TIBCO ActiveMatrix® BPM Business Data Services Developer Guide

Software Release 4.3
April 2019

Important Information

Contents

TIBCO Documentation and Support Services 9

Business Data 11

Business Data Services (BDS) 11

BOM Class 12

Business Objects 13

Business Object Scope 13

Business Object Creation by Factory 13

Retrieving and Setting Business Object Attributes 13

Invoking Operations on Business Object Attributes 14

BOM Relationships and Process Local Data 14

Composition 14

Specialization and Generalization 14

UML Relationships Supported by Process Local Data 16

Assignment by Value and by Reference 16

BOM Native Type or Primitive Type Object to Business Object Attribute 17

Assigning a Business Object 18

Assigning a Basic Type Object to a Process Data Field 19

Significance of the Script Boundary 19

Business Data Definition 20

Business Data Projects 20

Business Data Project Versioning 21

Support for Local BOMs in Local Business Object Projects 22

Support for Local BOMs in Analysis Projects or BPM Developer Projects 22

Creating a Business Object Model (BOM) 23

Creating User-defined BOMs in the Business Objects Folder 23

Importing XSDs and WSDLs into Business Objects 23

Importing WSDLs into the Service Descriptors Folder 23

Importing a WSDL when Defining a Web Service Task 24

Generating a WSDL for a Web Service You Are Creating 24

BOM Native Types 25

Value Spaces for BOM Native Types 26

BOM Design-time Model 27

Primitive Types 27

Regular Expression Patterns for Text Fields 28

Multiplicity 30

Size Restrictions 30

Default Values . . . 31
BOM Labels and Names . . . 32
Label to Name Algorithm . . . 32
BOM Class Label and Name . . . 33
BOM Attribute Label and Name . . . 33
BOM Package Label and Name . . . 34
Reserved Words . . . 35
Name Clashes . . . 35

BDS Generation and Business Data Usage in TIBCO BPM . . . 37

BDS Design-time Validations . . . 37
Process Validations . . . 38
BDS Runtime Validations . . . 39
Process Migration . . . 39
Using BDS in Tasks . . . 41
Defining Web Services . . . 41
Business Data Scripting . . . 42
Supplying xsi:type Information in XML Sent to ActiveMatrix BPM . . . 43

Business Data Scripting by Example . . . 44

Factories . . . 44
Creating a New Business Object . . . 45
Create an Instance of a Class . . . 45
Create a Copy of a Business Object . . . 47
Using the Special Value Null . . . 47
Checking for Null Attributes . . . 47
Assigning a Null Value . . . 48
Using Content Assist . . . 49
Working with Single Instance Attributes of Business Objects . . . 51
Multiple Instances of a BOM Class . . . 51
Multiple Instances of a BOM Class in a Process Data Field . . . 51
Multiple Instances of a BOM Class in a BOM Class Attribute . . . 54
Multiple Instances in Sequences and Groups . . . 55
Passing Multiplicity to a Form . . . 55
Working with Temporary Variables and Types . . . 55
Loops Within Scripts . . . 56
Scripting Containment Relationships . . . 58
Using the List set() Method . . . 60
Working with Numeric Types . . . 66
Working with Basic Integer Numbers . . . 66
Signed Integers . . . 67

Working with Basic Decimal Numbers . . . 68
Implicit Conversions Between Numeric Types . . . 69
Working with Dates and Times . . . 71
Dates and Times . . . 71
Durations . . . 72
Using Date and Time Types with Durations . . . 73
Comparing Dates and Times . . . 73
Working with Enumerated Types (ENUMs) . . . 74
Working with Primitive Types . . . 76
Using Enumerated Types as Extensions of Primitive Types . . . 77
Return Values from Scripts . . . 78
Scripting with Web Services . . . 78
Passing Arrays to Web Services . . . 81
Parse Functions . . . 82
Advanced Scripting Examples . . . 84
Working with Fixed Length Integers (BigInteger) . . . 84
Unsupported Conversions . . . 86
Working with Fixed Decimals (BigDecimal) . . . 86
Creating and Initializing Fixed Decimal Values . . . 86
Simple Operations . . . 86
Rounding . . . 87
Unsupported Conversions . . . 89
Comparing Fixed Decimals and BigDecimals . . . 89
Object BOM Native Type . . . 91
Using the Object BOM Native Type . . . 91
Restrictions . . . 92
Object BOM Native Type and ScriptUtil.setObject() . . . 92
Additional JavaScript Global Functions . . . 94
Business Data Modeling Best Practice . . . 95
Store Local BOMs in Business Data Projects . . . 95
Keep Local BOMs and Global BOMs in Separate Business Data Projects . . . 95
Upgrade Business Data Projects and Dependent Process Projects Together . . . 95
Use Pre-Compilation for Projects That Contain Large or Generated BOMs . . . 95
Choose Appropriate Data Types . . . 96
Use Sub-Packages to Aggregate Related Concepts . . . 96
Process Data Field Granularity . . . 96
BOM Class Attribute and Variable Names . . . 96
Do Not Split a Namespace Across Projects . . . 97
Do Not Modify Generated BOMs . . . 97

Business Data Scripting Best Practice . . . 97
Troubleshooting . . . 99
Viewing BDS-generated BDS Plug-in Hidden Projects . . . 99
Troubleshooting BDS Scripting . . . 100
Reasons to Avoid Deleting Case Objects . . . 100
Reserved keywords to avoid using for attribute names . . . 100
BDS Classes Do Not Appear or Changes Do Not Appear . . . 101
Break Script into Smaller Scripts . . . 101
Examine the Server Logs . . . 101
Write Variable Values and Progress Updates from the Script to the BPM Log File . . . 102
eval() . . . 102
Use the Process Debugger . . . 102
Catch Exceptions . . . 102
BDS Plug-in Generation Problems . . . 103
Supplemental Information . . . 105
Data Type Mappings . . . 105
BOM Native Type to BDS Type Mapping . . . 105
XSD Type to BDS Type Mapping . . . 106
JDBC Database Type to BOM Data Type Mapping . . . 108
Process Primitive Data Type Mapping . . . 109
Unsupported XSD Constructs . . . 110
BDS Limitations . . . 110
Fixed Attribute Overwrite . . . 111
Multiplicity Ordering in a Sequence or Choice . . . 111
Nested xsd any in Sequences . . . 111
xsd any ##local . . . 112
Recurring Elements in Sequence . . . 112
The block Function . . . 112
Data Mapping . . . 113
Converting Values Between Different BOM Attribute Types . . . 113
Mapping to or from Process Basic Types . . . 118
JavaScript Features not Supported in TIBCO BPM Scripting . . . 119
Using If For and While Expressions . . . 120
Reserved Words in Scripts . . . 120
Business Data Scripting . . . 121
Static Factory Methods . . . 121
DateTimeUtil . . . 121
DataUtil . . . 123
ScriptUtil . . . 123

IpeScriptUtil (and ScriptUtil) Conversion Date and Time and String Functions . . . 125
BOM Native Type Methods . . . 128
Fixed Length Integer (BigInteger) Methods . . . 129
Fixed Point Decimal (BigDecimal) Methods . . . 130
Date Time Datetime and Datetimetz (XMLGregorianCalendar) Methods . . . 133
Duration Methods . . . 136
Text (String) Methods . . . 137
Other Supported Methods . . . 138
List Methods . . . 138
ListIterator Methods . . . 139
Other JavaScript Functions . . . 139
Math Methods . . . 140
Process Manager and Work Manager Scripting . . . 141
Process Instance Attributes and Methods . . . 141
Organization Model Attributes and Methods . . . 144
WorkManagerFactory . . . 144
WorkItem . . . 144
OrgModel . . . 148

TIBCO Documentation and Support Services

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit https://docs.tibco.com.

Product-Specific Documentation

Documentation for TIBCO products is not bundled with the software. Instead, it is available on the TIBCO Documentation site. To directly access documentation for this product, double-click the following file:

`TIBCO_HOME/release_notes/TIB_amx-bpm_version_docinfo.html`

where *TIBCO_HOME* is the top-level directory in which TIBCO products are installed. On Windows, the default *TIBCO_HOME* is `C:\tibco`. On UNIX systems, the default *TIBCO_HOME* is `/opt/tibco`.

The following documents for this product can be found on the TIBCO Documentation site:

- TIBCO ActiveMatrix BPM SOA Concepts
- TIBCO ActiveMatrix BPM Concepts
- TIBCO ActiveMatrix BPM Developer's Guide
- TIBCO ActiveMatrix BPM Web Client Developer's Guide
- TIBCO ActiveMatrix BPM Tutorials
- TIBCO ActiveMatrix BPM Business Data Services Developer Guide
- TIBCO ActiveMatrix BPM Case Data User Guide
- TIBCO ActiveMatrix BPM Event Collector Schema Reference
- TIBCO ActiveMatrix BPM - Integration with Content Management Systems
- TIBCO ActiveMatrix BPM SOA Composite Development
- TIBCO ActiveMatrix BPM Java Component Development
- TIBCO ActiveMatrix BPM Mediation Component Development
- TIBCO ActiveMatrix BPM Mediation API Reference
- TIBCO ActiveMatrix BPM WebApp Component Development
- TIBCO ActiveMatrix BPM Administration
- TIBCO ActiveMatrix BPM Performance Tuning Guide
- TIBCO ActiveMatrix BPM SOA Administration
- TIBCO ActiveMatrix BPM SOA Administration Tutorials
- TIBCO ActiveMatrix BPM SOA Development Tutorials
- TIBCO ActiveMatrix BPM Client Application Management Guide
- TIBCO ActiveMatrix BPM Client Application Developer's Guide
- TIBCO Openspace User's Guide
- TIBCO Openspace Customization Guide

- TIBCO ActiveMatrix BPM Organization Browser User's Guide (Openspace)
- TIBCO ActiveMatrix BPM Organization Browser User's Guide (Workspace)
- TIBCO ActiveMatrix BPM Spotfire Visualizations
- TIBCO Workspace User's Guide
- TIBCO Workspace Configuration and Customization
- TIBCO Workspace Components Developer Guide
- TIBCO ActiveMatrix BPM Troubleshooting Guide
- TIBCO ActiveMatrix BPM Deployment
- TIBCO ActiveMatrix BPM Hawk Plug-in User's Guide
- TIBCO ActiveMatrix BPM Installation: Developer Server
- TIBCO ActiveMatrix BPM Installation and Configuration
- TIBCO ActiveMatrix BPM Log Viewer
- TIBCO ActiveMatrix BPM Single Sign-On
- Using TIBCO JasperReports for ActiveMatrix BPM

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit http://www.tibco.com/services/support.
- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support portal at https://support.tibco.com.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to https://support.tibco.com. If you do not have a user name, you can request one by clicking Register on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to https://community.tibco.com.

Business Data

Business data is structured data that contains information about real-world entities that an organization deals with, for example Customer, Order, and Orderline.

Each of these entities will have a number of attributes, for example name, address, and date. These entities will also be related to each other in different relationships and with different multiplicities.

A fundamental part of developing a business process is to understand the data that the process manipulates and depends upon.

TIBCO Business Studio provides a tool, Business Object Modeler, that allows you to build up a description of the business data that the process will manipulate. The resulting Business Object Model (BOM) contains the different types of objects that the business uses, their attributes, and their relationships to each other. For more information on using Business Object Modeler, see *TIBCO Business Studio Modeling User's Guide*.

With the introduction of version 3.0, ActiveMatrix BPM supports two sorts of business data - normal (or local) business data, and case data. This guide describes the use of normal business data. Additional information about case data is provided in the *TIBCO ActiveMatrix BPM Case Data User's Guide*.

Object Orientation

The BPM runtime is an Object Oriented (OO) system, as is the scripting environment that supports it. The topics in this Introduction describe important concepts that stem from the use of this OO design. It is important to understand these concepts to achieve the desired behavior when using scripts to manipulate business data.

If you have difficulty understanding any of these concepts, refer to the examples described in Business Data Scripting by Example, then return to this section. One of the best ways to understand these concepts is to learn by example.

Business Data Services (BDS)

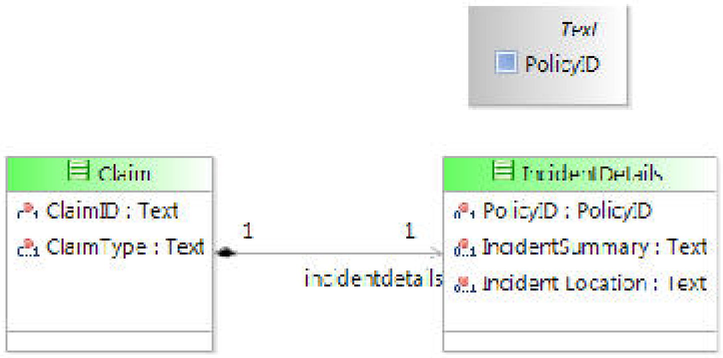
Business Data Services (BDS) is the component responsible for handling business data in TIBCO ActiveMatrix BPM. Use TIBCO Business Studio Business Object Modeler to define a BOM.

BDS converts the design-time BOM definitions into BDS Plug-ins that act as the runtime model for Business Data. The BDS Plug-ins are based on the Eclipse Modeling Framework (EMF). For more information about EMF, see:

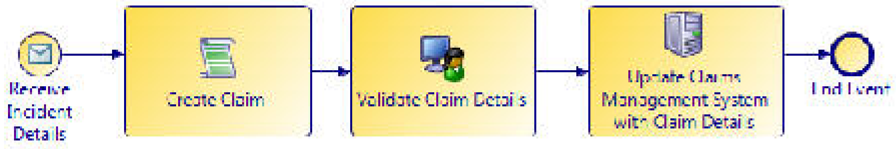
http://www.eclipse.org/modeling/emf/

You can interact with business data (any BOM-based data) from any activity in a Process (for example, a user task, script task, database task, and so on). This automatically results in an indirect interaction with BDS Component.

For example, consider a claims handling system, the data of which might be modeled as follows:



The following business process shows the tasks that handle a claim:

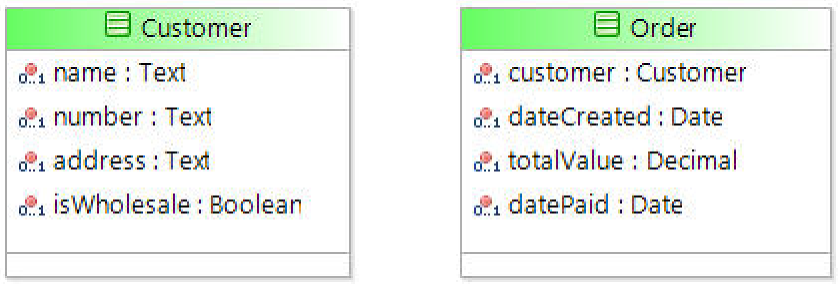


In this example, the user interacts with BDS in each of the activities:

- In the Create Claim script task, the script constructs the Business Objects.
- In the Validate Claim Details user task, a claims handler validates whether the details are correct.
- In the Update Claims Management System with Claim Details service task, details of the claim are persisted to a database.

BOM Class

As the name implies, the whole focus of OO is on objects. Objects are created to represent real world things such as a customer or an order.



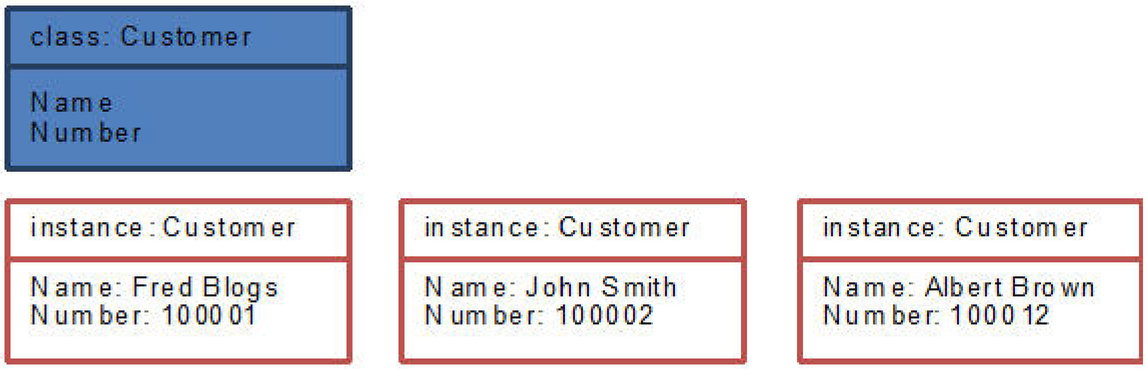
However, before we can have objects we must create a template, or pattern, for each type of object we want to process. This template is called a *BOM class,* and defines what the different objects will be like. For example, you can define a Customer BOM class and an Order BOM class, which would model real world customers and orders. Creating BOM classes for your application is a design-time activity.

Business Objects

At runtime, instances of these BOM classes are created to represent particular instances of the generic BOM class. These instances are referred to as Business Objects. For example, the Customer BOM class can have two Business Objects to represent two actual customers, John Smith and Fred Blogs.

Each of these BOM classes and Business Objects have attributes. For example, a Customer may have attributes that include a name and number, and an Order may have attributes that include Customer and DateCreated. Although Customer attributes differ from Order attributes, generally all Customers have the same set of attributes.

The diagram below shows a simple Customer class and three Business Objects, which are instances of the class:



Business Object Scope

The life span of a Business Object is typically bounded by the Process Instance in which it is created. This is called Process Local Scope.

It is possible in a service task to write information out to, or read from, a database. The structure of the database table may match one of the normal BOM classes, or the data in one of the normal BOM classes may have to be mapped into a Business Object that matches the database table structure depending on the implementation.

Business Object Creation by Factory

Business Objects are created by factories. There is a factory method for each class (for example, createCustomer, createOrder, and so on) within the BOM.

The factory methods have to be used when a Business Object is created. This happens *implicitly* when objects are created within User Tasks, but has to be done *explicitly* when an object is created in a script. See Business Data Scripting by Example for more information about factories.

Retrieving and Setting Business Object Attributes

Using Business Data Scripting capabilities, retrieving Business Object attributes, for example the attributes of a customer instance, is as simple as running a script:.

```
var custName =  customerInstance.name;
```

You can set the value of a Business Object attribute as follows:

```
customerInstance.name = "Clint Hill";
```

Invoking Operations on Business Object Attributes

As well as attributes, some classes have methods which perform operations on the object.

For example, the String class is used to represent a Text attribute's value, such as the name of a customer. Two of its methods are `toUpperCase()` and `toLowerCase()`.

```
lowercaseName = customer.name.toLowerCase();
uppercaseName = customer.name.toUpperCase();
```

In this example, if the Text attribute `customer.name` was "Fred Blogs", then the first assignment would set the `lowercaseName` variable to the value `fred blogs`, and the second assignment would set the `uppercaseName` variable to the value `FRED BLOGS`. In some cases, such as the example cited above, methods can return values. In other cases, methods can alter some of the attributes of the instance. It is important to know how the methods behave when you are using them.

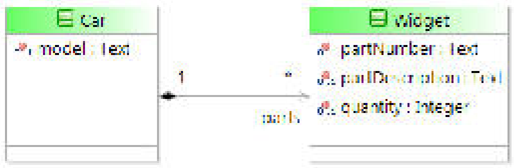
BOM Relationships and Process Local Data

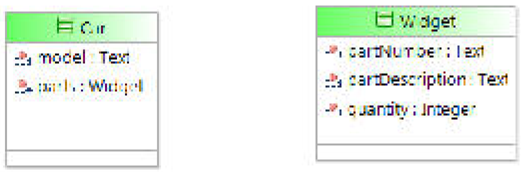
This section describes how BOM relationships are honored by process local data.

Composition

The Composition relationship is used to model the concept that "X is composed of Y". For example, a Car is made up of a number of Widgets.

This can be drawn in either of the following ways in the Business Object Modeler:





- In the first example, there is a line drawn between the Car and Widgets, which is labelled "parts".
- In the second example, there is a "parts" attribute in the Car class.

At runtime, these two approaches are treated identically. It does not matter which is used.

The Composition relationship is also known as the Containment relationship.

It is important to understand that in this kind of relationship, if a Car object is composed of a number of Widget objects, those Widgets cannot be components with other Car objects.

This makes sense if you consider the example of a steering wheel. It cannot be in two cars at the same time. Also, if the steering wheel from one car was installed into another car, it would no longer exist in the original car.

Similarly, if a Business Object contained in Business Object A is added to Business Object B, it will no longer exist in Business Object A.

Specialization and Generalization

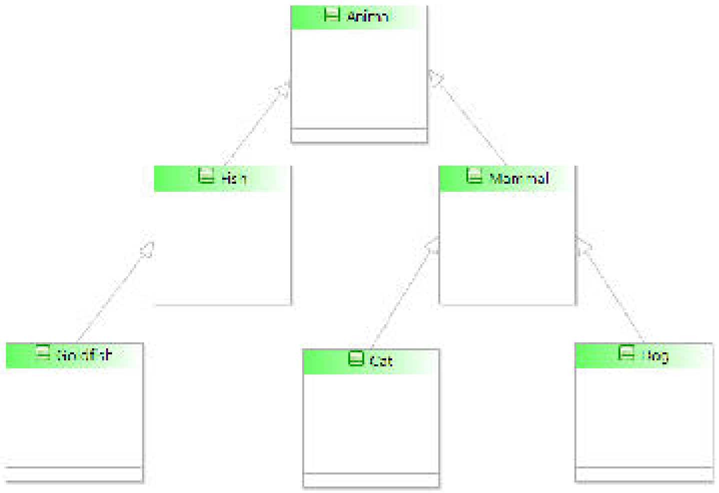
A useful aspect of OO technology that can be used in BOM and scripting is Specialization and Generalization. For example, consider the following terms: Animal, Fish, Mammal, Goldfish, Cat, and

Dog. Looking at these terms you can see that we can link some of them with the phrase "is-a" (or "is-an").

For example:

- A Goldfish is-a Fish.
- A Fish is-an Animal. (Also, a Goldfish is-an Animal).
- A Cat is-a Mammal.
- A Mammal is-an Animal. (Also, a Cat is-an Animal and a Dog is-an Animal).
- A Dog is-a Mammal.

In the Business Object Modeler, these relationships would be represented like this:

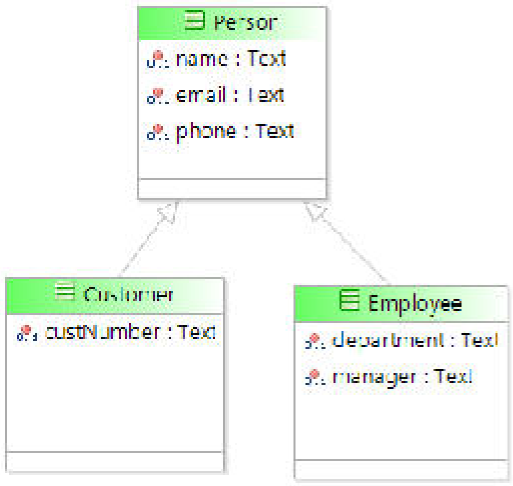


In OO terms, this means that a Mammal is a specialization of an Animal, and a Cat is a specialization of a Mammal, and so on.

If someone wants a pet, and they specify that they want a Mammal, a supplier can give them a Cat or a Dog because they both satisfy the "is-a" Mammal requirement, or they can even be given a Hamster, as a hamster "is-a" Mammal. However, providing a Goldfish does not satisfy the "is-a" Mammal requirement.

While building up the model of what objects a business deals with, you may discover that some classes have some things in common. For example, there might be a Customer and an Employee class, both classes representing a person. When modeling this, you can create a Person class that holds the common attributes of Customer and Employee (for example, name, email, and telephone). The Customer class can then specialize the Person class, adding the extra attributes that only a customer has (for example, customerNumber). Similarly, the Employee class can specialize the Person class, by adding any attributes that only the Employee had (for example, department, manager and so on).

Generalization is the reverse of specialization (looking at the same relationship from the opposite direction). We can say that the Person class is a generalization of the Customer and Employee classes. Alternatively, we can say that the Customer and Employee classes are specializations of the Person class. Having done this, it is possible in a process to create a list of all the Customers and Employees that have been involved in a particular Order by creating a list of Person instances (each instance of which could either be a Customer or an Employee). The relationship between these classes can be represented in the BOM as shown below:



Assigning a Subtype to a Super Type

It is always acceptable to assign a subtype to a super type.

Using the diagram above as an example, if you had a Customer business object, as its BOM class is a subtype of the Person class, you can assign the Customer business object to a business object attribute of type Person (because you can say that the Customer "is-a" Person).

Assigning a Super Type to a Subtype

Assigning a super type to a subtype is acceptable when the super type actually refers to an instance of the subtype (or one of its subtypes). In other words, it passes the "is-a" test.

For example, if you have a Business Object Attribute of the Customer subtype that you want to assign from another attribute of the Person type, it will only work if the Person attribute is actually referring to an instance of the Customer BOM class. If the Person attribute is referring to a Business Object of the generic Person, or the Employee BOM classes then it will not work, as they do not satisfy the "is-a Customer" requirement.

UML Relationships Supported by Process Local Data

The Business Object Modeler uses terms and notation similar to UML (Unified Modeling Language), so an understanding of UML can be useful.

BDS and Process Local Data support Generalization and Specialization relationships and Uni-directional Composition relationships.

Although TIBCO Business Studio BOM Editor also supports Association and Aggregation relationships, these relationships cannot be used in processes.

Assignment by Value and by Reference

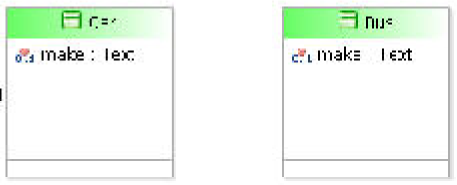
Assignments can be made either by reference or by value. For an assignment *by reference*, the entity to which the assignment is made refers to the entity being assigned. In other words, subsequent changes to the entity by either the new or existing reference is reflected in both places (there is only one entity; it isn't copied). However, for a *by value* assignment, a copy of the assigned entity is made, and that is what is applied to the entity receiving the assignment. This results in two independent objects. Therefore, changes in one place do not affect the other.

Assignment Conventions

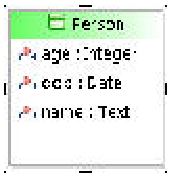
Assignment to...	...of type...	...behaves as follows:
Business Object attribute or composition	BOM Native or Primitive Type	Effectively by value. For efficiency's sake, objects are only copied where they are mutable (where their internal value can be changed). In other words, by reference behavior is sometimes used, but always behaves like by value. See BOM Native Type or Primitive Type Object to Business Object Attribute .
	BOM Class	By reference. **Important note**: If the BOM Object being assigned is already contained by another BOM object's containment, it will be removed from that containment automatically because it is impossible for an object to be contained by two containers at the same time. If this is not the desired behavior, make a copy of the object first. See Assigning a Business Object .
Process Data Field	BOM Class	By reference. See Assigning a Business Object .
	Basic Type	By value. See Assigning a Basic Type Object to a Process Data Field .

BOM Native Type or Primitive Type Object to Business Object Attribute

Assigning a BOM Native Type or Primitive Type object to a Business Object attribute is by value:



```
car.make = bus.make;
bus.make = "Ford" // will not affect car.make
```

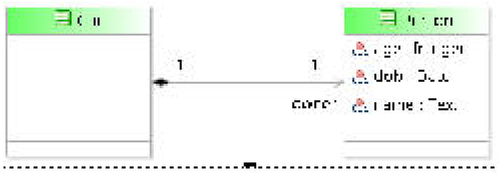


```
person1.age = person2.age;
person2.age = person2.age + 1;    // Will not increment person1.age
```

```
person1.dob = DateTimeUtil.createDate("1968-01-04");
person2.dob = person1.dob;
person2.dob.setYear(1970); // Value now 1970-01-04; person1.dob
                           // is still 1968-01-04
```

Assigning a Business Object

Assigning a Business Object is by reference.



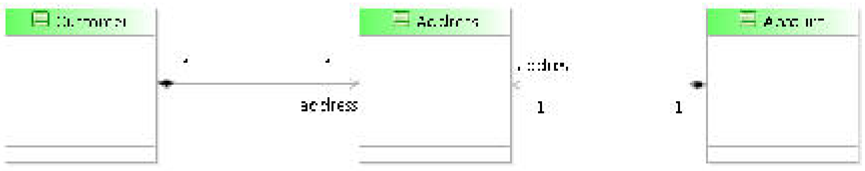
```
personDataField = car.owner; // Business Object assigned to data
                             // field by reference
personDataField.age = 25;    // Also affects car.owner.age
```

```
var tempPerson = personDataField;    // Business Object assigned to
                                 // local variable by reference
tempPerson.name = "Bob";            // Also affects personDataField.name
```

```
var owner = com_example_refval_Factory.createPerson();
car.owner = owner;
owner.name = "Ludwig"; // Also affects car.owner.name;
```

If a Business Object is assigned, but is already contained in another Business Object's containment, it is automatically removed from that containment. It is impossible for an object to be contained by two containers at the same time. If this is not the desired behavior, make a copy of the object first using the ScriptUtil.copy(...) utility, see Create a Copy of a Business Object .

In the next example, although all assignments are by reference (there is only ever one Address), the final line of the script removes the Address from Customer, leaving Customer with no Address:



```
var address = com_example_refval_Factory.createAddress();
customer.address = address;
account.address = customer.address; // Removes address from
                                    // customer
```

If this script is modified to use ScriptUtil.copy(...), account and customer end up with independent copies of the address:

```
var address = com_example_refval_Factory.createAddress();
customer.address = address;
account.address = ScriptUtil.copy(customer.address);
// Creates an independent Address
```

Assigning a Basic Type Object to a Process Data Field

Assigning a Basic Type object to a process Data Field is by value.

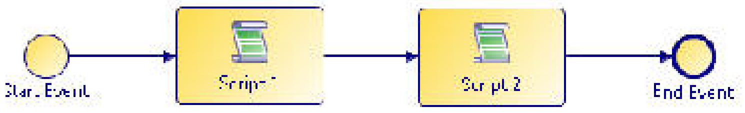
Data Fields
greetingDataField (greetingDataField)

```
var greeting="Hello";
greetingDataField = greeting;
greetingDataField = "Goodbye"; // Will not affect local variable
                               // greeting
```

Significance of the Script Boundary

When a script completes, all process Data Fields are independently converted to a form that can be stored in a database. Therefore, in a later script, modifying a value in one place never affects the other, regardless of whether a by reference assignment occurred in an earlier script.

This is illustrated by the following two scripts, which can be assumed to run one after the other, operating on the same process Data Fields:



Data Fields
personDataField1 (personDataField1)
personDataField2 (personDataField2)

Script 1:

```
personDataField1.age = 20;
personDataField2 = personDataField1; // Both data fields now
                                     // refer to same object
personDataField1.age = 35;       // This affects both data fields
```

Script 2:

```
// Each data field now refers to an independent Person object
personDataField1.age = 40; // will not affect
                           // personDataField2.age (still 35)
```

Business Data Definition

This section describes how you define business data in TIBCO Business Studio Object Model Editor, and the types of data it supports.

Business Data Projects

Business Data projects are used to store business data models that can be referenced by process projects.

A Business Data project can contain either or both of the following types of Business Object Model (BOM):

- **Global BOM** - a BOM that contains at least one case class or global class. A global BOM that contains a case class is also referred to as a **case data model**. You create a global BOM manually in the project's **Business Objects** folder.
- **Local BOM** - a BOM that contains only local classes. You can create a local BOM in two ways:
  - You can create one manually in the project's **Business Objects** folder.
  - You can add a WSDL or XSD to the project's **Service Descriptors** special folder. When you do this, a local BOM representing the WSDL or XSD is automatically generated in the project's **Generated Business Objects** folder. An automatically generated local BOM is also referred to as a **generated BOM**.

When you manually create a BOM the **Global Data** tools are available in the palette in the BOM Editor. **Global Data** tools are not available for an automatically generated local BOM.

Best Practice

For best results, follow these guidelines when creating and using Business Data projects:

- Keep local BOMs and WSDLs or XSDs (and their generated BOMs) in suitable Business Data projects, rather than in Analysis or BPM Developer projects. This:
  - makes it easier to organize and share local data among different processes. (Using a Business Data project, the local data only needs to be defined and deployed once. If you use an Analysis or BPM Development project - that is, the same project as a business process that uses the data - whenever that project is deployed or generated as a DAA, BDS Plug-ins corresponding to the referenced BOMs are packaged as part of the DAA. That is, every deployed process has its own copy of any local data it uses.)
  - provides better design-time performance, particularly for projects involving large numbers of local/generated BOMs (by avoiding unnecessary regeneration of BDS Plug-ins).
- Keep local BOMs and global BOMs in separate Business Data projects, unless you have a compelling reason to keep them together. This is particularly important for application upgrade, as local and global BOMs have different compatibility requirements:
  - Local BOMs: You can still upgrade a Business Data project if a local BOM contains incompatible changes, but doing so could result in failure to migrate a dependent process instance to the upgraded version. See Process Migration.
  - Global BOMs: You cannot upgrade a Business Data project if a global BOM contains incompatible (that is, destructive) changes. See "Upgrading a Case Data Model" in the *Case Data User's Guide*.
- If a process project references a class in a local or global BOM in a Business Data project, the version number of the Business Data project is used in the reference when the DAA for the process project is

generated. This creates an exact-match dependency on the version number from the process application to that version of the BDS application. Consequently, when you upgrade a local BOM or a global BOM, you should also upgrade any existing process project that references that Business Data project - even if that process project makes no use of the updated parts of the BOM. Keeping BDS applications and dependent process applications in step in this way facilitates subsequent deployments or undeployments of either application.

- Configure a project that contains generated BOMs to use pre-compilation, so that the BDS plugins derived from the BOMs are not generated each time that the project is built. Generated BOMs are often large and, being derived from WSDLs or XSDs, usually will not change very often, so using pre-compilation can significantly improve design-time performance. (To configure the project to use pre-compilation, right-click the project in Project Explorer and choose **Pre-compile Project** > **Enable**. See "Pre-Compiling Projects" in the *TIBCO Business Studio Modeling Guide* for more information about pre-compilation.)

Business Data Project Versioning

BOMs use the version number of their parent Business Data project. Correct version management of Business Data projects is essential when upgrading BOMs or process applications that reference them.

Version numbers are set on the **Properties** > **Lifecycle** dialog of the Business Data project.

The default format for a Business Data project's version number is:

major.*minor*.*micro*.[*qualifier*]

where:

- *major* defines the major version number of the project.
- *minor*.*micro*.[qualifier] defines the minor version number of the project.
- *qualifier* is an optional parameter that, if used, will be replaced by the timestamp value in the **Properties** > **Lifecycle** > **Build Information** > **Build Version** field. **Build Version** is a timestamp that is updated whenever the project is updated.

If you import the project into a different major version of Business Studio (for example, from a 3.x version to a 4.x version) the project is updated automatically and the **Build Version** timestamp is changed, even if you make no changes to the project.

Process projects also use a *qualifier* in their version number, but this is handled differently. On a process project version number, *qualifier* is replaced by a timestamp when the DAA is created. Importing a process project into a different major version of Business Studio does not change the *qualifier* timestamp.

The current version number is used when the Business Data project is deployed, and is visible:

- in TIBCO Administrator, as the business data application's **Application Template Version**.
- (if the project contains a case data model) in the Openspace Data Admin gadget, as the case model's version number.

If a process project references a class in a local or global BOM in a Business Data project, the version number of the Business Data project is used in the reference when the DAA for the process project is generated. This creates an exact-match application dependency from the process application to that version of the BDS application.

Consequently, when you upgrade a Business Data project (either by modifying a local or global BOM, or by importing the project into a new major version of Business Studio), you should also upgrade any existing process application that references that Business Data project - even if that process application makes no use of any updated parts of a BOM. Keeping BDS applications and dependent process applications in step in this way facilitates subsequent deployments or undeployments of either application.

See Also

"Case Data Model Versioning" in the *TIBCO ActiveMatrix® BPM Case Data User's Guide*.

Support for Local BOMs in Local Business Object Projects

From version 4.0 of Business Studio, you can no longer create a Local Business Object project. Local BOMs (whether user-created or generated) in existing Local Business Object projects created in pre-4.0 versions of Business Studio are still supported.

You can either:

- continue to reference local BOMs in Local Business Object projects from BPM process projects, in which case they will be handled by ActiveMatrix BPM exactly the same way as in earlier versions. (When you generate a DAA for or deploy a BPM process project that references a BOM in the Local Business Object project, a BDS Plug-in corresponding to the BOM is generated and packaged as part of the DAA.)
- move local BOMs or WSDLs into new or existing Business Data projects.
- refactor the Local Business Object project into a Business Data project. (Right-click the Local Business Object Model project, select **Refactor** > **Convert to Business Data Project**, then click **OK**.)

If you move or refactor local BOMs or WSDLs into Business Data projects, you must update any BPM process projects that reference those BOMs to reference them from their new location. It is good practice to only move or refactor local BOMs or WSDLs if you are making other significant changes to your process application.

Support for Local BOMs in Analysis Projects or BPM Developer Projects

You can still use local BOMs in Analysis projects and BPM Developer projects, but you should only do this when it is necessary. Best practice is to put local BOMs, WSDLs and XSDs in Business Data projects.

When you create a new BPM Developer project, the project does not by default contain a **Business Object Model** folder or **Service Descriptors** folder. When you create a new Analysis project, the project does not by default contain a **Service Descriptors** folder.

You should only add local BOMs, WSDLs or XSDs to a new Analysis project or BPM Developer project if you need to maintain compatibility with existing (pre-version 4.0) projects:

- If you want to add local BOMs to a project, you must first manually add a special folder of type **Business Object Model** to the project.
- If you want to add WSDLs or XSDs to a project, you must first add a special folder of type **Service Descriptors** to the project.

You can either select these assets when you create the project using the **New Analysis Project** or **New BPM Developer Project** wizard, or manually add them to the project later.

Local BOMs (whether user-created or generated) in existing Analysis or BPM Developer projects created in pre-4.0 versions of Business Studio are still supported. You can either:

- continue to use them as they are, in which case they will be handled by ActiveMatrix BPM exactly the same way as in earlier versions. (A BDS Plug-in corresponding to each BOM in the project is generated when you generate a DAA for or deploy the project.)
- move local BOMs or WSDLs into new or existing Business Data projects. If you do this, you must update any BPM process projects that reference the BOM to reference them from their new location.

  It is good practice to only move local BOMs or WSDLs if you are making other significant changes to your process application.

Creating a Business Object Model (BOM)

There are two ways to create a BOM, manually in the Business Object Modeler, by one who knows the structure of the data that the business uses; or by importing existing XSD or WSDL files that contain data definitions used by existing applications.

There can be multiple BOMs with cross-references between them, although there cannot be circular references (dependencies) between BOMs.

Creating User-defined BOMs in the Business Objects Folder

You can create a new BOM during or after creating a project.

BOMs are created from the context menu in the Business Objects folder in the Project Explorer.

User-defined BOMs can be found in the Business Objects folder of the TIBCO Business Studio project.

Importing XSDs and WSDLs into Business Objects

If your processes need to manipulate structured data whose structure is defined in XSD files or WSDL files, the structure of the data can be imported into the Business Objects folder.

This is only done on WSDL files if you are not going to call the interfaces in the web service defined in the WSDL file. If you intend to call the interfaces, import the WSDL as described in Importing WSDLs into the Service Descriptors Folder .

Once the import has completed, you can explore the data structures that have been imported in the Business Objects folder. Structured data can also be manipulated in scripts in the same way as user-defined BOMs.

If you modify these imported BOMs, the copy of the definitions in TIBCO Business Studio will differ from the source that they came from, causing TIBCO Business Studio to display a warning.

