Chapter 4
Language Features for Object-Oriented Implementation

Many modern programming language features can be divided into two parts: basic features that are essential to use the programming paradigm and supporting concepts that are needed to facilitate the construction of more complex systems. So far, we have covered core language issues for the object-oriented paradigm, such as classes, inheritance, interfaces, and so on.

In this chapter we will study several concepts that fall in the supporting category. We begin in Sect. 4.1 with a study of how to organise source files (and class files) in a Java application. Following this, in Sect. 4.2, we look at an important type of class called collection class.

In Sect. 4.3 we study exceptions, which are situations in which the system reports an error and abandons the current operation. Dynamic binding in object-oriented languages leads to situations where a type of an object has to be determined explicitly by the program at runtime; this necessitates the need for run time type identification (RTTI), which is introduced in Sect. 4.4. In Sect. 4.5 we study how to build graphical user interface (GUI) programs. The problem of providing long-term storage of objects is discussed in Sect. 4.6.

While these concepts are not directly related to each other, they are all widely regarded as being essential for software system design today, and the reader must gain a reasonable grasp of these topics before undertaking the analysis and design of object-oriented systems (which we start in Chap. 6).

4.1 Organising the Classes

In any complex system, it is essential that the components be located in a manner that facilitates easy access. Classes and interfaces are modules that make up our software system and our first order of business is to have a system for organising these.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_4

4.1.1 Creating the Files

There are some general rules and conventions related to file organisation. Typical practice is to put at most one class or interface in a single file. The file must be named `<class/interface name>.java`. Java requires that with more than one class or interface in a file, only one of the outer classes/interfaces can be public; if there is a public class/interface in a file, the name of that class/interface must be used for naming the file.

4.1.2 Packages

One major theme in object-oriented paradigm is reuse. This decreases development time, reduces code size, and increases reliability. The Java language comes with a large number of classes (numbering in the thousands) that can be used for a variety of uses: networking, GUI, database management, and so on.

We will use some classes from Java quite extensively so that we can focus more on the design issues. This is also consistent with the theme of reuse.

The Java classes are spread over what are called **packages**, which we briefly discuss here.

A package is a collection of classes. It is usually named as a sequence of lowercase letters and periods. Some of the major packages are `java.lang`, `java.util`, `java.awt`, `javax.swing`, `java.io`, and `java.lang.reflect`.

The package `java.lang` contains classes and interfaces that are fundamental to the language. These include `String`, `Thread`, `Runnable`, `Integer`, `Double`, etc. The package `java.util` contains interfaces and classes for storing lists and sets, among others. Graphical programs can make use of members in `java.awt` and/or `javax.swing`. To perform input and output, one may use the package `java.io`. Classes and interfaces can be interrogated using `java.lang.reflect`, which is said to be a sub-package of `java.lang`.

Java automatically makes the classes and interfaces in the `java.lang` package available. Programs that use classes from other packages must, however, **import** them from the appropriate package. For instance, to use the class `Vector` which resides in `java.util`, the code must resort to one of the several approaches.

One way is to prefix the class name with the name of the package.

```
java.util.Vector myVector = new java.util.Vector();
```

The above can be cumbersome and few programmers resort to it.

A second approach is to import that class. Write

```
import java.util.Vector;
```

This is fine if the code is using only a few classes from a package. To import all of the members of a package, code as below.

```
import java.util.*;
```

There is no serious drawback to doing the above. In some cases, class/interface names from two packages may conflict, which then has to be resolved by prefixing the class name with the package name in the code itself.

Also, note that importing all members of a package does not import sub-packages. For example, although there are packages `java.awt` and `java.awt.image`, the statement

```
import java.awt.*;
```

does not import the class `java.awt.image.ColorModel`. We need to write

```
import java.awt.image.*;
```

as a separate statement.

Users can put classes they create in their own package by writing

```
package <package-name>;
```

This must appear as the first statement in the file.

After compilation, the class file must be copied into a sub-directory with the same name as the package name. This sub-directory must appear within a directory that is listed in the environment variable `CLASSPATH`, the setting of which is dependent on the operating system.

4.1.3 Protected Access and Package Access

We have seen the use of `protected` access specifier in Chap. 3. Suppose we have a field `x` defined as `protected` in a class `C`. Then, the field can also be accessed in classes that reside in the same package as `C`. For example, the following code is legal.

```
package mypackage;
public class C {
  protected int x;
}
package mypackage;
import mypackage.C;
```

```
public class D {
  public void f(C c) {
    c.x = 1;
  }
}
```

If we omit any explicit access specifier in the definition of a method or field, the access is said to be a package access, which means that only the code residing in a class within the same package can access the method or field.

4.2 Collection Classes

The `java.util` package contains a number of useful interfaces and classes that we will use in our examples. The interface `java.util.Collection`, for instance, contains methods for manipulating a collection. Some of the methods in this interface are:

1. `boolean add(Object object)`: adds the supplied object to the collection.
2. `boolean addAll(Collection collection)`: adds all objects in the supplied collection to this collection.
3. `void clear()`:removes all of the elements from this collection.
4. `boolean contains(Object object)`: returns true if and only if this collection contains the supplied object.
5. `int size()`: returns the number of elements in this collection.
6. Methods for removing objects, checking if the collection is empty, etc.

