;.</td>
</tr>
</tbody>
</table>

\textbf{Notes}

If \textit{addNum} is null, the function adds the condition to all reference addresses associated with the current record.
addAddressStdCondition

Adds a standard condition to the specified reference address. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addNum</td>
<td>long</td>
<td>Reference address number or null.</td>
</tr>
<tr>
<td>cType</td>
<td>string</td>
<td>Type of the standard condition.</td>
</tr>
<tr>
<td>cDesc</td>
<td>string</td>
<td>Description of the standard condition.</td>
</tr>
<tr>
<td>cStatus</td>
<td>string</td>
<td>Condition status. (optional)</td>
</tr>
</tbody>
</table>

**Notes**

If addNum is null, the function adds the condition to all reference addresses associated with the current record.

addAllFees

Adds all fees within a fee schedule to the record. Optionally flags the fees for automatic invoicing by the script.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsched</td>
<td>string</td>
<td>Fee schedule to be added.</td>
</tr>
<tr>
<td>fperiod</td>
<td>string</td>
<td>Fee period to be used.</td>
</tr>
<tr>
<td>qty</td>
<td>integer</td>
<td>Quantity to be entered.</td>
</tr>
<tr>
<td>finvoice</td>
<td>string</td>
<td>Flag for invoicing (&quot;Y&quot; or &quot;N&quot;).</td>
</tr>
</tbody>
</table>
**addAppCondition**

Adds the condition to the record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cType</td>
<td>string</td>
<td>Type of condition (from admin-&gt;condition-&gt;condition type).</td>
</tr>
<tr>
<td>cStatus</td>
<td>string</td>
<td>Status (from admin-&gt;condition-&gt;condition status).</td>
</tr>
<tr>
<td>cDesc</td>
<td>string</td>
<td>Description of the condition.</td>
</tr>
<tr>
<td>cComment</td>
<td>string</td>
<td>Condition comment.</td>
</tr>
<tr>
<td>cImpact</td>
<td>string</td>
<td>Must be Lock, Hold, Notice, Required, or &quot;&quot;.</td>
</tr>
</tbody>
</table>

**addASITable**

Populates the ASI table with values.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>Name of the ASI table to add to the record.</td>
</tr>
<tr>
<td>tableValueArray</td>
<td>array of associative arrays</td>
<td>Values to populate the table.</td>
</tr>
</tbody>
</table>

**Notes**

tableValueArray is an array of arrays. Each array object within tableValueArray must contain an associative index for each column in the target table.

**Example**

```javascript
masterArray = new Array();
elementArray = new Array();
elementArray["Table Column 1"] = "Row 1, column 1 Value";
elementArray["Table Column 2"] = "Row 1, column 2 Value";
```
masterArray.push(elementArray);
addASITable("table name",masterArray);

This example populates the 2-column table with one row.

**addASITable4ACAPageFlow**

Used by page flow scripts to add rows to an ASIT table. You can use this function to dynamically populate an ASIT based on data from earlier pages.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DestinationTableGroupModel</td>
<td>appSpecificTableGroupModel</td>
<td>ASIT object from the current record in ACA.</td>
</tr>
<tr>
<td>tableName</td>
<td>string</td>
<td>Destination table name.</td>
</tr>
<tr>
<td>tableValueArray</td>
<td>associative array</td>
<td>Array of ASI table values to add.</td>
</tr>
</tbody>
</table>

**Example**

The following example adds a row to the TBL-DOCREQ table.

```javascript
var cap = aa.env.getValue("CapModel");
var conditionTable = new Array();
var c = new Array();
c["Document Type"] = new asiTableValObj("Document Type","Document","Y");
c["Name"] = new asiTableValObj("Name","Dangerous / Vicious Dog Waiver","Y");
conditionTable.push(c);
asit = cap.getAppSpecificTableGroupModel();
new_asit = addASITable4ACAPageFlow(asit,"TBL-DOCREQ", conditionTable);
```

**addContactStdCondition**

Adds a standard condition to the specified reference contact. If contactSeqNum is null, the function adds the condition to all reference contacts associated with the current record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.
addCustomFee

Adds a custom fee *feecode* to the record, from the fee schedule *feesched* with fee period *feeperiod*.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contactSeqNum</td>
<td>long</td>
<td>Reference contact sequence number or null.</td>
</tr>
<tr>
<td>cType</td>
<td>string</td>
<td>Type of the standard condition.</td>
</tr>
<tr>
<td>cDesc</td>
<td>string</td>
<td>Description of the standard condition.</td>
</tr>
<tr>
<td>cStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feecode</td>
<td>string</td>
<td>Fee code to be added.</td>
</tr>
<tr>
<td>feesched</td>
<td>string</td>
<td>Fee schedule of the fee to be added.</td>
</tr>
<tr>
<td>feeDescr</td>
<td>string</td>
<td>A description of the custom fee item.</td>
</tr>
<tr>
<td>feePeriod</td>
<td>string</td>
<td>Fee period to be used.</td>
</tr>
<tr>
<td>feeAm</td>
<td>double</td>
<td>Fee quantity.</td>
</tr>
<tr>
<td>feeACC</td>
<td>string</td>
<td>Fee account code 1.</td>
</tr>
<tr>
<td>capID (optional)</td>
<td>CapIDModel</td>
<td>Record to add fee to.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the Fee Sequence number of the fee added.

The fee period *feeperiod* must be a valid fee period for *feecode* in *feesched*, or this function throws an error.

**See also**

addAllFees
addFee

Adds a single fee `fcode` to the record, from the fee schedule `fsched` with fee period `fperiod` and quantity of `fqty`.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcode</td>
<td>string</td>
<td>Fee code to add.</td>
</tr>
<tr>
<td>fsched</td>
<td>string</td>
<td>Fee schedule of the fee to add.</td>
</tr>
<tr>
<td>fperiod</td>
<td>string</td>
<td>Fee period to use.</td>
</tr>
<tr>
<td>qty</td>
<td>integer</td>
<td>Quantity to enter.</td>
</tr>
<tr>
<td>finvoice</td>
<td>string</td>
<td>Flag for invoicing (&quot;Y&quot; or &quot;N&quot;).</td>
</tr>
<tr>
<td>capID (optional)</td>
<td>CapIDModel</td>
<td>Record to add fee to.</td>
</tr>
</tbody>
</table>

**Returns**

The fee period `fperiod` must be a valid fee period for `fcode` in `fsched`, or this function throws an error.

**Notes**

If `finvoice` is Y, the function invoices the fee. If `finvoice` is N, the function assesses the fee but does not invoice the fee.

If you use the `capID` optional parameter, the function updates record `capID`. If you do not use the `capID` parameter, the function updates the current record.

getApplication(), getParent(), createChild() functions each returns a record ID object that you can use in the `capID` parameter.

**See Also**

addAllFees

addFeeWithExtraData

Identical to the addFee function, but also allows you to populate the comment and user defined fields.

**Version**

1.6
Parameters

Parameter | Type     | Description
----------|----------|-------------
fc ode    | string   | Fee code to be added.
fsched    | string   | Fee schedule of the fee to be added.
fperiod   | string   | Fee period to be used.
fq ty     | integer  | Quantity to be entered.
finvoice  | string   | Flag for invoicing ("Y" or "N").
feeCap    | CapIDModel | Record ID object.
feeComment| string   | Comment field on the fee item.
UDF1      | string   | Value for user defined field on fee item.
UDF2      | string   | Value for user defined field on fee item.

cType     | string   | Condition type.
cStatus   | string   | Condition status.
cDesc     | string   | Condition (30 characters maximum).
cComment  | string   | Condition comment (free text).
cImpact   | string   | Condition severity: Lock, Hold, Notice, Required, or "."
stateLicNum  | string   | State license number.

addLicenseCondition

Adds the condition \((cType, cStatus, cDesc, cComment, cImpact)\) to the reference record for each licensed professional on the record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter | Type     | Description
----------|----------|-------------

cType     | string   | Condition type.
cStatus   | string   | Condition status.
cDesc     | string   | Condition (30 characters maximum).
cComment  | string   | Condition comment (free text).
cImpact   | string   | Condition severity: Lock, Hold, Notice, Required, or "."
stateLicNum  | string   | State license number.

Notes

If you use the \(stateLicNum\) parameter, the function adds the condition to the licensed professional reference record whose State License Number is \(stateLicNum\). This licensed professional may not be on the current record.
addLicenseStdCondition

Adds a standard condition to the specified reference licensed professional. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>licSeqNum</td>
<td>long</td>
<td>Reference license sequence number or null.</td>
</tr>
<tr>
<td>cType</td>
<td>string</td>
<td>Type of the standard condition.</td>
</tr>
<tr>
<td>cDesc</td>
<td>string</td>
<td>Description of the standard condition.</td>
</tr>
<tr>
<td>cStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
</tbody>
</table>

**Notes**

If licSeqNum is null, the function adds the condition to all reference licensed professionals associated with the current record.

addLookup

Adds a lookup entry to an existing standard choices item. Adds a new value called `stdValue` with the value description of `stdDesc` to standard choices item name `stdChoice`.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoice</td>
<td>string</td>
<td>Standard choices item name.</td>
</tr>
<tr>
<td>stdValue</td>
<td>string</td>
<td>Standard choices value.</td>
</tr>
<tr>
<td>stdDesc</td>
<td>string</td>
<td>Standard choices value description.</td>
</tr>
</tbody>
</table>

**Notes**

If the standard choices item `stdChoice` already has a value entry called `stdValue`, the function does not add or update `stdValue`. This function does not create the standard choices item `stdChoice` if it does not exist.
addParcelAndOwnerFromRefAddress

Copies the associated parcel and owner from a reference address to the specified record. If you do not specify a record, the function uses the current record as the target.

**Version**
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refAddress</td>
<td>long</td>
<td>Reference address number to copy data from.</td>
</tr>
<tr>
<td>capID (optional)</td>
<td>CapIDModel</td>
<td>Target record for parcel and owner.</td>
</tr>
</tbody>
</table>

addParcelCondition

Adds a condition to the reference parcel whose number is parcel/Num. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

**Version**
2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parcelNum</td>
<td>string</td>
<td>Parcel number to add the condition to. If null, the function adds the condition to all parcels on the record.</td>
</tr>
<tr>
<td>cType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>cStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>cDesc</td>
<td>string</td>
<td>Condition name.</td>
</tr>
<tr>
<td>cComment</td>
<td>string</td>
<td>Condition comment.</td>
</tr>
<tr>
<td>cImpact</td>
<td>string</td>
<td>Condition severity.</td>
</tr>
</tbody>
</table>

**Notes**

The condition’s Type, Condition (description), Status, Severity and Comment corresponds to cType, cDesc, cStatus, cImpact, and cComment, respectively. The condition’s Apply and Effective dates equal the current date. The condition’s Applied By and Action By staff names equal the current user’s name.

If you use null for the parcel/Num parameter, the function adds the condition to all parcels on the current record.
addParcelDistrict

Adds a district to the parcel on a record.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parcelNum</td>
<td>string</td>
<td>Parcel number that district adds to.</td>
</tr>
<tr>
<td>districtValue</td>
<td>string</td>
<td>Value of district entry to add.</td>
</tr>
</tbody>
</table>

Notes

Does not edit reference parcel data.
If parcelNum is null, the function adds the district to all parcels on the current record.

addParent

Adds the current record as a hierarchal child to the parent record parentAppNum.

Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentAppNum</td>
<td>string</td>
<td>App number (B1_ALT_ID) of the record to be parent of the current record.</td>
</tr>
</tbody>
</table>

addrAddCondition

Adds a condition (pType, pStatus, pDesc, pComment, plmpact) to the address on the record whose address number is pAddrNum.

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAddrNum</td>
<td>number</td>
<td>Address number. Use null for all addresses on record.</td>
</tr>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
</tbody>
</table>
Returns

True if the function adds the condition, false otherwise.

Notes

If pAddrNum is null, adds the condition to all the addresses on the record. If pAllowDup is N, the function does not add a condition to the address if the same condition is already on the address. If pAllowDup is Y, the function adds the condition to the address even if this action duplicates the condition on the address.

The function adds the condition to the reference Address record. The function adds the condition only if you use the Search button on the record’s Address screen or use the Get Associated Object button on the record’s parcel screen to add the address to the record. If you enter the address manually, the function does not add the condition.

The pAddrNum value comes from B3ADDRES.L1_ADDRESS_NBR, not B3ADDRES.B1_ADDRESS_NBR.

addReferenceContactByName

Adds a reference contact to the current record, based on the name of the contact. The function only adds the first matching contact.

Version

1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vFirst</td>
<td>string</td>
<td>First name of reference contact.</td>
</tr>
<tr>
<td>vMiddle</td>
<td>string</td>
<td>Middle name of reference contact.</td>
</tr>
<tr>
<td>vLast</td>
<td>string</td>
<td>Last name of reference contact.</td>
</tr>
</tbody>
</table>

addressExistsOnCap

Returns true if there is at least one address on the record.
### addStdCondition

Retrieves all standard conditions named `cDesc` whose type is `cType` and adds them to the record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>capID</code> (optional)</td>
<td>CapiDModel</td>
<td>Record ID to check.</td>
</tr>
</tbody>
</table>

**Notes**

The function assigns the following values to the condition:

- Status = Applied
- Applied By = current user
- Action By = current user
- Apply Date = current date
- Effective Date = current date
- Expiration Date = `blank`

You can only use the function with Accela Automation 6.4 and later.

### addTask

*Dynamically adds a task.*

**Version**

2.0
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceTaskName</td>
<td>string</td>
<td>Name of the task to replicate.</td>
</tr>
<tr>
<td>newTaskName</td>
<td>string</td>
<td>Name of the new task.</td>
</tr>
<tr>
<td>insertTaskType</td>
<td>char</td>
<td>Type of task to add (P for parallel or N for next).</td>
</tr>
<tr>
<td>recordId (optional)</td>
<td>CapIdModel</td>
<td>Record to which to add the task.</td>
</tr>
</tbody>
</table>

Notes

The function uses the source task for all task information such as assignment and statuses. If insertTaskType equals N, the function adds the task to the end of the workflow in series.

addTimeAccountingRecord

Adds a time accounting entry that associates with a record.

Version

2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskUser</td>
<td>string</td>
<td>User ID of the Accela Automation user.</td>
</tr>
<tr>
<td>taGroup</td>
<td>string</td>
<td>Group of the time accounting entry.</td>
</tr>
<tr>
<td>taType</td>
<td>string</td>
<td>Type of the time accounting entry.</td>
</tr>
<tr>
<td>dateLogged</td>
<td>string</td>
<td>Date of the time accounting entry.</td>
</tr>
<tr>
<td>hoursSpent</td>
<td>string</td>
<td>Number of hours for the entry.</td>
</tr>
<tr>
<td>itemCap</td>
<td>CapIdModel</td>
<td>Record to associate to the entry.</td>
</tr>
<tr>
<td>billableBool</td>
<td>boolean</td>
<td>True to set the billable flag, otherwise false.</td>
</tr>
</tbody>
</table>

Example

```javascript
    capID = aa.cap.getCapID("11CAP-00000-0000D").getOutput()
    addTimeAccountingRecord("BSMITH", "Actual", "Inspection", "07/28/2011", "1.1", capID, true);
```

addTimeAccountingRecordToWorkflow

Adds a time accounting entry associated with a workflow task on a record.
**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskUser</td>
<td>string</td>
<td>User ID of the Accela Automation user.</td>
</tr>
<tr>
<td>taGroup</td>
<td>string</td>
<td>Group of the time accounting entry.</td>
</tr>
<tr>
<td>taType</td>
<td>string</td>
<td>Type of the time accounting entry.</td>
</tr>
<tr>
<td>dateLogged</td>
<td>string</td>
<td>Date of the time accounting entry.</td>
</tr>
<tr>
<td>hoursSpent</td>
<td>string</td>
<td>Number of hours for the entry.</td>
</tr>
<tr>
<td>itemCap</td>
<td>CapIDModel</td>
<td>Record to associate to the entry.</td>
</tr>
<tr>
<td>taskName</td>
<td>string</td>
<td>Name of the task to associate with the entry.</td>
</tr>
<tr>
<td>processName</td>
<td>string</td>
<td>Name of the workflow process that contains the task.</td>
</tr>
<tr>
<td>billableBool</td>
<td>boolean</td>
<td>True to set the billable flag, otherwise false.</td>
</tr>
</tbody>
</table>

**Example**

```java
capID = aa.cap.getCapID("11CAP-00000-0000D").getOutput()
addTimeAccountingRecordToWorkflow("BSMITH","Actual","Inspection","07/28/2011","1.1",capID,"Inspection","BLD_MAIN",true);
```

**addToASITable**

Adds one row of values (*tableValues*) to the application specific info (ASI) table called *tableName*.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>The application specific information table name.</td>
</tr>
<tr>
<td>tableValues</td>
<td>array of strings</td>
<td>Values for a single table row, as an associative array of strings.</td>
</tr>
<tr>
<td>capID (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

**Notes**

The *tableValues* parameter must be an associative array of string values, where each element name is a column name in the ASI table *tableName*, and the element stores the column value. If
you use the `capID` parameter, the function adds `tableValues` to `tableName` in the record whose record ID object is `capID`.

The parameter `tableValues` does not have to contain all the columns in the ASI table `tableName`. The ASI table `tableName` must already exist on the record.

### allTasksComplete

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stask</td>
<td>string</td>
<td>Process name of workflow to check.</td>
</tr>
<tr>
<td>igTask1 ... igTaskn</td>
<td>string</td>
<td>Names of tasks to ignore. Enter one or more task name parameters. Case sensitive. (optional)</td>
</tr>
</tbody>
</table>

**Returns**

Returns `true` if all tasks (excluding tasks in optional `igTask1… igTaskn` list) in workflow process / subprocess `stask` are complete. Returns `false` if any task is incomplete.

**Notes**

`stask` is `R1_PROCESS_CODE` in the GPROCESS and SPROCESS tables.

**Examples**

To determine if all tasks in workflow BLDG are completed:

```plaintext
allTasksComplete("BLDG")
```

To determine if all tasks in workflow BLDG are complete, except for the Optional Review task and Closure task:

```plaintext
allTasksComplete("BLDG","Optional Review", "Closure")
```

### appHasCondition

**Version**
1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
</tbody>
</table>
Returns

Returns **true** if the record has a record condition whose type is \textit{pType}, name is \textit{pDesc}, status is \textit{pStatus}, and severity is \textit{plmpact}; otherwise, returns **false**.

Notes

Use **null** in place of any parameter if you do not want to filter by that item. For example, to check if the record has any condition at all, use \texttt{appHasCondition(null, null, null, null)}.

applyPayments

On the current record (capID) this function takes any unapplied payments and distributes them to any invoiced fee items.

Version

2.0

Parameters

None

Notes

The function loops through all fee items and applies the payments until all funds are applied, or no more unpaid fee items remain.

appMatch

Version

1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ats</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Case sensitive. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
<tr>
<td>capID (optional)</td>
<td>CapiDMModel</td>
<td>Record to check.</td>
</tr>
</tbody>
</table>
**Returns**

Returns **true** if `ats` matches the current record’s record type, **false** if it does not.

**Notes**

Compares the current record type to `ats`. You can use the asterisk (*) as a wildcard to match all entries for a given level. For example: `appMatch("Building/*/Sign/*/??")` evaluates to True for record type Building/Commercial/Sign/Billboard as well as Building/Residential/Sign/Garage Sale.

`ats` must contain 3 slash characters (/). Do not add spaces immediate before or after the slash (/).

**appNamesIsUnique**

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gaGroup</code></td>
<td>string</td>
<td>Record group (the 1st level of record type).</td>
</tr>
<tr>
<td><code>gaType</code></td>
<td>string</td>
<td>Record type (the 2nd level of record type).</td>
</tr>
<tr>
<td><code>gaName</code></td>
<td>string</td>
<td>Record name to test.</td>
</tr>
</tbody>
</table>

**Returns**

Returns **true** if none of the other records, whose app type begins with `gaGroup / gaType`, used the record name `gaName`. Returns **false** if `gaName` is not unique.

**assignCap**

Assigns the staff whose user ID is `assignId` to the current record. Also assigns the user’s department.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assignId</code></td>
<td>string</td>
<td>User ID of the user to whom to assign the record.</td>
</tr>
<tr>
<td><code>capID (optional)</code></td>
<td>CapIDModel</td>
<td>Record ID to which to assign the user.</td>
</tr>
</tbody>
</table>
Notes
If you use the optional parameter capID, the function assigns the staff and department to the record capID instead.

assignInspection
Assigns the inspector whose user ID is iName to the inspection whose sequence number is iNumber.

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iNumber</td>
<td>number</td>
<td>Inspection sequence number.</td>
</tr>
<tr>
<td>iName</td>
<td>string</td>
<td>Inspector's user ID.</td>
</tr>
<tr>
<td>capID (optional)</td>
<td>CapIDModel</td>
<td>Record ID to which to assign the inspector.</td>
</tr>
</tbody>
</table>

Notes
The inspection must already be scheduled on the record.

assignTask
Assigns the staff whose user ID is username to workflow task wfstr.

Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task to which to assign a user.</td>
</tr>
<tr>
<td>username</td>
<td>string</td>
<td>User ID of the user to whom to assign the task. Case sensitive.</td>
</tr>
<tr>
<td>wfProcess (optional)</td>
<td>string</td>
<td>Process name of workflow for wfstr. Case sensitive.</td>
</tr>
</tbody>
</table>

Notes
The function does not create a workflow history for the record.
If record’s workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to activate.
wfProcess is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. username and wfProcess are normally in uppercase.

**autoAssignInspection**

Uses the automatic inspection assignment function to assign the specified inspection.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iNumber</td>
<td>long</td>
<td>Sequence number for the inspection to assign.</td>
</tr>
</tbody>
</table>

**branch**

Executes the standard choice script control whose name is *iNumber* as a sub-control.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoice</td>
<td>string</td>
<td>Standard choices item name string. Case sensitive.</td>
</tr>
</tbody>
</table>

**Notes**

The script *stdChoice* must contain only valid criteria/action pairs sequentially numbered.

**Example**

```
branch("Inspection:Update Expiration")
```

**branchTask**

Updates the workflow task *wfstr* as follows

- Status = *wfstat*
- Status Date = current date
- Status Comment = *wfcomment*
- Action By = current user

**Version**

1.3
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td>wfstat</td>
<td>string</td>
<td>Status.</td>
</tr>
<tr>
<td>wfcomment</td>
<td>string</td>
<td>Comment.</td>
</tr>
<tr>
<td>wfnote</td>
<td>string</td>
<td>Note to add to the workflow task.</td>
</tr>
<tr>
<td>wfProcess</td>
<td>string</td>
<td>ID (R1_PROCESS_CODE) for the process that the task belongs to. Required for multi-level workflows.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes

The function closes the task wfstr and the workflow proceeds to the branch task.

If record’s workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to edit.

---

### capHasExpiredLicProf

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pDateType</td>
<td>string</td>
<td>Expiration date to check. Options (use one): EXPIRE, INSURANCE, BUSINESS.</td>
</tr>
<tr>
<td>pLicType</td>
<td>string</td>
<td>License type.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pCapId</td>
<td>CapIDModel</td>
<td>Record ID object of record. If null, the function applies to the current record.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Returns**

*Returns true if any licensed professional on the record has expired; otherwise, returns false.*

**Notes**

Checks for expiration by retrieving the licensed professional reference record having the same license # and checking the expiration date specified by pDateType. If the expiration date is on or before the current date, the script returns true. Skips disabled licensed professionals.

