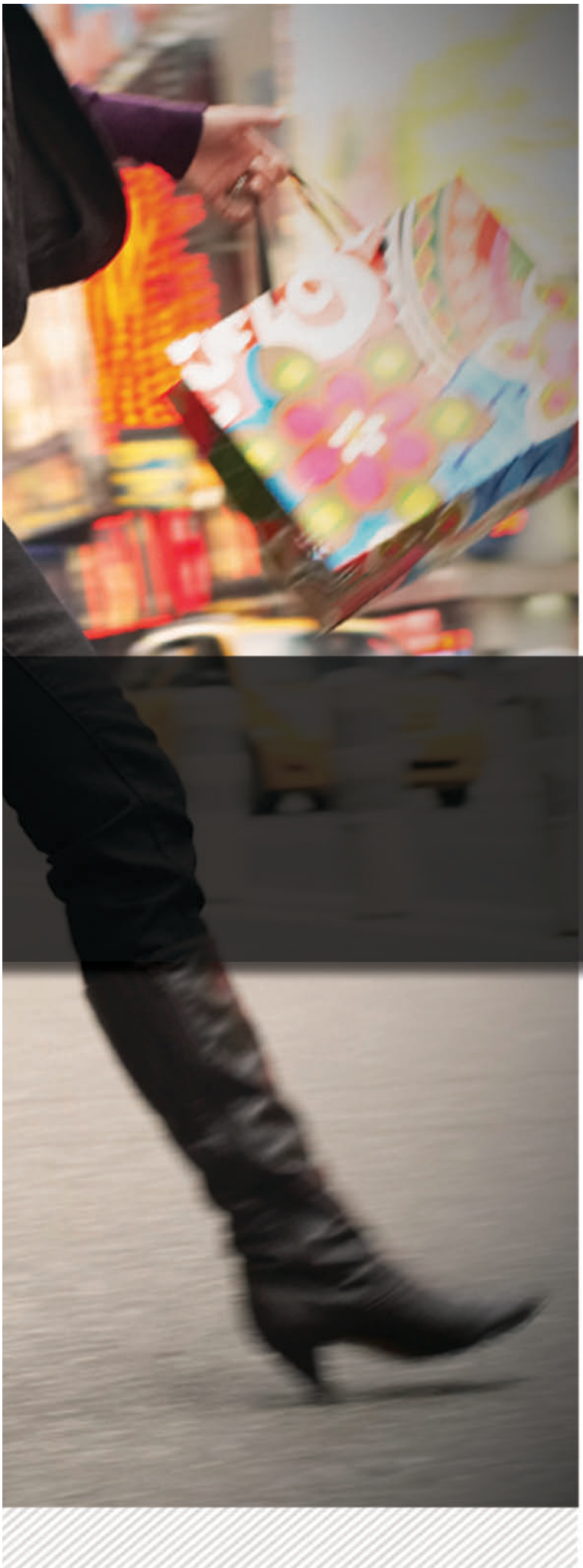


demandware
move faster, grow faster

WORKING WITH THE DEMANDWARE APIs

Student Guide

demandware move faster, grow faster.

Table of Contents

demandware move faster, grow faster.

Introduction

About the Course

Audience	Developers
Duration	8 hours
Prerequisites	To successfully meet the objectives of this course, we suggest that developers complete the *Developing in Demandware* course.
System Requirements	Laptop or desktop with UX Studio installed and access to a sandbox.
Course Materials	`Working with the DW APIs Exercises.zip` file—Contains the code examples for the exercises

Course Objectives

Welcome to *Working with the Demandware APIs*. After completing this course, you will be able to:

- Describe the key components of the Demandware programming environment and how the APIs are used within the environment.
- Apply the APIs in the `dw.catalog` package to implement new catalog, category, product, and price book functionality.
- Apply the APIs in the `dw.campaign` package to implement new promotion and campaign functionality.
- Apply the APIs in the `dw.order` package to implement new cart, order and line item functionality.

Module Objectives

The following table describes the objectives for each module:

Module	Objectives
APIs and the Demandware Programming Environment	▪ Describe the key components of the Demandware programming environment and how the APIs are used within the environment. ▪ Identify the Demandware Script API packages and describe the general capabilities of each. ▪ Identify the Demandware Pipelet API groups provided for Demandware platform objects and describe the general capabilities of each.
Working with the `dw.catalog` Package	▪ Describe the `Category`, `CategoryMgr`, `Product`, and `ProductPriceModel` classes. ▪ Apply the APIs in the `dw.catalog` package to implement new catalog, product, and price book functionality.
Working with the `dw.campaign` Package	▪ Describe the methods and objects of the `Promotion`, `Campaign`, `PromotionPlan`, and `PromotionMgr` classes. ▪ Apply the APIs in the `dw.campaign` package to implement new promotion and campaign functionality.
Working with the `dw.order` Package	▪ Describe the methods and objects of the `LineItem`, `LineItemCtnr`, and `Order` classes. ▪ Apply the APIs in the `dw.order` package to implement new order and line item functionality.

About the Course Exercises

This course includes scripting exercises which require that you add missing API methods to implement functionality. You'll be given scripts and templates containing comments that show where the missing API methods should be inserted, for example:

```
if( /* 1.Check if product is a master or not */ )
{
    var varModel : dw.catalog.ProductVariationModel = /* 2.Get variation model of the product */;
    var varAttrsColor : dw.catalog.ProductVariationAttribute = /* 3.Get all values for attributes called 'color' */;
     args.selectableColors= new ArrayList();
      if( varAttrsColor != null )
      {
          var allColors : dw.util.Collection = varModel.getAllValues( varAttrsColor );
```

You'll inspect the API documentation (https://info.demandware.com) to determine which methods to use to fill in the blanks. Then, you'll insert them in the scripts and templates, and test the pipelines.

Next, you'll fill in the blanks:

Script Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to check if the product is a master or not:

   _____

2. Code used to get variation model of the product:

   _____

3. Code used to get all values for attributes called 'color':

   _____

The exercise files are included in the `Working with DW APIs Exercises.zip` file provided by your instructor. Within the zip file, the `Exercise Files` folder contains the script and template code you'll need to complete. The `Solution Files` folder contains the completed scripts and templates.

You'll verify your scripts and templates in UX Studio by building pipelines that invoke them.

Module 1: APIs and the Demandware Programming Environment

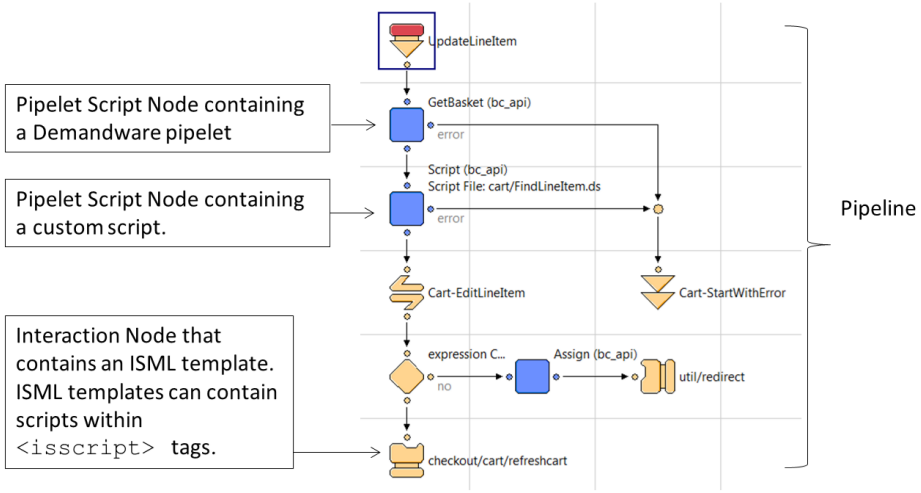
Learning Objectives

After completing this module, you will be able to:

- Describe the key components of the Demandware programming environment and how the APIs are used within them.
- Identify the Demandware Script API packages and describe the general capabilities of each.
- Identify the Demandware Pipelet API groups provided for Demandware platform objects and describe the general capabilities of each.

Lesson 1.1: Review of the Demandware Programming Environment

This module reviews the Demandware programming environment. In the *Developing in Demandware* course, you learned about the following components that invoke the APIs in the Demandware development environment.



This example shows a Demandware pipeline. Among other types of nodes, pipelines contain these node types that support scripting:

- **Pipelet Script Nodes** contain the system pipelets provided by Demandware and can contain custom scripts that you develop, as well.
- **Interaction Nodes** contain Internet Store Markup Language (ISML) templates. ISML markup tags control how dynamic data is embedded and formatted on the page. ISML supports an `<isscript>` tag for including custom scripts within ISML templates.

The Model-View-Controller Pattern

The Demandware programming environment supports the Model-View-Controller (MVC) pattern.

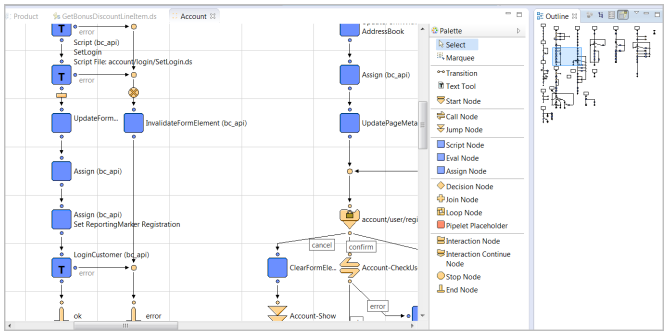
In the MVC pattern:

- ISML templates generate the views.
- Pipelines are the controllers, managing the business logic flow.
- Pipelets implement models by accessing Demandware Commerce data.

Although ISML supports scripts through the use of the `<isscript>` tag, it's best to follow the MVC pattern and use ISML for displaying data rather than implementing business logic. Implementing your pipelines in this way supports readability and makes it easier to maintain your code.

Pipelines

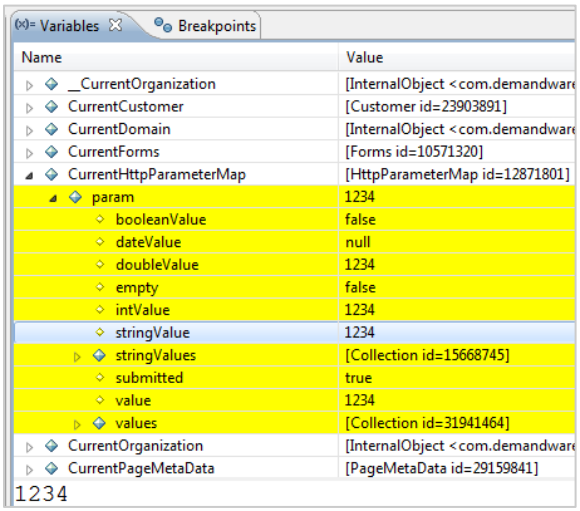
The Demandware platform uses pipelines to encapsulate storefront business processes. As a developer, you can use the Demandware-provided pipelines and you can create your own pipelines to implement custom storefront business processes. The pipelines are represented as flowcharts containing pipeline elements that model the business logic, such as start nodes for initiating a pipeline, interaction nodes for displaying request results, and script nodes for executing Demandware Script (DWScript) code.



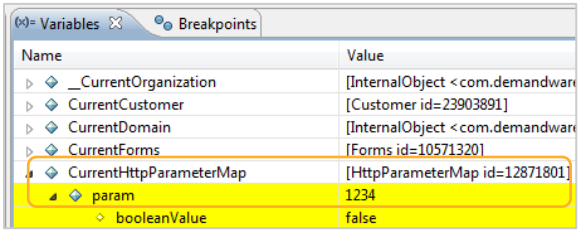
The Pipeline Dictionary

When a pipeline is executed, the input and output data for the pipeline are stored on the pipeline dictionary (`pdict`) as a hashmap (key/value pairs). The pipeline dictionary persists as long as the pipeline executes.

When you run a pipeline, you can use the pipeline debugger to see the pipeline values:



An important key in the pipeline dictionary (`pdict`) is the `CurrentHttpParameterMap` key.



The `CurrentHttpParameterMap` key contains the parameters that can be passed via URL query strings, for example:

```
http://student1.training.dw.demandware.net/on/demandware.store/Sites-SiteGenesis-Site/default/Call-Start?param=1234
```

The values on the `pdict` are available to the current pipeline, as well as all sub-pipelines invoked using **Call Nodes** or **Jump Nodes**.

Demandware Script (DWScript)

Demandware Script (DWScript) is the JavaScript-based server-side language used for coding in the Demandware platform. You use DWScript code anytime you need to access data, such as products, catalogs, and prices. You can include DWScript in:

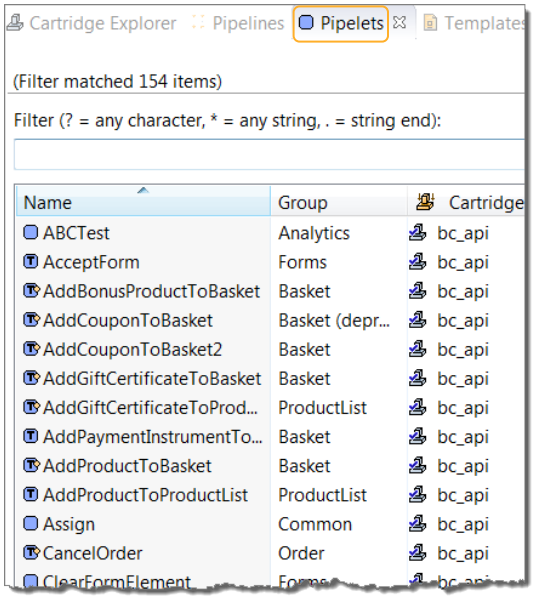
- Decision Nodes in pipelines
- Interaction Nodes ISML templates within `<isscript>` tags or expressions
- Script pipelets

Scripts are located in the `cartridge/scripts` directory and have a `.ds` extension. When you use a script from a different cartridge in a script pipelet, the script must be fully qualified: `cartridge:filename.ds`.

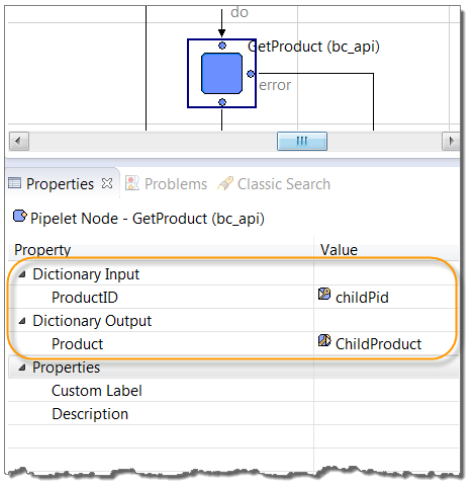
Demandware provides the Demandware Script APIs which provide classes and methods for all Demandware platform objects.

Pipelets

Pipelets are the building blocks that implement storefront business logic in pipelines. Demandware provides pre-coded pipelets (in the `bc_api` cartridge) which implement common functionality to manage storefront objects, such as products, baskets, orders, and customers.



When you use a pipelet in a pipeline, you configure it in the **Properties** tab by setting the input and output values to pipeline dictionary variables.

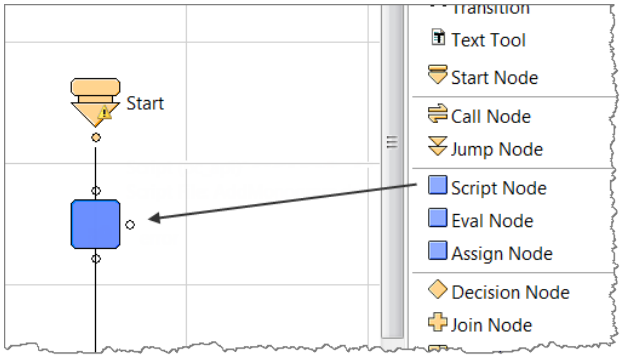


There are three types of pipelet nodes you can add to pipelines. You use:

- **Script Nodes** to invoke a script file – either one of the Demandware-provided pipelets or your own custom pipelets.
- **Eval Nodes** to invoke the Eval pipelet which evaluates data in the pipeline dictionary (`pdict`).
- **Assign Nodes** to assign values to specific keys on the pipeline dictionary.

Custom Pipelets

In addition to the Demandware pipelets, you can create custom pipelets using Demandware Script (DWScript) to implement business logic in pipelines. To create a custom pipelet, you add a **Script Node** in UX Studio.



You can assign an existing script to the **Script Node** or enter a name to create a new script. Demandware provides a template for the script file which you modify to create your script:

```
1 /**
2 * Demandware Script File
3 * To define input and output parameters, create entries of the form:
4 *
5 * @<paramUsageType> <paramName> : <paramDataType> [<paramComment>]
6 *
7 * where
8 *   <paramUsageType> can be either 'input' or 'output'
9 *   <paramName> can be any valid parameter name
10 *   <paramDataType> identifies the type of the parameter
11 *   <paramComment> is an optional comment
12 *
13 * For example:
14 *
15 *-   @input ExampleIn : String This is a sample comment.
16 *-   @output ExampleOut : Number
17 *
18 */
19 importPackage( dw.system );
20
21 function execute( args : PipelineDictionary ) : Number
22 {
23
24     // read pipeline dictionary input parameter
25     // ... = args.ExampleIn;
26
27     // insert business logic here
28
29     // write pipeline dictionary output parameter
30
31     // args.ExampleOut = ...
32
33    return PIPELET_NEXT;
```

You'll set the input and output parameters by removing the comments on their lines (line 15 and 16 in this example) and adding your own functions below.

The Demandware APIs

The following lessons cover the Demandware APIs:

- The Demandware pipelets that you insert directly in pipelines
- The Demandware Script APIs that you use in ISML scripts and in custom pipelets

Lesson 1.2: The Demandware Pipelets

The Demandware pre-coded pipelets implement business logic for Demandware objects. Each group of pipelets provides business logic for particular object classes. The pipelet groups are described in the following table:

Pipelet Group	Overview
Analytics	Pipelets for running analytics, such as A/B testing.
Basket	Pipelets for managing the basket, including pipelets to: ▪ Get baskets ▪ Add products to baskets ▪ Apply gift certificate, coupons, and payment instruments to a basket ▪ Update order line items ▪ Start the checkout process and verify payment cards ▪ Manage shipments and addresses
Catalog	Pipelets for managing the catalogs, categories, and products, including pipelets to: ▪ Access a product or category given an ID ▪ Get the set of products visited in the current session ▪ Set a price book to be considered for price lookup ▪ Process product option and variation selections ▪ List brands for all products in the master catalog

Common	Pipelets that handle assignments, evaluations, paging, sending mail, and updating page metadata.
Content	Provides the `GetContent` pipelet which retrieves content given an ID.
Customer	Pipelets for managing customers, including pipelets to: ▪ Create, retrieve, and remove customer records and related data including addresses and payment instruments ▪ Manage authentication of customers and agent users (who log in on behalf of customers)
CustomObject	Pipelets for creating, removing, and searching for custom objects.
Cybersource	Pipelets for managing Cybersource requests.
Forms	Pipelets for managing forms, including pipelets to update, accept, clear, and invalidate forms.
ImpEx	Pipelets for importing and exporting Demandware Commerce data.
Job	Provides the `RunJobNow` pipelet used to initiate jobs.
Order	Pipelets for managing orders, including pipelets to: ▪ Create, place, retrieve, and cancel orders ▪ Create, retrieve, and redeem gift certificates ▪ Reserve inventory and export to Order Center
Payment	Provides the `GetPaymentProcessor` pipelet which retrieves a payment processor given an ID.
ProductList	Pipelets to manage product lists including pipelets to: ▪ Create, retrieve, remove product lists ▪ Add and remove products from the product list, including gift certificates
Scripting	Provides the `Script` pipelet that executes a Demandware script.
Search	Pipelets that execute product, content, and system object searches, as well as pipelets to update search indexes and manage search suggestions and URL redirects.

SiteMap	Provides the `SendGoogleSiteMap` pipelet to serve Google site requests.
SourceCode	Pipelets to apply a source code to the current session and to handle source code redirects.
Store	Provides the `GetNearestStores` pipelet which returns the list of stores within a pre-configured distance of a given location.
Util	Pipelets to invalidate the cache, perform URL redirects, set the locale for a request, and set the currency for a session.
VeriSign	Provides the `VSAuthorizeCreditCard` pipelet which performs credit card authorization via the VeriSign payment service.

For More Information

To learn about the individual pipelets, see the Demandware online help: `https://info.demandware.com` and select **Demandware API > Demandware Pipelets**. The online help provides a description of each pipelet and its input and output parameters.