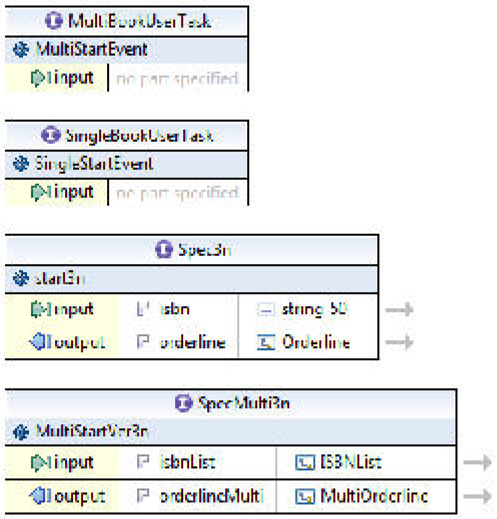
Despite the warning, this mismatch is normal if you have data types that are defined in an external system, then imported and extended to cope with known and future requirements. The following is an example of how this might be used: If you have an existing database, the structure of its tables can be exported into an XSD file, which can then be imported into TIBCO Business Studio. The information about the table structure can be used in tasks that interact with the database.

Once the data structure in the XSD (or WSDL) file has been imported into a BOM, web services can be defined and WSDLs generated for other applications to invoke the new web services that use the structured data from the imported files.

Importing WSDLs into the Service Descriptors Folder

If you have a web service that you want to invoke, the WSDL defining that service can be imported into the Service Descriptors folder. This is done by right-clicking the Service Descriptors folder of a TIBCO Business Studio project, and selecting Import, then Service Import Wizard. This allows you to import a WSDL file from a number of sources, including a file or a URL.

Once imported, you can view the services defined by the WSDL file, for example:



As part of the importing of the WSDL file, a BOM is generated that contains all the structured data definitions used by the web service. This generated BOM will be created in the Generated Business Objects folder and can be viewed in the same way as the user-defined BOMs above. However, if you attempt to change one of the generated BOMs, a warning is generated:.

TIBCO recommends that you do not change a generated file, as you risk losing changes if the file is regenerated.

In order to call a web service, a service task must be added to a BPM process and the Task Type must be set to Web Service. Then the operation must be selected from the operations that were imported from the WSDL file by clicking the Select button in the Service Task General Properties tab.

Script tasks and user tasks can process business objects created from the structured data imported from the WSDL file in the same way as the structured data defined in the user-defined BOMs. These Business Objects can then be mapped onto the input and output parameters of the Web Service task that is being called.

Importing a WSDL when Defining a Web Service Task

When a web service task is created in a process, there is an Import WSDL option in the Properties tab to import the definition of the web service from a WSDL file (if the WSDL file has not already been imported into the project as described previously). Once the WSDL file has been imported it can be viewed in the Service Descriptors folder of the TIBCO Business Studio project to explore the different services provided by the web service.

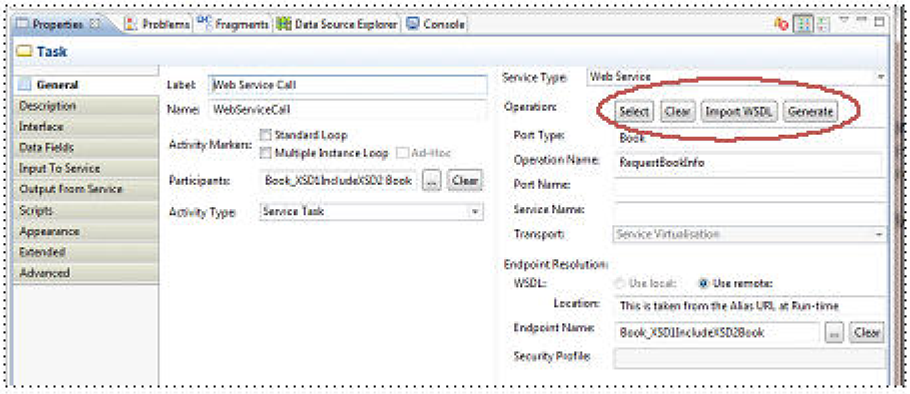
As previously explained, the data types used by the web service are defined in a BOM created in the Generated Business Objects folder.

Generating a WSDL for a Web Service You Are Creating

If you do not have a WSDL file for a web service (for example, because it has not yet been produced), you can use the Generate WSDL button in the Web Service General Properties tab to generate a WSDL file that you can use to define the contract that the Web Service provides.

The Properties tab for a web service task has buttons for:

- Selecting an existing WSDL service
- Importing a WSDL if the WSDL has not already been imported
- Generating a WSDL. If the WSDL does not exist yet, it can be created here



Label: Web Service Call
Name: WebServiceCall
Activity Markers: Standard Loop, Multiple Instance Loop, Ad-Hoc
Participants: Book_XSD1IncludeXSD2 Book
Activity Type: Service Task
Service Type: Web Service
Operation:
Port Type: Book
Operation Name: RequestBookInfo
Port Name:
Service Name:
Transport: Service Virtualization
Endpoint Resolution:
WSDL: Use local: [✓ Use remote:]
Location: This is taken from the Alias URL at Run-time
Endpoint Name: Book_XSD1IncludeXSD2Book
Security Profile:

For a more detailed overview of defining Web Services, see Scripting with Web Services.

BOM Native Types

A number of BOM Native Types are supported by BOM Editor.

- Attachment
- Boolean
- Date
- Date Time
- Date Time and Time Zone
- Decimal
- Duration
- ID
- Integer
- Object
- Text
- Time
- URI

There are also:

- two Decimal sub-types (Fixed Point and Floating Point)
- two Integer sub-types (Fixed Length and Signed Integer).
- 4 sub-types of Object (`xsd:any`, `xsd:anyType`, `xsd:anySimpleType` and `xsd:anyAttribute`).

Therefore, in total, there are 18 different types, if you include the sub-types as types.

The Attachment type is not currently supported by the BPM runtime.

Using Business Object Modeler, these BOM Native Types can be used to generate Primitive Types that have some application-specific restrictions, for example, a range or a regular expression that it must match.

Value Spaces for BOM Native Types

The value space of the different BOM Native Types is shown in the following table.

Value Spaces for BOM Native Types

BOM Native Type	Value Space
Attachment	N/A
Boolean	true, false
Date	Year in range [-999,999,999 – 999,99,999] Month in range [1 – 12] Day in range [1 - 31] (dependent on month & year) For example: "2011-12-31"
Datetime	Date fields according to Date type above Time fields according to Time type below Optional timezone offset in range [-14:00 - +14:00 or Z for Zero offset] For example: "2011-12-31T23:59:59" or "2011-12-31T23:59:59-05:00"
Datetimetz	Date and Time fields according to Datetime type above, but timezone is mandatory For example: "2011-12-31T23:59:59Z" or "2011-12-31T23:59:59+05:00"
Decimal – Fixed Point	An arbitrarily long integer number with the which has its decimal point moved to the left or right by up to 231 places. For example: 1234567890.1234567890
Decimal – Floating Point	• Negative numbers between -1.79769E+308 and -2.225E-307 • 0 • Positive values between 2.225E-307 and 1.79769E+308. For example: 1.23
Duration	A duration consists of year, month, day, hour, minute, second and sign attributes. The first 5 fields are non-negative integers, or they can be null if not set. That is, they can have values in the range [0 - 2,147,483,647]. The seconds field is a non-negative decimal (or null) in the range [0 - 1.79769E+308] although it is fetched from the object as an integer and the sign can be "+" or "-". For example: P3DT2H for 3 days and 2 hours

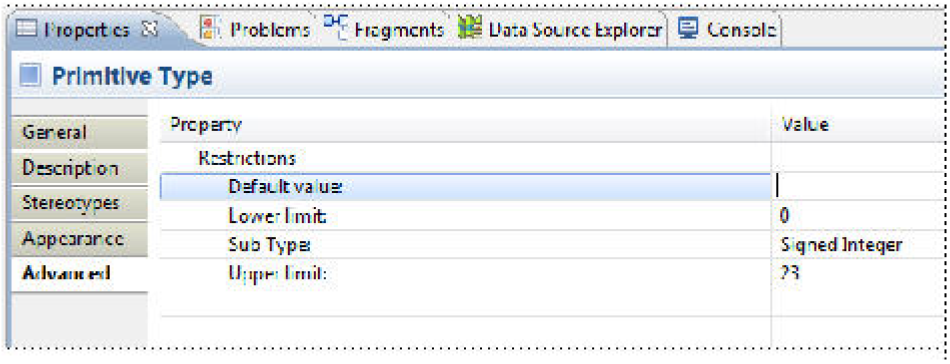
BOM Native Type	Value Space
ID	Non-colonized name. Starts with a letter or "_", and may be followed by more letters, numbers, "_", "-", ".", or combinations of characters and extenders. See http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName for more details. For example: ID1234
Integer – Fixed Length	Arbitrarily large length integer For example: 12345678901234567890
Integer – Signed	Integers in the range -2,147,483,648 to 2,147,483,647 For example: 123456789
Object – xsd:any	A block of XML satisfying `xsd:anySimpleType` schema definition. See http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. This can also be given the value of a Business Object.
Object - xsd:anyAttribute	A block of XML satisfying `xsd:anyAttribute` schema definition. See http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.
Object - xsd:anySimpleType	A block of XML satisfying xsd:anySimpleType schema definition See http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. This can also be given the value of a BOM Primitive
Object - xsd:anyType	A block of XML satisfying xsd:anyType schema definition See http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. This can also be given the value of a Business Object or the value of a BOM Primitive.
Text	Arbitrary length of long text string of Unicode characters For example: Fred Blogs
Time	Hour in range [0 – 24] (for value 24 minutes and seconds must be 0) Minute in range [0 – 59] Second in range [0 – 60] (NB 60 only allowed for a leap second) For example: 12:34:56
URI	Refers to `xsd:anyURI`. See World Wide Web Consortium, XML Linking Language (XLink) available at: http://www.w3.org/TR/2001/REC-xlink-20010627/ For example: http://www.tibco.com

BOM Design-time Model

The BOM Design-time model is used to define your business object model, and is created using a set of tools in the Business Object Modeler.

Primitive Types

In a BOM, it is possible to define your own Primitive Types, which are specializations of BOM Native Types. Usually, the value space of a user-defined Primitive Type is restricted in some way, for example, a type called Hour can be defined, based on the Integer(Signed) type with the values restricted to numbers between 0 and 23. The sub-type and restrictions are defined on the Property tab for Primitive Types:

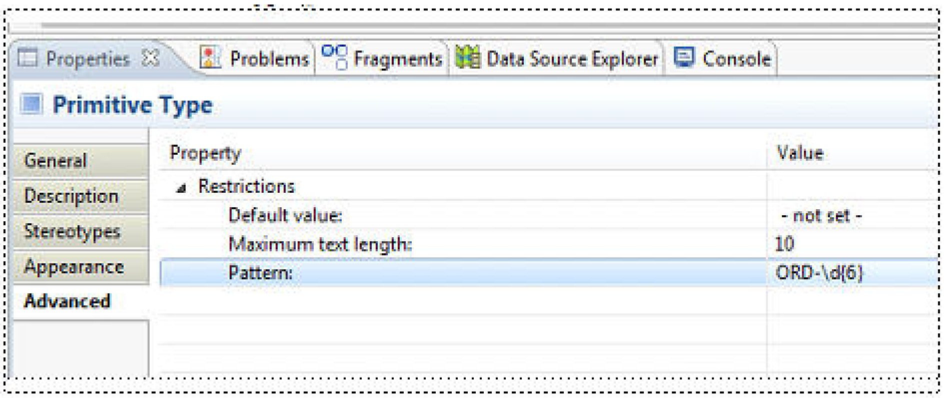


Property	Value
Restrictions	
Default value	
Lower limit	0
Sub Type	Signed Integer
Upper limit:	23

Regular Expression Patterns for Text Fields

When defining a Primitive Type, text fields can be constrained to match certain patterns.

For example, if you want an OrderId type to be formatted as ORD-NNNNNN, where "N" represents a digit, then you can specify this by using a Pattern in the Advanced Property Sheet for the Primitive Type:



Property	Value
Restrictions	
Default value:	- not set -
Maximum text length:	10
Pattern:	ORD-\d{6}

In the pattern shown in the above example, the "\d" represents any digit and the "{6}" means there must be 6 of the preceding element (in this case, a digit).

Additional example patterns that can be used are listed below.

Regular Expression Example Patterns

Pattern	Examples	Description
ID-[0123456789abcdef]*	Correct: ID-deadbeef ID- ID-1234 ID-0fff1234 Incorrect: ID ID-0A	The content in the square brackets may include any number or lowercase letter of the alphabet. In other words, the square brackets may contain any hex digit. The asterisk means zero or more of the previous element. In this case, any number of lowercase hex digits may be used.

Pattern	Examples	Description
ID-[a-f0-9]+	Correct: ID-0000dead Incorrect: ID- ID-0000DEAD	As in the previous example, the "a-f" means any letter in the range "a" to "f" and the "0-9" means any digit. The "+" means one or more.

[^;]+;	Correct:
	Hello;
	Incorrect:
	Hello;Bye;
		The "^" at the start of a range refers to characters outside the allowable range, so "[^;]" matches any character apart from ";". The "+" after it means one or more, and must be followed by a ";".

\d{1,3}\.\d{8}	Correct: 1.12345678 123.12345678 Incorrect: 1.234	"\d" means any digit. "{1,3}" means between 1 and 3 repetitions of the previous element. "\." matches a ".". However, because "." normally matches any character, in this case it must be escaped. Similar to the "\d" sequence, you can use "\w" for any word character and "\s" for any white space character. The "\D", "\W", and "\S" sequences inverse character sets.
\i\c*	Correct: xml:schema :schema Incorrect: 0xml:schema -schema	"\i" matches the initial character of any valid XML name, and "\c" matches any character that can appear after the initial character. Similarly, the "\I" and "\C" are the negations.
[\i-[:]][\c-[:]]*	Correct: xmlschema _schema Incorrect: xml:schema :schema	"[a-z-[aeiou]] a-z"matches any lowercase letter, adding "–[aeiou]" at the end removes the vowels from the set of letters that are matched. The example on the left removes ":" from the "\i" and "\c" lists of characters so the pattern now matches non-colonized names.
[-+]?\d+(\.\d+)?	Correct: 0.1 -2.34 +3 Incorrect: 4.	If you want to include a "-" in a range, then it should be the first character, as it has a special meaning if not the first character. The "?" indicates that the previous element is optional. The parenthesis marks form a group. The whole group is optional because of the "?" at the end. If a decimal point appears, then decimal digits must follow in this example.

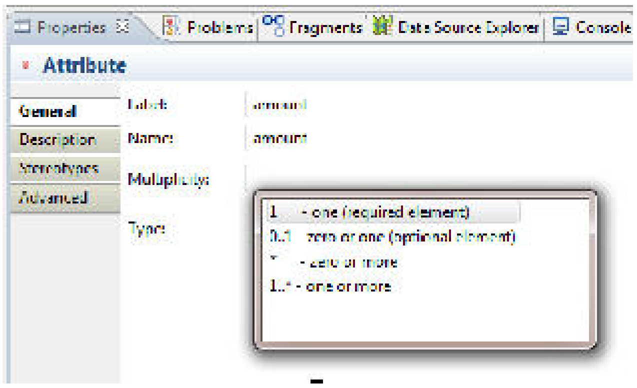
You can read more about regular expressions at:

- http://www.regular-expressions.info/xml.html
- http://www.w3.org/TR/xmlschema-2/#regexs
- http://www.xmlschemareference.com/regularExpression.html

Multiplicity

When BOM class attributes and compositions are defined, the developer can define how many values the attribute can or must have.

When defining the multiplicity for an attribute, content assist is available by typing Ctrl-SPACE. You can then select one of the following options:



Label: amount
Name: amount
Multiplicity:

1 - one (required element)
0..1 zero or one (optional element)
* - zero or more
1..* - one or more

It should be noted, however, that multiplicity can have other values, such as the following:

Multiplicity	Meaning
2..3	There must be 2 or 3
4	There must be 4
4..*	There must be 4 or more

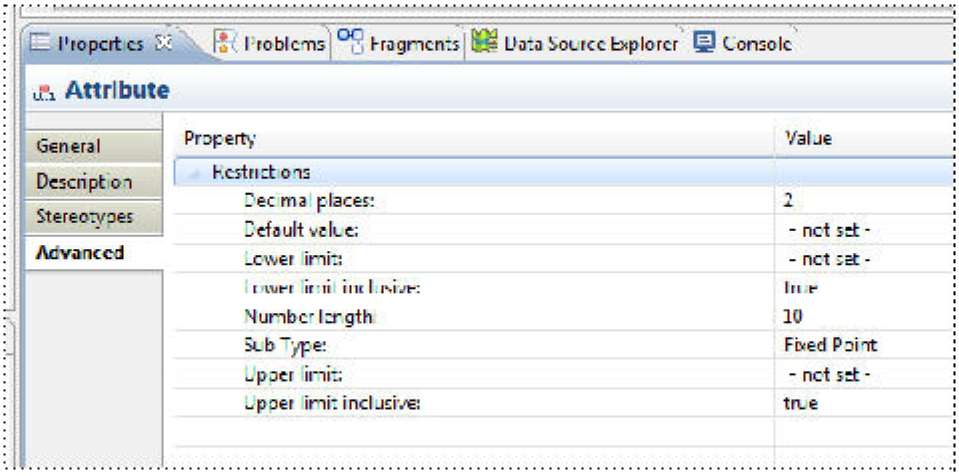
If there can be more than one element, then the attribute being defined will be a List. Therefore, when referenced in scripts, the List methods must be used.

Size Restrictions

When Text attributes are defined, they have a maximum size defined. This can be changed in the attribute’s Advanced Property tab.

When Fixed Length sub-types of Integer and Decimal attributes are defined, the length for the attribute is specified in the Advanced Property sheet. Additionally, for the Fixed Length Decimal sub-type, the number of decimals for the attribute is set.

For all Integer and Decimal sub-types, upper and lower bounds can be set on the value that the attribute can take. Again, this is done through the attribute’s Advanced Property tab. For example:


Properties | Problems | Fragments | Data Source Explorer | Console

Attribute

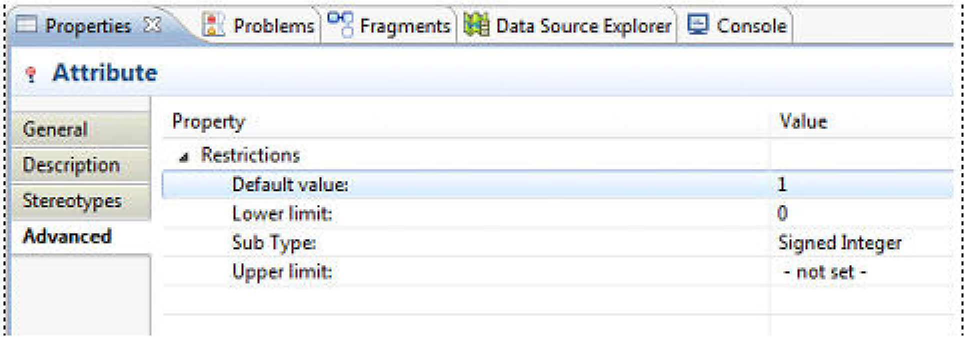
General | Description | Stereotypes | Advanced

Property	Value
Restrictions	
Decimal places:	2
Default value:	- not set -
Lower limit:	- not set -
Lower limit inclusive:	true
Number length:	10
Sub Type:	Fixed Point
Upper limit:	- not set -
Upper limit inclusive:	true

Default Values

Default values can be used when defining attributes for BOM classes. This is done through the Advanced tab in the Properties.

If an attribute has a default value, then as soon as an instance of the class that it is an attribute of is created, the attribute has that value. For example, if a class has an integer type attribute called quantity, it might be given a default value of 1 in the Advanced Property tab as shown below:


Properties | Problems | Fragments | Data Source Explorer | Console

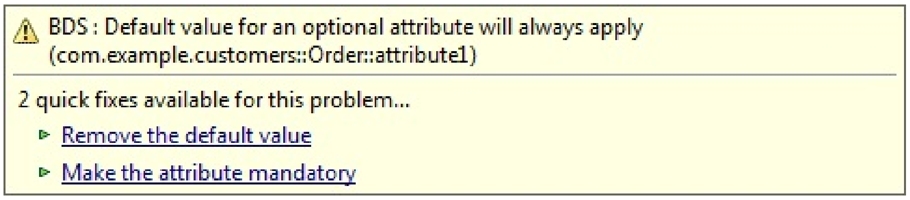
Attribute

General | Description | Stereotypes | Advanced

Property	Value
Restrictions	
Default value:	1
Lower limit:	0
Sub Type:	Signed Integer
Upper limit:	- not set -

When an instance of this class is created, the quantity attribute will be 1.

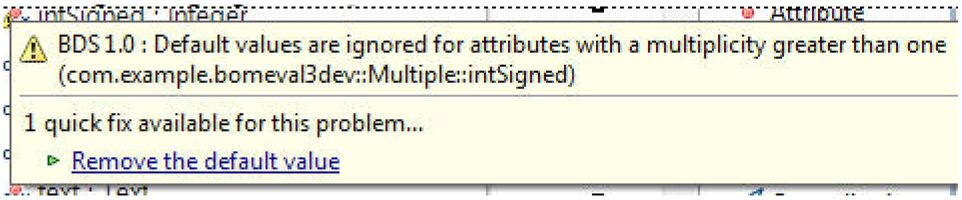
If you attempt to set a default on an optional attribute, for example, one with a potential multiplicity of zero, the following warning is generated:


BDS : Default value for an optional attribute will always apply (com.example.customers::Order::attribute1)

2 quick fixes available for this problem...

- Remove the default value
- Make the attribute mandatory

If you attempt to set a default on an attribute with a multiplicity greater than 1, the following warning is generated:


BDS 1.0 : Default values are ignored for attributes with a multiplicity greater than one (com.example.bomeval3dev::Multiple::intSigned)

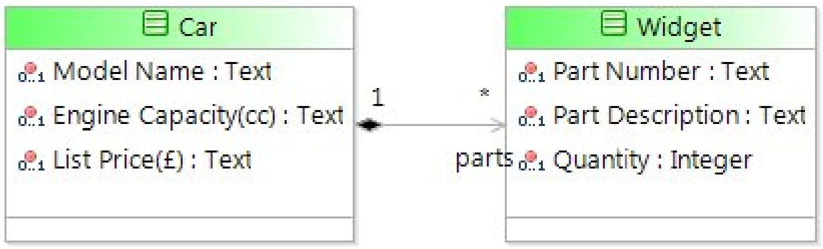
1 quick fix available for this problem...

- Remove the default value

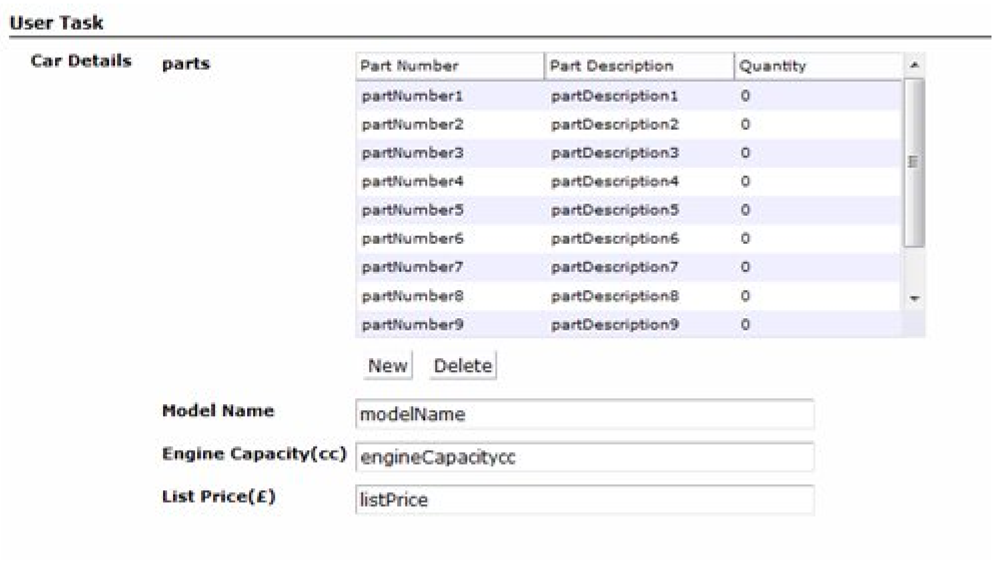
BOM Labels and Names

BOM packages, BOM classes, and BOM attributes have a Label as well as a Name.

A *Label* is a free format field that is designed to be a user-friendly description of the attribute or class. The label text is displayed by the BOM editor as shown below:



And by the default Forms:



Part Number	Part Description	Quantity
partNumber1	partDescription1	0
partNumber2	partDescription2	0
partNumber3	partDescription3	0
partNumber4	partDescription4	0
partNumber5	partDescription5	0
partNumber6	partDescription6	0
partNumber7	partDescription7	0
partNumber8	partDescription8	0
partNumber9	partDescription9	0

Model Name	modelName
Engine Capacity(cc)	engineCapacitycc
List Price(£)	listPrice

However, the Label cannot be used with the entire product, as in scripting. For example, the names that can be used to refer to attributes are considerably constrained. Therefore, a name is defined for each class and attribute. The default name is generated by using the characters from the label that are allowed in names. Space and most punctuation is removed. The image of the form above shows the names as well as the labels for the different fields. You can see that the spaces, brackets, and currency symbol have all been ignored when the names were generated. In scripting, the attribute name is used. For example:

```
var engineSize = car.engineCapacitycc;
var carPrice = car.listPrice + delivery + tax;
```

Label to Name Algorithm

All characters in the label that are not valid characters in the name are removed when the label is converted to the default name.

Only the characters A-Z (excluding accented characters), a-z, and the underscore character "_" are valid for use as a name's first character.

For subsequent characters in the name, the same characters are valid as for the initial character, with the addition of the digits 0-9.

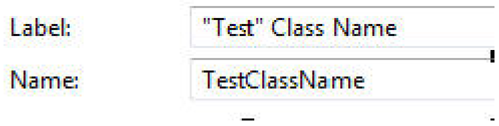
Spaces and punctuation characters are not allowed in names, including the following:

Character	Character Description
.	Dot
,	Comma
-	Hyphen

BOM Class Label and Name

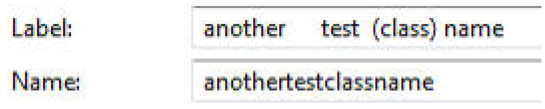
Any characters are valid in a label. As the label is converted to the class name, any illegal name characters are ignored.

In the following example, you can see the quotation marks and spaces in the Label are removed in the Name:



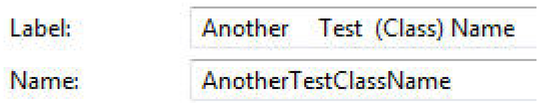
Label: "Test" Class Name
Name: TestClassName

The Label is used in BOM diagrams and the Name is used in scripts. Because the Name is used in scripts, it is easier to read if initial capital letters are used for each separate word, as in the example above. The following example shows a Label and Name pair where only lowercase letters are used



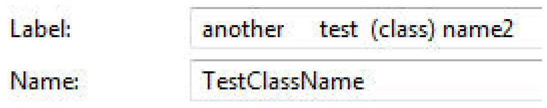
Label: another test (class) name
Name: anothertestclassname

Compared that to the following:



Label: Another Test (Class) Name
Name: AnotherTestClassName

The Name field is generated from the Label field whenever the label is changed unless the Name field has been manually set, in which case the name must then be manually changed. For example:

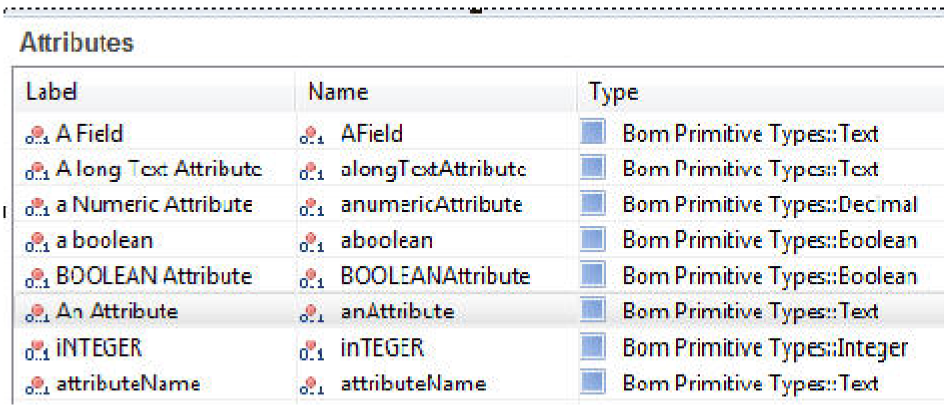


Label: another test (class) name2
Name: TestClassName

BOM Attribute Label and Name

For the names of class attributes, there is a requirement that the first two characters of the name must use the same case.

To meet this requirement, the software automatically uses lowercase letters for the first two characters if they are of different cases, as shown in the following examples:



Attributes

Label	Name	Type
A Field	AField	Bom Primitive Types::Text
A long Text Attribute	alongTextAttribute	Bom Primitive Types::Text
a Numeric Attribute	anumericAttribute	Bom Primitive Types::Decimal
a boolean	aboolean	Bom Primitive Types::Boolean
BOOLEAN Attribute	BOOLEANAttribute	Bom Primitive Types::Boolean
An Attribute	anAttribute	Bom Primitive Types::Text
iNTEGER	inTEGER	Bom Primitive Types::Integer
attributeName	attributeName	Bom Primitive Types::Text

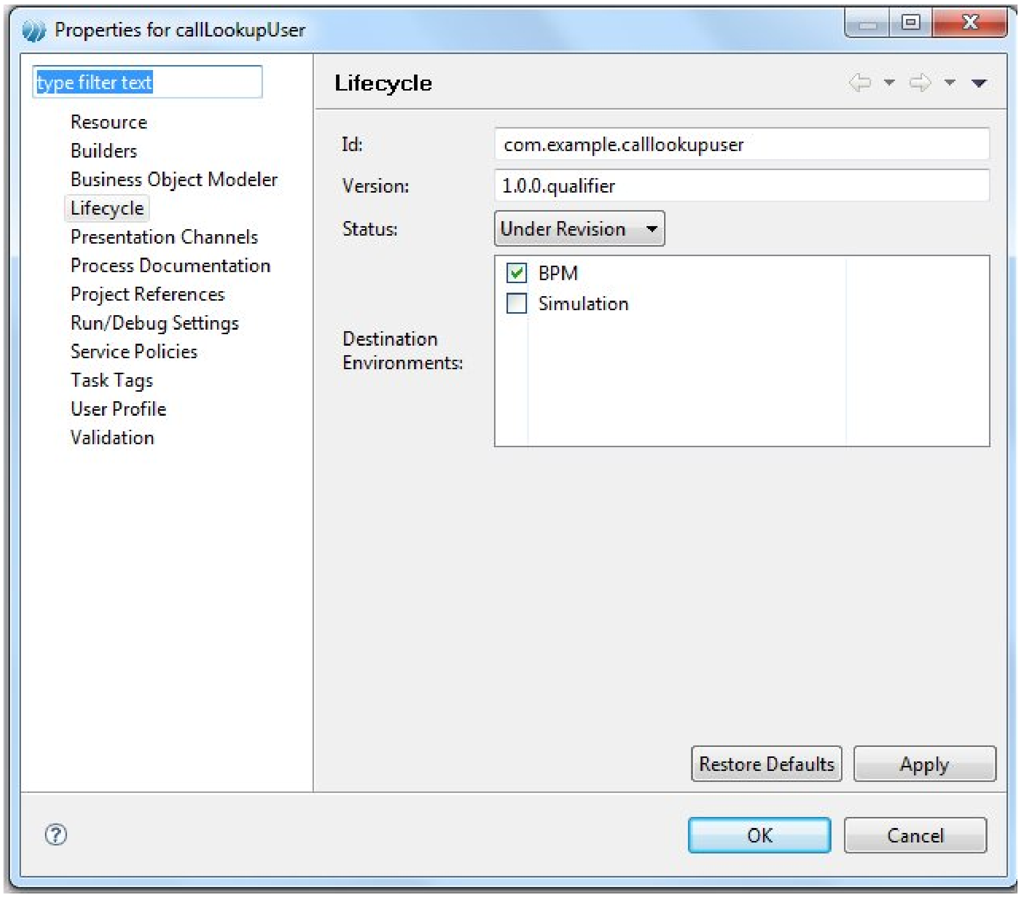
When creating attributes for BOM classes, it is recommend that you use uppercase letters for the initial letter of each word in the label, and if possible, do not use, a single letter word as the first word. The names will follow the camelcase convention. ("Camelcase" names have lots of "humps"!)

BOM Package Label and Name

The BOM top-level package or model name must be made up of dot-separated segments, each beginning with a letter or underscore and containing only letters, underscores, and numbers, and must avoid reserved words in a similar manner as BOM class names.

Each BOM must have a unique name across all other projects in the TIBCO Business Studio workspace.

The BOM package name must differ from the TIBCO Business Studio project lifecycle ID, which is also a dot-separated name in the same format. The project lifecycle ID can be viewed by right-clicking on the Project and selecting **Properties** > **Lifecycle**.



Properties for callLookupUser

Lifecycle

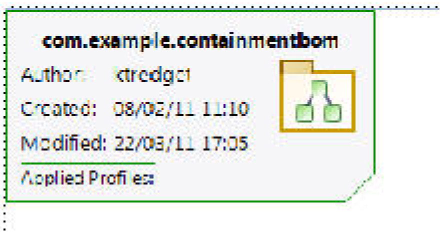
Id: com.example.calllookupuser

Version: 1.0.0.qualifier

Status: Under Revision

Destination Environments: [✓ BPM] Simulation

The BOM Label is free format text, meaning it can be any text that you want to display. The Label is displayed in the BOM editor, and by default is the same as the name of the BOM, as shown in the following example:


com.example.containmentbom
Author: ktredgct
Created: 08/02/11 11:10
Modified: 22/03/11 17:05
Applied Profiles

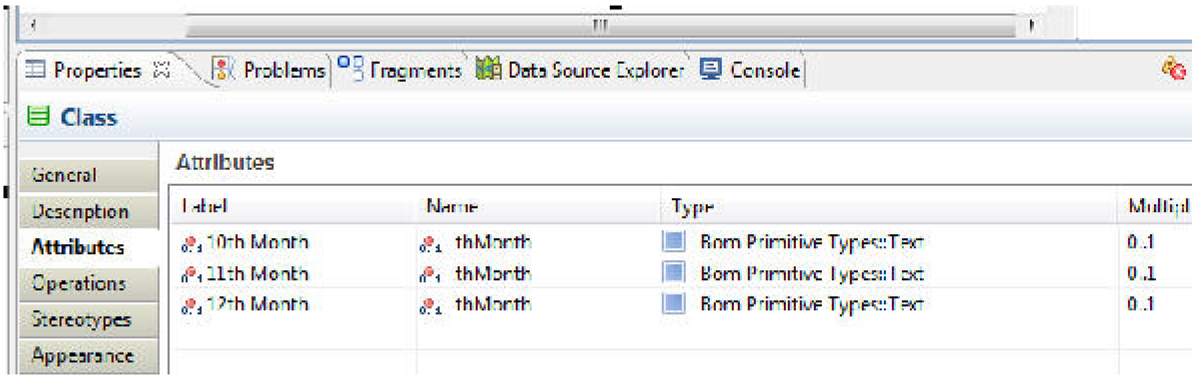
Reserved Words

There are certain reserved words that cannot be used as Names, but there are no such restrictions for Labels.

Words that are keywords in Java or JavaScript cannot be used as Class or Attribute names, as they would cause problems when running the Java script that referenced the class or attribute. A full list of reserved words can be found in Reserved Words in Scripts.

Name Clashes

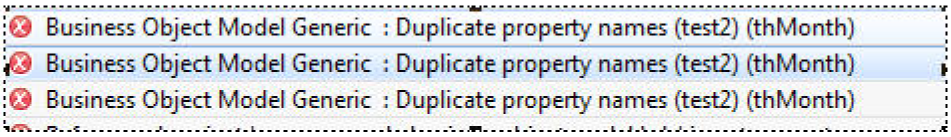
If two classes or attributes end up with the same name after the Label to Name mapping has been done, an error is generated against the BOM.



Attributes

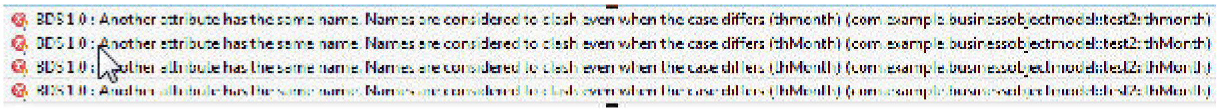
Label	Name	Type	Multipl
10th Month	thMonth	Bom Primitive Types::Text	0..1
11th Month	thMonth	Bom Primitive Types::Text	0..1
12th Month	thMonth	Bom Primitive Types::Text	0..1

Errors can be found in the Problems tab or by using the red-cross marker, as shown below:

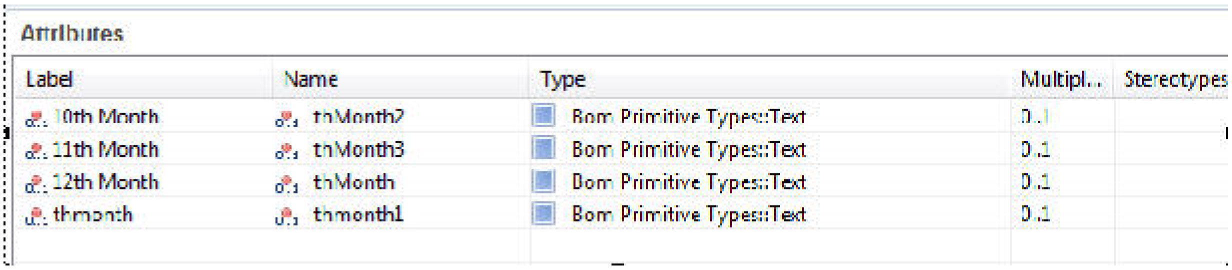

```
Business Object Model Generic : Duplicate property names (test2) (thMonth)
Business Object Model Generic : Duplicate property names (test2) (thMonth)
Business Object Model Generic : Duplicate property names (test2) (thMonth)
```

This is a generic BOM problem and has to be fixed manually.

Because of the internal workings of BDS, there are further restrictions on names that can cause other clashes. For example, attribute and class names must be unique and ignore case and underscores. Therefore, if another attribute is added with the name "thmonth", then BDS generates the following errors indicating a name clash:


```
BDS 1.0 : Another attribute has the same name. Names are considered to clash even when the case differs (thmonth) (com.example.businessobjectmodel::test2::thmonth)
BDS 1.0 : Another attribute has the same name. Names are considered to clash even when the case differs (thMonth) (com.example.businessobjectmodel::test2::thMonth)
BDS 1.0 : Another attribute has the same name. Names are considered to clash even when the case differs (thMonth) (com.example.businessobjectmodel::test2::thMonth)
BDS 1.0 : Another attribute has the same name. Names are considered to clash even when the case differs (thMonth) (com.example.businessobjectmodel::test2::thMonth)
```

This problem marker has a quick fix which can automatically rename an attribute to resolve any clashes, as shown below:



Attributes

Label	Name	Type	Multipl...	Stereotypes
10th Month	thMonth2	Bom Primitive Types::Text	0..1	
11th Month	thMonth3	Bom Primitive Types::Text	0..1	
12th Month	thMonth	Bom Primitive Types::Text	0..1	
thmonth	thmonth1	Bom Primitive Types::Text	0..1	

BDS Generation and Business Data Usage in TIBCO BPM

Business Data Services generate BDS Plug-ins from BOM Definitions. BDS Plug-ins are based on Eclipse Modeling Framework (EMF). BDS Plug-ins contain generated interfaces and classes that extend the base interfaces and classes defined in EMF, and are capable of being serialized and deserialized to or from XML very easily.