The `List` interface extends `Collection`. A list is a collection of objects where the objects are put in a sequence. Thus, it has all the methods that pertain to a collection and the ones that are specific to lists such as `void add(int index, Object object)` which inserts the given object at the position specified by the index in this list.

There are two major implementations of `List`: `LinkedList` and `ArrayList`. The names of the classes indicate how they are implemented.

Using the above classes, it is easy to create and use lists. The following simple class creates a sequence of `String` objects, stores them in a list, and prints the list.

```
import java.util.*;
public class ListUseExample {
  public static void main(String[] s) {
    List list = new ArrayList();
    for (int count = 1; count <= 10; count++) {
      list.add(new String("String " + count));
    }
```

```
      for (int count = 0; count <= 9; count++) {
        System.out.println(list.get(count));
      }
    }
  }
```

Since `ArrayList` implements the `List` interface, the following code is legal:

```
List list = new ArrayList();
```

Into this list we are adding 10 Strings, ‘`"String1"` through `"String10"`. The add method adds at the end of the list. Lists are indexed from 0, so `"String1"` is at index 0 and `"String10"` is at index 9. The `get` method returns the element at the specified index. The second `for` loop prints the `String` objects at positions 0 through 9.

4.3 Exceptions

We saw in Chap. 3 that casting an objectto a type to which it does not conform causes an error. More specifically, the system throws an **exception**, which results in a crash. This is a rather loose description of what happens, and the following discussion is more accurate and complete.

Recall the Chap. 3 example of the three classes, `Student`,`Undergraduate Student`, and `GraduateStudent`, where the last two classes inherit from the first. The following code has a problem because we are casting an `Undergraduate Student` object as a `GraduateStudent` object. We are asking the system to do something that it cannot.

```
Student student = new UndergraduateStudent();
GraduateStudent graduateStudent = (GraduateStudent) student;
```

To be more precise, when the code reaches the second line and the cast is attempted, the system abandons the operation, generates an object that represents this abnormal operation, and **throws** the object. This and similar problematic situations always cause a `Throwable` object to be generated and thrown and the offending operation to be abandoned. The specific type of the object depends on the type of operation. Here are some examples.

1. An attempt is made to access an array with an invalid index. The object generated is of type `ArrayIndexOutOfBoundsException`.
2. A null reference is used to access a field or method of an object. In this case, the object generated is of type `NullPointerException`.

3. An error occurs while an input or output operation occurs. The object in this case is of type `IOException`.
4. An attempt to cast an object fails as in the student example. The exception type is called `ClassCastException`.

If we want to avoid a crash because of a bad cast or any other erroneous piece of code, we have to put the offending code within a try block and catch the exception object.

```
try {
  Student student = new UndergraduateStudent();
  GraduateStudent graduateStudent = (GraduateStudent) student;
  // process the object
} catch (ClassCastException cce) {
  // Object is not of type graduate student.
  // do some operation to recover from the error
}
```

An application may choose to catch exceptions that its code may throw; for this, these statements have to be enclosed in a `try` block. The block begins with the keyword `try` followed by the left-curly bracket `{`, a sequence of statements (that may have any statements including more try blocks) ending with a `}`. This should be followed by at least one `catch` block.

A catch block begins with the keyword `catch` followed by a pair of parentheses with an exception name (which is a class name) and a reference to refer to the exception object. The catch block typically contains code to rectify the problem.

When a statement in a `try` block throws an exception, the system throws an object of a certain exception type and the try block is abandoned. The system then checks to see if there is a `catch` block for that exception type associated with this `try` block. If so, that `catch` block is entered and the code in it is executed. Once the `catch` block is entered, the exception is caught and this instance of the exception cannot crash the program.

Let us trace the above code for the case. When the class cast is performed, Java throws an object of type `ClassCastException`. The rest of the code, including the assignment in the `try` block is abandoned. Java searches to see if there is a `catch` block for the type of the exception raised, which is `ClassCastException`. Since there is one, the corresponding `catch` block is entered and the code in it is executed. The parameter `cce` refers to the object thrown.

We can put multiple `catch` blocks for a single `try` block. Here is a piece of code that handles three different types of exceptions:

```
try {
  if (myObject.getField1().equals(someObject)) {
    int index = myObject.getIndex();
    int value = Integer.parseInt(JOptionPane.showInputDialog(null,
                                                  "Enter a number"));
```

```
      myArray[index] = value;
    } catch (NullPointerException npe) {
      System.out.println("Null pointer " + npe);
      System.exit(0);
    } catch (ArrayIndexOutOfBoundsException aiofbe) {
      System.out.println("Array index out of range " + aiofbe);
      return;
    } catch (NumberFormatException nfe) {
      System.out.println("Invalid entry; exception " + nfe);
      return;
    }
  }
```

`NumberFormatException` occurs when we try to convert a string that does not have a numeric value in it to a number.