You can also access the documentation in UX Studio by clicking **Help > Help Contents**.

Lesson 1.3: The Script APIs

The Demandware Script APIs are organized into packages that include classes and methods for Demandware Commerce objects. Some packages have "models" which are constructs for grouping the methods for Demandware Commerce objects.

The objects and methods in all packages besides `TopLevel` need to be imported or fully qualified.

The Script API packages are described in the following table:

Package	Overview
`TopLevel`	General purpose classes such as global, argument, arrays, Booleans, and dates. The `TopLevel` package is available implicitly to all scripts and does not have to be fully qualified when its objects and methods are used.

`dw.campaign`	Classes for managing campaigns, promotions, and A/B tests. This class is covered in the "Working with the `dw.campaign` Package" module.
`dw.catalog`	Classes for managing catalogs, categories, products, prices, stores, and search. This class is covered in the "Working with the `dw.catalog` Package" module.
`dw.content`	Classes for managing content assets, library folders, and the content library of the current site, including content searches.
`dw.crypto`	Classes for managing encryption.
`dw.customer`	Classes for managing customer data including customer profiles, credentials, customer groups, order history, product lists, and payment instruments.
`dw.io`	Classes for input/output supporting XML streams, CSV files, text files, and random access files.
`dw.net`	Classes for handling FTP, HTTP, SFTP, WebDAV, and email operations.
`dw.object`	Classes for managing system objects, active data, custom objects, and their attributes in the Demandware framework.
`dw.ocapi.data.content`	Script hooks that can be registered to customize Open Commerce API (OCAPI) data resources. OCAPI provides REST-based web services to integrate storefront data into applications.
`dw.ocapi.data.customer`	Script hooks that can be registered to customize Open Commerce API (OCAPI) customer resources.
`dw.ocapi.shop.account`	Script hooks that can be registered to customize Open Commerce API (OCAPI) account resources.
`dw.ocapi.shop.basket`	Script hooks that can be registered to customize Open Commerce API (OCAPI) basket resources.
`dw.oms`	Methods for managing business objects from the Demandware Order Management System.
`dw.order`	Classes for managing shopping carts (baskets), orders, product and shipping line items, and payment processors. This class is covered in the "Working with the `dw.order` Package" module.
`dw.order.hooks`	Classes that provide hooks that can be registered to customize

	workflows, for example, order center functionality and shipping order lifecycles.
`dw.rpc`	Classes for managing web services. Use this class if your web service WSDL uses RPC-style calls. Use the `dw.wc` for web services otherwise.
`dw.suggest`	Classes for managing suggestions such as brand, product, category, content, and phrase suggestions.
`dw.system`	Classes that manage: ▪ Hooks ▪ Internal objects ▪ Jobs ▪ Logs ▪ Pipeline dictionaries ▪ Requests ▪ Responses ▪ Sessions ▪ Sites ▪ Site preferences ▪ Status ▪ System
`dw.util`	Utility methods for managing objects such as arrays, calendars, collections, currency, dates, hash maps and sets, lists, locales, maps, and sets, templates, and UUIDs, among others.
`dw.value`	Classes for representing money, quantity, mime-encoded text, and enumeration types.
`dw.web`	Classes for managing web sessions and pages. The `dw.web` package is available implicitly to all scripts and does not have to be fully qualified when its objects and methods are used.
`dw.ws`	Classes for managing web services. If your web service WSDL uses RPC-style calls, use the `dw.rpc` class instead.

Static versus Dynamic Methods

Examples in this course use both static and dynamic methods.

- When you declare a static method, you precede the method with a class name, for example: `<className>.<methodName>()`
- When you declare a dynamic method, you must create an instance first, then precede the method with the instance name, for example: `<instanceName>.<methodName>()`

This course doesn't differentiate between static and dynamic methods. To determine whether a method is static or dynamic, refer to the Demandware online help.

For More Information

To learn about the individual pipelets, see the Demandware online help: https://info.demandware.com and select **Demandware API > Demandware Script**. The online help provides a description of each pipelet and its input and output parameters.

You can also access the documentation in UX Studio by clicking **Help > Help Contents**.

Knowledge Check

1. You can use Demandware Script in which of these constructs?

	Decision nodes in pipelines
	ISML templates
	Transitions in pipelines
	Script nodes in pipelines

2. To create a custom pipelet, you:
   a. Create an ISML template and create the custom pipelet within `<ispipelet>` tags.
   b. Add a Script node to a pipeline.
   c. Add an Assign node to a pipeline.
   d. Clone a Demandware pipelet and update the code.
   e. b and c

Knowledge Check Answers

1. You can use Demandware Script in which of these constructs?

✓	Decision nodes in pipelines
✓	ISML templates
	Transitions in pipelines
✓	Script nodes in pipelines

2. To create a custom pipelet, you:

   a. Create an ISML template and create the custom pipelet within `<ispipelet>` tags.
   b. Add a Script node to a pipeline.
   c. Add an Assign node to a pipeline.
   d. Clone a Demandware pipelet and update the code.
   e. b and c

   Answer: b

Module 2: Working with the `dw.catalog` Package

Learning Objectives

At the end of this module, you will be able to:

- Describe the `Category`, `CategoryMgr`, `Product`, and `ProductPriceModel` classes.
- Apply the APIs in the `dw.catalog` package to implement new catalog, product, and price book functionality.

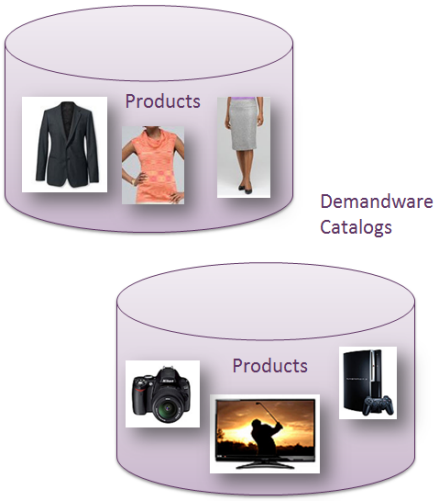
Lesson 2.1: Understanding the `Catalog` Class

Use the `Catalog` package to manage products—including their catalogs, categories, and prices.

- The `Catalog` class provides methods for accessing the name, ID, description, and root of the catalog.
- The `CatalogMgr` class provides methods for accessing the catalogs, categories, and sorting information.

Catalog and Category Data Objects

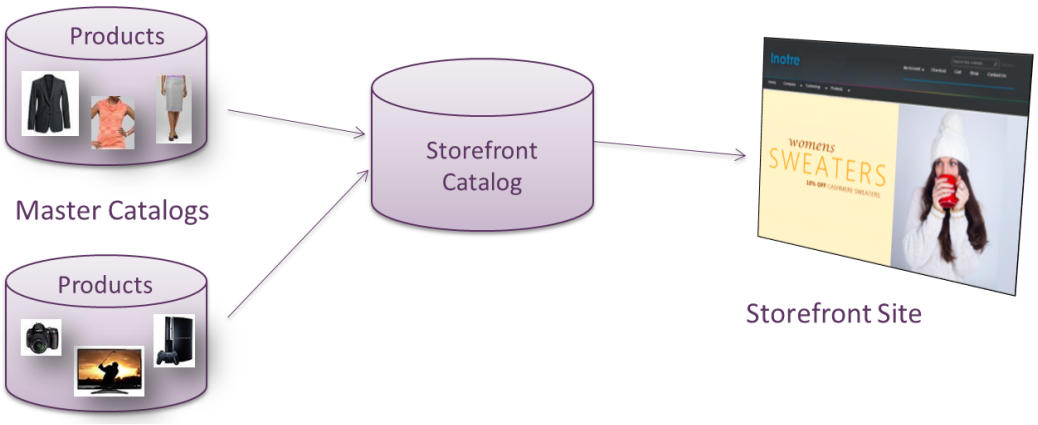
Catalogs are containers of products and other product-related information:



Every product in the system is “owned by” a master catalog, although products can be assigned to other catalogs. To access catalogs, use the static method `CatalogMgr.getCatalog`.

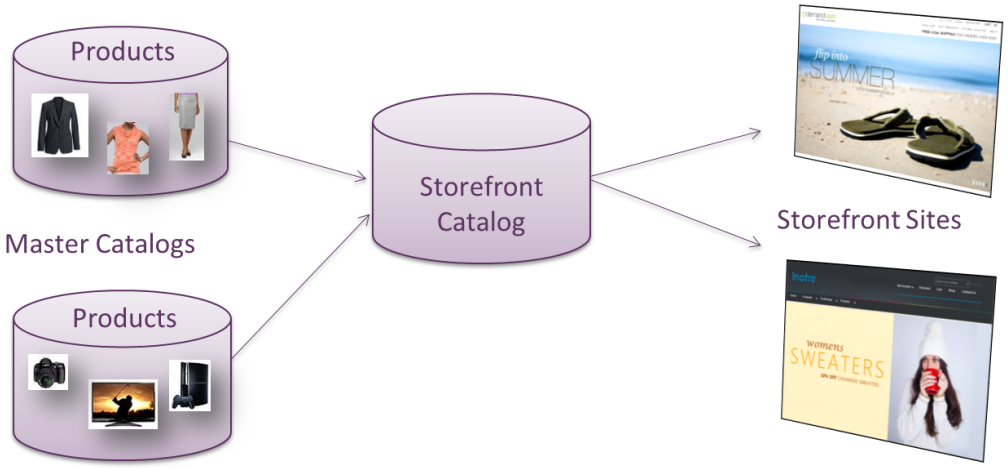
Each site is assigned:

- One or more master catalogs which contain the actual products.
- A single storefront catalog (also called a site catalog). The storefront catalog doesn't actually contain the products—it's a navigation catalog with categories that include only products available online.



Use the static method `CatalogMgr.getSiteCatalog()` method to return the storefront (site) catalog for the current storefront site.

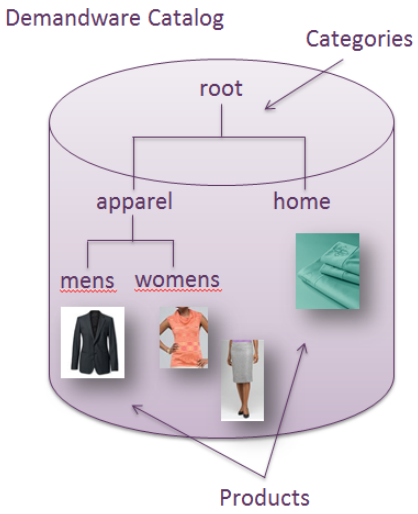
Merchants assign catalogs to storefront sites. They can assign catalogs to multiple sites and they can assign multiple catalogs to a site. Each site also has a single storefront catalog which contains only the products offered online.



As you develop and test scripts and templates, you might need to view the catalog and category data for the storefront site you're working with. To view the catalogs for a site using Business Manager, click **Site > Products and Catalogs > Catalogs**.

Managing Categories

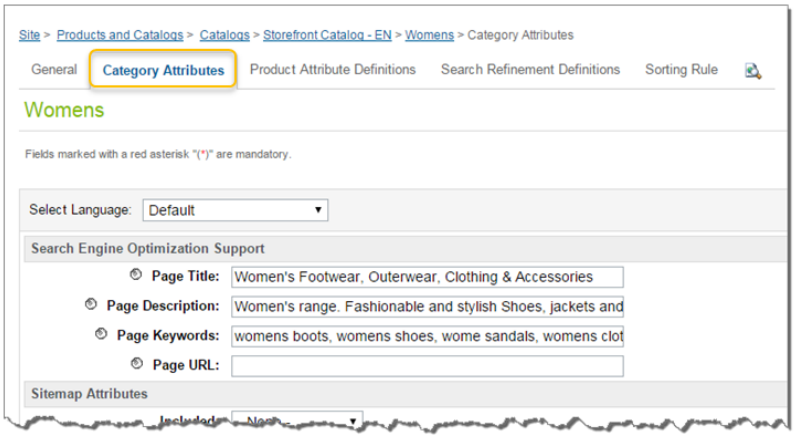
Catalogs contain a tree of categories with a single top-level root category. You use the `CatalogMgr.getCategory` method to return the categories of a catalog.



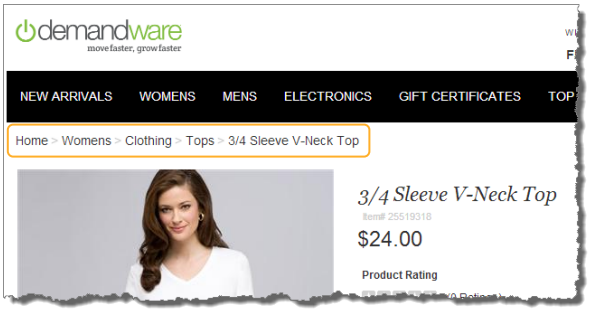
Merchants assign products to categories within the product's parent catalog or other catalogs. Merchants can assign a product to multiple categories within the same catalog. Products not assigned to categories are considered "uncategorized."

A product has both a classification category and a primary category.

- **Classification categories** "own" the products and define the attribute set of each product. The classification category of the product can be defined in any catalog, but it's typically defined in the product's parent category. A product can have one classification category only.



- **Primary categories** define the category that appears in the breadcrumbs when a product is accessed via search.



Master Catalogs vs. Storefront Catalogs

Organizations typically set up:

- **Master catalogs** that contain product information and details. Master catalogs are often generated via a data feed from a Product Information Management (PIM) system.
- **Storefront (site) catalogs** that define the category structure of the storefront site. They provide the assignments of products to categories. You can determine the categories to which a product is assigned using the `Product.getCategories` method. To determine the primary category for a product within the current storefront catalog, use the `Product.getPrimaryCategory` method.

Lesson 2.2: Understanding the `Product` Class

The `Product` class represents the items customers can purchase on the site. Each product has a product ID.

Product Data Objects

Products can be simple products or one of the following:

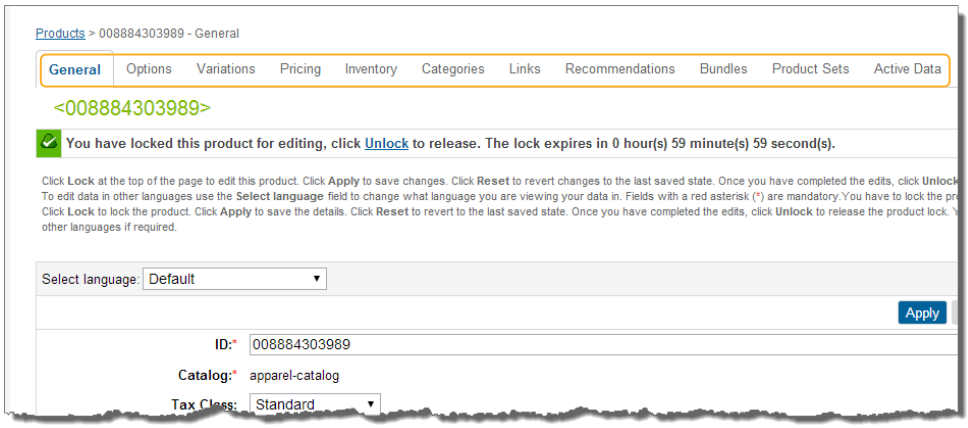
- **Master Products** are essentially templates for related products that differ by a set of defined variation attributes. If your site is based on SiteGenesis, by default customers can't order the master product, only variants. You can customize this behavior using the API.
- **Variants** represent the variations of a master product—the orderable products. Use `ProductVariationModel` to manage the variations of the master product.
- **Option Products** define additional options that are ordered separately from the product, such as warranties. Use `ProductOptionModel` to access the option information for an option product.
- **Product Sets** are sets of products the merchant sells as a collection on the storefront site. The product set is not an entity that customers can order—each product in the set has an individual price. Members of the set are called "product-set-products."
- **Product Bundles** are sets of products the merchant sells as a collection, but unlike product sets, the product bundle is ordered as a single unit.

To access products on the storefront site, in Business Manager, click **Site > Products and Catalogs > Products**.

Each product record lists the product's ID, its master catalog, the type of product (such as variation master, variation product, part of a product set, option product, and product set) and other details:

Select All	ID	Name	Catalog	Color	Refinement Color	Type	Status	View
	008885004847	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: White / Size: 12]	Apparel Master Catalog	JJI15XX	White	Variation Product		
	008885004854	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: White / Size: 14]	Apparel Master Catalog	JJI15XX	White	Variation Product		
	008885004861	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: White / Size: 16]	Apparel Master Catalog	JJI15XX	White	Variation Product		
	008885004878	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: White / Size: 4]	Apparel Master Catalog	JJI15XX	White	Variation Product		
	008885004885	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: White / Size: 6]	Apparel Master Catalog	JJI15XX	White	Variation Product		
	008885004892	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: White / Size: 8]	Apparel Master Catalog	JJI15XX	White	Variation Product		
	008885005141	No-Iron Platinum Easy Care Sleeveless Fitted Shirt [Color: Light Blue / Size: 10]	Apparel Master Catalog	JJA70XX	Blue	Variation Product		
	008885005158	No-Iron Platinum Easy Care Sleeveless Fitted Shirt	Apparel Master Catalog	JJA70XX	Blue	Variation Product		

If you select an item, you can further investigate its variations, categories, options, and other values:



Knowledge Check

The Knowledge Check answer key is on the following page.

1. Classification categories define:
    a. The classes and subclasses of the site refinements
    b. The categories that appear in the breadcrumbs when a product is accessed via search
    c. The attribute set of the product
    d. The catalog hierarchy for multiple sites
    e. a and b

Knowledge Check Answers

1. Classification categories:
   a. Define the classes and subclasses of the site refinements
   b. Define the categories that appear in the breadcrumbs when a product is accessed via search
   c. Define the attribute set of the product
   d. Define the catalog hierarchy for multiple sites
   e. a and b

   Answer: c.

Lesson 2.3: Understanding the `ProductVariationModel` Class

The `ProductVariationModel` class, from the `dw.catalog` package, represents the variation information for a master product.

Product detail pages display product variation attributes, in this case color and size:

For the ¾ Sleeve V-Neck Top, the following methods return the variation information described in the "Values Returned" column.

Method	Description	Values Returned in this Example
`getProductVariationAttributes()`	Returns the variation attributes of the master product	color, size
`getAllValues(ProductVariationAttribute)`	Returns the variation attribute values	For variation attribute "color," returns Black, Butter, Icy Mint, Grey Heather, and White For variation attribute "size," returns XS, S, M, L, XL
`getVariants()`	Returns all variant products, enumerated by attribute values	XS Black, XS Butter, XS Icy Mint, … S Black, S Butter, S Icy Mint, … M Black, M Butter, M Icy Mint, …

`getVariationValue(Variant, ProductVariationAttribute)`	Returns the variation attribute values of the variants	For the variant, Medium/Grey Heather, this method returns Grey Heather for the color attribute.

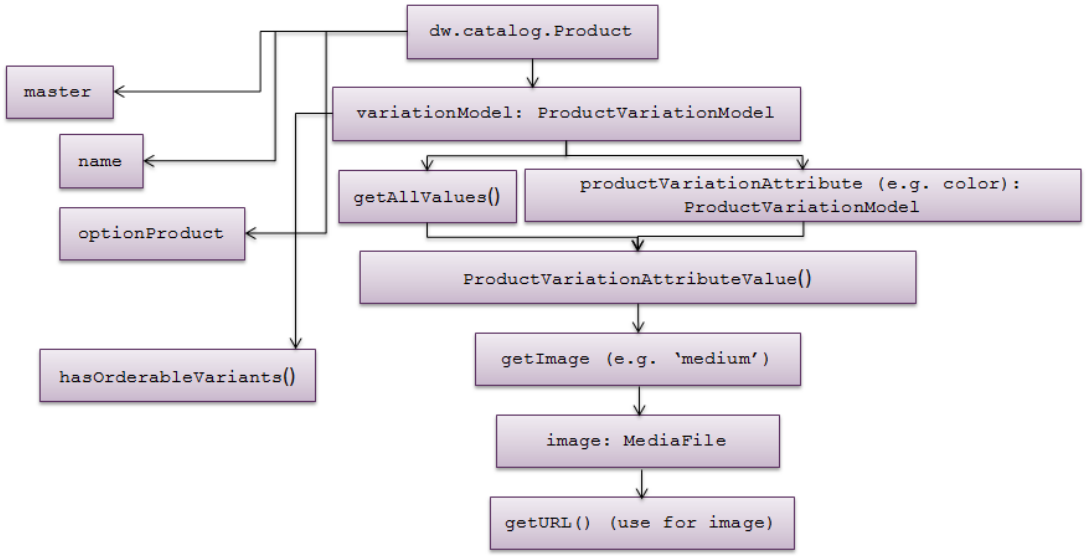
`ProductVariationModel` considers only variants that are:

- Complete – the variant has a value for each variation attribute
- Currently online

Methods that return variants will not return a value if the variant is incomplete or offline. Likewise, the attributes of incomplete or offline variants are not considered.

