

Getting started with the Rational Rhapsody API

Last update: February 2015

© Copyright IBM® Corporation 2000, 2015.

Table of Contents

| | |
|---|----|
| Where do I start ?..... | 3 |
| What is the main object I have to create in order to access all other objects in the API ?..... | 4 |
| So I guess that the next step should be to create a Rhapsody project or open an existing Rhapsody project ?..... | 4 |
| Now that I have a project, I can begin adding elements to the model?..... | 4 |
| What if I want to use an existing package in a project?..... | 5 |
| OK – I used findNestedElement to get the package/class/other element that I want to add new elements to. What methods do I use to add new elements ?..... | 5 |
| What if I want to perform an action on a group of elements, for example, all classes in a package or all diagrams in my model?..... | 7 |
| Are there any inheritance relationships between the various interfaces? If so, is there a diagram anywhere of these relationships?..... | 8 |
| How about properties? Rhapsody has thousands of them – can I use the Rhapsody API to modify the value of specific properties?..... | 9 |
| What other actions can be performed on a collection?..... | 10 |
| Can I use the API to add diagrams to my model?..... | 14 |
| OK, now that I have added a diagram, how can I add elements to the diagram?..... | 14 |
| Does the API also provide methods for generating code?..... | 17 |
| OK, I've generated code for the model. Are there also API methods for building and running the application?..... | 18 |
| What about methods for saving a model after changes were made to it?..... | 19 |
| How about an example of creating a statechart?..... | 21 |
| What about extracting information from a model, for example, generating a report or exporting diagrams as graphic files?..... | 26 |
| Can I see some sample code that shows some of the things you can do with classes and their attributes and operations?..... | 29 |

Where do I start ?

OK, so you've decided that the Rhapsody API may be useful for you and you want to get started with the Java version of the API. First thing you've got to do is set up a Java project in your IDE and define references to the necessary libraries.

The steps below assume you are using Eclipse as your code editor. If you're using an editor that does a little less spoon-feeding than Eclipse, we'll assume you know what you have to do to adapt these instructions to your coding environment.

1. Create a new Java project.
2. Select the new project and open the popup menu – select Properties.
3. Open the Java Build Path set of tabs.
4. On the Libraries tab, choose Add External JARs...
5. Select the rhapsody.jar file, located in [rhapsody installation]/share/javaapi.
6. Once rhapsody.jar appears in the list, use the + sign to expand the elements below it.
7. Select Native Library Location and set it to point to [rhapsody installation]/share/javaapi, which is the directory that contains the rhapsody.dll library. (If you are working on Linux, the library file is named rhapsody.so)

If you are using the 32-bit version of Rational Rhapsody, but are using the Rhapsody API for applications that will be run with a 64-bit version of Java, point to the directory [Rational Rhapsody installation directory]Share/JavaAPI/64Bit, which contains a 64-bit version of rhapsody.dll.

Similarly, if you are using the 64-bit version of Rational Rhapsody, but are using the Rhapsody API for applications that will be run with a 32-bit version of Java, point to the directory [Rational Rhapsody installation directory]Share/JavaAPI/32Bit, which contains a 32-bit version of rhapsody.dll.

To shorten this process for future Rhapsody API projects, you may want to use the Eclipse option of defining a user library for these files.

That's it – you're good to go. As soon as you start writing code in your project, you should have access to code completion for the Rhapsody API, and any other goodies that Eclipse provides such as tooltips for displaying the Javadoc documentation.

What is the main object I have to create in order to access all other objects in the API ?

The first thing you have to do in your java file is create an IRPApplication object, which represents Rhapsody. So any application you write will include a line like the following:

```
static IRPApplication app =  
RhapsodyAppServer.getActiveRhapsodyApplication();
```

If you use this line, you can then type a period after `app` in your next line of code and you'll see the methods that are now available to you.

So I guess that the next step should be to create a Rhapsody project or open an existing Rhapsody project ?

Right. So you'll want to use one of the following IRPApplication methods:

- `createNewProject`
- `openProject`
- `activeProject`

```
app.createNewProject("1:\\temp\\_hello_world", "Hello_World");  
app.openProject("1:\\temp\\hello_world\\Hello_World.rpy");  
app.activeProject();
```

When creating a new project, you have to provide the path where the project should be created, and the name to use for the project name.

When opening an existing project, you have to provide the full path for the relevant .rpy file.

The `activeProject` method returns the project currently open in Rhapsody.

Now that I have a project, I can begin adding elements to the model?

Yes. Since most elements are added to packages, the next step should be to create a new package.

```
IRPPProject prj = app.activeProject();  
IRPPackage pkg = prj.addPackage("GreeterPackage");
```

The `addPackage` method returns the `IRPPackage` object that was created.

What if I want to use an existing package in a project?

Rhapsody has two generic methods that are used to get model elements, such as a package, so that you can work with them.

- `findNestedElement`
- `findNestedElementRecursive`

Both of these methods belong to `IRPModelElement`.

The difference between the two methods is `findNestedElement` only searches the first level of elements below the current element, while `findNestedElementRecursive` searches all the way down to the lowest hierarchical level.

Both of these methods take two `String` arguments – the first is the name of the element you are looking for, while the second is the type of element, such as `Package`.

So to get an existing package, you'll use a line like this:

```
IRPPackage packageToUse =  
(IRPPackage)prj.findNestedElement("GreeterPackage", "Package");
```

Note that the object returned was cast as an `IRPPackage` object. This is necessary because the method is a generic one and therefore always returns an object of type `IRPModelElement`.

There are certain inheritance relationships between the different types of elements that can be defined in Rhapsody. We'll discuss this at a later point. For now, we'll just mention that `IRPModelElement` is the base for most of the elements available in Rhapsody.

OK – I used `findNestedElement` to get the package/class/other element that I want to add new elements to. What methods do I use to add new elements ?

To add new elements, you can use:

- specific “add” methods that add elements of a specific type, such as `IRPPProject.addPackage`, or `IRPPackage.addClass`
- the generic “add” method, `addNewAggr`. This method takes two string parameters – the first is the type of elements you want to add, and the second string is the name to use for the new element.

A few things to keep in mind when using these “add” methods:

- The `addNewAggr` method belongs to `IRPModelElement` so it can be used to add any type of element to any type of element that can logically contain it.
- If you use the generic `addNewAggr` method to attempt to add an element to a parent element that cannot contain it, for example, adding a class to an attribute, an exception will be thrown.

- The specific “add” methods return the type of object that was added, for example, IRPPackage.addClass returns an IRPClass object, but the generic addNewAggr method always returns an object of type IRPModelElement. This means that you have to cast the returned element.