Although the above pieces of code are technically correct, we should not, in general, use `try` and `catch` blocks to handle exceptions such as `ArrayIndexOutOfBoundsException` and `NullPointerException` because they can be avoided by properly debugging the program. On the other hand, there is a class of exceptions called **checked exceptions** that can occur even in correct programs. The `try` and `catch` blocks are appropriate for processing such checked exceptions. *One of the characteristics of a well-designed software system is that it appropriately uses exceptions to handle unexpected situations.*

4.4 Run-Time Type Identification

Although polymorphism and dynamic binding are powerful tools, they are not sufficient to take care of all the issues that arise when dealing with an inheritance hierarchy. Consider, for example, a `Shape` class with two subclasses, `Square` and `Circle`. Let `ShapeList` be a collection of `Shape`. If we access an item from this collection, we know that it will be of type `Shape`, but we do not know whether it will be a `Square` or a `Circle`.

Say, we have an application that needs to know the number of `Circle` objects in a `ShapeList` collection. This could be implemented as a public method in `ShapeList`.

```
public int circleCount()
```

or as a client method that takes a reference to a `ShapeList`.

```
int circleCount(Shapelist shapeList)
```

In either case, the method will iterate through all the items in the collection, check which ones are of type `Circle`, etc. We therefore need some mechanism to detect whether a given `Shape` object is a `Circle`. Applying polymorphism and dynamic binding would suggest that we have a method in the `Shape` class (named `isCircle`, say,) that returns `true` when a `Shape` object is a `Circle`, but having

such a method defeats the purpose of having dynamic binding in the first place! Also, such a solution would be inelegant if we had a large hierarchy.

A more subtle problem arises with client methods. Consider a class `Investment` with two subclasses `Deposit` and `Stock`. A deposit would accrue interest, whereas stocks pay dividend. A client method that computes taxes would look something like this:

```
double computeTax(Investment investment) {
  // find the total amount of income from the investment
  // and take appropriate action
}
```

In case of `Stock` objects, `computeTax` would invoke a method `getDividend` whereas for `Deposit` objects, the method `getInterest` would be invoked. In this case, we have a situation where methods needed for one subclass do not make sense for sibling classes.

Although such scenarios are not very common, we need a mechanism that can handle these. All object-oriented languages provide some form of **run-time type identification (RTTI)** that can take care of these situations cleanly. In the first example, we need a mechanism to test whether a given `Shape` object is a `Circle`, whereas in the second, we want to be sure that we downcast the `Investment` object correctly and apply the right method.

RTTI in Java can be done in one of three ways. In the rest of this section, we elaborate the approaches.

4.4.1 Reflection: Using the `Class` Object

Java supports the notion of **reflection** which is based on the notion of a special class known as `Class`. Associated with each class is a `Class` object, a reference to which can be obtained using the `getClass` method. The `Class` object, which is automatically created at run time, belongs to the class `Class`. This class has several methods that can be invoked to find out various properties of the class, such as the name, the list of fields and methods, etc. In particular, the method `getName` returns a `String` object holding the name of the class. To check if a given `Shape` object is a `Circle` using these methods, we do the following:

```
Shape shape;
// code to create a Shape object
// and store its reference in shape
if (shape.getClass().getName().equals("Circle")) {
  // take appropriate action
}
```

The method `getClass()` is defined by Java for the `Object` class, and is therefore automatically available for any user-defined class. In our example, `getClass` returns an object that stores information about the `Circle` class, and the method `getName` on that object returns the string `"Circle"`. While this serves our purpose, it suffers from one drawback: *the compiler cannot check for typographical errors in the string against which we are checking the name.* The following code, for instance, would compile correctly.

```
Shape shape;
// code to create s Shape object
// and store its reference in shape
if (shape.getClass().getName().equals("circle")) {
  // take appropriate action
}
```

Typing "circle" instead of "Circle" gives us an incorrect answer because the error in code cannot be caught by the compiler.

4.4.2 Using the `instanceof` Operator

This problem that we talked about above can be resolved if we use the `instanceof` operator to query the type of an object. Our code would the look like this:

```
Shape shape;
// code to create s Shape object
// and store its reference in shape
if (shape instanceof Circle) {
  // take appropriate action
}
```

The operator returns `true` if the object `shape` is an instance of the class `Circle`. In this case, the compiler ensures that `Circle` is a known class and flags an error otherwise. In case of the `computeTax` method, we create a similar solution.

```
double computeTax(Investment investment) {
  double amount;
  if (investment instanceof Deposit) {
    amount = (Deposit) investment.getInterest();
    // code for computing tax on amount
  } else if (investment instanceof Stock) {
    amount = (Stock) investment.getDividend();
    // code for computing tax amount
  }
  // return tax
}
```

The example above seems to suggest that using the `instanceof` operator is always a better alternative to using `getClass().getName()`, but that is not the case. In some situations `instanceof` does not give us sufficient information since it would return `true` for all ancestors. An example of a situation where `instanceof` cannot be used is given in Chap. 5.

4.4.3 Downcasting

As we know from Chap. 3, we can cast a superclass reference to a subclass. For example, we could code

```
double computeTax(Investment investment) {
  double amount;
  Deposit deposit = (Deposit) investment;
  amount = deposit.getInterest();
  // code for computing tax on amount
  // rest of the method not shown
}
```

The downcast could, of course, fail, in which case the system throws an instance of `ClassCastException`. Although `ClassCastException` is a `RuntimeException` and should not normally be caught, this could be considered an appropriate situation where it should be handled. We can rewrite the method as below.

```
double computeTax(Investment investment) {
  double amount;
  try {
    Deposit deposit = (Deposit) investment;
    amount = deposit.getInterest();
    // code for computing tax on amount
  } catch(ClassCastException cce) {
    try {
      Stock stock = (Stock) investment;
      amount = stock.getInterest();
      // code for computing tax on amount
    } catch(ClassCastException cce) {
      cce.printStackTrace();
    }
  }
  // return tax
}
```

The example above seems to suggest that downcasting and the `instanceof` operator can be used interchangeably. Although they are functionally equivalent, there is a stylistic difference in that exceptions, ideally, should not be thrown unless an exceptional situation occurs. In Chap. 10 we find a situation where downcasting is a natural solution to the problem at hand, and in Chap. 11 we have an example of a situation where the `instanceof` operator provides an elegant solution.