Exercise: Explore Product Variations

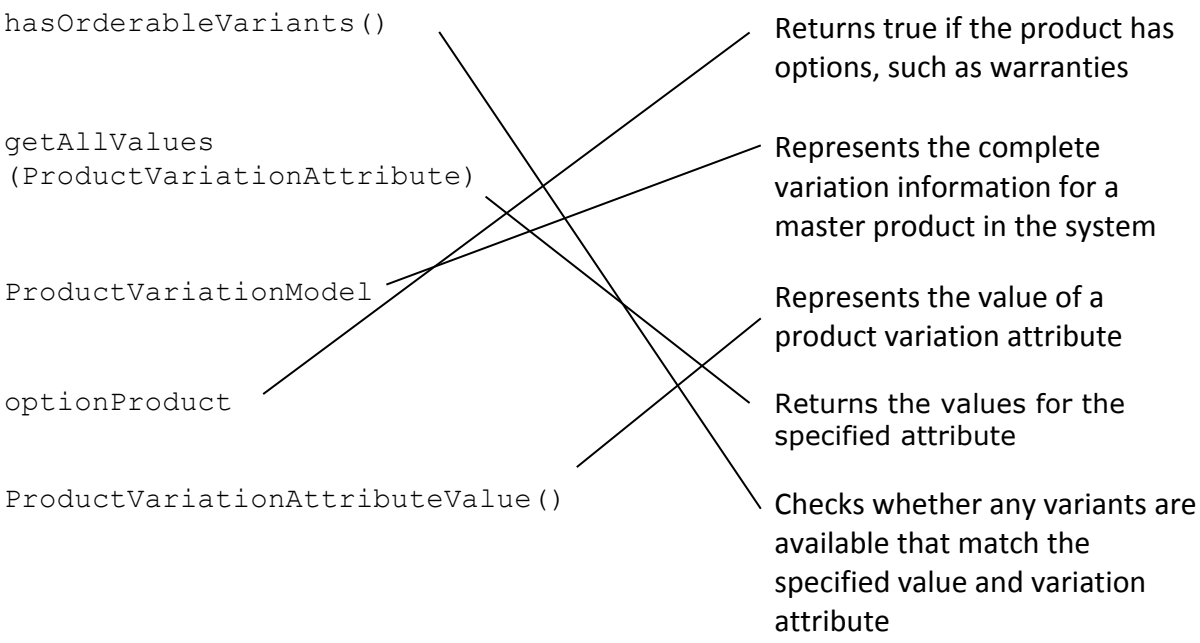
Using the following graph as your guide, click through and review the API documentation (https://info.demandware.com). These are the methods and objects you'll use in the exercises that follow.



After reviewing the documentation for the methods and objects in the preceding graph, match the API elements on the left with the definitions on the right. (Answers on following page)

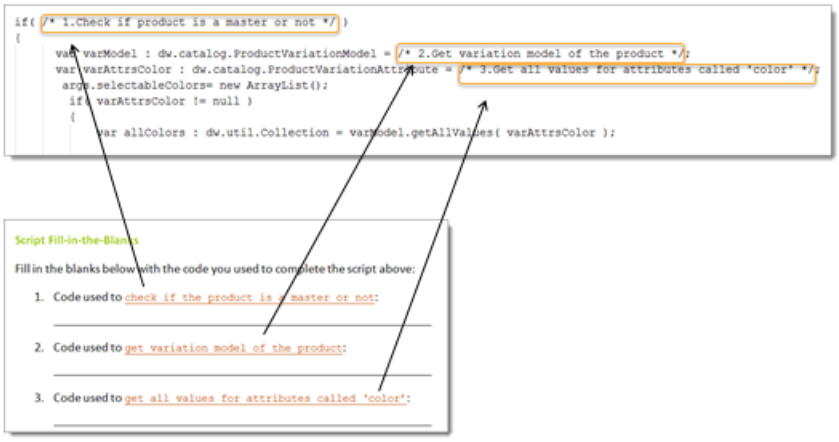
`hasOrderableVariants()`	Returns true if the product has options, such as warranties
`getAllValues(ProductVariationAttribute)`	Represents the complete variation information for a master product in the system
`ProductVariationModel`	Represents the value of a product variation attribute
`optionProduct`	Returns the values for the specified attribute
`ProductVariationAttributeValue()`	Checks whether any variants are available that match the specified value and variation attribute

Exercise Answers: Explore the `Product` Class



Exercise: Product Variations Fill-in-the-Blanks

In this exercise, you'll build a pipeline (`ShowColorVariation`) that checks whether a given product is a master and returns the first color variation. You'll replace the comments in the script and template with the missing API objects and methods by consulting the `dw.catalog` API documentation, for example:



You'll then fill in the blanks in your student guide.

You're provided with an incomplete script, `product.ds`, and a corresponding incomplete ISML template, `product.isml`. (Both are included in the `Working with DW APIs Exercises.zip` file).

- The `product.ds` script checks whether the product is a master and returns the first color variation.
- The `product.isml` template displays the product name, the image of the first variation, and a link to the product display page.

Following are descriptions of the APIs you'll use to fill in the blanks.

Script Fill-in-the-Blanks

Fill in the blanks with the APIs needed to complete the `product.ds` script you've been given:

1. Code used to `check if the product is a master or not`:

   _____

2. Code used to `get variation model of the product`:

   _____

3. Code used to `get all values for attributes called 'color'`:

   _____

4. Code used to `check if present color is orderable attribute value`:

   _____

Complete the Script: `product.ds`

The script checks whether the product is a master product and returns the first color variation.

```
/**
*   @input ProductID : String
*   @output Product : Object
*   @output selectableColors : dw.util.ArrayList
*/
importPackage( dw.system );
importPackage( dw.catalog );
importPackage( dw.web );
importPackage(dw.util);

function execute( args : PipelineDictionary ) : Number
{
    args.Product = ProductMgr.getProduct(args.ProductID);

    if (args.Product == null)
    {
       trace(Resource.msgf('productnotfoundwithid.message', 'product', null, args.ProductID));
        return PIPELET_ERROR;
    }
```

```
    if( /* 1.Check if product is a master or not */ )
    {
     var varModel : dw.catalog.ProductVariationModel =
    /* 2.Get variation model of the product */;
     var varAttrsColor : dw.catalog.ProductVariationAttribute =
    /* 3.Get all values for attributes called 'color' */;
    args.selectableColors= new ArrayList();
    if( varAttrsColor != null )
        {
         var allColors : dw.util.Collection = varModel.getAllValues( varAttrsColor );
         for each( var color in allColors )
         {
          if( /* 4.Check if present color is orderable attribute value */ )
               {
               args.selectableColors.add( color );
               }
          }
         }
    }

     return PIPELET_NEXT;

}
```

Template Fill-in-the-Blanks

Fill in the blanks with the APIs you needed to complete the `product.isml` template you've been given:

1. Code used to `get product Id`:

   _____

2. Code used to `get variation attribute for color`:

   _____

3. Code used to `get the URL for selecting variation value`:

   _____

4. Code used to `get URL for medium image from firstColorVariation`:

   _____

5. Code used to `get medium image from product itself`:

   _____

6. Code used to `get product name`:

   _____

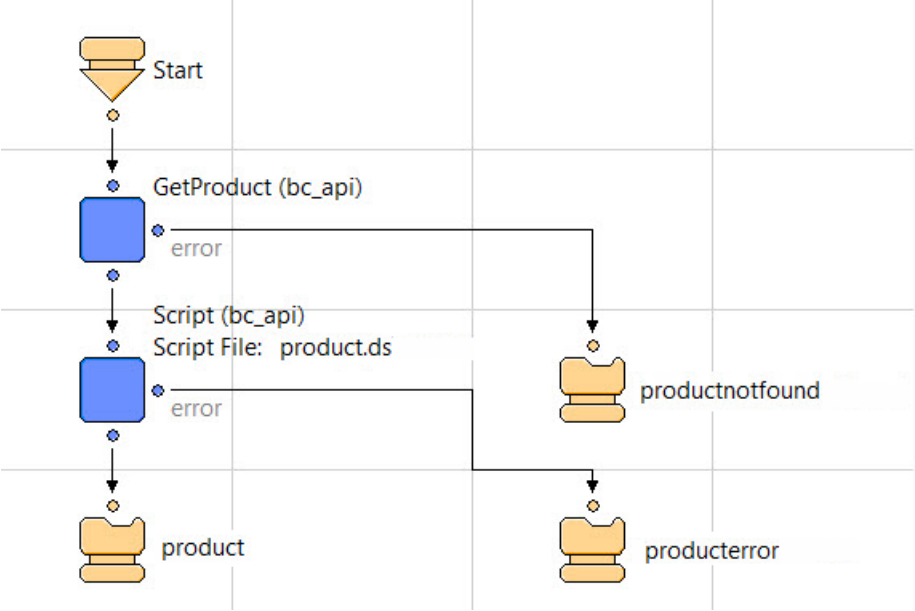
Complete the Template: `product.isml`

The template displays the image of the first variation and a link to the product display page.

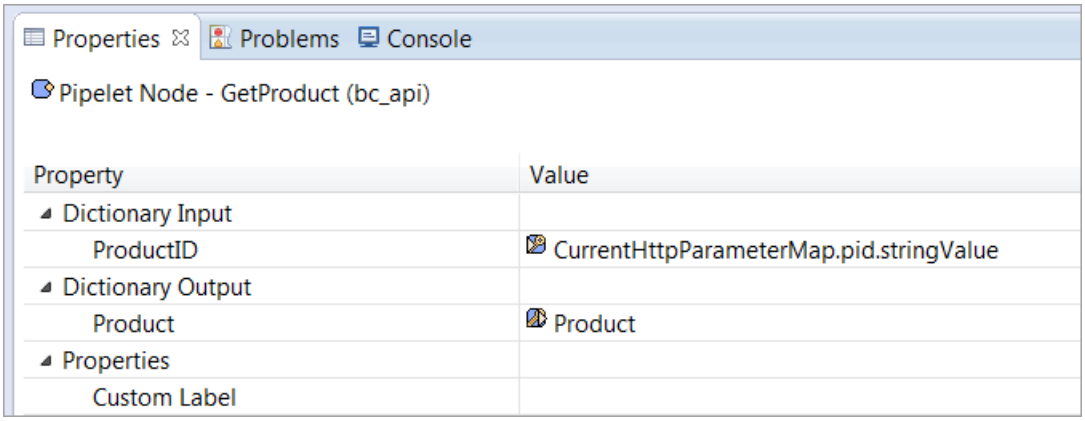
```
<!--- TEMPLATENAME: product.isml --->

<isset name="Product" value="${pdict.Product}" scope="page"/>
<isset name="selectableColors" value="${pdict.selectableColors}" scope="page"/>

<isset name="productUrl" value="${URLUtils.http('Product-Show', 'pid',
<!-- 1.Get product ID --> )}" scope="page"/>

<isif condition="${!empty(selectableColors) && selectableColors.size() > 0}">
<isset name="colorVarAttr" value="${Product.variationModel.
<!-- 2.Get variation attribute for color -->}" scope="page"/>

<isset name="productUrl" value="${Product.variationModel.
<!-- 3. Get URL for selecting variation value -->('Product-Show', colorVarAttr,
selectableColors.get(0))}" scope="page"/>
</isif>

    <isif condition="${!empty(selectableColors) && selectableColors.size() > 0}">
       <isset name="firstColorVariation" value="${selectableColors.get(0)}" scope="page"/>
       <isset name="image"
       value="<!-- 4.Get URL for medium image from firstColorVariation -->" scope="page"/>

    <iselse>
       <isset name="image"
       value="<!-- 5.Get medium image from product itself -->" scope="page"/>

    </isif>

<a href="${productUrl}" title="<!-- 6.Get product name --> ${Product.name}">
<img src="${image.getURL()}" alt="${Product.name}"
    title="<!-- Repeat product name -->" height='195' width='195' />
</a>

<a href="${productUrl}"
    title=" <!-- Repeat product name --> "> <!-- Repeat product name -->  </a>
```

Create the `ShowColorVariation` pipeline

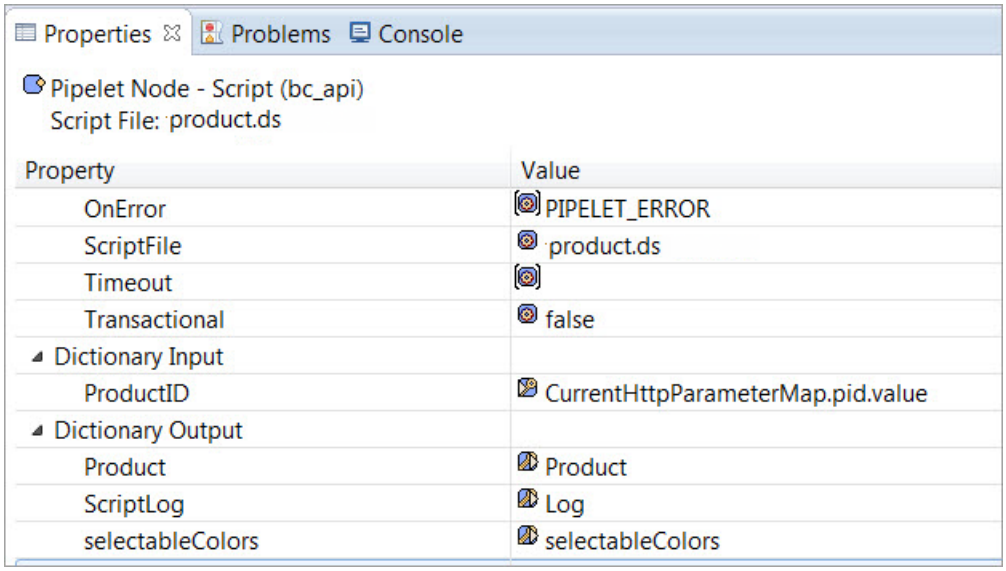
Create the following pipeline which calls the `product.ds` script and the corresponding `product.isml` template.



You'll set the following properties for the `GetProduct` pipelet:



Configure the properties of the `product.ds` script node as follows:



Fill in the blanks in the following `product.ds` script and `product.isml` template code—denoted by the numbered comments in the incomplete script and template.

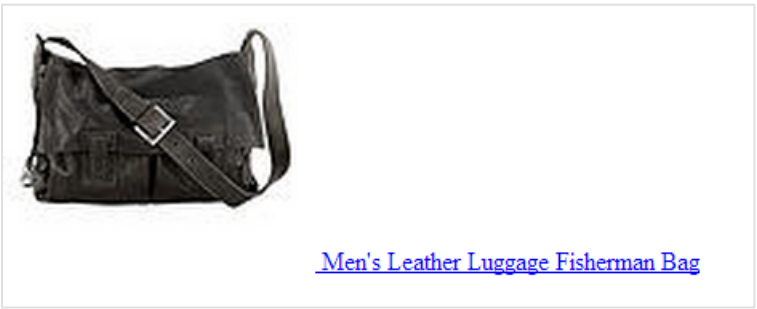
The completed script and template are included in this guide following the example.

Test your Solution

After you fill in the missing methods in the `product.ds` script and the `product.isml` template, you'll test your solution by calling the `ShowProduct` pipeline with a master product, for example:

`http://instance.realm.client.demandware.net/on/demandware.store/Sites-YourSite-Site/default/ShowColorVariation-Start?pid=M1355`

The image of the first product color variation and the link to the item's product details page display:



Try testing the pipeline using other product IDs from your storefront, as well.

Exercise Answers: Product Variations Fill-in-the-Blank

Script Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to check if the product is a master or not:
   ```
   args.Product.master
   ```
2. Code used to get variation model of the product:
   ```
   args.Product.variationModel
   ```
3. Code used to get all values for attributes called 'color':
   ```
   varModel.getProductVariationAttribute("color")
   ```
4. Code used to check if present color is orderable attribute value:
   ```
   varModel.hasOrderableVariants( varAttrsColor, color )
   ```

Template Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to get product Id:
   ```
   Product.ID
   ```
2. Code used to get variation attribute for color:
   ```
   getProductVariationAttribute('color')
   ```
3. Code used to get the URL for selecting variation value:
   ```
   urlSelectVariationValue('Product-Show', colorVarAttr,
   selectableColors.get(0))
   ```
4. Code used to get URL for medium image from firstColorVariation:
   ```
   ${firstColorVariation.getImage('medium')}
   ```
5. Code used to get medium image from product itself:
   ```
   ${Product.getImage('medium',0)}
   ```
6. Code used to get product name:
   ```
   ${Product.name}
   ```

Script Solution: `product.ds`

```
/**
*   @input ProductID : String
*   @output Product : Object
*   @output selectableColors : dw.util.ArrayList
*/
importPackage( dw.system );
importPackage( dw.catalog );
importPackage( dw.web );
importPackage(dw.util);

function execute( args : PipelineDictionary ) : Number
{
    args.Product = ProductMgr.getProduct(args.ProductID);

    if (args.Product == null)
    {
      trace(Resource.msgf('productnotfoundwithid.message', 'product', null, args.ProductID));
       return PIPELET_ERROR;
    }

    if( args.Product.master )
    {
           var varModel : dw.catalog.ProductVariationModel = args.Product.variationModel;
           var varAttrsColor : dw.catalog.ProductVariationAttribute =
                               varModel.getProductVariationAttribute("color");
            args.selectableColors= new ArrayList();
               if( varAttrsColor != null )
               {
                var allColors : dw.util.Collection =
                               varModel.getAllValues( varAttrsColor );

                for each( var color in allColors )
                  {
                   if( varModel.hasOrderableVariants( varAttrsColor, color ) )
                     {
                      args.selectableColors.add( color );
                      }
                  }
               }
    }

     return PIPELET_NEXT;

}
```

Template Solution: `product.isml`

```
<!--- TEMPLATENAME: productfound.isml --->

<isset name="Product" value="${pdict.Product}" scope="page"/>
<isset name="selectableColors" value="${pdict.selectableColors}" scope="page"/>

<isset name="productUrl" value="${URLUtils.http('Product-Show', 'pid', Product.ID)}"
scope="page"/>

<isif condition="${!empty(selectableColors) && selectableColors.size() > 0}">
<isset name="colorVarAttr"
value="${Product.variationModel.getProductVariationAttribute('color')}" scope="page"/>
```

```
<isset name="productUrl" value="${Product.variationModel.urlSelectVariationValue('Product-
Show', colorVarAttr, selectableColors.get(0))}" scope="page"/>
</isif>
    <isif condition="${!empty(selectableColors) && selectableColors.size() > 0}">
        <isset name="firstColorVariation" value="${selectableColors.get(0)}" scope="page"/>
        <isset name="image" value="${firstColorVariation.getImage('medium')}" scope="page"/>

    <iselse>
        <isset name="image" value="${Product.getImage('medium',0)}" scope="page"/>

    </isif>
<a href="${productUrl}" title="${Product.name}">
<img src="${image.getURL()}" alt="${Product.name}" title="${Product.name}" height='195'
width='195' />
</a>
    <a href="${productUrl}" title="${Product.name}"> ${Product.name}      </a>
```

Lesson 2.4: Understanding the `ProductAttributeModel` Class

The `ProductAttributeModel` class represents the attribute information for products. The class provides methods for accessing the attribute definitions and groups for a product. The following table shows some ways to obtain attribute information using the `ProductAttributeModel` class.