Some examples:

```
//find existing package and add class to it
IRPPProject prj = app.activeProject();

IRPPackage packageToUse =
(IRPPackage)prj.findNestedElement("GreeterPackage", "Package");

IRPClass writerClass = packageToUse.addClass("TextWriter");


// create new package and add a class to it
IRPPProject prj = app.activeProject();

IRPPackage pkg = prj.addPackage("GreeterPackage");

IRPClass composerClass =
(IRPClass)pkg.addNewAggr("Class", "TextComposer");
```

What if I want to perform an action on a group of elements, for example, all classes in a package or all diagrams in my model?

The API includes an interface called `IRPCollection` that represents a collection of model elements. This interface includes methods that can help you iterate through the collection and perform an action on one or more elements in the collection, for example, `getCount()` and `getItem(int index)`.

There are a number of methods that return `IRPCollection` objects. These include:

- methods for returning collections of specific types of model elements, for example, `getPorts()`, and `getEvents()`.
- generic methods that return "mixed" collections, for example, `getNestedElements()` and `getNestedElementsRecursive()`. `getNestedElements()` returns a collection of all the elements in the first level below the current element, while `getNestedElementsRecursive()` returns all the contained elements down to the lowest hierarchical level.

The following code is a simple example of performing an action on a collection of items, in this case all elements that make up the model.

```
Map<String,String> theMap = new HashMap();

IRPCollection allColl = prj.getNestedElementsRecursive();

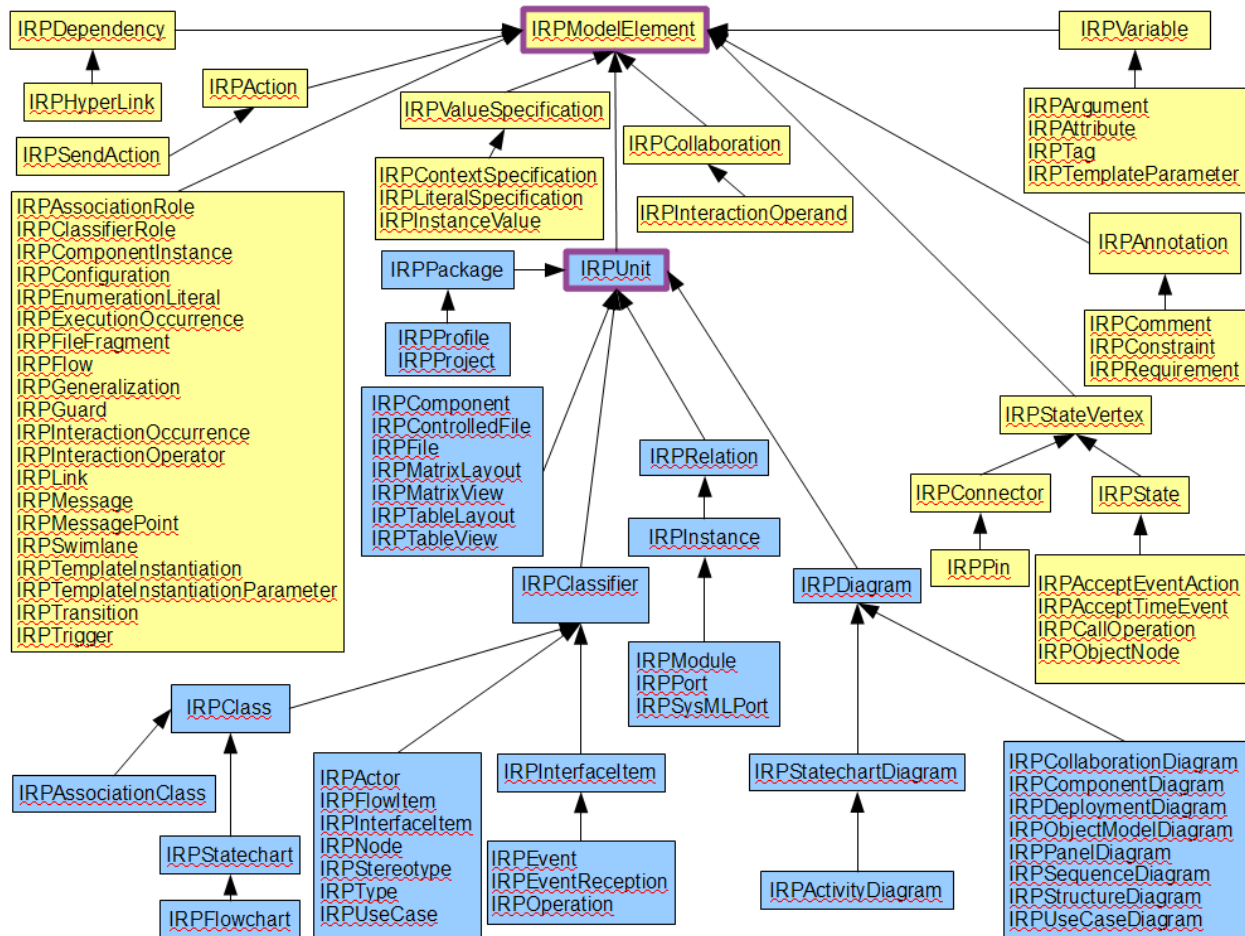
for (int i = 1; i <= allColl.getCount(); i++) {
    IRPModelElement element = (IRPModelElement)allColl.getItem(i);
    theMap.put(element.getMetaClass(), element.getInterfaceName());
}

Set<String> mapKeySet = theMap.keySet();

for (String strMetaClass : mapKeySet) {
    System.out.println(strMetaClass + "\t" + theMap.get(strMetaClass));
}
```

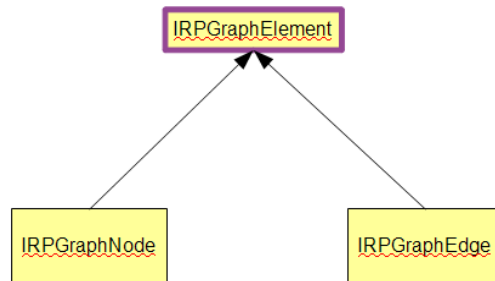
Are there any inheritance relationships between the various interfaces? If so, is there a diagram anywhere of these relationships?

The following diagrams illustrate the relationships between the different interfaces in the Rhapsody API.



Just a few points to help clarify the diagram:

- All the interfaces in the diagram extend IRPModelElement, directly or via IRPUnit which is derived from IRPModelElement.
- IRPUnit contains methods used for dealing with Rhapsody units, such as saving and loading. So the interfaces that extend IRPUnit are those that represent model elements that Rhapsody allows you to save as individual files.
- IRPStatechartDiagram represents the actual statechart diagram, while IRPStatechart represents the concept reflected by the diagram.



How about properties? Rhapsody has thousands of them – can I use the Rhapsody API to modify the value of specific properties?