Use parameter pLicType to check a specific license type. Use parameter pCapId to check licensed professionals on a record other than the current record.
capIdsFilterByFileDate

Searches though the records in `pCapIdArray` and returns only records whose file date is between `pStartDate` and `pEndDate`, as an array of capId (CapIDModel) objects.

**Version**
1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCapIdArray</td>
<td>array of CapIDModel objects</td>
<td>Array of record ID (CapIDModel) objects to filter.</td>
</tr>
<tr>
<td>pStartDate</td>
<td>string</td>
<td>Start date of the file date range, in MM/DD/YYYY format.</td>
</tr>
<tr>
<td>pEndDate</td>
<td>string</td>
<td>End date of the file date range, in MM/DD/YYYY format.</td>
</tr>
</tbody>
</table>

**Notes**
To find the number of records returned, store the return value to a variable and use the length property to find the number of records in the array.

**Example**
```
capArray = capIdsFilterByFileDate(myCapArray, "01/01/2006", "12/31/2006"); capCount = capArray.length;
```

capIdsGetByAddr

Returns records that have the same property address as the current record, as an array of capId (CapIDModel) objects.

**Version**
1.4

**Parameters**

None
**Returns**

*If the current record has no property address, returns false.*

**Notes**

The function matches addresses based on these fields:

- House Nbr Start
- Street Direction
- Street Name
- Street Suffix
- Zip

You can use this function with all events except ApplicationSubmitBefore. The records returned include the current record. If the current record has more than one property address, the function uses the first address to match.

To find the number of records returned, store the return value to a variable and use the length property to find the number of records in the array.

**Example**

```javascript
capArray = capIdsGetByAddr(); logDebug("Number of CAPs: ", + capArray.length);
```

capIdsGetByParcel

Returns records that have the same parcel as the current record, as an array of capId (CapIDModel) objects.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pParcelNum</td>
<td>string</td>
<td>Parcel number to search for. If <strong>null</strong> or omitted, the function uses the first parcel number on the current record.</td>
</tr>
</tbody>
</table>

**Returns**

*If the current record has no parcel, returns false.*

**Notes**

The records returned include the current record.

To find the number of records returned, store the return value to a variable and use the length property to find the number of records in the array.
Example

capArray = capIdsGetByParcel(); logDebug("Number of CAPs: " + capArray.length);

capSet

Version

2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desiredSetId</td>
<td>string</td>
<td>The ID of the set to create or operate on by the capSet object.</td>
</tr>
</tbody>
</table>

Notes

capSet is a helper object that assists in managing Accela Automation Sets of records. If the desiredSetId already exists as a Set, it loads automatically. If the desiredSetId does not exist, function creates it as an empty set.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refresh()</td>
<td>The capSet object reloads and all properties refresh.</td>
</tr>
<tr>
<td>add(capId)</td>
<td>Adds the supplied capId to the set.</td>
</tr>
<tr>
<td>remove(capId)</td>
<td>Removes the supplied capId from the set.</td>
</tr>
<tr>
<td>update()</td>
<td>The header information about the set updates to the current values. This header information includes the set name and set comment.</td>
</tr>
</tbody>
</table>

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The Id of the set.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the set.</td>
</tr>
<tr>
<td>comment</td>
<td>The set comment.</td>
</tr>
<tr>
<td>size</td>
<td>The number of records in the set.</td>
</tr>
<tr>
<td>empty</td>
<td>True if the set has no members.</td>
</tr>
<tr>
<td>members</td>
<td>An array or CapIDModel objects representing the membership of the set.</td>
</tr>
</tbody>
</table>

checkCapForLicensedProfessionalType

Returns true if a licensed professional of the type exists on the current record.

Version

1.6
Parameters

Parameter | Type    | Description
----------|---------|-------------
llicProfType | string | Licensed professional type to check for.

checkInspectionResult

Version
1.3

Parameters

Parameter | Type | Description
----------|------|-------------
insp2Check | string | Inspection to check. Case sensitive.
insp2Result | string | Inspection result (or status) to look for. Case sensitive.

Returns

Returns true if the inspection insp2Check has the result of insp2Result, or false if it does not.

Notes

You can use Scheduled as the value for the insp2Result in parameter to check if inspection insp2Check is scheduled (not yet resulted).

childGetByCapType

Searches through all child records and returns the record ID object for the first child record whose record type matches pCapType.

Version
1.4

Parameters

Parameter | Type | Description
----------|------|-------------
pCapType | string | Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.

pParentCapId | (optional) CapiIDModel | Record ID object for parent record. Use null if you use the skipChildCapId parameter.

skipChildCapId | (optional) CapiIDModel | Record ID object of child record to skip.
Notes
If you use the $pParentCapId$ parameter, the function searches child records of the record whose record ID object is $pParentCapId$. If you use the $skipChildCapId$ parameter, the function skips over any child record whose record ID object is $skipChildCapId$.

To find the sibling of the current record, use the function `getParent()` as the parentCapId parameter and capId as the skipChildCapId parameter.

Example
```plaintext
siblingCapId = childGetByCapType("*//*//*", getParent(), capId)
```

See also
gETCHILDREN

closeCap

Version
2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userId</td>
<td>string</td>
<td>ID of user who closes the record.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record to perform action on.</td>
</tr>
</tbody>
</table>

Notes
Sets the Closed Date value to the current date and the Close by Staff field to the ID of the user who closes the record.

closeSubWorkflow

A function that is useful when working with sub-processes.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisProcessID</td>
<td>long value</td>
<td>ID of the process to check.</td>
</tr>
<tr>
<td>wfStat</td>
<td>string</td>
<td>Status to use when closing the parent task.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record to perform action on.</td>
</tr>
</tbody>
</table>
**Notes**

Checks all the tasks in the subprocess for completeness. If all tasks are complete, the function closes the parent task with the specified status.

**Example**

```
closeSubWorkflow(wfProcessID,"Completed");
```

**closeTask**

Updates the workflow task *wfstr* as follows:

- Status = *wfstat*
- Status Date = current date
- Status Comment = *wfcomment*
- Action By = current user

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>wfstr</em></td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td><em>wfstat</em></td>
<td>string</td>
<td>Status to update.</td>
</tr>
<tr>
<td><em>wfcomment</em></td>
<td>string</td>
<td>Comment to add.</td>
</tr>
<tr>
<td><em>wfnote</em></td>
<td>string</td>
<td>Note to add to the workflow task.</td>
</tr>
<tr>
<td><em>wfProcess</em> (optional)</td>
<td>string</td>
<td>ID (R1_PROCESS_CODE) for the process that the task belongs to. Required for multi-level workflows.</td>
</tr>
</tbody>
</table>

**Notes**

Closes the task *wfstr* and promotes the workflow to the next task, even if *wfstat* loops or branches. If workflow needs to loop or branch, use loopTask or branchTask functions.

If record’s workflow contains duplicate *wfstr* tasks, use *wfProcess* parameter to specify the process or subprocess whose *wfstr* to edit.

This old name for this function is closeWorkflow^2.

**comment**

You can use this function to display messages to the user, as well as variables to aid in debugging issues.
Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstr</td>
<td>string</td>
<td>Comment to display.</td>
</tr>
</tbody>
</table>

Notes
Use logMessage and logDebug functions instead.
Adds the message cstr to the message/debug window when the script executes. If you enable debugging (i.e., showDebug = true), the comment shows in the debug messages. If you enable messages (i.e., showMessage = true), the comment shows in the messages. If you do not enable debugging or messages, the comment does not display.

Use this function instead of directly assigning value to message variable in script control.

Example

```
true ^ comment("calcValue is " + calcValue)
true ^ comment("The building fees have been added automatically")
```

comparePeopleGeneric

This function passes as a parameter to the createRefContactsFromCapContactsAndLink function.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>peop</td>
<td>peopleModel</td>
<td>The peopleModel object containing the criteria.</td>
</tr>
</tbody>
</table>

Returns
Takes a single peopleModel as a parameter, and returns the sequence number of the first G6Contact result. Returns null if there are no matches.

Notes
To use attributes, you must implement Salesforce case 09ACC-05048.
completeCAP

**Version**
1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userId</td>
<td>string</td>
<td>ID of user that completes the record.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record with which to perform the action.</td>
</tr>
</tbody>
</table>

**Notes**
Assigns the staff whose user ID is `userId` to the Completed by Staff field on a record. Also sets the Completed by Date value to the current date.

If you use the capId optional parameter, the function updates record `capId`. If you do not use the `capId` parameter, the function updates the current record.

contactAddFromUser

**Version**
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pUserId</td>
<td>string</td>
<td>User ID used as criteria to search for contact.</td>
</tr>
</tbody>
</table>

**Notes**
Searches for a reference contact that matches the supplied `userId`, based on first, middle, and last names. If the function finds a matching contact, the function adds the record contact to the current record.

contactSetPrimary

Sets the supplied contact to be the primary contact on the current record

**Version**
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pContactNbr</td>
<td>long</td>
<td>Sequence number of the contact to make primary.</td>
</tr>
</tbody>
</table>
**contactSetRelation**

Sets the relationship code on the supplied contact, on the current record.

**Version**
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pContactNbr</td>
<td>long</td>
<td>Sequence number of the contact.</td>
</tr>
<tr>
<td>pRelation</td>
<td>string</td>
<td>Set to this relationship code.</td>
</tr>
</tbody>
</table>

**convertDate**

Converts a scriptDateTime date to a javascript date.

**Version**
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisDate</td>
<td>scriptDateTime</td>
<td>The date to convert.</td>
</tr>
</tbody>
</table>

**convertStringToPhone**

Converts the string to phone codes (A=1, D=3, etc), useful with the setIVR function.

**Version**
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>theString</td>
<td>string</td>
<td>String containing information to convert.</td>
</tr>
</tbody>
</table>

**copyAddresses**

Copies all property addresses from record pFromCapId to record pToCapId. If record pToCapId has a primary address, any primary address in pFromCapId becomes non-primary when copied over.

**Version**
1.4
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>pToCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy. If null, the function uses the current record.</td>
</tr>
</tbody>
</table>

**Notes**

getApplication( ), getParent( ), createChild(), createCap() functions each returns a record ID object.

**copyAppSpecific**

Copies all app spec info values from current record to the record whose record ID object is `newCap`. If the target record does not have the same app specific info field, the does not copy the value.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newCap</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>ignoreArr</td>
<td>string array</td>
<td>Array of ASI labels not to ignore and not copy.</td>
</tr>
</tbody>
</table>

**copyASIFields**

Copies all ASI fields from the `sourceCapId` record to the `targetCapId` record with the exception of the ASI subgroups listed in `ignore_1` to `ignore_n`.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>targetCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy.</td>
</tr>
<tr>
<td>ignore_1 to ignore_n</td>
<td>string</td>
<td>ASI subgroups to ignore during the copy.</td>
</tr>
</tbody>
</table>
Notes
This function moves the ASI fields themselves, not the values. You can add an ASI group to a record that did not previously include the ASI group. This function does not copy the form portlet designer settings, which can cause problems.

copyASITables
Copies ASI Tables from one Record to another. This function depends on the addASITable function.

Version
2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>pToCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy.</td>
</tr>
<tr>
<td>ignoreArr</td>
<td>string array</td>
<td>Array of table names to ignore and not copy.</td>
</tr>
</tbody>
</table>

copyCalcVal
Copies the calculated job value from the current record to the record whose record ID object is pToCapId.

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromcap</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>newcap</td>
<td>CapIDModel</td>
<td>ID of record to which to copy.</td>
</tr>
</tbody>
</table>

copyConditions
Copies all conditions from record capId to the current record (if you do not specify toCapId) or the specified record.

Version
1.3
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>toCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parcelIdString</td>
<td>string</td>
<td>Parcel number of source parcel.</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
true ^ subdivapp =
getApplication(lookup("SubdivisionXref",{SubDiv}));
copyConditions(subdivapp)
```

**copyConditionsFromParcel**

Copies conditions from the reference parcel `parcelIdString` and adds them as conditions to the current record (not to parcels on the current record).

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parcelIdString</td>
<td>string</td>
<td>Parcel number of source parcel.</td>
</tr>
</tbody>
</table>

**copyContacts**

Copies all contacts from record `pFromCapId` to record `pToCapId`.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>pToCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy. If null, the function uses the current record.</td>
</tr>
</tbody>
</table>

**Notes**

If target record has a primary contact and the source record also has a primary contact, the target record ends up with 2 primary contacts.

getApplication(), getParent(), createChild(), createCap() functions each return a Cap ID object.
**copyContactsByType**

Copies only contacts of the specified type from record *pFromCapId* to record *pToCapId*.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>pToCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy. If null, the function uses the current record.</td>
</tr>
<tr>
<td>pContactType</td>
<td>string</td>
<td>Contact type to copy.</td>
</tr>
</tbody>
</table>

**Notes**

If target record has a primary contact and the source record also has a primary contact, the target record ends up with 2 primary contacts.

getApplication( ), getParent( ), createChild(), createCap() functions each return a Cap ID object.

**copyFees**

Copies all fees from record *sourceCapId* to record *targetCapId*. Excludes voided or credited fees.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy fees.</td>
</tr>
<tr>
<td>targetCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy.</td>
</tr>
</tbody>
</table>

**copyLicensedProf**

Copies all licensed professionals from *sCapId* to record *tCapId*.

**Version**

1.6
### copyOwner

Copies a contact from `sCapID` to `tCapID`.

#### Version

1.6

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sCapID</td>
<td>CapIDModel</td>
<td>ID of record from which to copy licensed professionals.</td>
</tr>
<tr>
<td>tCapID</td>
<td>CapIDModel</td>
<td>ID of record to which to copy.</td>
</tr>
</tbody>
</table>

### copyOwnersByParcel

Copies reference owners from all attached parcels to the current record.

#### Version

2.0

#### Parameters

None

### copyParcelGisObjects

Copies parcel GIS objects to the record.

#### Version

1.3

#### Parameters

None

### copyParcels

Copies all parcels, and parcel attributes, from record `pFromCapId` to record `pToCapId`.
**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>pToCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy. If null, the function uses the current record.</td>
</tr>
</tbody>
</table>

**Notes**

capId is the record ID object for the current record.
getApplication( ), getParent( ), createChild(), createCap() functions each return a record ID object.

**copySchedInspections**

Copies all scheduled inspections from record pFromCapId to record pToCapId.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFromCapId</td>
<td>CapIDModel</td>
<td>ID of record from which to copy.</td>
</tr>
<tr>
<td>pToCapId</td>
<td>CapIDModel</td>
<td>ID of record to which to copy. If null, the function uses the current record.</td>
</tr>
</tbody>
</table>

**Notes**

Includes inspections that have a pending-type result, but copies status over as Scheduled. You do not need to copy the inspection type to the target record. The function can copy duplicate inspections to the target record.
capId is the record ID object for the current record.
getApplication( ), getParent( ), createChild(), createCap() functions each return a record ID object.

**countActiveTasks**

Returns the number of active tasks in the workflow whose process name is processName.

**Version**

1.4
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>processName</td>
<td>string</td>
<td>Process name of workflow.</td>
</tr>
</tbody>
</table>

### countIdenticalInspections

Returns the number of inspections that have the same inspection description and status (or result) as the inspection in the current event.

**Version**

1.4

**Parameters**

None

**Notes**

Use this function only with the following events:

- InspectionResultSubmitAfter
- InspectionScheduleAfter
- InspectionScheduleBefore

### createAddresses

Adds an address to the record.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetCapID</td>
<td>CapIDModel</td>
<td>Record ID object.</td>
</tr>
<tr>
<td>addressModel</td>
<td>AddressModel</td>
<td>Address.</td>
</tr>
</tbody>
</table>

### createCap

Creates a record of type pCapType with the record name of pAppName.

**Version**

1.4
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCapType</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes.</td>
</tr>
<tr>
<td>pAppName</td>
<td>string</td>
<td>Record name.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the new record’s record ID object.

**createCapComment**

Creates a record comment for the specified record

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vComment</td>
<td>string</td>
<td>Comment to add.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record for which to create a comment.</td>
</tr>
</tbody>
</table>

**createChild**

Creates a record of type grp/typ/stype/cat with the record name, and links it as a child to the current record’s hierarchy.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grp</td>
<td>string</td>
<td>App Group. Top classification of the record.</td>
</tr>
<tr>
<td>typ</td>
<td>string</td>
<td>App Type. Second classification of the record.</td>
</tr>
<tr>
<td>stype</td>
<td>string</td>
<td>App SubType: 3rd Classification of the record.</td>
</tr>
<tr>
<td>cat</td>
<td>string</td>
<td>App Category: 4th Classification of the record.</td>
</tr>
<tr>
<td>desc</td>
<td>string</td>
<td>Record name.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record to be the parent of new record.</td>
</tr>
</tbody>
</table>
**Returns**
The new child record’s ID.

**Notes**
The function copies the following data from the current record to the new child record.
- parcels
- contacts
- property addresses

**createParent**
Creates a record of type `grp/typ/stype/cat` with the record name, and links it as a parent to the current record’s hierarchy.

**Version**
2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grp</td>
<td>string</td>
<td>App Group. Top classification of the record.</td>
</tr>
<tr>
<td>typ</td>
<td>string</td>
<td>App Type. Second classification of the record.</td>
</tr>
<tr>
<td>stype</td>
<td>string</td>
<td>App SubType: 3rd Classification of the record.</td>
</tr>
<tr>
<td>cat</td>
<td>string</td>
<td>App Category: 4th Classification of the record.</td>
</tr>
<tr>
<td>desc</td>
<td>string</td>
<td>Record name.</td>
</tr>
</tbody>
</table>

**Returns**
The new parent record’s record ID object, to be used in other functions.

**Notes**
The following data are copied from the current record to the new parent record.
- parcels
- contacts
- property addresses

**createPendingInspection**
Creates a pending inspection of the specified group and type on the specified record.

**Version**
2.0
createPendingInspFromReqd

Creates a pending inspection for all inspections that are configured as required in the inspection group associated to the record type.

Version
2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iGroup</td>
<td>string</td>
<td>Inspection group of the inspection to create.</td>
</tr>
<tr>
<td>iTYPE</td>
<td>string</td>
<td>Inspection type of the inspection to create.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record on which to create the inspection.</td>
</tr>
</tbody>
</table>

Notes
Uses the current record (capId global variable) if no capId parameter supplied.

createPublicUserFromContact

Creates a public user account (Accela Citizen Access) with information based on the contact.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contactType</td>
<td>string</td>
<td>The public user is based on this contact type, default is Applicant.</td>
</tr>
</tbody>
</table>

Notes
Useful for automatically creating an online account for applicants that apply in the office.

- Creates the public user record
- Assigns to current agency
- Activates for the current agency
- Issues a password reset to their email address
- Sends activation email

createRefContactsFromCapContactsAndLink

This function can be used as the basis for maintaining a contact-centric database within Accela Automation.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCapId</td>
<td>CapIdModel</td>
<td>Record to work with.</td>
</tr>
<tr>
<td>contactTypeArray</td>
<td>array</td>
<td>The contact types to process, or null for all. This parameter is ignored if the REF_CONTACT_ENFORCE_TYPE_FLAG_WITH_EMSE standard choice is configured. See description for more detail.</td>
</tr>
<tr>
<td>ignoreAttributeArray</td>
<td>array</td>
<td>An array of attributes to ignore when creating a REF contact, or null.</td>
</tr>
<tr>
<td>replaceCapContact</td>
<td>boolean</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>overwriteRefContact</td>
<td>boolean</td>
<td>If true, refreshes the linked ref contact with record contact data.</td>
</tr>
<tr>
<td>refContactExists</td>
<td>function</td>
<td>Function used to determine if the reference contact exists.</td>
</tr>
</tbody>
</table>

**Example**