4.5 Graphical User Interfaces: Programming Support

In this section we discuss the basics of creating graphical user interfaces (GUI) in Java. We would like to emphasise the word 'basics'. The goal is to help the reader create simple GUIs and provide him/her with enough knowledge to explore and understand the extensive functionality provided by Java in this area.

Java GUI programs can take two forms: **applets** and **applications**. Applets are programs that need a web browser to live in; in other words, the applet occupies part of a web page. When a page containing an applet is downloaded, the applet comes along with the web page and gets executed by the browser. This helps provide more functionality than is otherwise possible using just text and graphics. We do not cover applets in this book.

GUI applications are standalone programs that can be executed like any other program but providing a graphical interface. They are only slightly more complicated to program than applets. With a knowledge of GUI applications, the reader should have little difficulty in learning to create applets.

4.5.1 The Basics

As a first step in grasping the fundamentals of GUI creation, let us take a simple GUI application and understand it. For this, consider the user interface given in Fig. 4.1.

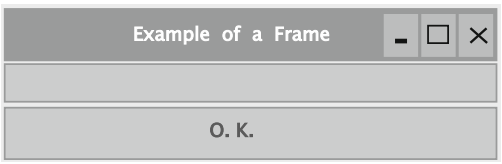

Fig. 4.1 A sample GUI screen

Let us break the shown interface into several parts.

1. An outer window with the title 'Example of a Frame', the minimise, maximise, and close buttons.
2. A white box, in which, although not obvious from the picture, the user can enter some text.
3. A button labelled 'O.K'.

Next, we will see the major steps in creating the interface from a programmer's perspective.

1. Create the window: The system will do most of the hard work. The programmer essentially says that a window is needed; the title for the window also can be supplied. The system draws the outline, the title bar, and supplies the three buttons: close, minimise, and maximise.
   A common class used for creating the window is `JFrame`. A possible code for creating the window is

```
new JFrame("Example of a Frame");
```

2. Create the two widgets, the text box and the button. The text box is created using the Java class `JTextField`, and the button is created using the class `JButton`. The system, once again, will perform the operations necessary to draw the two widgets.
   The first line in the following code fragment creates the button. Notice that we pass the label for the button while constructing it. The second line obviously constructs the text field. The parameter for the text field contains the length in characters for this widget. (We would like to note that the resulting text field's size may not precisely fit the number of characters specified as parameter.)

```
JButton button1 = new JButton("O.K.");
JTextField textField1 = new JTextField(20);
```

3. Next, we put the widgets in the frame. The frame has several **panes**. The widgets are stored in what is termed the **content pane.** While adding, we need to specify where the widgets should be added. By default, the content pane of a frame is divided into five parts as shown in Fig. 4.2.
   The five areas of the pane are referred to by the constants `BorderLayout.SOUTH`, `BorderLayout.NORTH`, `BorderLayout.WEST`, `BorderLayout.EAST`, and `BorderLayout.CENTER`. A widget is added by issuing the method `add` on the content pane object, which is obtained by issuing the method `getContentPane` on the frame. The `add` method requires the widget reference and the area of the pane; If no area is specified, the widget is stored in the centre area. Thus, in the code below, the text field is stored in the centre, whereas the button is stored in the south.

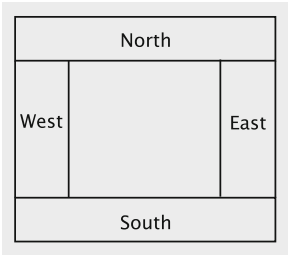

Fig. 4.2 Border layout

```
frame.getContentPane().add(textField1);
frame.getContentPane().add(button1, BorderLayout.SOUTH);
```

4. Until this time, the frame is not visible. The last step in this example is to display it. This is done by first issuing the `pack` method on the frame so that it is sized to fit the preferred size of the widgets based on the current layout.

```
frame.pack();
frame.setVisible(true);
```

The complete code for the example is given below.

```
import javax.swing.*;
import java.awt.*;
public class FrameDemo {
  public static void main(String[] s) {
    JFrame frame = new JFrame("Example of a Frame");
    JButton button1 = new JButton("O.K.");
    JTextField textField1 = new JTextField(20);
    frame.getContentPane().add(textField1);
    frame.getContentPane().add(button1, BorderLayout.SOUTH);
    frame.pack();
    frame.setVisible(true);
  }
}
```

The classes `JFrame`, `JTextField`, and `JButton` are in the package `javax.swing`, whereas `BorderLayout` resides in `java.awt`.