Using TIBCO BPM, you can create Business Objects from a variety of tasks in a BPM Process. For example, UserTask, ScriptTask, WebServiceTask, and so on. ScriptTask and Web-ServiceTask use JavaScript as the script grammar for interacting with Business Objects.

When a DAA (Distributed Application Archive) is created for a project containing a Business Object Model, or the project is deployed, the BDS Generator generates BDS Plug-ins that correspond to the BOM.

The BDS Plug-ins will be generated in a hidden project whose name corresponds to a BOM root package. For example, if the BOM root package **com.example.businessobjectmodel**, BDS Plug-ins are generated in a project named **.com.example.businessobjectmodel.bds**.

BDS Design-time Validations

During BOM creation, as well as processes that use BOMs, TIBCO Business Studio's Problems view may show various errors and warnings from BDS.

This section lists the validation messages that may be seen. Each problem is classified as either an error or a warning. Errors are shown with a red marker and will prevent application deployment until they are resolved, whereas warnings, shown with a yellow marker, are merely advisory. Unless otherwise noted, the messages listed in this section are errors.

Validations in BOM Editor

While BOM Editor is being used to produce a BOM, it checks to make sure that the BOM you have created is valid and alerts you to any problems that have been found.

For many of the issues, it can help you solve the problem through Quick Fixes. For example, if you have duplicate Names, or reserved names for Classes or Attributes, it will resolve the problem names by adding a numerical suffix.

Concerning the Model (Top-Level Package)

- Name must be dot-separated segments, each beginning with a letter or underscore and containing only letters, underscores and numbers, avoiding reserved words.
- Duplicate model name '*modelname*'.
- '*modelname*' cannot be the same name as the project life cycle id.

Concerning Sub-Packages

- Name must begin with a letter or underscore and contain only letters, underscores and numbers, avoiding reserved words.
- Name must not contain reserved words.

Concerning Classes, Enumerations, and Primitive Types

- Name must begin with a letter or underscore and contain only letters, underscores and numbers.
- Name must not be a reserved word.
- Name must not match the first segment of the fully-qualified package name.

- Another type has the same name. Names are considered to clash even when the case differs.
- Generated Java interface name '*name*' will conflict with the implementation class for '*nameofsomethingelse*'.
- Generated Java interface name '*name*' will conflict with the EMF factory interface for package '*packagename*'.
- Generated Java interface name '*name*' will conflict with the EMF factory implementation class for package '*packagename*'.
- Generated Java interface name '*name*' will conflict with the EMF package interface for package '*packagename*'.
- Generated Java interface name '*name*' will conflict with the EMF package implementation class for package '*packagename*'.
- Generated Java interface name '*name*' will conflict with an EMF utility class for package '*packagename*>'.

Concerning Attributes

- Another attribute has the same name. Names are considered to clash even when the case differs.
- Default values are ignored for attributes with a multiplicity greater than one [warning].
- It makes no sense having a default value for an optional attribute, as it will always apply [warning].
- Attribute name must begin with a letter or underscore and contain only letters, underscores and numbers. If it starts with two letters, they must be of the same case.
- Attribute name must not be a reserved word.
- The Attachment type is not supported.

Concerning Primitive Types

- The Attachment type is not supported.

Concerning Enumerations

- Enumeration must contain at least one literal.

Concerning Enumeration Literals

- Enumeration Literal name must begin with an upper-case letter and contain only upper-case letters and numbers.

Process Validations

BDS performs a number of validations on Processes that make use of BOMs:

- Primitive Types cannot be used for Data Fields or Parameters
- Enumeration Types cannot be used for Data Fields or Parameters.
- Activities responsible for generating WSDL operations cannot have array parameters associated. Instead, please create a Class to contain the array.
- Document literal type bindings must have formal parameters of type external reference.

BDS Runtime Validations

When a task completes, the BDS ensures that all BDS data is in a valid state. It does this by verifying that all constraints that exist on the fields are satisfied. If there are any problems, an exception is raised.

For example, if a field has a multiplicity of 3..6, then an exception is generated if the field has more than 6 values or less than 3 values as specified by the multiplicity rule. Similarly, lengths of Text and Fixed Length Numeric fields, ranges of numeric fields, and regular expressions for Text fields are all checked when each task completes.

Process Migration

If you want to be able to migrate a process instance from one version of a process template to a later one, you will need to make sure that the data in the first process template is compatible with the second process template version.

As much of the structure of the data comes from the BOM, you have to ensure that the BOM used by the two versions of the TIBCO Business Studio project is compatible. This means that the BOM used by the process template that you are migrating from must be a subset of the BOM used by the process template that you are migrating to.

Therefore, if you want to be able to migrate process instances from the old process template to the new version of the process template, you can only make compatible changes to the BOM. If incompatible changes are made to the BOM, there is a possibility that process instances will not be able to be migrated to the new version of the process templates.

A compatible change adds a new entity to a BOM, or makes an existing entity less restrictive, for example, the addition of a new class, or increasing the length of a text attribute from 50 to 60 characters. Examples of ‘incompatible changes’ include removal of a class, making an optional attribute mandatory, or adding a maximum value to an integer attribute that was previously unrestricted.

The following changes in the process template are considered as compatible when migrating a process instance from one version of a process template to a later one.

General Changes

Any BOM entity’s label can be changed (as long as the name remains the same).

Diagrams can be rearranged, annotated, and so on.

Changes Within the BOM’s Top Level or Sub-Package

- Addition of a new class, primitive type or enumeration.
- Addition of a sub-package.

Changes to a Class

- Addition of new attributes and composition attributes, as long as they are optional (for example, they must have multiplicity with a lower bound of zero, such as 0..1 or "*").

Changes to Class Attributes and Composition Relationships

The multiplicity of an attribute or composition relationship may be changed, as long as it makes it less restrictive (for example, it either increases the upper bound or decreases the lower bound) and it does not change between having a maximum multiplicity of 1 and greater than 1. Examples are given in the following table.

From	To	Valid?
1..5	1..8	Yes - increase in multiplicity
1	0..1	Yes – made optional
0..1	*	No (cannot change from single to many)
1	1..*	No (cannot change from single to many)
*	1..*	No (might have zero)
0..1	1	No (might have zero)
*	4..*	No (might have less than 4)

The attribute type cannot be changed. If an attribute's type remains the same, its restrictions may be altered, as long as they are less restrictive than the old restrictions.

Restriction	Permitted Change
Default value	May be changed
Lower limit	May be decreased or removed if a former lower limit was set
Upper limit	May be increased or removed if a former upper limit was set
Lower limit inclusive	May be changed from false to true
Upper limit inclusive	May be changed from false to true
Maximum text length	May be increased
Number length	May be increased
Decimal places	May be increased (if length increased by the same amount or more)
Pattern	Can be removed

Changes to an Enumeration

Addition of new enumeration literals.

Change to a Primitive Type

When the type has a BOM Native Type as its superclass, it may be altered subject to the compatible change rules described in Changes to Class Attributes and Composition Relationships.

Using BDS in Tasks

General

All Tasks have an Interface tab that can be used to restrict which fields the task has access to. By default, no fields are listed, which means there are no restrictions and the Task has access to all the fields in the process.

If the Interface tab is used, and a new field is added to the process, the new field will not be available in the Task until it is added to the Interface.

Each field that is specified in the Interface tab is specified to be one of the following:

- In
- Out
- In / Out

These specifications define whether the value is input or output to the task. There is also a Mandatory flag that can be specified, which controls whether the field has to have a value.

User Task

BOM fields can be displayed and updated in User Task steps. After a User Task has been completed, all the BOM fields that are In / Out or Out fields will be initialized.

Script Task

Any BOM fields that have not been initialized by a previous task have a null value, and therefore need to be initialized using the factory method before any of the attributes can be referenced. See Creating a New Business Object for more details on using BOM fields in Script Tasks, and using Scripts in general.

Forms

If the form generated from a task includes a sequence, choice, or group with multiplicity assigned to it, that multiplicity is not reflected in the form. See Passing Multiplicity to a Form for details of this restriction.

Defining Web Services

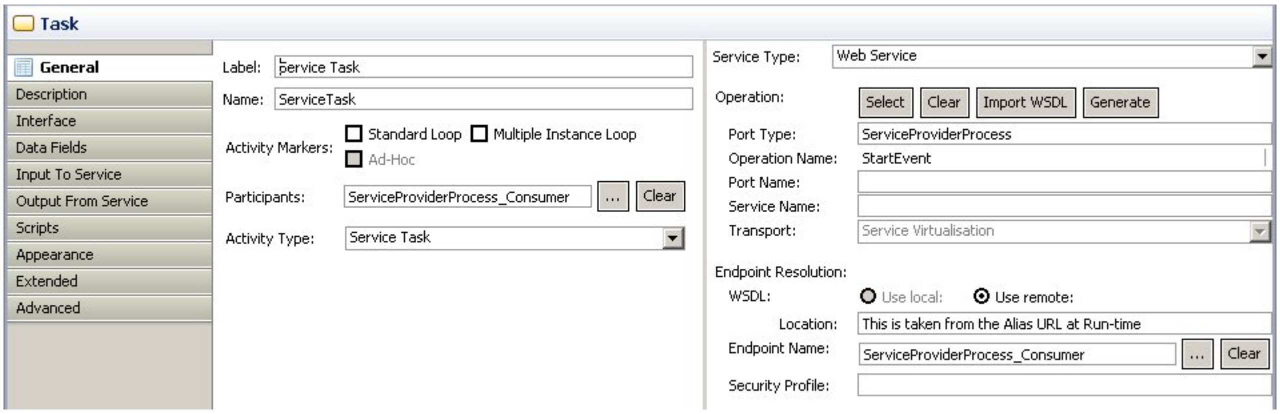
There are some restrictions as to what types of web services the Web Service Task can invoke. If you need to call a web service that uses WSDL features not supported by ActiveMatrix BPM, then the Mediation feature should be used, as it supports more WSDL features and is capable of mediation with other systems.

To define a web service task, do the following:

Procedure

1. In the General tab of the **Properties** view, type `Service Task` in the **Label** field.
2. Set the Service Type to **Web Service**.
3. If the WSDL has already been imported, click **Select** to select the service to be called. If the WSDL has not been imported, click **Import WSDL** to import it.

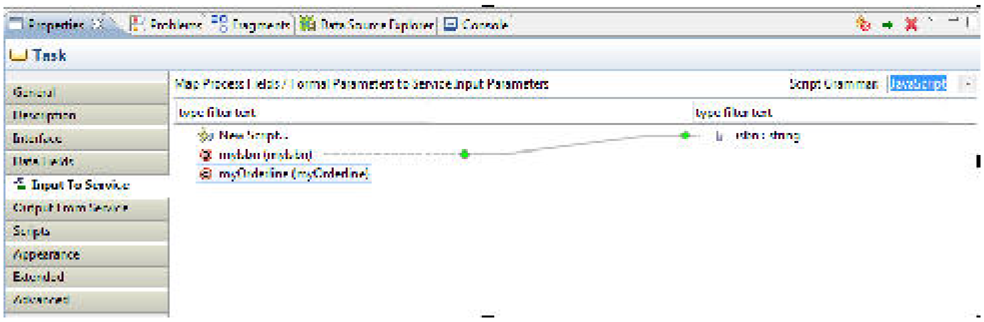
Result



Label: Service Task
Name: ServiceTask
Activity Markers: ☐ Standard Loop ☐ Multiple Instance Loop ☐ Ad-Hoc
Participants: ServiceProviderProcess_Consumer
Activity Type: Service Task

Service Type: Web Service
Port Type: ServiceProviderProcess
Operation Name: StartEvent
Port Name:
Service Name:
Transport: Service Virtualisation
Endpoint Resolution:
WSDL: ○ Use local: ● Use remote:
Location: This is taken from the Alias URL at Run-time
Endpoint Name: ServiceProviderProcess_Consumer
Security Profile:

Having defined the webservice to call, you need to map the input and output data from the web service call. If the process that is calling the webservice has a field of the same structure that the webservice takes as an input parameter, it can be mapped straight across on the Input To Service properties sheet. If not, the fields can be mapped individually. Similarly, if the process has a variable with the same structure as the response message, then it can be mapped on the Output From Service property sheet. The following example shows input fields being mapped:

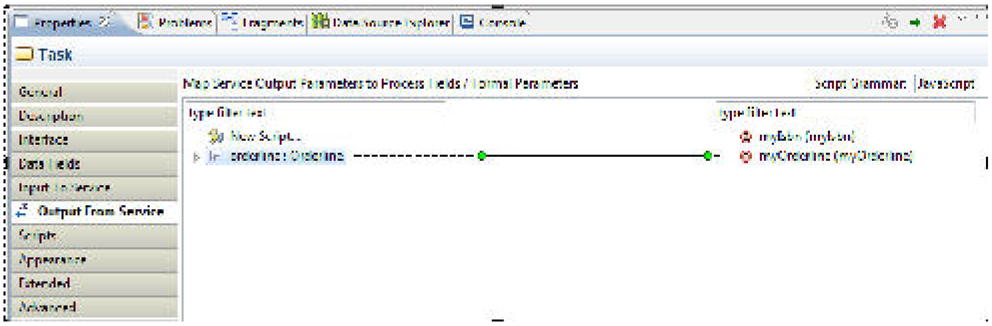


Map Process Fields / Formal Parameters to Service Input Parameters
Script Grammar: JavaScript

New Script...
myLabn (myLabn)
myOrderline (myOrderline)

idbn : string

The following example shows output fields being mapped, and a Business Object and all its attributes:



Map Service Output Parameters to Process Fields / Formal Parameters
Script Grammar: JavaScript

New Script...
orderline : Orderline

myLabn (myLabn)
myOrderline (myOrderline)

Business Data Scripting

Scripting in BDS uses a script language that is based on JavaScript with extensions to support the different aspects of BDS.

You can learn more about JavaScript from many sources. A useful introduction to JavaScript can be found at the W3Schools web site http://www.w3schools.com/.

Supplying xsi type Information in XML Sent to ActiveMatrix BPM gives an example of how to use Business Data Scripting in practice in ActiveMatrix BPM. It does not include a full description of the syntax of the JavaScript language. Those not familiar, or who are struggling with the syntax, are encouraged to first learn the basics of JavaScript before progressing onto ActiveMatrix BPM Scripting.

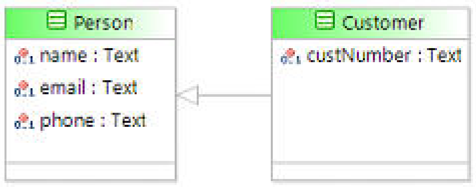
Supplying xsi:type Information in XML Sent to ActiveMatrix BPM

When using external clients to pass XML representing BOM information to ActiveMatrix BPM, you must provide an `xsi:type` when using extended types.

This is described in the specification:

.http://www.w3org/TR/xmlschema-1/#xsi_type

An example of this is shown below:



This example shows that if you wish to pass an XML into an interface that is expecting a Person from a Customer, it must contain an `xsi:type` as illustrated below:

```
<tns1:PersonElement xmlns:tns1="http://example.org/math/types/"
xsi:type="tns1:Customer">
    <name>Fred</name>
    <email>fred@myemail.com</email>
    <phone>01234 567890</phone>
    <custNumber>44556677</custNumber>
</tns1:PersonElement >
```

http://www.w3 BPM adds an `xsi:type` value to XML data passed to external systems in a verbose manner. This means that, wherever possible, the `xsi:type` is present. For clarity, this data is fully compliant with the use of `xsi:type` as described in the specifications.

Business Data Scripting by Example

The data objects that are passed around the ActiveMatrix BPM system, both within and between processes, can sometimes be mapped as whole objects from one process to another, or attributes of one object can be mapped onto attributes of another object in graphical ways using the mappers. However, there are some places where the processes require custom processing of the data beyond the direct mapping of attributes. In these cases, the scripting capabilities can be used.

This section illustrates how to write BDS server-side scripts, through a number of examples. The scripting is provided by the Business Data Services (BDS) component that allows process definers to manipulate the data objects defined within the BOM.

BPM Script scriptingcan be used in a lot of places within ActiveMatrix BPM processes by selecting **JavaScript** as the script grammar, for example:

- Script tasks within processes
- Action Scripts that are run on particular events related to tasks (for example, initiate, complete, timeout, cancel, open, submit and close)
- Timer Scripts - used to calculate a deadline or duration of a task within a process
- Condition Scripts – used to determine which direction flow should take within a process
- Loop Scripts – control how many times loops are executed within processes

The BPM Script is based on the JavaScript language, but has some unique restrictions and extensions, all of which are described later in this guide.

Factories

At runtime, when a new object needs to be created to represent a particular Business Object (instance of a BOM class) or other variable, a Factory method needs to be used to create the object instead of using the `new` keyword that is usually used in JavaScript.

Only values for attributes of the following types do not need to be created with Factories:

- Boolean
- Text
- Integer (Signed integer sub-type)
- Decimal (Floating point decimal sub-type)

The primitive field types that represent measurements of time are created by DateTimeUtil factory methods:

-
  ```
  DateTimeUtil.createDate()
  DateTimeUtil.createTime()
  DateTimeUtil.createDatetime()
  DateTimeUtil.createDatetimetz()
  DateTimeUtil.createDuration()
  ```

To create fixed integer and fixed decimal object instances, the following two ScriptUtil factory methods are used:

-
  ```
  ScriptUtil.createBigInteger()
  ScriptUtil.createBigDecimal()
  ```

Boolean fields can be assigned with the keywords `true` and `false`, or the result of an expression (such as `(2 == value))`. However, if you want to convert a text value true or false to a Boolean, then the ScriptUtil.createBoolean() can be used.

The factory methods for BOM classes can be found in Factory classes whose names matches the package name of the BOM, for example, for a BOM with a package name `com.example.ordermodel` the factory class would be called `com_example_ordermodel_Factory`, and there would be methods in the factory called createClassname for each class in the BOM. For example if the BOM contained Classes called Order, OrderLine, and Customer, there would be the following factory methods:

-
  ```
  com_example_ordermodel_Factory.createOrder()
  com_example_ordermodel_Factory.createOrderLine()
  com_example_ordermodel_Factory.createCustomer()
  ```

If the scriptingguide BOM contains a sub-package called `ordersystem`, there would be a factory created for the classes in that sub-package. Creating objects for classes in the sub-package would be done in a similar way to creating objects in the top-level package, for example:

```
order = com_example_scriptingguide_ordersystem_Factory.createOrder();
```

the name of the factory contains the package and sub-package hierarchy in its name, separated by "_".

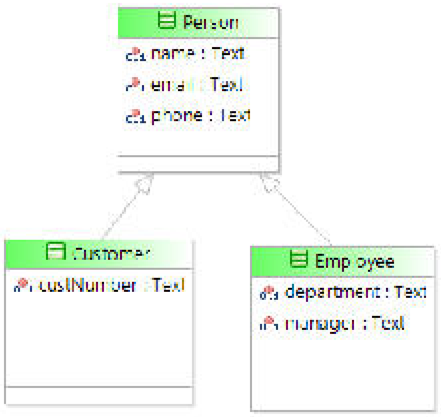
Creating a New Business Object

You can either create a new Business Object directly, or use a copy of an existing object.

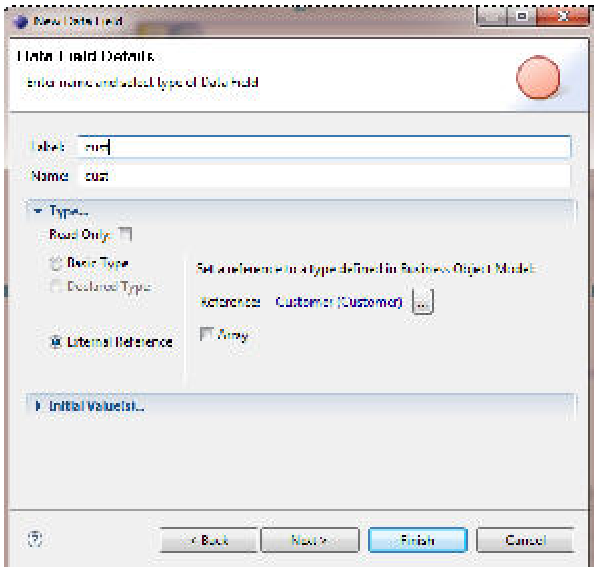
Create an Instance of a Class

You can create a new Business Object directly.

Taking the example of the Person/Customer/Employee BOM:



In order to create an customer instance, we first need a data field to hold the instance. This is done by creating a new Data Field in a process, and setting the Type to be an External Reference to a type in the BOM. Here we have created a field called `cust` that will be able to hold instances of the Customer class instances:



New Data Field

Data Field Details

Enter name and select type of Data Field

Label: cust

Name: cust

Type...

Read Only: ☐

Basic Type

Declared Type

External Reference

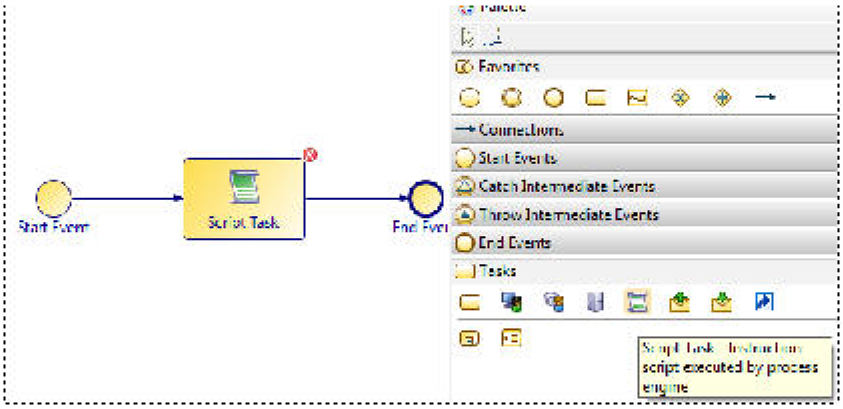
Set a reference to a type defined in Business Object Model:

Reference: Customer (Customer)

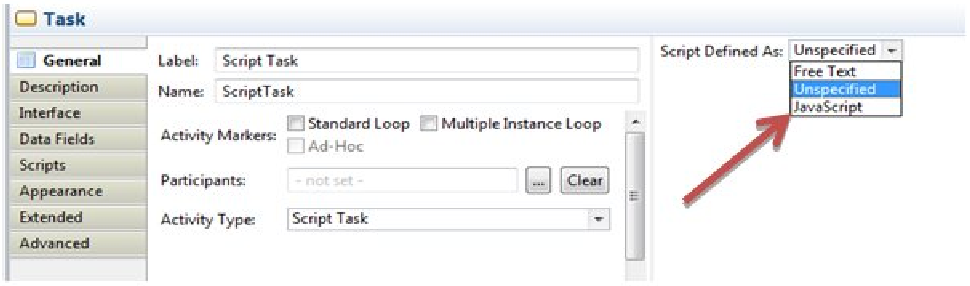
☐ Array

Initial Value(s)...

Then a Script Task can be dragged onto the process diagram from the Tasks palette:


Start Event
Script Task
End Event
Palette
Favorites
Connections
Start Events
Catch Intermediate Events
Throw Intermediate Events
End Events
Tasks
Script Task - Instructions script executed by process engine

View the Script Task properties and from **Script Defined As** menu, select **JavaScript**:



Label: Script Task

Name: ScriptTask

Activity Markers: ☐ Standard Loop ☐ Multiple Instance Loop ☐ Ad-Hoc

Participants: - not set -

Activity Type: Script Task

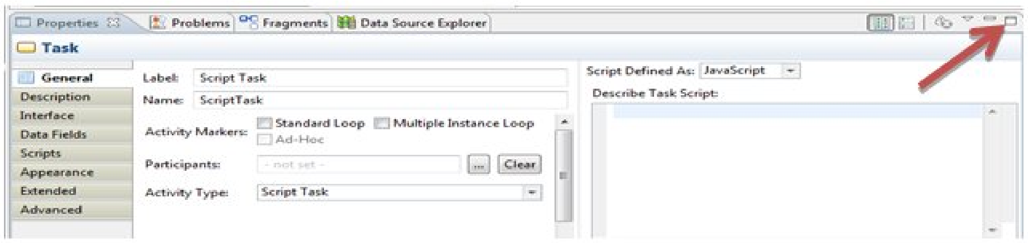
Script Defined As: Unspecified

Free Text

Unspecified

JavaScript

The Describe Task Script dialog is displayed. Type the script. Maximize the Describe Task Script dialog, by clicking **Maximize**, as shown below:



Label: Script Task

Name: ScriptTask

Activity Markers: ☐ Standard Loop ☐ Multiple Instance Loop ☐ Ad-Hoc

Participants: - not set -

Activity Type: Script Task

Script Defined As: JavaScript

Describe Task Script:

It is very important to understand that when a process starts, the `cust` process data field will not contain or refer to an instance of the Customer class, it just has the ability to. So the first thing to do before attempting to access the attributes of the process data field is create an instance of the Customer object and assign it to the `cust` process data field. The instances of the Customer class are created by the "Customer Factory Method", the name of the factory that creates Customer instances is based on the name of the BOM package that contains the Customer class as described in the previous section.

```
cust = com_example_scriptingguide_Factory.createCustomer();
```

One of the differences between BPM Script and standard JavaScript is the new operator is not supported. Factory methods have to be used to create objects.

Create a Copy of a Business Object

One way of creating a new Business Object is to create a copy of an existing object.

The ScriptUtil utility method, as shown below, is provided for doing this:

```
cust1 = com_example_scriptingguide_Factory.createCustomer();
cust2 = ScriptUtil.copy(cust1);
cust3 = cust1;
```

The script in the example above, sets the `cust2` process data field to refer to a copy of the Business Object that the `cust1` process data field refers to, and `cust3` to refer to the same Business Object that the `cust1` process data field refers to.

The `ScriptUtil.copy()` method performs a "deep" copy, which means that it copies all the objects contained by the Business Object being copied, as well as the Business Object itself. It is only for copying whole Business Objects, not for just copying BOM Primitive values.

Using the Special Value Null

The special value written as null can be used in several differnet ways.

Checking for Null Attributes

This section is intended chiefly for those readers not familiar with JavaScript.

The default value of the `cust` variable when the process starts is a special value written as null. If our script was running later on in the process, and there was a possibility that an earlier script might have set the `cust` variable to refer to a Customer, but it could still be null, then this can be checked in the script before calling the factory method, as shown below:

```
if (null == cust)
{
    cust = com_example_scriptingguide_Factory.createCustomer();
}
else
{
    // cust was assigned in an earlier script
}
```

There are several things to note here:

- `if (CONDITION) {IF-BLOCK} else {ELSE-BLOCK}` is used for testing and conditionally executing statements. Between the "()" (parenthesis mark), there should be a condition that results in a `true` or `false` result. If the value results in `true`, then the `IF-BLOCK` statements between the first "{}" (curly braces) are processed. If the value results in `false` the statements between the second curly braces in the ELSE-BLOCK are processed. There can be multiple statements between the curly braces. These are referred to as a block of statements. In BPM Script, curly braces are mandatory. In JavaScript, they are only required if there is more than one statement to be processed.

- The "==" operator is used to test for equality. Here, it is being used to test if the `cust` variable has the value null. Writing `null == cust` instead of `cust == null` can help if you forget to use " ==" and use" =" instead, since `cust = null` is valid in some places. However, `null = cust` is never valid, so the syntax checker would help you in this case.
- The "//" in the `else-block` is used to introduce a comment. The rest of the line following "//" is ignored by the script processing engine.
- If a UserTask is processed that has a BOM field as an Out or In / Out parameter in its Interface (the default for all fields is In / Out), then after the UserTask is complete the BOM field will always refer to an object, so it will not be necessary to initialize the BOM field from the Factory method in any scripts that follow the UserTask that outputs the field.
- This is also true for any other task that has a mandatory Out or In / Out parameter (the difference between UserTasks and Forms is that it always creates objects, even for Optional parameters).

Once we know that the `cust` field has a value, we can then set the name. We can check to see if attributes have been previously assigned by comparing them against null, although this will only work for attributes that do not have a default value. For example:

```
if (null == cust)
{
    cust = com_example_scriptingguide_Factory.createCustomer();
}
/*
Set the cust.name if not already set
*/
if (null == cust.name)
{
    cust.name = customerName;
}
```

The example above shows how to use a multi-line comment. The comment is opened with a "/*", then all text until a matching "*/" is ignored by the script engine.

Similarly, you should check that an attribute is not null before using any methods on an object, as shown below:

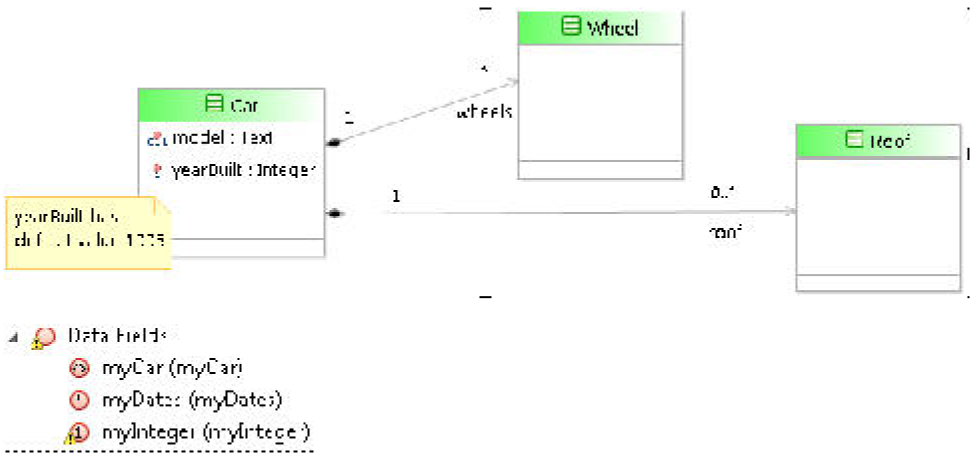
```
/*
 * Set the cust.name if not already set
 */
if (null != cust.dateOfBirth)
{
    year = cust.dateOfBirth.getYear();
}
```

Otherwise you will get a runtime exception.

Assigning a Null Value

The value of single value Data Fields and Business Objects' attributes and compositions can be cleared by assigning them the special value, null. If a default value is specified for a Business Object attribute, then assigning null will return the attribute to its default value.

The following diagram and script illustrate this:



```
// Clear car's model value (attribute)
myCar.model = null;
// Restore car's yearBuilt to its default: 1995 (attribute)
myCar.yearBuilt = null;
// Remove car's roof value (composition)
myCar.roof = null;
// Clear myCar Data Field (Business Object)
myCar = null;
// Clear myInteger Data Field (Integer basic type)
myInteger = null;
```

For Data Fields or Business Object attributes and compositions that have a multiplicity greater than one, the assignment of null is not possible. Instead, values can be removed using the appropriate List methods. Specifically, `remove` for the removal of a single specific value, or `clear` for the removal of all values. This is discussed further in Removing an Item from a List or a Containment Relationship. In the above example, this applies to Car's wheels composition and the myDates data field.

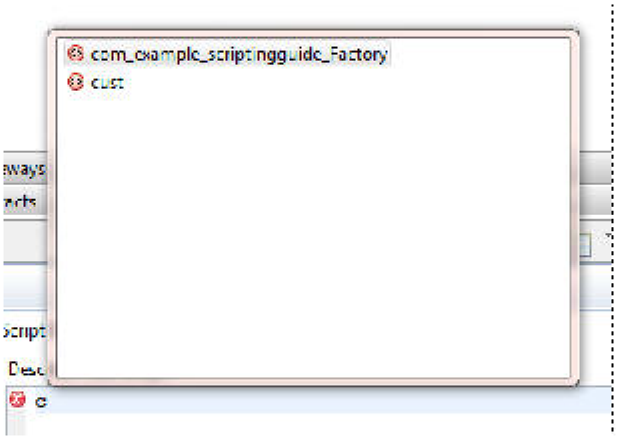
When dealing with attributes with a multiplicity greater than one, operations that add a null to the list will result in nothing being added, resulting in an unchanged attribute. For example, the following script is equivalent to a no-op, with no changes made to the list.

```
// Adding null to the list of a car's wheels does nothing
myCar.wheels.add(null);
```

Using Content Assist

TIBCO Business Studio can provide some helpful assistance when entering scripts.

If you cannot remember whether you had called the field `cust` or `customer` you can type the first few letters and press Ctrl+Space. You are prompted with a list of words, variables, methods and so on, that are appropriate for where you are in the script. So, in our example, we can type `c` then press Ctrl+Space. A list containing options appears, as shown below:



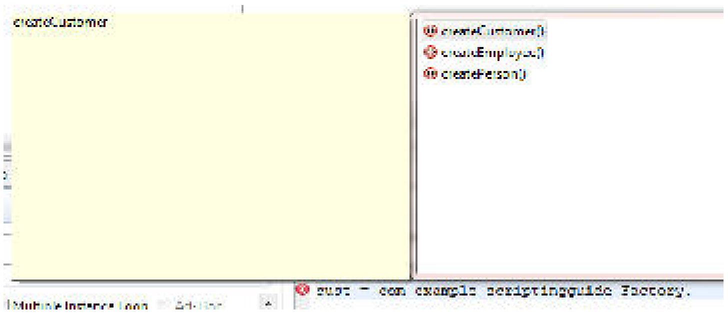
To insert `cust` in the script, you can:

- Select `cust` and press ENTER. A list of words, variables, methods and so on, associated with `cust` is displayed.
- Type `u`. Only items beginning with "cu" are displayed.
- Press ENTER.
- Double-click `cust`.

Next, type `=co` and press Ctrl+SPACE. Only the content assist that matches "co" in our example is displayed. Press ENTER to insert the Factory name into the script, as shown below:

```
cust = com_example_scriptingguide_Factory
```

Next, type "." to give a list of the factory methods. This allows you to choose the type of Business Object to create, as shown below:



```
cust = com_example_scriptingguide_Factory.
```

Since the Business Object we want is already selected, press ENTER to cause the text `createCustomer()` to be added to the script. Press ENTER to complete the line.

```
cust = com_example_scriptingguide_Factory.createCustomer();
```

Working with Single Instance Attributes of Business Objects

To add contact details to a Customer instance, you can write a script..

```
    if (null != cust)
    {
            cust.phone      = phoneNumber;
            cust.email      = emailAddress;
            cust.address    = postalAddress;
    }
```

Ensure that no variables used to assign attributes are `null`. Otherwise, the script will cause a runtime exception that can cause the process to fail when the script is run.

If the address attribute of the Customer class is an attribute of an Address type rather than a Text type, the address attribute needs to be set to refer to an instance of the Address class before the attributes of the `customer.address` can be set. For example, the following can be done:

```
if (null != cust)
{
    cust.phone       = phoneNumber;
    cust.email       = emailAddress;
    if (null == cust.address)
    {
        cust.address = com_example_scriptingguide_Factory.createAddress();
     }
    cust.address.street   = streetAddress;
    cust.address.district = districtAddress;
    cust.address.city     = cityAddress;
    cust.address.country  = countryAddress;
    cust.address.postcode = postCode;
}
```

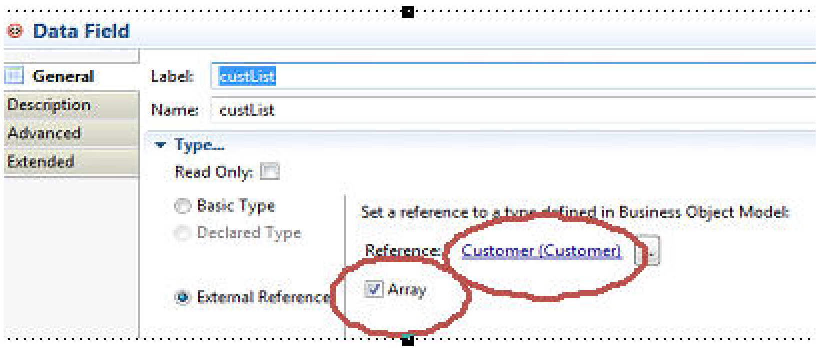
Multiple Instances of a BOM Class

It is possible for the process data field to refer to multiple instances.

Multiple Instances of a BOM Class in a Process Data Field

If a process data field is flagged as being an Array Field when the process data field is created (or its properties are changed later), then instead of referring to a single Business Object, the process data field will refer to multiple instances of the Business Object. This is done through a List object, which can contain multiple Business Objects.