```javascript
iArr = new Array();
iArr.push("Partner Percent")
createRefContactsFromCapContactsAndLink(capId,null,iArr,false,true,comparePeopleGeneric);
```

In this example, when this code is executed, the function loops through all contacts on the current record. If the contact was hand-entered (not selected and validated from reference contacts) the reference contacts searches for a match using the comparePeopleGeneric function. If a match is found, the record contact links to the reference contact. Also, the reference contact refreshes with data from the cap contact. All attributes refresh except for the “Partner Percent” field.

**Version 2.0 Update:** This function now checks for the presence of a standard choice “REF_CONTACT_CREATION_RULES”. See screenshot below for configuration.
This setting determines whether to create the reference contact, as well as the contact type with which to create the reference contact. If this setting is configured, the function ignores the contactTypeArray parameter. The “Default” in this standard choice determines the default action of all contact types. Other types can be configured separately. Each contact type can be set to “I” (create ref as individual), “O” (create ref as organization), “F” (follow the indiv/org flag on the cap contact), “D” (Do not create a ref contact), or “U” (create ref using the transactional contact type).

createRefLicProf

Creates a new reference Licensed Professional from the Contact on the current record whose contact type is pContactType.

Version

1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlpId</td>
<td>string</td>
<td>State license number.</td>
</tr>
<tr>
<td>rlpType</td>
<td>string</td>
<td>License type.</td>
</tr>
<tr>
<td>pContactType</td>
<td>string</td>
<td>Contact type. (optional)</td>
</tr>
</tbody>
</table>

Notes

The Licensed Professional has the state license # of rlpId and license type of rlpType. If a reference Licensed Professional with state license # rlpId already exists, it updates with data from the Contact.

Contact’s State field must be populated for the Licensed Prof to be created.

The function does not copy the Contact’s middle name and address line 3 to the Licensed Prof.
If available, the following app specific info fields copy to the Licensed Prof (field labels must match exactly):

- Insurance Co
- Insurance Amount
- Insurance Exp Date
- Policy #
- Business License #
- Business License Exp Date

**createRefLicProfFromLicProf**

Retrieves the first licensed professional on the record and creates a reference licensed professional record. If a reference record already exists for this licensed professional, updates the reference licensed record with the licensed professional's data from the record.

**Version**

1.4

**Parameters**

None

**dateAdd**

Returns date that results from adding *amt* days to *td*, as a string in "MM/DD/YYYY" format.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>td</em></td>
<td>string</td>
<td>Starting date, in format &quot;MM/DD/YYYY&quot; (or any string that converts to JS date). If null is used, <em>td</em> is the current date.</td>
</tr>
<tr>
<td><em>amt</em></td>
<td>integer</td>
<td>Number of days to add to <em>td</em>. Use negative number (e.g. –20) to subtract days from <em>td</em>.</td>
</tr>
<tr>
<td><em>workDays</em></td>
<td>string</td>
<td>'Y' if <em>amt</em> workdays should be added to <em>td</em>. Omit if <em>amt</em> calendar days should be added to <em>td</em>.</td>
</tr>
</tbody>
</table>

**Notes**

Does not work if date is wfDate. Returns NaN/NaN/NaN.
dateAddMonths

Returns date that results from adding \( p\text{Months} \) months to \( p\text{Date} \), as a string in “MM/DD/YYYY” format.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pDate</td>
<td>string</td>
<td>Starting date, in format “MM/DD/YYYY” (or any string that converts to JS date). If null is used, ( td ) is the current date.</td>
</tr>
<tr>
<td>pMonths</td>
<td>integer</td>
<td>Number of months to add to ( p\text{Date} ). Use negative number (e.g. -12) to subtract months from ( td ).</td>
</tr>
</tbody>
</table>

**Notes**

If \( p\text{Date} \) is the last day of the month, the returned date is the last day of the month. If \( p\text{Date} \) is not the last day of the month, the new date has the same day of month, unless such a day doesn't exist in the new month (e.g. if baseDate is 1/30/2007 and the returned month is February), in which case the new date is the last day of the month.

Does not work if baseDate is wDate. Returns NaN/NaN/NaN.

dateFormatted

Returns formatted date in YYYY-MM-DD or MM/DD/YYYY format (default).

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMonth</td>
<td>string</td>
<td>Month of new date, as 2-digit month.</td>
</tr>
<tr>
<td>pDay</td>
<td>string</td>
<td>Day of new date, as 2-digit day.</td>
</tr>
<tr>
<td>pYear</td>
<td>string</td>
<td>Year of new date as 4-digit year.</td>
</tr>
<tr>
<td>pFormat</td>
<td>string</td>
<td>Format to produce string in.</td>
</tr>
</tbody>
</table>

dateNextOccur

Returns the next occurrence of \( p\text{Month} \) and \( day \) after \( p\text{Date} \). If \( oddEven \) is “odd”, gets the next occurrence of \( p\text{Month} \) and \( day \) after \( p\text{Date} \) in an odd year (for example, year is an odd number). If \( oddEven \) is “even”, gets the next occurrence of \( p\text{Month} \) and \( day \) after \( p\text{Date} \) in an even year.
Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pMonth</td>
<td>string</td>
<td>Month of new date, as 2-digit month.</td>
</tr>
<tr>
<td>pDay</td>
<td>string</td>
<td>Day of new date, as 2-digit day.</td>
</tr>
<tr>
<td>pDate</td>
<td>string</td>
<td>Date from which new date is determined. In format MM/DD/YYYY or YYYY-MM-YY as used by wfDate variable.</td>
</tr>
<tr>
<td>oddEven</td>
<td>string</td>
<td>Specifies if the new date should be in an odd or even year. Enter “odd” or “even”.</td>
</tr>
</tbody>
</table>

Notes
The pDate parameter can be a date string in MM/DD/YYYY format, or an event-specific variable (e.g. wfDate) whose date format is YYYY-MM-DD.

deactivateTask
Deactivates the task, similar to setting Active? = N in the workflow supervisor portlet

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task to be deactivated.</td>
</tr>
<tr>
<td>wfProcess</td>
<td>string</td>
<td>Process name of workflow task wfstr.</td>
</tr>
</tbody>
</table>

deleteTask
Permanently removes the named task from the workflow.

Version
1.6
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetCapId</td>
<td>CapIDModel</td>
<td>Record to affect.</td>
</tr>
<tr>
<td>deleteTaskName</td>
<td>string</td>
<td>Name of task to delete.</td>
</tr>
</tbody>
</table>

**editAppName**

Updates record name to *newName*.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newName</td>
<td>string</td>
<td>New record name.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

**Returns**

*Returns true if successful or false if update fails.*

**editAppSpecific**

Updates the value of the app specific info field *itemName* with the value *itemValue*. Also updates the internal list of values, so that future criteria/action pairs do not see the correct value. If no *capId* is supplied, then the current record is used.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemName</td>
<td>string</td>
<td>App Specific Info field to edit.</td>
</tr>
<tr>
<td>itemValue</td>
<td>string</td>
<td>Value that the app spec info field <em>itemName</em> should be changed to.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record whose app spec info field <em>itemName</em> is to be changed to <em>itemValue</em>.</td>
</tr>
</tbody>
</table>
editBuildingCount

Edits the building count on the record detail.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numBuild</td>
<td>string</td>
<td>New number of buildings.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>The capId to affect.</td>
</tr>
</tbody>
</table>

duginCapContactAttribute

Changes the value of a record contact attribute.

Version
2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contactSeq</td>
<td>long</td>
<td>Sequence number of the record contact to edit.</td>
</tr>
<tr>
<td>pAttributeName</td>
<td>string</td>
<td>Label of the attribute to edit.</td>
</tr>
<tr>
<td>pNewAttributeValue</td>
<td>string</td>
<td>New value of the attribute.</td>
</tr>
<tr>
<td>itemCapId (optional)</td>
<td>CapIDModel</td>
<td>Record on which the record contact belongs.</td>
</tr>
</tbody>
</table>

Notes

The attribute name must be in ALL CAPS.

Example

```
ediganCapContactAttribute(60549773,"HAIR COLOR","Yellow",thisCapId);
```

dChanenChannelReported

Changes the channel reported value to value passed to function.

Version
2.0
editChannelReported

Example

```
editChannelReported("PHONE", capId);
```

**editContactType**

Updates Contact Type for all contacts on a record to newtype when the existing Contact Type is equal to the existingType.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>existingType</td>
<td>string</td>
<td>Existing contact type.</td>
</tr>
<tr>
<td>newType</td>
<td>string</td>
<td>New contact type.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object.</td>
</tr>
</tbody>
</table>

**Notes**

getApplication( ), getParent( ), createChild( ) functions each returns a record ID object that can be used in the capId parameter

editHouseCount

Updates the record's house count field to numHouse.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numHouse</td>
<td>string</td>
<td>New house count.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>
**Returns**

Returns **true** if successful or **false** if update fails.

**editInspectionRequiredFlag**

Sets the inspection milestone flag ‘Inspection Required” to Y or N.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspType</td>
<td>string</td>
<td>Inspection type to edit.</td>
</tr>
<tr>
<td>reqFlag</td>
<td>boolean</td>
<td>If true, sets the required flag to “Y”, otherwise “N”.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Target record ID.</td>
</tr>
</tbody>
</table>

**editLookup**

Attempts to find existing standard choices value called `stdValue` in the standard choices item called `stdChoice`. If found, updates the existing Value Description for `stdValue`. If `stdValue` is not found, adds the new value `stdValue` with the Value Desc of `stdDesc`.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoice</td>
<td>string</td>
<td>Name of standard choice.</td>
</tr>
<tr>
<td>stdValue</td>
<td>string</td>
<td>Name of standard choice value.</td>
</tr>
<tr>
<td>stdDesc</td>
<td>string</td>
<td>New standard choice description.</td>
</tr>
</tbody>
</table>

**editPriority**

Updates the record's Priority field to `priority`.

**Version**

1.5
editRefLicProfAttribute

Updates the attribute (template data) on a reference licensed professional record.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pLicNum</td>
<td>string</td>
<td>License number of reference LP.</td>
</tr>
<tr>
<td>pAttributeName</td>
<td>string</td>
<td>Label of the attribute to update.</td>
</tr>
<tr>
<td>pNewAttributeValue</td>
<td>string</td>
<td>New attribute value.</td>
</tr>
</tbody>
</table>

Returns

Returns true if successful or false if update fails.

editReportedChannel

Updates the record's Reported Channel field to reportedChannel.

Version
1.5

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reportedChannel</td>
<td>string</td>
<td>New reported channel value.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

Returns

Returns true if successful or false if update fails.
**editScheduledDate**

Edits the schedule date in record detail on the selected record.

*Version*

1.6

*Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduledDate</td>
<td>string</td>
<td>New schedule date value.</td>
</tr>
<tr>
<td>[capId]</td>
<td>CapIDModel</td>
<td>Record ID to modify.</td>
</tr>
</tbody>
</table>

**editTaskComment**

Adds the status comment `wfcomment` to workflow task `wfstr`. If `wfstr` has an existing comment, the comment is replaced by `wfcomment`. `wfstr` does not have to be active. Status date is not updated. No workflow history record is created.

*Version*

1.3

*Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task whose comment should be updated.</td>
</tr>
<tr>
<td>wfcomment</td>
<td>string</td>
<td>Comment to be given to <code>wfstr</code>.</td>
</tr>
<tr>
<td>wfProcess</td>
<td>string</td>
<td>Process name of workflow task <code>wfstr</code>.</td>
</tr>
</tbody>
</table>

*Notes*

If record’s workflow contains duplicate `wfstr` tasks, use parameter `wfProcess` to specify the process or subprocess whose `wfstr` should be edited.

**editTaskDueDate**

Sets the due date of the workflow task `wfstr` to `wfdate`. If `wfstr` is `**`, sets due dates on all workflow tasks on the record. No workflow history record is created.

*Version*

1.3
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task.</td>
</tr>
<tr>
<td>wfname</td>
<td>string</td>
<td>Due date to be given to wfstr.</td>
</tr>
<tr>
<td>wfProcess</td>
<td>string</td>
<td>Process name of workflow task wfstr.</td>
</tr>
</tbody>
</table>

Notes

If record’s workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr should be edited.

editTaskSpecific

Updates the value of the task specific info field itemName for workflow task wfname to the value itemValue. Also updates the internal list of values, so that future criteria/action pairs see the correct value. If capId is supplied, updates the specified task specific info field on the record whose record ID object is capId.

Version

1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfname</td>
<td>string</td>
<td>Workflow task.</td>
</tr>
<tr>
<td>itemName</td>
<td>string</td>
<td>Task Specific Info field to edit.</td>
</tr>
<tr>
<td>itemValue</td>
<td>string</td>
<td>Value that the task spec info field itemName should be changed to.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record whose task spec info field itemName is to be changed to itemValue.</td>
</tr>
</tbody>
</table>

email

Sends an email to the email address pToEmail from the email address pFromEmail. The email’s subject line is pSubject and its content is pText.

Version

1.4
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pToEmail</td>
<td>string</td>
<td>Email address of recipient.</td>
</tr>
<tr>
<td>pFromEmail</td>
<td>string</td>
<td>Email address of sender.</td>
</tr>
<tr>
<td>pSubject</td>
<td>string</td>
<td>Text that appears in subject line of email.</td>
</tr>
<tr>
<td>pText</td>
<td>string</td>
<td>Text that appears in body of email.</td>
</tr>
</tbody>
</table>

emailContact

Sends an email to the contact on the current record whose Contact Type is contactType. Uses the email address in the contact screen. Default contact is “Applicant”.

Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mSubj</td>
<td>string</td>
<td>Text that appears in subject line of email.</td>
</tr>
<tr>
<td>mText</td>
<td>string</td>
<td>Text that appears in body of email.</td>
</tr>
<tr>
<td>contactType</td>
<td>string</td>
<td>Contact Type that email is sent to. Default is “Applicant”. (optional)</td>
</tr>
</tbody>
</table>

Example

```
inspResult.equals("Passed") ^ emailContact("Inspection Results", "Your inspection " + inspType + " has passed.", "Contractor")
```

dendBranch

Immediately stops execution of the branch (standard choice) that is currently executing. Script controls continue executing from the calling standard choice, if any.

Version
1.6

Parameters

None

Example

```
01 true ^ endBranch()
02 true ^ comment("this will not execute")
```
executeASITable

Executes an ASI table as if it were script commands. No capability for else or continuation statements. Assumes that there are at least three columns named "Enabled", "Criteria", and "Action". Replaces token in the controls.

Version
1.5

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableArray</td>
<td>array</td>
<td>Application specific info table array.</td>
</tr>
</tbody>
</table>

exists

Searches the array eArray for the value eVal. Returns true if the value is found in the array.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eVal</td>
<td>string</td>
<td>The search value.</td>
</tr>
<tr>
<td>eArray</td>
<td>array of strings</td>
<td>Potential matches.</td>
</tr>
</tbody>
</table>

Example

Values = new Array("Apple","Pear","Banana");
X = exists("Apple",Values);
X is true.

externalLP_CA

Validates a license with the California State License Board and refreshes LP information with results.

Version
1.6
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>licNum</td>
<td>string</td>
<td>Valid CA license number. Non-alpha, max 8 characters. If null, the function uses the LPs on the supplied record ID.</td>
</tr>
<tr>
<td>rlpType</td>
<td>string</td>
<td>License professional type to use when validating and creating new LPs.</td>
</tr>
<tr>
<td>doPopulateRef</td>
<td>boolean</td>
<td>If true, creates/refreshes a reference LP of this number/type.</td>
</tr>
<tr>
<td>doPopulateTrx</td>
<td>boolean</td>
<td>If true, copies create/refreshed reference LPs to the supplied Cap ID. doPopulateRef must be true for this to work.</td>
</tr>
<tr>
<td>itemCap</td>
<td>CapIDModel</td>
<td>If supplied, licenses on the record are validated. Is also refreshed if doPopulateRef and doPopulateTrx are true.</td>
</tr>
</tbody>
</table>

Notes

See the “CSLB Interface using the externalLP_CA function - v3_0.pdf” document for detailed information.

Example

appsubmitbefore (validates the LP entered, if any, and cancels the event if the LP is inactive, cancelled, expired, etc.)

```cslbMessage =
externalLP_CA(CAELicenseNumber,false,false,CAELicenseType,null);```

appsubmitafter (update all CONTRACTOR LPs on the record and REFERENCE with data from CSLB. Link the record LPs to REFERENCE. Pop up a message if any are inactive...)

```cslbMessage =
externalLP_CA(null,true,true,"CONTRACTOR",capId)```  

feeAmount

Returns the total amount of the all fees on the record whose fee code is feestr. If optional fStatus1, ... fStatusn parameter(s) are supplied, also checks that feestr has one of the statuses in fStatus1, ... fStatusn.

Version

1.5
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feestr</td>
<td>string</td>
<td>Fee code.</td>
</tr>
<tr>
<td>fStatus1 …</td>
<td>string</td>
<td>List of fee statuses to check for. Enter one or</td>
</tr>
<tr>
<td>fStatusn</td>
<td>(optional)</td>
<td>more statuses.</td>
</tr>
</tbody>
</table>

Notes
A fee has one of the following statuses: NEW, INVOICED, VOIED, CREDITED.

feeAmountExcept

Returns the total amount of the all fees on the record. Ignores fees that are supplied as additional parameters.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkCapId</td>
<td>CapIDModel</td>
<td>Record ID to search.</td>
</tr>
<tr>
<td>feeCodeToIgnore</td>
<td>string</td>
<td>One or more fee codes to ignore.</td>
</tr>
</tbody>
</table>

feeBalance

Returns the total balance due for all fees on the record whose fee code is feestr. If parameter feeSchedule is used, retrieves those fees whose schedule is feeSchedule.

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feestr</td>
<td>string</td>
<td>Fee code.</td>
</tr>
<tr>
<td>feeSchedule</td>
<td>string</td>
<td>Fee schedule.</td>
</tr>
</tbody>
</table>

feeCopyByDateRange

On the current record, searches for fees in the given date and status criteria, then copies the fees onto the current record.
Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pStartDate</td>
<td>string</td>
<td>Starting search date for fee items.</td>
</tr>
<tr>
<td>pEndDate</td>
<td>string</td>
<td>Ending search date for fee items.</td>
</tr>
<tr>
<td>feeStatus</td>
<td>string</td>
<td>Search for fee items of this status.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>feeStatus</td>
<td>string</td>
<td>Search for fee items of this status.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

feeExists

Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feeestr</td>
<td>string</td>
<td>Fee code of fee to check for.</td>
</tr>
<tr>
<td>fStatus₁ … fStatusₙ</td>
<td>string</td>
<td>List of fee statuses to check for. Enter one or more statuses.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Returns

Returns true if a fee whose fee code is feeestr has been added to the record.

Notes

If optional fStatus₁ … fStatusₙ parameter(s) are supplied, also checks that feeestr has one of the statuses in fStatus₁ … fStatusₙ.

A fee has one of the following statuses: NEW, INVOICED, VOIDED, CREDITED.

Example

To determine if fee “FEE001” has been added and not invoiced:

feeExists("FEE001","NEW")
**feeGetTotByDateRange**

*Version*
1.3

*Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pStartDate</td>
<td>string</td>
<td>Start of date range, in format MM/DD/YYYY.</td>
</tr>
<tr>
<td>pEndDate</td>
<td>string</td>
<td>End of date range, in format MM/DD/YYYY.</td>
</tr>
<tr>
<td>fStatus₁ …</td>
<td>string</td>
<td>List of fee statuses to check for. Enter one or more statuses.</td>
</tr>
<tr>
<td>fStatusₙ</td>
<td>(optional)</td>
<td></td>
</tr>
</tbody>
</table>

*Returns*

Returns total amount of fees that were assessed during the date range *pStartDate to pEndDate*.

*Notes*

If optional *fStatus₁ … fStatusₙ* parameter(s) are supplied, the fee must have one of the statuses in *fStatus₁ … fStatusₙ*.

A fee has one of the following statuses: NEW, INVOICED, VOIDED, CREDITED.

Fees are retrieved by their initial assess date, not invoiced date.

**feeQty**

*Version*
1.6

*Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feestr</td>
<td>string</td>
<td>Fee item to search.</td>
</tr>
</tbody>
</table>

*Returns*

On the current record, returns the quantity field of the given fee item.

**getAddressConditions**

Searches for address conditions by the following parameters. Additionally *pType, pStatus, pDesc, and pImpact* can be passed as null values for wildcard searches.
**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
<tr>
<td>plImpact</td>
<td>string</td>
<td>Condition impact code.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIdModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

**Notes**

This function can only work well when the Accela Automation site supports Arabic.

**getAppIdByASI**

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIName</td>
<td>string</td>
<td>App specific info field name to search for.</td>
</tr>
<tr>
<td>ASIValue</td>
<td>string</td>
<td>App specific info field value to search for. Record ID object for record whose app spec info field ASIValue is to be changed to ASIValue.</td>
</tr>
<tr>
<td>ats</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the record number (cap ID string) of the first record whose record type matches ats and whose application specific info field ASIValue has the value of ASIValue.

**getAppIdByName**

**Version**

1.3
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaGroup</td>
<td>string</td>
<td>Record group</td>
</tr>
<tr>
<td>gaType</td>
<td>string</td>
<td>Record type</td>
</tr>
<tr>
<td>gaName</td>
<td>string</td>
<td>Record name</td>
</tr>
</tbody>
</table>

Returns

Returns the cap ID string of the first record whose record type begins with `gaGroup / gaType` and whose record name is `gaName`.

Notes

The parameter `gaType` is the 2nd value in the 4 level record type.

getApplication

Version

1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationNumber</td>
<td>string</td>
<td>Application # (B1_ALT_ID).</td>
</tr>
</tbody>
</table>

Returns

Returns the record ID object for record `applicationNumber` that can be used by other functions.

getAppSpecific

Version

1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemName</td>
<td>string</td>
<td>Application Specific Info field to get.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

Returns

Returns the value of the application spec info field `itemName`. If you provide `capId`, returns the value of `itemName` on the record whose record ID object is `capId`. 
**getCapByAddress**

*Version*
1.4

*Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ats</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
</tbody>
</table>

*Returns*

Returns the first record having the same address as the current record and whose record type matches ats, as a record ID object. If the search does not return any records, the function does not return any value.

*Notes*

The function matches addresses by Street # (start), Street Name, Street Direction, Street Suffix, and Zip. The function can return the current record.

**getCAPConditions**

Searches for record conditions by the following parameters. Additionally you can pass pType, pStatus, pDesc, and pImpact as null values for wildcard searches.

*Version*
2.0

*Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
<tr>
<td>pImpact</td>
<td>string</td>
<td>Condition impact code.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIdModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

*Notes*

This function can only work well when the Accela Automation site supports Arabic.
getCapId

Gets the ID of the record associated with the event.

getCapsWithConditionsRelatedByRefContact

Searches for records that share the same reference contact and same record condition, and returns the result as an array of CapIDModels.

Version

2.0

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemCap</td>
<td>string</td>
<td>The capIDModel of record.</td>
</tr>
<tr>
<td>capType</td>
<td>string</td>
<td>Application type.</td>
</tr>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type, leave null for wildcard search.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
<tr>
<td>pImpact</td>
<td>string</td>
<td>Condition impact code.</td>
</tr>
</tbody>
</table>

getChildren

If you use the `skipChildCapId` parameter, the function excludes any child record whose record ID object is `skipChildCapId`.

Version

1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCapType</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
<tr>
<td>pParentCapId</td>
<td>CapIDModel</td>
<td>Record ID object for parent record. Use null if <code>skipChildCapId</code> parameter is used.</td>
</tr>
<tr>
<td>skipChildCapId</td>
<td>CapIDModel</td>
<td>Record ID object of child record to exclude.</td>
</tr>
</tbody>
</table>
**Returns**

Returns all child records whose record type matches `pCapType`, as an array of record ID objects. If the `pParentCapId` parameter is used, returns child records of the record whose record ID object is `pParentCapId`.

**Notes**

If the `skipChildCapId` parameter is used, the function excludes any child record whose record ID object is `skipChildCapId`.

**See also**

`childGetByCapType`

---

**getChildTasks**

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskName</td>
<td>string</td>
<td>Name of criteria parent task.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

**Returns**

Returns an array of `taskScriptModel` objects, which represent the child tasks (sub process) of the criteria task.

---

**getConditions**

Searches for cap conditions, address conditions, contact conditions, parcel conditions, and licensed professional conditions by the following parameters. Additionally `pType`, `pStatus`, `pDesc`, and `pImpact` can be passed as null values for wildcard searches.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
</tbody>
</table>
### Notes
This function can only work well when the Accela Automation site supports Arabic.

### getContactArray
Retrieves field values and customizes attribute values for all contacts and returns them as an array of associative arrays. Each element in the outer array contains an associative array of values for one contact. Each element in each inner associative array is a different field.

#### Version
2.0

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capIdFrom</td>
<td>CapIDModel</td>
<td>Record ID object for source application.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Notes
The following fields are retrieved:

<table>
<thead>
<tr>
<th>Contact Field</th>
<th>Element Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td>firstName</td>
</tr>
<tr>
<td>Middle Name</td>
<td>middleName</td>
</tr>
<tr>
<td>Last Name</td>
<td>lastName</td>
</tr>
<tr>
<td>Business Name</td>
<td>businessName</td>
</tr>
<tr>
<td>Phone 1</td>
<td>phone1</td>
</tr>
<tr>
<td>Phone 2</td>
<td>phone2</td>
</tr>
<tr>
<td>Contact Type</td>
<td>contactType</td>
</tr>
<tr>
<td>Relationship</td>
<td>relation</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>contactSeqNumber</td>
</tr>
<tr>
<td>Reference Contact ID</td>
<td>refSeqNumber</td>
</tr>
<tr>
<td>E-mail</td>
<td>email</td>
</tr>
<tr>
<td>Address Line 1</td>
<td>addressLine1</td>
</tr>
</tbody>
</table>
All custom attributes are also added to the associative array, where the element name is the attribute name (in upper-case). Note that the attribute name may not be the same as the attribute label.

If the parameter `capIdFrom` is used, function retrieves contacts from the record whose record ID object is `capIdFrom`.

### getContactConditions

Searches for contact conditions by the following parameters. Additionally `pType`, `pStatus`, `pDesc`, and `pImpact` can be passed as null values for wildcard searches.

#### Version

2.0

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
<tr>
<td>pImpact</td>
<td>string</td>
<td>Condition impact code.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIdModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

#### Notes

This function can only work well when the Accela Automation site supports Arabic.

### getCSLBInfo

Selects the first licensed professional on the record and retrieves its data from the California State License Board (CSLB). If `doWarning` is `true`, shows a warning message if the license has expired. If `doPop` is `true`, updates the record's licensed professional with data from CSLB.
### Version
1.4

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doPop</td>
<td>boolean</td>
<td>Use true if the record's license professional must be updated with data from the California State License Board (CSLB); otherwise, use false.</td>
</tr>
<tr>
<td>doWarning</td>
<td>boolean</td>
<td>Use true if warning message should appear if license has expired; otherwise, use false.</td>
</tr>
</tbody>
</table>

### Returns
Returns false if the record has no licensed professional, if the license cannot be found at CSLB, or if any error is encountered.

### Notes
The following fields are updated:
- Business Name
- Phone Number
- Address Line 1
- Issued Date
- Address Line 2
- Expiration Date
- City
- State
- Zip

### getDepartmentName

### Version
1.4

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>string</td>
<td>User's ID.</td>
</tr>
</tbody>
</table>

### Returns
Returns the department of the user whose ID is username.
getGISBufferInfo

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>string</td>
<td>GIS service name.</td>
</tr>
<tr>
<td>layer</td>
<td>string</td>
<td>GIS layer on which the function creates buffer zones around the input GIS object.</td>
</tr>
<tr>
<td>numDistance</td>
<td>integer</td>
<td>The distance (in feet) around the to-be-buffered GIS object in which buffer zones are created on the specified layer. A positive distance means creating buffers outside the GIS object while a negative distance means creating buffers inside the GIS object. When the buffer distance is negative, Accela GIS checks whether the to-be-buffered GIS object is a point, line, or polygon. If it is a point or line, Accela GIS changes the negative buffer distance to 0.01 to avoid the script error.</td>
</tr>
<tr>
<td>attribute1...attributeN</td>
<td>strings</td>
<td>Additional attributes of the GIS layer to retrieve.</td>
</tr>
</tbody>
</table>

Returns

Returns an array of associative arrays. Each element in the outer array is a GIS object (from the indicated layer) within the buffer from the record’s GIS object. Each element in the inner associative array is a requested attribute.

Example