You can get or set the value of a property for a model element using the following IRPModelElement methods:

- `getPropertyValue`
- `getPropertyValueExplicit`
- `setPropertyValue`

The get methods take a single parameter—the name of the property. In this context, the full name of the property must be provided, using a period as the delimiter, for example, `CPP_CG.Attribute.AccessorGenerate`

The `getPropertyValueExplicit` method only returns the value of the property if the default value has been overridden. If the value of the property is the default value for the property, this method throws an exception

The set method takes two String parameters. The first parameter is the name of the property (full name, with period as delimiter), while the second is the value to use for the property.

The following sample code demonstrates use of the get and set methods for properties:

```
//open existing project
app.openProject("l:\\temp\\hello_world\\Hello_World.rpy");
IRPPProject prj = app.activeProject();
IRPPPackage packageToUse = prj.addPackage("GreeterPackage");
IRPClass writerClass = packageToUse.addClass("TextWriter");
String currentPropertyValue =
writerClass.getPropertyValue("CPP_CG.Class.GenerateDestructor");
```

```

System.out.print("At the beginning, the property has its default value
which is\n " + "\t" + currentPropertyValue + "\n\n");

String newPropertyValue = "False";

writerClass.setPropertyValue("CPP_CG.Class.GenerateDestructor",
newPropertyValue);

currentPropertyValue =
writerClass.getPropertyValue("CPP_CG.Class.GenerateDestructor");

System.out.print("After our call to the set function, the value of the
property is\n " + "\t" + currentPropertyValue + "\n\n");

currentPropertyValue =
writerClass.getPropertyValueExplicit("CPP_CG.Class.GenerateDestructor");

System.out.print("Because we modified the default value, the method
getPropertyValueExplicit returns the value we provided, which is\n " +
"\t"
+ currentPropertyValue + "\nIf we had not modified the default value, it
would have thrown an exception.");

```

What other actions can be performed on a collection?

The following sample code demonstrates all of the IRPCollection methods. The following points will help you understand the code:

- There are two ways to add new items to a collection: You can use the addItem method to tack the specified element to the end of the collection. However, if you are adding a number of items, it may be more efficient to call setSize to set the new size of the collection and then call setModelElement to place elements in specific locations in the collection.
- When using the getItem method or setModelElement method, the index parameter is based on an index value of 1 for the first element (not 0).

```

package documentation.rhapsodyapi;

import java.util.List;

import com.telelogic.rhapsody.core.*;

public class CollectionHandler {

    static IRPApplication app =
RhapsodyAppServer.getActiveRhapsodyApplication();

    // main method

    public static void main(String[] args) {

        if(app != null)

```

```

        doCollectionTricks();
    }

    // this method demonstrates IRPCollection's methods
    private static void doCollectionTricks() {

        // create and open project
        app.createNewProject("1:\\temp\\_sample_code",
"Collection_Tricks");
        IRPProject prj =
app.openProject("1:\\temp\\_sample_code\\Collection_Tricks.rpy");

        // create packages and classes
        IRPPackage vehiclePackage = prj.addPackage("Vehicles");
        IRPClass carClass = vehiclePackage.addClass("Car");
        IRPClass jeepClass = vehiclePackage.addClass("Jeep");
        IRPClass convertibleClass =
vehiclePackage.addClass("Convertible");
        IRPClass busClass = vehiclePackage.addClass("Bus");
        IRPClass truckClass = vehiclePackage.addClass("Truck");
        IRPPackage airVehiclePackage =
prj.addPackage("Air_Vehicles");
        airVehiclePackage.addClass("Helicopter");

        // add inheritance relationship
        convertibleClass.addGeneralization(carClass);

        prj.save();

        // create collection of all classes in the Vehicles package
        and then display contents of collection
        IRPCollection classCollection = vehiclePackage.getClasses();
        List currentContentsOfCollection = classCollection.toList();
        int numberOfItemsInList = currentContentsOfCollection.size();
    }
}

```

```

        IRPClass tempClass = null;

        System.out.println("The collection currently contains the
following " + classCollection.getCount() + " items:");

        // this block uses the get method of List so index starts at 0;
note that for the get method of IRPCollection

            // (used later on), the index starts at 1

        for(int i = 0; i < numberOfItemsInList ; i++) {

            tempClass = (IRPClass)(currentContentsOfCollection.get(i));

            System.out.println("    " + tempClass.getDisplayName());

        }

        // now we'll empty out the collection

        classCollection.empty();

        System.out.println("After calling the empty method, the
collection now contains " + classCollection.getCount() + " items.");

        // note that addItem increases the size by one each time - to
add multiple elements, it's preferable to

            // call setSize and then use setModelElement to place
elements in an open spot

        classCollection.addItem(carClass);

        classCollection.addItem(jeepClass);

        System.out.println("After adding classes, the collection now
contains the following " + classCollection.getCount() + " items:"

            + "\n    " +
((IRPClass)classCollection.getItem(1)).getDisplayName() + "\n    "

            +
((IRPClass)classCollection.getItem(2)).getDisplayName());

        // now extend size of collection and add additional classes

        classCollection.setSize(5);

        // important: note that index count begins at one

        classCollection.setModelElement(3, busClass);

        classCollection.setModelElement(4, convertibleClass);

        classCollection.setModelElement(5, truckClass);

        numberOfItemsInList = classCollection.getCount();

        System.out.println("After adding more classes, the collection

```

```

now contains the following " + classCollection.getCount() + " items:");
    for(int i = 1; i < numberOfItemsInList+1 ; i++) {
        tempClass = (IRPClass)(classCollection.getItem(i));
        System.out.println("    " + tempClass.getDisplayName()
+ " at position " + (i));
    }

    // this section creates a new class diagram and populates it.
The populateDiagram method takes 3 parameters,

        // the first two being collections: a collection of
model elements and a collection of strings

        IRPDDiagram classDiagramToCreate =
vehiclePackage.addObjectModelDiagram("Classes in Vehicles package");

        IRPCollection classesToAddToDiagram =
vehiclePackage.getClasses();

        IRPCollection typesOfRelationsToShow =
app.createNewCollection();

        typesOfRelationsToShow.setSize(2);

        typesOfRelationsToShow.setString(1, "Inheritance");

        typesOfRelationsToShow.setString(2, "Dependency");

        classDiagramToCreate.populateDiagram(classesToAddToDiagram,
typesOfRelationsToShow, "fromto");

    } //end of method doCollectionTricks
}

```

Can I use the API to add diagrams to my model?

The IRPPackage interface contains methods for adding the following types of diagrams:

- activity diagrams
- collaboration diagrams
- component diagrams
- deployment diagrams
- object model diagrams
- sequence diagrams
- use case diagrams

You can also use IRPModelElement's generic addNewAggr method to add a diagram to your model.

To add a statechart or activity diagram to a class, however, the following methods must be used:

- IRPClassifier.addActivityDiagram
- IRPClassifier.addStatechart

All of the diagram types extend the interface IRPDiagram, which provides methods for basic diagram-related actions, such as:

- adding a new element or an existing element to the diagram
- drawing the relations between elements on a diagram
- returning a collection of all elements contained in the diagram (or certain subsets of elements)
- exporting the diagram in different graphic formats

OK, now that I have added a diagram, how can I add elements to the diagram?