Let us consider a process data field that holds a List of Customer objects called `custList`. The properties sheet for `custList` has **Array** selected and is of type Customer:


Data Field

General
Description
Advanced
Extended

Label: custList
Name: custList
Type...
Read Only: [ ]
Basic Type
Declared Type
[✓ External Reference]
Set a reference to a type defined in Business Object Model:
Reference: Customer (Customer)
[✓] Array

Array fields with a multiplicity greater than 1 are implemented using the List object. The List objects do not need to be created. They are created by default as empty Lists. If you want to associate a particular

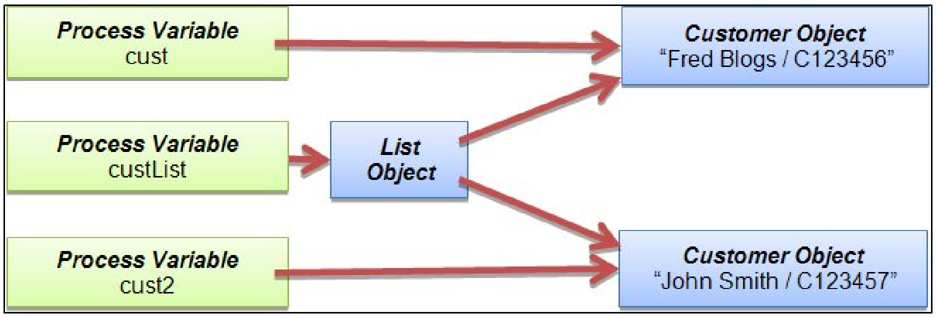
Customer with the `custList` variable, you can assign the `cust` field with a single instance field. This is shown below.

```
cust = com_example_scriptingguide_Factory.createCustomer();
cust.name = "Fred Blogs";
cust.custNumber = "C123456";
```

Another way is to add the new customer to the `custList`. We can add multiple customers to a list as well, as shown below:

```
// Using cust variable created above (Fred Blogs / C123456) is
// added to the List custList:
custList.add(cust);
// Now add a second customer to the list (John Smith):
cust2 = com_example_scriptingguide_Factory.createCustomer();
cust2.name = "John Smith";
cust2.custNumber = "C123457";
custList.add(cust2);
```

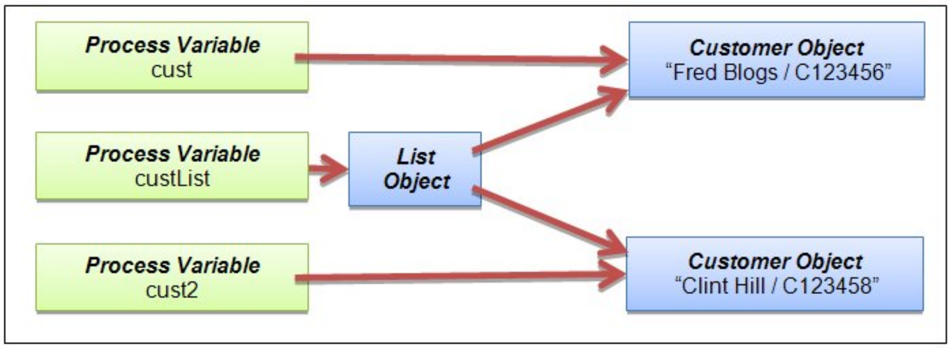
This can be pictured as follows:



WARNING: If, after you used the script above, you then used the following script to add a third customer to the list, this would go wrong on two accounts.

```
cust2.name = "Clint Hill";
cust2.custNumber = "C123458";
custList.add(cust2);
```

First, a new Customer instance has not been created for Clint Hill, so the first two lines above modify the John Smith Customer instance to refer to Clint Hill. Then when the `add()` method is called for the third time, it will attempt to add a second reference to the same Customer instance. However, this add will fail because the List type used does not allow the same object to be included more than once. So the list ends up containing the Fred Blogs and Clint Hill Customer instances but not John Smith:



Instead, a Customer instance must be created using the factory method for each Customer to be added to the list. If references to the individual customer `cust` are not required outside of the script then local script, variables can be used in place of the process variable `cust`. The example below shows two script local variables `c1` and `c2` being used to correctly add two Customers to a `custList`:

```
// Create first customer instance
var c1 = com_example_scriptingguide_Factory.createCustomer();
c1.name = "Fred Blogs";
c1.custNumber = "C123456";
custList.add(c1);
// Create second customer instance
var c2 = com_example_scriptingguide_Factory.createCustomer();
c2.name = "John Smith";
c2.custNumber = "C567890";
custList.add(c2);
```

It is not necessary to use different variable names for the two Customer instances, variable `c` could have been used throughout the script in place of `c1` and `c2`, but then the word `var` would have to be removed from the line that contains the second call to the `createCustomer()`:

```
var c = com_example_scriptingguide_Factory.createCustomer();
c.name = "Fred Blogs";
c.custNumber = "C123456";
custList.add(c);
c = com_example_scriptingguide_Factory.createCustomer();
c.name = "John Smith";
c.custNumber = "C567890";
custList.add(c);
```

Another way of creating the second and subsequent instances would be to make copies of the first. This can be useful if there are a lot of attributes with the same value, for example:

```
// Create first customer instance
var c1 = com_example_scriptingguide_Factory.createCustomer();
c1.name = "Fred Blogs";
c1.custNumber = "C123456";
c1.isRetail = true;
c1.dateAdded = DateTimeUtil.createDate();
custList.add(c1);
// Create second customer instance by copying the first
var c2 = ScriptUtil.copy(c1);
c2.name = "John Smith";
c2.custNumber = "C567890";
custList.add(c2);
```

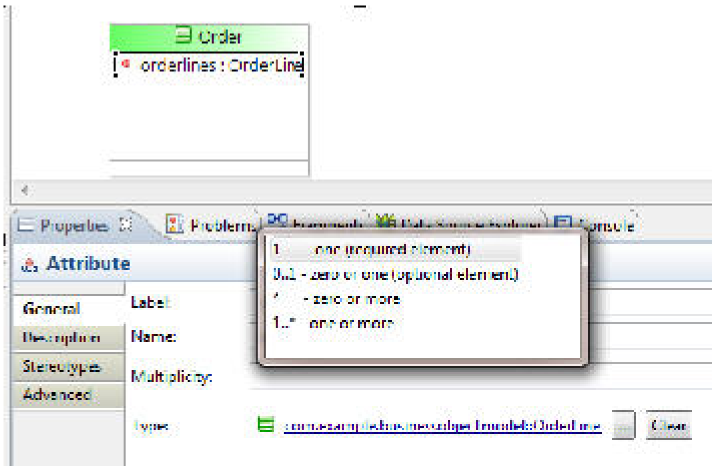
We can use the same `var` to store the new customer once the first customer has been added to the list. We do not need to keep a reference to it any longer:

```
// Create first customer instance
var cust = com_example_scriptingguide_Factory.createCustomer();
cust.name = "Fred Blogs";
cust.custNumber = "C123456";
cust.isRetail = true;
cust.dateAdded = DateTimeUtil.createDate();
custList.add(cust);
// Create second customer instance by copying the first
cust = ScriptUtil.copy(cust);
cust.name = "John Smith";
cust.custNumber = "C567890";
custList.add(cust);
```

Multiple Instances of a BOM Class in a BOM Class Attribute

Just as we defined Process Fields that contained Multiple Instances of Customer Business Objects in the previous section, we can also specify that an attribute of a BOM Class can have and must have multiple instances, for example, an Order may contain multiple OrderLine objects.

In BOM Editor, this is configured by setting the multiplicity of the attribute, as shown here:



The above screenshot shows some example values that can be used to specify the multiplicity, however, other values can also be used, for example, 3..6 would mean between 3 and 6 instances must be added to the field.

If the multiplicity is greater than one (e.g. "*"; "1..*" or "3..6") then a List is used to manage the field, and values must be added to the field using List methods. Otherwise, if the multiplicity is 1 (for example, multiplicity is "1" or "0..1"), then a straightforward assignment can be used.

When the multiplicity of an attribute is greater than one, a List is used to manage the data at runtime, in the same way Process Fields are managed when the Array checkbox is set in the Field Properties. To manage the multiple orderlines associated with an Order, the following script can be written.

```
var orderline = com_example_scriptingguide_Factory.createOrderLine();
orderline.partNumber = 10023;
orderline.quantity = 3;
order.orderlines.add(orderline);
orderline = com_example_scriptingguide_Factory.createOrderLine();
orderline.partNumber = 10056;
orderline.quantity = 1;
order.orderlines.add(orderline);
```

or using the `ScriptUtil.copy()` method:

```
var orderline = com_example_scriptingguide_Factory.createOrderLine();
orderline.partNumber = 10023;
orderline.quantity = 3;
order.orderlines.add(orderline);
orderline = ScriptUtil.copy(orderline);
orderline.partNumber = 10056;
orderline.quantity = 1;
order.orderlines.add(orderline);
```

The List object provides methods for finding out how many items there are in the list, accessing particular entries in the list, and enumerating the list. Some further examples of working with multi-instance fields and attributes are provided in the following sections:

- Looping Through Lists and Copying Elements in Lists.
- Scripting Containment Relationships.
- Using the List set() Method.
- Removing an Item from a List or a Containment Relationship.

To learn more about what you can do with the List object, see Using the List set() Method.

Multiple Instances in Sequences and Groups

BDS supports the use of multiplicity on a sequence, choice, or group in an imported schema or WSDL.

This support is only available on imported data. You cannot define multiplicity on a sequence, choice, or group in a user-defined BOM.

For example, an imported sequence might be defined in an XML schema as follows:

<xs:complexType name="PlaneOptionalElms">

<xs:sequence minOccurs="0" maxOccurs="unbounded">

<xs:element name="freightDetails" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>

<xs:element name="passengers" type="xs:int" minOccurs="0" maxOccurs="unbounded"/>

</xs:sequence>

</xs:complexType>

<xs:element name="PlaneOptionalElmsElement" type="PlaneOptionalElms"/>

The support for multiplicity on imported data means that, for example, important data passed in to your process over a web service, and defined with multiplicity on a sequence or a choice, will maintain the order in the content of the original message. However, you can retrieve the data only by requesting all occurrences of a given element within the sequence or choice. This applies when data is passed to scripts, mapping screens, or forms.

Note that if you use a script to populate or to add to a sequence or a choice with a multiplicity greater than one, the order in which the data is added in the script will be maintained in the order of the BOM class instance containing the data. So if you add data in a particular order, that same order will appear in the XML if the data is later passed over a web service.

Passing Multiplicity to a Form

When an ordered set of data produced by a sequence, choice, or group with a multiplicity greater than one is passed to a user task or to a form, the BOM Class involved can only be used as an **In** parameter or data field on a user task. This prevents any changes to the data altering the order of an already set sequence or choice.

If you wish to add more data to an existing sequence or to populate a new sequence, TIBCO suggests that you write a pageflow to collect the data from a form, and then use a script to copy the data into the sequence with multiplicity.

Working with Temporary Variables and Types

When writing scripts, the need for temporary variables often arises. In JavaScript, a temporary variable is declared using the `var` keyword. This saves having to add all variables as Process Data Fields, which is especially useful if the variables are only used within a single script. Adding them to the list of Data Fields would just end up complicating the process.

For example, to declare (that is, to tell the script engine about) two temporary variables called `x` and `carName` you can write:

```
var x;
var carName;
```

The variables can also be initialized (given initial values) when they are declared like this:

```
var x = 5;
var carName = "Herbie";
```

When writing BPM scripts, it is always best to initialize variables when they are declared so that TIBCO Business Studio's Script Validation and Content Assist code knows what type the variables are. If you do not initialize a variable, you will get the following warnings:

Unable to determine type, operation may not be supported and content assist will not be available

The content assist will not work. So, using the example from the previous section, it would not be a good idea to write:

```
var c1;
var c2;
c1 = com_example_scriptingguide_Factory.createCustomer();
c1.name = "Fred Blogs";
c1.custNumber = "C123456";
custList.add(c1);
c2 = com_example_scriptingguide_Factory.createCustomer();
c2.name = "John Smith";
c2.custNumber = "C123457";
custList.add(c2);
```

Instead, the variables should be initialized when they are declared:

```
var c1 = com_example_scriptingguide_Factory.createCustomer();
c1.name = "Fred Blogs";
c1.custNumber = "C123456";
custList.add(c1);
var c2 = com_example_scriptingguide_Factory.createCustomer();
c2.name = "John Smith";
c2.custNumber = "C123457";
custList.add(c2);
```

Loops Within Scripts

This section is intended chiefly for those readers not familiar with JavaScript.

There are three different JavaScript loop syntaxes that can be used in BPM Script, and one that cannot be used. The ones that are supported are:

```
while (CONDITION) { BLOCK }
do { BLOCK } while (CONDITION);
for (INITIALIZER; CONDITION; INCREMENT) { BLOCK }
```

One that is not supported is:

```
for (FIELD in ARRAYFIELD) { BLOCK }
```

Here is a simple `while` loop that loops until the `ix` variable becomes less than 0:

```
var result = "";
var ix = 10;
while (ix >= 0)
{
   result = result + " " + ix;
   ix = ix - 1;
}
```

Things to note:

- `result = result + " + ix;` can be written `result += " " + ix;` using abbreviations allowed in JavaScript.
- `ix = ix - 1;` can be written `ix--;` or `--ix;` using abbreviations allowed in JavaScript.
- `ix++;` or `++ix;` can be used similarly instead of `ix = ix + 1;` .
- The curly braces are required for loops in TIBCO Business Studio JavaScripts just as they are for if/ else statements.

The `do-while` loop is very similar, but the condition is evaluated at the end of the loop, so the loop is always executed at least once:

```
var result = "";
var ix = 10;
do
{
   result += " " + ix;
   ix--;
}
while (ix >= 0);
```

The `for` loop is similar to the `while` loop, but has two extra expressions in it. One is executed before the loop starts and the other is executed at the end of each loop. It results in a more compact script. Here is the equivalent for loop to the above `while` loop:

```
var result = "";
for (var ix = 10; ix >= 0; ix--);
{
   result += " " + ix;
}
```

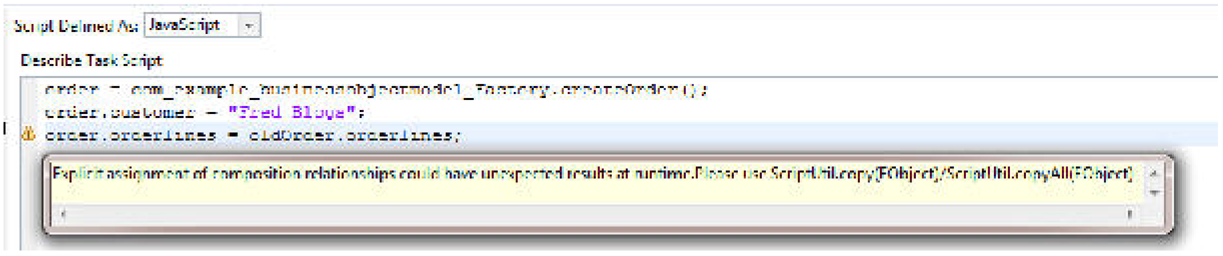
Looping Through Lists and Copying Elements in Lists

In order to iterate through a List, a ListIterator is used. First you need to obtain the ListIterator from the List using the `listIterator()` method. Two methods on the ListIterator are used to enumerate the items in the List: the `hasNext()` method returns `true` if there are more items in the list, and the `next()` method returns the next item in the list (or the first item in the list the first time it is called):

```
// Iterate through the list of customers to calculate the total
// credit limit:
var totalCreditLimit = 0;
for (var iterator=custlist.listIterator(); iterator.hasNext(); )
{
    // Get the first/next item in the list
    var customer = iterator.next();
    // add customer's credit limit to total credit limit
    totalCreditLimit += customer.creditLimit;
    // above statement is equivalent to:
    //totalCreditLimit = totalCreditLimit +     //          customer.creditLimit;
}
```

Note that when the three temporary variables: `totalCreditLimit`, `iterator` and `customer` are declared, they are all initialized so that TIBCO Business Studio knows what type they are. This helps with Script Validation and Content Assist.

If one of the inputs to a script is a field called `oldOrder`, which contains multiple Orderline Business Objects in an attribute called `orderliness`, and you want to copy these to another Order Business Object called `order`, then the `orderlines` List in the new Order object cannot just be assigned:



```
order = com_example_businessobjectmodel_Factory.createOrder();
order.customer = "Fred Bloggs";
order.orderlines = oldOrder.orderlines;
```

Explicit assignment of composition relationships could have unexpected results at runtime.Please use ScriptUtil.copy(EObject)/ScriptUtil.copyAll(EObject)

Instead, the instances in the List need to be copied over. You might think of doing it like this:

```
for (var iterator=oldOrder.orderlines.listIterator(); iterator.hasNext(); )
{
    var orderline = iterator.next();
    order.orderlines.add(orderline);
}
```

However, this should not be done, as Business Objects can only be contained in one Business Object. Attempting to add a Business Object to a second Business Object in a containment relationship could have unexpected consequences, and therefore should not be done (it will remove the object from the `oldOrder` Business Object). Instead, the content of the List needs to be copied over, as shown below:

```
for (var iterator=oldOrder.orderlines.listIterator(); iterator.hasNext(); )
{
  var orderline = iterator.next();
  var newOrderline = com_example_scriptingguide_order1_Factory.createOrderline();
  newOrderline.amount = orderline.amount;
  newOrderline.description = orderline.description;
  newOrderline.productCode = orderline.productCode;
  order.orderlines.add(newOrderline);
}
```

This can be simplified by using the `ScriptUtil.copy()` method to:

```
for (var iterator= oldOrder.orderlines.listIterator(); iterator.hasNext(); {
   var orderline = iterator.next();
   order.orderlines.add(ScriptUtil.copy(orderline));
}
```

It can be simplified even more by using the `ScriptUtil.copyAll()` method, as shown below:

```
order.orderlines.addAll(ScriptUtil.copyAll(oldOrder.orderlines));
```

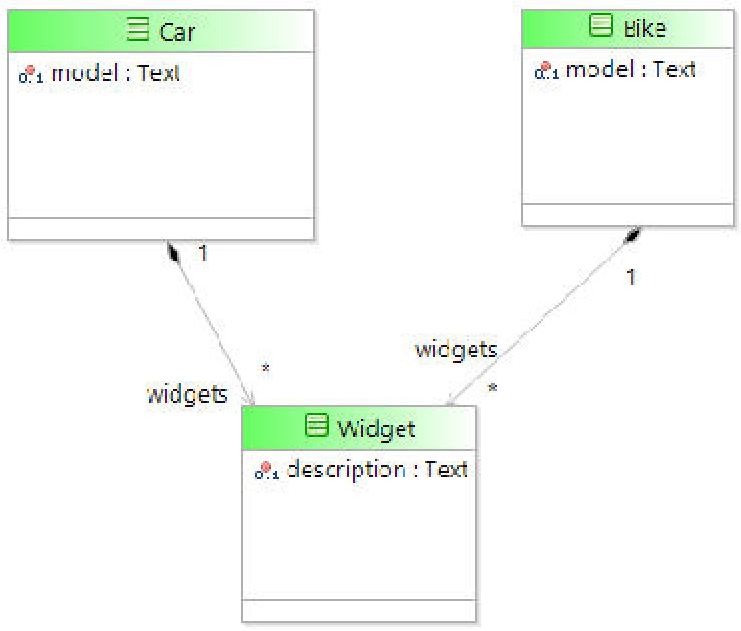
This method will copy all elements from the `oldOrder.orderlines` List to the `order.orderlines` List.

Scripting Containment Relationships

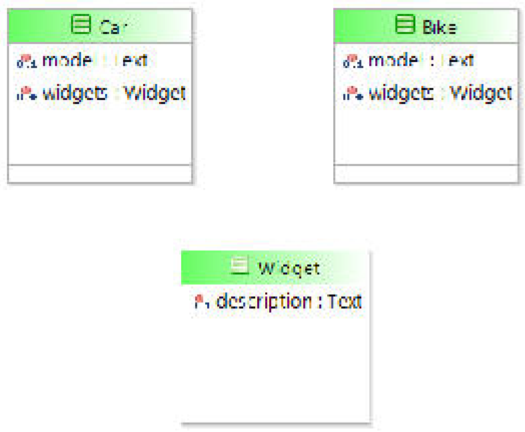
The BOM editor allows you to say that one class is contained within another. For example, if there were classes for Car and Widget, the Widget class can be said to be contained by the Car class. For the contained relationship, the contained objects are affected by the container's lifecycle events. So when the Container (Car in our example) is destroyed, all the contained (Widget in our example) objects will be destroyed too.

The diagram below shows that Widget objects can be contained by a Car or a Bike. However, for an individual Widget object instance, it can only belong to a Car or a Bike instance, not both at once, as when its parent object is destroyed, the child object is destroyed also.

There are two ways to model this in the BOM editor. First, a Composition link can be drawn between the two classes, like this:



Alternatively, the Car and Bike class can be given an attribute called widgets of type Widget, as shown below:



Both of these two Car/Widget relationships appear the same when scripting. There is an attribute of the Car object called widgets, which will be a List type, that can contain Widget objects. This would be processed in a similar way to the List processing examples already covered.

For example, to create a Car and add two Widgets, we can write:

```
var car = com_example_scriptingguide_containment_Factory.createCar();
car.model = "Saloon";
var widget = com_example_scriptingguide_containment_Factory.createWidget();
widget.description = "M8 Bolt";
car.widgets.add(widget);
widget = com_example_scriptingguide_containment_Factory.createWidget ();
widget.description = "M8 Nut";
car.widgets.add(widget);
```

We have already mentioned that the contained objects can only be contained by one container, so an object cannot be added to the same container more than once. Another aspect of the way that this relationship is enforced is that if an object is contained within container A, and it is then added to container B, as part of the process of inserting the object into container B, it is implicitly removed from container A.

For example, if you have a car object that contains a number of Widgets, and you attempt to copy them into another Car or Bike object using the following script, it will fail as described below:

```
for (var iter = car.widgets.listIterator(); iter.hasNext(); )
{
    bike.widgets.add(iter.next());
}
```

As mentioned above, adding the Car’s Widgets to the Bike removes them from the Car. This interferes with the iterator which is attempting to iterate over a changing list. When a list is being iterated over, it should not be changed unless by means of the ListIterator methods. Instead, the following should be done:

```
for (var iter = car.widgets.listIterator(); iter.hasNext(); )
{
    bike.widgets.add(ScriptUtil.copy(iter.next()));
}
```

The above script takes copies of the objects, and leaves the original copies contained in the Car object. You can also use the `copyAll()` method discussed in the previous section:

```
bike.widgets.addAll(ScriptUtil.copyAll(car.widgets));
```

Using the List set() Method

It is important to note that the List `set()` method cannot be used for adding new items to a list, for example, if you have an empty list, you cannot add two elements like this:

```
bike.widgets.set(0, widgetA);    // This is wrong!
bike.widgets.set(1, widgetB);     // This is wrong!
```

The reason this is wrong is that the `set()` method is for updating existing entries. The above will fail because the list is empty. Instead, the `add()` method must be used for adding new entries into a list. Existing entries can be directly updated, so `set()` may not even be needed:

```
var widget = bike.widgets.get(0);
widget.description = “Widget A”;
```

Removing an Item from a List or a Containment Relationship

In order to remove an item from a containment relationship or a list, the `remove()` method should be used. It can be used with the object to be removed, or the index of the object to be removed:

```
//Remove an object from a collection
order.orderlines.remove(orderline1);
```

or:

```
order.orderlines.remove(0);
```

Be careful using the first example above. The Business Object, or other value passed, must be the same Business Object instance that is in the list, and not a copy of it. This method checks to see if it is the same object that was added. It does not compare the contents of the objects.

If you don't know which item you want to remove, you should use the Iterator's remove method. To remove an item from a list means iterating through the list to find the item and then deleting it. This is done using the list iterator as we have done before:

```
// Iterate through the list of customers removing customers with large credit
// limit
for (var iterator=custlist.listIterator(); iterator.hasNext(); )
{
     var customer = iterator.next();
     // check if credit limit above 1,000,000
     if (customer.creditLimit >= 1000000)
     {
        iterator.remove();
     }
}
```

Alternatively this can be done just using the methods on the List object:

```
// Iterate through the list of customers removing customers with large credit
// limit
for (var ix=0; ix < custlist.size(); ix++)
{
     var customer = custlist.get(ix);
     // check if credit limit above 1,000,000
     if (customer.creditLimit >= 1000000)
     {
         custlist.remove(ix);
         ix--;  // decrement index so next iteration check the new nth item
      }
}
```

If the list index is managed in the script, as in the second example, you have to be careful not to skip past elements after an element that is remove by only incrementing the index if the item is not removed! So using the iterator is easier.

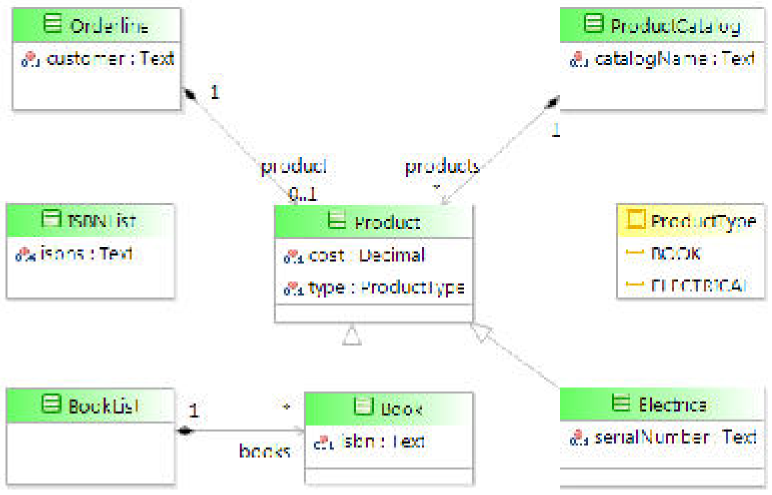
There is also a `clear()` method on Lists that can be used to remove all entries in the list, for example:

```
bike.widgets.clear();
custList.clear();
```

Scripting on Business Objects That Use Inheritance

Using BOM Editor, you can define BOM classes that inherit attributes from other classes. Another way of expressing this is to say that you can create BOM classes that specialize other classes.

An example of this is shown in the screenshot below:



In this example, there is a general Product type, and then there are two different types of Products:

- Book, which additionally have an `isbn` attribute
- Electrical, which additionally have a `serialNumber` attribute.

If we had a Process Data Field called `orderline` of type Orderline, then the containment of type Product, can be assigned Business Objects of type Product, Book, or Electrical:

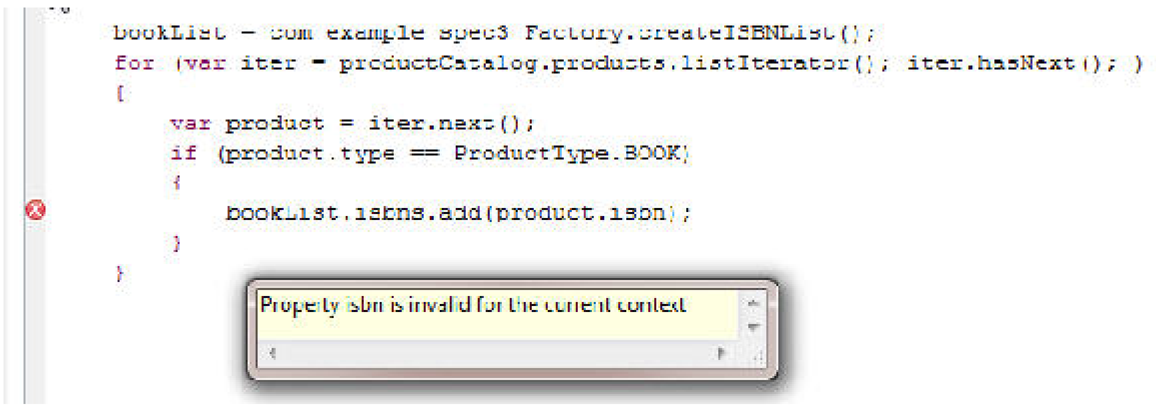
```
var book = com_example_spec3_Factory.createBook();
orderline.product = book;      // Set the product attribute to a Book
                           // Business Object
var elec = com_example_spec3_Factory.createElectrical();
orderline.product = elec;  // Set the product attribute to an
                           // Electrical Business Object
var prod = com_example_spec3_Factory.createProduct();
orderline.product = prod;  // Set the product attribute to a
                           // Product Business Object
```

Similarly, if we have a Process Data Field called `productCatalog` of type ProductCatalog, the products containment can contain many Product Business Objects, some of these which can Book Business Objects, some Electrical Business Objects, and some may just be Product Business Objects. For example, you can add all three types to the products list, as shown here:

```
var book = com_example_spec3_Factory.createBook();
var elec = com_example_spec3_Factory.createElectrical();
var prod = com_example_spec3_Factory.createProduct();
productCatalog.products.add(book);
productCatalog.products.add(elec);
productCatalog.products.add(prod);
```

If a Process Data Field productCatalog of type ProductCatalog type appears on a Form in a UserTask, it will just show the Product details. You will not be able to access the attributes of the Book or Electrical classes, however, it does allow you to add new instances of Book, Electrical, or the base type Product.

Similarly, in scripts, if you iterate through the products List, the Script Editor just gives you the content assist for the Product object. You will get an error if you attempt to access the `isbn` attribute of a Product Business Object, as shown below:



```
bookList = com_example_spec3_Factory.createISBNList();
for (var iter = productCatalog.products.listIterator(); iter.hasNext(); )
{
    var product = iter.next();
    if (product.type == ProductType.BOOK)
    {
        bookList.isbns.add(product.isbn);
    }
}
```

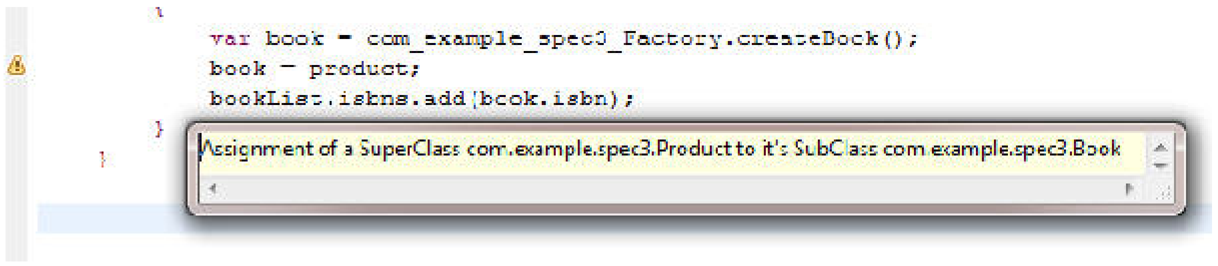
Property isbn is invalid for the current context

The above code sample shows an attempt to create a List of Text that contains the ISBN number of the Product instances that are Book subtype Book instances.

To get around this problem, we can create a variable called `book`, which we initialize to an object of type Book. Then we assign the Product Business Object to the book variable, after which TIBCO Business Studio allows access to the attributes of the Book class. However, a warning is given that the assignment may not work at runtime:

```
bookList = com_example_spec3_Factory.createISBNList();
for (var iter = productCatalog.products.listIterator(); iter.hasNext(); )
{
     var product = iter.next();
     if (ProductType.BOOK == product.type)
     {
         var book = com_example_spec3_Factory.createBook();
         book = product;
         bookList.isbns.add(book.isbn);
     }
}
```

The TIBCO Business Studio Script Editor gives a warning about the assignment of a field that it thinks contains a Product to a field that it treats as holding a Book. However we know, from the test that we conducted previously, that in this case it is safe:



```
        {
            var book = com_example_spec3_Factory.createBook();
            book = product;
            bookList.isbns.add(book.isbn);
        }
    }
```

Assignment of a SuperClass com.example.spec3.Product to it's SubClass com.example.spec3.Book

If we do not have the above check, then the above code can fail at runtime when attempting to access the `isbn` attribute if the product (and hence the book) was referring to a Product or an Electrical Business Object as these do not have the `isbn` attribute.

It is always OK to assign a sub-type (specialized type) object to a supertype (generalized) attribute or variable because you can say that the sub-type object satisfies the "is-a" relationship. In our example, Book "is-a" Product.

However, it is not always OK to do things the other way around. Assigning an attribute or process data field that is a Book type, from a variable or attribute of a Product type, will only work at runtime if the Product actually refers to an instance of the Book class (or a sub-type). If the Product field or attribute actually refers to a Product or Electrical Business Object, then it does not satisfy the "is-a" Book condition. The assignment will fail when the value is saved at the end of the task.

If, instead of building up a list of ISBN Text values, we wanted to create a List of Products that were also Books and, if the `bookList` is a Book type Process Data field, then we can write:

```
bookList = com_example_spec3_Factory.createBookList();
for (var iter = order.products.listIterator(); iter.hasNext(); )
{
    var product = iter.next();
    if (ProductType.BOOK == product.type)
    {
       bookList.add(product);
    }
}
```

However, if the booklist refers to a BookList type Business Object with an attribute or composition relationship called Books, then instead of writing:

```
bookList = com_example_spec3_Factory.createBookList();
for (var iter = order.products.listIterator(); iter.hasNext(); )
{
     var product = iter.next();
     if (ProductType.BOOK == product.type)
     {
         bookList.books.add(product);    // THIS IS WRONG
     }
}
```

We should write:

```
bookList = com_example_spec3_Factory.createBookList();
for (var iter = order.products.listIterator(); iter.hasNext(); )
{
     var product = iter.next();
     if (ProductType.BOOK == product.type)
     {
        bookList.books.add(ScriptUtil.copy(product));
     }
}
```

Otherwise, we are moving the Book Product out of the containment relationship with the order and products into the relationship with booklist/books. Remember that a contained Business Object can only be in one container at a time. To stop this from happening, you must make a copy of the object, and add that to the booklist/books containment relationship.

Working with Strings or Text Fields

String values can easily be assigned using either single or double quotation marks. However, they must be of the same type.

For example:

```
var firstString = "Hello World!";          // quoted using double quote character
var nextString = 'Hello Fred!';                // quoted using single quote
character
var thirdString = "Fred's World";          // includes single quote so used double
                                           // quote
var fourthString = ' "The Old House" ';    // includes double quote so used
                                           // single quote
var fifthString = "Fred's \"Old\" House"; // string includes both so need \
                                           // character to escape use of quote in
                                           // string
```

String values can be compared with the "==" operator, for example:

```
if (firstString == nextString)
{
    // do something
}
else
{
    // do something else
}
```

There are a number of methods on the String class that can be used to manipulate the value of the String object, for example, considering the following String variable:

```
var str = "Hello TIBCO!";
```

The following operations can be done on the String.

String Operations

Expression	Result	Comment
`str.length`	12	Returns length of string
`str.substr(0,5)`	Hello	Return substring starting at offset 0, 5 characters long
`str.substr(6)`	TIBCO!	Return substring starting at 6th position in string
`str.slice(6,9)`	TIB	Returns substring starting at offset 6 and finishing before offset 9
`str.slice(-6).toLowerCase();`	tibco!	Returns substring starting 6 characters from end of String and changes all letters to lowercase
`str.slice(0, str.indexOf(" "))`	Hello	Returns first word in string, or whole string if one word
`str.slice(str.lastIndexOf(" ")+1)`	TIBCO!	Returns last word in String, or whole string if one word

For more information about String class methods, see Text (String) Methods.

The String objects are immutable, so when one of the above functions returns a String value, it is a reference to a new String. The original String is not changed.

If you want to restrict what Strings can be put into certain Text fields, consider using the User-defined Types described in Working with Primitive Types.

Working with Booleans

Boolean fields can be simply assigned from constants, other Boolean fields, or expressions.

For example:

```
customer.initialized     = true;
customer.isOnCreditHold = false;
customer.staffDiscount  = memberOfStaff;
customer.isWholesale    = ! isRetailCustomer;
```

The exclamation mark "!" is the "not" operator, changing the sense of a true value to false, and a false value to true.

When attempting to convert a text field value to a Boolean (for example, from "true" to `true`), the `ScriptUtil.createBoolean()` method should be used. If the text field is not exactly true or false, attempting to assign a text field to a Boolean will generate an exception. Using the `createBoolean()` method if the value of the Text field is TRUE (in any case), then the Boolean result is true, otherwise it is false.

Similarly, if you want to convert a numeric value (0 or 1) to a Boolean, then the `ScriptUtil.createBoolean()` method should be used. For example:

```
Customer.isTrade = ScriptUtil.createBoolean(isTradeParameter);
```

can be used to convert from a text (true/false) or numeric (1/0) Boolean representation to the Boolean type. Values greater than or equal to 1 get converted to true, and values less than or equal to 0 get converted to false.

Boolean values can be compared with the "==" and "!=" operators, for example:

```
if (cust1.isWholesale == cust2.isWholesale)
{
   …
```

Boolean values can also be combined with the following logical operators.

Operators that can be used with Boolean Values

Operator	Description	Example
&&	And – both are true	cust.isWholesale && order.discountApplied
||	Or – either is true	cust.isWholesale || order.discountApplied
!	Not – reverses result	!(cust.isWholesale && order.discountApplied)

Working with Numeric Types

BDS supports different types of both Integer and Decimal numbers.

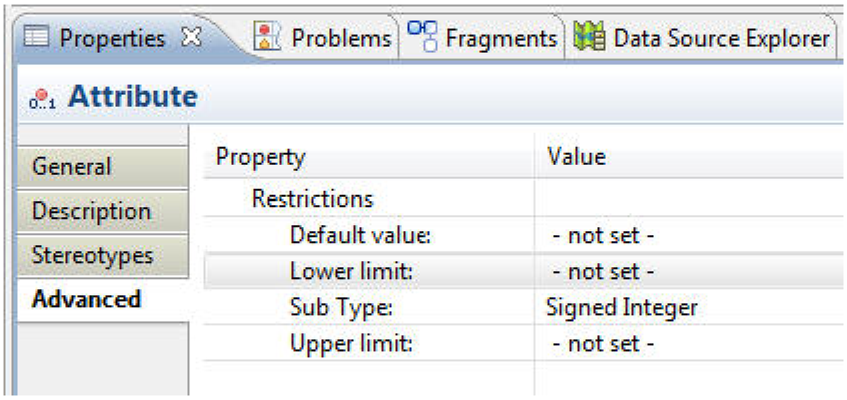
Working with Basic Integer Numbers

There are two types of integers (whole numbers such as 1, 2, 457, and so on) that are supported: Signed Integers and Fixed Length Integers. When working with integer numbers, you need to be aware of what the largest value is that you could be dealing with.

If you are dealing with smaller numbers, for example, a number of people, then the signed integer type can cope with numbers up to 2,000,000,000 (actually, numbers up to 2,147,483,647, or 231-1), however, if you are dealing with larger, for example, astronomical numbers, then the fixed form of integers needs to be used. The larger Fixed Integers are dealt with in Advanced Scripting Examples.

Signed Integers

For smaller numbers, either form of integers can be used, but the Signed Integer sub-type is easier to use from a scripting point of view, so it is probably the sub-type of choice. In order to select the sub-type, select the attribute in the BOM class, and look at the Advanced Properties sheet:



Property	Value
Restrictions	
Default value:	- not set -
Lower limit:	- not set -
Sub Type:	Signed Integer
Upper limit:	- not set -

In scripting, to work out the average weight of a team member, you can do the following:

```
var totalKgs = 0;
var teamSize = 0;
for (var iterator = team.members.listIterator(); iterator.hasNext(); )
{
     var member = iterator.next();
     totalKgs = totalKgs + member.weightKgs;
     teamSize = teamSize + 1;
}
if (teamSize > 0)
{
     team.averageWeight = totalKgs / teamSize;
}
else
{
     team.averageWeight = 0;
}
```

Note that the two lines in the loop that update the running totals can be shortened to:

```
totalKgs += member.weightKgs;
teamSize++;
```

using the arithmetic abbreviations that can be used in scripting.

When dividing, do not divide by 0. The code above checks for this special case. The operators for comparing signed integers are:

Operators for Comparing Signed Integers

Operator	Description	Example	Result
==	Equals	1 == 2 12 == 12	false true
!=	Not Equals	1 == 2 12 == 12	true false

Operator	Description	Example	Result
<	Less than	1 < 2 12 < 12 21 < 20	true false false
<=	Less than or equals	1 <= 2 12 <= 12 21 <= 20	true true false
>=	Greater than or equals	1 >= 2 12 >= 12 21 >= 20	false true true
>	Greater than	1 > 2 12 > 12 21 > 20	false false true

See Working with Fixed Length Integers (BigInteger) for more information.

Working with Basic Decimal Numbers

Just as Integers have two variants, there are two variants of Decimal attributes in BOM classes: Floating Point and Fixed Point. The sub-type to use is selected in the Advanced Property sheet for the attribute, just as the Integer attributes in Signed Integers.

The Floating Point variant can store:

- Negative numbers between -1.79769E+308 and -2.225E-307
- 0
- Positive values between 2.225E-307 and 1.79769E+308.

The Floating Point variant stores numbers with 16 significant digits of accuracy. However, it should not be used for storing values that have to be exact, for example, money amounts. Rounding errors may occur, especially if large amounts are involved.

The Fixed Point variant can store numbers to an arbitrarily large size, and can perform arithmetic using many different types of rounding as required. However, the downside is that the Fixed Point attributes are implemented as BigDecimal objects, which, like the BigInteger objects in the previous section, have to be manipulated using their methods (add(), subtract, divide(), multiple(), and so on.) instead of the normal arithmetic operators ("+", "-", "/", "*", and so on.).

For more information about the Fixed Point decimal attributes, see Advanced Scripting Examples.

Here are some examples that demonstrate how decimals can be used in scripts. For Floating Point decimals, assuming that averageWeight is now a Floating Point decimal attribute of the team Data Field, we can write:

```
var totalKgs = 0.0;
var teamSize = 0;
for (var iterator = team.members.listIterator(); iterator.hasNext(); )
{
     var member = iterator.next();
     totalKgs   = totalKgs + member.weightKgs;
     teamSize   = teamSize + 1;
}
team.averageWeight = totalKgs / teamSize;
```

As in the integer example, the two additional lines can be abbreviated as:

```
totalKgs += member.weightKgs;
teamSize++;
```

To compare Floating Point decimal values, use the standard "<", "<=","==", "!=", ">=" and, ">" operators. It is possible that two numbers that appear to be the same may not be equal using the "==" operator due to rounding errors in the way that the numbers are represented. (These operators are the same as for the Signed Integers. For more information about how these operators can be used, see Operators for Comparing Signed Integers.)

As an example of how Floating Point numbers may not be exactly as they seem, the result of the following expression is false due to rounding errors:

```
(((1/10) * 14) == 1.4)
```

To get around this problem, the values should be rounded before comparison.

Rounding of Floating Point Decimals can be done using the `ScriptUtil.round()` method, for example, to convert a number to 3 decimal places you can write:

```
roundedValue = ScriptUtil.round(value, 3);
```

This converts 1234.56789 to 1234.568 using HALF_UP rounding. If you wanted to round down, then the `Math.floor()` method can be used instead of the `ScriptUtil.round()` method:

```
roundedValue = Math.floor(value*1000)/1000;
```

The `ScriptUtil.round()` method can also be used to round to a power of 10, for example, to round to the nearest 100, the power of 10 is 2, so use:

```
roundedValue = ScriptUtil.round(value, -2);
```

This rounds 1234.56789 to 1200. The Math class provides other methods, for example, log & trig functions and `random()` and `floor()` functions. See Math Methods for details.