```javascript
x = getGISBufferInfo("NewtonCounty","Parcels","50","NAME1","TOTACRES");
x[0]["TOTACRES"] = 0.46
x[0]["NAME1"] = "JENNINGS DEMETRIA C"
x[1]["TOTACRES"] = 0.46
x[1]["NAME1"] = "SIMMS ROCK & VALARIE"
x[2]["TOTACRES"] = 0.46
x[3]["NAME1"] = "PAUL NEVILLE & MARGARET"
```

getGISInfo

Use with all events (and master scripts) except ApplicationSubmitBefore.

Version
1.4
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>string</td>
<td>GIS service name.</td>
</tr>
<tr>
<td>layer</td>
<td>string</td>
<td>GIS layer.</td>
</tr>
<tr>
<td>attributename</td>
<td>string</td>
<td>Name of attribute to retrieve.</td>
</tr>
</tbody>
</table>

### Returns

Returns the attribute value for `attributename` in the GIS layer for the last GIS object on the record.

#### getGISInfoArray

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>string</td>
<td>GIS service name.</td>
</tr>
<tr>
<td>layer</td>
<td>string</td>
<td>GIS layer.</td>
</tr>
<tr>
<td>attributename</td>
<td>string</td>
<td>Name of attribute to retrieve.</td>
</tr>
</tbody>
</table>

**Returns**

Similar to `getGISInfo`, except it returns an array of values for the given attribute, instead of the first value found.

#### getGuideSheetObjects

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspId</td>
<td>long</td>
<td>Sequence number of the inspection that contains the guidesheet objects to retrieve.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapiIDModel</td>
<td>Record Id to search.</td>
</tr>
</tbody>
</table>
**Returns**
Returns an array of guideSheetObject objects that represent the guidesheet data on the inspection.

**Notes**
See the guideSheetObject for more information

---

**getInspector**

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insp2Check</td>
<td>inspDesc</td>
<td>Inspection description.</td>
</tr>
</tbody>
</table>

**Returns**
Returns the user ID of the inspector assigned to inspection *insp2Check* whether scheduled or completed.

**Notes**
If more than one *insp2Check* is on the record, the first inspection found is selected, which may or may not be the *insp2Check* with the earliest inspection date.

---

**getLastErrorInspector**

**Version**
1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insp2Check</td>
<td>string</td>
<td>Inspection description.</td>
</tr>
</tbody>
</table>

**Returns**
Returns the user ID of the last inspector to result the inspection *insp2Check*. 
getLastScheduledInspector

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insp2Check</td>
<td>string</td>
<td>Inspection description.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the user ID of the last inspector to be schedule on the inspection `insp2check`.

getLicenseConditions

Searches for licensed professional conditions by the following parameters. Additionally `pType`, `pStatus`, `pDesc`, and `pImpact` can be passed as null values for wildcard searches.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
<tr>
<td>pImpact</td>
<td>string</td>
<td>Condition impact code.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIdModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

**Notes**

This function can only work well when the Accela Automation site supports Arabic.

getLicenseProfessional

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemcapId</td>
<td>CapIdModel</td>
<td>Record ID to use.</td>
</tr>
</tbody>
</table>
**Returns**

Returns an array of LicensedProfessional objects that represent all LPs on the specified record.

### getParcelConditions

Searches for parcel conditions by the following parameters. Additionally `pType`, `pStatus`, `pDesc`, and `pImpact` can be passed as null values for wildcard searches.

#### Version

2.0

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pType</td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td>pStatus</td>
<td>string</td>
<td>Condition status.</td>
</tr>
<tr>
<td>pDesc</td>
<td>string</td>
<td>Condition description.</td>
</tr>
<tr>
<td>pImpact</td>
<td>string</td>
<td>Condition impact code.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

#### Notes

This function can only work well when the Accela Automation site supports Arabic.

### getParent

#### Version

1.3

#### Parameters

None

#### Returns

Returns the record ID object for the first parent of the current record.

### getParents

#### Version

1.5
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemCap (optional)</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
</tbody>
</table>

**Returns**

Returns all parents on the current record in a record ID object array. If itemCap parameter is passed, only returns parent records whose record type matches the itemCap parameter string pattern.

### getRefLicenseProf

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refstlic</td>
<td>string</td>
<td>State license number to search for.</td>
</tr>
</tbody>
</table>

**Returns**

Returns a reference licensed professional object for the LP that matches the state license number value.

### getRelatedCapsByAddress

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ats</td>
<td>string</td>
<td>Four level application type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
</tbody>
</table>

**Returns**

Returns all records having the same address as the current record and whose record type matches ats, as an array of record ID objects. If the function does not find any related records, the function does not return any value.
Notes
The function matches addresses by Street # (start), Street Name, Street Direction, and Street Suffix. The function does not include the current record in the returned array. Retrieve records do not have to be a parent or child of the current record.

getRelatedCapsByParcel

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ats</td>
<td>string</td>
<td>Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.</td>
</tr>
</tbody>
</table>

Returns
Returns all records having the same parcel as the current record and whose record type matches ats, as an array of record ID objects. The function does not include the current record in the returned array. If the function does not find any related records, the function does not return any value.

Notes
Records retrieved do not have to be a parent or child of the current record.

getReportedChannel

Version
1.5

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for application.</td>
</tr>
</tbody>
</table>

Returns
Returns the value of the Reported Channel field as a string. If null, the function returns an empty string.
getScheduledInspId

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insp2Check</td>
<td>string</td>
<td>Inspection description.</td>
</tr>
</tbody>
</table>

Returns

Returns the internal sequence number for the inspection record that matches the description. Only returns values for scheduled inspections, not resulted inspections.

Notes

You can use the returned sequence number with other functions, such as autoAssignInspection.

getShortNotes

Version
1.5

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

Returns

Returns the value of the Short Notes field as a string. If null, the function returns an empty string.

getTaskDueDate

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td>wffProcess (optional)</td>
<td>string</td>
<td>Workflow process name.</td>
</tr>
</tbody>
</table>
**Returns**

Returns the due date of the requested workflow task on the current record.

**Notes**

If a record's workflow contains duplicate \textit{wfstr} tasks, use parameter \textit{wfProcess} to specify the process or subprocess whose \textit{wfstr} to check.

\textit{wfProcess} is R1\_PROCESS\_CODE in the GPROCESS and SPROCESS tables. \textit{wfProcess} is normally in uppercase.

**getTaskStatusForEmail**

This function retrieves all completed tasks on workflow \textit{stask} and returns their task name, status, and comments (if any) in the following format:

- Task Name: \{task name\}
- Task Status: \{task status\}
- Task Comments: \{status comments\}

The function repeats the previous block for each completed task.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stask</td>
<td>string</td>
<td>Process name of workflow.</td>
</tr>
</tbody>
</table>

**hasPrimaryAddressInCap**

Checks whether a record has a primary address.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capID</td>
<td>CapIDModel</td>
<td>Record ID object.</td>
</tr>
</tbody>
</table>

**insertSubProcess**

Dynamically adds a workflow process as a subprocess to an existing task.
**Version**
2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskName</td>
<td>string</td>
<td>Name of the task that is the parent for the sub-process.</td>
</tr>
<tr>
<td>process</td>
<td>string</td>
<td>Name of the reference workflow process that the function adds a subprocess.</td>
</tr>
<tr>
<td>completeReqd</td>
<td>boolean</td>
<td>True if you must complete the subprocess before you promote the parent task.</td>
</tr>
<tr>
<td>itemCap</td>
<td>CapIDModel</td>
<td>Optional target capId.</td>
</tr>
</tbody>
</table>

**Example**

```
insertSubProcess("Reviews","PLAN_REVIEW_VER1",true);
```

**inspCancelAll**

Cancels all scheduled and incomplete inspections on the current record.

**Version**
1.4

**Parameters**

None

**Returns**

Returns `true` if at least one inspection is cancelled; otherwise, returns `false`.

**invoiceFee**

Invoices all assessed fees with fee code of `fcode` and fee period of `fperiod`.

**Version**
1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcode</td>
<td>string</td>
<td>Fee code of the fee to invoice.</td>
</tr>
<tr>
<td>fperiod</td>
<td>string</td>
<td>Fee period of the fee to invoice.</td>
</tr>
</tbody>
</table>

Returns

Returns **true** if the function finds the assessed. Otherwise, returns **false**.

### isScheduled

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspType</td>
<td>string</td>
<td>Inspection description.</td>
</tr>
</tbody>
</table>

**Returns**

Returns **true** for scheduled or resulted inspections `inspType` for the current record.

**Notes**

To identify a scheduled, but not yet resulted inspection, use the `checkInspectionResult` function and use Scheduled for the `insp2Result` parameter.

### isTaskActive

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
</tbody>
</table>
|wfProcess  
(optional) | string  | Workflow process name.    |

**Returns**

Returns **true** if workflow task `wfstr` is active, or **false** if it is not.

If used with the `WorkflowTaskUpdateAfter` event, this function returns **true** if `wfstr` becomes active as a result of the `WorkflowTaskUpdateAfter` event. The function returns **false** if `wfstr` becomes inactive as a result of the `WorkflowTaskUpdateAfter` event.

**Notes**

If record’s workflow contains duplicate `wfstr` tasks, use parameter `wfProcess` to specify the process or subprocess whose `wfstr` to check.
**isTaskComplete**

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td>wfProcess</td>
<td>string</td>
<td>Workflow process name.</td>
</tr>
</tbody>
</table>

**Returns**

Returns true for a completed workflow task *wfstr*. Otherwise, returns false.

If used with the WorkflowTaskUpdateAfter event, this function returns true if *wfstr* becomes completed as a result of the WorkflowTaskUpdateAfter event.

**Notes**

If record’s workflow contains duplicate *wfstr* tasks, use parameter *wfProcess* to specify the process or subprocess whose *wfstr* to check.

*wfProcess* is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. *wfProcess* is normally in uppercase.

---

**isTaskStatus**

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td>wfstat</td>
<td>string</td>
<td>Workflow status.</td>
</tr>
<tr>
<td>wfProcess</td>
<td>string</td>
<td>Workflow process name.</td>
</tr>
</tbody>
</table>

**Returns**

Returns **true** if workflow task *wfstr* has the current status of *wfstat*, or **false** if it does not.

Returns **false** if the function does not find *wfstr*.
Notes
If record’s workflow contains duplicate \textit{wfstr} tasks, use parameter \textit{wfProcess} to specify the process or subprocess whose \textit{wfstr} to check.

\textit{wfProcess} is R1\_PROCESS\_CODE in the GPROCESS and SPROCESS tables. \textit{wfProcess} is normally in uppercase.

\textbf{jsDateToASIDate}

Converts the JavaScript Date object to a string, with a zero pad date format, that you can use in ASI, TSI, and ASI Table date fields.

\textbf{Version}

1.5

\textbf{Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateValue</td>
<td>JavaScript date</td>
<td>JavaScript date object.</td>
</tr>
</tbody>
</table>

\textbf{jsDateToMMDDYYYY}

Converts the JavaScript Date object \textit{pJavaScriptDate} to a string in the format MM/DD/YYYY.

\textbf{Version}

1.4

\textbf{Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pJavaScriptDate</td>
<td>JavaScript date</td>
<td>JavaScript date object.</td>
</tr>
</tbody>
</table>

\textbf{Returns}

Returns the date as a string in the format MM/DD/YYYY.

\textbf{Notes}

Use this function to display a JavaScript date in the format MM/DD/YYYY. Do not use the result of this function directly to compare against another date.

\textbf{licEditExpInfo}

Changes the record’s expiration status to \textit{pExpStatus} and expiration date to \textit{pExpDate}.

\textbf{Version}

1.4
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pExpStatus</td>
<td>string</td>
<td>Expiration status. Use null if you only edit expiration date.</td>
</tr>
<tr>
<td>pExpDate</td>
<td>string</td>
<td>Expiration date. Use null if you only edit expiration status.</td>
</tr>
</tbody>
</table>

**Notes**

If `pExpStatus` is null, expiration status does not change. If `pExpDate` is null, expiration date does not change. Use this function with license records only, that is the record type begins with Licenses.

`pExpDate` can be in YYYY-MM-DD or MM/DD/YYYY format.

Script throws an error if record does not have Renewal Info.

---

**loadAddressAttributes**

Populates `thisArr` as a associate array of address attributes and address values based on the address associated with the record.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisArr</td>
<td>array</td>
<td>Target array of address attributes.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapiIDModel</td>
<td>Record ID to search.</td>
</tr>
</tbody>
</table>

---

**loadAppSpecific[4ACA]**

Retrieves all application specific info fields and adds them to the associative array `thisArr`.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisArr</td>
<td>array</td>
<td>Associative array.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapiIDModel</td>
<td>ID for record from where to copy all app spec info fields.</td>
</tr>
</tbody>
</table>

---
Notes

The element name is the application specific info field name and the element value is the field value. If the user configurable variable `useAppSpecificGroupName` on the master script equals `true`, the function appends group name to the beginning of the field name with a period, that is CONSTRUCTION_INFO.Construction Type. The function does not retrieve application specific information table data.

If the function uses the `capld` parameter, the function retrieves application info fields from the record whose record ID object is `capld`.

The `loadAppSepecific4ACA` performs the same function as `loadAppSpecific` but it specially works with Accela Citizen Access pageflow scripts.

loadASITable

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tname</td>
<td>string</td>
<td>Name of ASI table to load.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID from which to load the table.</td>
</tr>
</tbody>
</table>

Returns

Returns an array of associate arrays that contain objects representing the contents of the ASI table for the selected record.

Notes

The underlying object is an "asiTableValObj" that contains three properties:

- `fieldValue` = value of the table
- `columnName` = name of the column for this value
- `readOnly` = Y for a read only field, N if not.

Example

```javascript
myTable = loadASITable(“EXAMPLE TABLE”)
firstRow = myTable[0];
columnA = firstRow[“Column A”]
columnB = firstRow[“Column B”]
comment(“value of column a is : “ + columnA.fieldValue)
comment(“column a read only property is : “ + columnA.readOnly)
```

The `fieldValue` property of the `asiTableValObj` object is the default property, so the following also works:
comment("value of column a is : " + columnA);

loadASITables[4ACA][Before]

Similar to the loadASITable function, except the function creates global variables for each ASI table on the requested record.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID from which to load the table.</td>
</tr>
</tbody>
</table>

Notes

You can edit the names of the tables remove whitespace and leading digits, so that they become appropriate JavaScript variables.

Example

```javascript
loadASITables();
if (typeof(PROPERNAMES) == "object")
    comment("number of rows in the ‘PROPER NAMES’ table : “ +
    PROPERNAMES.length)
```

Variables are not created for tables that do not have any data, so you must first use the JavaScript typeof operator to check for the presence of the table variable, as shown in the previous example.

By default, all master scripts execute loadASITables.

The loadASITables4ACA performs the same function as loadASITables but it specially works with Accela Citizen Access pageflow scripts.

The loadASITablesBefore is an alternate version of this function that works specifically with the ApplicationSubmitBefore event.

loadFees

Version
1.5

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object of record from which to load fees.</td>
</tr>
</tbody>
</table>
Returns
Retrieves all assessed fees for the record capId and returns them as an array of associative arrays.

Notes
Each element in the outer array contains an associative array of values for one fee. Each element in each inner associative array is a different field. The function retrieves the following fields:

<table>
<thead>
<tr>
<th>Fee Field</th>
<th>Element Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Num</td>
<td>sequence</td>
</tr>
<tr>
<td>Fee Code</td>
<td>code</td>
</tr>
<tr>
<td>Description</td>
<td>description</td>
</tr>
<tr>
<td>Unit</td>
<td>unit</td>
</tr>
<tr>
<td>Amount</td>
<td>amount</td>
</tr>
<tr>
<td>Amount Paid</td>
<td>amountPaid</td>
</tr>
<tr>
<td>Applied Date</td>
<td>applyDate</td>
</tr>
<tr>
<td>Effective Date</td>
<td>effectDate</td>
</tr>
<tr>
<td>Status</td>
<td>status</td>
</tr>
<tr>
<td>Received Date</td>
<td>redDate</td>
</tr>
<tr>
<td>Fee Period</td>
<td>period</td>
</tr>
<tr>
<td>Display Order</td>
<td>display</td>
</tr>
<tr>
<td>Account Code 1</td>
<td>accCodeL1</td>
</tr>
<tr>
<td>Account Code 2</td>
<td>accCodeL2</td>
</tr>
<tr>
<td>Account Code 3</td>
<td>accCodeL3</td>
</tr>
<tr>
<td>Fee Formula</td>
<td>formula</td>
</tr>
<tr>
<td>Sub Group</td>
<td>subGroup</td>
</tr>
<tr>
<td>Calculation Flag</td>
<td>calcFlag</td>
</tr>
</tbody>
</table>

loadGuideSheetItems

Version
1.6
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspId</td>
<td>long</td>
<td>Inspection sequence number to load.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record to search.</td>
</tr>
</tbody>
</table>

Returns

Returns an associative array of guidesheet items from the indicated inspection.

Example

```javascript
gsArray = loadGuideSheetItems(234323);
comment(gsArray["Privacy Violation"]) 
```

Displays the value of the Privacy Violation guidesheet item.

loadParcelAttributes

Retrieves all parcel fields (including custom attributes) and adds them to the associative array `thisArr`.

Version

1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisArr</td>
<td>array</td>
<td>associative array.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for the record from where to copy parcel attributes.</td>
</tr>
</tbody>
</table>

Notes

The element name is the field name (prefixed with "ParcelAttribute.") and the element value is the field value. The function includes the following standard parcel fields:

- ParcelAttribute.Block
- ParcelAttribute.Book
- ParcelAttribute.CensusTract
- ParcelAttribute.CouncilDistrict
- ParcelAttribute.ExemptValue
- ParcelAttribute.ImprovedValue
- ParcelAttribute.InspectionDistrict
- ParcelAttribute.LandValue
- ParcelAttribute.LegalDesc
- ParcelAttribute.Lot
- ParcelAttribute.MapNo
- ParcelAttribute.MapRef
- ParcelAttribute.ParcelStatus
- ParcelAttribute.SupervisorDistrict
- ParcelAttribute.Tract
- ParcelAttribute.PlanArea
If the record has multiple parcels, the function only retrieves fields for the last parcel. If the function uses the `capId` parameter, the function retrieves parcel fields from the record whose record ID object is `capId`.

### loadTasks

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ltcapidstr</td>
<td>string</td>
<td>Application # (B1_ALT_ID).</td>
</tr>
</tbody>
</table>

**Returns**

Returns an array of workflow task objects for the record `ltcapidstr`.

### loadTaskSpecific

Retrieves all task specific info fields and adds them to the associative array `thisArr`.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thisArr</td>
<td>array</td>
<td>Associative array.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIdModel</td>
<td>Record ID object for the record from where to copy all task specific info fields.</td>
</tr>
</tbody>
</table>

**Notes**

The element name is the task specific info field name and the element value is the field value. If the user configurable variable `useTaskSpecificGroupName` on the master script equals `true`, the function prepends the workflow process code and workflow task name to the field name, for example, BLDGPROCESS.Application Submittal.Date Received.

If the function uses the `capId` parameter, the function retrieves task specific info fields from the record whose record ID object is `capId`.

### logDebug

Displays debug information, depending on the showDebug global variable setting.
**Version**  
1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstr</td>
<td>string</td>
<td>Value to display on the debug window.</td>
</tr>
<tr>
<td>debugLevel</td>
<td></td>
<td>Debug content destination.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

debugLevel overrides this setting for this message only.

- `debugLevel = false` // no output
- `debugLevel = 1`    // screen output
- `debugLevel = 2`    // output to biz server log
- `debugLevel = 3`    // output to screen and biz log

**lookup**

Looks up `valueName` in standard choices item `stdChoice`, and returns its value description. Essentially uses standard choices as a lookup table.

**Version**  
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoice</td>
<td>string</td>
<td>Standard choices item name.</td>
</tr>
<tr>
<td>stdValue</td>
<td>string</td>
<td>Standard choices value.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the Value Desc corresponding to the standard choices value `stdValue` in the standard choices item `stdChoice`. If the function does not find `stdValue`, returns **undefined**.

**lookupDateRange**

Matches `dateValue` against a series of dates in the standard choices called `stdChoiceEntry`.

**Version**  
1.4
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoiceEntry</td>
<td>string</td>
<td>Item Name of standard choices used as lookup table.</td>
</tr>
<tr>
<td>dateValue</td>
<td>string</td>
<td>Date that determines which row to return. Use string in format MM/DD/YYYY, e.g. “07/21/2000”.</td>
</tr>
<tr>
<td>valueIndex</td>
<td>(optional)</td>
<td>integer</td>
</tr>
</tbody>
</table>

Returns

If `dateValue` falls after date 1 but before or on date 2, returns the value following the caret (^) on date 1’s right. If the function uses the `valueIndex` parameter, returns the value immediately after the `valueIndex`th caret (^), following the matching date.

Notes

Set up the standard choices lookup table as follows:

- **Value** column = Four digit incremental index. Must be left zero padded to four digits. Entire table must be consecutive.
- **Value Desc** column = at least two values separated with the caret (^) symbol. Returns the first value as the effective date (MM/DD/YYYY format). Returns the remaining values by the function.

Examples

```
lookupDateRange("test date lookup","5/5/2002")  returns 33333
lookupDateRange("test date lookup","1/5/2000",2)  returns 12222
lookupDateRange("test date lookup","1/1/2010")   returns 44444
lookupDateRange("test date lookup","1/1/1999")   returns undefined since there is no entry effective for that date.
lookupDateRange("test date lookup","1/5/2000",3)  returns undefined since there are not 3 values.
```

Sample script controls:
lookupFeesByValuation

Looks up the Value Desc for the stdChoiceValue Value in the standard choices called stdChoiceEntry.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoiceEntry</td>
<td>string</td>
<td>Item name of standard choices used as lookup table.</td>
</tr>
<tr>
<td>stdChoiceValue</td>
<td>string</td>
<td>Standard choices value.</td>
</tr>
<tr>
<td>capval</td>
<td>number</td>
<td>Number value (e.g. valuation) to compare.</td>
</tr>
<tr>
<td>valueIndex</td>
<td>(optional) integer</td>
<td>Determines which value to return. Defaults to 1, the first value.</td>
</tr>
</tbody>
</table>

**Notes**

Compares capval against the series of numbers in the Value Desc. If valueIndex is null or 1, uses the value following the 1st pipe (|) on the matching number's right to calculate the base fee. If valueIndex is 2, uses the value following the 2nd pipe (|) on the matching number's right to calculate an add on fee.

Set up the standard choices lookup table as follows:

- **Value** column = Lookup value.
- **Value Desc** column = one or more 3-number series, where
  - 1st number = number to compare compareValue against
  - 2nd number = base fee
  - 3rd number = used to calculate add-on fee

Use a pipe(|) to separate each number. Use a caret(^) to separate each 3-number series.