While there are a number of methods for adding model elements to a diagram, the simplest approach is to use IRPDiagram.populateDiagram.

populateDiagram takes the following three arguments:

- an IRPCollection object consisting of the model elements you would like to place on the diagram
- a second IRPCollection object consisting of the types of relations that you would like to have displayed on the diagram

- a String argument that represents which related elements you would like to have included on the diagram, in addition to those you specified directly. This argument can take any of the following values: among, from, to, fromto.

The following code creates three classes, two of which are derived from the third class. It then creates an object model diagram and places the three classes on the diagram, showing the inheritance relationships.

```
app.openProject("1:\\temp\\_sample_diagrams\\Diagram_sample.rpy");
IRPPProject prj = app.activeProject();
IRPPackage vehiclePackage = prj.addPackage("Vehicles");
IRPClass car = vehiclePackage.addClass("Car");
IRPClass jeep = vehiclePackage.addClass("Jeep");
IRPClass convertible = vehiclePackage.addClass("Convertible");
jeep.addGeneralization(car);
convertible.addGeneralization(car);

IRPDDiagram vehicleOmd =
vehiclePackage.addObjectModelDiagram("vehicle_omd");

IRPCollection vehicleClasses = vehiclePackage.getClasses();
IRPCollection relationTypes = app.createNewCollection();
relationTypes.setSize(1);
relationTypes.setString(1, "Inheritance");
vehicleOmd.populateDiagram(vehicleClasses, relationTypes, "fromto");
```

The next code snippet is similar to the first but demonstrates the use of “fromto” as the third argument. In this case, one of the derived classes serves as a base class for a fourth class. Even though this fourth class is not specified to be added to the diagram, it is included in the diagram because it has a relation with one of the other three classes. This may seem a little superfluous in this example, but the “fromto” and other values for the third argument are useful in larger models where you may not be aware of all the relations. The lines of code that were added to the previous sample are highlighted.

```
app.openProject("1:\\temp\\_sample_diagrams\\Diagram_sample.rpy");
IRPPProject prj = app.activeProject();
IRPPackage vehiclePackage = prj.addPackage("Vehicles");
IRPClass car = vehiclePackage.addClass("Car");
IRPClass jeep = vehiclePackage.addClass("Jeep");
IRPClass convertible = vehiclePackage.addClass("Convertible");
IRPClass coolJeep = vehiclePackage.addClass("CoolJeep");
```

```
jeep.addGeneralization(car);
convertible.addGeneralization(car);
coolJeep.addGeneralization(jeep);

IRPDiagram vehicleOmd =
vehiclePackage.addObjectModelDiagram("vehicle_omd");

IRPCollection vehicleClasses = vehiclePackage.getClasses();
IRPCollection relationTypes = app.createNewCollection();
relationTypes.setSize(1);
relationTypes.setString(1, "Inheritance");
vehicleOmd.populateDiagram(vehicleClasses, relationTypes, "fromto");
```


Does the API also provide methods for generating code?

The IRPApplication interface contains the following methods for code generation:

- generate()
generates code for the active configuration
- generateElements(IRPCollection)
generates code for the specified elements
- generateEntireProject()
generates code for the entire project
- generateMainAndMakeFiles()
generates only the main file and makefiles
- generateWithDependencies()
generates code for the active configuration and for any component on which the active component depends

There are also regenerate methods corresponding to each of these methods, except for generateMainAndMakeFiles.

The behavior of the regenerate methods is identical to that of the regenerate options in the GUI, meaning they will always go through the code generation process even if Rhapsody has not detected any changes to the elements, however, they will not overwrite the source code file if no changes were made.

In the sample code below, three classes are created. At first, code is generated only for two of the classes. The second code generation call generates code for all elements in the active configuration. If you pause execution after the first code generation method call, you will see that no code was generated for the Jeep class.

```
app.openProject("1:\\temp\\_sample_diagrams\\Diagram_sample.rpy");
IRPPProject prj = app.activeProject();
IRPPackage vehiclePackage = prj.addPackage("Vehicles");
IRPClass car = vehiclePackage.addClass("Car");
IRPClass jeep = vehiclePackage.addClass("Jeep");
IRPClass convertible = vehiclePackage.addClass("Convertible");
jeep.addGeneralization(car);
convertible.addGeneralization(car);
IRPCollection classesToGenerate = app.createNewCollection();
classesToGenerate.addItem(car);
classesToGenerate.addItem(convertible);
app.generateElements(classesToGenerate);
// now generate for all elements in active configuration
```

```
app.generate();
```

The generate() method generates code for the active configuration. Use the IRPApplication methods setComponent and setConfiguration to choose the relevant configuration before calling the generate() method.

OK, I've generated code for the model. Are there also API methods for building and running the application?

Before building the application, use the following methods to specify how the application should be built:

- IRPComponent.setBuildType(String)
The possible argument values are "executable" and "library".
- IRPConfiguration.setBuildSet(String)
The possible argument values are "debug" and "release".

The following IRPApplication methods can be used to build/rebuild an application:

- build()
- buildEntireProject()
- buildWithDependencies()
- rebuild()
- rebuildEntireProject
- rebuildWithDependencies()

You can then run the application by calling IRPApplication.runApplication().

The following sample code creates a single class with a single operation that displays some text. It then generates the code, builds the application, and runs the application.

A few things to note:

- Rhapsody automatically creates a file containing a main method
- The IRPConfiguration.addInitialInstance method is used to create objects from within the main function.
- The IRPConfiguration.setInitializationCode method is used to add any additional code to the main function.

```
app.openProject("1:\\temp\\_sample_diagrams\\Diagram_sample.rpy");  
IRPPProject prj = app.activeProject();  
IRPPPackage helloWorldPackage = prj.addPackage("HelloWorld");  
IRPClass writer = helloWorldPackage.addClass("Writer");  
IRPOperation displayTextOp = writer.addOperation("displayText");  
String bodyContent = "System.out.println(\"Hello World from Rational
```

```

Rhapsody\");";

displayTextOp.setBody(bodyContent);

// create initial instance of Writer class

IRPConfiguration activeCfg = prj.getActiveConfiguration();

activeCfg.addInitialInstance(writer);

// add initialization code

String initCode = "p_Writer.displayText();"; // when initial instance of
an object is created, p_ prefix is added to class name

activeCfg.setInitializationCode(initCode);

app.generate();

app.build();

app.runApplication();

```

What about methods for saving a model after changes were made to it?

The following methods can be used to save an entire model. Their behavior is similar to that of the corresponding menu items in the GUI:

- IRPProject.save()
- IRPProject.saveAs(String filename)
The format of the string used for the file name should be similar to that used to open a project, for example, "I:\\temp_sample_diagrams\\Diagram_sample.rpy"
- IRPProject.saveAsPrevVersion(String filename, String prevVersion)
This method represents the option provided to save a model in the format of an older version of Rhapsody. The second String argument should consist of the version number that should be used, for example "7.4".