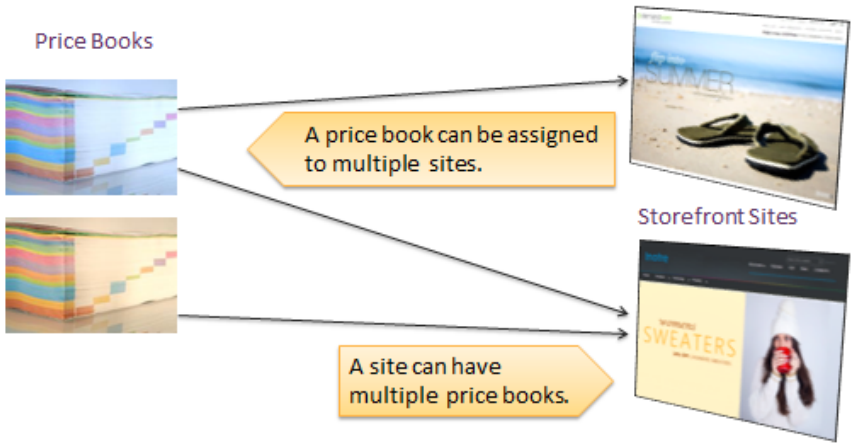
Method or Object	Description
`ProductAttributeModel()`	Represents the attribute groups of the system object type 'Product' and their attributes.
`Category.getProductAttributeModel()`	Retrieves the attribute model for a category. The model represents the catalog attribute groups, the category's attribute groups, and the attribute groups of any parent categories of the category.
`Product.getAttributeModel()`	Retrieves the attribute model for a product. The model represents the catalog product attribute groups, the attribute groups of the product's classification category, and attribute groups of any parent categories. The model also provides access to the product's attribute values. If there is no classification category, only the catalog product attribute group is considered by the model.

Lesson 2.5: Understanding the `ProductPriceModel` Class

The `ProductPriceModel` class provides methods for accessing price book information.

Price Book Data Objects

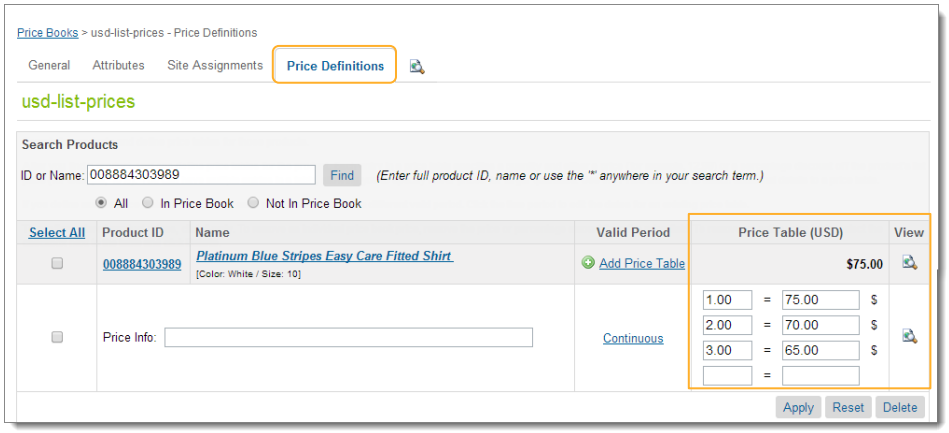
Price books contain the pricing rules for products. A site can have multiple price books. Likewise, a price book can be assigned to multiple sites.



Price books can have a single price or "tiered prices" for products. Tiered prices map prices to the quantity the customer purchases, for example, "buy one item for $75 or buy two for $70 each."

To view the price books for a site using Business Manager, click **Site > Products and Catalogs > Price Books**.

To view the prices in a particular price book, click a price book ID, then click the **Price Definitions** tab. Find a product, then you can view the pricing rules for that product. In the following example, the product has tiered pricing as indicated by the price table.



Merchants can schedule the lifecycle of a price book. For example, a holiday sale price book might be scheduled for November 26 – December 14. A list price book typically has permanent prices with no end date.

To access the price book information for a product, use the `Product.getPriceModel` method.

Calculating the Price of a Product

When a customer accesses a product on the storefront, the site performs a price lookup based on the price model represented by the `ProductPriceModel` class.

The system:

1. Gets all applicable price books based on the current site, the time, session, customer, and active source codes.
2. Identifies all prices in the applicable price books for the requested quantity.
3. Calculates the lowest price of the identified prices.

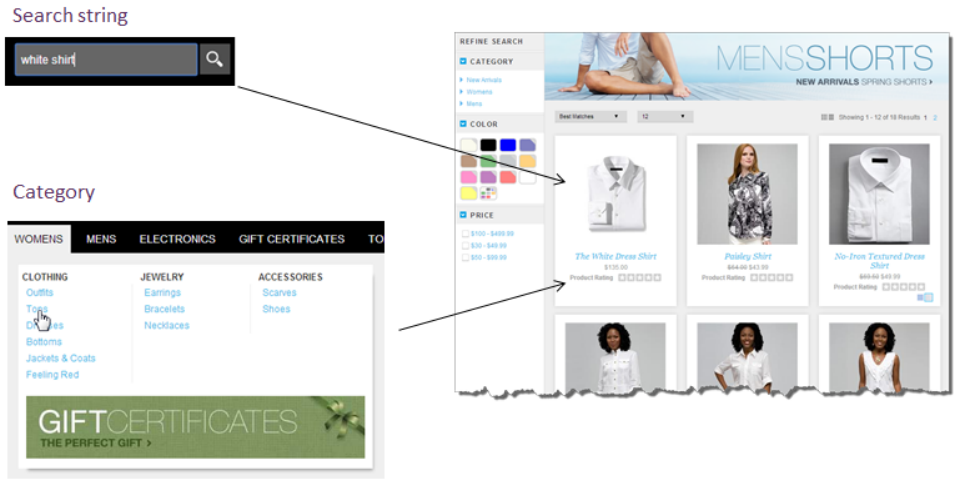
Accessing Product Price Information Using the APIs

- Generally, to perform a standard price lookup, use `Product.getPriceModel().getPrice()`.
- To retrieve a table of the tiered prices, use `Product.getPriceModel().getPriceTable()`.

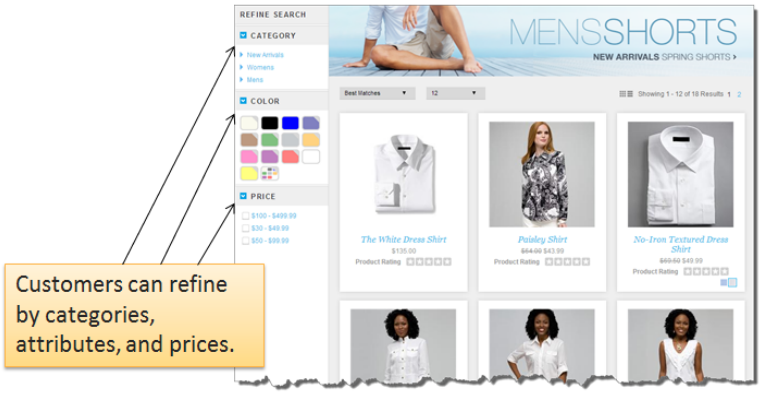
- To determine which price book a product's price was derived from, use `ProductPriceModel.getPriceInfo()`.
- To look up a product's price in a particular price book, use `ProductPriceModel.getPriceBookPrice()`.

Lesson 2.6: Understanding the `ProductSearchModel` Class

The `ProductSearchModel` class represents product search results and refinement information. The class provides methods for generating search URLs. A customer enters a search string or selects a category and the `Search` pipelet generates a search URL, for example: `http://<hostname>/s/SiteGenesis/search?q=white%20shirt`.



The `ProductSearchRefinements` method provides access to refinement options for the product search. Customers "refine" the results by specifying additional product criteria.



You can use the `ProductSearchModel` to retrieve product data, such as all orderable products. By using the search model, you access the search index rather than accessing the database as you would for other `ProductMgr` and `Category` methods, such as `ProductMgr.queryAllSiteProducts()` or `Category.getProducts()`. Using methods such as `ProductSearchModel.orderableProductsOnly()` can improve performance. The *Developing for Performance and Scalability* course provides details and examples that use the search model to access product information efficiently.

Knowledge Check

The Knowledge Check answer key is on the following page.

1. Sites can have more than one of which objects? *Select all objects that apply*.
   a. Price book
   b. Storefront catalog
   c. Category
   d. Master product
   e. Catalog

2. Select all of the following statements that apply to price books in Demandware Commerce.

	Contain pricing rules for products
	Can be assigned to one storefront site only
	Can have tiered pricing
	Must be assigned a start and end date
	Are accessed by a price lookup that calculates the lowest price

Knowledge Check Answers

1. Sites can have more than one of which objects? *Select all objects that apply.*
   a. Price book
   b. Storefront catalog
   c. Category
   d. Master product
   e. Catalog

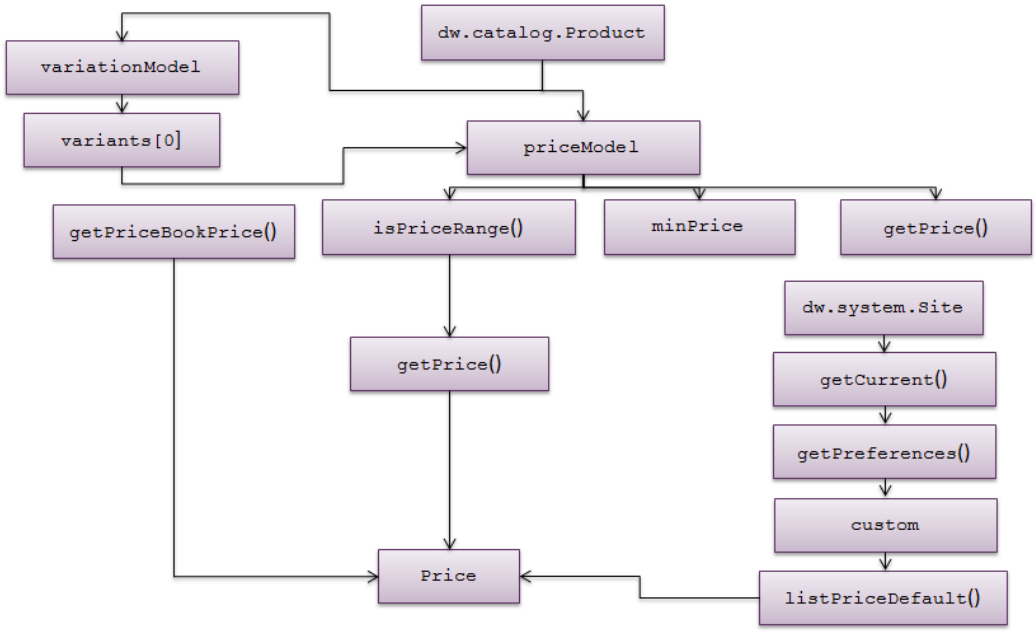
   Answer: a, c, d, e. Multiple price books, categories, master products, and catalogs can be applied to a site. Each site can have only one storefront catalog.

2. Select all of the following statements that apply to price books in Demandware Commerce.

✓	Contain pricing rules for products
	Can be assigned to one storefront site only
✓	Can have tiered pricing
	Must be assigned a start and end date
✓	Are accessed by a price lookup that calculates the lowest price

Exercise: Explore Price Lookup

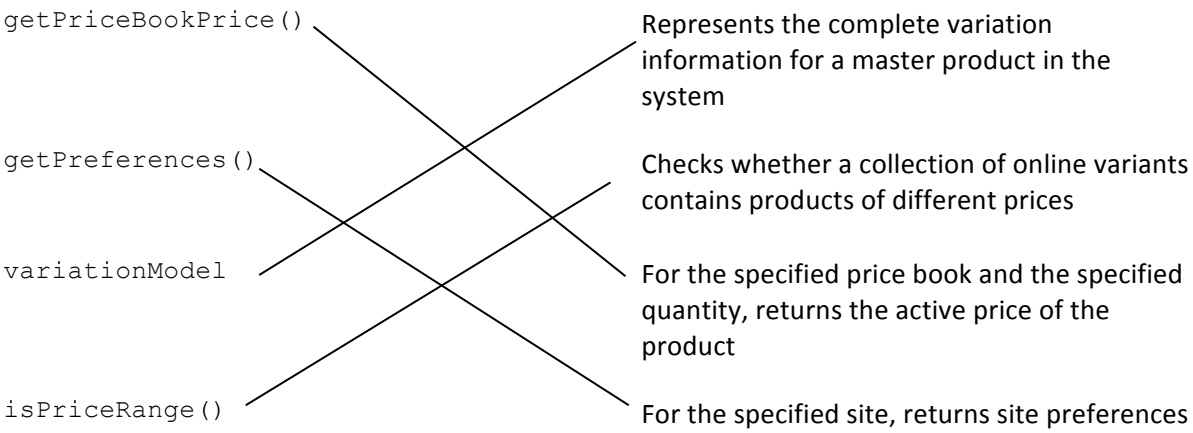
Using the following graph as your guide, click through and review the API documentation (https://info.demandware.com). These are the methods and objects you'll use in the exercises that follow.



After reviewing the documentation for the methods and objects in the preceding graph, match the API elements on the left with the definitions on the right. (Answers on following page)

`getPriceBookPrice()`	Represents the complete variation information for a master product in the system
`getPreferences()`	Checks whether a collection of online variants contains products of different prices
`variationModel`	For the specified price book and the specified quantity, returns the active price of the product
`isPriceRange()`	For the specified site, returns site preferences

Exercise Answers: Explore Price Lookup



Exercise: Price Book Fill-in-the-Blanks

- In this exercise, you'll build a pipeline (`ShowPrice`) which returns the price of a product, as well as its sales price if the product is on sale.
- You're provided with an incomplete ISML template, `price.isml` (included in the `Working with DW APIs Exercises.zip` file under `Exercise Files/Catalog Exercises`).
- You'll fill in the blanks in the `price.isml` template you're given and create a pipeline to run it. Following are descriptions of the APIs you'll use to fill in the blanks.

Template Fill-in-the-Blanks

Fill in the blanks with the APIs needed to complete the `price.isml` template you've been given:

1. Code used to `check if the product is a master product and the price is of a range format`:

   _____

2. Code used to `get the first variant of OrgProduct`:

   _____

3. Code used to `check if the product is an option product`:

   _____

4. Code used to get the price model of the product:

   _____

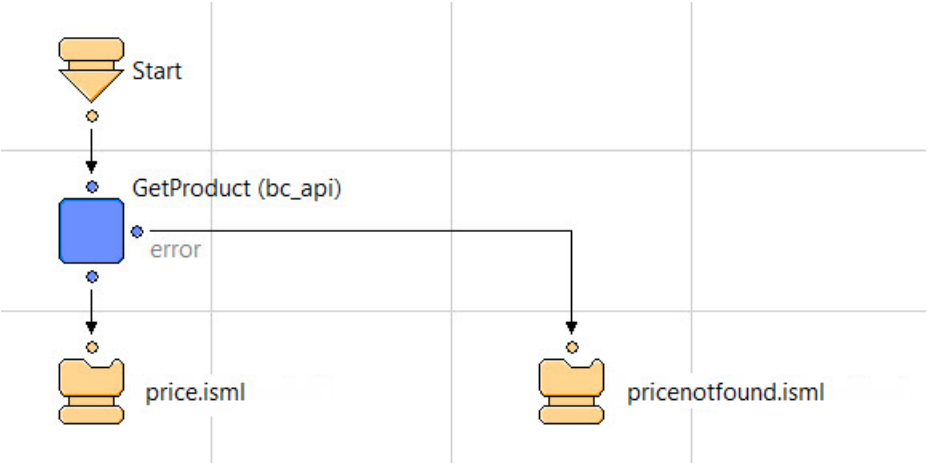
5. Code used to get the price from the price model:

   _____

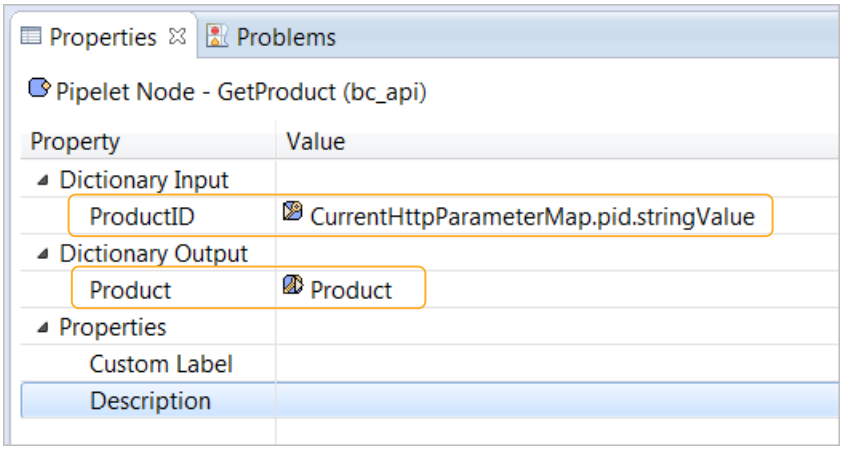
Complete the template: price.isml

```
<!--- TEMPLATENAME: price.isml --->
${pdict.Product.name}
<isset name="Product" value="${pdict.Product}" scope="page"/>

<isif condition="${/** 1.Check if product is a master product and price is range format **/}">
    <isprint value="${Product.priceModel.minPrice}"/>
    <iscomment>If without a price range get the pricing from its first
         variant
    </iscomment>
<iselseif condition="${Product.master && !Product.priceModel.isPriceRange()}"/>

<isset name="OrgProduct" value="${Product}" scope="page"/>
<isset name="Product" value="${/** 2.Get the first variant of OrgProduct **/}" scope="page"/>
</isif>
<iscomment>  Regular pricing through price model of the product. If the product is an option
             product, we have to initialize the product price model with the option model.
</iscomment>
<isif condition="${/** 3.Check if the product is an option product **/}">
    <isset name="PriceModel"
           value="${Product.getPriceModel(Product.getOptionModel())}"
          scope="page"/>
<iselse>
    <isset name="PriceModel" value="${/** 4.Get price model of the product **/}"
scope="page"/>
</isif>

<iscomment>Check whether the product has price in the sale pricebook. If so, then display two
prices: crossed-out standard price and sales price.</iscomment>

<isinclude template="product/components/standardprice"/>
<isset name="SalesPrice" value="${/** 5.Get price from the price model **/}"
       scope="page"/>
<isset name="ShowStandardPrice" value="${StandardPrice.available &&
       SalesPrice.available && StandardPrice.compareTo(SalesPrice) == 1}"
       scope="page"/>

<isif condition="${ShowStandardPrice}">
    <span class="product-standard-price" title="Regular Price">
       <isprint value="${StandardPrice}"/></span>
       <span class="product-sales-price" title="Sale Price">
       <isprint value="${SalesPrice}"/></span>
<iselse>
    <span class="product-sales-price" title="Sale Price">
    <isprint value="${SalesPrice}"/></span>
</isif>

<iscomment>Restore current product instance</iscomment>
<isset name="Product" value="${OrgProduct}" scope="page"/>
```

To begin, create the following pipeline which includes the `price.isml` template:



You'll set the following properties for the `GetProduct` pipelet:



Fill in the blanks in the `price.isml` template code—denoted by the numbered comments in the `price.isml` template.

The completed template is included in this guide following the example.

Test your solution

Test your solution by calling the `ShowPrice` pipeline with a master product, for example:

`http://instance.realm.client.demandware.net/on/demandware.store/Sites-YourSite-Site/default/ShowPrice-Start?pid=M1355`

The name of the item displays, followed by the original price, followed by the sales price—if the item is on sale:

Men's Leather Luggage Fisherman Bag $180.00 $162.00

Test the pipeline using other product IDs from your storefront, as well—items on sale and items that aren't on sale.