```plaintext
01 appMatch("Building/Residential/SFD/*") ^lookupIndex=1
02 appMatch("Building/Residential/Duplex/*") ^lookupIndex = 2
03 true ^ addFee("FEECODE","FEESCHED","FEEPERIOD", lookupDateRange("test date lookup", filedate, lookupIndex), "Y")
```
**Example**

```plaintext
06  true ^ theBase = lookupFeesByValuation("PlanCheck2007","A-1-Group2",5600)
07  true ^ theAddOn = lookupFeesByValuation("PlanCheck2007","A-1-Group2",5600,2)
08  true ^ newTotal = newTotal +(parseFloat(theBase)+parseFloat(theAddOn))
```

`lookupFeesByValuationSlidingScale`

Similar to the `lookupFeesByValuation` function, but introduces another element in the standard choice tables which serves as a divisor for the capval.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdChoiceEntry</td>
<td>string</td>
<td>Item name of standard choices used as lookup table.</td>
</tr>
<tr>
<td>stdChoiceValue</td>
<td>string</td>
<td>Standard choices value.</td>
</tr>
<tr>
<td>capval</td>
<td>number</td>
<td>Number value (e.g. valuation) to compare.</td>
</tr>
<tr>
<td>valueIndex (optional)</td>
<td>integer</td>
<td>Determines which value to return. Defaults to 1, the first value.</td>
</tr>
</tbody>
</table>
Notes
Set up the **standard choices** lookup table as follows:
- **Value** column = Lookup value.
- **Value Desc** column = one or more 3-number series, where
  - 1st number = number to compare `compareValue` against
  - 2nd number = divisor (e.g., 100, 1000, etc.)
  - 3rd number = base fee
  - 4th number = used to calculate add-on fee

Use a pipe(|) to separate each number. Use a caret(^) to separate each 4-number series.

**loopTask**

**Version**
1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wfstr</code></td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td><code>wfstat</code></td>
<td>string</td>
<td>Status to assign.</td>
</tr>
<tr>
<td><code>wfcomment</code></td>
<td>string</td>
<td>Comment to add.</td>
</tr>
<tr>
<td><code>wfnote</code></td>
<td>string</td>
<td>Note to add to the workflow task.</td>
</tr>
<tr>
<td><code>wfProcess</code></td>
<td>string</td>
<td>ID (R1_PROCESS_CODE) for the process that the task belongs to. Required for multi-level workflows.</td>
</tr>
</tbody>
</table>

**Notes**
Updates the workflow task `wfstr` as follows:
- Status = `wfstat`
- Status Date = current date
- Status Comment = `wfcomment`
- Action By = current user

Closes task `wfstr` and promotes workflow to the loop task.
If record’s workflow contains duplicate `wfstr` tasks, use parameter `wfProcess` to specify the process or subprocess whose `wfstr` to check.
matches

Version
1.3

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eVal</td>
<td>string</td>
<td>String to match.</td>
</tr>
<tr>
<td>argList</td>
<td>strings</td>
<td>The m1 [, ... mn] list. List of values to test for a match. Enter any number of values, each enclosed in double quotes and separated by comma.</td>
</tr>
</tbody>
</table>

Returns

Returns true if the function finds value in the m1 [, ... mn] list. Function looks for an exact, case-sensitive match. Returns false if the function finds nothing in the m1 [, ... mn] list that matches value.

nextWorkDay

Version
1.4
utility
Get

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>td (optional)</td>
<td>string</td>
<td>Date, in format “MM/DD/YYYY” (or any string that converts to a JavaScript date).</td>
</tr>
</tbody>
</table>

Returns

Returns the first agency work day following the current date, by checking the Agency Workday calendar defined for the agency. If the function uses the td parameter, returns the first agency work day following td. The date returned is a string in the format MM/DD/YYYY.

Notes

You can only use this function with Accela Automation 6.3.2 and later.

openUrlInNewWindow

Opens a new browser window and shows the web page whose URL is myurl.
Version 1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myurl</td>
<td>string</td>
<td>URL of web page to open.</td>
</tr>
</tbody>
</table>

Notes
Either user-configurable variable `showDebug` or `showMessage` must be `true` for this function to work.

parcelConditionExists

Version 1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condtype</td>
<td>string</td>
<td>Condition type.</td>
</tr>
</tbody>
</table>

Returns
Returns `true` if any parcel has a condition of type `condtype`; otherwise, returns `false`.

parcelExistsOnCap

Version 1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID to check.</td>
</tr>
</tbody>
</table>

Returns
Returns true if a parcel exists on the record

paymentByTrustAccount

This function uses the trust account associated with a record to pay for a specific fee item.
**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fSeqNbr</td>
<td>long</td>
<td>Sequence number of the fee item to be paid.</td>
</tr>
<tr>
<td>itemCap</td>
<td>CapIDModel</td>
<td>Optional target record ID.</td>
</tr>
</tbody>
</table>

**Notes**

The logic behind the function is:

- Retrieves the primary trust account on the record.
- Initiates payment from this trust account for the amount of the fee.
- If payment successful, applies payment to the fee.
- Generates a receipt for the payment.
- Returns false if any of the previous fails. Otherwise returns true.
- You can only pay invoiced fees.

**Example**

```c
feeSeq = addFee(“C”,“F”,“P”,20,”Y”);
paymentByTrustAccount(feeSeq);
```

**paymentGetNotAppliedTot**

Gets the total amount of unapplied payments on the current record (capId), as a float number.

**Version**

2.0

**Parameters**

None

**proximity**

**Version**

1.3
Proximity to Attribute

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>string</td>
<td>GIS service name.</td>
</tr>
<tr>
<td>layer</td>
<td>string</td>
<td>GIS layer, i.e., object that function is testing proximity to.</td>
</tr>
<tr>
<td>numDistance</td>
<td>integer</td>
<td>Distance of parcel, associated with the current record, to the object that you identify with the layer parameter.</td>
</tr>
<tr>
<td>distanceType</td>
<td>string</td>
<td>Unit for distance measurement.</td>
</tr>
<tr>
<td>attributeName</td>
<td>string</td>
<td>Attribute name.</td>
</tr>
<tr>
<td>attributeValue</td>
<td>string</td>
<td>Attribute value.</td>
</tr>
</tbody>
</table>

**Returns**

Returns true if the record has a GIS object in numDistance proximity that contains an attribute called attributeName with the value attributeValue.

**Example**

```javascript
proximityToAttribute("flagstaff","Parcels","50","feet","BOOK","107") ^ DoStuff...
```
**refLicProfGetAttribute**

**Version**
1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pLicNum</td>
<td>string</td>
<td>State license number.</td>
</tr>
<tr>
<td>pAttributeName</td>
<td>string</td>
<td>Custom attribute name.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the value of the custom attribute named `pAttributeName` for the reference Licensed Professional whose license # is `pLicNum`.

**Notes**

Note that `pAttributeName` is not necessarily the same as the attribute label. You can find the attribute name in the attribute's configuration screen.

If the function does not find a reference Licensed Professional with license # of `pLicNum`, the function returns NO LICENSE FOUND. If the function does not find the attribute `pAttributeName`, the function returns ATTRIBUTE NOT FOUND.

**refLicProfGetDate**

**Version**
1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pLicNum</td>
<td>string</td>
<td>State license number.</td>
</tr>
<tr>
<td>pDataType</td>
<td>string</td>
<td>Date field to retrieve. Options (use one): EXPIRE, ISSUE, RENEW, INSURANCE, BUSINESS.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the date specified by `pDataType` for the reference Licensed Professional whose license # is `pLicNum`. The date returned is a JavaScript Date object.

**Notes**

The table below shows the date returned for each `pDataType` parameter value.

<table>
<thead>
<tr>
<th>dateType</th>
<th>Date Field Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPIRE</td>
<td>License Expiration Date</td>
</tr>
</tbody>
</table>
If the function does not find a reference Licensed Professional with license # of \texttt{pLicNum}, the function returns NO LICENSE FOUND. If the function does not find a date, the function returns NO DATE FOUND. If \texttt{pLicNum} is empty, the function returns INVALID PARAMETER. The function skips disabled reference Licensed Professional.

To format a JavaScript Date as a MM/DD/YYYY string, use function \texttt{jsDateToMMDDYYYY}.

**removeAllFees**

Removes all un-invoiced fees on the record

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemCap</td>
<td>CapIDModel</td>
<td>The capIDModel of record.</td>
</tr>
</tbody>
</table>

**removeASITable**

Removes all entries for ASI Table Name

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>Table name to remove.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

**removeCapCondition**

Deletes the condition whose type is \texttt{cType} and name is \texttt{cDesc} from the current record. If you use the optional parameter \texttt{capId}, the function deletes the condition from the record \texttt{capId}.  

<table>
<thead>
<tr>
<th>ISSUE</th>
<th>License Issue Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>RENEW</td>
<td>License Last Renewal Date</td>
</tr>
<tr>
<td>INSURANCE</td>
<td>Insurance Expiration Date</td>
</tr>
<tr>
<td>BUSINESS</td>
<td>Business License Expiration Date</td>
</tr>
</tbody>
</table>
**removeFee**

Deletes all assessed fees with the fee code of `fcode` and fee period of `fperiod`. The function does not delete invoiced fees.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fcode</code></td>
<td>string</td>
<td>Fee code of the fee to delete.</td>
</tr>
<tr>
<td><code>fperiod</code></td>
<td>string</td>
<td>Fee period of the fee to be delete.</td>
</tr>
</tbody>
</table>

**removeParcelCondition**

Removes the condition whose name is `cDesc` and type is `cType` from the reference parcel whose number is `parcelNum`. If you set the parameter `parcelNum` to `null`, the function removes any condition, whose name is `cDesc` and type is `cType`, from all parcels on the record.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parcelNum</code></td>
<td>string</td>
<td>Parcel number from which to remove the condition.</td>
</tr>
<tr>
<td><code>cType</code></td>
<td>string</td>
<td>Condition type.</td>
</tr>
<tr>
<td><code>cDesc</code></td>
<td>string</td>
<td>Condition name.</td>
</tr>
</tbody>
</table>

**removeTask**

Dynamically edits the workflow on the indicated record by removing the task.
**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetCapId</td>
<td>CapIdModel</td>
<td>Record Id to edit.</td>
</tr>
<tr>
<td>removeTaskName</td>
<td>string</td>
<td>Name of the task to remove.</td>
</tr>
<tr>
<td>wfProcess (optional)</td>
<td>string</td>
<td>Workflow process name.</td>
</tr>
</tbody>
</table>

**replaceMessageTokens**

Used for formatting emails, this function parses through the string, replacing tokens with variable values.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>string</td>
<td>String to do the token replacement.</td>
</tr>
</tbody>
</table>

**Notes**

The function replaces values inside pipes (e.g. \( |capIdString| \)) by their script values.

The function replaces values inside curly brackets (e.g. \( |ASIVal| \)) by ASI values.

**Example**

```
EmailAddress = "Thank you for submitting |capIdString| on |fileDate|. The balance due is |balanceDue|. The ASI field is |ASI Field|
EmailSend = replaceMessageTokens(EmailAddress);
```

This function can access any variable that the script uses.

**resultInspection**

This function posts a result for a scheduled inspection. If no scheduled inspection exists (of that type for the record) then the function does nothing.

**Version**

1.6
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspType</td>
<td>string</td>
<td>Inspection type to result.</td>
</tr>
<tr>
<td>inspStatus</td>
<td>string</td>
<td>Resulting status.</td>
</tr>
<tr>
<td>resultDate</td>
<td>string</td>
<td>Posted date of the result.</td>
</tr>
<tr>
<td>resultComment</td>
<td>string</td>
<td>Comment to add to the result.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID to result.</td>
</tr>
</tbody>
</table>

**scheduleInspectDate**

Schedules the inspection `iType` for the date `DateToSched`. If you supply `inspectorID`, the function assigns the scheduled inspection to the inspector whose Accela Automation user ID is `inspectorID`.

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTYPE</td>
<td>string</td>
<td>Inspection type.</td>
</tr>
<tr>
<td>DateToSched</td>
<td>string</td>
<td>Scheduled date of inspection.</td>
</tr>
<tr>
<td>inspectorID (optional)</td>
<td>string</td>
<td>User ID of inspector.</td>
</tr>
<tr>
<td>inspTime (optional)</td>
<td>string</td>
<td>Inspection time in HH12:MIAM format or AMPM (e.g. “12:00PM” or “PM”).</td>
</tr>
<tr>
<td>inspComm (optional)</td>
<td>string</td>
<td>Inspection comment.</td>
</tr>
</tbody>
</table>

**Note**

To specify the optional inspection time without passing in inspection use `scheduleInspectDate("Desc","01/01/2001",null, "AM")`.

To specify the option inspection comment without the other option parameters you can use `scheduleInspectDate("Desc","01/01/2001",null,null, "My Comment")`.

**scheduleInspection**

Schedules the inspection `iType` for `DaysAhead` days after current date. If you supply `inspectorID`, the function assigns the scheduled inspection to the inspector whose Accela Automation user ID is `inspectorID`. 
Version
1.5

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iType</td>
<td>string</td>
<td>Inspection type.</td>
</tr>
<tr>
<td>DaysAhead</td>
<td>number</td>
<td>Number of days in the future to schedule the inspection for.</td>
</tr>
<tr>
<td>inspectorID</td>
<td>string</td>
<td>User ID of inspector.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inspTime</td>
<td>string</td>
<td>Inspection time in HH12:MIAM format or AMPM (e.g. “12:00PM” or “PM”).</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inspComm</td>
<td>string</td>
<td>Inspection comment.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes
To specify the optional inspection time without passing in inspection use
scheduleInspectDate("Desc",5,null,"AM").

To specify the option inspection comment without the other option parameters you can use
scheduleInspectDate("Desc",5,null,null,"My Comment");

searchProject

Searches the entire hierarchy on the current record for related records that match the criteria.

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pProjType</td>
<td>app type string</td>
<td>Record type marking highest point to search. Ex. Building/Project/NA/NA.</td>
</tr>
<tr>
<td>pSearchType</td>
<td>app type string</td>
<td>Record type to search for. Ex. Building/Permit/NA/NA.</td>
</tr>
</tbody>
</table>

Returns
Returns CapID array of all unique matching SearchTypes

setIVR

Sets the record tracking number for IVR
**setTask**

Helper function to edit the active and complete flags on a task.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ivrnum</td>
<td>long</td>
<td>New IVR tracking number.</td>
</tr>
</tbody>
</table>

**Example**

To set a task to inactive/complete:

```javascript
setTask("Peer Review","N","Y");
```

**stripNN**

Strips all non-numeric characters from the string. Only numerals and the period character remain.

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fullStr</td>
<td>string</td>
<td>String to strip.</td>
</tr>
</tbody>
</table>
**taskCloseAllExcept**

Closes all tasks on the record except for tasks in the list \( \text{wfTask}_1 \ldots \text{wfTask}_n \). If you only supply the parameters \( p\text{Status} \) and \( p\text{Comment} \), the function closes all tasks on the record.

**Version**

1.4

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p\text{Status} )</td>
<td>string</td>
<td>Status to assign to tasks.</td>
</tr>
<tr>
<td>( p\text{Comment} )</td>
<td>string</td>
<td>Status comment to add to tasks.</td>
</tr>
<tr>
<td>( \text{wfTask}_1 \ldots \text{wfTask}_n )</td>
<td>string</td>
<td>Names of tasks to exclude. Enter one or more tasks separated by commas, each in double-quotes.</td>
</tr>
</tbody>
</table>

**Notes**

Before the function closes each task, the function updates the task as follows:

- Status = \( p\text{Status} \)
- Status Date = current date
- Status Comment = \( p\text{Comment} \)
- Action By = current user

**taskStatus**

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{wfstr} )</td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td>( \text{wfProcess} ) (optional)</td>
<td>string</td>
<td>ID (R1_PROCESS_CODE) for the process that the task belongs to.</td>
</tr>
<tr>
<td>( \text{capId} ) (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record to use.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the status of the workflow task \( \text{wfstr} \):
**Notes**

If record’s workflow contains duplicate schedule ins *wfstr* tasks, use parameter *wfProcess* to specify the process or subprocess whose *wfstr* to check.

If you use the parameter *capId*, the function retrieves data from the record *capId*.

### taskStatusDate

**Version**

1.5

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>wfstr</em></td>
<td>string</td>
<td>Workflow task name.</td>
</tr>
<tr>
<td><em>wfProcess</em> (optional)</td>
<td>string</td>
<td>ID (R1_PROCESS_CODE) for the process that the task belongs to.</td>
</tr>
<tr>
<td><em>capId</em> (optional)</td>
<td>CapIDModel</td>
<td>Record ID object for record to use.</td>
</tr>
</tbody>
</table>

**Returns**

Returns the current status date of the workflow task *wfstr*.

**Notes**

If record’s workflow contains duplicate *wfstr* tasks, use parameter *wfProcess* to specify the process or subprocess whose *wfstr* to use.

If you use the parameter *capId*, the function retrieves data from the record *capId*.

### transferFunds

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>parentAppNum</em></td>
<td>string</td>
<td>Record number to transfer funds to.</td>
</tr>
<tr>
<td><em>dollarAmount</em></td>
<td>number: double</td>
<td>Amount to transfer.</td>
</tr>
</tbody>
</table>

**Notes**

If the current record has sufficient funds (i.e. non-applied amount), transfers *dollarAmount* from the current record to the record *parentAppNum*. The function records the transaction as a Fund Transfer transaction on both records. If current record does not have sufficient funds, no fund transfer takes place.
**updateAddresses**

Updates the address in a record.

**Version**

2.0

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetCapID</td>
<td>CapIDModel</td>
<td>Record ID object.</td>
</tr>
<tr>
<td>addressModel</td>
<td>AddressModel</td>
<td>Address.</td>
</tr>
</tbody>
</table>

**updateAppStatus**

Updates record status of record to `stat` and adds `cmt` to the status update history.

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stat</td>
<td>string</td>
<td>Status to update the record to.</td>
</tr>
<tr>
<td>cmt</td>
<td>string</td>
<td>Comment to add to status update history.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object.</td>
</tr>
</tbody>
</table>

**Notes**

If you use the `capId` optional parameter, the function updates record `capId`. If you do not use the `capId` parameter, the function updates current record.

The `getApplication()`, `getParent()`, `createChild()`, `createCap()` functions each return a record ID object that you can use in the `capId` parameter.

**updateFee**

**Version**

1.5
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcode</td>
<td>string</td>
<td>Fee code of the fee to be updated/added.</td>
</tr>
<tr>
<td>fsched</td>
<td>string</td>
<td>Fee schedule of the fee to be updated/added.</td>
</tr>
<tr>
<td>fperiod</td>
<td>string</td>
<td>Fee period of the fee to be updated/added.</td>
</tr>
<tr>
<td>fqty</td>
<td>integer</td>
<td>Quantity to be updated/added.</td>
</tr>
<tr>
<td>finvoice</td>
<td>string</td>
<td>Flag for invoicing (&quot;Y&quot; or &quot;N&quot;).</td>
</tr>
<tr>
<td>pDuplicate</td>
<td>string</td>
<td>Allow duplicate invoiced fee (&quot;Y&quot; or &quot;N&quot;).</td>
</tr>
<tr>
<td>pFeeSeq</td>
<td>integer</td>
<td>Attempts to update a specific fee item.</td>
</tr>
</tbody>
</table>

Returns

For an updated fee, the function returns null. For an added fee, the function returns the fee sequence number.

Notes

If a fee whose fee code is fcode and fee period is fperiod has been assessed and not invoiced, updates the quantity on the fee to fqty. If invoice is Y, then invoices the fee. If there is more than one assessed fee with fcode and fperiod, updates the first fee found. If the fee is not found, adds the fee.

If this fee already exists and is invoiced, adds another instance of the same fee, unless pDuplicate is N. The duplicate fee has an adjusted quantity, which is fqty less quantity on previous fee.

If you use the pFeeSeq parameter, the function attempts to find the specified fee. If the function does not find the specified fee sequence number, the function adds a new fee based on the pDuplicate fee flag.

Warning: If adjusted quantity can be negative, do not use this function to add a fee. Accela Automation's cashier feature does not handle negative fees well. Set pDuplicate parameter to N.

updateRefParcelToCap

Refreshes parcel data on the specified record. The function refreshes parcel data on the record with reference parcel values.

Version

1.6
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID to process.</td>
</tr>
</tbody>
</table>

**updateShortNotes**

Updates the short notes on the specific capId detail record

**Version**

1.6

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newSN</td>
<td>string</td>
<td>New short notes value.</td>
</tr>
<tr>
<td>capId</td>
<td>CapIDModel</td>
<td>Record ID to update.</td>
</tr>
</tbody>
</table>

**updateTask**

Updates the workflow task *wfstr* as follows:

- Status = *wfstat*
- Status Date = current date
- Status Comment = *wfComment*
- Action By = current user

**Version**

1.3

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>wfstr</em></td>
<td>string</td>
<td>Name of workflow task to update.</td>
</tr>
<tr>
<td><em>wfstat</em></td>
<td>string</td>
<td>Status to update task to.</td>
</tr>
<tr>
<td><em>wfComment</em></td>
<td>string</td>
<td>Comment to update status comment to.</td>
</tr>
<tr>
<td><em>wfnote</em></td>
<td>string</td>
<td>Note to update task note to.</td>
</tr>
<tr>
<td><em>wfProcess</em></td>
<td>string</td>
<td>Workflow process that <em>wfstr</em> belongs to.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID object.</td>
</tr>
</tbody>
</table>
Notes
The workflow does not promote to the next task. To promote the workflow to the next task, use the closeTask, branchTask or loopTask function.

If record’s workflow contains duplicate `wfstr` tasks, use parameter `wfProcess` to specify the process or subprocess whose `wfstr` to check.

If you use the `capId` parameter, the function updates the record `capId`. If you use the `capId` parameter, you must use the `wfProcess` parameter by entering a process string or entering the word `null`.

updateTaskAssignedDate
Updated the assigned date of the workflow task `wfstr`. The function does not create a workflow history record.

Version
1.6

Parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wfstr</code></td>
<td>string</td>
<td>Workflow task to edit.</td>
</tr>
<tr>
<td><code>wfAssignDate</code></td>
<td>string</td>
<td>New assignment date.</td>
</tr>
<tr>
<td><code>wfProcess</code></td>
<td>string</td>
<td>Process name of workflow for <code>wfstr</code>. Case sensitive.</td>
</tr>
</tbody>
</table>

Notes
If record’s workflow contains duplicate `wfstr` tasks, use parameter `wfProcess` to specify the process or subprocess whose `wfstr` to activate.

updateTaskDepartment
Updated the assigned department for the workflow task `wfstr`. The function does not create a workflow history record.

Version
1.6
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfstr</td>
<td>string</td>
<td>Workflow task to edit.</td>
</tr>
<tr>
<td>wfDepartment</td>
<td>string</td>
<td>New department code.</td>
</tr>
<tr>
<td>wfProcess (optional)</td>
<td>string</td>
<td>Process name of workflow for wfstr. Case sensitive.</td>
</tr>
</tbody>
</table>

Notes

If record’s workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to activate.

Assigned department must be a string with 7 values separated by slashes, such as "ADDEV/DPE/ONLINE/LICENSE/NA/NA/NA"

updateWorkDesc

Updates the work description on the specific capId detail record.

Version

1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newWorkDes</td>
<td>string</td>
<td>New work description value.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIdModel</td>
<td>Record ID to update.</td>
</tr>
</tbody>
</table>

validateGisObjects

Version

1.3

Parameters

None

Returns

Returns true if all GIS objects on the current record validate in GIS, or false if any GIS object on the current record does not validate in GIS.
workDescGet

Version
1.4

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCapId</td>
<td>CapIDModel</td>
<td>Record ID object for record.</td>
</tr>
</tbody>
</table>

Returns

Returns work description for the record whose record ID object is pCapId.

Notes

The getApplication(), getParent(), createChild(), createCap() functions each return a record ID object.

zeroPad

Version
1.6

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>string</td>
<td>Number to zero pad.</td>
</tr>
<tr>
<td>count</td>
<td>integer</td>
<td>Number of digits required.</td>
</tr>
</tbody>
</table>

Returns

A zero-padded string of the supplied number that is count digits long.

Example

zeroPad("5",4) = "0005"
APPENDIX B:  
MASTER SCRIPT  
OBJECT LIST

Objects
- Fee
- genericTemplateObject
- guideSheetObject
- licenseProfObject
- licenseObject
- Task

Fee
Defines the fee object for use by fee functions, loadFees for example.

Parameters
- sequence code description
- unit amount amountPaid
- applyDate effectDate expireDate
- status recDate period
- display accCodeL1 accCodeL2
- accCodeL3 formula udes
- UDF1 UDF2 UDF3
- UDF4 subGroup calcFlag
- calcProc auditDate auditID
- auditStatus

genericTemplateObject
You can use this object to interact with the Application Specific Information and Application Specific Information Tables stored as generic template information on licensed professionals and conditions
Version
2.0

Constructors
Loads the genericTemplate objects and makes object data accessible through the read only parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtmp</td>
<td>genericTemplateModel</td>
<td>Generic template model from which to read information from.</td>
</tr>
</tbody>
</table>

Example