The following sample code opens a project, adds a class, saves the project, and then closes the project.

```

package documentation.rhapsodyapi;

import java.lang.*;

import com.telelogic.rhapsody.core.*;

public class HelloWorldModel {

    static IRPApplication app =
    RhapsodyAppServer.getActiveRhapsodyApplication();

    // create main function

    public static void main(String[] args) {

```

```

        if(app != null)
            openAndCloseProject();
    }

    // create second function that the main function calls
    private static void openAndCloseProject() {
        app.createNewProject("l:\\temp\\_sample_code", "Open_Close");
        IRPPProject prj =
app.openProject("l:\\temp\\_sample_code\\Open_Close.rpy");
        IRPPackage vehiclePackage = prj.addPackage("Vehicles");
        vehiclePackage.addClass("Car");
        prj.save();
        prj.close();
    }
}

```

How about an example of creating a statechart?

The following points will help you understand the sample code.

- All top-level states are added to the "root" state, which you get by calling `IRPStatechart.getRootState()`. Elements such as termination states and condition connectors are also added to the root state.
- All statecharts have a default transition. You create a default transition by calling `IRPState.CreateDefaultTransition` and providing the root state as the parameter.
- Timeout transitions are created using the `IRPTransition.setItsTrigger` method, as follows: `screamTimeoutTransition.setItsTrigger("tm(10000)")`. The number represents the number of milliseconds.
- When you create a diagram with the API, the graphical representation is not created by default. This means that the first time you open the diagram in Rhapsody, you will be asked if the graphics should be created. You can create the graphical representation directly by calling `IRPStatechart.createGraphics()`.

The statechart represents the behavior of a coach of an American football team during the course of a game.

```
package documentation.rhapsodyapi;

import javax.swing.JFrame;
import javax.swing.JOptionPane;
import com.telelogic.rhapsody.core.*;

public class StatechartDesigner {

    static IRPApplication app =
RhapsodyAppServer.getActiveRhapsodyApplication();

    public static void main(String[] args) {

        if(app != null){

            designStatechart();

        }

    }

    private static void designStatechart() {

        // create and open project
```

```

        app.createNewProject("1:\\temp\\_sample_code",
"Statechart_Design");

        IRPProject prj =
app.openProject("1:\\temp\\_sample_code\\Statechart_Design.rpy");

        // add package and class
        IRPPackage footballTeamPkg = prj.addPackage("FootballTeam");
        IRPClass coach = footballTeamPkg.addClass("Coach");

        // create statechart for class
        IRPStatechart coachChart = coach.addStatechart();

        // add three states
        IRPState rootState = coachChart.getRootState();
        IRPState thinkingState = rootState.addState("Thinking");
        IRPState screamingState = rootState.addState("Screaming");
        IRPState celebratingState =
rootState.addState("Celebrating");

        // add entry and exit actions
        celebratingState.setEntryAction("exchangeHighFives();");
        celebratingState.setExitAction("calmDown();");

        // add default transition
        thinkingState.createDefaultTransition(rootState);

        // add other transitions and set triggers, guards, actions
for transitions

        // also, add condition connector, transition to condition,
and transitions out of condition

        IRPTransition tdTransition =
thinkingState.addTransition(celebratingState);

        IRPTrigger tdTrigger =
tdTransition.setItsTrigger("touchdown");

        tdTransition.setItsAction("raiseHands();");

```

```

        IRPConnector tranqConditionConnector =
rootState.addConnector("Condition");

        IRPTransition badCall =
thinkingState.addTransition(tranqConditionConnector);

        badCall.setItsTrigger("bad_call_by_referee");

        IRPAttribute takingTranquilizers =
coach.addAttribute("isTakingTranquilizers");

        takingTranquilizers.setTypeDeclaration("RhpBoolean");

        takingTranquilizers.setDefaultValue("true");

        IRPTransition calmTransition =
tranqConditionConnector.addTransition(thinkingState);

        calmTransition.setItsGuard("takingTranquilizers == true");

        IRPTransition notCalmTransition =
tranqConditionConnector.addTransition(screamingState);

        notCalmTransition.setItsGuard("else");

        notCalmTransition.setItsAction("throwDownHeadset();");

        // add transitions with timeouts for returning to Thinking
state

        IRPTransition screamTimeoutTransition =
screamingState.addTransition(thinkingState);

        screamTimeoutTransition.setItsTrigger("tm(10000)");

        IRPTransition celebTimeoutTransition =
celebratingState.addTransition(thinkingState);

        celebTimeoutTransition.setItsTrigger("tm(20000)");

        // add termination state and transitions

        IRPState termState = rootState.addTerminationState();

        IRPTransition gameOver1Trans =
thinkingState.addTransition(termState);

```

```

        gameOver1Trans.setItsTrigger("game_over");
        gameOver1Trans.setItsGuard("noPenalty == true");

        // for next two transitions, use same trigger and guard
defined for first transition

        IRPTransition gameOver2Trans =
celebratingState.addTransition(termState);

gameOver2Trans.setItsTrigger(gameOver1Trans.getItsTrigger().getBody());

        gameOver2Trans.setItsGuard(gameOver1Trans.getItsGuard().getBody());

        IRPTransition gameOver3Trans =
screamingState.addTransition(termState);

gameOver3Trans.setItsTrigger(gameOver1Trans.getItsTrigger().getBody());

        gameOver3Trans.setItsGuard(gameOver1Trans.getItsGuard().getBody());
        // create graphics for statechart
        coachChart.createGraphics();

        // display dialog containing information about the statechart

        String intro = "This message box will explain a bit about the
statechart, but it's really just a way" +

            " to demonstrate how to use the various 'get' methods
provided in the API.\n\n";

        String itsClassName = "This statechart is for the class " +
coachChart.getItsClass().getDisplayName() + ".\n\n";

        int numOfTriggers = coachChart.getAllTriggers().getCount();

        String allTriggerText = "The statechart contains the
following triggers:\n\n";

        IRPEvent tempTrigger;

        for (int trigCount = 0; trigCount < numOfTriggers; trigCount+
+){

```



```

        tempTrigger =
(IRPEvent)coachChart.getAllTriggers().getItem(trigCount + 1);

        allTriggerText = allTriggerText + " * " +
tempTrigger.getDisplayName() + "\n\n";

    }

    String tdTriggerText = "When the " +
tdTransition.getItsTrigger().getBody() + " event occurs, the coach leaves
the "

        + tdTransition.getItsSource().getDisplayName() + "
state, " +

        "performs the " + tdTransition.getItsAction().getBody()
+ " action, and enters the "

        + tdTransition.getItsTarget().getDisplayName() + "
state.\n\n";

    String celebStateText = "When entering the " +
celebratingState.getDisplayName() +

        " state, the coach performs the action " +
celebratingState.getEntryAction() + ".\n\n" +

        "When exiting this state, the coach performs the action "
+ celebratingState.getExitAction() + ".\n\n";

    IRPGuard calmGuard = calmTransition.getItsGuard();

    String guardInfo = "Even if a bad call is made by the
referee, the coach will go back to the Thinking state if the following
condition is met:\n\n" + " * " + calmGuard.getBody() + "\n\n";

    JOptionPane.showMessageDialog(new JFrame(),intro +
itsClassName + allTriggerText + tdTriggerText + celebStateText +

        guardInfo);

    } // end of method designStatechart
}

```

What about extracting information from a model, for example, generating a report or exporting diagrams as graphic files?