Exercise Answers

1. Code used to check if the product is a master product and the price is of a range format:

   ```
   Product.master && Product.priceModel.isPriceRange()
   ```

2. Code used to get the first variant of OrgProduct:

   ```
   OrgProduct.variationModel.variants[0]
   ```

3. Code used to check if the product is an option product:

   ```
   Product.optionProduct()
   ```

4. Code used to get the price model of the product:

   ```
   Product.getPriceModel()
   ```

5. Code used to get the price from the price model:

   ```
   PriceModel.getPrice()
   ```

Template Solution for `price.isml`

```
<!--- TEMPLATENAME: price.isml --->
${pdict.Product.name}
<isset name="Product" value="${pdict.Product}" scope="page"/>

<isif condition="${Product.master && Product.priceModel.isPriceRange()}">
    <isprint value="${Product.priceModel.minPrice}"/>
    <iscomment>If without a price range get the pricing from its first variant
    </iscomment>
<iselseif condition="${Product.master && !Product.priceModel.isPriceRange()}"/>

<isset name="OrgProduct" value="${Product}" scope="page"/>
<isset name="Product" value="${OrgProduct.variationModel.variants[0]}"
       scope="page"/>
</isif>
<iscomment>  Regular pricing through price model of the product. If the product is
       an option product, we have to initialize the product price model with the
       option model.
</iscomment>
<isif condition="${Product.optionProduct}">
    <isset name="PriceModel"
       value="${Product.getPriceModel(Product.getOptionModel())}" scope="page"/>
<iselse>
    <isset name="PriceModel" value="${Product.getPriceModel()}" scope="page"/>
</isif>

<iscomment>Check whether the product has price in the sale pricebook. If so, then display two
prices: crossed-out standard price and sales price.</iscomment>
```

```
<isinclude template="product/components/standardprice"/>
<isset name="SalesPrice" value="${PriceModel.getPrice()}" scope="page"/>
<isset name="ShowStandardPrice" value="${StandardPrice.available &&
    SalesPrice.available && StandardPrice.compareTo(SalesPrice) == 1}" scope="page"/>

<isif condition="${ShowStandardPrice}">
    <span class="product-standard-price" title="Regular Price">
    <isprint value="${StandardPrice}"/></span>
    <span class="product-sales-price" title="Sale Price">
    <isprint value="${SalesPrice}"/></span>
<iselse>
    <span class="product-sales-price" title="Sale Price">
    <isprint value="${SalesPrice}"/></span>
</isif>

<iscomment>Restore current product instance</iscomment>
<isset name="Product" value="${OrgProduct}" scope="page"/>
```

Exercise: Catalog Scenarios

In this activity, you're provided with scenarios for managing catalogs, categories, products, and prices. Specify the classes and methods from the `dw.catalog` package that you'd use to implement the scenario.

1. Which class and method from the `dw.catalog` package would you use to return all categories of a product?

   _____

2. Which class and method from the `dw.catalog` package would you use to retrieve all products of a particular category?

   _____

3. Which class and method from the `dw.catalog` package would you use to retrieve the price of a product?

   _____

4. Which class and method from the `dw.catalog` package would you use to retrieve all attributes of a particular category?

   _____

Exercise Answers: Catalog Scenarios

1. Which class and method from the `dw.catalog` package would you use to return all categories of a product?

   `Product.getAllCategories`

2. Which class and method from the `dw.catalog` package would you use to retrieve all products of a particular category?

   `Category.getProducts`

3. Which class and method from the `dw.catalog` package would you use to retrieve the price of a product?

   `ProductPriceModel.getPrice`

4. Which class and method from the `dw.catalog` package would you use to retrieve all attributes of a particular category?

   `Category.getProductAttributeModel`

Module 3: Working with the `dw.campaign` Package

Learning Objectives

At the end of this module, you will be able to:

- Describe the methods and objects of the `Promotion`, `Campaign`, `PromotionPlan`, and `PromotionMgr` classes from the `dw.campaign` package.
- Apply the APIs in the `dw.campaign` package to implement new promotion and campaign functionality.

Lesson 3.1: Understanding the `Promotion` and `Campaign` Classes

The `Campaign` class is used for managing promotions and campaigns.

- The `Campaign` class manages experiences (promotions, slot configurations, and sorting rules).
- The `Promotion` class provides methods for accessing the basic attributes of a promotion such as name, callout message, and description.
- The `PromotionMgr` class lets you access campaign and promotion details, display promotions, and calculate and apply discounts (price reductions).

Promotion and Campaign Data Objects

In Demandware, a campaign is a container for one or more experiences, where an experience is made up of promotions, the content slots where they display, and the sorting rules that apply to the promotions.

In this example, the Fall Sweater Campaign is a container for three promotions, a “10% Off Cashmere Sweaters” promotion, a “Buy 1 Cable Sweater, Get 2nd Half Off” promotion, and a “Free Shipping” promotion.

Qualifiers

Promotions and campaigns are triggered by “qualifiers,” including coupons, source codes, and customer groups.

- Customers enter **coupon codes** on the storefront during checkout to enable the promotion.
- Customers can click hyperlinked **source codes** on web pages or emails that direct them to the site and enable the promotion.
- Customers are admitted into **customer groups** defined by a list of emails or dynamically by a rule. Group membership qualifies customers for specific promotions.

Qualifiers can apply to a promotion or a campaign. If a qualifier is applied to a campaign, it applies to all promotions contained in the campaign.

To create a campaign, you create its promotions, assign qualifiers, assign applicable content to content slots, and you assign sorting rules, such as “Customer Favorites.” As with promotions, you can create a schedule for a campaign, as well.

To create a promotion, click **Online Marketing > Promotions** and click **New**. You can create promotions while creating Campaigns, as well. To create a campaign, click **Online Marketing > Campaigns** and click **New**.

In this example, each of the three promotions has its own qualifier to trigger the promotion:

- The “10% off cashmere sweaters” promotion is triggered by the coupon code, CASHMERE.
- The “Free shipping” promotion is triggered by the “MyStore credit card customers” customer group.
- The “Buy 1 cable sweater, get 2nd half off” promotion is triggered by a source code with a link to a page on the site.

Campaigns are created for a single site. Settings in the campaign are inherited by its experiences. For example, promotions inherit the qualifiers of their containing campaigns. So if a customer group is defined for a campaign, its promotions inherit the customer group.

If merchants set multiple qualifiers for a campaign or promotion, they can choose to allow *any* of those qualifiers to trigger the promotion. Similarly, they can specify that *all* specified qualifiers be required to trigger the promotion.

Likewise, a promotion inherits its schedule from its containing campaign. Merchants can set a schedule for the promotion to override the schedule of the containing campaign, as long as the date range specified for the promotion is within the campaign date range. Promotions can be scheduled or continuous.

In addition to specifying schedules to activate and deactivate campaigns and promotions, merchants can enable or disable campaigns and promotions explicitly. Disabled campaigns and promotions are inactive. If a campaign is disabled, its promotions are deactivated, as well, even if they are set to “enabled.” The campaign setting overrides the promotion setting.

- Use the `PromotionMgr.getCampaigns()` method to return all campaigns for the site.
- Use the `Campaign.getPromotions()` method to return the promotions assigned to a campaign.
- Use the `getCoupons()`, `getSourceCodeGroups()`, and `getCustomerGroups()` methods for returning the qualifiers for the campaign.
- Use the `getStartDate()` and `getEndDate()` methods to get the schedule range of the campaign. Like the `Campaign` class, the `Promotion` class provides these same methods to return qualifiers, and schedules.
- Use the `PromotionMgr` class to return promotions and campaigns for a particular site.

Knowledge Check

The Knowledge Check answer key is on the following page.

1. Match the definitions on the left with the terms on the right.

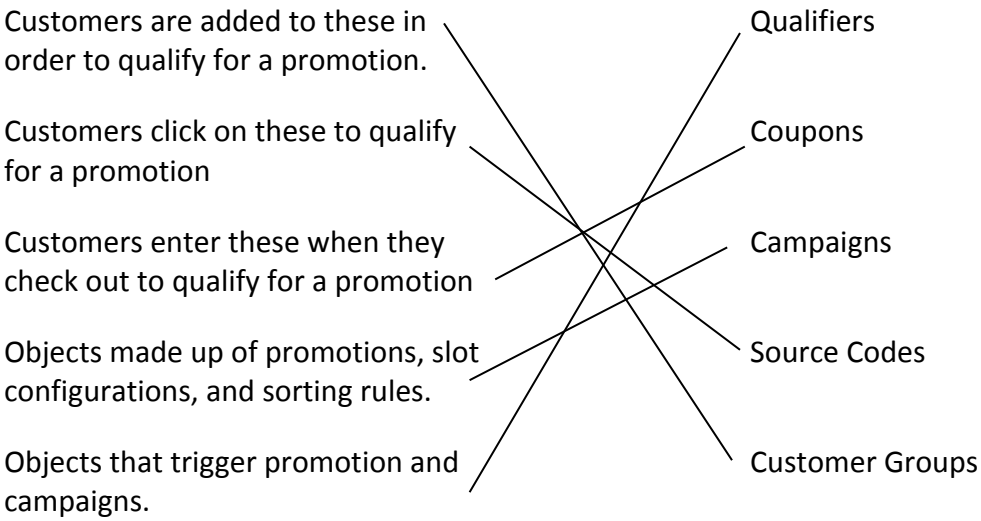
Customers are added to these in order to qualify for a promotion.	Qualifiers
Customers click on these to qualify for a promotion	Coupons
Customers enter these when they check out to qualify for a promotion	Campaigns
Objects made up of promotions, slot configurations, and sorting rules.	Source Codes
Objects that trigger promotions and campaigns.	Customer Groups

2. Merchants can create a campaign for
   a. At most three sites
   b. As many sites as they want
   c. One site

3. A campaign is scheduled for January 2 - January 16. Select the promotions below that cannot be a part of the campaign.
   a. 20% Off Parka promotion, scheduled for January 9 - January 17
   b. Jewelry Daily Deal promotion, scheduled on two consecutive Tuesdays beginning on Tuesday, January 3
   c. 50% Any One Item promotion, scheduled for January 2

Knowledge Check Answers

1. Match the definitions on the left with the terms on the right.



2. Merchants can create a campaign for
   a. At most three sites
   b. One site
   c. As many sites as they want

Answer: b. Campaigns can be set for one site only.

3. A campaign is scheduled for January 2 - January 16. Select the promotion below that cannot be a part of the campaign.
   a. 20% Off Parka promotion, scheduled for January 9 - January 17
   b. Jewelry Daily Deal promotion, scheduled on two consecutive Tuesdays beginning on Tuesday, January 3
   c. 50% Any One Item promotion, scheduled for January 2

Answer: a. To be included in the campaign, the promotion must fall within the schedule range of the campaign. The 20% Off Parka promotion falls outside the campaign's range.

Lesson 3.2: Understanding the `PromotionPlan` Class

The `PromotionPlan` class represents a set of `Promotion` objects. Use the `PromotionPlan` class to:

- Display active or upcoming promotions for a storefront using the `PromotionMgr` class.
- Calculate a discount plan, a set of discounts represented by the `DiscountPlan` class, and apply it to the line item container. Discounts can be a fixed price, a percentage off, an amount off or free. The `DiscountPlan` class provides methods to access the plan discounts, add additional discounts to the plan, or remove discounts from the plan.

Methods that return promotions sort the promotions by the following criteria in the order shown:

1. **Exclusivity**: Exclusivity defines whether the promotion can be combined with other promotions. If **Exclusivity** is **No**, customers can use the promotion with any other promotions. If **Exclusivity** is **Class**, customers can use only one promotion per class, where classes include order, shipping, or product promotions. If **Exclusivity** is **Global**, customers can't use the promotion with other promotions. Promotions with global exclusivity take precedence, followed by promotions with class exclusivity, followed by those with no exclusivity.
2. **Rank**: Ranked promotions are sorted in ascending order.
3. **Promotion class**: Product promotions take precedence, followed by order promotions, followed by shipping promotions.
4. **Discount type**: Fixed price promotions take precedence, followed by free promotions, followed by amount-off promotions, followed by percentage-off promotions, and finally, bonus product promotions.
5. **Best discount**: Best discount promotions are sorted in descending order. For example, 30% off takes precedence over 20% off.
6. **ID**: The final sorting criteria are IDs sorted in alphanumeric ascending order.

Lesson 3.3: Understanding the `PromotionMgr` Class

The methods in the `PromotionMgr` class let you:

- Access campaign and promotion definitions.
- Display active or upcoming promotions in a storefront
- Calculate and apply promotional discounts to line item containers.

To access campaign and promotion definitions:

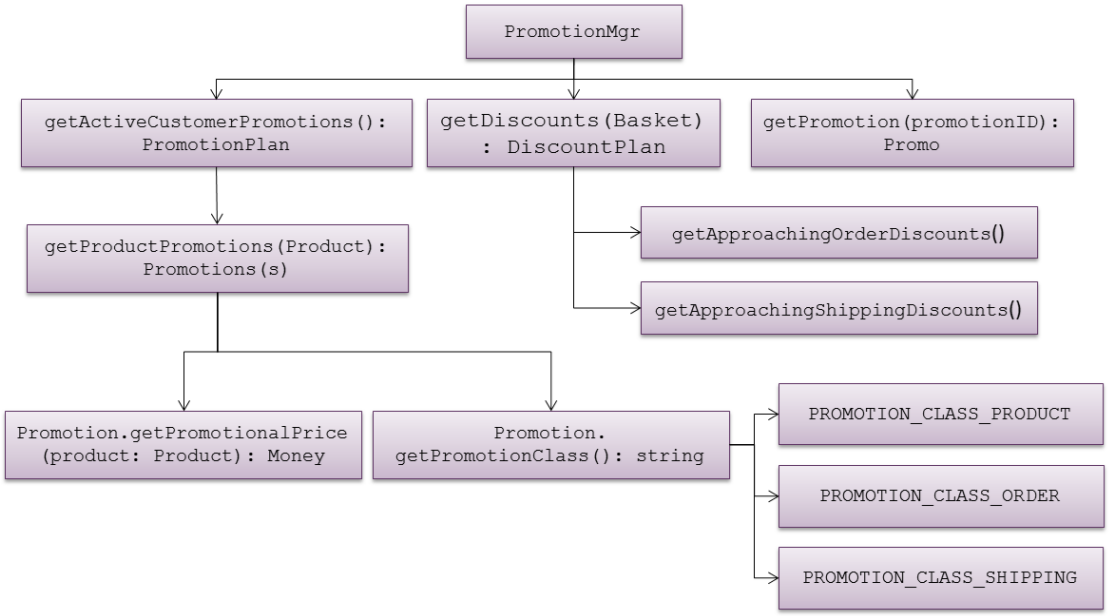
Use the `getCampaigns()`, `getCampaign(String)`, `getPromotions()`, and `getPromotion(String)` methods.

To display active or upcoming promotions:

- Use the `getActivePromotions()` method to return the `PromotionPlan` instances that have all enabled promotions scheduled for *now*.
- Use the `getActiveCustomerPromotions()` method to return a `PromotionPlan` instance with all enabled promotions applicable for the current customer and active source code that are scheduled for *now*.
- `getUpcomingPromotions(previewTime)` returns a `PromotionPlan` instance with all enabled promotions that are not scheduled for *now*, but are scheduled for anytime between *now* and *now + previewTime(hours)*. The parameter *previewTime* specifies for how many hours promotions should be previewed.

Exercise: Explore Promotions

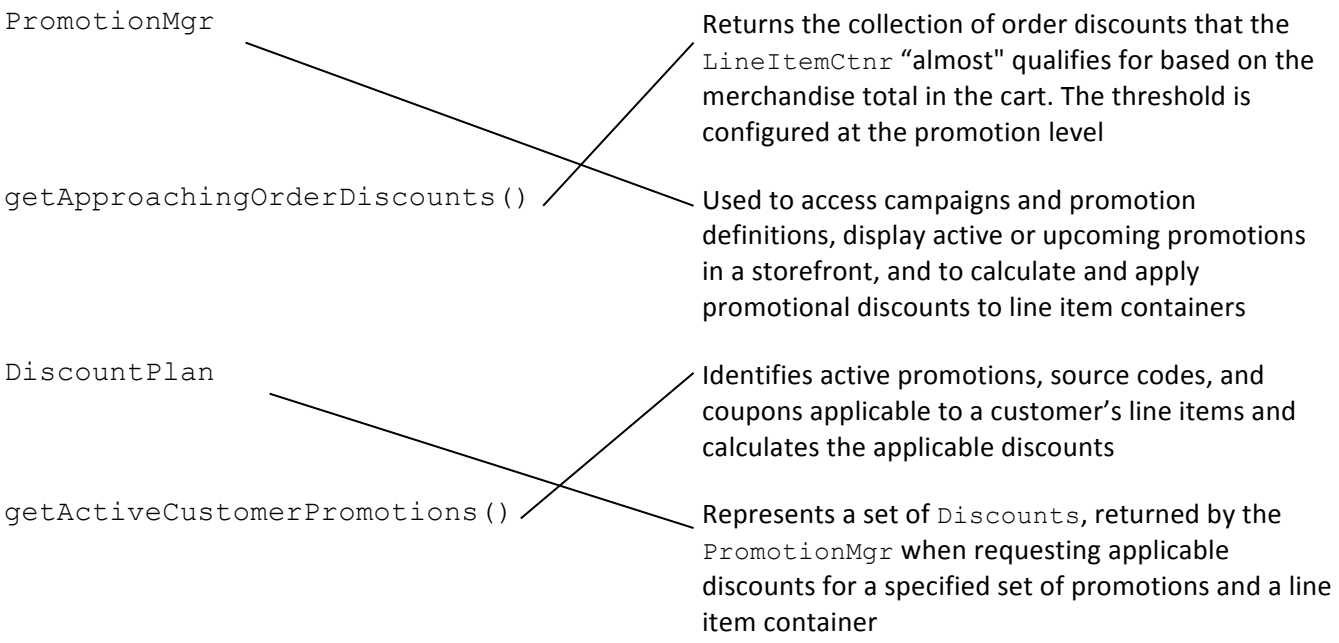
Using the following graph as your guide, click through and review the API documentation (https://info.demandware.com). These are the methods and objects you'll use in the exercises that follow.



After reviewing the documentation for the methods and objects in the preceding graph, match the API elements on the left with the definitions on the right. (Answers on following page)

`PromotionMgr`	Returns the collection of order discounts that the `LineItemCtnr` "almost" qualifies for based on the merchandise total in the cart. The threshold is configured at the promotion level
`getApproachingOrderDiscounts()`	Used to access campaigns and promotion definitions, display active or upcoming promotions in a storefront, and to calculate and apply promotional discounts to line item containers
`DiscountPlan`	Identifies active promotions, source codes, and coupons applicable to a customer's line items and calculates the applicable discounts
`getActiveCustomerPromotions()`	Represents a set of `Discounts`, returned by the `PromotionMgr` when requesting applicable discounts for a specified set of promotions and a line item container

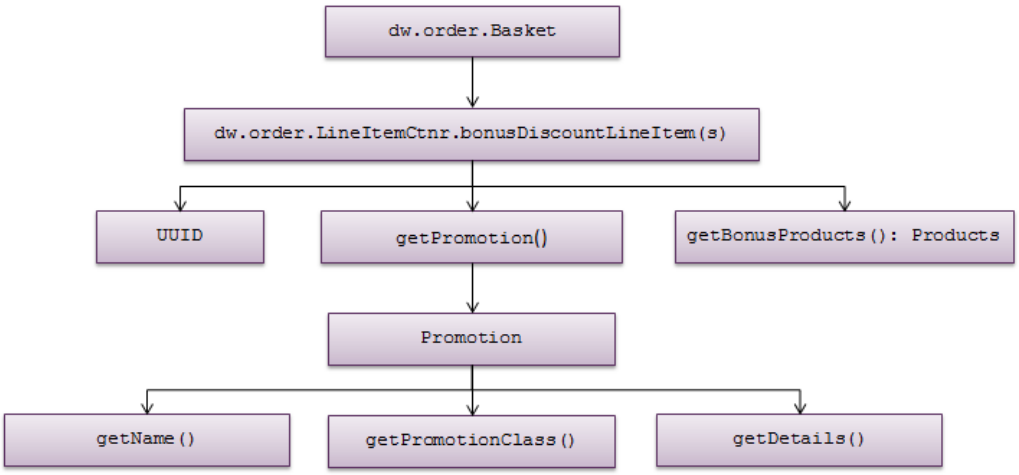
Exercise Answers: Explore Promotions



Exercise: Explore Bonus Discounts

In order to complete the fill-in-blank coding exercise for this module, you'll need to use methods from the `dw.order.Basket` class, along with the `Campaign` class.