```javascript
var cond = aa.capCondition.getCapCondition(capId,445392).getOutput();
var tmpObj = genericTemplateObject(cond.getTemplateModel());
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI</td>
<td>An associative array comprised of data from all the ASI fields that the generic template contains as an associative array. The constructor sets the associated hasASI flag to true if the function finds valid ASI fields during creation. The object stores the associative array as ASI[label name] format.</td>
</tr>
<tr>
<td>ASIT</td>
<td>An associative array comprised of data from all the ASIT fields that the generic template contains as an associative array. The constructor sets the associated hasASIT flag to true if the function finds valid ASI fields during creation. The object stores this table in the ASIT[tableName][row][column] format.</td>
</tr>
<tr>
<td>hasASI</td>
<td>Boolean flag set to indicate if object has valid ASI loaded.</td>
</tr>
<tr>
<td></td>
<td>true = valid ASI found</td>
</tr>
<tr>
<td></td>
<td>false = no ASI found</td>
</tr>
<tr>
<td>hasTables</td>
<td>Boolean flag set to indicate if object has valid ASIT loaded.</td>
</tr>
<tr>
<td></td>
<td>true = valid ASIT found</td>
</tr>
<tr>
<td></td>
<td>false = no ASIT found</td>
</tr>
</tbody>
</table>

Example

```javascript
If(tmpObj.hasASI)
    var tmpObj = tmpObj.ASI["My ASI Field"]; //List all ASI
If(tmpObj.hasASIT)
    For(a in tmpObj.ASI)
        logDebug(a + " : " + tmpObj.ASI[a]);
```
Example

//List all ASI Table values
If(tmpObj.hasTables)
    for(table in tmpObj.hasASIT)
        for(row in tmpObj.hasASIT[table])
            for(col in tmpObj.hasASIT[table][row])
                logDebug(table + " : " + row + " : " + col + " : " + tmpObj.hasASIT[table][row][col];

guideSheetObject

A helper object which represents the data that the guidesheet contains. You can retrieve these objects for a given inspection. Each guide item is represented as a separate object.

You can use this object with Inspection Guidesheets to simplify the interaction with the various guidesheet items and to expose the Applications Specific Information and Application Specific Information Tables for use.

Version

2.0

Constructors

Loads the guideSheetObject for the provided guideSheet and guideSheetItem.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gguidesheetModel</td>
<td>ggguideSheetModel</td>
<td>Guidesheet object to retrieve.</td>
</tr>
<tr>
<td>gguidesheetItemModel</td>
<td>ggguideSheetModel</td>
<td>Guidesheet item to retrieve.</td>
</tr>
</tbody>
</table>

Example

var guideObj = guideSheetObject(guideSheet, guideItem);

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsType</td>
<td>Guidesheet type.</td>
</tr>
<tr>
<td>gsSequence</td>
<td>Guidesheet system sequence number.</td>
</tr>
<tr>
<td>gsDescription</td>
<td>Guidesheet description.</td>
</tr>
<tr>
<td>gsIdentifier</td>
<td>Guidesheet identifier.</td>
</tr>
<tr>
<td>item</td>
<td>Guidesheet item model object.</td>
</tr>
<tr>
<td>text</td>
<td>Guidesheet item text identifier.</td>
</tr>
<tr>
<td>status</td>
<td>Guidesheet item status value.</td>
</tr>
</tbody>
</table>
Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadInfo</td>
<td>This method populates the info parameter with the Application Specific Information contained in the guidesheet item model.</td>
</tr>
<tr>
<td>infoTables</td>
<td>This method populates the infoTables parameter with the Application Specific Information Table data that the guidesheet item model contains.</td>
</tr>
</tbody>
</table>

Constructor

Populates licenseProfObject with the license number and license type.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>licnumber</td>
<td>string</td>
<td>License number to retrieve. This number is the RSTATE_LIC value.</td>
</tr>
<tr>
<td>ictype</td>
<td>string</td>
<td>License type to retrieve.</td>
</tr>
</tbody>
</table>
Example

```javascript
var myLic = licenseProfObject("1234","Business");
```

## Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribs</td>
<td>An associate array populated with all the valid licensed professional attributes. When valid attributes exist the validAttrs flag sets to true indicating values are available. Use the <code>getAttribute</code> and <code>setAttribute</code> methods to access the licensed professional attribute instead of directly accessing the attribs parameter.</td>
</tr>
<tr>
<td>infoTables</td>
<td>This parameter exposes the people info tables multiple dimension array of the following format. <code>infoTables[tableName][row][column]</code> To access the value of this field you must use the <code>getValue()</code> for the column and to set the value you must use the <code>setValue(val)</code>. To add or delete rows please review the methods section for <code>addTableRow()</code>, <code>removeTable()</code>, and <code>removeTableRow()</code></td>
</tr>
<tr>
<td>refLicModel</td>
<td>This parameter loads on object creation and provides direct access to the licensed professional model.</td>
</tr>
<tr>
<td>valid</td>
<td>Boolean flag set to indicate if object has a valid reference license professional loaded. true = valid professional found false = no professional found</td>
</tr>
<tr>
<td>validAttrs</td>
<td>Boolean flag set to indicate if object has valid reference licensed professional attributes loaded. true = valid attributes found false = no attributes found</td>
</tr>
<tr>
<td>validTables</td>
<td>Boolean flag set to indicate if object has valid people info tables loaded true = valid tables found false = no tables found</td>
</tr>
</tbody>
</table>

### Example (attribs)

```javascript
if(myLic.validAttrs)
    var myValu = myLic.attribs["Is Valid Business?"];

//List attributes
if(myLic.validAttrs)
    for(attrib in myLic.attribs)
        logDebug(attrib + " : " + myLic.attribs[attrib]);
```

### Example (infoTables)

```javascript
//get value
myLic.infoTables["Codes"][0]["Type"].getValue();
```
//set value
myLic.infoTables["Codes"] [0]["Type"].setValue("Type III");

//list all values
If (myLic.validTables)
  for (table in myLic.infoTables)
    for (row in myLic.infoTables[table])
      for (col in myLic.infoTables[table][row])
        logDebug(table + " : " + row + " : " + col + " : " + myLic.infoTables[table][row][col].getValue();

Example (refLicModel)
  myLic.refLicModel.getLicenseType();

Example (valid)
  var myLic = licenseProfObject("1234","Business");
  if (myLic.valid)
    //do actions

Methods

addTableFromASIT

This method copies ASI Tables to reference licensed professional people info tables. This method attempts to add all rows from the ASI Table array to the people info table array for all matching columns.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>Name of people info table.</td>
</tr>
<tr>
<td>ASITArray</td>
<td>ASIT Array</td>
<td>ASI table array that master script loads.</td>
</tr>
</tbody>
</table>

Return

If ASI Table loads successfully into the people info tables, the method returns true. If the load fails the method returns false.

Example

  myLic.addTableFromASIT("myTable", CERTIFICATIONS);

addTableRow

Add a new row to the people info table utilizing an associative string array.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>Name of people info table.</td>
</tr>
<tr>
<td>valueArray</td>
<td>string array</td>
<td>Associative string array where the index name is the column name to load.</td>
</tr>
</tbody>
</table>

Example

```javascript
var newRow = new Array();
newRow["Column1"] = "A";
newRow["Column2"] = "B";
myLic.addTableRow("myTable", newRow);
myLic.updateRecord();
```

**copyToRecord**
Copies the current reference licensed professional to the specified record id.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId</td>
<td>CapIDModel</td>
<td>Record to copy the licensed professional to.</td>
</tr>
<tr>
<td>replace</td>
<td>boolean</td>
<td>Flag if existing LP should be replace if found.</td>
</tr>
</tbody>
</table>

Example

```javascript
myLic.copyToRecord(capId, true);
```

**disable**
Disables the licensed professional

Parameters
None

**enable**
Enables the licensed professional

Parameters
None

**getAssociatedRecords**
Retrieves all records associated to the reference licensed professional in an array.
Parameters
None

Example
```
var capArray = myLic.getAssociatedRecords();
```

`getAttribute`
Get method for getting a licensed professional attribute value.

Parameters
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeName</td>
<td>string</td>
<td>Reference license professional attribute name.</td>
</tr>
</tbody>
</table>
```

Notes
Method handles error checking. Use this method instead of directly accessing the parameter.

Example
```
var val = myLic.getAttribute("myValue");
```

`getMaxRowByTable`
Gets the max row number for a people info table.

Parameters
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>People info table name to get the max row from.</td>
</tr>
</tbody>
</table>
```

Return
Returns -1 if no rows exist.

`refreshTables`
Refreshes the people info table arrays in the object with the data found in database.

Parameters
None

`removeTable`
Removes all rows from a people info table.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>People info table name to remove.</td>
</tr>
</tbody>
</table>

*removeTableRow*

Removes provided row index from provided table.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>People info table name to remove row from.</td>
</tr>
<tr>
<td>rowIndex</td>
<td>long</td>
<td>Row index to remove.</td>
</tr>
</tbody>
</table>

Return

If method removes the row, returns true. Otherwise, returns false.

*setAttribute*

Sets a reference license professional attribute to the provided value and performs error checking.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeName</td>
<td>string</td>
<td>Reference license professional attribute name.</td>
</tr>
<tr>
<td>attributeValue</td>
<td>string</td>
<td>Reference license professional attribute value to set.</td>
</tr>
</tbody>
</table>

Return

If method sets value, returns true. Otherwise, returns false.

*Example*

```java
If( myLic.setAttribute("myValue","newValue") )
logDebug("Value Updated");
```

*setDisplayInACA4Table*

Sets the flag to display the reference people info table in Accela Citizen Access.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>Name of the people info table.</td>
</tr>
<tr>
<td>visibleFlag</td>
<td>string</td>
<td>Valid flag values are Y to display the table in Accela Citizen Access or N to hide the table from Accela Citizen Access.</td>
</tr>
</tbody>
</table>
**setTableEnabledFlag**

Sets the enabled flag displayed on the people info tables to yes or no for the provided table row.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableName</td>
<td>string</td>
<td>People info table name to remove row from.</td>
</tr>
<tr>
<td>rowIndex</td>
<td>long</td>
<td>Row index to remove.</td>
</tr>
<tr>
<td>isEnabled</td>
<td>boolean</td>
<td>Enabled flag.</td>
</tr>
</tbody>
</table>

**Return**

Returns true if update is successful.

**Example**

```java
myLic.setTableEnabledFlag("myTable", 0, false);
```

**updateFromAddress**

This method updates the reference professional with the address information from the provided record.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId</td>
<td>CapIDModel</td>
<td>Record to get the address information from.</td>
</tr>
</tbody>
</table>

**Return**

If update is successful the method returns true, otherwise the method returns false.

**Notes**

The method first attempts to use the primary address. If no primary address exists the method selects the first address available on the Record.

If the method finds an address the method then attempts to copy the Address Line 1, Address Line 2, City, State, and Zip to the reference licensed professional. In the event an Address Line 1 is not available it attempts to create the line one by concatenating the house number, street direction, street name, street suffix, unit type, and unit number.

**updateFromRecordContactByType**

This method attempts to update the contact information on a reference licensed professional from a record contact.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId</td>
<td>CapIDModel</td>
<td>Record to get the contact information from.</td>
</tr>
<tr>
<td>contactType</td>
<td>string</td>
<td>Contact type to search, use &quot;&quot; for primary.</td>
</tr>
<tr>
<td>updateAddress</td>
<td>boolean</td>
<td>Set to true to update address information.</td>
</tr>
<tr>
<td>updatePhoneE</td>
<td>boolean</td>
<td>Set to true to update phone information and email information.</td>
</tr>
</tbody>
</table>

Return

If the update is successful the method returns true. If the update fails it returns false.

Notes

To attempt to use the primary contact use an empty string ("") from the contact type. If you provide a contact type and there are multiple with the same contact type, the method uses the first occurrence of the contact type in the event.

When found the method updates the first, middle, last, and business name on the reference licensed professional with the first, middle, last, and business name of the contact record.

If the updateAddress flag is true then the method attempts to copy the address line 1, address line 2, address line 3, city, state, and zip from the contact record to the associate fields of the reference licensed professional.

If the updatePhoneEmail flag is true then the method also attempts to copy the phone1, phone2, phone3, email, and fax to the associate fields on the reference licensed professional record.

updateFromRecordLicensedProf

This method attempts to update the reference licensed professional utilizing a transactional licensed professional.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capId</td>
<td>CapIDModel</td>
<td>Record to get the license professional information from.</td>
</tr>
</tbody>
</table>

Return

If the update is successful the method returns true. If the update fails it returns false.

Notes

This method searches the provided record for a transactional license professional of the same number and the same type. If the method finds a match, the method attempts to copy all licensed professional information from the transactional record to the reference record.
**updateRecord**

This method commits all changes made to the reference licensed professional object to the database.

**Parameters**

None

**Return**

If the update is successful the method returns true. If the update fails it returns false.

**Notes**

If you do not invoke this method, you lose all updates made to the licensed professional prior to the last update.

**Example**

```javascript
myLic.updateRecord();
```

**licenseObject**

This function creates a helper object that you can use to view and modify license information and expiration information.

**Version**

1.6

**Constructors**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>licNumber</td>
<td>string</td>
<td>State license number of the reference licensed professional to link to the license object.</td>
</tr>
<tr>
<td>capId (optional)</td>
<td>CapIDModel</td>
<td>Record ID to use for the license object. Identifies the record from which to load renewal information.</td>
</tr>
</tbody>
</table>

**Notes**

This constructor populates the licenseObject for the license number specified and the currently loaded capId. If licNumber has a value, the helper object attempts to replicate changes to a reference license professional, as well as the record.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refProf</td>
<td>The referenced licensed professional.</td>
</tr>
<tr>
<td>b1Exp</td>
<td>Contains the b1 record (renewal status on record).</td>
</tr>
<tr>
<td>b1ExpDate</td>
<td>Returns the license expiration date in mm/dd/yyyy format (read only).</td>
</tr>
<tr>
<td>b1ExpCode</td>
<td>Returns the expiration code.</td>
</tr>
<tr>
<td>b1Status</td>
<td>Returns the license renewal status (read only).</td>
</tr>
<tr>
<td>refExpDate</td>
<td>Returns the license professional expiration date in mm/dd/yyyy format (read only).</td>
</tr>
<tr>
<td>licNum</td>
<td>The license number.</td>
</tr>
</tbody>
</table>

Example

```javascript
var licObj = licenseObject("1234");
```

Methods

**getCode**

Gets the expiration status of the record.

Parameters

None

Return

Returns the expiration code configured for the license.

Example

```javascript
var licObj = licenseObject("1234");
var code = licObj.getCode();
```

**getStatus**

Gets the expiration status of the record.

Parameters

None

Return

Returns the expiration status
Example

```javascript
var licObj = licenseObject("1234");
var status = licObj.getStatus();
```

**setExpiration**

Sets the expiration date on the license record and associate reference license professional to the provided value.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expDate</td>
<td>string</td>
<td>Expiration date in string format.</td>
</tr>
</tbody>
</table>

**Example**

```javascript
licObj.setExpiration("01/01/2020");
```

**setIssued**

Sets the issued date on the license record and associate reference license professional to the provided value.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>issDate</td>
<td>string</td>
<td>Issued date in string format.</td>
</tr>
</tbody>
</table>

**Example**

```javascript
licObj.setIssued("01/01/2000");
```

**setLastRenewal**

Sets the renewed date on the license record and associate reference license professional to the provided value.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>renewDate</td>
<td>string</td>
<td>Renewed date in string format.</td>
</tr>
</tbody>
</table>

**Example**

```javascript
licObj.setLastRenewal("01/01/2000");
```

**Task**

Defines the task object for use by task functions, loadTasks for example.
### Parameters

<table>
<thead>
<tr>
<th>status</th>
<th>comment</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>statusdate</td>
<td>process</td>
<td>processID</td>
</tr>
<tr>
<td>step</td>
<td>active</td>
<td></td>
</tr>
</tbody>
</table>
This following example illustrates how to define an expression that uses a state agency web service to validate a licensed professional.

In the example, you manually enter an EMSE script in the script mode window of Expression Builder for a selected Professional Execute Fields. The EMSE script verifies the license type and license number on a new application. The expression also updates the licensed professional license to the most current information, for example, status, expiration date, and address.

### Note:
You must build or deploy a Web Service Stub to interface with the external Web Services. For more information about building a Web service stub for your agency, see the Creating an External Web Service Stub (08ACC-04275) – Accela Automation Technical Bulletin.pdf.

**To verify the licensed professional**
1. Navigate to the Building portlet.
2. Select a permit.
3. Click the Professionals tab.
   
   *Accela Automation displays the Professionals tab in the detail portlet*
4. Click the New button to enter the licensed professional information.
5. Selects License Type from the drop-down list, or enter a License #.
   
   If the license number and license type validate successfully, Accela Automation creates the new record. If the license number and license type are not valid, Accela Automation displays the EMSE error message on the Professionals tab.

**To create the EMSE script to validate licensed professionals**
1. Create a New expression and navigate to the Expression Name field.
   
   *Accela Automation displays the New Expression fields where you define the criteria.*
2. Enter an **Expression Name**. This scenario uses **Licensed Professional Validation**.

3. Select **Record Detail** from the **Target Portlet** drop-down list.
   
   This step specifies that the expression takes effect in the Record detail portlet or application for the selected record type. This scenario uses the record type **Building/Building Permit/Commercial/All Categories**. In general, you do not need to perform this step.

4. Select **Script Mode** in the **Edit Mode** section.
   
   *Accela Automation re-populates the page to display the Script fields.*
5. Select **Professional** from the **Target Portlet** drop-down list.
   This step specifies that the expression takes effect in the Professional portlet for the selected record type.

6. In the **ASI Group** field, select the group that contains the record type and the fields for which you want to create an expression.

7. Use the Variables section to specify the fields affected by the expression.
Accela Automation displays ASI, Professional, Record Detail, and Session Variables in the Variables field.

8. Click the **Execute Fields** list picker.

A pop-up window displays the Execute Fields list.

9. Expand **Professional** and click the **License #** and **License Type** options.

Accela Automation loads the License # and License Type options in the Execute Fields list.

10. In the Script field, enter the EMSE script. The script for this scenario is:

```javascript
/*----------------------------------------------------------------/
| Program : LicProfAddBefore.js
| Event   : LicProfAddBefore
|/

var LicProfModel = aa.env.getValue("LicProfModel");
var licenseType = LicProfModel.licenseType;
```
var licenseNbr = LicProfModel.licenseNbr;

var licenseValidateReturnCode = "0";
var licenseValidateReturnMessage = "Follow Licenses are invalid:";

if (!validateLicense(licenseType, licenseNbr)) {
    licenseValidateReturnCode = "-1";
    licenseValidateReturnMessage += "<br>";
    licenseValidateReturnMessage += " * License type: " + licenseType;
    licenseValidateReturnMessage += " , License number: " + licenseNbr;
}

// check whether something wrong
if (licenseValidateReturnCode != "0") {
    aa.env.setValue("ScriptReturnCode", licenseValidateReturnCode);
    aa.env.setValue("ScriptReturnMessage", licenseValidateReturnMessage);
}

// check whether the licenseType and licenseNbr is valid.
function validateLicense(licenseType, licenseNbr) {
    var accelawsUrl = 'https://www4.cbs.state.or.us/exs/bcd/acela/ws/accelaws.cfc?method=lic_valid&returnformat=json';
    var client = aa.httpClient;

    // set url parameters
    var params = client.initPostParameters();
    params.put('p_lic_type', licenseType);
    params.put('p_lic_num', licenseNbr);

    // do validate via web service
    var scripResult = client.post(accelawsUrl, params);

    // check the return value
    if (scripResult.getSuccess()) {
        var resultString = String(scripResult.getOutput());
//Convert to jsonObject
var result = eval("("+resultString+"))
var valid = String(result["VALID"])

if (valid.toUpperCase() == "TRUE")
{
    return true;
}
else
{
    aa.print("ERROR: Failed to validate license: " + scripResult.getErrorMessage());
    return false;
}

return false;

11. **Click the Validate button** to check the EMSE script for errors.

12. **Click the Submit button.**
The following sections introduce the basic concepts that you need to write scripts and understand scripts that others write. Accela uses JavaScript as the basis for the Accela Automation scripting engine. Accela has extended pure JavaScript to include features that allow you to interact directly with Accela Automation in your scripts.

Topics:

- Understanding Scripts
- Using Variables
- Using Expressions
- Controlling What Happens Next
- Using Functions
- Using Objects, Properties, and Methods

**Understanding Scripts**

To help you understand scripts, this section uses an example: a complete script that responds to a specific event. This section also includes information on writing scripts from scratch and a simple tool that can help writing scripts easier.

Topics:

- Our First Example
- Writing And Testing Our First Script
- Using Jext To Make Writing Scripts Easier

**Our First Example**

This example is a complete script that responds to an `InspectionScheduleAfter` event by inserting a new smart notice with information about the scheduled inspection. The example is several lines long, and contains comments at lines 1, 6, 9, 16, 19, and 26 that briefly explain what is happening in each section in the script.