The API contains two methods for generating a report from your model, one that uses the basic report generator, and one that uses ReporterPLUS:

- `IRPApplication.report(String format, String outputFileName)`
This method uses Rhapsody's basic report generator, allowing you to specify the output format ("ASCII" or "RTF"), and where the output file should be saved. There is no need to specify the file extension; Rhapsody will append ".txt" or ".rtf" to the string you provide. Note that the report will include information for the elements in the scope of the active component.
- `IRPProject.generateReport(String modelscope, String templatename, String docType, String filename, int showDocument, int silentMode)`
This method generates a report using the reporting capabilities of ReporterPLUS. The arguments to be provided represent: the name of the package to report on (or empty string for the whole model), the name of the template to use, the type of output to generate ("doc", "html", "ppt", or "txt"), the filename for the generated output, whether the output should be displayed (use 1 to display, 0 otherwise), and whether or not the ReporterPLUS wizard should be displayed if not all of the required information is provided as arguments (use 1 to skip display of the wizard in such cases).
There is no need to specify the file extension; Rhapsody will append the appropriate extension to the string you provide.

When this method is used to generate a report, the Rhapsody model is saved before the report is generated.

With regard to exporting diagrams as graphic-format files, the API provides methods for:

- Saving a diagram as an emf file
`IRPDiagram.getPicture(String filename)`
- Saving a diagram in a graphic format other than emf
`IRPDiagram.getPictureAs(String firstFileName, String imageFormat, int getImageMaps, IRPCollection diagrammap)`
The relevant strings for the imageFormat argument are "EMF", "BMP", "JPEG", "JPG", "TIFF".

For both of these export methods, the filename argument should include the appropriate extension. Rhapsody does not add the extension automatically as it does for the report generation methods.

In addition to allowing you to specify a graphic-file format, the `getPictureAs` method includes arguments that allow you to

- a) export the diagram as a number of files rather than shrinking the diagram*
- b) retrieve diagram element information that can be used to create an HTML image map.*

For a detailed explanation of these additional arguments, see the Javadoc output for the API ([installation directory]\Doc\java_api).

The following sample code illustrates the use of the report generation methods and the methods for exporting diagrams.

In this sample, the line that calls the `generateReport` method must be modified before you run it because it has to point to the location of the ReporterPLUS templates in your installation of Rhapsody.

```
package documentation.rhapsodyapi;

import com.telelogic.rhapsody.core.*;

public class ExtractInfo {

    static IRPApplication app =
RhapsodyAppServer.getActiveRhapsodyApplication();

    // create main function

    public static void main(String[] args) {

        if(app != null)

            extractInformation();

    }

    // create second function that the main function calls

    private static void extractInformation() {

        app.createNewProject("1:\\temp\\_sample_code",
"Extract_Info");

        IRPProject prj =
app.openProject("1:\\temp\\_sample_code\\Extract_Info.rpy");

        IRPPackage vehiclePackage = prj.addPackage("Vehicles");

        IRPClass car = vehiclePackage.addClass("Car");

        IRPClass jeep = vehiclePackage.addClass("Jeep");

        IRPClass convertible =vehiclePackage.addClass("Convertible");

        IRPClass coolJeep = vehiclePackage.addClass("CoolJeep");

        jeep.addGeneralization(car);

        convertible.addGeneralization(car);

        coolJeep.addGeneralization(jeep);

    }

}
```

```

        IRPDiagram vehicleOmd =
vehiclePackage.addObjectModelDiagram("vehicle_omd");

        IRPCollection vehicleClasses = vehiclePackage.getClasses();
        IRPCollection relationTypes = app.createNewCollection();
        relationTypes.setSize(1);
        relationTypes.setString(1, "Inheritance");
        vehicleOmd.populateDiagram(vehicleClasses, relationTypes,
"fromto");

        prj.save();

        app.report("RTF",
"1:\\temp\\_sample_code\\reports_and_pictures\\project_report");

        // update the following line to point to the directory that
contains the Reporter Plus templates

        prj.generateReport("",
"1:\\programs\\rhapsody_752_build_1471822\\reporterplus\\Templates\\Rhaps
ody HTML Exporter.tpl", "html",
"1:\\temp\\_sample_code\\reports_and_pictures\\project_report_rep_plus",
0, 1);

vehicleOmd.getPicture("1:\\temp\\_sample_code\\reports_and_pictures\\omd.
emf");

vehicleOmd.getPictureAs("1:\\temp\\_sample_code\\reports_and_pictures\\om
d.jpg", "JPG", 0, null);

    }
}

```

Can I see some sample code that shows some of the things you can do with classes and their attributes and operations?

```
package documentation.rhapsodyapi;

import com.telelogic.rhapsody.core.*;

public class ClassDemo {

    static IRPApplication app =
RhapsodyAppServer.getActiveRhapsodyApplication();

    // main method

    public static void main(String[] args) {

        if(app != null)

            manipulateClassData();

    }

    // this method demonstrates some of the things you can do with
classes, and their attributes and operations

    private static void manipulateClassData() {

        // for this example, keep in mind that IRPClass inherits from
IRPClassifier

        // create and open project

        app.createNewProject("1:\\temp\\_sample_code",
"Class_Tricks");

        IRPProject prj =
app.openProject("1:\\temp\\_sample_code\\Class_Tricks.rpy");

        // create package and classes

        IRPPackage cameraPackage = prj.addPackage("Cameras");

        IRPClass cameraClass = cameraPackage.addClass("Camera");

        IRPClass stillsCameraClass =
cameraPackage.addClass("StillsCamera");

        IRPClass videoCameraClass =
cameraPackage.addClass("VideoCamera");

        IRPClass comboCameraClass =
cameraPackage.addClass("ComboCamera");
```

```

        // can use either of these two methods for establishing
inheritance relationship

        stillsCameraClass.addGeneralization(cameraClass);

        videoCameraClass.addGeneralization(cameraClass);

        comboCameraClass.addSuperclass(cameraClass);


        // print name of base class, delete generalization, and then
print again to verify deletion

        IRPCollection classesStillsInheritsFrom =
stillsCameraClass.getGeneralizations();

        int numberOfBaseClasses =
classesStillsInheritsFrom.getCount();

        IRPGeneralization tempGeneralization;

        System.out.println("The class " + stillsCameraClass.getName()
+ " inherits from the following base classes:");

        // note that when using getItem to get an item from an
IRPCollection object, the index starts at 1, not 0

        for(int i = 1; i < numberOfBaseClasses+1 ; i++) {

            tempGeneralization = (IRPGeneralization)
(classesStillsInheritsFrom.getItem(i));

            System.out.println("    " +
tempGeneralization.getDisplayName());

        }

        stillsCameraClass.deleteGeneralization(cameraClass);

        classesStillsInheritsFrom =
stillsCameraClass.getGeneralizations();

        numberOfBaseClasses = classesStillsInheritsFrom.getCount();

        System.out.println("\nNow, after deletion, " +
stillsCameraClass.getName() + " inherits from the following base
classes:");

        if (numberOfBaseClasses==0) {

            System.out.println("    No base classes found.\n");

        } else {

            for(int i = 1; i < numberOfBaseClasses+1 ; i++) {

                tempGeneralization = (IRPGeneralization)
(classesStillsInheritsFrom.getItem(i));

                System.out.println("    " +