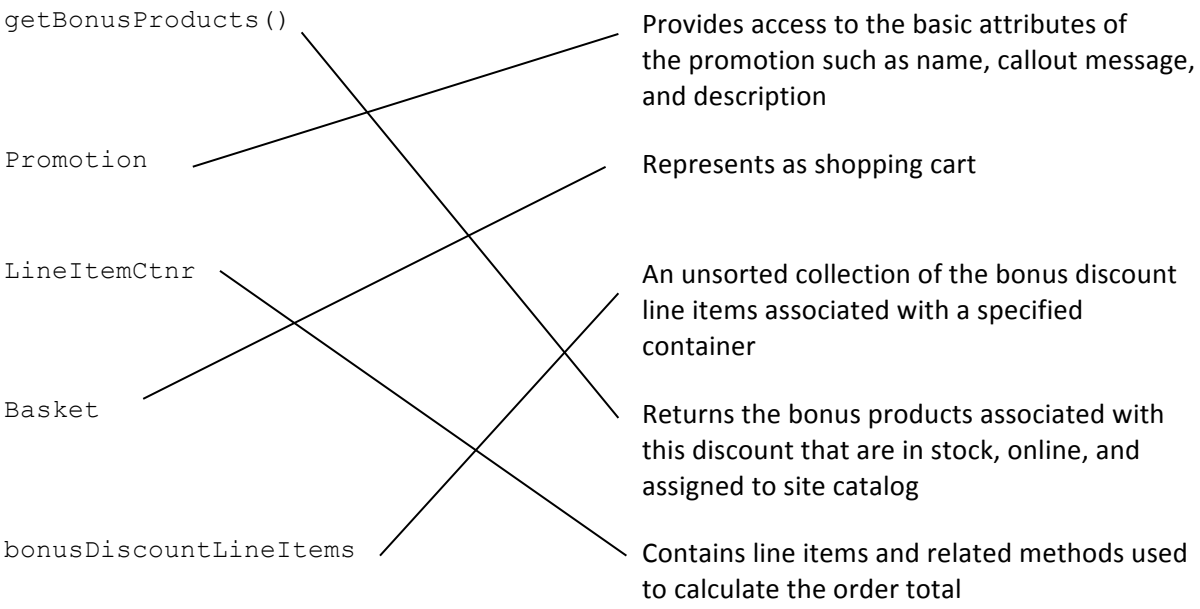
Using the following graph as your guide, click through and review the API documentation (`https://info.demandware.com`). These are the methods and objects you'll use in the exercises that follow.



After reviewing the documentation for the methods and objects in the preceding graph, match the API elements on the left with the definitions on the right. (Answers on following page)

`getBonusProducts()`	Provides access to the basic attributes of the promotion such as name, callout message, and description
`Promotion`	Represents a shopping cart
`LineItemCtnr`	An unsorted collection of the bonus discount line items associated with a specified container
`Basket`	Returns the bonus products associated with this discount that are in stock, online, and assigned to site catalog
`bonusDiscountLineItems`	Contains line items and related methods used to calculate the order total

Exercise Answers: Explore Bonus Discounts



Exercise: Promotion Fill-in-the-Blanks

In this exercise, you'll build a pipeline (`PrintBonusProduct`) which calls a script to print the bonus product and the promotion details of a campaign you'll create. You're provided with an incomplete script, `printBonusProduct.ds`, and a corresponding incomplete ISML template, `printBonusProduct.isml` (included in the `Working with DW APIs Exercises.zip` file). (included in the `Working with DW APIs Exercises.zip` file under `Exercise Files/Campaign Exercises`). You'll replace the comments in the script and template with the missing API objects and methods by consulting the `dw.campaign` API documentation.

The `PrintBonusProduct` pipeline that you'll be creating adds a bonus product to a customer's cart if the customer's order is over $150.

You'll fill in the blanks in the script and template you're given and create a pipeline to run them. Following are descriptions of the APIs you'll use to fill in the blanks.

Script Fill-in-the-Blanks

Fill in the blanks with the APIs needed to complete the `printBonusProduct.ds` script you've been given:

1. Code used to `get active promotions for current customer`:

   _____

2. Code used to `get promotions for the order`:

   _____

3. Code used to `get ith order promotion`:

   _____

4. Code used to `get callout message from promotion`:

   _____

5. Code used to `get details of promotion`:

   _____

Complete the Script: `printBonusProduct.ds`

The script adds a bonus product to a customer's cart if the customer's order is over $150.

```
/**
* Demandware Script File
* To define input and output parameters, create entries of the form:
*
* @<paramUsageType> <paramName> : <paramDataType> [<paramComment>]
*
* where
*   <paramUsageType> can be either 'input' or 'output'
*   <paramName> can be any valid parameter name
*   <paramDataType> identifies the type of the parameter
*   <paramComment> is an optional comment
*
* For example:
*
*-   @input ExampleIn : String This is a sample comment.
*-   @output ExampleOut : Number
*-
* @output promoCallout : String
* @output promoDetails: String
*
*/
importPackage( dw.system );
importPackage(dw.campaign);
importPackage(dw.util);
```

```
function execute( args : PipelineDictionary ) : Number
{
  var promoPlan:PromotionPlan=/* 1.Get active promotions for current customer */;
  var orderPromotions:Collection=/* 2.Get promotions for the order */;
  for(var i=0;i<orderPromotions.length;i++){
    var promo:Promotion =/* 3.Get ith order promotion */;
    args.promoCallout=/* 4.Get callout message from promotion */;
    args.promoDetails=/* 5.Get details of promotion */;

    return PIPELET_NEXT;
  }

}
```

Template Fill-in-the-Blanks

Fill in the blanks with the APIs needed to complete the `printBonusProduct.isml` template you've been given:

1. Code used to `print callout message`:

   _____

2. Code used to `print promotion details`:

   _____

3. Code used to `include util/modules.isml to incorporate producttile custom tag below`:

   _____

4. Code used to `create a variable, basket, which represents present shopping cart and load it on the page scope`:

   _____

5. Code used to `get line items of basket`:

   _____

6. Code used to `get product from oneItem`:

   _____

7. Code used to `use producttile custom tag to print the product showing promotion also`:

   _____

Complete the Template: `printBonusProduct.isml`

The template displays the promotion details, product line items, and the bonus product.

```
<!--- TEMPLATENAME: printBonusProduct.isml --->

<html>
<head>
  This is the basket page ${ <!-- 1.Print callout message --> }
</head>
<body>

<h1>${ <!-- 2.Print promotion details --> } </h1>
<!-- 3.Include util/modules.isml to incorporate producttile custom tag below -->


<!-- 4. Use isset tag to create a variable, basket, which represents present shopping cart
and load it on the page scope -->


<isloop items= "${ <!-- 5.Get line items of basket --> }" var="oneItem" >

   <isset name="oneProduct" value="${ <!-- 6.Get product from oneItem --> }" scope="page" />
   <!-- 7.Use producttile custom tag to print the product showing promotion also -->

</isloop>

</body>
</html>
```

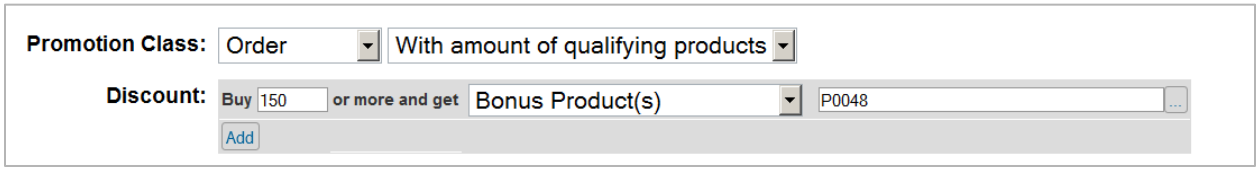
Create the BonusProduct promotion

When you develop your script, you'll test it on your storefront site. You'll need to create the related promotion and campaign on the site.

1. Navigate to **Online Marketing > Promotions**.
2. Click **New** to create a new promotion.
3. Set the **ID** and **Name** to **BonusProduct**.
4. Set the **Callout Message** to *This is a Bonus Product Promotion for orders over $150*.
5. Set **Promotion Details** to *If you have an order over $150, you'll receive a laptop briefcase as a bonus product*.
6. Set **Exclusivity** to **NO**.
7. Click **Apply**.

   The Promotion Rule section displays below.
8. For **Promotion Class**, select **Order** and select **With amount of qualifying products**.

9. For **Discount**, enter 150 for **Buy**, select **Bonus Product** for **get**, and enter **P0048** as the bonus product:



10. Click **Apply**.

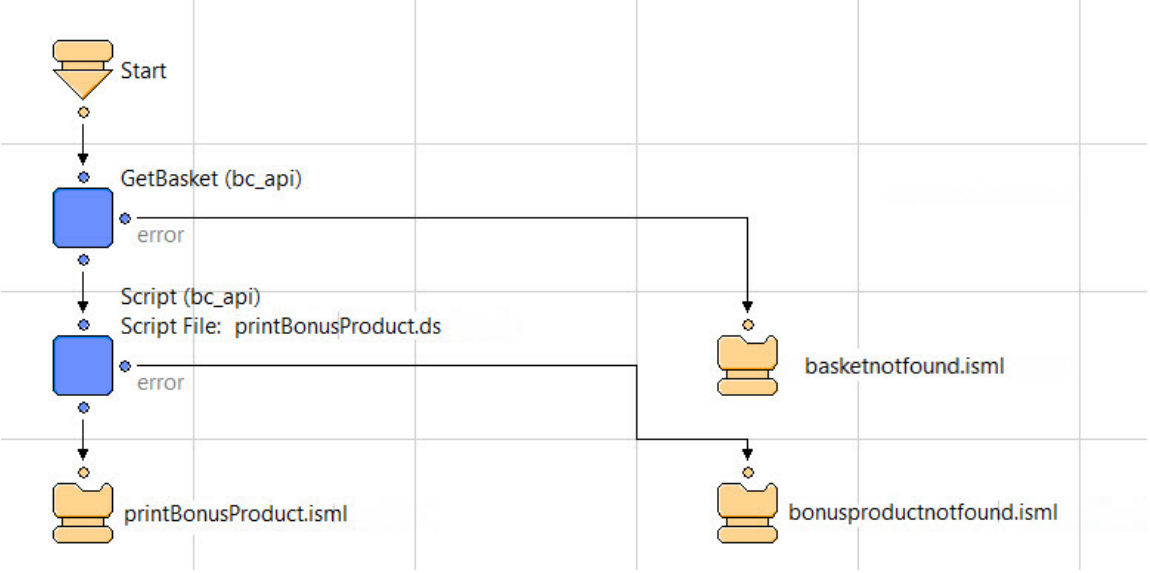
Create a campaign that contains the `BonusProduct` promotion

Now that you've created the `BonusProduct` promotion, you'll create the campaign that contains it.

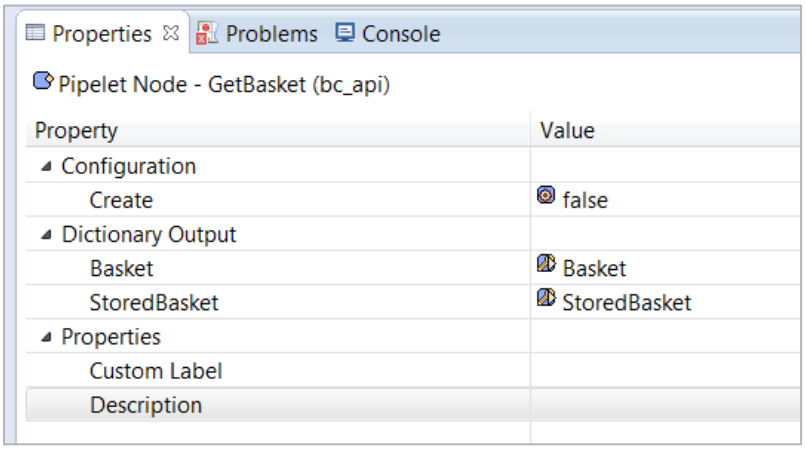
1. Navigate to **Online Marketing > Campaigns**.
2. Click **New** to create a new campaign.
3. Set the **ID** to **BonusProductCampaign**.
4. For **Start Date**, select **No Start Date**.
5. For **End Date**, select **No End Date**.
6. For **Customer Group**, click **Edit**, select **Everyone**, and click **Apply**.
7. Click **Add Experience > Add Promotion**, select **BonusProduct**, and click **Apply** to add the experience.
8. Click **Apply** to create the campaign.

Create the `PrintBonusProduct` pipeline

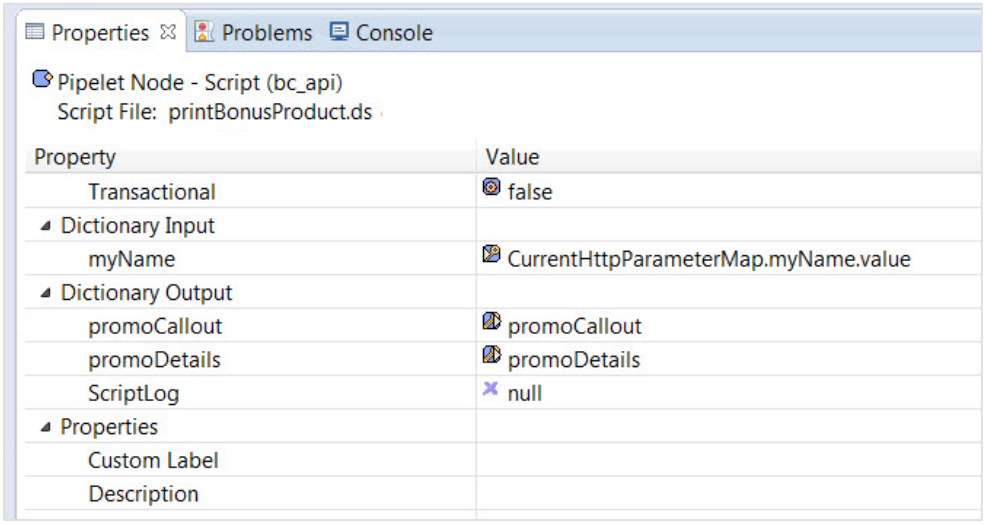
Create the following pipeline which calls the `printBonusProduct.ds` script and the corresponding `printBonusProduct.isml` ISML template.



You'll set the following properties for the `GetBasket` pipelet:



Configure the properties of the `printBonusProduct.ds` script node as follows:



Fill in the blanks in the `printBonusProduct.ds` script and the `printBonusProduct.isml` template—denoted by the numbered comments in the script and template.

The completed script and template are included in this guide following the example.

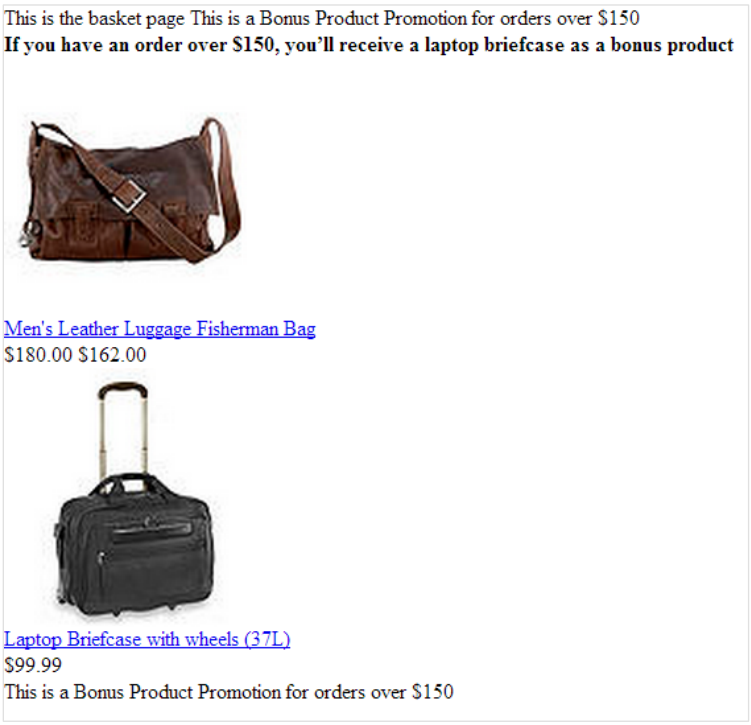
Test your solution

After you fill in the missing methods in the following script and template, you'll test your solution by following these steps:

1. Navigate to your storefront and add products that total more than $150.
2. Execute your pipeline:

   `http://`*`instance`*`.`*`realm`*`.`*`client`*`.demandware.net/on/demandware.store/`*`Sites-YourSite-Site`*`/default/printBonusProduct-Start`

The items in the basket are displayed, followed by the bonus product as shown in the following sample output:



Script Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to get active promotions for current customer:
   ```
   PromotionMgr.getActiveCustomerPromotions()
   ```
2. Code used to get promotions for the order:
   ```
   promoPlan.getOrderPromotions()
   ```
3. Code used to get ith order promotion:
   ```
   orderPromotions[i]
   ```
4. Code used to get callout message from promotion:
   ```
   promo.getCalloutMsg().getMarkup()
   ```
5. Code used to get details of promotion:
   ```
   promo.getDetails().getMarkup()
   ```

Template Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to print callout message:

   ```
   pdict.promoCallout
   ```

2. Code used to print promotion details:

   ```
   pdict.promoDetails
   ```

3. Code used to include util/modules.isml to incorporate producttile custom tag below:

   ```
   <isinclude template="util/modules.isml" />
   ```

4. Code used to create a variable, basket, which represents present shopping cart and load it on the page scope:

   ```
   <isset name="basket" value="${pdict.Basket}" scope="page"/>
   ```

5. Code used to get line items of basket:

   ```
   basket.allLineItems
   ```

6. Code used to get product from oneItem:

   ```
   oneItem.product
   ```

7. Code used to use producttile custom tag to print the product showing promotion also:

   ```
   <isproducttile product="${oneProduct}" showpricing="${true}"
   showswatches="${true}" showpromotion="${true}" />
   ```

Completed Code for `printBonusProduct.ds`

```
/**
*
*   @output promoCallout : String
*   @output promoDetails: String
*
*
*/
importPackage( dw.system );
importPackage(dw.campaign);
importPackage(dw.util);

function execute( args : PipelineDictionary ) : Number
{
  var promoPlan:PromotionPlan=PromotionMgr.getActiveCustomerPromotions();
  var orderPromotions:Collection=promoPlan.getOrderPromotions();
  for(var i=0;i<orderPromotions.length;i++){
   var promo:Promotion = orderPromotions[i];
   args.promoCallout=promo.getCalloutMsg().getMarkup();
   args.promoDetails=promo.getDetails().getMarkup();

   return PIPELET_NEXT;
  }
}
```

Completed Code for `printBonusProduct.isml`

```
<html>
<head>
  This is the basket page ${pdict.promoCallout}
</head>
<body>

<h1>${pdict.promoDetails} </h1>
<isinclude template="util/modules.isml" />

<isset name="basket" value="${pdict.Basket}" scope="page"/>

<isloop items="${basket.allLineItems}" var="oneItem" >

       <isset name="oneProduct" value="${oneItem.product}" scope="page" />
        <isproducttile product="${oneProduct}" showpricing="${true}" showswatches="${true}"
        showpromotion="${true}" />
</isloop>
</body>
</html>
```

Exercise: Campaign Scenarios

In this activity, you're provided with scenarios for managing promotions and campaigns. Specify the classes and methods from the `dw.campaign` package that you'd use to implement the scenario.

1. Which class and method from the `dw.campaign` package would you use to display all of the promotions applicable to a product on its product details page?

   _____

2. Which class and method from the `dw.campaign` package would you use to retrieve all customer groups applicable to a campaign?

   _____

3. Which class and method from the `dw.campaign` package would you use to display the percentage discount for a product in the cart?

   _____

4. Which classes and methods from the `dw.campaign` package would you use to retrieve a promotion's schedule range and that of a specified campaign to validate whether the promotion can be included in the campaign?

   _____

5. Which class and method from the `dw.campaign` package would you use to determine which Summer Swim campaign promotions will be available at some point during the first two weeks of June?

   _____

Exercise Answers: Campaign Scenarios

1. Which class and method from the `dw.campaign` package would you use to display all of the promotions applicable to a product on its product details page?

   `PromotionPlan.getProductPromotions()`

2. Which class and method from the `dw.campaign` package would you use to retrieve all customer groups applicable to a campaign?

   `Campaign.getCustomerGroups()`

3. Which class and method from the `dw.campaign` package would you use to display the percentage discount for a product in the cart?

   `DiscountPlan.getProductDiscounts()`

4. Which classes and methods from the `dw.campaign` package would you use to retrieve a promotion's schedule range and that of a specified campaign to validate whether the promotion can be included in the campaign?

   `Promotion.getStartDate()`, `Promotion.getEndDate()`, `Campaign.getStartDate()`, `Campaign.getEndDate()`

5. Which class and method from the `dw.campaign` package would you use to determine which Summer Swim campaign promotions will be available at some point during the first two weeks of June?