Another way of getting the same result is to make `FrameDemo` a subclass of `JFrame` and have the button and text field as fields. The code is given below.

```
import javax.swing.*;
import java.awt.*;
public class FrameDemo2 extends JFrame {
  private JButton button1;
  private JTextField textField1;
```

```
  public FrameDemo2(String title) {
    super(title);
    button1 = new JButton("O.K.");
    textField1 = new JTextField(20);
    getContentPane().add(textField1);
    getContentPane().add(button1, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
  public static void main(String[] s) {
    new FrameDemo2("Example of a Frame");
  }
}
```

4.5.2 Event Handling

The program we developed in the previous section does not really do anything useful. The three buttons, minimise, maximise, and close, work, but that functionality is provided by the system itself.

To make a Java GUI application do anything useful when users interact with it, we need to handle events. Whenever the user does something on the widgets, for example clicking a button or hitting the enter key while the cursor is in a text field, the system generates what are known as **events.** The default action for events is to do nothing. The programmer must decide what should happen when events occur.

Event handling is best explained via an example. Taking a button click as an example, we first note that such an event generates an action event, represented by the class `ActionEvent` in the package `java.awt.event`. However, that event does not result in any meaningful action unless some object **listens** to it and takes some action in response.

To process action events, therefore, two things must happen.

1. An object must become a listener to an action event by implementing the interface `ActionListener` (in the package `java.awt.event`). An example is given below.

   ```
   public class SomeClass implements ActionListener {
   ```

   Java then knows that objects of type `SomeClass` are capable of handling action events. The interface itself is implemented by coding the method `actionPerformed`, which is the only method in `ActionListener`. This method has just one parameter, which is of type `ActionEvent` and represents the event.

```
public class SomeClass implements ActionListener {
  // fields and other methods
  public void actionPerformed(ActionEvent event) {
    // code to process the event
  }
}
```

2. It is not enough for a class (and thus objects of the class) to have the ability to process events; it must also request that it be told of those events. Suppose that `button1` is of type `JButton`. Then the following code in `SomeClass` requests that objects of type `SomeClass` be notified when action events occur on that button.

```
button1.addActionListener(this);
```

Let us now modify the GUI program, `FrameDemo2`, so that whenever the button is clicked, the program displays some message. We will make it a little more interesting by actually displaying how many times the button was clicked.

For this, we need to remember the number of times the button was clicked; this is done by introducing a field in `FrameDemo2`. The code for `actionPerformed` is then

```
public void actionPerformed(ActionEvent event) {
  textField1.setText("You clicked " + ++count + " times so far.");
}
```

A shortcoming of the program is that when the close button is clicked, the frame disappears, but the process itself remains. What really happens is that the GUI part of the application has exited but the non-GUI part is still alive. Clicking on the close button is a type of event called window event. As you might expect, there is a class called `WindowEvent`, and, you guessed it right, an interface called `WindowListener`, again in the package `java.awt.event`.

However, there are seven methods in this interface. They correspond to actions on the window such as making it an icon, activating it, closing it, etc. Only one of these actions is relevant, which is handled by putting some code within the method called `windowClosing` as shown next.

```
public void windowClosing(WindowEvent event) {
  System.exit(0);
}
```

The complete code for the new version is given below.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class FrameDemo2 extends JFrame implements
                    ActionListener, WindowListener {
  private JButton button1;
  private JTextField textField1;
  private int count;
  public FrameDemo2(String title) {
    super(title);
    button1 = new JButton("O.K.");
    textField1 = new JTextField(20);
    getContentPane().add(textField1);
    getContentPane().add(button1, BorderLayout.SOUTH);
    button1.addActionListener(this);
    addWindowListener(this);
    pack();
    setVisible(true);
  }
  public void windowOpened(WindowEvent event) {
  }
  public void windowIconified(WindowEvent event) {
  }
  public void windowDeiconified(WindowEvent event) {
  }
  public void windowClosed(WindowEvent event) {
  }
  public void windowActivated(WindowEvent event) {
  }
  public void windowDeactivated(WindowEvent event) {
  }
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
  public void actionPerformed(ActionEvent event) {
    textField1.setText("You clicked " + ++count + " times so far.");
  }
  public static void main(String[] s) {
    new FrameDemo2("Example of a Frame");
  }
}
```

Another type of widget that we will use in this book is a **label**, which displays a piece of text or an image. It cannot be used by the user to enter information.

Here is how to create a label.

```
JLabel nameLabel = new JLabel("Name:");
```

The usual method of adding widgets applies to labels as well.

4.5.3 More on Widgets and Layouts

Let us now extend the program to have two buttons side by side in the 'southern' part of the frame. Our goal here is to display different messages when the two buttons are pressed.

The problem presents two difficulties:

1. We can put only one widget directly in `BorderLayout.SOUTH`.
2. We need to know which button is clicked.

To handle the first situation, we introduce a new container called a panel, available via the class `JPanel`. Suppose that `button1` and `button2` are `JButton` objects. Then, we create a `JPanel` object and put the buttons in it, and then put the panel itself in the content pane as shown below.