```

```

tempGeneralization.getDisplayName());

        }

    }

    // restore inheritance
    stillsCameraClass.addGeneralization(cameraClass);

    // getBaseClassifiers is similar to getGeneralizations but
will return classes, not generalizations

    IRPCollection allBaseClasses =
stillsCameraClass.getBaseClassifiers();

    System.out.println("After restoring inheritance, the class "
+ stillsCameraClass.getName() + " inherits from these classes:");

    numberOfBaseClasses = allBaseClasses.getCount();

    IRPClass tempClass;

    for(int i = 1; i < numberOfBaseClasses+1 ; i++) {
        tempClass = (IRPClass) (allBaseClasses.getItem(i));

        System.out.println("    " + tempClass.getDisplayName());
    }

System.out.println("=====
=====");

    // like adding a base class, there are also two methods for
deleting a base class - deleteGeneralization (used above) and
deleteSuperclass (below)

    IRPCollection classesVideoInheritsFrom =
videoCameraClass.getGeneralizations();

    numberOfBaseClasses = classesVideoInheritsFrom.getCount();

    System.out.println("The class " + videoCameraClass.getName() +
" inherits from the following base classes:");

    for(int i = 1; i < numberOfBaseClasses+1 ; i++) {

        tempGeneralization = (IRPGeneralization)
(classesVideoInheritsFrom.getItem(i));

        System.out.println("    " +
tempGeneralization.getBaseClass().getDisplayName());
    }

```

```

        videoCameraClass.deleteSuperclass(cameraClass);

        classesVideoInheritsFrom =
videoCameraClass.getGeneralizations();

        numberOfBaseClasses = classesVideoInheritsFrom.getCount();

        System.out.println("\nNow, after calling
deleteSuperclass, " +videoCameraClass.getName() + " inherits from the
following base classes:");

        if (numberOfBaseClasses==0) {

            System.out.println("    No base classes found.");

        } else {

            for(int i = 1; i < numberOfBaseClasses+1 ; i++) {

                tempGeneralization = (IRPGeneralization)
(classesVideoInheritsFrom.getItem(i));

                System.out.println("    " +
tempGeneralization.getBaseClass().getDisplayName());

            }

        }

System.out.println("=====
=====");

        // restore video camera inheritance
        videoCameraClass.addGeneralization(cameraClass);

        // get and print all classes derived from cameraClass
        IRPCollection allDerivedClasses =
cameraClass.getDerivedClassifiers();

        System.out.println("These classes are derived from " +
cameraClass.getName() + ":");

        int numberOfDerivedClasses = allDerivedClasses.getCount();

        for(int i = 1; i < numberOfDerivedClasses+1 ; i++) {

            tempClass = (IRPClass)(allDerivedClasses.getItem(i));

            System.out.println("    " + tempClass.getDisplayName());

        }

        // get a specific class derived from cameraClass

```



```

        tempClass =
(IRPClass)cameraClass.findDerivedClassifier("StillsCamera");

        System.out.println("\nThis is the derived class you were
looking for:\n" + "    " + tempClass.getDisplayName());

        // now let's search in the other direction - find specific
base class

        // first method, findBaseClassifier, returns
IRPClassifier,

        // second method, findGeneralization, returns
IRPGeneralization, which represents the inheritance relationship

        tempClass =
(IRPClass)stillsCameraClass.findBaseClassifier("Camera");

        System.out.println("\nThis is the base class you were looking
for:\n" + "    " + tempClass.getDisplayName());

        tempGeneralization =
(IRPGeneralization)stillsCameraClass.findGeneralization("Camera");

        System.out.println("\nThis is the base class of the
generalization you were looking for:\n" + "    " +
tempGeneralization.getBaseClass().getDisplayName());

System.out.println("=====
=====");

        // now let's add some attributes

        // for the first one, we'll use a variable for the attribute
created so we can use it later

        IRPAttribute resolution =
cameraClass.addAttribute("maxResolution");

        cameraClass.addAttribute("weight");

        cameraClass.addAttribute("internalMemory");

        cameraClass.addAttribute("color");

        // print list of attributes

        IRPCollection cameraAttributes = cameraClass.getAttributes();

        System.out.println("The camera class contains the following
attributes:");

        IRPAttribute tempAtt;

```

```

        for(int i = 1; i < cameraAttributes.getCount()+1 ; i++) {
            tempAtt = (IRPAttribute)cameraAttributes.getItem(i);
            System.out.println("    " + tempAtt.getDisplayName());
        }

        // delete one of the attributes
        IRPAttribute attToDelete =
cameraClass.findAttribute("weight");

        cameraClass.deleteAttribute(attToDelete);

        // print list again
        cameraAttributes = cameraClass.getAttributes();

        System.out.println("\nNow, after deleting the weight
attribute, the camera class contains the following attributes:");

        for(int i = 1; i < cameraAttributes.getCount()+1 ; i++) {
            tempAtt = (IRPAttribute)cameraAttributes.getItem(i);
            System.out.println("    " + tempAtt.getDisplayName());
        }

System.out.println("=====
=====");

        // check and change visibility of an attribute
        System.out.println("The visibility setting for the attribute
" + resolution.getDisplayName() +
            " is\n    " + resolution.getVisibility() +
"\nwhich is the default setting\n");

        // possible parameter values for setVisibility are public,
private, protected,

        // project (for internal in c#) and projectOrProtected
(for protected internal in c#)
        resolution.setVisibility("private");

        System.out.println("Now, the visibility setting for the
attribute " + resolution.getDisplayName() +
            " is\n    " + resolution.getVisibility());

System.out.println("=====
=====");

        // set constant, static attributes