   `PromotionMgr.getActivePromotionsForCampaign()`

Module 4: Working with the `dw.order` Package

Learning Objectives

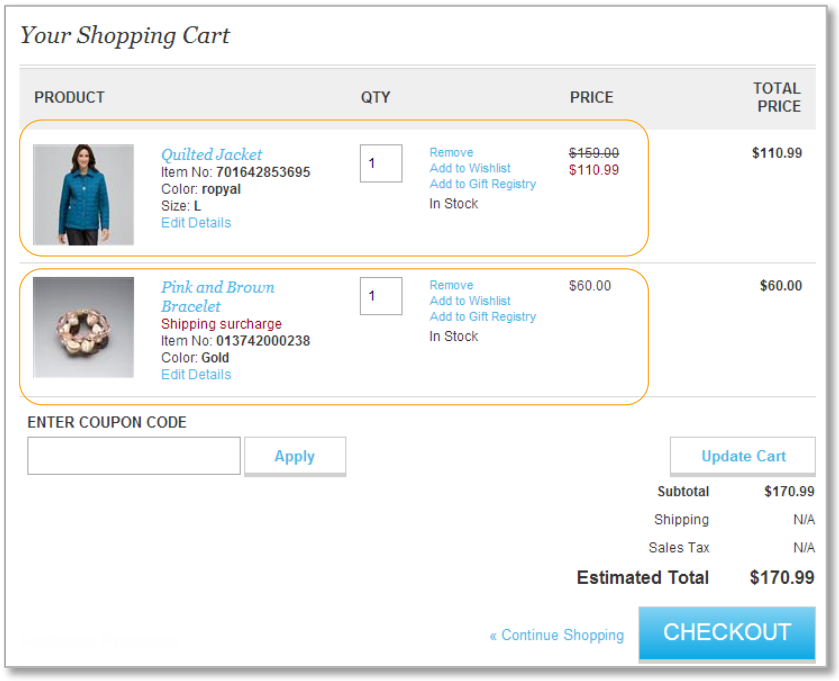
At the end of this module, you will be able to:

- Describe the methods and objects of the `Basket`, `LineItem`, and `LineItemCtnr`, and `Order` classes from the `dw.order` package.
- Apply the APIs in the `dw.order` package to implement new order and line item functionality.

Lesson 4.1: Understanding the `LineItem` Class

A line item is any item included in the calculation of the order total, including products, gift cards, shipping costs, and price adjustments. When customers add items to the cart and then purchase them, an order containing the line items is created.

In the following example, there are two products in the cart. The customer checks out and an order is created with two product line items—the quilted jacket and the pink and brown bracelet.



`LineItem` Subclasses

The following table describes some important subclasses of the `LineItem` class.

Class	Overview
`ProductLineItem`	Represents a specific product included in an order.
`ProductShippingLineItem`	Represents product-level shipping information—the shipping methods and costs applied to the specified product for a given shipping method.
`ShippingLineItem`	Represents shipment-level shipping information—the shipping methods and costs applied to a specified shipment.
`PriceAdjustment`	Represents an adjustment to the price of an order.
`GiftCertificateLineItem`	Represents a gift certificate line item in the cart.

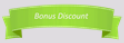
Price Adjustments

Price adjustments can be applied at the product level (applied to a `ProductLineItem`), the shipping level (applied to a `ProductShippingLineItem` or a `ShippingLineItem`), or the order level (applied to a `LineItemCtnr`).

- Price adjustments are generated by the `dw.catalog.PromotionMgr.applyDiscounts()` method and are applied to the order. These adjustments correspond to the promotions that triggered the discounts. If the promotion qualifier is a coupon, a coupon line item is created in the `LineItemCtnr`—the container class for the `LineItem` objects.
- The `dw.campaign.PromotionMgr.applyDiscounts()` method generates adjustments automatically based on promotions. You can create a custom adjustment using the `ProductLineItem.createPriceAdjustment()` method.

Lesson 4.2: Understanding the `LineItemCtnr` Class

The `LineItemCtnr` class is the container object for the `LineItem` class. A `LineItemCtnr` instance contains the line items in an order, which can be of the following types:

	`ProductLineItem`
	`ProductShippingLineItem`
	`ShippingLineItem`
	`GiftCertificateLineItem`
	`CouponLineItem`
	`BonusDiscountLineItem`

The `LineItemCtnr` class provides methods to:

- Create and remove line items
- Create price adjustments
- Create payment instruments
- Retrieve an order's line items
- Retrieve an order's product line items
- Return the item totals
- Return order-level promotions

`LineItemCtnr` Price-Related Methods

The `LineItemCtnr` class also provides price-related methods:

Net-based price methods return the amount for each line item type in the container before tax has been calculated, where a container is a basket or order, for example:

- The `getMerchandizeTotalNetPrice()` method returns the price of all merchandise in the container (the basket or order). The merchandise includes all product line items aside from shipping and price adjustment line items.
- The `getShippingTotalNetPrice()` method returns the price of all shipments in the container.

- The `getGiftCertificateTotalNetPrice()` method returns the price of all gift certificates in the container.

Tax-based price methods represent the amount of tax for each line item type in the container (the basket or order), for example:

- The `getMerchandizeTotalTax()` method returns the total tax for all merchandise in the container. The merchandise includes all product line items aside from shipping and price adjustment line items.
- The `getShippingTotalTax()` method returns the tax applied to all shipments in the container.
- The `getGiftCertificateTotalTax()` method returns the tax applied to all gift certificates in the container.

Gross-based price methods represent the amount for each line item type or for the entire container (the basket or order) after tax has been calculated:

- The `getMerchandizeTotalGrossPrice()` method returns the price of all merchandise in the container, including tax on the merchandise. The merchandise includes all product line items aside from shipping and price adjustment line items.
- The `getShippingTotalGrossPrice()` method returns the price of all shipments in the container, including tax on the shipments in the container.

Additionally, there are methods that aggregate all of the line item values in a line item container (a basket or order). These methods include `getTotalNetPrice()`, `getTotalTax()`, and `getTotalGrossPrice()`. These methods return the totals of all items in the container and include order-level promotions.

Note that all merchandise-related methods do not include gift certificate values in the values they return. Gift certificates are not considered merchandise since they don't represent products.

Basket Class

The `Basket` class is a subclass of the `LineItemCtnr` class. A basket represents a shopping cart. Baskets contain the line items that will become the customer's order. As such, the `Basket` class inherits the methods of the `LineItemCtnr` class. When the customer places an order, the items purchased are removed from the basket.

Lesson 4.3: Understanding the `Order` Class

The `Order` class and its corresponding `OrderMgr` class provide objects and methods for accessing and retrieving information about orders.

The `Order` and `OrderMgr` methods allow you to access orders.

To create an order, you add the `CreateOrder` Demandware pipelet to a pipeline. The `CreateOrder` pipelet creates a preliminary order. The order is preliminary until payment is authorized, then you use the `PlaceOrder` pipelet to create the actual order—after payment has been authorized.

The `Order` class is a child of `LineItemCtnr`, so it inherits all the methods of the `LineItemCtnr` class. You use the `Order` class to access order details, such as customer information, adjusted order totals, and payment.

To query, search for, and process orders, you use the `OrderMgr` class. The `OrderMgr` class provides the `processOrders()` method which processes a set of orders that match a search query. Use this method to process a set of orders efficiently in batch mode.

Knowledge Check

The Knowledge Check answer key is on the following page.

1. Which of the following are subclasses of the `LineItemCtnr` class? *Select all classes that apply*.
   a. `Basket` class
   b. `ProductLineItem` class
   c. `Order` class
   d. `LineItem` class
   e. `ShoppingCart` class

2. The `applyDiscounts` method performs which of the following actions? *Select all items that apply*.
   a. Creates coupon line items if the qualifier is a coupon
   b. Creates product shipping line items if the qualifier is a shipping promotion
   c. Can be called only after the `createPriceAdjustment` method has calculated the price adjustments
   d. Can apply price adjustments at the product level and shipping level, but not the order level
   e. Generates adjustments automatically based on promotions

Knowledge Check Answers

1. Which of the following are subclasses of the `LineItemCtnr` class? *Select all classes that apply*.
   a. `Basket` class
   b. `ProductLineItem` class
   c. `Order` class
   d. `LineItem` class
   e. `ShoppingCart` class

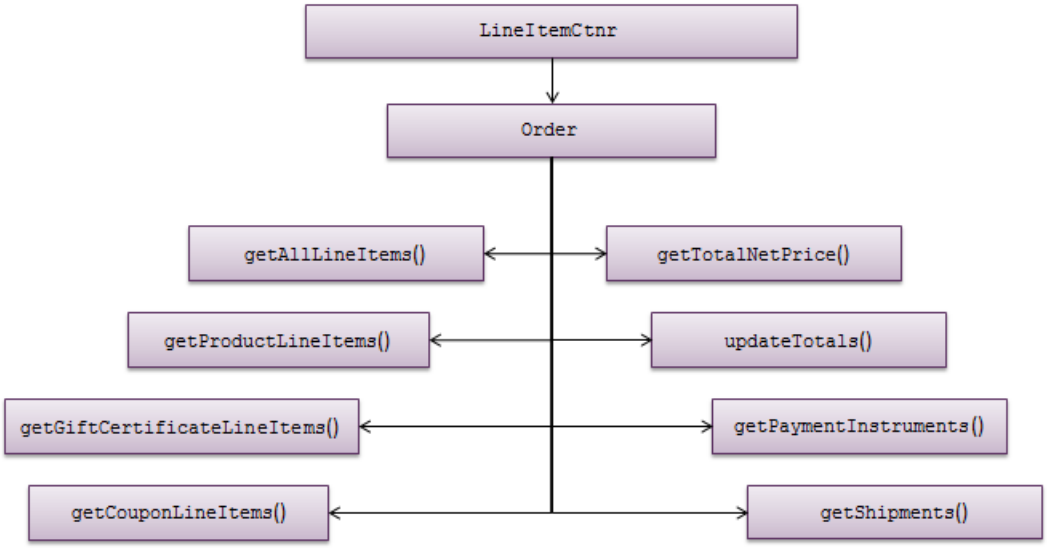
Answer: a, c

2. The `applyDiscounts` method performs which of the following actions? *Select all items that apply*.
   a. Creates coupon line items if the qualifier is a coupon
   b. Creates product shipping line items if the qualifier is a shipping promotion
   c. Can be called only after the `createPriceAdjustment` method has calculated the price adjustments
   d. Can apply price adjustments at the product level and shipping level, but not the order level
   e. Generates adjustments automatically based on promotions

Answer: a, e

Exercise: Explore Orders

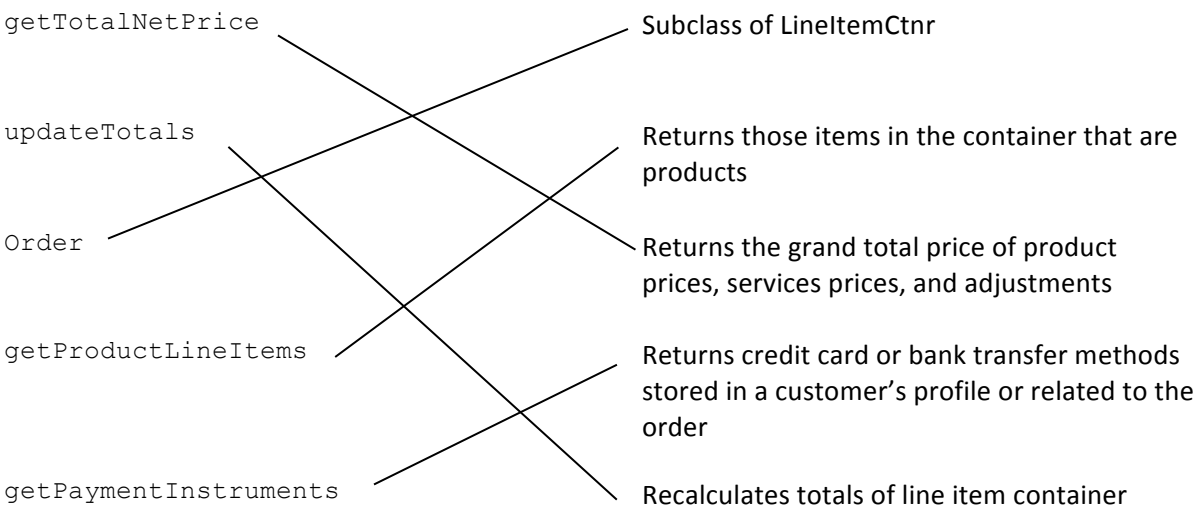
Using the following graph as your guide, click through and review the API documentation (`https://info.demandware.com`). These are the methods and objects you'll use in the exercises that follow.



After reviewing the documentation for the methods and objects in the preceding graph, match the API elements on the left with the definitions on the right. (Answers on following page)

`getTotalNetPrice()`	Subclass of `LineItemCtnr`
`updateTotals()`	Returns items in the container that are products
`Order`	Returns the grand total price of product prices, services prices, and adjustments
`getProductLineItems()`	Returns credit card or bank transfer methods stored in a customer's profile or related to the order
`getPaymentInstruments()`	Recalculates totals of line item container

Exercise Answers: Explore the `LineItemCtnr` Class



Exercise: Order Fill-in-the-Blanks

In this exercise, you'll build a pipeline (`MergeBaskets`) that merges baskets. In this scenario, a customer logs in to the storefront site and adds items into the basket. She gets busy and returns to the site after three hours when the session has already expired. When she comes back, she adds more items to the shopping cart without logging in. When she logs in, the baskets need to be merged. The script you'll implement performs this type of basket merge—it merges the stored basket with the current basket.

To implement the scenario, you're provided with an incomplete script, `mergebaskets.ds`, and a corresponding ISML template, `mergebaskets.isml` (included in the `Working with DW APIs Exercises.zip` file under `Exercise Files/Order Exercises`). You'll fill in the blanks in the script you're given and create a pipeline to run it. Following are descriptions of the APIs you'll use to fill in the blanks.

Script Fill-in-the-Blanks

Fill in the blanks with the APIs needed to complete the `mergebaskets.ds` script you've been given:

1. Code used to `import dw.order package`:

   _____

2. Code used to `get "Basket" from the pdict`:

   _____

3. Code used to `get "StoredBasket" from the pdict`:

   _____

4. Code used to `get all product items from basket in the form of an iterator`:

   _____

5. Code used to `get all product items from storedbasket in the form of an iterator`:

   _____

6. Code used to `loop over basketProductLineItems using a variable called "item"`:

   _____

7. Code used to `remove item from basket (not stored basket)`:

   _____

8. Code used to `loop over storedBasketProductLineItems using a variable called "item2"`:

   _____

9. Code used to `remove item from basket`:

   _____

10. Code used to `update the totals of the basket`:

    _____

Complete the Script: `mergebaskets.ds`

Fill in the blanks in the `mergebaskets.ds` script—denoted by the numbered comments. The `mergebaskets.ds` script gets the current basket and the stored basket and merges these.

The completed script is included in this guide following the example.

```
/**
*   @input Basket : dw.order.Basket
*   @input StoredBasket : dw.order.Basket
*   @output mergedBasket: dw.order.Basket
**/
importPackage( dw.system );
/* 1.Import dw.order package */
importPackage( dw.value );
importPackage( dw.util );
importScript("storefront:checkout/Utils.ds");

function execute( args : PipelineDictionary ) : Number
{
```

```
   var basket : Basket = /* 2.Get "Basket" from pdict */;
   var storedbasket : Basket = /* 3.Get "StoredBasket" from pdict */;
   var basketProductLineItems : Iterator = /* 4.Get all product items from "basket" in the
form of an iterator */;
   var storedBasketProductLineItems : Iterator =  /* 5.Get all product items from
"storedbasket" in the form of an iterator */;
   var validCartProductTime : Number =
200;//Site.getCurrent().preferences.custom.ValidCartProductTime;
   for each (/* 6.Loop over basketProductLineItems using a variable called item */)
   {
         if (checkValidity(item , validCartProductTime )) {
          /* 7.Remove item from basket (not stored basket) */;
         }
   }

    for each (/* 8.Loop over storedBasketProductLineItems using a variable called item2 */)
   {
         if (checkValidity(item2 , validCartProductTime)) {
          /* 9.Remove item from stored basket */;
         } else {
             if(item2.product!=null){

basket.createProductLineItem(item.product,(!empty(item2.optionModel)?item.optionModel:null),
basket.defaultShipment).setQuantityValue(item.quantityValue);
             }
         }
          /* 10.Update the totals of the basket */;
   }

   args.mergedBasket=basket;
   return PIPELET_NEXT;
}

function checkValidity( productLineItem : dw.order.ProductLineItem , validCartProductTime :
Number)  {

           var currentDate : Date = new Date();
           var oldDate : Date = new Date(productLineItem.creationDate);
         if ((currentDate - oldDate)/86400000 >= validCartProductTime){// 86400000
milliseconds = 1 day
          return true;
         }

return false;
}
```

Template: `mergebaskets.isml`

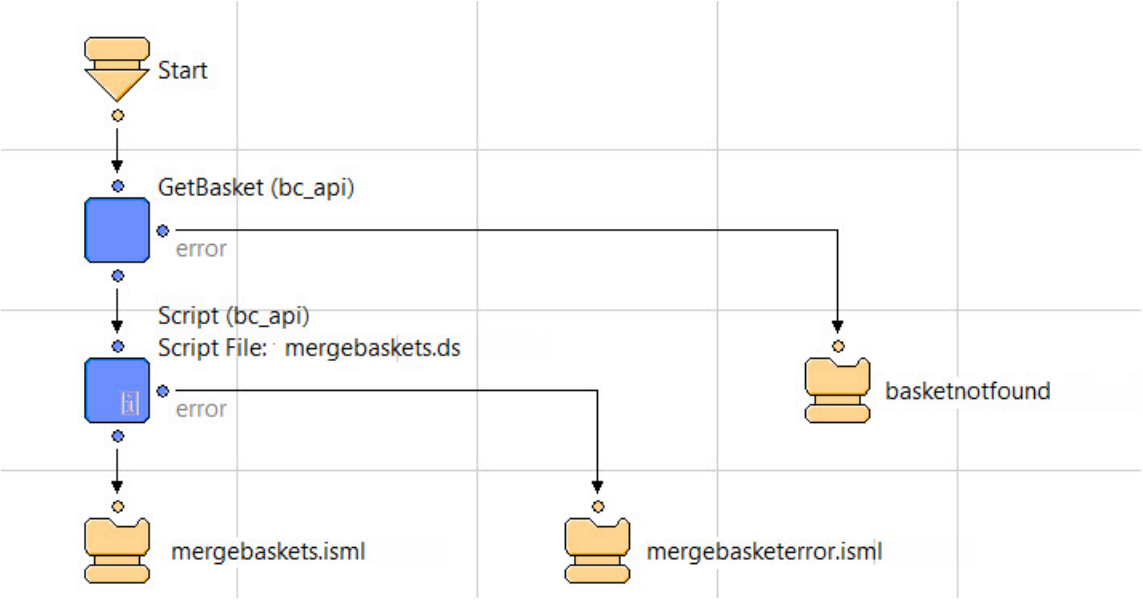
For this example, you're given the completed `mergebaskets.isml` template which displays contents of the saved basket and the merged basket:

```
<!--- TEMPLATENAME: mergebaskets.isml --->
<h1>Printing stored basket first which should now be empty</h1><br>
<isif condition = "${pdict.StoredBasket.productQuantityTotal == 0}">
The Basket is empty
<iselse>
Basket product names are: <br/>
```

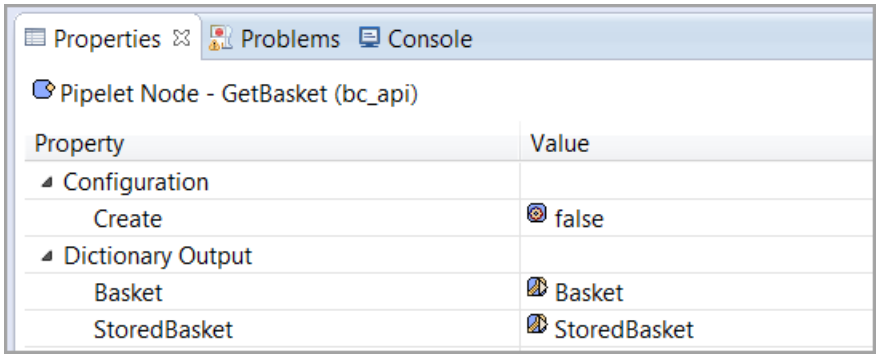
```
<isloop items="${pdict.StoredBasket.getProductLineItems() }" var="productLineItem"
status="loopstate" >
${productLineItem.product.name}<br/>
</isloop>
</isif>

<h1>---------Printing merged basket now-----------</h1><br>
<isif condition = "${pdict.Basket.productQuantityTotal == 0}">
The Basket is empty
<iselse>
Basket product names are: <br/>
<isloop items="${pdict.Basket.getProductLineItems() }" var="productLineItem"
status="loopstate" >

${productLineItem.product.name}<br/>

</isloop>
</isif>

<isset name="Basket" value=null scope="pdict"/>
```

Create the `MergeBaskets` pipeline

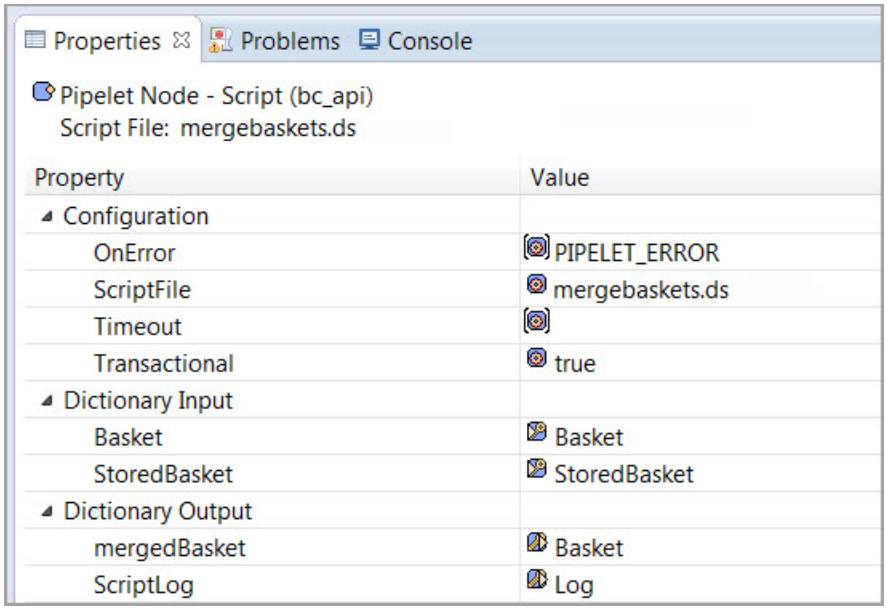
Create the following pipeline which calls the `mergebaskets.ds` script and the corresponding `mergebaskets.isml` template.