See also Working with Fixed Decimals (BigDecimal).

Implicit Conversions Between Numeric Types

Data of one type can be converted to another type either implicitly or explicitly.

This section describes some of the implications of the *implicit* conversions that are supported between different numeric data types. *Explicit* conversions can be carried out using the factory methods provided, as described in Supplemental Information.

When data of one numeric type is converted to another, there is not always any simple direct conversion, for example if you store a decimal value into a non-decimal field, or a BigInteger into an Integer.

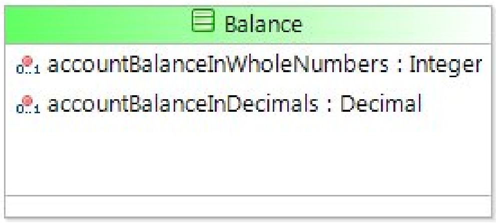
In general:

- If a decimal is stored into a non-decimal (such as BigDecimal stored into BigInteger), then the part of the data will be discarded. No rounding up is performed in this case: "9.99" is stored as "9".
- If the maximum numeric size is exceeded (such as BigInteger stored into Integer) in such a way that the numeric value cannot be stored in the target type, then a **NumberFormatException** will be generated.

Support for assigning different numeric types

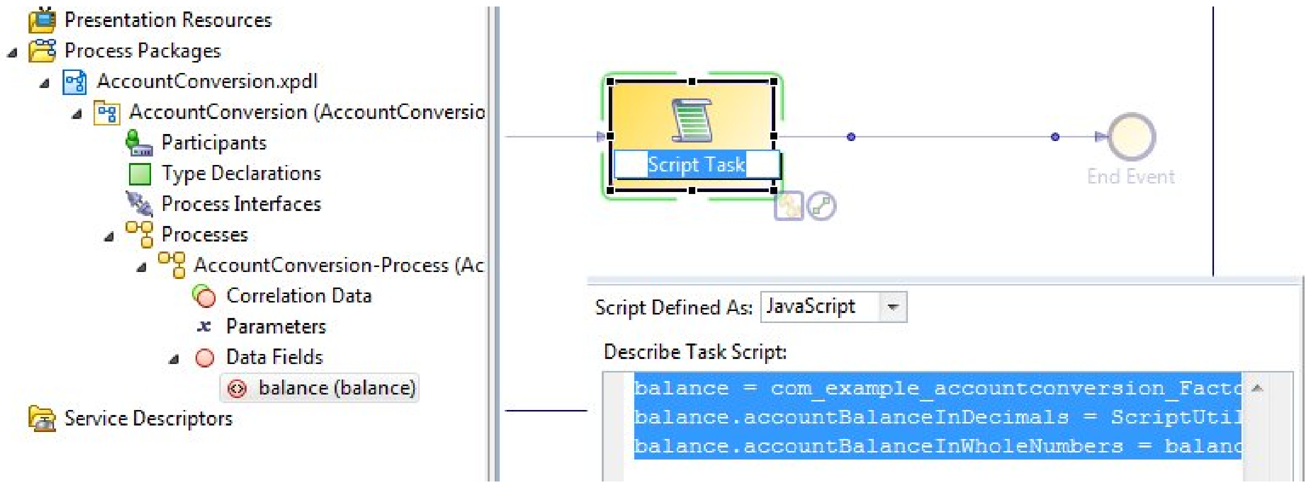
Data of one numeric type can be converted to another simply by assigning an object of one type to a value of a different type. In the following example, a BigDecimal item is converted to a BigInteger value.

A business process used by a bank contains a BOM class **Balance**, with two attributes:


Balance
accountBalanceInWholeNumbers : Integer
accountBalanceInDecimals : Decimal

- **accountBalanceinWholeNumbers** This is an integer, of sub-type Fixed Length (BigInteger), to provide an approximate value for the balance.
- **accountBalanceInDecimals** This is a decimal, of sub-type Fixed Point.

In the process, a data field **balance** is defined as an external reference to the BOM class **Balance**. The field is used in a script task, as shown in the following illustration.



Script Defined As: JavaScript

Describe Task Script:

```
balance = com_example_accountconversion_Facto
balance.accountBalanceInDecimals = ScriptUtil
balance.accountBalanceInWholeNumbers = baland
```

An extract from the script is shown both in the illustration above and in the snippet that follows:

```
/*
Map the integer accountBalanceInWholeNumbers to the decimal
accountBalanceInDecimals
*/
balance = com_example_accountconversion_Factory.createBalance();
balance.accountBalanceInDecimals = ScriptUtil.createBigDecimal("100.1");
balance.accountBalanceInWholeNumbers = balance.accountBalanceInDecimals;
```

BDS checks which data type is being passed in as input and performs the conversion accordingly. In this example, the code would convert the decimal 100.1 to the integer value 100.

Support for adding to Lists

BDS supports adding an item of one numeric type to a List object that is listing items of a different numeric type: for example, adding a Floating Point decimal to a list of Integers.

For example, if the variable approxWeight in the example below is an integer, and weightObservations is a list of these integers, you can add a decimal preciseWeight value to the list.

```
// Using the approxWeight variable already created, a new
// weight is added to the List weightObservations:
weightObservations.add(approxWeight);
...
// It is necessary to add to the list a more precise observation
// The fact that this is a decimal makes no difference to the
// script user:
weightObservations.add(preciseWeight);
```

So if the following observations are added to the list:

approxWeight = 200;
preciseWeight = 324.26;
approxWeight = 196

The weightObservations list would contain the values [200, 324, 196] after the script has run.

As this example shows, some data may be lost when converting from decimal to integer.

Working with Dates and Times

Dates and Times

The Date, Time, Datetime, and Datetimetz types are represented using a XMLGregorianCalendar object within the Script Engine. This provides methods to manipulate the date/time type variables and attributes.

The date/time attributes and fields can be initialized in scripts using methods on the DateTimeUtil factory, as shown below.

DateTimeUtil Factory Methods

Factory Method	BOM Type	Comment
createDate()	Date	This type is used to hold a Date, for example, 1st January 2011.
createTime()	Time	This type is used to hold a time, for example 4:25 P.M.

Factory Method	BOM Type	Comment
createDatetime()	Datetime	This type is used to hold a date and time, with an optional timezone offset.
createDatetimetz()	Datetimetz	This type is used to hold a date and time, with a mandatory timezone offset.

With no parameters these methods create an object representing the current date or time, or alternatively, they can be given a string or other parameter types to construct the appropriate object with the required value. One thing to be aware of if using String values to create date/time objects is that all the relevant fields need to be specified or an exception will be thrown. Specifically, the seconds need to be specified for all types except the Date type. The following are some examples of date/time types.

```
time = DateTimeUtil.createTime("17:30:00");
time = DateTimeUtil.createTime(17,30,0,0);    // equivelant to “17:30:00”
date = DateTimeUtil.createDate("2010-12-25");
date = DateTimeUtil.createDate(2010, 12, 25);    // equivelant to “2010-12-25”
datetime = DateTimeUtil.createDatetime("2010-12-25T15:00:00");
datetime = DateTimeUtil.createDatetime("2010-12-25T15:00:00Z");
datetime = DateTimeUtil.createDatetime("2010-12-25T15:00:00+05:00");
datetimetz = DateTimeUtil.createDatetimetz("2010-12-25T15:00:00Z");
datetimetz = DateTimeUtil.createDatetimetz("2010-12-25T15:00:00+05:00");
```

When initializing the datetime, the timezone is optional, but when initializing the datetimetz, the timezone is required. The timezone can be designated using one of the following formats:

- `Z` for Zulu, or Zero, timezone offset (GMT or UTC time).
- `+HH:MM` for timezones that are ahead of UTC time, e.g. Berlin timezone is `+01:00`.
- `-HH:MM` for timezones that are behind UTC time, e.g. USA Pacific Time has a timezone of `-08:00`.

When working with global data, you should use Datetimetz to represent any timezone-dependent Datetime values entered from a form. This is because global data automatically sets any Datetime/Datetimetz values to UTC, meaning that dates in forms may not be the same ones you originally created (they could be one day different due to timezone differences).

See the reference section for more choices of parameter values, for example, for separated parameters for year, month, and day when create date types.

Durations

An important attribute/variable type when dealing with dates and times is the Duration type, which is used to hold periods of time, such as 1 year, minus 10 days, or 2 hours.

A duration object can be created in a similar way:

```
duration = DateTimeUtil.createDuration("P1Y");        // 1 year
duration = DateTimeUtil.createDuration("-P10D");        // minus 10 days
duration = DateTimeUtil.createDuration("PT2H");        // 2 hours
duration = DateTimeUtil.createDuration("PT23.456S");     // 23.456 seconds
duration = DateTimeUtil.createDuration("P1DT2H");    // 1 day and 2 hours
```

When constructing a period of time from a string, always begin with a `P` for `period`. Then add *n*`Y`, *n*`M`, or *n*`D`, where *n* is a number of years, months, or days. Any fields that are zero can be omitted. If there is any time component to the Duration, a `T` must follow the date parts, followed by *n*`H`, *n*`M`, or *n*`S` for specifying hours, minutes, or seconds.

A leading minus sign can be used to create a negative duration. This can be used with the `add()` method to subtract a time period. The Duration type can also be created by specifying each of the components as integers with a flag to say whether the duration is positive:

```
duration = DateTimeUtil.createDuration(true, 1, 0, 0, 0, 0, 0);    // 1 year
duration = DateTimeUtil.createDuration(false,0, 0,10, 0, 0, 0);    // minus 10
days
duration = DateTimeUtil.createDuration(true, 0, 0, 0, 2, 0, 0);    // 2 hours
duration = DateTimeUtil.createDuration(true, 0, 0, 0, 0, 0, 23.456);    // 23.456
seconds
duration = DateTimeUtil.createDuration(true, 0, 0, 1, 2, 0, 0);    // 1 day and 2
hours
```

Using Date and Time Types with Durations

One important point to be aware of is that the date/time (XMLGregorainCalendar) objects are not immutable, so the `add()` and `setXXX()` methods update the object that the method is on, rather than return a new value.

To add 2 hours onto a Datetime, write:

```
datetime.add(DateTimeUtil.createDuration(“PT2H”));
```

Not:

```
datetime = datetime.add(DateTimeUtil.createDuration(“PT2H”));
```

The second code results in datetime being set to null, since the `add()` method does not return a value. Duration objects are Immutable, like BigDecimal and BigInteger objects.

If you want to subtract a time period from a date or time type, you can add a negative duration. This is the same as in normal arithmetic where there are two ways of taking 2 from 10. The result of 10 - 2 is the same as 10 + -2. In order to subtract durations, we must use the format of adding a negative amount. The following example calculates 1 year ago.

```
var date = DateTimeUtil.createDate();    date.add(DateTimeUtil.createDuration("-
P1Y"));
```

The following example calculates a datetime corresponding to 36 hours ago.

```
var datetime = DateTimeUtil.createDatetime();
datetime.add(DateTimeUtil.createDuration(false,0,0,0,36,0,0));
```

Comparing Dates and Times

In order to compare two date/time types, either the `compare()` or `equals()` method should be used. The `equals()` method is just a wrapper around the `compare()` method. The `compare()` method should be used to compare items that either do or do not have a timezone. The method still works if the date/times are more than 14 hours apart, but if they are less than 14 hours apart, the result is deemed to be indeterminate.

An example of using the `compare()`method to compare two date fields is shown below:

```
// Verify that end date is greater than start date
if (enddate.compare(startdate) == DatatypeConstants.GREATER)
{
     // End date is greater than start date
}
```

Do not use the XMLGregorianCalendar `compare()` method in the same way you would use the `compareTo()` method to compare BigInteger and BigDecimal objects due to the possibility of it returning `INDETERMINATE`. To check for greater than or equals, use the following:

```
// Verify end date is greater than or equal to start date
if (enddate.compare(startdate) == DatatypeConstants.GREATER  ||
enddate.compare(startdate) == DatatypeConstants.EQUAL)
{
     // End date is greater than or equal to start date
}
```

Using the following will also include the INDETERMINATE result:

```
// Verify end date is greater than or equal to start date
if (enddate.compare(startdate) != DatatypeConstants.LESSER)
{
     // End date is greater than or equal to start date - OR INDETERMINATE!!
}
```

The XMLGregorianCalendar class does not provide a method for finding the difference between two XMLGregorianCalendar objects, so one is provided in ScriptUtil. To find out how many days have elapsed since the start of the year, write:

```
// Calculate date of first day of the year by getting current date
//   and setting day and month to 1
var startOfYear = DateTimeUtil.createDate();
startOfYear.setDay(1);
startOfYear.setMonth(1);
// get today's date
var today = DateTimeUtil.createDate();
// Subtract the start of year from today to work out how many days have elapsed
var duration = ScriptUtil.subtract(today, startOfYear);
// Extract the days from the duration type and add 1
dayOfYear = duration.getDays() + 1;
```

You can read about the different methods that are available on the date/time attributes in Business Data Scripting.

Working with Enumerated Types (ENUMs)

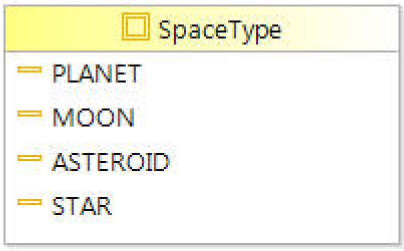
If you want to categorize objects as different types, instead of using a number or a free format string, use an Enumerated Type (ENUM). Enumerated Types provide a better solution because they are restricted, in that they can only take a fixed limited number of values. The names of the values can be made meaningful.

The use of ambiguous enumerations in script, when two enumerations with the same name exist in the same xpdl package, is managed by using a fully-qualified name (qualified by the package name) for the enumerations in the script.

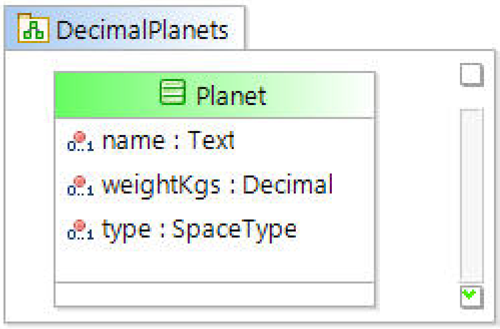
The qualified name of enumerations to be used in script is similar to the Factory names, with the qualified name formatted to replace dot '.' by '_' an underscore character. For example, `com.example.shared.ColorEnum` will be used as `com_example_shared_ColorEnum` in script.

Unqualified names (for example, `Color.GREEN`) are supported **only** in validation for unambiguous situations (the unqualified name will not be available in content assist).

An ENUM is created in the BOM editor by selecting the Enumeration type from the Elements section of the Palette. Having selected the Enumeration Element, it can be named, and values can be added to it. The following is an example of an Enumerated type called SpaceType, with PLANET, MOON, ASTEROID, and STAR values:



Having defined the Enumeration type, a class attribute can be set to that type:



The following is an example of a loop, which can be used to calculate the average weight of the planets in a list of astronomical bodies.

```
var dTotalKgs    = 0.0;
var dPlanetCount = 0;
for (var iterator = solarSystem.objectList.listIterator(); iterator.hasNext(); )
{
     var body = iterator.next();
     if (SpaceType.PLANET == body.type)
     {
         dTotalKgs = dTotalKgs + body.weightKgs;
         dPlanetCount ++;
     }
}
solarSystem.averagePlanetWeight = dTotalKgs / dPlanetCount;
```

A Business Object attribute that is configured to be of a particular Enumeration type can only be assigned with values of that enumeration type. Either constants of that type, such as:

```
    body.type = SpaceType.PLANET;
```

or other attributes of that type, such as:

```
    aggregation.type = body.type;
```

An attribute of an enumeration type cannot be assigned from any other type. For example, the following is not valid:

```
body.type = "PLANET"; // This is wrong!
```

The Enumerated Type also enables you to get a specific enumeration literal from its text value. For example, if you want to pass an enumeration value as a string to an external application and then pass it back to a process. You can use this in scripts with all available enumeration data types. Either text data types, as in the following example:

```
spaceTypeEnum = SpaceType.get('PLANET');
```

and non-text data types, such as:

```
OrderSizeEnum = OrderSize.get('100');
```

For non-text data types, the mapped text value must be the text used in the enumeration value and not its enumeration attribute name. For example, for an integer `OrderSizeEnum (Large-100, Medium-50, Small-20)` the mapping must use the values '100', '50', '20', and not the enumeration attribute names, Large, Medium, Small.

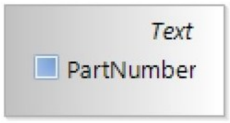
Working with Primitive Types

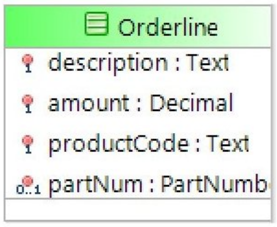
If you want to have field that can contain a restricted set of values, but the set of values is too big for an enumerated type, you can use a Primitive Type. For example, if you need to store a Part Number in a field, this field will probably have a restricted format, such as `PN-123456`. If all Part Numbers have a fixed format like this, set up a User-defined Type to hold the Part Number, and restrict the type so that it only holds strings that start with `PN-` and are followed by six digits.

To do this, select the Primitive Type from the Elements section of the Palette. Having created one of these, you can name it. For our example, call it PartNumber. In the Advanced Properties sheet, you can define restrictions that are imposed on the field, such as numeric ranges for numeric fields and patterns for text fields. The patterns are specified using regular expressions. For our example, use:

```
PN-\d{6}
```

Which means `PN-` followed by 6 digits (`\d` is the code for a digit and `{6}` means six of the previous entity).





Set a class attribute to contain a PartNumbertype attribute as shown above. In scripts and forms, you can only assign values to the partNum attribute that matched the pattern `PN-\d{6}`.

Attributes of Primitive Types can be assigned in the same way as the BOM Native Type on which they are based, so, using the above example, the partNum field can be assigned using:

```
order.orderline.partNum = "PN-123456";
```

If a script is written with an invalid format value, as shown in the example below:

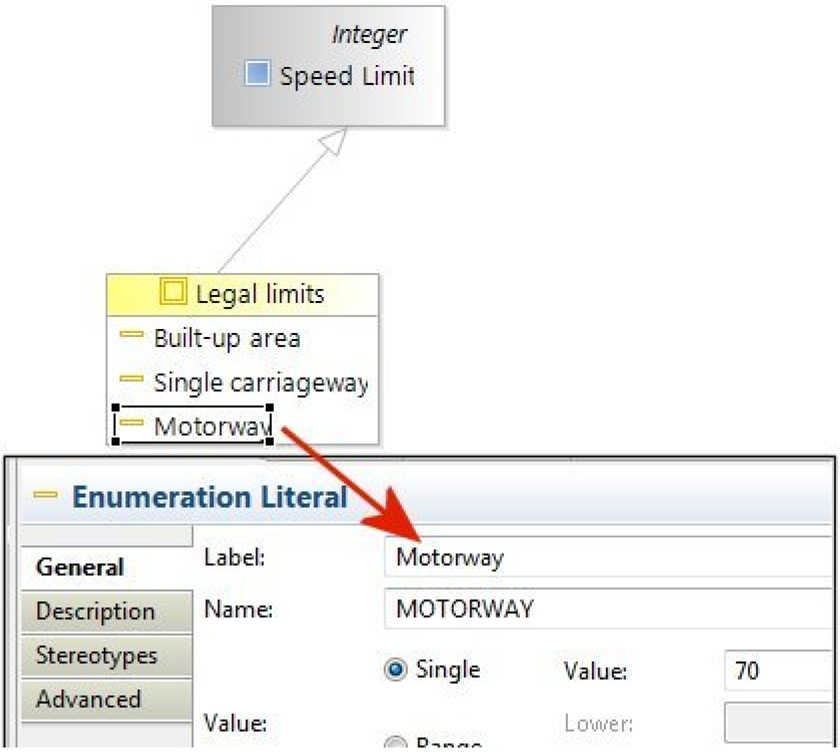
```
order.orderline.partNum = "ROB-123456";
```

the script editor will not detect this as an error, since it does not check that Strings have the correct content. Instead, this will cause a runtime validation exception when the Task that contains the script completes.

Using Enumerated Types as Extensions of Primitive Types

A BOM Primitive Type, of any type (superclass), can be mapped as a Generalization of an Enumeration in order to extend the primitive type's range of possible values.

For example, the following illustration shows an Integer primitive type, **Speed Limit**. The possible kinds of **Speed Limit** are given as enumeration literals within an Enumeration called **Legal limits**. The value of each of these enumeration literals is an integer representing the speed limit currently in force in the circumstances described by the enumeration literal. For example, the literal **Motorway** has the value of **70**, because the speed limit on motorways is 70 mph. If this speed limit were to increase to 80 mph, for example, the value of this literal could be changed to 80.



Label: Motorway
Name: MOTORWAY
Value: 70

Return Values from Scripts

The following expressions use these return values from scripts to perform certain functions:

- Conditional flows – Boolean expressions that control whether a path is followed or not.
- Loop Conditions – Boolean expressions that determine whether a loop should continue or not.
- Task TimerEvent – Datetime expressions that determine when a task should timeout.

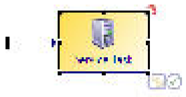
These expressions can be multi-line expressions. The value of the script is the value of the last line in the script. For example, if we want a script to calculate a timeout to be the end of the month, which is 7 days in the future, we can use the following:

```
var datetime = DateTimeUtil.createDatetimetz();        // get current datetime
datetime.add(DateTimeUtil.createDuration("P7D"));    // change to 7 days time
datetime.setDay(1);                                // adjust in case on 31-Jan
datetime.add(DateTimeUtil.createDuration("P1M"));    // move on to next month
datetime.add(DateTimeUtil.createDuration("-P1D"));    // back to end of prev.
month
datetime.setTime(0,0,0);                            // clear hours, minutes & seconds
datetime;                                            // return value of script
```

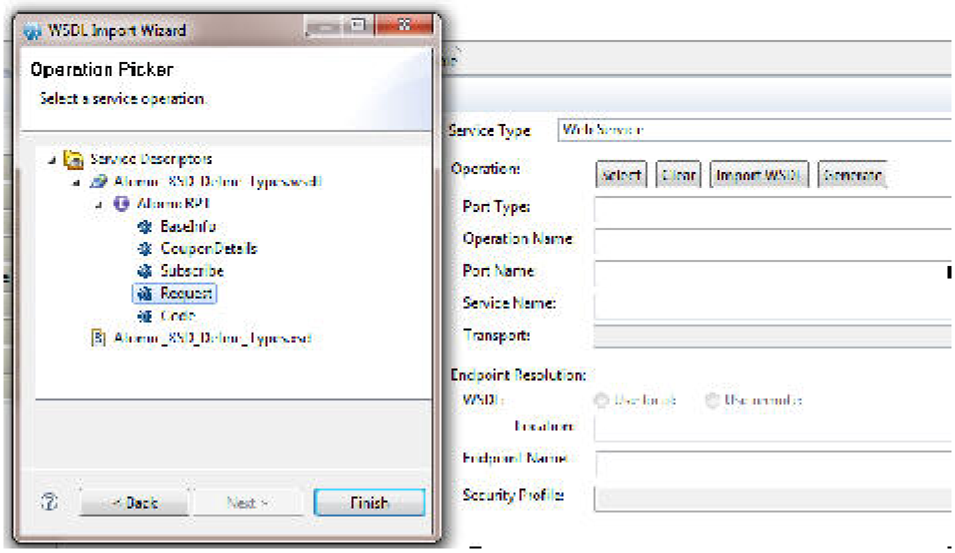
Scripting with Web Services

To invoke a web service and use scripts to prepare data for the web service, you perform the tasks described in this topic.

First, add a Service Task onto the process diagram by dragging it from the tasks palette:



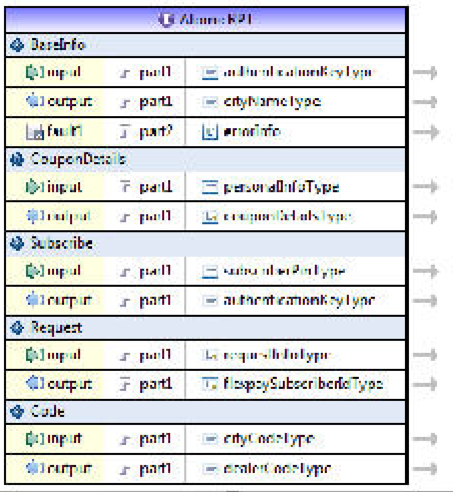
Then, in the General Properties sheet, set the Service Type to **Web Service**, then click **Import WSDL**. Locate your WSDL using one of the mechanisms provided (file location, URL, and so on), and select the webservice you want to invoke from the WSDL:



Service Descriptors
- AlarmRPI_XSD_Defined_Types.wsdl
  - AlarmRPI
    - BaseInfo
    - CouponDetails
    - Subscribe
    - Request
    - Code
- AlarmRPI_XSD_Defined_Types.xsd

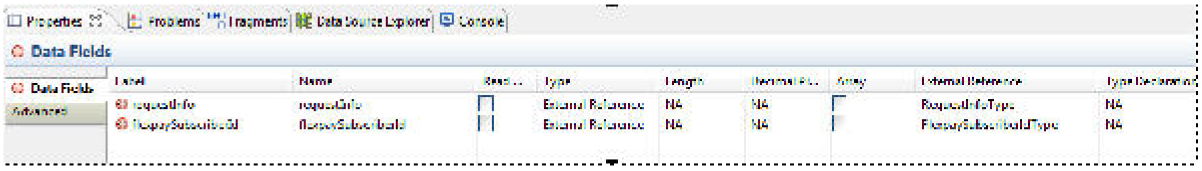
Service Type: Web Service
Operation:
Port Type:
Operation Name:
Port Name:
Service Name:
Transport:
Endpoint Resolution:
WSDL: Use local / Use remote
Location:
Endpoint Name:
Security Profile:

Looking in the Package Explorer, we can see the WSDL file under the Service Descriptors heading. You can open the WSDL by double-clicking the WSDL entry (or right-clicking on the WSDL entry and selecting OpenWith/WSDL Editor). Viewing the WSDL in the WSDL Editor, we can see what types the Web Service takes as parameters:



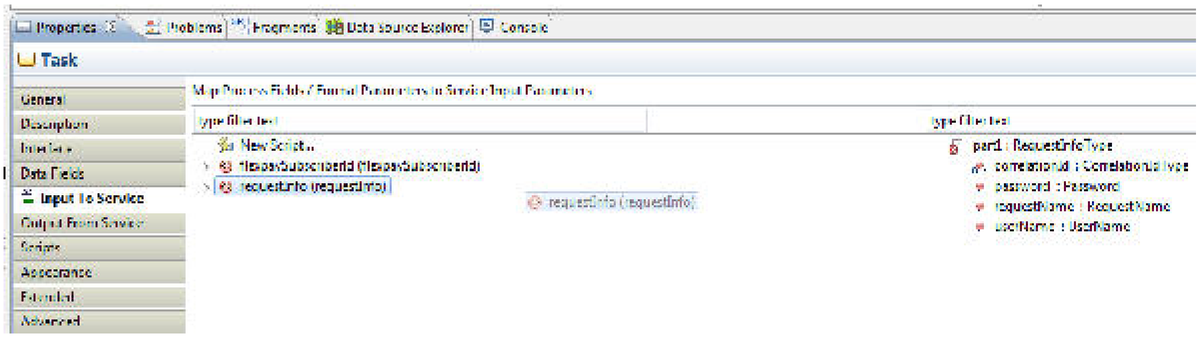
AlarmRPI		
BaseInfo		
input	part1	authenticationKeyType
output	part1	cityNameType
fault1	part2	errorInfo
CouponDetails		
input	part1	personalInfoType
output	part1	couponDetailsType
Subscribe		
input	part1	subscriberInfoType
output	part1	authenticationKeyType
Request		
input	part1	requestInfoType
output	part1	flexpaySubscriberIdType
Code		
input	part1	cityCodeType
output	part1	dealerCodeType

We can now create two Data Fields in the process of the appropriate types:

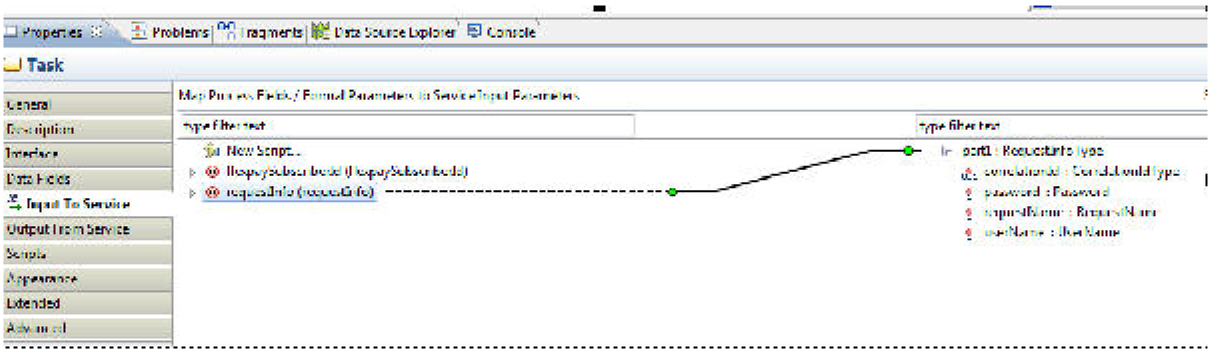


Label	Name	Read...	Type	Length	Decimal Pl...	Array	External Reference	Type Declaration
requestInfo	requestInfo	☐	External Reference	NA	NA	☐	RequestInfoType	NA
flexpaySubscriberId	flexpaySubscriberId	☐	External Reference	NA	NA	☐	FlexpaySubscriberIdType	NA

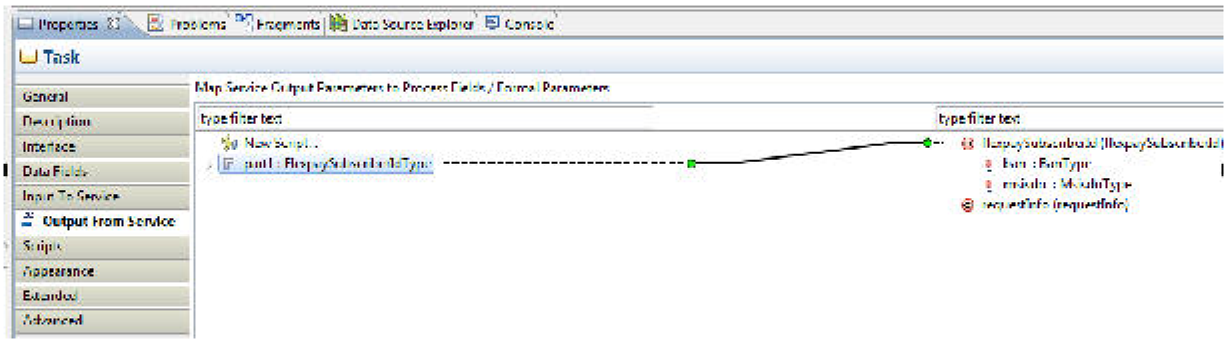
Then, from the **Input To Service** property sheet, map the **requestInfo** field to the **RequestInfoType** input of the Web Service by dragging the field name onto the parameter name, as shown in the diagram below:



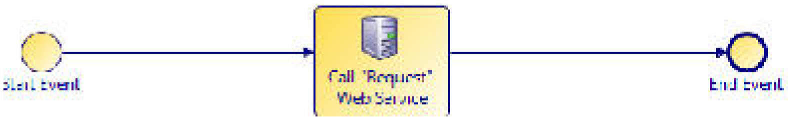
The mapping will then be shown like this:



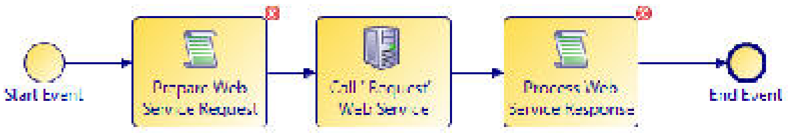
Then, repeat for the output of the Web Service:



Then, on the **General** property sheet, change the name of the task to something appropriate, for example, **Call "Request" Web Service**. Then, we have completed the Web Service task:



Now we need to prepare the data to go into the webservice and process the data that comes out of the web service. This can be done in scripts. To do this, drag two script tasks onto the Process Diagram before and after the Web Service call and name them appropriately:


Start Event
Prepare Web Service Request
Call "Request" Web Service
Process Web Service Response
End Event

The error markers in the figure above indicate that the scripts have not been written yet. In the **General** property sheet of the first script, set the **Script Defined As** property to **JavaScript**, and enter the script. First, we need to make the RequestInfo variable refer to an actual RequestInfo object by initializing it using the factory method (don't forget to use the content assist to help. See Using Content Assist for more information). The following script is one example:

```
// Prepare Web Serice Call Request
requestInfo =
com_amsbqa_linda01_xsd_define_types_types_Factory.createRequestInfoType();
requestInfo.correlationId = 123456;
requestInfo.password      = "Password!";
requestInfo.requestName   = "Search";
requestInfo.userName      = "Fred Blogs";
```

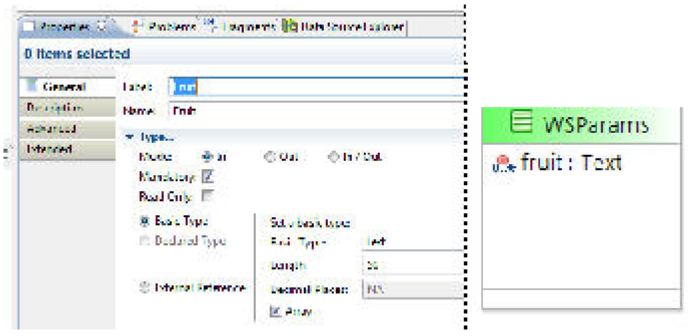
Something similar can be done in the script after the Web Service task. It is not necessary to make the **flexpaySubscriberId** field point to an object, because this will have already been done by the Web Service task. All that remains to do is process the values, for example:

```
// Process response from Web Service Call
if (null != flexpaySubscriberId)
{
    ban    = flexpaySubscriberId.ban;
    msisdn = flexpaySubscriberId.msisdn;
}
```

Passing Arrays to Web Services

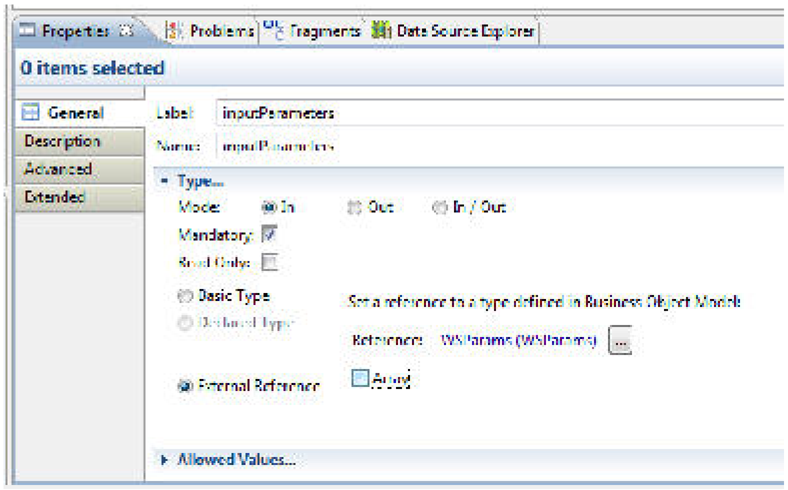
Web Services cannot be passed (in or out) of an array field. Therefore, if a process needs a field that contains multiple instances of a Business Object that it wants to pass in or out of a web service, it will have to wrap it in a BOM class. The BOM class must contain a multi-instance attribute. The multi-instance attribute must contain the multiple instances rather than an Array field.

For example, if you want to pass an array of Strings to a webservice, then you cannot just have a Text parameter flagged as an Array:



Instead, create a BOM class to wrap the multiple Text values like this (Note that multiplicity of `0..*` indicates that the attribute can contain zero or more values which is like the **Array** setting on a Process Field):

And then reference this type as a parameter:


0 items selected

Label: inputParameters

Name: inputParameters

Type...

Mode: [✓ In] Out In / Out

Mandatory: ✓

Read Only: ☐

Basic Type

Declared Type

[✓ External Reference]

Set a reference to a type defined in Business Object Model:

Reference: WSParams (WSParams)

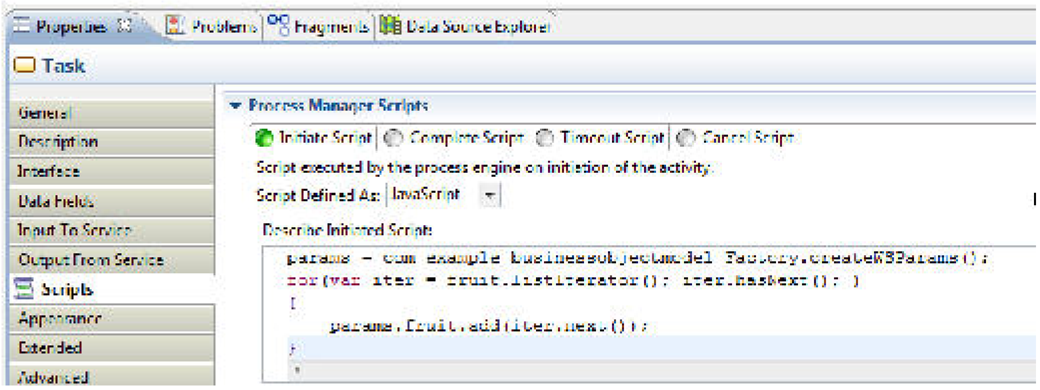
☐ Array

Allowed Values...

If you define a webservice with an Array parameter, the following error message appears:

BDS Process 1.0 : Activities responsible for generating WSDL operations cannot have array parameters associated, instead please create a business object class to contain the array. (ProcessPackageProcess:StartEvent)

When calling the webservice, copy the text array into the parameter structure with a loop like this in the ServiceTask's Init Script, as shown below:


Task

Process Manager Scripts

[✓ Initiate Script] Complete Script Timeout Script Cancel Script

Script executed by the process engine on initiation of the activity:

Script Defined As: JavaScript

Describe Initiated Script:

```
params = com.example.businessobjectmodel.Factory.createWSParams();
for(var iter = fruit.listIterator(); iter.hasNext(); )
{
    params.fruit.add(iter.next());
}
```

If an array of Business Objects was being passed using the `ScriptUtil.copyAll()` method, as described in Looping Through Lists and Copying Elements in Lists, could be used to copy the array of Business Objects in a single statement, but that cannot be used for Basic fields like the Text array in this example.

An Array object can be passed to other Task types, for example, User Tasks or ScriptTasks. Only the WebService task does not support Array fields.

Parse Functions

This section contains notes on parseInt() and parseFloat().

parseInt()

The `parseInt(string [, radix])` function parses a string and returns an integer.

The radix parameter specifies which numeral system is to be used, for example, a radix of 16 (hexadecimal) indicates that the number in the string should be parsed from a hexadecimal number to a decimal number.

If the radix parameter is omitted, JavaScript assumes the following:

- If the string begins with `0x`, the radix is 16 (hexadecimal).
- If the string begins with `0`, the radix is 8 (octal). This feature is deprecated.
- If the string begins with any other value, the radix is 10 (decimal).

parseFloat()

The `parseFloat()` function parses a string and returns a floating point decimal number.