```
JPanel panel = new JPanel();
panel.add(button1);
panel.add(button2);
getContentPane().add(panel, BorderLayout.SOUTH);
```

Notice that we issue the `add` method on the panel object itself because it has a much simpler organisation than `JFrame`. Panels add the widgets from left to right.

To handle clicks, we need to listen to their occurrences on both buttons. The method `actionPerformed` must be modified to determine which action event—click on `button1` or `button2`—has occurred. The identity of the button is established by asking the event object itself. Every event supports a method called `getSource` that returns a reference to the object that generated the event. The code is thus

```
if (event.getSource() == button1) {
  textField1.setText("Hello");
} else if (event.getSource() == button2) {
  textField1.setText("Hi");
}
```

The complete program is

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class FrameDemo3 extends JFrame implements
                    ActionListener, WindowListener {
  private JButton button1;
  private JButton button2;
  private JTextField textField1;
```

```
  public FrameDemo3(String title) {
    super(title);
    button1 = new JButton("Print Hello");
    button2 = new JButton("Print Hi");
    JPanel panel = new JPanel();
    panel.add(button1);
    panel.add(button2);
    textField1 = new JTextField(20);
    getContentPane().add(textField1);
    getContentPane().add(panel, BorderLayout.SOUTH);
    button1.addActionListener(this);
    button2.addActionListener(this);
    addWindowListener(this);
    pack();
    setVisible(true);
  }
  public void windowOpened(WindowEvent event) {
  }
  public void windowIconified(WindowEvent event) {
  }
  public void windowDeiconified(WindowEvent event) {
  }
  public void windowClosed(WindowEvent event) {
  }
  public void windowActivated(WindowEvent event) {
  }
  public void windowDeactivated(WindowEvent event) {
  }
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == button1) {
      textField1.setText("Hello");
    } else if (event.getSource() == button2) {
      textField1.setText("Hi");
    }
  }
  public static void main(String[] s) {
    new FrameDemo3("Example of a Frame");
  }
}
```

4.5.4 Drawing Shapes

Suppose we want to draw shapes such as squares and circles in a window. This can be accomplished by first creating a `JFrame` and then storing a `JPanel` object within it. Whenever Java thinks the window should be refreshed (examples: the program is de-iconified; the window becomes uncovered) or when the application code makes an explicit request that the window be refreshed, a method called `paintComponent` within the frame is executed, which calls the `paintComponent` method in the `JPanel` class. The method returns nothing (`void`) and has a single parameter of type `Graphics`. Here is an example:

```
public void paintComponent(Graphics g) {
  g.drawRect(30, 75, 100, 50);
  g.drawOval(30, 40, 50, 50);
}
```

The first statement in the method draws a rectangle 100 pixels wide and 50 pixels high. The left edge of the rectangle is 30 pixels from the left edge of the frame and the top edge is 75 pixels from the top of the frame. Within a graphics window, the coordinate values increase as we move from left to right and from top to bottom.

The second statement draws a circle using a method that can draw an oval. The third and fourth coordinates are the width and height of the oval, both of which are the same, so we end up with a circle. The circle fits within a rectangle whose left edge is 30 pixels from the left edge of the frame and top edge is 40 pixels from the top of the frame.

The code for the panel that is stored in the frame is given below.

```
private class DrawingPanel extends JPanel {
  public void paintComponent(Graphics g) {
      g.drawRect(30, 75, 100, 50);
      g.drawOval(30, 40, 50, 50);
  }
}
```

The following code instantiates the panel and adds it to the frame.

```
getContentPane().add(new DrawingPanel());
```

4.5.5 Displaying a Piece of Text

To display a piece of text, the `drawString` method can be used. Suppose we wish to display the text `OOAD` on the screen. In addition to the text, we need to specify the

x and y coordinates of the starting point as parameters to the `drawString` method, which is invoked on the `Graphics` object.

```
g.drawString("Java", 100, 200);
```

The above line causes `Java` to be displayed starting at the point whose x coordinate is 100 and y coordinate is 200. The display uses the graphics object's current settings of font and colour. The reader may wish to consult the Java documentation to get more details.

4.6 Long-Term Storage of Objects

Most, if not all, business systems need to maintain data for long periods of time. Since main memory is volatile and the amount of data that needs to be maintained is large compared to the amount of main memory, many application systems store most of the data on secondary storage and retrieve it as needed.

Files on disk are represented as objects in an object-oriented program. Suppose that we have a file named `f1` on disk, which we would like to read in an object-oriented program. For this, we create an object that gets associated with the file. When the object is manipulated, the file gets manipulated accordingly. The idea is shown in Fig. 4.3. Here object `o1` represents the file `f1`, which resides on disk. The file can be read, written, etc. by manipulating the object.

To put this idea into practice, we need to find a class that can be instantiated to get objects such as `o1`. Usually, such classes are part of the application programming interface supported by the language. For example, in Java there is a class called `ObjectInputStream`, using which we can read files containing objects.

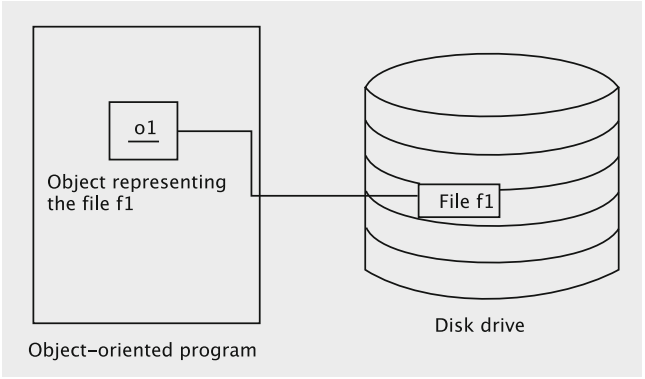

Fig. 4.3 Representation of a file as an object

As we will see later, we run into some difficulties when we read and write objects. To ease the process, we first show how to store and retrieve contents of primitive variables (like `int` and `char`).