You'll set the following properties for the `GetBasket` pipelet:



Configure the properties of the `mergebaskets.ds` script node as follows:



Test your solution

After you fill in the missing methods in the `mergebaskets.ds` script, you'll test your solution by following these steps:

1. Navigate to your storefront and add one or more items to your cart without logging in.
2. Wait for at least 60 minutes.

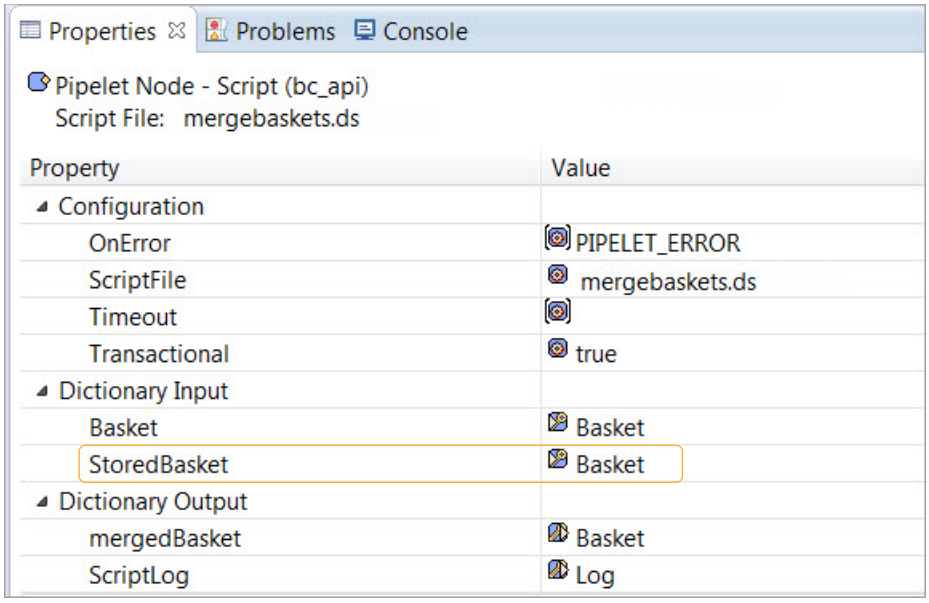
   Waiting for 60 minutes ensures that the current `Basket` gets saved into `StoredBasket`.
3. Log in on the same computer as a user or create an account if you don't have one.
4. Add one or more items to the cart.

5. Call the `MergeBaskets` pipeline:

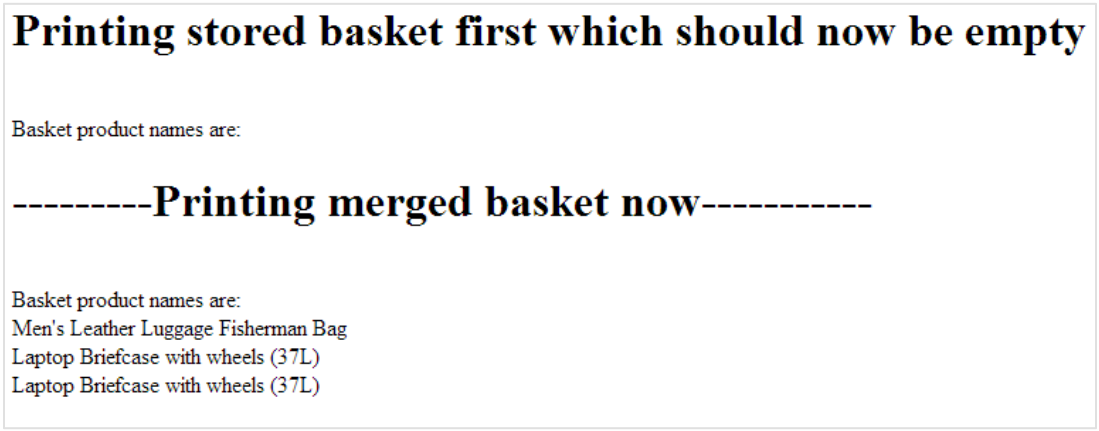
   `http://instance.realm.client.demandware.net/on/demandware.store/Sites-YourSite-Site/default/MergeBaskets-Start`

   The contents of the stored basket are displayed, followed by those of the merged basket.

Note: You can do a quick test of the pipeline without waiting the full 60 minutes for the current basket to be saved into the stored basket. To do so, set the script's `StoredBasket` input to `Basket` rather than `StoredBasket`:



Following is the output when `StoredBasket` is set to `Basket`:



Exercise Answers: MergeBaskets Fill-in-the-Blank

Script Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to `import dw.order package`:
   ```
   importPackage( dw.order );
   ```
2. Code used to `get "Basket" from the pdict`:
   ```
   args.Basket
   ```
3. Code used to `get "StoredBasket" from the pdict`:
   ```
   args.StoredBasket
   ```
4. Code used to `get all product items from basket in the form of an iterator`:
   ```
   basket.getAllProductLineItems().iterator()
   ```
5. Code used to `get all product items from storedbasket in the form of an iterator`:
   ```
   storedbasket.getAllProductLineItems().iterator()
   ```
6. Code used to `loop over basketProductLineItems using a variable called "item"`:
   ```
   var item : ProductLineItem  in basketProductLineItems
   ```
7. Code used to `remove item from basket (not stored basket)`:
   ```
   basket.removeProductLineItem(item)
   ```

8. Code used to loop over storedBasketProductLineItems using a variable called "item2":

   ```
   var item2 : ProductLineItem in storedBasketProductLineItems
   ```

9. Code used to remove item from stored basket:

   ```
   storedbasket.removeProductLineItem(item2)
   ```

10. Code used to update the totals of the basket:

    ```
    basket.updateTotals()
    ```

Script Solution: mergebaskets.ds

```
/**
*   @input Basket : dw.order.Basket
*   @input StoredBasket : dw.order.Basket
*   @output mergedBasket: dw.order.Basket
**/
importPackage( dw.system );
importPackage( dw.order );
importPackage( dw.value );
importPackage( dw.util );
importScript("storefront:checkout/Utils.ds");

function execute( args : PipelineDictionary ) : Number
{

   var basket : Basket = args.Basket;
   var storedbasket : Basket = args.StoredBasket;
   var basketProductLineItems : Iterator = basket.getAllProductLineItems().iterator();
   var storedBasketProductLineItems : Iterator =
storedbasket.getAllProductLineItems().iterator();
   var validCartProductTime : Number =
200;//Site.getCurrent().preferences.custom.ValidCartProductTime;
   for each (var item : ProductLineItem  in basketProductLineItems)
   {
         if (checkValidity(item , validCartProductTime )) {
          basket.removeProductLineItem(item);
         }
   }
    for each (var item2 : ProductLineItem in storedBasketProductLineItems)
   {
         if (checkValidity(item2 , validCartProductTime)) {
          storedbasket.removeProductLineItem(item2);
         } else {
             if(item2.product!=null){
basket.createProductLineItem(item2.product,(!empty(item2.optionModel)?item2.optionModel:null),
basket.defaultShipment).setQuantityValue(item2.quantityValue);
             }
         }
        basket.updateTotals();
   }

   args.mergedBasket=basket;
   return PIPELET_NEXT;
}
```

```
function checkValidity( productLineItem : dw.order.ProductLineItem , validCartProductTime :
Number)  {

           var currentDate : Date = new Date();
           var oldDate : Date = new Date(productLineItem.creationDate);
         if ((currentDate - oldDate)/86400000 >= validCartProductTime){// 86400000
milliseconds = 1 day
          return true;
         }
return false;
}
```

Exercise: Calculate Shipping Cost Fill-in-the-Blanks

In this exercise, you'll build a pipeline (`CalculateShippingCost`) that calculates the cost of shipping for the items in the basket.

To implement the scenario, you're provided with an incomplete script, `calculateshippingcost.ds`, and a complete ISML template that displays the results, `calculateshippingcost.isml` (included in the `Working with DW APIs Exercises.zip` file under `Exercise Files/Order Exercises`). You'll fill in the blanks in the script you're given and create a pipeline to run it. Following are descriptions of the APIs you'll use to fill in the blanks.

Script Fill-in-the-Blanks

Fill in the blanks with the APIs needed to complete the `calculateshippingcost.ds` script you've been given:

1. Code used to `set to type Money from dw.value`:

   _____

2. Code used to `get "Basket" from the pdict`:

   _____

3. Code used to `iterate through the shipments in basktContainer`:

   _____

4. Code used to `shippingMethodName on pdict`:

   _____

5. Code used to set the `shippingCost on pdict to the total shipping price of basktcontainer`:

   _____

Complete the Script: `calculateshippingcost.ds`

Fill in the blanks in the `calculateshippingcost.ds` script—denoted by the numbered comments in the script. The `calculateshippingcost.ds` script calculates the cost of shipping the items in the basket.

The completed script is included in this guide following the example.

```
/**
*
*
*    @input Product:dw.catalog.Product
*    @input Basket:dw.order.Basket
*    @output shippingCost : dw.value.Money
*    @output shippingMethodName:String
*
*/
importPackage( dw.order ); // Defines ShippingMgr
importPackage(dw.catalog);
importPackage(dw.util);
function execute( args : PipelineDictionary ) : Number
{
    var shippingCost : /* 1.Set to type Money from dw.value */;

    var basktContainer:LineItemCtnr=/* 2.Get "Basket" from the pdict */;

    var items:Iterator=/* 3.Iterate through the shipments in basktContainer */;
    while(items.hasNext()){
        var x:Shipment = items.next();
        /* 4.Set shippingMethodName on pdict */=x.shippingMethod.displayName;
        /* 5.Set shippingCost on pdict to the total shipping price of basktContainer */;
    }
   return PIPELET_NEXT;

}
```

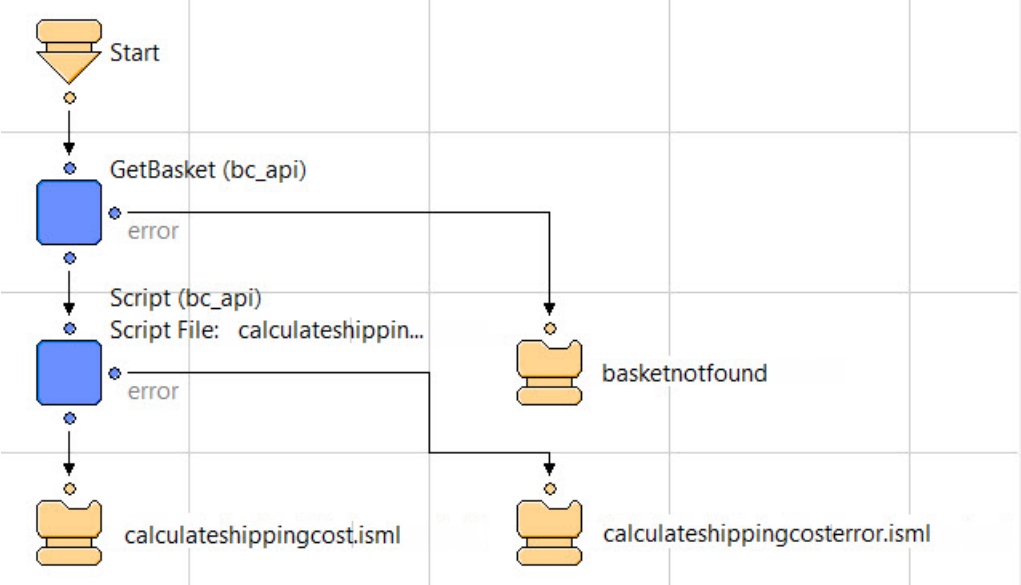
Template: `calculateshippingcost.isml`

For this example, you're given the following completed `calculateshippingcost.isml` template which displays the shipping cost for the items in the basket:

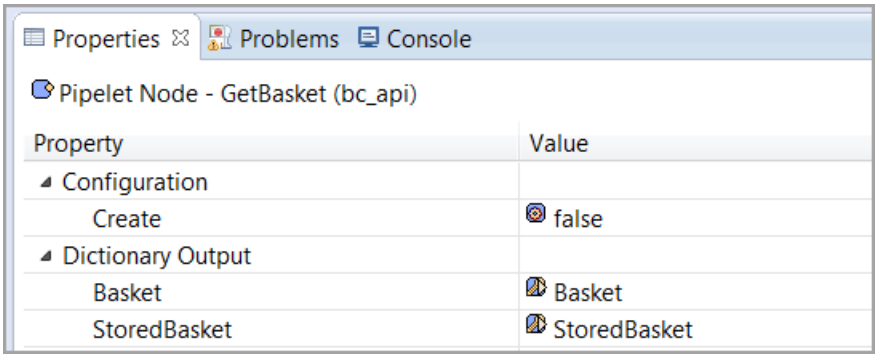
```
<!--- TEMPLATENAME: basketCost.isml --->

Shipping Method: ${pdict.shippingMethodName}
<br>
Basket shipping cost is ${pdict.shippingCost}
```

Create the `CalculateShippingCost` pipeline

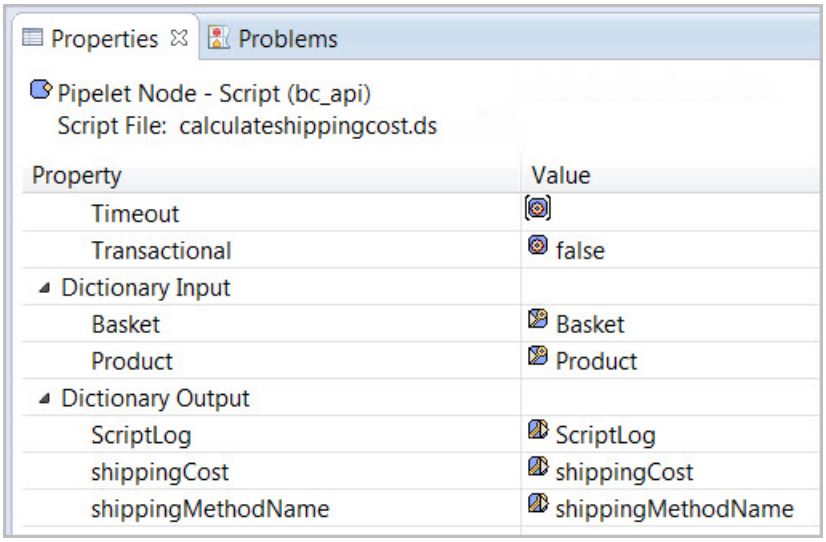
Create the following pipeline which calls the `calculateshippingcost.ds` script and the corresponding `calculateshippingcost.isml` template.



You'll set the following properties for the `GetBasket` pipelet:



Configure the properties of the `calculateshippingcost.ds` script node as follows:



Test your solution

After you fill in the missing methods in the following script, you'll test your solution by following these steps:

1. Navigate to your storefront and add one or more products to your cart.
2. Sign in to your account or create an account if you don't have one.
3. Update your profile and ensure that you've entered a zip code.

   The user must have a zip code in order for the storefront site to calculate shipping costs.
4. Click **CHECKOUT**.

   You need to start the checkout process in order for the storefront site to calculate the shipping costs for the items in the cart.
5. Execute your pipeline:

   `http://instance.realm.client.demandware.net/on/demandware.store/Sites-YourSite-Site/default/CalculateShippingCost-Start`

   The shipping method and shipping cost for the items in the basket are displayed, for example:

   Shipping Method: Ground
   Basket shipping cost is USD 9.99

Exercise Answers: Calculate Shipping Cost Fill-in-the-Blank

Script Fill-in-the-Blanks

Fill in the blanks below with the code you used to complete the script above:

1. Code used to set to type Money from dw.value:
   `dw.value.Money`
2. Code used to get "Basket" from the pdict:
   `args.Basket`
3. Code used to iterate through the shipments in basktContainer:
   `basktContainer.shipments.iterator()`
4. Code used to set shippingMethodName on pdict:
   `args.shippingMethodName`
5. Code used to set the shippingCost on pdict:
   `args.shippingCost`
6. Code used to set shippingCost on pdict to the total shipping price of basktContainer:
   `args.shippingCost=basktContainer.shippingTotalPrice`

Script Solution: calculateshippingcost.ds

```
/**
*
*
*   @input Product:dw.catalog.Product
*   @input Basket:dw.order.Basket
*   @output shippingCost : dw.value.Money
*   @output shippingMethodName:String
*
*/
importPackage( dw.order ); // Defines ShippingMgr
importPackage(dw.catalog);
importPackage(dw.util);
function execute( args : PipelineDictionary ) : Number
{
    var shippingCost : dw.value.Money;
    var basktContainer:LineItemCtnr=args.Basket;
    var items:Iterator=basktContainer.shipments.iterator();
    while(items.hasNext()){
       var x:Shipment = items.next();
       args.shippingMethodName=x.shippingMethod.displayName;
       args.shippingCost=basktContainer.shippingTotalPrice;
      }
    return PIPELET_NEXT;
}
```

Exercise: Order Scenarios

In this activity, you're provided with scenarios for managing orders. Specify the classes and methods from the `dw.order` package that you'd use to implement the scenario.

1. Which class and method from the `dw.order` package would you use to find all orders generated between July 3 and July 10?

   _____

2. Which class and method from the `dw.order` package would you use to retrieve the email address of the customer associated with an order?

   _____

3. Which class and method from the `dw.order` package would you use to recalculate the order totals?

   _____

4. Which class and method from the `dw.order` package would you use to create a custom price adjustment?

   _____

5. Which class and method from the `dw.order` package would you use to return the default shipping method for the site?

   _____

Exercise Answers: Order Scenarios

In this activity, you're provided with scenarios for managing orders. Specify the classes and methods from the `dw.order` package that you'd use to implement the scenario.

1. Which class and method from the `dw.order` package would you use to find all orders generated between July 3 and July 10?

   ```
   OrderMgr.queryOrder()
   ```

2. Which class and method from the `dw.order` package would you use to retrieve the email address of the customer associated with an order?

   ```
   LineItemCtnr.getCustomerEmail()
   ```

3. Which class and method from the `dw.order` package would you use to recalculate the order totals?

   ```
   LineItemCtnr.updateTotal()
   ```

4. Which class and method from the `dw.order` package would you use to create a custom price adjustment?

   ```
   ProductLineItem.createPriceAdjustment()
   ```

5. Which class and method from the `dw.order` package would you use to return the default shipping method for the site.

   ```
   getDefaultShippingMethod()
   ```

Congratulations

Congratulations on completing the *Working with the Demandware APIs* course! For more information about the APIs, see the documentation at info.demandware.com.