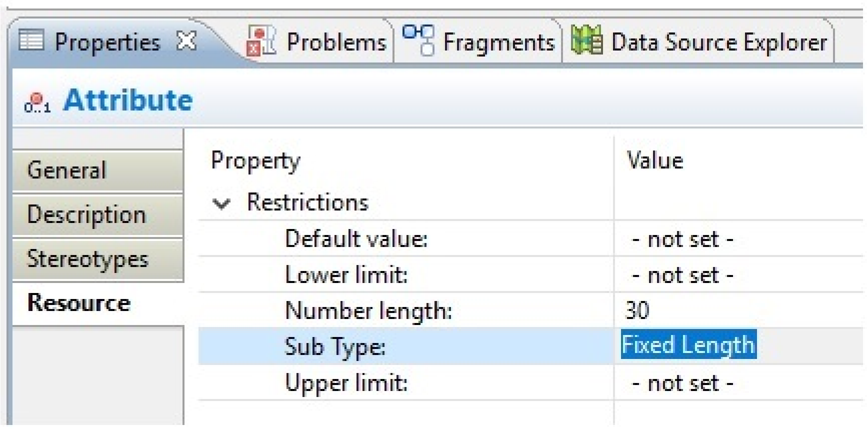
This function determines if the first character in the specified string is a number. If it is, it parses the string until it reaches the end of the number, and returns the number as a number, not as a string.

Advanced Scripting Examples

This section gives some examples of scripting using some classes and methods that require special attention.

Working with Fixed Length Integers (BigInteger)

If we are working with large integer numbers, and want to work out the weight of the average planet, then we would want to use the Fixed Length integers which are implemented using Java BigIntegers. The "+", "-", "*", "/", "==", ">", "<", and so on, operators cannot be used. Instead, the methods of the BigInteger class have to be used when doing arithmetic and comparisons. In the BOM editor, the attribute sub-type should be set to Fixed Length:



Property	Value
Restrictions	
Default value:	- not set -
Lower limit:	- not set -
Number length:	30
Sub Type:	Fixed Length
Upper limit:	- not set -

The script should be written as shown below:

```
var totalKgs    = ScriptUtil.createBigInteger(0);
var planetCount = ScriptUtil.createBigInteger(0);
var one         = ScriptUtil.createBigInteger(1);
for (var iterator = planets.planetList.listIterator(); iterator.hasNext(); )
{
    var planet = iterator.next();
    totalKgs = totalKgs.add(planet.weightKgs);
    planetCount = planetCount.add(one);
}
if (planetCount.compareTo(one) >= 0)
{
        planets.averageWeight = totalKgs.divide(planetCount);
}
else
{
    planets.averageWeight = ScriptUtil.createBigInteger("0");
}
```

Or, since the number of planets will not have a very large value, we can have the planet counter as a signed integer, and then convert it into BigInteger for the divide operation at the end of the script:

```
var totalKgs    = ScriptUtil.createBigInteger(0);
var planetCount = 0;
for (var iterator = planets.planetList.listIterator(); iterator.hasNext(); )
{
     var planet = iterator.next();
     totalKgs = totalKgs.add(planet.weightKgs);
     planetCount ++;
}
if (planetCount >= 1)
{
 planets.averageWeight = totalKgs.divide(ScriptUtil.createBigInteger(planetCount));
}
```

```
else
{
 planets.averageWeight = ScriptUtil.createBigInteger("0");
}
```

In order to compare Fixed Integers, you have to use the `compareTo()` or `equals()` methods. The `equals()` method returns true or false depending on whether the values are equal or not. Given two BigInteger variables `x` and `y`, the expression

```
    x.compareTo(y) <op> 0
```

returns the same results as the following when using the Signed Integer sub-type if `x` and `y` were signed integer values:

```
    x <op> y
```

(Where <op> is one of the 6 comparison operators: {==, !=, <, <=, >=, >}).

For example, assuming that variable one has a value of 1 and variable two has a value of 2:

```
var one = ScriptUtil.createBigIntger(1);
var two = ScriptUtil.createBigIntger(2);
```

Then:

Operator	Description	Example	Result
x.compareTo(y) == 0	Equals	one.compareTo(two) == 0	false
		one.compareTo(one) == 0	true
x.compareTo(y) != 0	Not Equals	one.compareTo(two) != 0	true
		one.compareTo(one) != 0	false
x.compareTo(y) < 0	Less than	one.compareTo(two) < 0	true
		one.compareTo(one) < 0	false
		two.compareTo(one) < 0	false
x.compareTo(y) <= 0	Less than or equals	one.compareTo(two) <= 0	true
		one.compareTo(one) <= 0	true
		two.compareTo(one) <= 0	false
x.compareTo(y) >= 0	Greater than or equals	one.compareTo(two) >= 0	false
		one.compareTo(one) >= 0	true
		two.compareTo(one) >= 0	true
x.compareTo(y) > 0	Greater than	one.compareTo(two) > 0	false
		one.compareTo(one) > 0	false
		two.compareTo(one) > 0	true

There are other methods available on the BigInteger objects, which you can read about in Fixed Length Integer (BigInteger) Methods. These are:

abs	add	compareTo	divide	equals	gcd	max
min	mod	multiply	negate	pow	remainder	subtract

One thing to note about BigIntegers is that they are Immutable, for example, they do not change once they are created. Therefore, all the above methods do not change the object that they are working on. However, if appropriate, they return a new BigInteger object that has the new value.

Another thing to bear in mind when creating BigInteger objects with the `ScriptUtil.createBigInteger()` method is that the number to create can be passed as a String or a Numeric type. It is important to be aware that the JavaScript numeric type is only accurate to about 16 significant figures, so when initializing large BigInteger values, the value should be passed as a String. Otherwise accuracy can decrease since it is converted to a double and then onto a BigInteger. For example:

```
var bigInt = ScriptUtil.createBigInteger(12345678901234567890)
```

The result is the creation of the number 9223372036854776, rather than 12345678901234567890 as might be expected.

For more details on the BigInteger type, see the Java documentation, located at the following web site:

http://download.oracle.com/javase/8/docs/api/java/math/BigInteger.html

Unsupported Conversions

Implicit conversions between different numeric types are *not* carried out for the BigDecimal and BigInteger methods.

See Implicit Conversions Between Numeric Types.

See Unsupported Conversions.

Working with Fixed Decimals (BigDecimal)

Fixed Point (BigDecimal) objects are immutable like the BigInteger objects, so those methods that generate new BigDecimal results all return the new value, rather than update the object that is being operated on.

Creating and Initializing Fixed Decimal Values

Instead of just initializing variables or attributes in scripts with numbers, as is done with Floating Point decimals, Fixed Decimal values are Java Objects that need to be created using the `ScriptUtil.createBigDecimal()` factory method.

For example:

```
var dTotalKgs     = ScriptUtil.createBigDecimal(0.0);
var dTotalKgs     = ScriptUtil.createBigDecimal("0.0");
```

The value passed to the factory method can either be a JavaScript number (or a BOM Floating Point attribute value), or a quoted string (or a BOM Text attribute value). In most cases, it is best to use a Text parameter, which is converted to the exact number, whereas some small errors can occur when using numeric parameters, especially if the values are large.

Simple Operations

Instead of using operators like "+" and "-" to perform basic arithmetic with Fixed Decimals, you must use methods to perform these operations.

For example, if we had two Business Objects called earth and moon, and each had a weight attribute of Fixed Decimal type, in order to add the two fixed decimals together, we would use the `add()` method of the Fixed Decimal attribute:

```
dTotalKgs = earth.weight.add(moon.weight);
```

There are similar methods called `subtract()`, `multiply()`, and `divide()` that can be used to perform the other arithmetic operators. For more details on these methods, see the reference section at the end of this document.

Rounding

If we needed more accuracy we would use a BigDecimal, also known as a Fixed Point Decimal, for the averageWeight attribute as represented in the code below (however, there is a problem with the code, which is discussed following it):

```
var dTotalKgs    = ScriptUtil.createBigDecimal(0.0);
var dPlanetCount = 0;
for (var iterator = planets.planetList.listIterator(); iterator.hasNext(); )
{
     var planet = iterator.next();
     dTotalKgs = dTotalKgs.add(planet.weightKgs);
     dPlanetCount ++;
}
planets.averageWeight =
dTotalKgs.divide(ScriptUtil.createBigDecimal(dPlanetCount));
```

This would work quite well now that we have 8 planets, but in the days before Pluto was downgraded to a Dwarf Planet we had nine planets, and attempting to divide a number by 9 often results in a recurring string of decimals if done exactly. This causes problems for BigDecimals, since the BigDecimal class stores numbers up to an arbitrary level of precision. However, it does NOT store numbers to an infinite level of precision, which would be required to store 1/9 = 0.111111…, for example. So when doing division operations that can result in recurring decimals or other overflows, Rounding Mode must be applied to the BigDecimal method that is being used.

There are two ways that Rounding Mode can be applied to the `divide()` method: either directly, or by way of a MathContext object that contains a precision and RoundingMode. If applied directly, it can be applied with or without the precision. There are eight possible values for RoundingMode: `UP`, `DOWN`, `CEILING`, `FLOOR`, `HALF_UP`, `HALF_DOWN`, `HALF_EVEN`, or `UNNECESSARY`. Additional details of the behavior regarding the different modes can be found in the reference section, but you can also see from the example here how different values are rounded according to the different RoundingModes.

Example Rounding Mode Results According to Single Digit Rounding Input

Input Number	Up	Down	Ceiling	Floor	Half _up	Half_down	Half_even	Unnecessary
5.5	6	5	6	5	6	5	6	5.5
2.5	3	2	3	2	3	2	2	2.5
1.6	2	1	2	1	2	2	2	1.6
1.1	2	1	2	1	1	1	1	1.1
1.0	1	1	1	1	1	1	1	1.0
-1.0	-1	-1	-1	-1	-1	-1	-1	-1.0
-1.1	-2	-1	-1	-2	-1	-1	-1	-1.1
-1.6	-2	-1	-1	-2	-2	-2	-2	-1.6

Input Number	Up	Down	Ceiling	Floor	Half _up	Half_down	Half_even	Unnecessary
-2.5	-3	-2	-2	-3	-3	-2	-2	-2.5
-5.5	-6	-5	-5	-6	-6	-5	-6	-5.5

There are three built-in MathContexts provided: DECIMAL32, DECIMAL64, and DECIMAL128. If the DECIMAL64 MathContext is used, then a precision (number of significant figures) and Rounding Mode (`HALF_UP`) equivalent to that used by the Floating Point Decimal sub-type (which is the same as the Double type used in languages such as Java and C, and the Numeric type used in JavaScript) is used.

The `UNNECESSARY` rounding mode is the default RoundingMode, and will generate an exception if the resultant number of digits used to represent the result is more than the defined precision.

The table below shows the results of dividing 12345 by 99999 using BigDecimals with different Precisions, RoundingModes, and MathContexts.

Example Rounding Mode Results According to BigDecimals Digit Rounding Input

Rounding Mode	Precision	Math Context	Result
			java.lang.ArithmeticException: Non-terminating decimal expansion; no exact representable decimal result.
UP			1
HALF_UP			0
UP	10		0.1234512346
HALF_UP	10		0.1234512345
UP	50		0.12345123451234512345123451234512345123451234512346
HALF_UP	50		0.12345123451234512345123451234512345123451234512345
		DECIMAL32	0.1234512
		DECIMAL64	0.1234512345123451
		DECIMAL128	0.1234512345123451234512345123451235
			java.lang.ArithmeticException: Non-terminating decimal expansion; no exact representable decimal result.

The `divide()` method called above should be changed to:

```
// 30 significant digits, rounding 0.5 up
planets.averageWeight = totalKgs.divide(ScriptUtil.createBigDecimal(planetCount),
30,RoundingMode.HALF_UP);
```

or

```
// 34 significant digits, rounding 0.5 up
planets.averageWeight = totalKgs.divide(ScriptUtil.createBigDecimal (planetCount),
MathContext.DECIMAL128);
```

or

```
// 30 significant digits, rounding 0.5 up (using MathContext)
var mc = ScriptUtil.createMathContext(30,RoundingMode.HALF_UP);
planets.averageWeight = totalKgs.divide(ScriptUtil.createBigDecimal
(planetCount),mc);
```

When comparing BigDecimal values it is best to use the `compareTo()` method, as the `equals()` method considers 1.4 to differ from 1.40. This is because 1.4 is stored in BigDecimal as the number 14 with a scale of 1 and a precision (for example, number of digits) of 2. 1.40 is stored as the number 140 with a scale of 2 and a precision of 3. The `equals()` method does not recognize them as the same. However, the `compareTo()` method sees that there is no difference between them, and returns 0, meaning they have the same value.

Unsupported Conversions

Implicit conversions between different numeric types are *not* carried out for the BigDecimal and BigInteger methods.

See Implicit Conversions Between Numeric Types.

For example, the following is *not* valid:

```
// Invalid example
var pi = ScriptUtil.createBigDecimal("3.14159");
var twoPi = pi.multiply(2);
// Implicit conversion of integer to BigDecimal is not done here
```

Instead, you must always pass BigDecimal parameters to the BigDecimal methods that require them as shown in this example:

```
var pi = ScriptUtil.createBigDecimal("3.14159");
var twoPi = pi.multiply(ScriptUtil.createBigDecimal(2));
```

Comparing Fixed Decimals and BigDecimals

The BigDecimal `compareTo()` method can be used in the same way as the BigInteger `compareTo()` method.

Detailed information on how this can be used is available in the section on Integers, but in summary, when comparing two decimal fields `x` and `y`, instead of using

```
x <relational_operator> y
```

you must use

```
x.compareTo(y) <relational_operator> 0
```

This will return the value that you expect the first expression to return. For example, if you want to use the expression

```
x <= y
```

you should write

```
x.compareTo(y) <= 0
```

As with all divide operations, care should be taken to ensure that the divisor is not zero, otherwise an exception will be generated. If there is a problem with the script so that there are no planets in the list, then the planetCount variable will be 0, and our divide operation will cause an exception. Therefore, scripts should be programmed defensively to protect against such things. The following example here is a version of the script that checks that the planetCount is greater than or equal to one using the BigDecimal `compareTo()` method:

```
var totalKgs    = ScriptUtil.createBigDecimal("0.0");
var planetCount = ScriptUtil.createBigDecimal ("0");
var one         = ScriptUtil.createBigDecimal ("1");
for (var iterator=planets.planetList.listIterator(); iterator.hasNext(); )
{
     var planet  = iterator.next();
     totalKgs    = totalKgs.add(planet.weightKgs);
     planetCount = planetCount.add(one);
}
if (planetCount.compareTo(one) >= 0)
{
     // 30 significant digits, rounding 0.5 up
     planets.averageWeight = totalKgs.divide(planetCount,30,RoundingMode.HALF_UP);
}
else
{
     planets.averageWeight = ScriptUtil.createBigDecimal("0.0");
}
```

When creating BigDecimal objects with the `ScriptUtil.createBigDecimal()` method, the number to create can be passed as a String or a Numeric type. It is important to be aware that the JavaScript numeric type is only accurate to about 16 significant figures, so when initializing BigDecimal types, if great accuracy is required, the value should be passed as a String. If the value is not passed in a String, the value entered in the script will first be converted to a Numeric type, which may introduce some rounding errors, even for values that you would not expect it to. For example, the value 0.1 may not be stored exactly in a Numeric type, as it results in a recurring sequence of binary digits when expressed in binary: 0.0001100110011001100110011...

When rounding BigDecimal variables, you need to be aware of how BigDecimal values are stored. They are stored as two integer values: unscaled value and scale. For example, if the number 123.456789 is stored as a BigDecimal value, it will have an unscaled value of 123456789 and a scale of 6. The value of a BigDecimal is:

```
(unscaled_value) * 10-scale
```

The `setScale()` method is used to round values. If the `setScale()` method is called with a scale of 10, then the scale would become 10 and the unscaled value would be changed to 1234567890000 so that the number still has the same numerical value. However, it would actually represent 123.4567890000. When reducing the number of decimal places, for example, to 3, rounding must take place often. You must tell `setScale()` how you want to round the value, otherwise an exception will be generated at runtime. To convert to 3 decimals using the `HALF_UP` rounding strategy, write:

```
roundDecimal = decimal.setScale(3, RoundingMode.HALF_UP);
```

This converts, in our example, 123.4567890000 to 123.457.

For more information on BigDecimal, see Fixed Point Decimal (BigDecimal) Methods, or the Java Documentation, available at the following web site:

http://download.oracle.com/javase/8/docs/api/java/math/BigDecimal.html

Object BOM Native Type

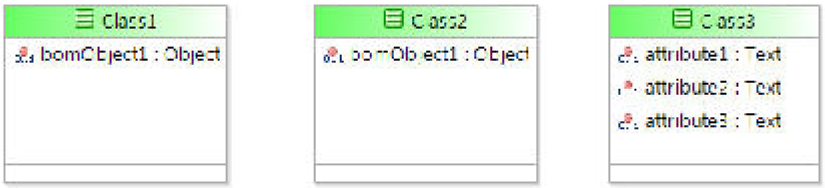
There are four different varieties of BOM objects. xsd:any, xsd:anyAttribute, xsd:anyType, xsd:anySimpleType.

All of these allow for different assignments. However, once data has been stored in an object, it cannot be read back out into its original type. It must remain in that object. It is also not possible to assign a BOM object of one variety into a BOM object of a different variety.

Using the Object BOM Native Type

xsd:any

The Object Type is used to handle sections of XML with an unknown format or where content is not known, but the data of which can still be passed on by the system. BOM Attributes can be defined as Object type (for example, `xsd:any`). An Object type BOM class attribute can be assigned either another Object attribute, or a BOM class. For example, given the following BOM:



If a Web Service process has the following fields and parameters

Data Field / Parameter	Type	Name
Input Parameter	Class1	inputField1
Data Field	Class2	bomField2
Data Field	Class3	bomField3
Output Parameter	Class2	outputField1

Then a script in the process can be written as

```
bomField2.bomObject1 = inputField1.bomObject1;
```

`bomField2` can be used as the input parameter to another Web Service, which would pass the `xsd:any` value from the input parameter of one service to the input parameter of another service.

You can also write

```
outputField1.bomObject1 = bomField3;
```

which would pass the Business Object `bomField3` in an `xsd:any` type construct in the response XML message for the web service.

Object BOM Native Type attributes can have a multiplicity greater than one, in which case the `add()` method will be used as usual for assigning values to the field, for example:

```
outputField1.bomObject1.add(bomField3);
```

xsd:anyAttribute

The xsd:anyAttribute is a very restrictive form of Object BOM Native Type. This type can only be assigned to itself. No other BOM type can be assigned either to or from it.

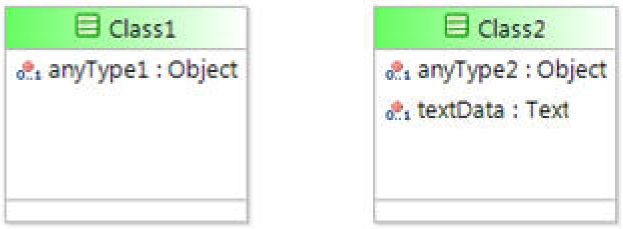
xsd:anySimpleType

The xsd:anySimpleType is very similar to xsd:any, it behaves in the same manner but instead of taking a BOM Class as its input it takes a primitive type.

xsd:anyType

The xsd:anyType is again similar to both the xsd:any and xsd:anySimpleType. The difference is that it can be assigned either a BOM Class or primitive type. This makes it the most flexible of storage types. One important difference is that if you wish to set the value of an xsd:anyType to the same value as either another xsd:anyType or a Business Object (that is, a BOM Class instance), then you must use ScriptUtil.copy() in order to take a copy of the source object before assigning it to the xsd:anyType.

```
// Copy an entire BOM Class Instance
```



Class1.anyType1 = ScriptUtil.copy(Class2); // Copy an anyType from one Class to another Class1.anyType1 = ScriptUtil.copy(Class2.anyType2); // Copy a text field into the anyType Class1.anyType1 = Class2.textData

Restrictions

There can only be one Object BOM Native Type attribute in any class hierarchy (this is the only case for `xsd:any` and `xsd:anyAttribute`), due to ambiguities with knowing how to parse incoming XML if there are more than one.

You cannot examine the contents of any of the Object BOM Native Types.

BPM Forms do not support the Object BOM Native Type, so do not use a Business Object that includes an Object BOM Native Type attribute as a parameter to the UserTask.

Object BOM Native Type and ScriptUtil.setObject()

The default assignment of a Business Object (for example, a BOM Class instance) to an Object BOM Native Type attribute (imported `xsd:any`) looks like this [from the previous example]:

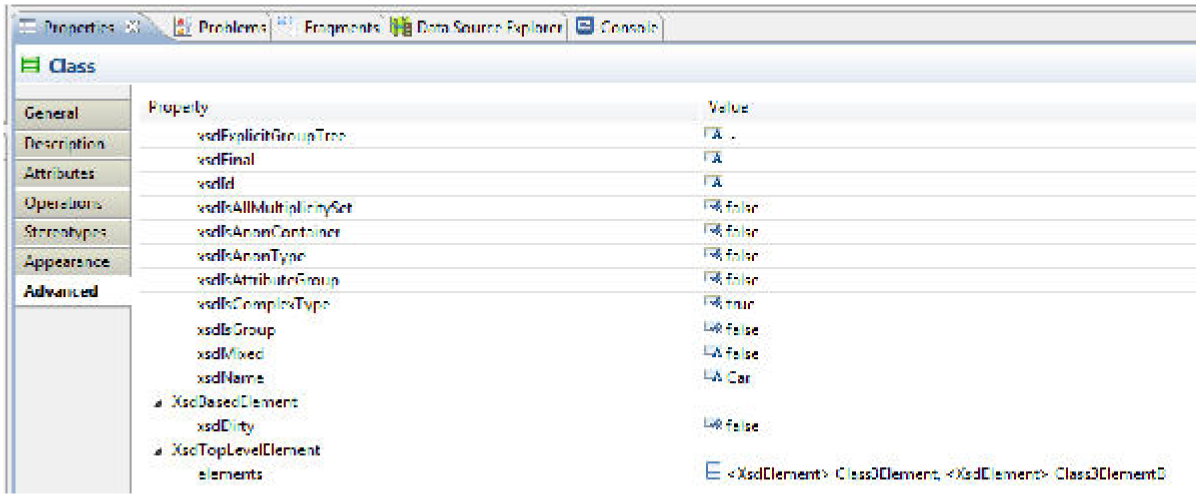
```
outputField1.bomObject1 = bomField3;
```

If the BOM was created in TIBCO Business Studio, then there is only one element for each type, so the above example will always produce the desired result. However, if the BOM was created by importing an XSD Schema and the simple assignment interface is used, then BDS will automatically select the best available element in which to store the complex object.

For situations where you wish to specify which element the complex data is stored as, a utility method can be used, as in the following example:

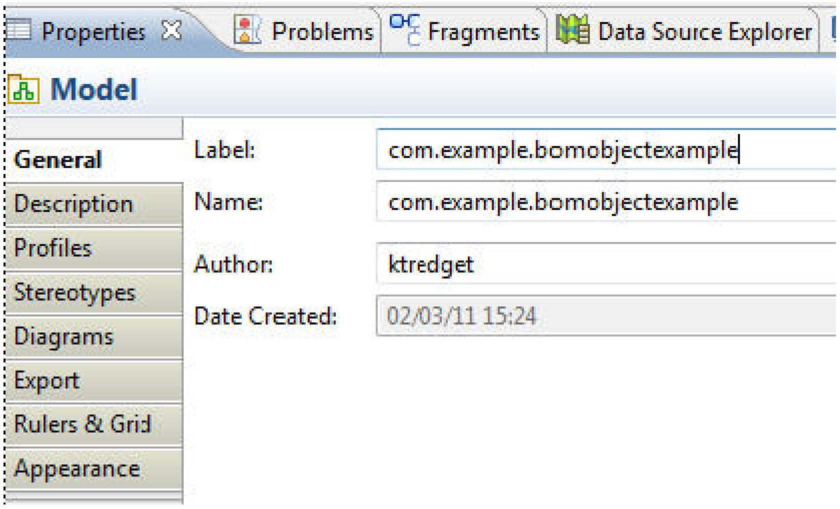
```
ScriptUtil.setObject(outputField1.bomObject1, bomField3,
                "com.example.bomobjectexample.Class3Element");
```

For the above example, to find the parameters that can be used in this case you need to find the name of the element associated with the class (type). Select the class in the BOM editor, and look at the advanced properties sheet:



Property	Value
xsdExplicitGroupTree	.
xsdFinal	
xsdId	
xsdIsAllMultiplicitySet	false
xsdIsAnonContainer	false
xsdIsAnonType	false
xsdIsAttributeGroup	false
xsdIsComplexType	true
xsdIsGroup	false
xsdMixed	false
xsdName	Car
XsdBasedElement	
xsdDirty	false
XsdTopLevelElement	
elements	<XsdElement> Class3Element, <XsdElement> Class3ElementB

At the bottom of the property sheet, the XsdTopLevelElement property is listed. In the screenshot above, there are two elements: Class3Element and Class3ElementB. This ScriptUtil function allows the script writer to define which element the `xsd:any` should be associated with.



Label: com.example.bomobjectexample
Name: com.example.bomobjectexample
Author: ktredget
Date Created: 02/03/11 15:24

The element name is combined with the BOM namespace to make up the third parameter to the `ScriptUtil.setObject()` method. For this BOM example, the namespace can be found in the Name field of the BOM properties sheet:

Concatenating the two parts results in the following `ScriptUtil.setObject()` line:

```
ScriptUtil.setObject(outputField1.bomObject1, bomField3,
                  "com.example.bomobjectexample.Class3Element");
```

Additional JavaScript Global Functions

This topic describes some additional JavaScript global functions you can use.

escape() and unescape()

The JavaScript `escape()` and `unescape()` functions can be used to encode strings that contain characters with special meanings or outside the ASCII character set.

The `escape()` function encodes a string. This function makes a string portable, so it can be transmitted across any network to any computer that supports ASCII characters.

The function does not encode A-Z, a-z, and 0-9. Additionally, the function encodes all other characters with the exception of: "*", "@", "-", "_", "+", ".", "/".

The `escape()` function maps:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

To:

```
%20%21%22%23%24%25%26%27%28%29*+%2C-./0123456789%3A%3B%3C%3D%3E
%3F@ABCDEFGHIJKLMNOPQRSTUVWXYZ%5B%5C%5D%5E_%60abcdefghijklmnopqrstuvwxyz%7B%7C%7D
%7E
```

encodeURI() and decodeURI()

The `encodeURI()` function is similar to the `escape()` function but encodes fewer character. Encoding the string:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Will produce:

```
%20!%22#$%25&'()*+,-./0123456789:;%3c=%3e?@ABCDEFGHIJKLMNOPQRSTUVWXYZ%5b%5c%5d%5e_
%60abcdefghijklmnopqrstuvwxyz%7b%7c%7d~
```

So only the following characters are encoded:

SPACE, "“", "&", "<", ">", "[", "\", "]", "^", "`", "{", "|", "}"

encodeURIComponent() and decodeURIComponent()

These functions are similar to the `encodeURI()` and `decodeURI()` methods, but they additionally encode and decode the following characters:

, / ? : @ & = + $ #

Business Data Modeling Best Practice

This section gives some brief guidance on best practice.

Store Local BOMs in Business Data Projects

Keep local BOMs and WSDLs or XSDs (and their generated BOMs) in suitable Business Data projects, rather than in Analysis or BPM Developer projects.

Keeping local BOMs separate from the process projects that reference them has the following advantages:

- It makes it easier to organize and share local data among different processes. (Using a Business Data project, the local data only needs to be defined and deployed once. If you use an Analysis or BPM Development project - that is, the same project as a business process that uses the data - whenever that project is deployed or generated as a DAA, BDS Plug-ins corresponding to the referenced BOMs are packaged as part of the DAA. That is, every deployed process has its own copy of any local data it uses.)
- It provides better design-time performance, particularly for projects involving large numbers of local/generated BOMs (by avoiding unnecessary regeneration of BDS Plug-ins).

Keep Local BOMs and Global BOMs in Separate Business Data Projects

Keep local BOMs and global BOMs in separate Business Data projects, unless you have a compelling reason to keep them together.

This is particularly important for application upgrade, as local and global BOMs have different compatibility requirements:

- Local BOMs: You can still upgrade a Business Data project if a local BOM contains incompatible changes, but doing so could result in failure to migrate a dependent process instance to the upgraded version. See Process Migration.
- Global BOMs: You cannot upgrade a Business Data project if a global BOM contains incompatible (that is, destructive) changes. See "Upgrading a Case Data Model" in the *Case Data User's Guide*.

Upgrade Business Data Projects and Dependent Process Projects Together

Keep BDS applications and dependent process applications in step to facilitate subsequent deployments or undeployments of either application.

If a process project references a class in a local or global BOM in a Business Data project, the version number of the Business Data project is used in the reference when the DAA for the process project is generated. This creates an exact-match application dependency from the process application to that version of the BDS application.

Consequently, when you upgrade a local BOM or a global BOM, you should also upgrade any existing process project that references that Business Data project - even if that process project makes no use of the updated parts of the BOM.

Use Pre-Compilation for Projects That Contain Large or Generated BOMs

Configure a project that contains generated or other large BOMs to use pre-compilation, so that the BDS plugins derived from the BOMs are not generated each time that the project is built.

Generated BOMs are often large and, being derived from WSDLs or XSDs, usually will not change very often, so using pre-compilation can significantly improve design-time performance.

To configure the project to use pre-compilation, right-click the project in Project Explorer and choose **Pre-compile Project** > **Enable**. See "Pre-Compiling Projects" in the *TIBCO Business Studio Modeling Guide* for more information about pre-compilation.

Choose Appropriate Data Types

Take care when selecting attribute types.

- Be aware of the value space of the default Signed Integer sub-type [-231,231-1]. If it is insufficient, use the Fixed Integer type.
- Be aware of the limitations of the default Floating Point Decimal sub-type - 16 significant digits. Limitation of accuracy and rounding issues may indicate that it is not suitable for handling large values.
- Remember that when you are converting a data item from one type to another, either explicitly or implicitly, it is possible to exceed the limitations of the target type and generate an error. See Business Data Scripting by Example for examples of conversion limitations.
- With Datetime types if a timezone is required, use Datetimetz.

Use Sub-Packages to Aggregate Related Concepts

It is a good practice to put all related concepts in a sub-package.

For example, in a claim model BOM there can be classes, primitive types, and enumerations relating to customers, policies, and claims. So three sub-packages can be created to collect the different types together in different groups. If the root package name is `com.example.claimmodel`, then there can be sub-packages called:

```
com.example.claimmodel.customer
com.example.claimmodel.policy
com.example.claimmodel.claim
```

Organizing classes, enumerations, and primitive types in sub-packages makes them easier to find when viewing the BOM, and also when scripting it means that the factory has fewer methods in it, which makes it easier to find the method that you need.

As an extension to this, you can actually have sub-packages in different BOM files. When doing this, it is important that each package or sub-package is only in one BOM file. Using the above example, you can have three BOM files for the three sub-packages, or you can have four if some things are defined in the root package. Alternatively, it can just be split into two BOM files with the root package and two sub-packages in one BOM file, and the third sub-package in the second file.

Process Data Field Granularity

If a number of related parameters are commonly passed to Tasks, it is a good idea to create a BOM class that contains all the parameters. Then, all the values can be passed to the tasks as a single parameter.

However, one thing to be aware of is that if you have parallel paths within your process that are both processing a Business Object, then the changes made by the first branch to complete may be overwritten by data from the second branch if it is all stored in a single Business Object. To get around this problem, each branch should only be given the data it needs. Then, the data should be merged back into the single Business Object after the branches have merged together.

BOM Class Attribute and Variable Names

It is recommended that BOM class names begin with an uppercase letter and that variable and attribute names begin with a lowercase letter so that they can easily be distinguished. This is the convention used by Java, and what is done by the label to name mapping if Labels contain words with initial capital letters.

In order to read variable names that are made up of several words, it is recommended that you use "camelcase", where the initial letter of every word (apart from the initial word in a data field or attribute name) is capitalized. A data field holding a customer account should be written `customerAccount`. Similarly you can have data fields called `headOfficeName`. This naming convention is used in the factory methods, so if there is a CustomerAccount class, then there will be a `createCustomerAccount()` method in the factory. If you enter Labels with initial capitals for each word then this will be achieved. Therefore, a label written as "Customer Account" will be converted to a class name of `CustomerAccount`, or an attribute name of `customerAccount`. If the label is written as "Customer account", the class name and attribute name will be `Customeraccount` and `customeraccount`, both of which are not so readable.

If you use certain reserved keywords for BOM attribute names and assign them a value using javascript, at runtime you get a scripting error while evaluating the javascript expression. Avoid using the following names for BOM attributes if you intend using such attributes in scripting:

- `- notify`
- `- equals`
- `- wait`
- `- finalize`
- `- hashCode`
- `- toString`

Do Not Split a Namespace Across Projects

Classes from the same package must not be defined in separate BOM files.

For example, the following classes must both be defined in the package that defines the `com.example.claimmodel.customerdetails` package:

```
com.example.claimmodel.customerdetails.Address
com.example.claimmodel.customerdetails.Customer
```

Do Not Modify Generated BOMs

Do not modify the contents of BOMs in the **Generated Business Objects** folder of a project.

The BOMs created in the Generated Business Objects folder as a result of importing XSDs or WSDLs into the Service Descriptors folder should not be edited because if the file is regenerated, then any changes made by editing the BOM could be lost.

If the intention is to import an XSD and generate a BOM, then the XSD (or WSDL) should be imported into the Business Objects folder.

Business Data Scripting Best Practice

This section contains some suggestions for Business Data scripting.

Keep the Scripts Small

It is recommended that you keep scripts small. You can do this by breaking potentially large chunks of logic into separate scripts.

Ensure Business Objects Are Valid Before Scripts Complete

Remember that the length, limits, and multiplicity are checked at the end of every script, so ensure that all the Out and In / Out fields for the task have valid values before the script task is completed.

Check for Nulls

It is very important to check that fields and attributes are not null before attempting to get their values.

Use Comments in Scripts

It is good practice to comment code to make it easier for others who follow you to understand what the scripts are doing.

Use Constant First when Comparing a Constant with a Variable

Using:

```
constant == variable
```

is safer than:

```
variable == constant
```

If "=" is used instead of "==" by mistake, the former construction will result in a syntax error.

Troubleshooting

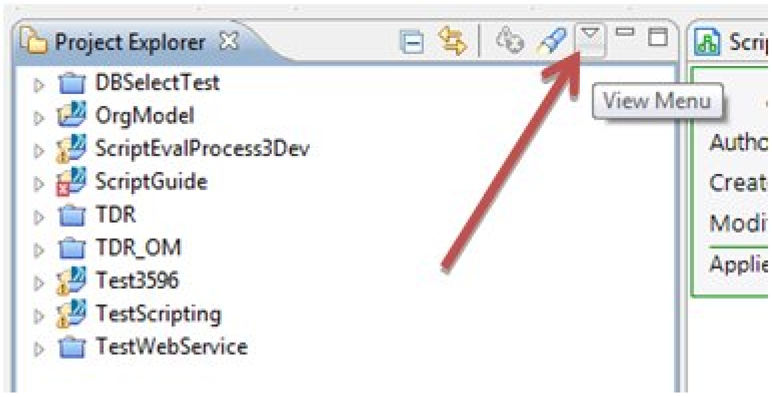
This section describes how to identify and resolve some problems you may encounter when using Business Data Services.

Viewing BDS-generated BDS Plug-in Hidden Projects

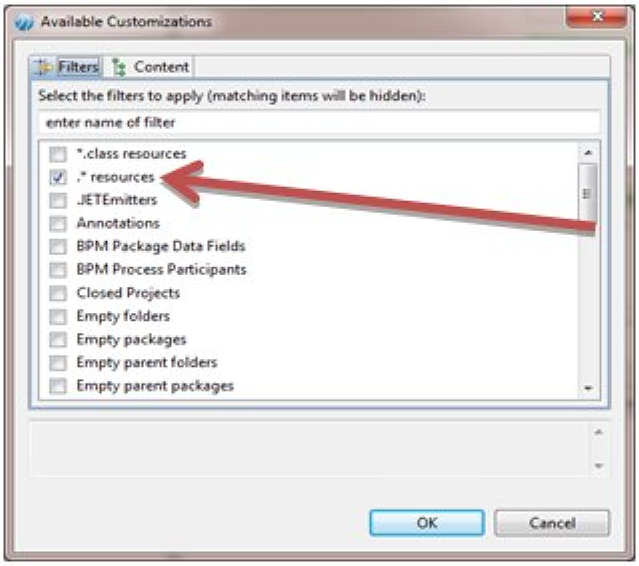
When a DAA (Distributed Application Archive) is created for a project containing a Business Object Model, or the project is deployed, the BDS Generator generates a BDS Plug-in that corresponds to the BOM.

These BDS Plug-ins can be seen in the Project Explorer, but are hidden by default.

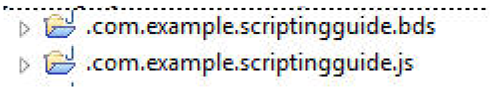
To view the BDS Plug-ins that have been generated, change the view so that it does not hide folder and file names that begin with ".". This can be done by clicking the **View** Menu in the Project Explorer:



Select **Customize View ...**. In the Available Customizations dialog, uncheck **.*resources**.



For each BOM, a pair of projects are created:


.com.example.scriptingguide.bds
.com.example.scriptingguide.js

Looking at these can be useful to understand how things are working.

Troubleshooting BDS Scripting

Occasionally, a script does not function as planned.

This section provides instruction on how to identify and resolve BDS scripting problems.

Reasons to Avoid Deleting Case Objects

The best practice is to *not* delete case objects as part of a normal operation. If you *do* delete case objects, the best practice is to only delete using a single case reference from within a service task in a business process.

The primary reason to not delete case objects is that if there are other processes (other than the process performing the deletion) that have a reference to the deleted case object, those processes become halted and cannot proceed. In addition, case history (audit trail) is constructed using case objects; if those objects are deleted, the history is no longer available.

Therefore, case objects should not be deleted until it is known for certain that no other processes are referencing the object, and that the case history is no longer needed.

If you delete a case object using a single case reference from within a service task in a business process, built-in checking is provided for other processes that are referencing the case object. (You can catch the `UnsafeToDeleteCaseError` error code.)

It is also possible to use the following methods to delete case objects, but these methods do not provide any error checking for other processes that are referencing the case object(s) that will be deleted. Using any of these methods could result in processes being halted because they are referencing a case object that no longer exists.

- Using the "deleteCase" operations in the BusinessDataServices API.
- Using a service task in a business process to delete:
  - multiple case objects using an array of case references.
  - a single case object using a case identifier.
  - a single case object using a composite case identifier.
- Using a service task in any type of process other than a business process - for example, a pageflow process or service process - to delete case objects (by any method).

Deletion of case objects is controlled by a system action (Delete Global Data) that defaults to `deny`, therefore you must be explicitly granted the permission to delete case objects.

Reserved keywords to avoid using for attribute names

If you use certain reserved keywords for BOM attribute names and assign them a value using javascript, at runtime you get a scripting error while evaluating the javascript expression.

Avoid using the following names for BOM attributes if you intend using such attributes in scripting:

- `notify`
- `equals`
- `wait`
- `finalize`
- `hashCode`
- `toString`

BDS Classes Do Not Appear or Changes Do Not Appear

Check that there are no problems with the BDS Plug-in generation, as described previously.

Break Script into Smaller Scripts

Add User Tasks between script tasks to see the field values.

Examine the Server Logs

TIBCO BPM components write out logging information as they process work. It can be useful to look at these logs when debugging scripts that are not working. In order to do this, ensure that the debugging level is turned up to maximum (see the Administrator interface documentation for your BPM runtime for more information about editing logging levels).

Log files are located in:

`CONFIG_HOME/tibcohost/INSTANCE_NAME/nodes/BPMNode/logs/BPM.log`

On Windows, the default location for *`CONFIG_HOME`* is:

`C:\ProgramData\amx-bpm\tibco\data`

On UNIX, the default location for *`CONFIG_HOME`* is:

`/opt/amxbpm/tibco/data`

The default *`INSTANCE_NAME`* is Admin-AMX BPM-AMX BPM Server.