The first step is to establish a connection with the disk file. An examination of the package `java.io` shows several possible classes that will let us create such objects. One of these is `FileOutputStream`. The documentation says that this class 'is meant for writing streams of raw bytes such as image data', which implies that we will need the support of other classes as well.

In any case, the code

```
FileOutputStream file = new FileOutputStream("someData");
```

will create a file named `someData` in the current directory.

An examination of the class reveals no useful methods for writing primitive variables. For that, there is a class called `ObjectOutputStream`, which can be constructed as below.

```
ObjectOutputStream output = new ObjectOutputStream(file);
```

One of the constructors for `ObjectOutputStream` accepts an `OutputStream` object as a parameter, and as shown in Fig. 4.4, `FileOutputStream` is a subclass of `OutputStream`.

We are using the constructor`ObjectOutputStream(OutputStreamout)`.

We now proceed to write several types of variables into this file.

```
int i = 7;
char c = 'q';
boolean b = true;
double d = 3.14;
output.writeInt(i);
output.writeChar(c);
output.writeBoolean(b);
output.writeDouble(d);
output.close();
```

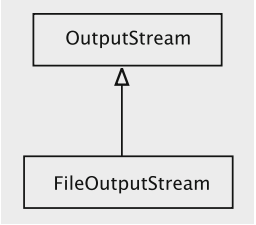

Fig. 4.4 FileOutputStream and OutputStream

To read back what we wrote, we need to create an object of type `ObjectInputStream`. The object can be constructed by first creating a `FileInputStream` object and passing that object to the constructor of `ObjectInputStream`.

```
FileInputStream file = new FileInputStream("someData");
ObjectInputStream input = new ObjectInputStream(file);
```

Next, we read the variables using the object `input`.

```
int i = input.readInt();
char c = input.readChar();
boolean b = input.readBoolean();
double d = input.readDouble();
```

4.6.1 Storing and Retrieving Objects

In this section we address the difficulties that we run into when we try to store objects on disk.

Let us revisit the code we wrote in the previous section and compare the nature of primitive types and objects. The Java API has methods such as `writeInt()` and `readInt()` because `int` is a primitive type in the language. In contrast, user-defined classes such as `Television` or `Account` are not known to the language designer, so there no methods such as `writeAccount` or `readTelevision` in the Java API. The set of application classes is infinite, so it is impossible to support a separate method for reading and writing instances of all these classes!

What is more realistic in a language is to have methods that write any object. So what we can do in the language is write code such below.

```
Television television = new Television();
Account account = new Account();
FileOutputStream file = new FileOutputStream("objectData");
ObjectOutputStream output = new ObjectOutputStream(file);
output.writeObject(television);
output.writeObject(account);
```

In the above, we write a `Television` object and a `Account` object using the same method `writeObject`. When we read, we should expect to retrieve these two objects back as in the code below.

```
Television television;
Account account;
FileInputStream file = new FileInputStream("objectData");
```

```
ObjectInputStream input = new ObjectInputStream(file);
television = input.readObject();
account = input.readObject();
```

4.6.2 Issues in Storing and Retrieving Objects

To gain a basic understanding of how to store and retrieve objects, we need to consider several issues.

Reconstruction

How is the system to reconstruct the object? To see the problem, it is instructive to look at the process of storing and retrieving a variable of a primitive type, say, an `int`. Assume that an `int` variable is represented using the 32 bit, 2's-complement notation. The bit pattern can be written to disk exactly as it appears in main memory. The result is that secondary storage will now contain 32 bits representing an integer. Suppose that the `int` variable contains the value 7. The value in disk will contain the following bit pattern:

```
0000 0000 0000 0000 0000 0000 0000 0111
```

When code such as `int a = input.readInt()` is executed to retrieve the value from the file, the system knows that it must look for a 32-bit sequence of data and interpret it as an integer. So, it reads that many bits from disk and stores them in the variable `a`.

On the other hand, not all objects have the same length and format. So, when an object is to be read back, information about how much to read and what the bits mean should be available.

Complexity

Consider the following class definitions.

```
public class StaffMember {
  private String name;
  private String phone;
  private Department department;
  // constructors and methods
}

public class Department {
  private String departmentName;
  private StaffMember manager;
```

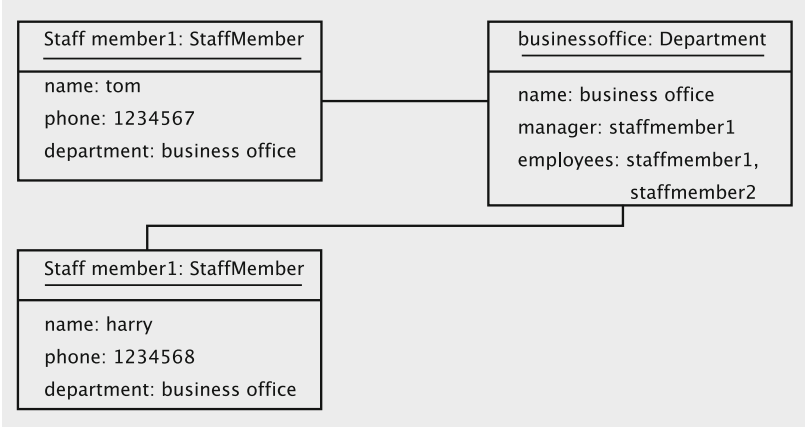

Fig. 4.5 A situation for storing objects

```
    private List employees;
    // constructors and methods
}
```