Note that each line that begins with a comment starts with a double slash. The double slash tells Accela Automation to ignore that line. It is good practice to add comments to your scripts.
//Get the permit id.
permitId1 = aa.env.getValue("PermitId1");
permitId2 = aa.env.getValue("PermitId2");
permitId3 = aa.env.getValue("PermitId3");

//Prepare the smart notice label.
noticeLabel = "Inspection schedule";

//Get some information about the scheduled inspection.
numberOfInspections = 
    aa.env.getValue("NumberOfInspections");
inspectionType = aa.env.getValue("InspectionType");
inspectionScheduleMode = 
    aa.env.getValue("InspectionScheduleMode");
inspectionDate = aa.env.getValue("InspectionDate");
inspectionTime = aa.env.getValue("InspectionTime");

//Prepare label for smart notice.
noticeLabel = "Inspection Scheduled!";

//Prepare the text of the new smart notice.
oticeText = numberOfInspections + " Inspection(s) " + 
    inspectionType + " " + 
    inspectionScheduleMode + "d on " + 
    inspectionDate + " " + 
    inspectionTime + ".";
Another important feature of our first example is that every line that is not a comment line seems to end with a semicolon ( ; ). We can see that lines 2, 3, 4, 7, 10, 11, 12, 13, 14, 17, 24, and 28 end with a semi-colon. The semi-colon tells Accela Automation that it has reached the end of a command, and should execute it. If we look at line 27 we see that it does not end with a semi-colon, but ends with a comma. The comma means that the command continues on to the next line. We see that line 28 ends with a semi-colon. The semi-colon means that there is one command that begins on line 27 and ends on line 28. If we look at lines 20, 21, 22, 23, and 24 we can see that these lines comprise one big command split across five lines to make it easier to read.

If you forget to end your commands with a semi-colon, Accela Automation is forgiving and the script may run correctly, but it is always good practice to end your commands with a semi-colon when necessary. Some kinds of commands do not have to end in a semi-colon. We investigate what kinds of commands end with a semi-colon and what kinds do not in later sections of this document.

Writing And Testing Our First Script

While you are learning to write scripts, it is useful to be able to test simple scripts and see the results immediately without having to attach your script to an event. To do this we use the Script Test page. For information on testing scripts, see Chapter 6: Script Testing on page 87. When we cover a new sample script in this document you can copy the script and paste it into the Script Text field on the Script Test page. After you have pasted the text of the script into the form, you can click the Submit button to run the script and view the result.

You can also type this script by hand. Typing the script can be a very helpful learning aid when you start with script writing. However, if you make a mistake in typing you receive a message telling you that there is a problem with your script. When you get a message telling you about the problem, check your script to make sure that it matches the example. Another good tip for learning to write script is to try to modify the sample script to see what happens.

Here is our first sample script for testing:

```
aa.print('Hello World.');</
```

If you have read the earlier sections of this document, you may recognize our sample script. The output of this script is

Hello World.

Let us look at exactly what is happening in our sample script. The script has one line that ends with a semi-colon just like the lines in the first example. The line begins with the two letters 'aa'. These two letters stand for Accela Automation. This line begins with 'aa' because we are going to tell Accela Automation to do something for us. A dot follows the 'aa'. The dot connects the word 'aa' to the word 'print', which means that the word 'print' is a method of the 'aa' object.
An object is a group of associated actions or functions. The previous sample script calls the object *aa*. The *aa* object can retrieve data from your database and then use the data to perform tasks. A simple example of the tasks that the *aa* object is capable of performing is the *print* task, but there are many tasks that the *aa* object can perform.

Objects can retrieve information, change stored information, and do many other things for you. Writing and implementing scripts is how we get the *aa* object (or any other object in JavaScript) to do work with raw data. When we choose a script to initialize in Accela Automation, we are essentially giving a command to our machine. We call the commands we give methods. We learn more about objects and methods in the section *Using Objects, Properties, and Methods* on page 265.

We now know that this line is asking Accela Automation to print something for us. After the word 'print' there is a left parenthesis. After the words 'Hello World.' there is a right parenthesis. When you are writing scripts, a you must follow a method name by a pair of parentheses. Sometimes there are things in between the parentheses, called parameters. Parameters tell a method how to do its job. When we look at the characters between the parentheses we see a single quote followed by the words Hello World, followed by a period, followed by another single quote.

Strings are words, numbers, or punctuation marks that appear between single or double quotes. We know that this script is telling Accela Automation to print the string 'Hello World.', which is exactly what appears in the *Script Output* box when you use the *Script Test* page to run this script. For practice, try to change the string passed to the print method of the *aa* object and see what happens. You can also try to add a second line after the first one with prints out something different, and see what happens then.

**Using Jext To Make Writing Scripts Easier**

The Jext editor is a freely available text editor with many features that make editing scripts easier. Here is a screenshot of Jext in action:
For Jext to recognize your scripts as JavaScript files you must save them with the extension ".js". When opened in Jext, the editor highlights the script text in different colors to make it easier to read. Other useful features include a counter in the bottom left that tells you what line of the script that you are on, and a file explorer in the along the left side to make finding and opening files easier.

Using Variables

A variable is a placeholder in your script that you use to store a value. A variable always has a name that you can use to represent its value. You can use the name of a variable to store something or retrieve it. Variable names, also known as identifiers, must begin with an underscore "_", or letter that you can follow with letters, underscores, and digits. Here are some sample variable names:

myVariable
Number_Of_Inspections
Test12

In this document, we always begin our variable names with a lower case letter, and capitalize the first letter of each subsequent word in the variable name. We recommend, but do not require, this method of variable naming. Let us look at an example of putting a value into a variable.
myVariable = 12;
aa.print(myVariable);

This script displays this output:

12

The first line of this script puts the value 12 into `myVariable`. The second line uses the print method of the `aa` object that we investigated earlier, but instead of putting a string of characters in between the left and right parentheses of the print method, we have put the name of our variable. This usage tells the print method to display whatever value `myVariable` contains. We can also put strings of characters into variables. Here is a script that uses a string as the value of `myVariable`:

```javascript
myVariable = "Hello World.";
aa.print(myVariable);
```

This script displays this output:

Hello World.

Another technique that we can use is to assign the value of one variable to the value of a different variable. Here is an example:

```javascript
firstVariable = 101;
secondVariable = firstVariable;
aa.print(secondVariable);
```

This script displays this output:

101

In this script, we assign the value 101 to `firstVariable`, and then assign the value of `firstVariable` to the value of `secondVariable`. Finally, the script prints out the value of `secondVariable`. When you use `firstVariable` on the right side of an equals sign, we call this evaluating a variable. To evaluate a variable is to retrieve its value. We are also evaluating a variable when we pass `secondVariable` to the print method of the `aa` object. One might ask, what happens if we try to evaluate the value of a variable to which you did not assign a value. For example,

```javascript
firstVariable = 101;
aa.print(secondVariable);
```

This script displays this output:

An error occurred while running your script.
ErrorType: org.mozilla.javascript.EcmaError
Error Detail:
undefined: "secondVariable" is not defined. (script; line 1)

In this example, we removed the second line of the script. This means there is no line in the script that assigns a value to `secondVariable`, but on the last line of the script, we try to print out the value of this variable. You receive an error if the script tries to evaluate a variable without an assigned value. When we try to execute this script we see an error message in the output box. The error message tells us that `secondVariable` is not defined.

There are many potential causes for errors. Accela Automation error messages provide meaningful information to help you solve problems with your scripts. Script writers frequently
misspell variable names, so it is a good idea to look carefully at your scripts, check for misspellings, missing semi-colons, and missing parentheses.

**Topics:**
- Numbers
- Strings
- True and False
- Arrays
- The Special Value “null”
- Objects

**Numbers**

There are many kinds of numbers you can assign to a variable. We do not provide an exhaustive list here, but we do go over some of the common kinds of numbers that we deal with. Numbers can be positive, negative, zero, integers, decimals, and have exponents and other characteristics. Here are some sample numbers:

```
12
0
-2
1.28
0.94871
-54.09
3.1E12
5E-14
```

The last two sample numbers have an exponent. You probably do not need to use the exponential form of a number, but if your script ever encounters a very large or very small number, then tries to print that value, it may appear in the exponential form. The number after the E is the number of places to the right that you should move the decimal point to get the non-exponential form of the number. If the number after the exponent is negative, it represents the number of places to the left that you need to move the decimal point to get the non-exponential form. If you find that you need to use a kind of number not mentioned here, we encourage you to look up more information on numbers in a JavaScript reference text.

**Precision of Numbers**

The precision of a number is the number of decimal digits in that number. You can use mathematical expressions (see Mathematical Expressions on page 255) or functions (see Using Functions on page 264) in your scripts to get various numbers. Because Accela Automation applies the Java class BigDecimal to control the precision of results for mathematical functions, but not for mathematical expressions. If your expected result is a number with decimal part, use mathematical functions instead of expressions to ensure the precision of the result.

With mathematical functions including add, subtract, multiply, divide and round, the default DEF_DIV_SCALE value is 2. You can customize DEF_DIV_SCALE.
For example:

```javascript
aa.print(123.3 / 100); // The precision in the expression is out of control.
aa.util.multiply(4.015, 100); // result: 401.5
var DEF_DIV_SCALE=3
aa.util.divide(123.3, 100) //default scale=2, result: 1.23
aa.util.divide(123.3, 100, DEF_DIV_SCALE) // result: 1.233
aa.util.round(12.1542, 1) // result: 12.2
aa.util.round(12.1542, 2) // result: 12.15
```

Strings

Strings comprise a number of characters in between a pair of double or single quotes. Here are some examples:

```
"Hello World."
"This string is surrounded by double quotes."
'Ok'
'A'
'!
```

These examples show that a string can be one or many characters long, surrounded by single or double quotes. The following example shows that a string can consist of digits:

```
"12345"
```

Escape Characters

The next example shows a string with a special escape character inside:

```
"Four score and \n seven years ago."
```

The escape character is the backslash "\" character that you follow with the "n" character. This special character means go to the next line down when printing out this string. This special character calls the new line character. Here is a sample script that prints out this example:

```javascript
aa.print("Four score and \n seven years ago.");
```

This script displays this output:

```
Four score and
seven years ago.
```

All special characters begin with a backslash. Let us look at the most commonly used special characters:
Because we use the single and double quotes to determine the ends of the string, we can only put them into a string by using the special character that represents them. Because we use the backslash to start a special character, we must use a double to put a backslash into a string. Other special characters allow you to insert characters from foreign languages, insert special symbols, and insert other character types. If you find that you need to use these other special characters, consult a standard JavaScript reference book.

### True and False

Now we encounter a new kind of variable type that we have not seen before. We call this type of variable Boolean. These variables can only hold either true or false. Here is an example:

```javascript
aTrueVariable = true;
aFalseVariable = false;
```

In this example, we assign the `true` value to one variable and the `false` value to the other variable. Unlike string values, you do not enclose the `true` and `false` values in quotes. You can assign these two words to variables as special values. You typically use Boolean variables as parameters for methods of objects or for controlling what happens next in your script. We see some examples of how to use Boolean variables later in this document.

### Arrays

An array is a special kind of variable that hold a list of values, and allows you to retrieve and store each of the values separately. Here is an example of creating and using an array:

```javascript
myVar = new Array();
myVar[1] = "Hello";
myVar[2] = "World";
aa.print(myVar[1]);
aa.print(myVar[2]);
```

This script displays this output:

```
Hello
World
```

The first line of the example tells Accela Automation that `myVar` is of the special Array type, that is, assign a new empty Array object to `myVar`. So we can say that `myVar` contains an array object. The second line of the script puts the string "Hello" in number one position of the array.

### Table 25: Common Special Characters in Scripting

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New Line</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
</tbody>
</table>

Because we use the single and double quotes to determine the ends of the string, we can only put them into a string by using the special character that represents them. Because we use the backslash to start a special character, we must use a double to put a backslash into a string. Other special characters allow you to insert characters from foreign languages, insert special symbols, and insert other character types. If you find that you need to use these other special characters, consult a standard JavaScript reference book.
The third line put the string "World" in the number two position of the array. The fourth line prints out the value stored in the number one position of the array. The fifth line prints out the value stored in the number two position of the array. You can store and retrieve values in any position of the array you like. There is also a position zero, and negatively numbered positions, but most of the time you only use the positively numbered positions.

You can find out how long an array is by using a property of all arrays. Here is an example:

```javascript
myVar = new Array();
myVar[1] = "Hello";
myVar[2] = "World";
aa.print(myVar.length);
```

This script displays this output:

```
3
```

Note that we are printing out something called `myVar.length` on the last line of the script. Whenever we need to know the length of an array we can always put `.length` after it to get the length. Length is a property of our array. We learn about more arrays and how properties work in the section Objects, Methods, and Properties later in this document. You may ask, if we assigned something to position one and two then why is `myVar.length` returning three? The answer is that we count position zero in the length of the array. Let us modify this example slightly and see what happens:

```javascript
myVar = new Array();
myVar[1] = "Hello";
myVar[4] = "World";
aa.print(myVar.length);
aa.print(myVar[2]);
```

This script displays this output:

```
5
undefined
```

We changed the third line to put a value in position four rather than in position two. The result is that the total length of the array is now five. There are empty elements in the array at positions 0, 2, and 3. On the last line of the script we tried to evaluate `myVar[2]` and received a special value called `undefined` that tells us that we never put anything into the array at that position.

### The Special Value “null”

The word `null` in a script means nothing. Some methods of some objects allow you to pass `null` in as the value of a parameter, usually to indicate that you do not want to send in any meaningful value for that parameter. We see a little later that Accela Automation may return `null` to your script when you try to retrieve some information from Accela Automation, usually to indicate that no information is available. We see some of the specific places that use `null` later in this document.
Objects

A variable can also contain an object. An object is a self-contained module of data and its associated processing. Let's look at an example:

```javascript
myVar = aa;
myVar.print("Hello World.");
```

This script displays this output:

```
Hello World.
```

Here we can see that we assigned the `aa` object to the `myVar` variable on the first line of the script. This assignment means that `myVar` contains the `aa` object. We then used `myVar` to execute the `print` method of the `aa` object. Here is another example:

```javascript
myVarOne = aa;
myVarTwo = aa;
myVarOne.print("Hello");
myVarTwo.print("World");
```

This script displays this output:

```
Hello
World
```

Notice that we assign the `aa` object to both of the variables in this script, and then call the `print` method on each one. This example shows us that what really happens when you assign an object to a variable is that the variable is only pointing at the object. The variable becomes like a handle to the object that you can use to manipulate it. You can have many variables that all point at the same object.

We look deeper into object in the section *Object, Methods, and Properties* later in this document. We learn more about variables as we learn about other aspects of writing scripts.

Using Expressions

An expression is a compound value that evaluates to determine a result. We have already encountered one example of an expression called an assignment statement. A simple expression uses an equals sign to assign a value to a variable. Expressions can contain operators that modify or join the values of some variables to come up with a final result. In this section, we look at several different forms of expressions.

Topics:
- Mathematical Expressions
- String Expressions
- Boolean Expressions
- Relational Operators
- Special Operators
- Operator Precedence
Mathematical Expressions

The kinds of expressions that most people are familiar with are arithmetic expressions. Here is an example of an expression that adds two numbers together and assigns the result to a variable:

```javascript
myVar = 2 + 2;
aa.print(myVar);
```

This script displays this output:

4

In this example the `+` operator joins two numbers. Here is another example:

```javascript
myVar = 1
myVar = myVar + 2;
aa.print(myVar);
firstVar = 7;
secondVar = 5;
myVar = firstVar + secondVar;
aa.print(myVar);
```

This script displays this output:

3
12

On the second line of this script, we add two to the current value of `myVar` and put the resulting new value back into `myVar`. On the seventh line of this script we add two variables to come up with a result that we place in `myVar`. There are operators for addition, subtraction, multiplication, division, and many more. We do not cover every arithmetic operator here, but here are six operators arithmetic operators you can use:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition.</td>
<td>myVar = 2 + 2; myVar now contains 4.</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction.</td>
<td>myVar = 4 - 2; myVar now contains 2.</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication.</td>
<td>myVar = 2 * 3; myVar now contains 6.</td>
</tr>
<tr>
<td>/</td>
<td>Division. Be careful not to divide by zero or you get an error.</td>
<td>myVar = 6 / 2; myVar now contains 3.</td>
</tr>
<tr>
<td>%</td>
<td>Modulus. The “remainder” operator. Tells you the left over amount, after division.</td>
<td>myVar = 5% 3; myVar now contains 2.</td>
</tr>
<tr>
<td>- (negation)</td>
<td>Negation. The “unary” operator. Takes whatever value you put immediately to the right of it, and reverses its sign.</td>
<td>someVar = 3; myVar = -someVar; myVar now contains -3.</td>
</tr>
</tbody>
</table>
You can also use more than one operator at a time in an expression. For example:

```javascript
firstVar = 7;
secondVar = 5;
myVar = firstVar - 2 + secondVar + 7;
aa.print(myVar);
```

This script displays this output:

`17`

When using a single line that contains several operators it is important to remember that, just as in your grade school mathematics classes, some operators have a higher precedence than others do. For example:

```javascript
myVar = 2 + 6 / 3;
```

myVar now contains 4.

The result of the expression in this example was four because the division operator has a higher precedence than the addition operator does. All operators, including the non-arithmetic operators, have a certain level of precedence. When two operators in the same expression have the same level of precedence, Accela Automation evaluates them in left to right order. For example:

```javascript
myVar = 6 * 3 / 3;
```

myVar now contains 6.

You can use parentheses to change the order in which to evaluate an expression:

```javascript
MyVar = (2 + 6) / 4;
```

MyVar now contains 2.

In general, Accela Automation evaluates everything inside a set of parentheses before anything outside the parentheses.

### String Expressions

String expressions are quite simple. There is only one operator that works on strings. We use the “+” operator for addition and to concatenate two strings together end to end to form a new string. Here is an example:

```javascript
firstVar = “Hello”;
secondVar = “World.”;
thirdVar = firstVar + secondVar;
aa.print(thirdVar);
```

This script displays this output:

```
HelloWorld.
```
Hello World.

You can concatenate more than two strings together:

```javascript
myVar = “Hello” + “to the “ + “world.”;
aa.print(myVar);
```

This script displays this output:

```
Hello to the world.
```

## Boolean Expressions

Boolean expressions always evaluate to either true or false. Table 27: Boolean Operators shows the most common Boolean operators for boolean expressions.

### Table 27: Boolean Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>And</td>
<td>myVar = true &amp;&amp; false; myVar now contains false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
<td>someVar = true; myVar = !someVar; myVar now contains false.</td>
</tr>
</tbody>
</table>

First let us examine the “and” operator “&&”. This operator is true when both of its operands are true, and false the rest of the time. The word “operands” refers to the thing that the operator is operating on. So for example, an inspector has an inspection scheduled for today and the inspector has called in sick. The facts that the inspector has an inspection scheduled and that he has called in sick can be operands of the && operator. Both are true, so the operation returns a result of **true**. Table 28: And Operator Results shows possible results of the “&&” operator.

#### Table 28: And Operator Results

<table>
<thead>
<tr>
<th>Example</th>
<th>myVar Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>myVar = true &amp;&amp; true;</td>
<td>true</td>
</tr>
<tr>
<td>myVar = true &amp;&amp; false;</td>
<td>false</td>
</tr>
<tr>
<td>myVar = false &amp;&amp; true;</td>
<td>false</td>
</tr>
<tr>
<td>myVar = false &amp;&amp; false;</td>
<td>false</td>
</tr>
</tbody>
</table>

You use the “&&” operator most often when you want to find out if two or more things are true at the same time.

Next we examine the “or” operator “||”. You use the “||” operator most often when you want to find out if at least one of two or more things is true. The result is true as long as at least one of the operands is true. You use vertical bar character (also called a pipe), that is on the same key as the backslash character, to type this operator. Press shift backslash to type this character. Table 29: Or Operator Results shows a set of examples for the “||” operator.

#### Table 29: Or Operator Results

<table>
<thead>
<tr>
<th>Example</th>
<th>myVar Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>myVar = true</td>
<td></td>
</tr>
<tr>
<td>myVar = true</td>
<td></td>
</tr>
<tr>
<td>myVar = false</td>
<td></td>
</tr>
<tr>
<td>myVar = false</td>
<td></td>
</tr>
</tbody>
</table>
Finally, we examine the "not" operator "!". The "!" is a unary operator that operates on only one operand. Like the unary minus sign, the not operator reverses the state of the value to which it applies (Table 30: Not Operator Results).

### Table 30: Not Operator Results

<table>
<thead>
<tr>
<th>Example</th>
<th>myVar Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>myVar = true</td>
<td></td>
</tr>
<tr>
<td>myVar = true</td>
<td></td>
</tr>
<tr>
<td>myVar = false</td>
<td></td>
</tr>
<tr>
<td>myVar = false</td>
<td></td>
</tr>
</tbody>
</table>

You can use multiple Boolean operators in a row, and use parentheses to change the order of precedence of Boolean operators, just like arithmetic operators. However, there is an additional aspect of Boolean operators not shared by other operators called “short-circuit evaluation.” Here are two examples of this:

```javascript
myVar = false && ???;
```

myVar contains false no matter what is on the right hand side of the "&&".

```javascript
myVar = true || ???;
```

myVar contains true no matter what is on the right hand side of the "||".

Short-circuit evaluation means that if Accela Automation can determine from the first part of an expression whether the whole expression is going to be true or false it does not bother to evaluate the rest of the expression.

### Relational Operators

Relational operators return either true or false. However, unlike Boolean operators they can take different kinds of operands like numbers and strings. The relational operators are ==, !=, <, >, <=, and >=.

The "equals" operator "==" tells us if two values are the same. See Table 31: Relational Operators.

### Table 31: Relational Operators

<table>
<thead>
<tr>
<th>Example</th>
<th>myVar Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>myVar = (true == true);</td>
<td>true</td>
</tr>
<tr>
<td>myVar = (true == false);</td>
<td>false</td>
</tr>
<tr>
<td>myVar = (false == true);</td>
<td>false</td>
</tr>
</tbody>
</table>
The "==" operator uses two equals signs to avoid confusion with the assignment operator. You usually use the "==" operator to find out if two things are the same. You can compare any two values using this operator. You can find out if a variable has a special value like `null` as in this example:

```javascript
myVar = (someVar == null);
```

The "!=" operator is the opposite of the "==" operator. See Table 32: Relational Operators.

<table>
<thead>
<tr>
<th>Example</th>
<th>myVar Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>myVar = (true != true);</code></td>
<td>false</td>
</tr>
<tr>
<td><code>myVar = (true != false);</code></td>
<td>true</td>
</tr>
<tr>
<td><code>myVar = (false != true);</code></td>
<td>true</td>
</tr>
<tr>
<td><code>myVar = (false != false);</code></td>
<td>false</td>
</tr>
<tr>
<td><code>myVar = (1 != 2);</code></td>
<td>true</td>
</tr>
<tr>
<td><code>myVar = (2 != 2);</code></td>
<td>false</td>
</tr>
<tr>
<td><code>myVar = (&quot;Hello&quot; != &quot;World&quot;);</code></td>
<td>true</td>
</tr>
<tr>
<td><code>myVar = (&quot;Hello&quot; != &quot;Hello&quot;);</code></td>
<td>true</td>
</tr>
</tbody>
</table>

The "<", ">", "<<=", and ">=" operators are useful when comparing two numbers. See Table 33: Relational Operators.