Checking the log file can help locate the cause of scripting problems. For example:

```
23 Mar 2011 17:53:50,417 [Default Service Virtualization Thread_72] [ERROR]
com.tibco.n2.brm.services.impl.AsyncWorkItemSchedulerServiceImpl - [ERROR] -
{BRM_WORKITEM_ASYNC_SCHEDULE_WORK_ITEM_WITH_MODEL_MESSAGE_FAILED} - Async schedule
work item with model message failed Â¬{extendedMessage=`Param [integer] Value
[111222333444] exceeds the defined maximum limit of 9`,
componentClassName=`com.tibco.n2.brm.services.impl.AsyncWorkItemSchedulerServiceImpl
`, requestReceived=`Wed Mar 23 17:53:50 GMT 2011`, hostAddress=`10.100.83.80`,
nodeName=`BPMNode`, eventType=`FAULT`, messageCategory=`WORK_ITEM`,
componentId=`BRM`, stackTrace=`com.tibco.n2.brm.services.WorkItemFault: Param
[integer] Value [111222333444] exceeds the defined maximum limit of 9
    at
com.tibco.n2.brm.model.util.DataModelFactory.checkDataTypeValue(DataModelFactory.jav
a:2280)
    at
com.tibco.n2.brm.model.util.DataModelFactory.checkDataTypeValues(DataModelFactory.ja
va:2185)
    at
com.tibco.n2.brm.model.util.DataModelFactory.checkItemBodyDataTypesFromPayload(DataM
odelFactory.java:1707)
    at
com.tibco.n2.brm.services.impl.WorkItemSchedulerBase.privScheduleWorkItem(WorkItemSc
hedulerBase.java:621)
    at
com.tibco.n2.brm.services.impl.WorkItemSchedulerBase.scheduleWorkItemWithWorkModel(W
orkItemSchedulerBase.java:1323)
    at
com.tibco.n2.brm.services.impl.AsyncWorkItemSchedulerServiceImpl.scheduleWorkItemWit
hModel(AsyncWorkItemSchedulerServiceImpl.java:263)
    at sun.reflect.GeneratedMethodAccessor635.invoke(Unknown Source)
…
    at java.lang.Thread.run(Thread.java:619)
`, serviceName=`AsyncWorkItemSchedulerService`, principalId=`tibco-admin`,
priority=`HIGH`, managedObjectId=`78`, hostName=`uk-keitht`,
creationTime=`2011-03-23T17:53:50.417+0000`,
methodName=`scheduleWorkItemWithModel`, methodId=`asyncScheduleWorkItemWithModel`,
principalName=`tibco-admin`, correlationId=`6fb66791-595c-499b-ad55-5b56b7404fac`,
threadId=`1056`, compositeApplicationName=`amx.bpm.app`, severity=`ERROR`,
message=`Async schedule work item with model message failed`,
contextId=`6fb66791-595c-499b-ad55-5b56b7404fac`, threadName=`Default Service
Virtualization Thread_72`, environmentName=`BPMEnvironment`, lineNumber=`290`,
messageId=`BRM_WORKITEM_ASYNC_SCHEDULE_WORK_ITEM_WITH_MODEL_MESSAGE_FAILED`, Â¬}
```

Write Variable Values and Progress Updates from the Script to the BPM Log File

A facility for writing messages from scripts to the server called the BPM Log file is provided.

For example, the following can be done from a script:

```
Log.write("New Customer Process, Customer: '"+cust.name + "' added");
```

This generates a message like the following in the BPM Log file, which can be found by searching for the text `stdout -` or part of the message, for example, `New Customer Process`.

```
11 Feb 2011 09:42:06,529 [PVM:Persistent STWorkProcessor:5] [INFO ] stdout - New
Customer Process, Customer: Fred Blogs added
```

See Examine the Server Logs for the location of the log.

However, this function is only useful if you have access to the BPM Log file which is stored on the server.

eval()

The `eval()` function provides the ability to execute a dynamic script useful for debugging purposes. This is given a string, and executes it as if it was part of a script.

For example, to test some expressions, you could enter them into a Text field on a Form. Then, get a script to execute the commands in the Text field.

```
eval (scriptField);
```

Although useful for experimenting with scripts and testing them, TIBCO recommends that you do not use this function in a production environment because it allows the execution of any script text that is provided at runtime.

Use the Process Debugger

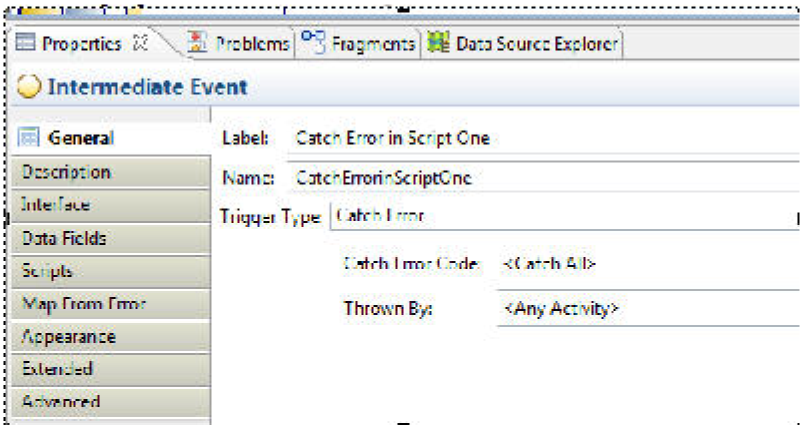
A debugger, which allows you to step through a process and to examine process flow and data manipulation, is provided with TIBCO Business Studio.

For more information, see the tutorial "Debugging a Business Process -> How to Debug a Business Process", in the TIBCO ActiveMatrix BPM Tutorials.

Catch Exceptions

If there is the possibility that a script will generate an exception at runtime, you can catch the exception and take corrective action in the process using a catch event.

For example, the following are some ScriptTasks with IntermediateCatch events attached to them, and the properties of one of them:



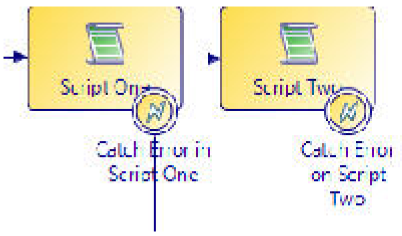
Label: Catch Error in Script One

Name: CatchErrorinScriptOne

Trigger Type: Catch Error

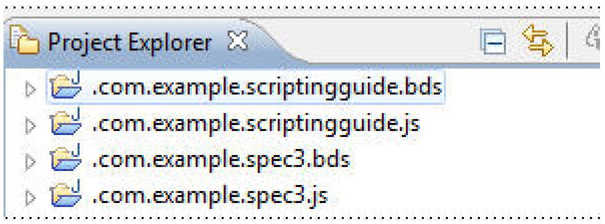
Catch Error Code: <Catch All>

Thrown By: <Any Activity>



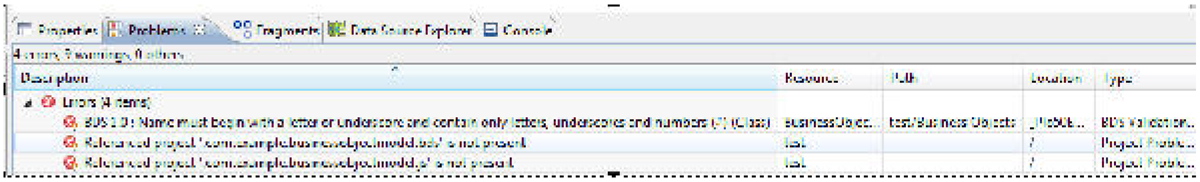
BDS Plug-in Generation Problems

Check to see if the BDS plug-ins are being created by looking for the hidden folders as described in the previous section. To verify that they are being generated, the BDS plug-in folders can be removed. Then, if a DAA is generated or the project is deployed, the BDS Plug-in folders should be regenerated:



.com.example.scriptingguide.bds
.com.example.scriptingguide.js
.com.example.spec3.bds
.com.example.spec3.js

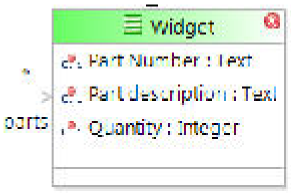
In the above screenshot, the two Scripting Guide folders can be deleted. Then the project that the Scripting Guide BOM comes from can be cleaned and rebuilt to regenerate the projects. If the projects are not regenerated, then click the **Problems** tab to check for reasons that the BOM generation may not be working:



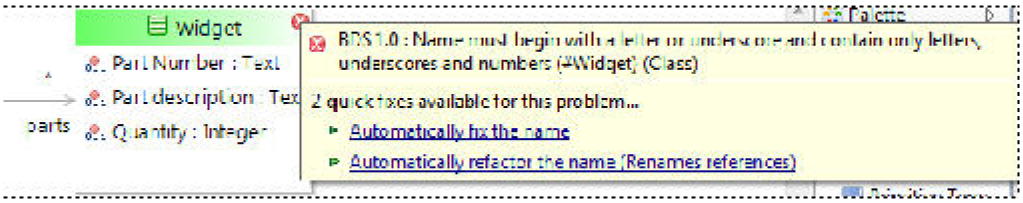
4 errors, 9 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (4 items)				
BDS 1.0 : Name must begin with a letter or underscore and contain only letters, underscores and numbers (-) (Class)	BusinessObjec...	test/Business Objects	_Pl6OE...	BDS Validation...
Referenced project 'com.example.businessobjectmodel.bds' is not present	test		/	Project Proble...
Referenced project 'com.example.businessobjectmodel.js' is not present	test		/	Project Proble...

The BOM editor warns you about problems by showing a red cross in the upper-right corner of the problem element:



Pointing to the red cross causes a dialog to display containing information about the error, for example:



BDS 1.0 : Name must begin with a letter or underscore and contain only letters, underscores and numbers (#Widget) (Class)

2 quick fixes available for this problem...

- Automatically fix the name
- Automatically refactor the name (Renames references)

Supplemental Information

This topic includes reference material to support your use of Business Data Services.

Data Type Mappings

This section contains tables showing data type mappings.

BOM Native Type to BDS Type Mapping

This table shows which Java Type the BOM Native Types are mapped onto, and whether they are mutable or not.

BOM Native Type to BDS Type Mapping

BOM Native Type	BDS Java Type	Mutable?
Attachment	N/A	N/A
Boolean	java.lang.Boolean	No
Date	javax.xml.datatype.XMLGregorianCalendar	Yes
Datetime	javax.xml.datatype.XMLGregorianCalendar	Yes
Datetimetz	javax.xml.datatype.XMLGregorianCalendar	Yes
Decimal – Fixed Point	java.Math.BigDecimal	No
Decimal – Floating Point	java.lang.Double	No
Duration	javax.xml.datatype.Duration	No
ID	java.lang.String	No
Integer – Fixed Length	java.Math.BigInteger	No
Integer – Signed	java.lang.Integer	No
Object - xsd.any	org.eclipse.emf.ecore.util.FeatureMap	No
Object - xsd.anyAttribute	org.eclipse.emf.ecore.util.FeatureMap	No
Object - xsd.anytype	EObject	No
Object - xsd.anySimpleType	java.lang.Object	No
Text	java.lang.String	No
Time	javax.xml.datatype.XMLGregorianCalendar	Yes
URI	java.lang.String	No

The values of mutable types can be changed, but the values of immutable types cannot. The methods that operate on them return new objects with the new values instead of mutating the value of the original object.

XSD Type to BDS Type Mapping

The following table shows what BDS types the different XSD types are mapped to

XSD Type	BOM Native Type	BDS Type
xsd:base64Binary	Text	String
xsd:byte xsd:byte (nillable)	Integer (signed)	Integer
xsd:decimal	Decimal (fixed – BigDecimal)	BigDecimal
xsd:float xsd:float (nillable)	Decimal (floating point – Double)	Double
xsd:gDay	Text	String
xsd:gMonth	Text	String
xsd:gMonthDay	Text	String
xsd:gYear	Text	String
xsd:gYearMonth	Text	String
xsd:hexBinary	Text	String
xsd:IDREF	Text	String
xsd:IDREFS	Text	String
xsd:integer	Integer (fixed – BigInteger)	BigInteger
xsd:language	Text	String
xsd:long xsd:long (nillable)	Integer (fixed – BigInteger)	BigInteger BigInteger
xsd:Name	Text	String
xsd:NCName	Text	String
xsd:negativeInteger	Integer (fixed – BigInteger)	BigInteger
xsd:NMTOKEN	Text	String
xsd:NMTOKENS	Text	String
xsd:nonNegativeInteger	Integer (fixed – BigInteger)	BigInteger

XSD Type	BOM Native Type	BDS Type
xsd:nonPositiveInteger	Integer (fixed – BigInteger)	BigInteger
xsd:normalizedString	Text	String
xsd:positiveInteger	Integer (fixed – BigInteger)	BigInteger
xsd:QName	Text	String
xsd:short xsd:short (nillable)	Integer (signed)	Integer
xsd:unsignedByte xsd:unsignedByte (nillable)	Integer (signed)	Integer
xsd:unsignedInt xsd:unsignedInt (nillable)	Integer (fixed – BigInteger)	BigInteger
xsd:unsignedLong	Integer (fixed – BigInteger)	BigInteger
xsd:unsignedShort xsd:unsignedShort (nillable)	Integer (signed)	Integer
xsd:string	Text	String
xsd:int xsd:int (nillable)	Integer (signed)	Integer
xsd:double xsd:double (nillable)	Decimal (floating point – double)	Double
xsd:ID	Text	String
xsd:date	Date	XMLGregorianCalen dar
xsd:datetime	Datetime	XMLGregorianCalen dar
xsd:duration	Duration	Duration
xsd:time	Time	XMLGregorianCalen dar
xsd:anyURI	URI	String
xsd:boolean xsd:boolean (nillable)	Boolean	Boolean
xsd:ENTITY	Text	String
xsd:ENTITIES	Text	String

XSD Type	BOM Native Type	BDS Type
xsd:anyType	Object	EObject
xsd:anySimpleType	Object	Java Object
xsd:token	Text	String
xsd:any	Object	FeatureMap
xsd:anyAttribute	Object	FeatureMap

JDBC Database Type to BOM Data Type Mapping

The following table shows mappings between:

- JDBC database types and
- TIBCO Business Studio Business Object Model (BOM) types.

See *TIBCO Business Studio Modeling User's Guide* for more detail.

JDBC Database Type	BOM Type
BIT	Boolean
TINYINT	Integer (signed)
SMALLINT	Integer (signed)
INTEGER	Integer (signed)
BIGINT	Integer (fixed - BigInteger)
FLOAT	Decimal (floating point - double)
REAL	Decimal (floating point - double)
DOUBLE	Decimal (floating point - double)
NUMERIC	Decimal (fixed point - BigDecimal)
DECIMAL	Decimal (fixed point - BigDecimal)
CHAR	Text
VARCHAR	Text
LONGVARCHAR	Text
DATE	Date
TIME	Time
TIMESTAMP	Datetime

JDBC Database Type	BOM Type
BINARY	Attachment(1)
VARBINARY	Attachment(1)
LONGVARBINARY	Attachment(1)
NULL	Text
OTHER	Text
JAVA_OBJECT	Attachment(1)
DISTINCT	Text
STRUCT	Text
ARRAY	Text
BLOB	Attachment(1)
CLOB	Text
REF	URL
DATALINK	URL
BOOLEAN	Boolean
ROWID	Text
NCHAR	Text
NVARCHAR	Text
LONGNVARCHAR	Text
NCLOB	Text
SQLXML	Text

(1) Not currently supported.

Process Primitive Data Type Mapping

TIBCO Business Studio supports the following process basic data types.

Process Primitive Type	Java Type Representation	Comments
Boolean	Boolean	

Process Primitive Type	Java Type Representation	Comments
Integer	Integer	Constrained to <=15 digits. TIBCO Business Studio validates that the upper limit is not exceeded.
Decimal	Double	A 64-bit floating point number.
Text	String	
Date	XMLGregorianCalendar	Date, without timezone offset.
Time	XMLGregorianCalendar	Time, without timezone offset.
Datetime	XMLGregorianCalendar	Date and time, with optional timezone offset.
Performer	String	

Unsupported XSD Constructs

The following XSD Constructs are not supported:

- XSD list
- XSD Redefine
- XSD Key
- XSD KeyRef
- XSD Unique
- XSD Notation

BDS Limitations

The following sub-sections contain further notes and restrictions on BDS and Forms.

Object Type

The BDS Object type is not supported in Forms, so Business Objects that include Object type attributes cannot be displayed on Forms.

Change of Order in Multiple Data

BDS supports the use of multiplicity on a sequence, choice, or group. See Multiple Instances in Sequences and Groups for details.

However:

- Data with multiplicity greater than one can only be used as an input parameter to a form.
- When an ordered set of data from such a source is loaded into a form, any ordering in the original is lost. The data is grouped by type rather than by its original ordering. The data is read-only, so the user cannot change it.

No Support for Nillable with Multiplicity Greater Than 1

BDS does not support having an element in an XSD or WSDL set to:

- nillable, and
- multipicity set to greater than 1.

BDS only allows one element in an XSD or WSDL to be set to nillable.

Fixed Attribute Overwrite

If an element or attribute in a complex object exists, it is possible in EMF to actually overwrite the fixed value with a different value. This is then persisted in the XML instead of the fixed value, thus creating invalid XML against the schema.

For example:

```
<xsd:complexType name="compType">
     <xsd:sequence>
          <xsd:element name="anElement" type="xsd:string" fixed="A fixed value"/>
     </xsd:sequence>
</xsd:complexType>
```

Multiplicity Ordering in a Sequence or Choice

If you have multiplicity (such as maxOccurs) in a sequence or choice in a user-defined BOM, the XML generated may not be valid. (Multiplicity is supported in imported schemas or WSDLs.) This problem could occur if users write scripts to populate these sequences or choices, and add elements in an incorrect order. Note that elements will appear in the XML *in the same order* that they were added in the script.

For example, the following is an XSD fragment:

```
<xs:sequence maxOccurs="unbounded">
<xs:element name="fruit" type="xs:string" minOccurs="1" maxOccurs="1"/>
<xs:element name="cake" type="xs:int" minOccurs="1" maxOccurs="1"/>
</xs:sequence>
```

Nested xsd any in Sequences

EMF is unable to handle nested sequences where there is an `xsd:any` or `xsd:anyattribute` in each sequence.

For example, in the following schema, there is a sequence within another sequence, of which contain an `xsd:any`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://example.com/
NestedAny" targetNamespace="http://example.com/NestedAny">
     <xs:element name="train" type="TrainType"/>
     <xs:complexType name="TrainType">
         <xs:sequence>
             <xs:element name="line" type="xs:string"/>
             <xs:element name="company" type="xs:string"/>
             <xs:any processContents="lax" minOccurs="1" maxOccurs="1"/>
               <xs:sequence>
                   <xs:any processContents="skip" minOccurs="1" maxOccurs="1"/>
               </xs:sequence>
         </xs:sequence>
     </xs:complexType>
</xs:schema>
```

xsd any ##local

When there is an `xsd:any`, where the namespace is set to `##local`, it is not possible to set another class in the BOM to it. This is because the BOM class will already have a namespace associated with it. EMF will not strip the namespace automatically.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://example.com/
nsLocalAny" targetNamespace="http://example.com/nsLocalAny">
     <xs:element name="TrainSpotter" type="ResearchType"/>
     <xs:complexType name="ResearchType">
        <xs:sequence>
            <xs:element name="details" type="xs:string"/>
        </xs:sequence>
     </xs:complexType>
     <xs:element name="train">
         <xs:complexType>
             <xs:sequence>
             <xs:element name="line" type="xs:string"/>
             <xs:element name="company" type="xs:string"/>
             <xs:any namespace="##local" processContents="lax" minOccurs="0"
maxOccurs="5"/>
             </xs:sequence>
         </xs:complexType>
     </xs:element>
</xs:schema>
```

Recurring Elements in Sequence

Having an ordered sequence with multiple instances of the same element name is not supported in XML schemas due to an EMF restriction.

For example, the following schema would not be supported:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://example.com/
RecurringElements" targetNamespace="http://example.com/RecurringElements">
     <xs:element name="BoatElement" type="Boat"/>
     <xs:complexType name="Boat">
         <xs:sequence>
            <xs:element name="power" type="xs:string" minOccurs="3"
maxOccurs="3"/>
            <xs:element name="hulltype" type="xs:string" minOccurs="1"
maxOccurs="1"/>
<xs:element name="power" type="xs:string" minOccurs="3" maxOccurs="3"/>
         </xs:sequence>
     </xs:complexType>
</xs:schema>
```

Attempting to import the above schema into TIBCO Business Studio will fail with the following message:

XML Schema contains unsupported duplicate element names inside the same complex type.

The block Function

EMF will not enforce a block if used on either Complex Types or Elements. They will be allowed to import and run as if the block does not exist.

For example, see the following schema fragment:

```
<xsd:complexType name="coreIdentifier" block="#all">
     <xsd:sequence>
         <xsd:element name="surname" type="xsd:string"/>
     </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="enhancedIdentifier">
     <xsd:complexContent>
          <xsd:extension base="ns1:coreIdentifier">
          <xsd:sequence>
              <xsd:element name="firstname" type="xsd:string"/>
          </xsd:sequence>
          </xsd:extension>
     </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="Household">
     <xsd:sequence>
         <xsd:element name="family" type="ns1:coreIdentifier"/>
     </xsd:sequence>
</xsd:complexType>
<xsd:element name="myBaseInfo" type="ns1:coreIdentifier"/>
<xsd:element name="myFullInfo" type="ns1:enhancedIdentifier"/>
<xsd:element name="myHousehold" type="ns1:Household"/>
```

Data Mapping

These tables show data mappings within the BOM.

See Converting Values Between Different BOM Attribute Types.

See Mapping to or from Process Basic Types.

Converting Values Between Different BOM Attribute Types

This section shows how to convert between fields of different types, for example, how to convert between a text field containing the String 123 and an integer field containing the number 123.

In some cases you can convert between two different types implicitly, without using the factory methods listed in this table. See Implicit Conversions Between Numeric Types

To Type	From Type	Example	Comments
Boolean	Text	bomField.booleanAttribute = ScriptUtil.createBoolean(bomField.textAttribute);	Parameter to createBoolean() should be true or false
Boolean	Integer-Signed	bomField.booleanAttribute = (1 == bomField.integerSigned);	
Boolean	Integer-Fixed	bomField.booleanAttribute = bomField.integerFixed.equals(ScriptUtil.createBigInteger(1));	
Boolean	Decimal Signed	bomField.booleanAttribute = (1.0 == bomField.decimalFloat);	Testing for equality may not give expected result due to floating point inaccuracies, e.g. (14/10==1.4) evaluates to false due to rounding errors

To Type	From Type	Example	Comments
Boolean	Decimal Fixed	bomField.booleanAttribute = (bomField.decimalFixed.compareTo(ScriptUtil.createBigDecimal(1)) == 0);	
Boolean	Date, Time, Datetime, Datetimetz Duration, ID, URI, Object, Attachment	N/A	N/A
Text	Boolean	bomField.text = bomField.bool;	Results in true or false value being assigned.
Text	Integer-Signed	bomField.text = bomField. intSigned;	
Text	Integer-Fixed	bomField.text = bomField.integerFixed;	
Text	Decimal Float	bomField.text = bomField.decimal;	
Text	Decimal Fixed	bomField.text = bomField.decimalFixed;	
Text	Date, Time, Datetime, Datetimetz	bomField.text = bomField.date; bomField.text = bomField.time; bomField.text = bomField.datetime; bomField.text = bomField.datetimetz;	yyyy-mm-dd hh:mm:ss yyyy-mm-ddThh:mm:ss yyyy-mm-ddThh:mm:ssZ
Text	Duration	bomField.text = bomField.duration;	For example, P12DT3H for 12 days and 3 hours.
Text	ID, URI	bomField.text = bomField.id; bomField.text = bomField.uri;	Same value
Text	Object, Attachment	N/A	N/A
Integer Signed	Text	bomField.intSigned = parseInt(bomField.text); bomField. intSigned = parseInt(bomField.text,10);	parseInt() stops at first non-number character. Copes with base.

To Type	From Type	Example	Comments
Integer Signed	Integer-Fixed	bomField. intSigned = parseInt(bomField.intFixed.toString());	Goes by a String Can result in loss of precision
Integer Signed	Decimal Float	bomField. intSigned = bomField.decFloat;	Rounds toward 0
Integer Signed	Decimal Fixed	bomField. intSigned = parseInt(bomField.intFixed.toString());	Goes by a String
Integer Signed	Boolean, Date, Time, Datetime, Datetimetz, Duration, ID, URI, Object, Attachment	N/A	N/A
Integer Fixed	Text	bomField.intFixed = ScriptUtil.createBigInteger(bomField.text);	Using ScriptUtil Factory
Integer Fixed	Integer-Signed	bomField.intFixed = ScriptUtil.createBigInteger(bomField.intSigned);	Using ScriptUtil Factory
Integer Fixed	Decimal Float	bomField.intFixed = ScriptUtil.createBigInteger(bomField.decFloat);	Rounds towards 0
Integer Fixed	Decimal Fixed	bomField.intFixed = ScriptUtil.createBigInteger(bomField.decFixed);	Rounds towards 0
Integer Fixed	Boolean, Date, Time, Datetime, Datetimetz, Duration, ID, URI, Object, Attachment	N/A	N/A
Floating Point	Text	bomField.decFloat = parseFloat(bomField.text);	NaN if starts with non-digit. Or ignores after non-digit, for example, 45z à 45.

To Type	From Type	Example	Comments
Floating Point	Integer-Fixed	bomField.decFloat =parseFloat(bomField.intFixed.toString());	First converts value to a String and then to a Floating Point. Possible loss of precision.
Floating Point	Decimal Fixed	bomField.decFloat =parseFloat(bomField.decFixed.toString());	First converts value to a String and then to a Floating Point. Possible loss of precision.
Floating Point	Boolean , Date, Time, Datetime, Datetimetz, Duration, ID, URI, Object, Attachment	N/A	N/A
Decimal Fixed	Text	bomField.decFixed = ScriptUtil.createBigDecimal(bomField.text);	Using ScriptUtil Factory
Decimal Fixed	Integer-Signed	bomField.decFixed = ScriptUtil.createBigDecimal(bomField.intSigned);	Using ScriptUtil Factory
Decimal Fixed	Decimal Float	bomField.decFixed = ScriptUtil.createBigDecimal(bomField.decFloat);	Using ScriptUtil Factory
Decimal Fixed	Integer Fixed	bomField.decFixed = ScriptUtil.createBigDecimal(bomField.text);	Using ScriptUtil Factory
Decimal Fixed	Boolean , Date, Time, Datetime, Datetimetz, Duration, ID, URI, Object, Attachment	N/A	N/A

To Type	From Type	Example	Comments
Date, Time, Datetime, Datetimetz	Text	bomField.date = DateTimeUtil.createDate(bomField.text); bomField.time = DateTimeUtil.createTime(bomField.text); bomField.datetime = DateTimeUtil.createDatetime(bomField.text); bomField.datetimetz = DateTimeUtil.createDatetimetz(bomField.text);	
Date, Time, Datetime, Datetimetz	Integer, Decimal Float	e.g. bomField.date = DateTimeUtil.createDate(bomField.intYear, bomField.intMonth, bomField.intDay);	See DateTimeUtil for more factory methods
Date, Time, Datetime, Datetimetz	Date, time, Datetime, Datetimetz	bomField.datetime = DateTimeUtil.createDatetime(bomField.date, bomField.time);	See DateTimeUtil for more factory methods
Date, Time, Datetime, Datetimetz	Boolean, Fixed Integer, Fixed Decimal, Duration, ID, URI, Object, Attachment	N/A	N/A
Duration	Text	bomField.duration = DateTimeUtil.createDuration(bomField.text);	
Duration	Signed Integer, Fixed Integer, Decimal Float, Decimal Fixed	bomField.duration = DateTimeUtil.createDuration(bomField.integerOrDecimal);	Specify duration in milliseconds using any of the 4 numeric sub-types
Duration	All other types	N/A	N/A
ID, URI	Text	bomField.uri = bomField.text; bomField.id = bomField.text;	

To Type	From Type	Example	Comments
ID, URI	All other types	N/A	N/A
Object	All Business Objects	N/A	Can only be assigned an Object attribute or a Business Object
Attachment	All Types	N/A	N/A

Mapping to or from Process Basic Types

There are 8 different Process Basic Types that Business Object Attributes can be assigned to or from. Some of these types have equivalent BDS Attribute types. Others are slightly different, as shown below.

Process Basic Type	Equivalent BOM Attribute Type	Comments
Text	Text	No problems mapping data
Decimal	Decimal (Floating Point sub-type)	No problems mapping data
Integer	Integer (Signed Integer sub-type)	The range of values supported by the Process Basic Integer is constrained to <=15 digits. The BOM Integer primitive type will only accommodate up to 10-digits. So, you need to make sure that assignments in JavaScript or in mappings are done accordingly to avoid truncation.
Boolean	Boolean	No problems mapping data
Date	Date	No problems mapping data
Time	Time	No problems mapping data
Datetime	Datetime	No problems mapping data. The Basic Datetime type can be assigned to a BOM Datetimetz Attribute, provided that it contains a timezone. If it doesn't, an exception will be raised.
Performer	Text	No problems mapping data

The Process Basic Types can be mapped to BOM Attributes of different types if the guidelines in the previous section are followed.

JavaScript Features not Supported in TIBCO BPM Scripting

Certain JavaScript features are not supported.

New Operator

The new operator from JavaScript is not supported. Instead, the factory methods must be used to create new instances of classes.

Switch Statement

The switch statement from JavaScript is not supported:

```
switch (value) {
    case constant:
         BLOCK;
         break;
    …
    default:
         BLOCK;
         break
}
```

Instead, an `if () {} else if () {}` … statement should be used.

try/catch Statement

The try/catch statement is not supported:

```
try {
        BLOCK;
}
catch (error) {
        BLOCK;
}
Finally {
        BLOCK;
}
```

The only way to catch an exception in a script is to catch the error on ScriptTask or another task.

JavaScript Functions

JavaScript functions are not supported. Code must be written out in full.

JavaScript RegExp Object

The JavaScript RegExp object is not supported. For example, the Text field `replace()` method only supports string substitution, not the substitution of a pattern matched by a regular expression.

"==="operator

The === operator (same value and type) is not supported.

JavaScript Arrays

Using [ ] for array indices is not supported, as there is no way to create JavaScript arrays in TIBCO BPM. Instead, the List type is used. Consequently, the following loop is not supported:

```
for (FIELD in ARRAYFIELD) {
         BLOCK;
}
```

Using If For and While Expressions

The curly braces are required in TIBCO Business Studio scripts for if, while, and for expressions.

Reserved Words in Scripts

Array	Boolean	Date	Math	Number	Object
RegExp	String	abstract	assert	bdsId(1)	bdsVersion1
boolean	break	byte	case	catch	char
class	const	continue	debugger	default	delete
do	double	else	enum	export	extends
false	final	finally	float	for	function
goto	if	implements	import	in	instanceof
int	interface	long	native	new	null
package	private	protected	public	return	short
static	strictfp	super	switch	synchronized	this
throw	throws	transient	true	try	typeof
upper	var	void	volatile	while	with

(1) Cannot be used, regardless of case. Other reserved words cannot be used in the case shown, but can be used by changing the case (for example, `while` is prohibited, but `WHILE` is acceptable).

Business Data Scripting

TIBCO Business Studio used with TIBCO BPM enables you to write scripts for various purposes.

Standard JavaScript is supported. This appendix describes the additional functions that are supported to help with BDS scripting.

Static Factory Methods

This section describes the following classes of factory methods: DataTimeUtil, ScriptUtil, IpeScriptUtil.

- DateTimeUtil
- ScriptUtil
- IpeScriptUtil (and ScriptUtil) Conversion Date and Time and String Functions

DateTimeUtil

These methods are used to create date and time objects of the types used in BPM, as shown in the following tables. Parameters, such as seconds and minutes, should be in the range 0-59. Other parameters, such as year, month, day, hours, and seconds, should also be within normal ranges.

Type Name	Factory Method
Date *This is a native type used in BPM for storing dates in YYYY-MM-DD format only.*	**createDate(int year, int month, int day)**
	createDate(Text isoFormatDate) This ignores any timezone offset. For example: 2009-11-30T23:50:00-05:00 becomes 2009-11-30.
	createDate(Datetime datetime) This ignores any timezone offset.
	createDate(Datetimetz datetime, Boolean normalize) This is a Boolean that, if set to True, normalizes to Zulu time. For example, 2009-11-30T23:50:00-05:00 becomes 2009-12-01. If set to False, it strips timezone. The same example date becomes 2009-11-30.
	createDate() This creates a Date object set to today's date.
Time	**createTime(int hour, int minute, int second, BigDecimal fractionalSecond)**
	createTime(int hour, int minute, int second, int millisecond)
	createTime(Text isoFormatTime) This ignores any timezone offset.
	createTime(Datetime datetime) This ignores any timezone offset.

Type Name	Factory Method
	createTime(Datetimetz datetimetz, Boolean normalize) This is a Boolean that, if set to True, normalizes to Zulu time. If it is set to False, it strips timezone as with **Date** previously mentioned.
	createTime() This creates a Time object set to the current time.
Datetime	**createDatetime(BigInteger) year, int month, int day, int hour, int minute, int second, BigDecimal fractionalSecond)**
	createDatetime(int year, int month, int day, int hour, int minute, int second, int millisecond)
	createDatetime(String lexicalRepresentation) This creates a new Datetime by parsing the String as a lexical representation. The Timezone offset is to be set only if specified in the string.
	createDatetime(Date **date, Time time)** No timezone offset set.
	createDatetime(Datetimetz datetime, Boolean normalize) This is a Boolean that, if set to True, normalizes to Zulu time. If it is set to False, it strips timezone, as with **Date** above.
	createDateTime() This creates a Datetime object set to the current date and time.
Datetimetz	**createDatetimetz(BigInteger) year, int month, int day, int hour, int minute, int second, BigDecimal fractionalSecond, int timezone_offset)** This is a constructor allowing for complete value spaces allowed by W3C XML Schema 1.0 recommendation for `xsd:dateTime` and related built-in datatypes.
	createDatetimetz(int year, int month, int day, int hour, int minute, int second, int millisecond, int timezone_offset) This is a constructor of value spaces that a `java.util.GregorianCalendar` instance would need to convert to an XMLGregorianCalendar instance.
	createDatetimetz(Datetime datetime, Integer offset_minutes)
	createDatetimetz(Date **date, Time time, Integer offset_minutes)**
	createDatetimetz(String lexicalRepresentation) This creates a new Datetimetz by parsing the String as a lexical representation. It takes the timezone offset from string, and defaults to zulu time if not specified.

Type Name	Factory Method
	createDateTimetz() This creates a Datetimetz object set to the current date and time.
Duration	**createDuration(boolean isPositive, BigInteger years, BigInteger months, BigInteger days, BigInteger hours, BigInteger minutes, BigDecimal seconds)** This obtains a new instance of a Duration specifying the Duration as isPositive, years, months, days, hours, minutes and seconds.
	createDuration(boolean isPositive, int years, int months, int days, int hours, int minutes, int seconds) This obtains a new instance of a Duration specifying the Duration as isPositive, years, months, days, hours, minutes, seconds.
	createDuration(long durationInMilliSeconds) This obtains a new instance of a Duration specifying the Duration as milliseconds.
	createDuration(String lexicalRepresentation) This obtains a new instance of a Duration specifying the Duration as its string representation PnYnMnDTnHnMnS.

DataUtil

DataUtil provides a single method that allows you to create a List object for use in scripting.

Return Type	method	Description
List*objects*	createList()	Create a List object for use in scripting.

ScriptUtil

ScriptUtil provides methods to create various types of object, to modify Duration objects, and to serialize business objects into or deserialize them from their XML representation.

Some of the functions provided by the ScriptUtil factory are also supported in an IpeScriptUtil factory. See IpeScriptUtil (and ScriptUtil) Conversion Date and Time and String Functions for more information.

Factory Methods

Return Type	Function	Comments
BigInteger	**createBigInteger(Integer)**	
	createBigInteger(Text)	
	createBigInteger(BigDecimal)	
BigDecimal	**createBigDecimal(Integer)**	

Return Type	Function	Comments
	createBigDecimal(Decimal)	This uses an implicit MathContext.DECIMAL64.
	createBigDecimal (Text)	
	createBigDecimal (BigInteger)	
MathContext	**createMathContext(Integer setPrecision)**	
	createMathContext(Integer setPrecision, RoundingMode setRoundingMode)	RoundingMode is an enumeration with the following values: `CEILING`, `FLOOR`, `UP`, `DOWN`, `HALF_UP`, `HALF_DOWN`, `HALF_EVEN`, `UNNECESSARY`
Boolean	**createBoolean(Text booleanText)**	

The MathContext object specifies the number of digits used to store the number (that is, the size of the mantissa), and also how to round numbers (of various forms, up, down, and nearest to use). For more information about MathContext, see http://java.sun.com/j2se/1.5.0/docs/api/java/math/MathContext.html.

MathContext is not used when creating BigDecimal objects unless otherwise stated. It is the scale parameter in BigDecimal methods that specifies the number of decimal places that should be maintained in the objects. See Other Supported Methods for more details on this format.

Duration Methods

The following ScriptUtil methods are provided to enhance the basic interfaces provided for Duration objects.

Return Type	Method	Comments
BigDecimal	getFractionalSeconds(Duration *dur*)	Returns the fractional part of the seconds of the duration.
Integer	getMilliseconds(Duration *dur*)	Returns the fractional second part of the duration measured in milliseconds. For example: `dstField.integerSigned = ScriptUtil.getMilliseconds(srcField.duration);`

XML Serialization Methods

The following ScriptUtil methods allow you to serialize business objects into or deserialize them from their XML representation.

Return Type	Method	Comments
Boolean	isConvertableToXML(Class *object*)	Returns `true` if the specified *object* can be converted to XML.
String	toXML(Class *object*)	Returns the XML representation of the specified *object*.
String	toXML(Class *object*, String *type*)	Returns the XML representation of the specified *object*, using the specified *type* (which can either be the element name or fully qualified class name.
Class	fromXML(String *string*)	Returns the business object represented by the the specified XML *string*.

IpeScriptUtil (and ScriptUtil) Conversion Date and Time and String Functions

The following functions are supported both in the **ScriptUtil** factory and, for compatibility with the TIBCO iProcess Suite, in an **IpeScriptUtil** factory designed for assistance in migrating iProcess scripts to BPM.

iProcess expressions support the addition and subtraction of dates and times, expressed in arithmetic form "*date* + *num*", "*time* - *time*", and so on. Operations of this kind in BPM should be performed by the supported add() or subtract() methods on the date and time objects. Functions relating to the internal operations of iProcess or its environment are not supported.

You can access these functions using either of the following:

- **ScriptUtil.**<*FunctionName*>
- **IpeScriptUtil.**<*FunctionName*>.

Return Type	Syntax and Example	Comments
Conversion Functions		

Text	DATESTR(*Date*)
	
	Field = ScriptUtil.DATESTR(DateTimeUtil.createDate());
		Converts a date field into a locale-specific string, for example, 20/08/2009.

Decimal	NUM(*Text*)
	
	Field = ScriptUtil.NUM("123");
		Converts String to Decimal.

Text	STR(*Decimal DECIMAL, Integer DECIMALS*) ScriptUtil.STR(2.3,2); // Generate "2.30"	Converts from Decimal to a string with a specified number of decimal places.

Return Type	Syntax and Example	Comments

Text	STRCONVERT(Text *TEXT*, Integer *OPCODE*)
	
	ScriptUtil.STRCONVERT("test",32);
		Depending on which bits are set in the opcode parameter, the following conversions are applied to the text parameter - the result is returned:
		
		1 Delete all spaces.
		
		2 Delete all leading spaces.
		
		4 Delete all trailing spaces.
		
		8 Reduce sequences of multiple spaces to single spaces.
		
		16 Convert to lowercase.
		
		32 Convert to uppercase.

Text	STRTOLOWER(*TEXT*)
	
	ScriptUtil.STRTOLOWER("TEST");
		Returns a lowercase copy of the string passed in.

Text	STRTOUPPER(*TEXT*)
	
	ScriptUtil.STRTOUPPER("test");
		Returns an uppercase copy of the string passed in.

Text	TIMESTR(*TIME*)	Converts a time field into a locale-specific string, for example, 21:23.
Date and Time Functions		

Date	**CALCDATE(Date** *date*, **Integer** *dDy*, **Integer** *dWk*, **Integer** *dMo*, **Integer** *dYr*)
	
	ScriptUtil.CALCDATE(DateTimeUtil .createDate(), 0, 0, 1, 0);
		Adds an offset to a date. Note that the offset is not restricted by the units used, for example, an offset expressed as a number of days is not restricted to the number of days in a week or a month.

Time	**Time CALCTIME**(**Time** time, **Integer** dHr, **Integer** dMi) newTime = ScriptUtil.CALCTIME(DateTimeUtil .createTime("12:00:00"),2,40);	Adds an offset to a time. The time plus offset is returned by the function.
Integer	**CALCTIMECARRYOVER**(**Time** time, **Integer** dHr, **Integer** dMi) carryDays = ScriptUtil.CALCTIMECARRYOVE R(DateTimeUtil.createTime("12:00:00 "),2,40);	Adds an offset to a time. The function returns an integer whose value is the number of days apart. The new time is from the original time, so for a 48 hour offset, the function would return 2 (or -2 for -48hours).

Date	**DATE(Integer** *day*, **Integer** *mon*, **Integer** *year***)**
	
	ScriptUtil.DATE(31,12,2009)
		Constructs a Date.

Return Type	Syntax and Example	Comments
Integer	DAYNUM (Date *date*) ScriptUtil.DAYNUM(DateTimeUtil.createDate("2001-10-08"));	Returns the day of the month of the specified date.
Text	**DAYSTR (Date *date*)** ScriptUtil.DAYSTR(DateTimeUtil.createDate("2001-10-08"));	Returns the day of the week as a string, for example, Monday, for the specified date.
Integer	**HOURNUM (Time *time*)** ScriptUtil.HOURNUM(DateTimeUtil.createTime("06:24:00"));	Returns the hour of the specified time.
Integer	**MINSNUM (Time *time*)** DateTimeUtil.MINSNUM(DateTimeUtil.createTime("06:24:00"));	Returns the minutes from the specified time.
Integer	MONTHNUM (Date *date*) ScriptUtil.MONTHNUM(DateTimeUtil.createDate("2001-10-08"));	Returns the month number (1-12) from the specified date.
Text	MONTHSTR (Date *date*) ScriptUtil.MONTHSTR(DateTimeUtil.createDate("2001-10-08"));	Returns the month name from the specified date, for example, January.
Time	TIME (Integer *hours*, Integer *minutes*) ScriptUtil.TIME(6,24);	Constructs a time.
Integer	Integer WEEKNUM (Date *date*) ScriptUtil.WEEKNUM(DateTimeUtil.createDate("2001-10-08"));	Returns the week number from the specified date.
Integer	YEARNUM (Date *date*) ScriptUtil.YEARNUM(DateTimeUtil.createDate("2001-10-08"));	Returns the year from the specified date.
String Functions		
Integer	RSEARCH ScriptUtil.RSEARCH("abc", "junkabcdefs");	Reverse search for *substring* in *string*. The indices are 1-based. Returns "5".
Integer	SEARCH ScriptUtil.SEARCH("abc", "junkabcdefs");	Search for *substring* in *string*. The indices are 1-based. Returns "5".

Return Type	Syntax and Example	Comments
Integer	STRLEN (*Text*) ScriptUtil.STRLEN("abcdef");	Count the number of characters in a string, returning the string length.
Text	SUBSTR ScriptUtil.SUBSTR("abcdefgh", 3, 3);	The indices are 1-based. Returns "cde".

BigDecimal	**getFractionalSecond(Duration dur)**
	dstField.integerSigned = ScriptUtil.getFractionalSecond(srcField.duration);
		Returns the fractional part of the seconds of the duration (0 <= value < 1.0).

Int	**getMilliseconds(Duration dur)**
	dstField.integerSigned = ScriptUtil.getMilliseconds(srcField.duration);
		Returns the duration measured in milliseconds

BDSObject	**copy(BDSObject)**	Copies a BDS object as a BDS Object so it can be included in a second collection, since each BDS object can only be referenced from one collection.

List<BDSObject	**copyAll(SrcBDSObjectList)**
	destList.addAll(ScriptUtil.copyAll (sourceList));
		Copies all the objects in the source List returning a new List that can be passed to the add `All()` method of another List.
		Note that this is for BDS objects only. It is not for use with Process Array fields.

Void	**setArrayElement(arrayData, Integer index, Object value)**	Set an element in a JavaScript list (that represents a process data array). It can be used to set the existing **or** append a new item to the end of the list if the index is passed as the zero-based index of the last item+1 (that is, the current size of list) for example, **setArrayElement(array, array.size(), value)**.
Object	**getArrayElement(arrayData, Integer index)**	Get an element in a JavaScript list (that represents a process data array).

BOM Native Type Methods

This section describes supported methods using different BOM native types.

These include the following:

- Fixed Length Integer (BigInteger) Methods
- Fixed Point Decimal (BigDecimal) Methods
- IpeScriptUtil (and ScriptUtil) Conversion Date and Time and String Functions
- Duration Methods
- Text (String) Methods

Fixed Length Integer (BigInteger) Methods

The following methods using the BigInteger numeric format are supported in TIBCO BPM.

Type	Method	Notes
BigInteger	abs()	Returns a BigInteger whose value is the absolute value of this BigInteger.
BigInteger	add(BigInteger val)	Returns a BigInteger whose value is (`this` plus `val`).
int	compareTo(BigInteger val)	Compares this BigInteger with the specified BigInteger.
int	compareTo(Object o)	Compares this BigInteger with the specified Object.
BigInteger	divide(BigInteger val)	Returns a BigInteger whose value is (`this` divided by `val`).
boolean	equals(Object x)	Compares this BigInteger with the specified Object for equality.
BigInteger	gcd(BigInteger val)	Returns a BigInteger whose value is the greatest common divisor of abs(`this`) and abs(`val`).
BigInteger	max(BigInteger val)	Returns the maximum of this BigInteger and `val`.
BigInteger	min(BigInteger val)	Returns the minimum of this BigInteger and `val`.
BigInteger	mod(BIgInteger m)	Returns a BigInteger whose value is (`this` mod `m`).
BigInteger	multiply(BigInteger val)	Returns a BigInteger whose value is (`this` times `val`).
BigInteger	negate()	Returns a BigInteger whose value is (minus `this`).
BigInteger	pow(int exponent)	Returns a BigInteger whose value is (`thisexponent`).
BigInteger	remainder(BigInteger val)	Returns a BigInteger whose value is (`this % val`).
BigInteger	subtract(BigInteger val)	Returns a BigInteger whose value is (`this` minus `val`).

Fixed Point Decimal (BigDecimal) Methods

Some numeric objects in BPM are expressed in BigDecimal format.

BigDecimal format can cope with arbitrarily large numbers. However, it is necessary to specify what precision to use in certain circumstances. For example, calculating one-third would produce an exception if precision is not specified. BigDecimal supports three standard levels of precision:

- **static MathContext DECIMAL32**: A MathContext object with a precision setting matching the IEEE 754R Decimal32 format, seven digits, a rounding mode of HALF_EVEN, and the IEEE 754R default (which is equivalent to "float" arithmetic).
- **static MathContext DECIMAL64**: A MathContext object with a precision setting matching the IEEE 754R Decimal64 format, 16 digits, a rounding mode of HALF_EVEN, and the IEEE 754R default (which is equivalent to "double" arithmetic).
- **static MathContext DECIMAL128**: A MathContext object with a precision setting matching the IEEE 754R Decimal128 format, 34 digits, a rounding mode of HALF_EVEN, and the IEEE 754R default.

In addition, when using the BigDecimal type, the rounding rules can be specified when a particular method of rounding is required for a particular type of calculation (for example, tax calculations). The type of rounding to be used by a BigDecimal operation can be specified when creating a MathContext, or passed directly to the relevant methods of BigDecimal. Possible values are:

- CEILING
- FLOOR
- UP
- DOWN
- HALF_UP
- HALF_DOWN
- HALF_EVEN
- UNNECESSARY

The following table lists the methods available for BigDecimal objects.

Type	Method	Notes
BigDecimal	abs()	Returns a BigDecimal whose value is the absolute value of this BigDecimal, and whose scale is `this.scale()`.
BigDecimal	add(BigDecimal augend)	Returns a BigDecimal whose value is `(this + augend)`, and whose scale is `max(this.scale(), augend.scale())`.
BigDecimal	add(BigDecimal augend, MathContext mc)	Returns a BigDecimal whose value is `(this + augend)`, with rounding according to the context settings.
int	compareTo(BigDecimal val)	Compares this BigDecimal with the specified BigDecimal.

Type	Method	Notes
BigDecimal	divide(BigDecimal divisor)	Returns a BigDecimal whose value is `(this / divisor)`, and whose preferred scale is `(this.scale() - divisor.scale())`. If the exact quotient cannot be represented (because it has a non-terminating decimal expansion), an ArithmeticException is thrown.
BigDecimal	divide(BigDecimal divisor, int scale, RoundingMode roundingMode)	Returns a BigDecimal whose value is `(this / divisor)`, and whose scale is as specified.
BigDecimal	divide(BigDecimal divisor, MathContext mc)	Returns a BigDecimal whose value is `(this / divisor)`, with rounding according to the context settings.
BigDecimal	divide(BigDecimal divisor, RoundingMode roundingMode)	Returns a BigDecimal whose value is `(this / divisor)`, and whose scale is `this.scale()`.
BigDecimal	divideToIntegralValue(BigDecimal divisor)	Returns a BigDecimal whose value is the integer part of the quotient `(this / divisor)` rounded down.
BigDecimal	divideToIntegralValue(BigDecimal divisor, MathContext mc)	Returns a BigDecimal whose value is the integer part of `(this / divisor)`.
BigDecimal	max(BigDecimal val)	Returns the maximum of this BigDecimal and `val`.
BigDecimal	min(BigDecimal val)	Returns the minimum of this BigDecimal and `val`.
BigDecimal	multiply(BigDecimal multiplicand)	Returns a BigDecimal whose value is `(this × multiplicand)`, and whose scale is `(this.scale() + multiplicand.scale())`.
BigDecimal	multiply(BigDecimal multiplicand, MathContext mc)	Returns a BigDecimal whose value is `(this × multiplicand)`, with rounding according to the context settings.
BigDecimal	negate()	Returns a BigDecimal whose value is `(-this)`, and whose scale is `this.scale()`.
BigDecimal	pow(int n)	Returns a BigDecimal whose value is `(thisn)`. The power is computed exactly, to unlimited precision.
BigDecimal	pow(int n, MathContext mc)	Returns a BigDecimal whose value is `(thisn)`.
int	precision()	Returns the precision of this BigDecimal.
BigDecimal	remainder(BigDecimal divisor)	Returns a BigDecimal whose value is `(this % divisor)`.

Type	Method	Notes
BigDecimal	remainder(BigDecimal divisor, MathContext mc)	Returns a BigDecimal whose value is `(this % divisor)`, with rounding according to the context settings.
BigDecimal	round(MathContext mc)	Returns a BigDecimal rounded according to the MathContext settings.
int	scale()	Returns the scale of this BigDecimal.
BigDecimal	scaleByPowerOfTen(int n)	Returns a BigDecimal whose numerical value is equal to `(this * 10n).`
BigDecimal	setScale(int newScale)	Returns a BigDecimal whose scale is the specified value, and whose value is numerically equal to this BigDecimal's.
BigDecimal	setScale(int newScale, RoundingMode roundingMode)	Returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
int	signum()	Returns the signum function of this BigDecimal.
BigDecimal	stripTrailingZeros()	Returns a BigDecimal that is numerically equal to this one, but with any trailing zeros removed from the representation.
BigDecimal	subtract(BigDecimal subtrahend)	Returns a BigDecimal whose value is `(this - subtrahend)`, and whose scale is `max(this.scale(), subtrahend.scale())`.
BigDecimal	subtract(BigDecimal subtrahend, MathContext mc)	Returns a BigDecimal whose value is `(this - subtrahend)`, with rounding according to the context settings.
String	toEngineeringString()	Returns a string representation of this BigDecimal, using engineering notation if an exponent is needed.
String	toPlainString()	Returns a string representation of this BigDecimal without an exponent field.
String	toString()	Returns the string representation of this BigDecimal, using scientific notation if an exponent is needed.
BigDecimal	ulp()	Returns the size of an `ulp` (a unit in the last place) of this BigDecimal.
BigInteger	unscaledValue()	Returns a BigInteger whose value is the unscaled value of this BigDecimal.

Date Time Datetime and Datetimetz (XMLGregorianCalendar) Methods

Type	Method	Date	Time	Date Time	Date Time tz	gDay	gMonth	gMonthDay	gYear	gYearMonth
void	**add(Duration duration)** Add duration to this instance.	Y	Y	Y	Y	Y	Y	Y	Y	Y
void	**clear()** Unset all fields to undefined.	Y	Y	Y	Y	Y	Y	Y	Y	Y
int	**compare(XMLGregorian Calendar xmlGregorianCalendar)** Compare two instances of XMLGregorianCalendar	Y	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
boolean	**equals(Object obj)** Indicates whether parameter obj is equal to this one.	Y	Y	Y	Y	Y	Y	Y	Y	Y
int	**getDay()** Return day in month or DatatypeConstants.FIELD_UNDEFINED.	Y	n/a	Y	Y	Y	n/a	Y	Y	n/a
BigDecimal	**getFractionalSecond()** Return fractional seconds.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
int	**getHour()** Return hours or `DatatypeConstants.FIELD_UNDEFINED`.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
int	**getMillisecond()** Return millisecond precision of `getFractionalSecond()`	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
int	**getMinute()** Return minutes or `DatatypeConstants.FIELD_UNDEFINED`.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a

Type	Method	Date	Time	DateTime	DateTimetz	gDay	gMonth	gMonthDay	gYear	gYearMonth
int	**getMonth()** Return number of month or `DatatypeConstants.FIELD_UNDEFINED`.	Y	n/a	Y	Y	n/a	Y	Y	n/a	Y
int	**getSecond()** Return seconds or `DatatypeConstants.FIELD_UNDEFINED`.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
int	**getTimezone()** Return timezone offset in minutes or `DatatypeConstants.FIELD_UNDEFINED` if this optional field is not defined.	n/a	n/a	Y	Y	n/a	n/a	n/a	n/a	n/a
int	**getYear()** Return low order component for XML Schema 1.0 dateTime datatype field for year or DatatypeConstants.FIELD_UNDEFINED.	Y	n/a	Y	Y	n/a	n/a	n/a	Y	Y
void	**setDay(int day)** Set days in month.	Y	n/a	Y	Y	Y	n/a	Y	n/a	n/a
void	**setFractionalSecond(BigDecimal fractional)** Set fractional seconds.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setHour(int hour)** Set hours.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setMillisecond(int millisecond)** Set milliseconds.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setMinute(int minute)** Set minutes.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a

Type	Method	Date	Time	DateTime	DateTimetz	gDay	gMonth	gMonthDay	gYear	gYearMonth
void	**setMonth(int month)** Set month.	Y	n/a	Y	Y	n/a	Y	Y	n/a	Y
void	**setSecond(int second)** Set seconds.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setTime(int hour, int minute, int second)** Set time as one unit.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setTime(int hour, int minute, int second, BigDecimal fractional)** Set time as one unit, including the optional infinite precision fractional seconds.	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setTime(int hour, int minute, int second, int millisecond)** Set time as one unit, including optional milliseconds. This method should not be used in scripts where the **time** field has been set before the script. In this case it is best to use two separate calls to set the hour, minute, second, and millisecond fields as: **setTime(int hour, int minute, int second)** **setMillisecond(int millisecond)**	n/a	Y	Y	Y	n/a	n/a	n/a	n/a	n/a
void	**setTimezone(int offset)** Set the number of minutes in the timezone offset.	n/a	n/a	Y	Y	n/a	n/a	n/a	n/a	n/a

Type	Method	Date	Time	Date Time	Date Timetz	gDay	gMonth	gMonthDay	gYear	gYearMonth
void	**setYear(BigInteger year)** Set low and high order component of XSD dateTime year field.	Y	n/a	Y	Y	n/a	n/a	n/a	Y	Y
void	**setYear(int year)** Set year of XSD dateTime year field. This method should not be used in scripts where the date field has been set before the script. In this case it is best to use setYear(BigInteger year) instead.	Y	n/a	Y	Y	n/a	n/a	n/a	Y	Y
String	**toXMLFormat()** Return the lexical representation of this instance.	Y	Y	Y	Y	Y	Y	Y	Y	Y

Duration Methods

Type	Method
Duration	**add(Duration rhs)** Computes a new duration whose value is `this+rhs`.
int	**compare(Duration duration)** Partial order relation comparison with this Duration instance.
boolean	**equals(Object duration)** Checks if this duration object has the same duration as another Duration object.
int	**getDays()** Obtains the value of the DAYS field as an integer value, or 0 if not present.
int	**getHours()** Obtains the value of the HOURS field as an integer value, or 0 if not present.
int	**getMinutes()** Obtains the value of the MINUTES field as an integer value, or 0 if not present.

Type	Method
int	**getMonths()** Obtains the value of the MONTHS field as an integer value, or 0 if not present.
int	**getSeconds()** Obtains the value of the SECONDS field as an integer value, or 0 if not present.
int	**getYears()** Get the years value of this Duration as an int or 0 if not present.
boolean	**isLongerThan(Duration duration)** Checks if this duration object is strictly longer than another Duration object.
boolean	**isShorterThan(Duration duration)** Checks if this duration object is strictly shorter than another Duration object.
Duration	**multiply(BigDecimal factor)** Computes a new duration whose value is factor times longer than the value of this duration.
Duration	**multiply(int factor)** Computes a new duration whose value is factor times longer than the value of this duration.
Duration	**subtract(Duration rhs)** Computes a new duration whose value is `this-rhs`.

Text (String) Methods

The following methods are supported for use with String objects.

Method	Description
Properties	
length	Returns the length of a string.
Methods	
charAt(index)	Returns the character at the specified zero-based index.
indexOf(str[,fromIndex])	Returns the position of the first found occurrence of a specified value in a string.
lastIndexOf(str[,fromIndex])	Returns the position of the last found occurrence of a specified value in a string.

Method	Description
replace()	Searches for a match between a substring (or regular expression) and a string, and replaces the matched substring with a new substring.
slice()	Extracts a part of a string and returns a new string.
substr()	Extracts the characters from a string, beginning at a specified start position, and through the specified number of characters.
substring()	Extracts the characters from a string, between two specified indices.
toLowerCase()	Converts a string to lowercase letters.
toUpperCase()	Converts a string to uppercase letters.

Other Supported Methods

The methods described in the sections List Methods and ListIterator Methods are supported for the types of object listed. Methods not described are not supported.

This section includes:

- List Methods
- ListIterator Methods

List Methods

The following List methods are supported for use in manipulating a range of values.

Type	Method	Notes
boolean	add(E o)	Appends the specified element to the end of this list.
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
void	clear()	Removes all of the elements from this list.
boolean	contains(Object o)	Returns true if this list contains the specified element.
E	get(int index)	Returns the element at the specified position in this list.
boolean	isEmpty()	Returns true if this list contains no elements.
ListIterator<E>	listIterator()	Returns a list iterator of the elements in this list (in proper sequence).
E	remove(int index)	Removes the element at the specified position in this list.

Type	Method	Notes
boolean	remove(Object o)	Removes the first occurrence in this list of the specified element.
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int	size()	Returns the number of elements in this list.
List	subList(int fromIndex, int toIndex)	Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
boolean	addAll(Collection c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).

ListIterator Methods

The following ListIterator methods are supported.

Type	Method	Notes
void	add(E o)	Inserts the specified element into the list.
boolean	hasNext()	Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious()	Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next()	Returns the next element in the list.
int	nextIndex()	Returns the index of the element that would be returned by a subsequent call to next.
E	previous()	Returns the previous element in the list.
int	previousIndex()	Returns the index of the element that would be returned by a subsequent call to previous.
void	remove()	Removes from the list the last element that was returned by next or previous.
void	set(E o)	Replaces the last element returned by next or previous with the specified element.

Other JavaScript Functions

Some other JavaScript methods are supported for Math objects.

Math Methods

The following methods are supported for use with Math objects.

Method	Description
Properties	
E	Returns Euler's number (approximately 2.718).
LN2	Returns the natural logarithm of 2 (approximately 0.693).
LN10	Returns the natural logarithm of 10 (approximately 2.302).
PI	Returns *Pi* (approximately 3.14159).
Methods	
abs(*x*)	Returns the absolute value of *x*.
acos(*x*)	Returns the arccosine of *x*, in radians.
asin(*x*)	Returns the arcsine of *x*, in radians.
atan(*x*)	Returns the arctangent of *x*, as a numeric value between -*Pi*/2 and +*Pi*/2 radians.
ceil(*x*)	Returns *x*, rounded up to the nearest integer.
cos(*x*)	Returns the cosine of *x* (where *x* is in radians).
exp(*x*)	Returns the value of *E*x.
floor(*x*)	Returns *x*, rounded down to the nearest integer.
log(*x*)	Returns the natural logarithm (base E) of *x*.
max(*x*,*y*, *z*, ... , *n*)	Returns the number with the highest value, using 2 arguments.
min(*x*,*y*, *z*, ... , *n*)	Returns the number with the lowest value, using 2 arguments.
pow(*x*,*y*)	Returns the value of *x* to the power of *y*.
random()	Returns a random number between 0 and 1.
round(*x*)	Rounds *x* to the nearest integer.
sin (*x*)	Returns the sine of *x* (where *x* is in radians).
sqrt(*x*)	Returns the square root of *x*.
tan(*x*)	Returns the tangent of the angle *x*.

Process Manager and Work Manager Scripting

TIBCO Business Studio used with TIBCO BPM enables you to write scripts for various purposes. This section describes the additional functions that are provided to help with process scripting, work item and organization model scripting.

Process Instance Attributes and Methods

This section summarizes the attributes and methods that are available for accessing information about process instances using the Process class.

Attribute/Method	Example	Description
priority : Integer	priority = Process. priority;	Returns priority of the Process Instance
getName() : String	name = Process.getName();	Returns Process Template name
getDescription() : String	description = Process. getDescription();	Returns description of Process
getStartTime() : Datetimetz	start = Process. getStartTime();	Returns date/time the Process Instance was started
getPriority() : Integer	priority = Process. getPriority();	Returns priority of the Process Instance
getOriginator() : String	originator = Process. getOriginator();	Returns originator of Process Instance, e.g. `uid=admin, ou=system`
getId() : String	id = Process. getId();	Returns Process Instance ID, for example, `pvm:0a128cu`

Attribute/Method	Example	Description
getActivityLoopIndex() : Integer	index = Process. getActivityLoopIndex()	• Retrieves the most local loop index. • Index starts from 0 (zero) for first loop/ multi-instance instance. • In a parallel loop/multi-instance task or embedded sub-process, all loop instances run concurrently. To pass the loop index of a particular loop instance to a task: – Use the input mapping functionality if this is available. – If it is not, use a local data field. – Use a process-level data field only as a last resort. Each loop instance will try to set the field with the index of that particular loop instance; but the field is common to all loop instances in the process and can only contain a single value. A task can successfully get and set the index in the same transaction, but if there is any delay in getting the index the field may have been overwritten by another instance. • For a non-multi-instance task, this will be the loop index for the nearest multi-instance embedded sub-process ancestor (defaulting to 0 if no multi-instance ancestor found). • In nested multi-instance situations, the user can transfer an embedded sub-process loop index into a embedded sub-process local data field.
addActivityLoopAdditionalInstances: (String, Integer)	add = Process.addActivityLoopAdditionalInstances ("UserTask",1)	Adds additional instances to a multi-instance loop task while that task is in progress
getActivityType(String) : String	type = Process.getActivityType('UserTask2');	Returns task type, for example, `userTask`
getActivityState(String) : String	state = Process.getActivityState('UserTask2');	Returns task state, for example, `done.state`
getActivityStartTime(String) : Datetimetz	started = Process.getActivityStartTime('UserTask2');	Returns time task was started

Attribute/Method	Example	Description
getActivityCompletionTime(String) : Datetimetz	completed = Process. getActivityCompletionTime('UserTask2');	Returns time task was completed
getActivityDeadline(String) : Datetimetz	deadtime = Process.getActivityDeadline('UserTask2');	Returns task deadline time

getActivityAttribute(String, String) : String	workItemId = Process.getActivityAttribute('UserTask2','WorkItemId');
	 completer = Process.getActivityAttribute('UserTask2','Completer');
		Returns value of attribute, for example, `430`
		 Only WorkItemId and Completer are supported as attribute names.

getActivityArrayAttribute(String, String) : List<String>	attrs = Process.getActivityArrayAttribute('UserTask2','Participant');	Returns array of attribute values. Only Participant is supported as an attribute name.**
setPriority() : Integer	priority = Process.setPriority();	Once a process instance has been created, it can change its own priority. The default value is 200. Valid entries are 100, 200, 300 and 400.
getOriginatorName(): String	name = Process.getOriginatorName()	Returns the process instance originator login name. For use with a pageflow or business process.
setContextVariable (String varName, BusinessObject data)	contextVariable = Process.setContextVariable (cargoType, steel);	For use with pageflow processes. Enables a BDS context variable to be created as an attribute on the main process. These context variables can be accessed by the main process or any of its sub-processes. For a pageflow process, there is only one thread of execution so a single script can access a context variable in a thread-safe manner. These methods are not intended for use in a business process.
getContextVariable (String varName, String className)	contextVariable = Process.getContextVariable (cargoType, "className");	For use with pageflow processes. Gets a BDS context variable created as in the previous method.

Attribute/Method	Example	Description
auditLog (String message)	Process.auditLog ("Unknown Cargo Type " + cargoType.name) ;	Adds a simple text entry to the audit log (process-instance related event log entry). Use of the Process.auditLog() method is not supported for in-memory process instances (service processes, pageflow processes or business services). If the Process.auditLog() method is used on one of these processes (whether started from within a parent business process or independently), no audit entries will be generated.

Organization Model Attributes and Methods

Process Manager scripting supports all the organization model methods that are supported in Work Manager scripts.

These are defined in OrgModel.

TIBCO Business Studio used with TIBCO BPM enables you to write scripts for various purposes. WorkManagerFactory describes the additional functions that are provided to help with work item and organization model scripting.

WorkManagerFactory

This table lists the WorkManagerFactory attributes and methods provided.

Attribute / Method	Comments
WorkManagerFactory	
getWorkItem() : WorkItem	Returns WorkItem object. See WorkItem for properties & methods.
getOrgModel() : OrgModel	Returns an OrgModel object. See OrgModel for methods.
getOrgModelByVersion(version: Integer) : OrgModel	Returns an OrgModel object. The parameter specifies which major version of the model is required.

WorkItem

This table lists the WorkItem attributes and methods provided.

Attribute / Method	Comments
WorkItem	
cancel : Boolean	Cancels an API that exists in BRM.
description : Text	The description of the work item.

Attribute / Method	Comments
priority : Integer	The specified priority of the work item.
getId() : Integer	Returns the work item's unique ID.
getVersion() : Integer	Returns the version number of the work item.
getWorkItemResource() : EntityDetail	Returns a resource that has this work item. It contains all the organizational entities whose work list currently contains this work item.
getWorkItemOffers() : List<EntityDetail>	Returns a work item resource object for the work item. If the item was originally offered to more than one organizational entity and is now allocated, this will contain all the entities to which the item was originally offered.
getContext() : ItemContext	Returns the work item's context information and provides read only methods to access the following information: • activity ID • activity name • application name • application instance • application ID • application instance description
getSchedule() : ItemSchedule	Returns the work item's schedule information and provides read only methods to access the start date and target date.
workItemAttributes.attribute1: Integer	These work item attributes (attribute1 and the others listed below) can be used to contain data associated with a work item and to sort and filter your work list. They are available where the WorkManagerFactory object can be accessed (for example, on a user task schedule script). For example, attribute2 can be used to hold a customer name, and attribute1 a customer reference number to aid work list sort and filter choices. The attribute1 work item attribute can be assigned integer values in the range -2,147,483,648 to 2,147,483,647.
workItemAttributes.attribute2: Text	Limited to 64 characters in length.
workItemAttributes.attribute3: Text	See description above.
workItemAttributes.attribute4: Text	

Attribute / Method	Comments
workItemAttributes.attribute5: BigDecimal	See description above. This can be assigned Decimal or BigDecimal values.
workItemAttributes.attribute6: DateTime	See description above.
workItemAttributes.attribute7: DateTime	See description above.
workItemAttributes.attribute8: Text workItemAttributes.attribute9: Text workItemAttributes.attribute10: Text workItemAttributes.attribute11: Text workItemAttributes.attribute12: Text workItemAttributes.attribute13: Text workItemAttributes.attribute14: Text	Limited to a maximum of 20 characters - anything larger will be truncated. See description above.
workItemAttributes.attribute15: Integer	The attribute15 work item attribute can be assigned integer values in the range -2,147,483,648 to 2,147,483,647.
workItemAttributes.attribute16: BigDecimal	See description above. This can be assigned Decimal or BigDecimal values.
workItemAttributes.attribute17: BigDecimal	See description above. This can be assigned Decimal or BigDecimal values.
workItemAttributes.attribute18: BigDecimal	See description above. This can be assigned Decimal or BigDecimal values.
workItemAttributes.attribute19: DateTime	See description above.
workItemAttributes.attribute20: DateTime	See description above.
workItemAttributes.attribute21: Text workItemAttributes.attribute22: Text workItemAttributes.attribute23: Text workItemAttributes.attribute24: Text workItemAttributes.attribute25: Text workItemAttributes.attribute26: Text	Limited to a maximum of 20 characters - anything larger will be truncated. See description above.

Attribute / Method	Comments
workItemAttributes.attribute27: Text workItemAttributes.attribute28: Text workItemAttributes.attribute29: Text workItemAttributes.attribute30: Text workItemAttributes.attribute31: Text workItemAttributes.attribute32: Text workItemAttributes.attribute33: Text workItemAttributes.attribute34: Text workItemAttributes.attribute35: Text workItemAttributes.attribute36: Text workItemAttributes.attribute37: Text workItemAttributes.attribute38: Text	Limited to a maximum of 64 characters - anything larger will be truncated. See description above.
workItemAttributes.attribute39: Text workItemAttributes.attribute40: Text	Limited to a maximum of 255 characters - anything larger will be truncated. See description above.
ItemContext	
getActivityId() : Text	
getActivityName() : Text	
getAppName() : Text	
getAppInstance() : Text	
getAppId() : Text	
getAppInstanceDescription() : Text	
ItemSchedule	
getStartDate() : Datetime	
getTargetDate : Datetime	

When using `getWorkItem()` or `getSchedule()`, note that the following will be shown:

- A null object when there is a null value.
- An empty object, "", when there is a 0 length value.

OrgModel

This table lists the OrgModel attributes and methods provided.

Attribute / Method	Comments
OrgModel	
ouByGuid (guid:Text) : EntityDetail	Returns the single EntityDetail that describes the Organizational Unit identified by its GUID. If no such Organizational Unit exists, the return value will be `null`.
ouByName(name:Text) : List<EntityDetail>	Returns the list of EntityDetails that describe the Organizational Units identified by the given name. If no such named Organizational Units exist, the return value will be an empty list.
groupByGuid(guid:Text) : EntityDetail	Returns the single EntityDetail that describes the Group identified by its GUID. If no such Group exists, the return value will be `null`.
groupByName(name:Text) : List<EntityDetail>	Returns the list of EntityDetails that describe the Groups identified by the given name. If no such named Groups exist, the return value will be an empty list.
resourceByGuid(guid:Text) : EntityDetail	Returns the single EntityDetail that describes the Human Resource identified by its GUID. If no such Human Resource exists, the return value will be `null`.
resourceByName(name:Text) : List<EntityDetail>	Returns the list of EntityDetails that describe the Human Resources identified by the given name. If no such named Human Resources exist, the return value will be an empty list.
resourceByLdapDN(ldapDN:Text): List<EntityDetail>	Returns the collection of Resources identified by the given LDAP DN, or an empty list if none can be found. Ideally, there should be only one such Resource for a given DN.
positionByGuid(guid:Text) : EntityDetail	Returns the single EntityDetail that describes the Position identified by its GUID. If no such Position exists, the return value will be `null`.
positionByName(name:Text) : List<EntityDetail>	Returns the list of EntityDetails that describe the Positions identified by the given name. If no such named Positions exist, the return value will be an empty list.

Attribute / Method	Comments
orgByGuid(guid:Text) : EntityDetail	Returns the single EntityDetail that describes the Organization identified by its GUID. If no such Organization exists, the return value will be `null`.
orgByName(name:Text) : List<EntityDetail>	Returns the list of EntityDetails that describe the Organizations identified by the given name. If no such named Organizations exist, the return value will be an empty list.
EntityDetail	
getEntityType() : Text	Returns the type identifier for this organizational model entity. Example values are: • ORGANIZATION • ORGANIZATIONAL_UNIT • GROUP • POSITION • RESOURCE
getGroups() : List<EntityDetail>	For Human Resource entities, this will return the EntityDetails that describe the Groups to which the Resource is associated.
getPositions() : List<EntityDetail>	For Human Resource entities, this will return the EntityDetails that describe the Positions to which the Resource is associated.
getName() : Text	The name of the organizational model entity.
getGuid() : Text	The GUID that uniquely identifies the organizational model entity.
getAlias() : Text	For Human Resource entities, this is the Alias of the LDAP Source from which the Resource is derived.
getDn() : Text	For Human Resource entities this is the Distinguishing Name (DN) of the LDAP entry from which the Resource is derived.

Attribute / Method	Comments
getResourceType() : Text	For entities of Entity Type RESOURCE, this identifies the type of Resource: • "DURABLE" • "CONSUMABLE" • "HUMAN" Currently, only HUMAN Resources are supported.
getResources() : List<EntityDetail>	For non-Resource entity types (such as Positions and Groups), this will return the Resource entities associated with that entity. For example, for a Position, it will be all the Resources that hold that Position.
getAttributeValue(attrName:Text) :List<Text>	For Resource entity types, this will return the value of the named Resource Attribute held by that Resource entity.
getAttributeType(attrName:Text) : Text	For Resource entity types, this will return the data type of the named Resource Attribute. Possible values are: • string • decimal • integer • boolean • datetime • date • time • enum • enumset

Business Data Services Glossary

A

Aggregation

Aggregation is a specialized form of association. Objects in an aggregation relation have their own lifecycle, but one object is related to the other object with a “has-a” type of relationship, for example, Department-Teacher.

Association

Association is a relationship where all the objects have their own lifecycle and there is no parent. For example: the Teacher-Student relationship. This is the most general of the UML relationships.

Attribute

A property of a Class, for example an Order class, may contain date and orderNumber attributes, amongst others.

The type of attribute can be one of the following:

- Primitive Type (see below)
- Enumerated Type (see below)
- Class Type (see below)

When choosing a type for an attribute, BOM Editor refers to the BOM Native Types as Primitive Types. Primitive Types which are not pre-defined, for example those that are defined within a BOM, have the package name of the BOM against them when selecting the Type to use. This can be useful, for instance, if there is more than one BOM that contains an OrderId class.

B

Basic Type

Process Template field values can be of a Basic Type or an External Reference Type that refers to a BOM Class.

The Basic Types are:

- Text
- Decimal
- Integer
- Boolean
- Date
- Time
- Date Time
- Performer

Note: There is no Datetimetz, Duration, BOM Object, URI, or ID type (nor an Attachment type), and the Performer field is a special type of Text field that contains an RQL query string.

The Basic Types are also known as Process Types.

BDS

Business Data Services

See Business Data Services.

BOM

Business Object Model

See Business Object Model (BOM).

BOM Class

An entity in the BOM that represents a particular part of the application data, for example, an Order or a Customer. A BOM Class is a template for a Business Object.

BOM Native Types

There are 13 predefined primitive types (or 15, if you count the numeric sub-types) that can be used to build other Primitive Types, or used as types of attributes of classes:

- Boolean, String
- Integer (Signed integer and Fixed integer sub-types)
- Decimal (Floating Point and Fixed Point sub-types)
- Date, Time, Datetime, Datetimetz, Duration
- URI, ID, Object, Attachment *

Note that the Attachment type is not currently supported.

Business Data

Structured data that contains information about real-world entities that an organization deals with, for example Customer, Order, and Orderline. Each of these entities will have a number of attributes, for example name, address, and date. These objects will also be related to each other in different relationships and with different multiplicities.

Business Data Application

An application in the ActiveMatrix BPM runtime that is created by deploying a Business Data project. A Business Data application contains the BDS Plug-ins generated from the BOM or BOMs in the Business Data project.)

Business Data Services

A Component of ActiveMatrix BPM that handles all the application data needs of the ActiveMatrix BPM system.

Business Object

An instance of a BOM Class. For example, for a Customer class, there is a Business Object that represents and holds information about a particular customer. Do not confuse this term with Object BOM Native Type.

Business Object Model (BOM)

The Model representing the structure of the application data created using the BOM editor. A BOM that contains only local classes is a local BOM. A BOM that contains at least one case or global class is a global BOM.

C

Composition

Composition is a specialized form of the Aggregation relationship. In this relationship if the parent object is deleted, all the child objects will be deleted too. This is not the case for Aggregation. An example of this type of relationship is School-Classroom. If the School is destroyed, the classrooms will be destroyed too.

E

EMF

Eclipse Modeling Framework.

See:

http://www.eclipse.org/modeling/emf/

Enum or Enumerated Type

A type that can have a restricted set of values. For example, DayOfWeek can have the following values:

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

G

Generalization

The Generalization relationship indicates that one of the two related classes (the subtype) is considered to be a specialized form of the other (the super type), and supertype is considered as a Generalization of the subtype. These types of relationships are characterized by the "is-a" phrase. For example, Oak "is-a" Tree. An Oak is a Specialization of the more general Tree class, and Tree is a generalization of the more specific Oak class.

Global BOM

A BOM that contains at least one case class or global class.

L

Local BOM

A BOM that contains only local classes. (A local BOM cannot contain a global class or a case class.)

P

Primitive Type

In a BOM, it is possible to define a type based on one of the BOM Native Types or another Primitive Type. When these Primitive Types are defined, it is possible to add some limitations using a regular expression or a range.

Data Fields of Primitive Types cannot be used in processes. They must be of BOM Types or Basic Types.

Process Instance

An instance of a flow through a Process Template with data values that reflect the information being processed by this particular instance of the process.

Process Local Data

Data that lives within a Process Instance. The object will either be a Business Object or a Basic Type.

Process Template

The definition of what a process should do.

Process Types

Basic Type.

See Basic Type.

R

RQL

Resource Query Language – a language for selecting which resources can have access to a UserTask in a process. For example:

```
resource(name="Clint Hill")
```

S

Specialization

Generalization.

See Generalization.

U

UML

Unified Modeling Language – an international standard modeling language supported by many tools. Identifies different types of relationships that can exist between the objects being modelled, for example: composition, generalization/specialization, association, and aggregation.

UserTask

A UserTask is a step in a process that is handled by a user and requires the user to complete a form. On completing the form, the user submits the values causing the changes to the field values to be saved. A user may Close a form so that they can complete it later.

W

WSDL

WSDL stands for Web Service Definition Language. A file with a .wsdl extension contains the definition of a Web Service, defining the format of the request and responses. A WSDL file can be imported by TIBCO Business Studio to make it easy to call web services.

X

XSD

XSD stands for XML Schema Definition. A file with a .xsd extension contains XML that defines the format that some other XML should take. This other XML is used to pass data between processes.