```

```

        cameraClass.findAttribute("internalMemory").setIsConstant(1);
        cameraClass.findAttribute("internalMemory").setIsStatic(1);
        cameraAttributes = cameraClass.getAttributes();
        for(int i = 1; i < cameraAttributes.getCount()+1 ; i++) {
            tempAtt = (IRPAttribute)cameraAttributes.getItem(i);
            if (tempAtt.getIsConstant() == 1) {
                System.out.println("The attribute " +
tempAtt.getDisplayName() + " is defined as constant");
            } else {
                System.out.println("The attribute " +
tempAtt.getDisplayName() + " is *not* defined as constant");
            }
            if (tempAtt.getIsStatic() == 1) {
                System.out.println("The attribute " +
tempAtt.getDisplayName() + " is defined as static\n");
            } else {
                System.out.println("The attribute " +
tempAtt.getDisplayName() + " is *not* defined as static\n");
            }
        }

System.out.println("=====
=====");

        //set attribute to "reference" - refers to "by reference" -
so not applicable for Java or c# models
        cameraClass.findAttribute("color").setIsReference(1);
        cameraAttributes = cameraClass.getAttributes();
        for(int i = 1; i < cameraAttributes.getCount()+1 ; i++) {
            tempAtt = (IRPAttribute)cameraAttributes.getItem(i);
            if (tempAtt.getIsReference() == 1) {
                System.out.println("The attribute " +
tempAtt.getDisplayName() + " is defined as by reference");
            } else {
                System.out.println("The attribute " +
tempAtt.getDisplayName() + " is *not* defined as by reference");
            }
        }

```

```

    }

}

System.out.println("=====
=====");

    // use of getAttributes vs getAttributesIncludingBases
    stillsCameraClass.addAttribute("hasTimer");

    IRPCollection stillsAttributes =
stillsCameraClass.getAttributes();

    System.out.println("The " +
stillsCameraClass.getDisplayName() + " contains the following attributes
of its own:");

    for(int i = 1; i < stillsAttributes.getCount()+1 ; i++) {
        tempAtt = (IRPAttribute)stillsAttributes.getItem(i);
        System.out.println("    " + tempAtt.getDisplayName());
    }

    IRPCollection stillsAttributesWithBases =
stillsCameraClass.getAttributesIncludingBases();

    System.out.println("\nThe " +
stillsCameraClass.getDisplayName() + " contains the following attributes
(includes those of its base class):");

    for(int i = 1; i < stillsAttributesWithBases.getCount()+1 ;
i++) {
        tempAtt =
(IRPAttribute)stillsAttributesWithBases.getItem(i);
        System.out.println("    " + tempAtt.getDisplayName());
    }

System.out.println("=====
=====");

    // add operations and then print list that displays certain
characteristics of the operations

    // for the first few, we'll use variables for the
operations created so we can use them later

    IRPOperation takePicture =
cameraClass.addOperation("takePicture");

```

```

        IRPOperation setResolution =
cameraClass.addOperation("setResolution");

        IRPOperation windFilm = cameraClass.addOperation("windFilm");
        cameraClass.addTriggeredOperation("checkBatteryStrength");
        cameraClass.addDestructor();

        //for addConstructor, the argument should be a string that
uses the format "name1,type1,name2,type2",

        // for example "a,int,b,int". For a constructor that does not
take arguments, use an empty string ("")

        cameraClass.addConstructor("a, int, b, int");

        IRPOperation tempOperation;

        IRPCollection cameraOperations = cameraClass.getOperations();

        System.out.println("The " + cameraClass.getDisplayName() + "
class contains the following operations:");

        for(int i = 1; i < cameraOperations.getCount()+1 ; i++) {

            tempOperation =
(IRPOperation) cameraOperations.getItem(i);

            if (tempOperation.getIsTrigger() == 1) {

                System.out.println("    " +
tempOperation.getDisplayName() + "(triggered operation)");

            } else {

                if (tempOperation.getIsDtor() == 1) {

                    System.out.println("    " +
tempOperation.getDisplayName() + "(destructor)");

                } else if (tempOperation.getIsCtor() == 1) {

                    System.out.println("    " +
tempOperation.getDisplayName() + "(constructor)");

                }

                else {

                    System.out.println("    " +
tempOperation.getDisplayName());

                }

            }

        }

    }

```

```

        // use the final keyword
        setResolution.setIsFinal(1);

        System.out.println("\nThe following methods were defined as
final:");

        cameraOperations = cameraClass.getOperations();
        for(int i = 1; i < cameraOperations.getCount()+1 ; i++) {
            tempOperation =
(IRPOperation) cameraOperations.getItem(i);
            if (tempOperation.getIsFinal() == 1) {
                System.out.println("    " +
tempOperation.getDisplayName() + " (final)");
            }
        }

System.out.println("=====
=====");

        // delete some operations and print out to verify
        cameraClass.deleteDestructor();
        // windFilm ? what is film ?
        cameraClass.deleteOperation(windFilm);
        cameraOperations = cameraClass.getOperations();

        System.out.println("Now, after deleting some operations, the
" + cameraClass.getDisplayName() + " class contains the following
operations:");

        for(int i = 1; i < cameraOperations.getCount()+1 ; i++) {
            tempOperation =
(IRPOperation) cameraOperations.getItem(i);
            System.out.println("    " +
tempOperation.getDisplayName());
        }

System.out.println("=====
=====");

        // set and then get the operation body
        takePicture.setBody("openShutter();\n closeShutter();");

        System.out.println("The body of the " +
takePicture.getDisplayName() + " method is as follows:\n"+
takePicture.getBody());

```

```

        // print out the return type of the method

        System.out.println("\nThe return type of the " +
takePicture.getDisplayName() + " method is " +
takePicture.getReturns().getDisplayName());

System.out.println("=====
=====");

        // modify visibility of an operation

        System.out.println("The visibility of the operation " +
setResolution.getDisplayName() + " is " + setResolution.getVisibility() +
", which is the default");

        setResolution.setVisibility("Private");

        System.out.println("\nNow, the visibility of the operation "
+ setResolution.getDisplayName() + " is " +
setResolution.getVisibility());

System.out.println("=====
=====");

        // create class diagram to illustrate relationship between
classes in model

        // note that this is just a simple class diagram to allow
you to view a diagram of your model - diagrams are discussed in detail in
another section

        IRPDiagram classDiagramToCreate =
cameraPackage.addObjectModelDiagram("Classes in Cameras package");

        IRPCollection classesToAddToDiagram =
cameraPackage.getClasses();

        IRPCollection typesOfRelationsToShow =
app.createNewCollection();

        typesOfRelationsToShow.setSize(1);

        typesOfRelationsToShow.setString(1, "Inheritance");

        classDiagramToCreate.populateDiagram(classesToAddToDiagram,
typesOfRelationsToShow, "fromto");

        // this code sample used println statements to reflect the
changes made to the model

        // you can also use a debugger to step through the code and
view the changes made to the model in Rhapsody's model browser

    }
}

```