<table>
<thead>
<tr>
<th>Example</th>
<th>myVar Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>myVar = 1 &lt; 2;</code></td>
<td>true</td>
</tr>
<tr>
<td><code>myVar = 2 &lt; 1;</code></td>
<td>false</td>
</tr>
<tr>
<td><code>myVar = 1 &gt; 2;</code></td>
<td>false</td>
</tr>
<tr>
<td><code>myVar = 2 &gt; 1;</code></td>
<td>true</td>
</tr>
<tr>
<td><code>myVar = 1 &lt;= 2;</code></td>
<td>true</td>
</tr>
</tbody>
</table>
Special Operators

We only cover one special operator here. We have already seen this operator when we first investigated creating an array. The "new" operator creates a new object. For example:

```javascript
myVar = new Array();
myVar now contains an Array object.
```

Arrays are really just a special kind of object. We go over more about arrays in a later section. For now, we should take note that the keyword "new" is a special kind of operator that creates a new copy of an object type. In this example the object type was "Array". We learn more about creating objects in the section Objects, Methods, and Properties. There are many special operators that we have not covered. For more information on special operators, consult a JavaScript reference.

Operator Precedence

The operators have the precedence in the order shown.

```
=  
||  
&&  
==  !=  
< > <= >=  
+ -  
* / %  
! - (unary minus)  
new
```

Controlling What Happens Next

There are several tools that you can use to indicate the next step in a script.

Topics:

- if ... else
You use the conditional when you want to perform a set of commands only if something is true. Here is an example:

```javascript
MyVar = 1;
if(myVar > 0) {
    aa.print("Yes.")
}
```

This script displays this output:

Yes.

The conditional begins with the word “if” followed by a pair of parentheses. You can place any expression between these parentheses. A pair of braces “{” and “}” follow the parentheses. The braces contain one or more commands. If the expression between the parentheses evaluates to true then the commands between the braces executes. If the expression between the parentheses evaluates to false then the commands between the braces do not execute. The example script does print out the word “Yes” because it is true that `myVar`, which has the value one, is greater than zero. If you set `myVar` was to negative one, then nothing prints out.

You use the “else” clause to specify what happens when the condition is false. Here is an example:

```javascript
MyVar = 1;
if(myVar > 2) {
    aa.print("Yes.")
} else {
    aa.print("No");
}
```

This script displays this output:

No.

In this example, we have changed the conditional to test if `myVar` is greater than two. Because the condition evaluates to false, the `else` block executes instead of the main block. We use the word “block” to refer to a group of commands between a matching pair of braces. The first block prints “Yes” if the condition is true. The second block prints “No” if the condition is false. Because the condition is false, this example prints out “No”.

You can also create a multi way branch with several blocks. Here is an example:

```javascript
myVar = "Bagels";
if(myVar == "Oranges") {
    aa.print("Fruit.")
} else if(myVar == "Bagels") {
```
aa.print("Cereals.");
} else if (myVar == "Spinach") {
    aa.print("Vegetables.");
} else {
    aa.print("I don’t known what food group that is in.";)
}

This script displays this output:
Cereals.

This script contains several possible blocks that can execute, depending on the value of myVar. Because myVar has the value “Bagels”, the second block executes, and the word “Cereals” prints out. The final else clause is optional.

for

There are several kinds of loops in JavaScript. The for loop allows a script to repeat a set of commands repeatedly until some condition is false. You typically use the for loop when you know how many times you want to repeat the loop. Here is an example:

```
for(i = 1; i < 6; i = i + 1) {
    aa.print("The current value of the loop counter is: " + i);
}
```

This script displays this output:

```
The current value of the loop counter is: 1
The current value of the loop counter is: 2
The current value of the loop counter is: 3
The current value of the loop counter is: 4
The current value of the loop counter is: 5
```

The for loop begins with the word “for” followed by a pair of parentheses that contain three expressions, each separated by semi-colons. After the parentheses are a pair of braces that contain the statements that repeat by the loop. The three expressions in between the parentheses determine how many times the loop repeats. Let us look at these three expressions:

```
i = 1; i < 6; i = i + 1
```

The first expression is i=1. This expression set the value of the variable i to one as you might expect. This first expression executes one time, before Accela Automation executes the body of the loop. The body of the loop is the block, surrounded by a pair of braces that comes right after the parenthesis. The second expression is i < 6. This is the condition of the loop, and you only execute the body of the loop if this condition is true. Accela Automation checks the condition just before the body of the loop executes, and the loop continues to repeat until it is false. The third expression, i = i + 1, tells the loop how to update the loop counter each time you reach the end of the body of the loop. When you reach the end of the body of the loop, Accela Automation executes this statement. In this case, the third expression adds one to the counter. From the output of the example, you can see that each time through the loop the counter value updates
and the counter value prints out. The loop stops when the value of \( i \) reaches six because the second expression is no longer true.

### `while`

The "while" loop is another loop that repeats until its condition is false. You typically use this loop when you do not know how many times the loop executes. Here is an example:

```javascript
myArray = new Array();
myArray[0] = "Oranges";
myArray[1] = "Bagels";
myArray[2] = "Spinach";
i=0;
while(i < myArray.length) {
    aa.print(myArray[i]);
    i = i + 1;
}
```

This script displays this output:

```
Oranges
Bagels
Spinach
```

In this example, you create an array with three elements and you set the loop counter variable \( i \) to zero. The loop begins at the word "while". Next is a pair of parentheses that contain the condition for the loop. While the condition is true, the body of the loop, which comes after the parentheses and is enclosed by a pair of brackets, repeats. We can see two commands inside the body of the loop. The first prints out the value of the array at the position that you indicate by the loop counter. The second line adds one to the loop counter. We need to be careful to remember to always add a line to add to the loop counter to the end of our `while` loop bodies, because if we do not then the loop never stops, and Accela Automation terminates the script after a time-out period has elapsed.

### `do ... while`

This loop also repeats until its condition is false. Use the "do" loop when you want to make sure to execute the body of your loop at least one time even if the condition is false before the loop starts. Here is an example:

```javascript
myArray = new Array();
myArray[0] = "Oranges";
myArray[1] = "Bagels";
myArray[2] = "Spinach";
i=0;
do {
    aa.print(myArray[i]);
    i = i + 1;
} while (i < myArray.length);
```
This script displays this output:

Oranges

In this example we can see that the body of the do loop executes one time even though the condition \( i < 0 \) is false before the loop begins. The example shows that a do loop begins with the word "do" followed by a block, surrounded by braces, for the body of the loop. After the block is the word "while" followed by a pair of parentheses that enclose the condition for the loop. Unlike the other two loops, this last line of the loop, after the parentheses, ends with a semi-colon. Remember to put a command to change the counter in the body of the loop if you are using a counter to control the loop.

### Using Functions

A function is a set of commands, with a name, that you can execute by calling that name and passing in any parameters that the function requires. You usually use functions when you have a set of commands that you want to be able to repeat at different places in your script, rather than at one place like with a loop. Let us look at an example:

```javascript
function timesTen(number) {
    result = number * 10;
    return result;
}
myNumber = timesTen(5);
aa.print(myNumber);
```

This script displays this output:

50

The first four lines of the script create the function. The fifth line stores the function result in a variable and the sixth line uses the variable value as a parameter. We call the four lines that create the function the definition of the function. You can place your function definitions at the beginning or end or your scripts.

Function definitions begin with the word “function” followed by a space, then the name of the function. We can see that the function name is “timesTen”. After the function name is a pair of parentheses that enclose the parameter list for the function. We can see that there is one parameter called “number.” You can declare as many parameters as you like, but you must separate each one by a comma. Note that parameter names must follow the same rules as variable names.

After the parameter list is a block of one or more commands, enclosed by a pair of braces. The first line in this block for the `timesTen` function takes the `number` parameter, multiplies it by ten, and puts the result in the `result` variable. The second line begins with the keyword “return.” This keyword means “send back to whoever called this function the following value.” The value following the word “return” on the second line of this script is the variable `result`. So when one calls the `timesTen` function, it takes its first parameter, multiplies it by ten, and gives as a result the value of the result of that command.
We can see the sixth line of the script calls the `timesTen` function and the value five passes in as its parameter. The script assigns the result of the `timesTen` function to the value of the `myNumber` variable. The last line of the script prints out the value.

**Using Objects, Properties, and Methods**

An object is a self-contained module of data and its associated processing. We get objects to do work, or retrieve things for us, by calling the methods of the objects. We can also retrieve things from an object using the object’s properties. A method is like a function provided to us by an object. When we write script that asks for an object to run a particular method, we say we are calling a method. You can call a method in the following way:

```
objectName.methodName(parameters);
```

Sometimes a method returns a value and that return value is a variable in your script, in which case you call the method this way:

```
myVariable = objectName.methodName(parameters);
```

A property is like a variable that is part of an object. You can always retrieve a property, but you cannot usually change the property. You can retrieve a property value as follows:

```
myVariable = objectName.propertyName;
```

Some objects are available to your script at all times, like the `aa` object. You retrieve other objects through method calls, or create objects directly by your script like in the examples that use an array. Several predefined objects are available to script writers. Some of these, like `Array`, `Math`, and `String` are part of the JavaScript language. Other predefined objects like `aa` are additions to JavaScript provided by Accela for interacting with Accela Automation.

**Topics:**

- The Array Object
- The Math Object
- The String Object

**The Array Object**

We have already seen one example of how to create an array, but there are other ways to create an array that can be more convenient. An example:

```
myArray = new Array("Oranges", "Bagels", "Spinach");
aa.print(myArray[0]);
aa.print(myArray[1]);
aa.print(myArray[2]);
```

This script displays this output:

```
Oranges
Bagels
Spinach
```
In this example, we initialize an array simultaneously with three elements. This approach provides an easy way to create a small array when you know the contents of that array. The Array object has the property length, and the methods concat, join, pop, push, reverse, shift, slice, splice, sort, and unshift among others.

The Math Object

This object provides access to most if not all of the mathematical functions that you might need when writing scripts. The object defines properties such as E, LN10, LN2, PI and others. Recognize PI as the familiar constant 3.14159. The other properties are also constants. The Math object defines many constants not already mentioned. The Math object also defines these methods: abs, acos, asin, atan, atan2, ceil, cos, exp, floor, log, max, min, pow, random, round, sin, sqrt, and tan. Let us look at an example of using the math object:

```javascript
piToTheThirdPower = Math.exp(Math.PI, 3);
aa.print(piToTheThirdPower);
```

This script displays this output:

23.140692632779267

The example calls the “exp” method of the Math object, passes in the PI property of the Math object as the first parameter of the method, and three as the second parameter of the method. The exp method takes its first parameter and raises it to the power of the second parameter. The output is 3.14159 * 3.14159 * 3.14159 = 23.140692632779267. Consult the Accela Automation Script Writer’s Object Model Reference documentation or a book on JavaScript for more information on the Math object.

The String Object

When you execute a script with the line:

```javascript
myVariable = “Hello World.”
```

You are really creating a String object. Let us look at an example:

```javascript
myString = “Hello World”;
aa.print(myString.length);
aa.print(myString.toUpperCase());
```

This script displays this output:

11

HELLO WORLD

We can see from the example that you can use the name of the variable that contains the string to retrieve the length of the string, and call a method of the String object that retrieves an upper case version of a string. The String object has a length property, and the methods slice, split, substr, substring, toLowerCase, and toUpperCase among others. Consult the Accela Automation Script Writer’s Object Model Reference or a book on JavaScript for more information on the String object.
Version 2.0 of the master script framework provides script includes and free-form script control sequencing. As a result, you need to follow a different upgrade path from 1.x versions of the master script than previously followed.

This appendix details the framework changes as well as the steps required to properly upgrade existing version to the new 2.0 release.

Topics:

- Execution FrameWork Changes
- Script Control Sequencing Changes
- Upgrading from 1.x to 2.x
- Resolved Issues and Edits to Existing Scripts
- New Master Scripts
- New Functions

**Execution FrameWork Changes**

In the 1.x framework each master script incorporated all functions and variables needed for the associated event. Many of these functions and variables were common across most of the scripts. If you needed to make a change to one of these functions, you had to manually copy the change into each of the scripts that used the common function. This made maintenance of the scripts difficult.

The 2.0 framework localizes these common functions into a couple script files and includes the script files by reference in each of the individual script files during runtime. This function localization enables you to implement a common change across all master scripts by implementing the change in one place (Chapter 3: Master Scripts on page 56).

**Script Control Sequencing Changes**

In the 1.x framework, you had to number script controls (stored in Standard Choices) sequentially to execute them properly (01, 02, 03, for example). If you made an entry out of
sequence or if you had a gap in the numbering, the script control and any following script controls did not execute.

The version 2.0 framework executes all enabled script controls in the displayed order. Figure 41: Example Script Control Sequencing shows a valid set of script controls. Note that the last line contains a valid alpha character.

Figure 41: Example Script Control Sequencing

Upgrading from 1.x to 2.x

The following section details the upgrade process and special considerations required when upgrading.

Topics:
- Configuring the Global Variables
- Migrating Custom Functions
- Installing Master Scripts
- Updating Script Control Sequences
- Reinstating 1.x Script Control Sequencing

Configuring the Global Variables

The 2.x master script framework uses the INCLUDES_ACCELA_GLOBALS file to set common parameters across all the master script files. You set variable parameters one time in the INCLUDES_ACCELA_GLOBALS file, then include a reference to this file in each of the master script files. For the 1.x master script framework, you set these variable parameters individually in each of the master script files.

Best practice is to set these variable parameters that same in each of the master script files. If the variable parameter settings differ across the master script files, evaluate these differences and determine whether you want to retain those differences (Configuring Global Variables on page 62).
To update your 1.x version scripts to 2.0

1. Replace global variables that are the same across your 1.x master script files:

   ```javascript
   var SCRIPT_VERSION = 2.0
   eval(getScriptText("INCLUDES_ACCELA_FUNCTIONS"));
   eval(getScriptText("INCLUDES_ACCELA_GLOBALS"));
   eval(getScriptText("INCLUDES_CUSTOM"));
   ```

   **Note:** Best practice is to align your scripts with a single set of variable definitions in the INCLUDES_ACCELA_GLOBALS files.

   **Note:** If a master script file must use a global variable definition different from the other master script files, ignore the following procedure for that master script file.

2. Change the variables in the INCLUDES_ACCELA_GLOBALS to match your existing implementation.

3. Incorporate a reference to the INCLUDES_ACCELA_GLOBALS in each of the your 1.x master scripts.

4. Remove the old global variable settings from your 1.x master script files.

5. In the END User Configurable Parameters section, change the script version variable to 2.0.

   ```javascript
   var SCRIPT_VERSION = 2.0
   ```

6. Save the new master script file.

Migrating Custom Functions

The next step to implement the 2.x master script framework is to identify customization made to the 1.x master scripts and migrate that functionality to the custom include file. Evaluate each installed master script independently.

To migrate a custom function

1. Locate your customization.

   **Note:** Modify older script controls, that use the `closeWorkflow` function, to use the `closeTask` function.

2. Copy the custom function into the INCLUDES_CUSTOM script.

   This step makes the customization available to all 2.x master scripts (Figure 42: Copying Customizations).

3. If you modified an Accela provided master script function, copy that function to your INCLUDES_CUSTOM file.
4. Save the INCLUDES_CUSTOM file.

Installing Master Scripts

**Note:** Upgrade and install all master scripts at the same time.

To install new master scripts

1. Add the new master script to Accela Automation (*Adding a Script on page 52*).

2. Associate an event with the script (*Enabling an Event on page 48* and *Associating Events with Scripts on page 55*).

Updating Script Control Sequences

To update script control sequences

**Caution:** Do not modify the INCLUDES_ACCELA_FUNCTIONS script file to include customizations.

*If the INCLUDES_ACCELA_FUNCTIONS file contains a function of the same name as the INCLUDES_CUSTOM file, the function in the INCLUDES_CUSTOM file overwrites the function in the INCLUDES_ACCELA_FUNCTIONS file, which can cause unknown consequences.*
1. Review script controls for sequences that did not execute in the 1.x framework, but do execute in the 2.x framework (Script Control Sequencing Changes on page 267).

**Note:** A script writer can deliberately number a script control out of sequence, in the 1.x framework, to disable it (Figure 43: Example of Out of Sequence Script Control that Executes in 2.x).

![Figure 43: Example of Out of Sequence Script Control that Executes in 2.x](image)

2. Properly disable out of sequence script controls.

### Reinstating 1.x Script Control Sequencing

You can reinstate the 1.x script control sequencing rules to disable out of sequence script controls.

To reinstate 1.x script control sequencing rules

1. Locate the getScriptAction_v1_6 function in the INCLUDES_ACCELA_FUNCTIONS.js file.
2. Copy the function to a text editor.
3. Rename the function to getScriptAction.
4. Paste the new function in the INCLUDES_CUSTOM file.

### Resolved Issues and Edits to Existing Scripts

Table 34: 2.x Framework Script Improvements lists improvements made to master scripts in the 2.x framework.

<table>
<thead>
<tr>
<th>ACA Page Flow Scripts</th>
<th>Added ASI and ASIT functions for Accela Citizen Access page flow scripts. Updated page flow master script samples to use them.</th>
</tr>
</thead>
<tbody>
<tr>
<td>addCustomFee</td>
<td>Fixed to allow passing feePeriod.</td>
</tr>
</tbody>
</table>
### Table 34: 2.x Framework Script Improvements

<table>
<thead>
<tr>
<th>Function/Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addStdCondition</td>
<td>Updated to perform exact match of criteria.</td>
</tr>
<tr>
<td>addTimeAccountingRecordToWorkflow</td>
<td>Updated to accept TA group codes and type codes.</td>
</tr>
<tr>
<td><strong>All ASI Table functions</strong></td>
<td>ASI Tables functions all use the ansiTableValObj when working with table values. AddToASITable and AddASITable functions can use either the objects or strings when adding values.</td>
</tr>
<tr>
<td><strong>All Master Scripts</strong></td>
<td>Moved systemUserObj declaration after determining public user flag.</td>
</tr>
<tr>
<td>All Scripts</td>
<td>Replaced logMessage() with logDebug() for error.</td>
</tr>
<tr>
<td>ApplicationStatusUpdate Before</td>
<td>Updated return code to &quot;-1&quot; when cancelling the event. Case 10ACC-03164 requires this change.</td>
</tr>
<tr>
<td>ApplicationSubmitAfter and ConvertToRealCapAfter</td>
<td>Fixed issue for when you can submit Accela Citizen Access records anonymously and the user ID is null.</td>
</tr>
<tr>
<td>asiTableValObj</td>
<td>Updated asiTableValObj to always return a string. Fixes issue when value is null.</td>
</tr>
<tr>
<td>Contact functions</td>
<td>Updated all contact functions to use correct permitId event parameter.</td>
</tr>
<tr>
<td>copyASI Tables</td>
<td>Removed the redundant parameter check.</td>
</tr>
<tr>
<td>createChild</td>
<td>Added the optional parameter - parent capId: the record id to use as the parent for which to create the child.</td>
</tr>
<tr>
<td>copyASI Tables, copyAppSpecific</td>
<td>Added ignoreArr logic to allow for exclusions.</td>
</tr>
<tr>
<td>createPublicUserFromContact</td>
<td>Edited to solve issue with long passwords not working. See <a href="http://community.accela.com/accela_citizen_access/t/32/t/1694.aspx">http://community.accela.com/accela_citizen_access/t/32/t/1694.aspx</a></td>
</tr>
<tr>
<td>createRefContactsFromCapContactsAndLink</td>
<td>Now returns the sequence number of the contact that was created/refreshed.</td>
</tr>
<tr>
<td>documentUploadBefore and documentUploadAfter</td>
<td>Accela Automation now provides CapID; removed check that made script for Accela Citizen Access only.</td>
</tr>
<tr>
<td>externalLP_CA_3_2</td>
<td>Made minor revision</td>
</tr>
<tr>
<td>getContactArray</td>
<td>Added extra data elements to array.</td>
</tr>
<tr>
<td>getContactArray</td>
<td>Added check for ApplicationSubmitAfter event. Because the contactsgroup array is only on pageflow, on Accela Citizen Access, pull it the normal way even though it is a partial record.</td>
</tr>
<tr>
<td>getParentLicenseCapID</td>
<td>Changed to first return the Parent record. If not found, return the EST record.</td>
</tr>
<tr>
<td>Inspection events</td>
<td>Added totalTime parameter for inspection events. Added consistency with the inspection result comment between all events.</td>
</tr>
<tr>
<td>InspectionMultiple events</td>
<td>Updates to master scripts to handle when you do not choose an inspector.</td>
</tr>
<tr>
<td>InspectionMultipleSchedule Events</td>
<td>Added inspector names.</td>
</tr>
</tbody>
</table>
Table 34: 2.x Framework Script Improvements

<table>
<thead>
<tr>
<th>Script/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InspectionMultipleScheduleBefore</td>
<td>Added variables for inspObj, inspectionType, and inspectionGroup.</td>
</tr>
<tr>
<td>InspectionResultModifyBefore</td>
<td>Added parameter inspTotalTime passed from the event.</td>
</tr>
<tr>
<td>InspectionScheduleAfter events</td>
<td>Fixed issue to accommodate new &quot;request&quot; functionality.</td>
</tr>
<tr>
<td>inspScheduleDate</td>
<td>Updated based on issue with resulting inspections.</td>
</tr>
<tr>
<td>licenseProfObject</td>
<td>Updated to take a null lictype and return first match on licNumber.</td>
</tr>
<tr>
<td>licenseProfObject</td>
<td>Fixed potential undefined object error.</td>
</tr>
<tr>
<td>loadASITablesBefore</td>
<td>Removed readOnly aspects not necessary in before script.</td>
</tr>
<tr>
<td>loadTasks</td>
<td>Added active flag attribute.</td>
</tr>
<tr>
<td>loadTasks</td>
<td>Updated to include step number of task.</td>
</tr>
<tr>
<td>loadTaskSpecific</td>
<td>Error message output references the correct object name.</td>
</tr>
<tr>
<td>logDebug</td>
<td>Fixed bug.</td>
</tr>
<tr>
<td>logDebug</td>
<td>Fixed to no longer check nextWorkingDay.</td>
</tr>
<tr>
<td>lookup</td>
<td>Fixed bug in the function strControl; duplicate declaration caused scope issues.</td>
</tr>
<tr>
<td>PaymentReceiveBefore/After</td>
<td>Added new fields.</td>
</tr>
<tr>
<td>setIvr</td>
<td>Removed comment and changed to LogDebug.</td>
</tr>
<tr>
<td>StdCondition</td>
<td>Updated job to do a check on type/desc and prevent adding incorrect values.</td>
</tr>
</tbody>
</table>

New Master Scripts

- ParcelAddBefore
- PaymentProcessingAfter
- PaymentProcessingBefore
- TimeAccountAddAfter
- TimeAccountingUpdateAfter
- UniversalMasterScript
- VoidPaymentAfter
- VoidPaymentBefore

New Functions

- addAddressStdCondition
- addASITable4ACAPageFlow
- addContactStdCondition
- addLicenseStdCondition
- addTask
- addTimeAccountingRecord
- addTimeAccountingRecordToWorkflow
- applyPayments
- capSet
- copyASITables
- copyContactsByType
- copyOwnersFromParcel
- createParent
- createPendingInspection
- createPendingInspectionFromReqd
- editCapContactAttribute
- editReportedChannel
- feeAmountExcept
- genericTemplateObject
- getGuideSheetObjects
- guideSheetObject
- insertSubProcess
- licenseProfObject
- loadASITablesBefore
- paymentByTrustAccount
- paymentGetNotAppliedTot
- removeTask
- setTask