Even with these relatively simple classes, we can get into tricky situations as shown in the object diagram in Fig. 4.5. There are two staff members, denoted by the two objects `staffmember1` and `staffmember2`. They correspond to employees 'Tom' and 'Harry' with phone numbers 1234567 and 1234568 respectively. Both staff members belong to the same department, Business Office. The business office's manager is Tom and it currently has just two members: Tom and Harry.

The structure is represented using a directed graph in Fig. 4.6, with the objects represented by vertices and references represented by links. Vertices v1 and v2 correspond to the two staff members and vertex v3 represents the business office. The arrows represent references maintained in the objects: for example, the arrow from v1 to v3 indicates that the object for Tom maintains a reference to the object corresponding to the business office.

A little thought reveals some difficulties in storing structures such as the above.

1. The structure is recursive. When we store v1, we need to copy v3 as well, but storing v3 requires that we copy v1. This cyclic nature of the relationship needs to be addressed so that we do not get into an infinite loop.
2. A single object may be referred to from more than one object. For example, v3 is referred to from both v1 and v2. When we write out v1 in Fig. 4.6, for example, we store an instance of v3. When we copy v2 to disk, a naive approach will store v3 once again, as in Fig. 4.7. (The figure does not show all the arcs.) Apart from wasting resources, storing multiple copies of an object means that while reading the data back, these multiple copies will be retrieved, and the resulting configuration after retrieval will be inconsistent with what existed in memory

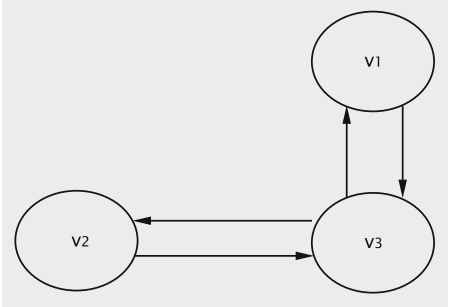

Fig. 4.6 Modelling the object structure using a directed graph

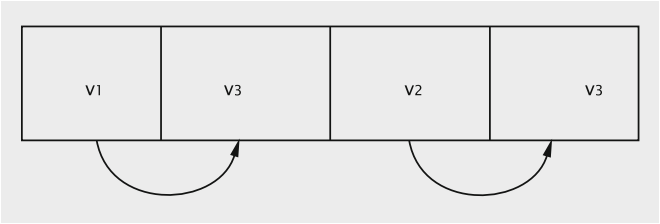

Fig. 4.7 Incorrect storage of the objects shown in Fig. 4.6

prior to the save. After the data in Fig. 4.7 is read back from disk, we end up with the configuration in Fig. 4.8, which is incorrect.

Clearly, a great deal of sophistication is demanded of the application programs to store objects on disk. Typically, we would like to avoid introducing such intricate code into our programs. Since many applications require such a functionality, it is better if such a facility were supported by the language itself. To handle this common problem, the Java designers have come up with a facility known as **serialization**.

4.6.3 The Java Serialization Mechanism

The problems we have discussed can be handled by using the Java serialization mechanism.[1] The major steps in storing a disk avoiding the problems we have discussed are given below. We omit some of the subtle issues involved in the process, deferring these aspects to Chap. 7.

[1]The serialization mechanism is a little more general than simply writing objects to disk. It can be thought of as a mechanism to construct a linear representation of a set of objects which can be used for a multitude of purposes.

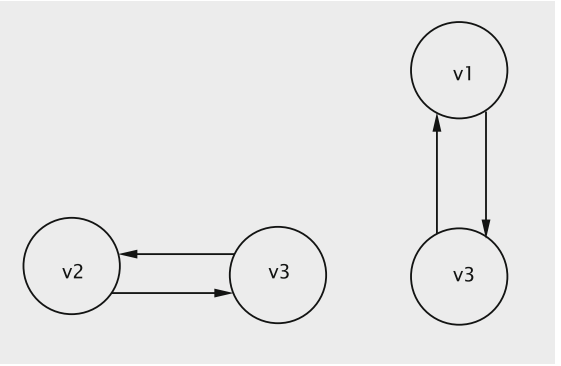

Fig. 4.8 Reconstructed relationships based on data retrieved from disk (see Fig. 4.7)

1. Make every class whose objects need to be serialized implement the interface `Serialisable` in the package `java.io`.
2. Open a disk file using the classes `ObjectOutputStream` and `FileOutputStream`.
3. Use the method `writeObject(Object)` in `ObjectOutputStream` to store objects.

The process of writing out objects using the above approach is called **serialization.**

The reverse process, whereby data written through serialization is read back to memory is called **deserialization**; this is effected as below.

1. Open a disk file using the classes `ObjectInputStream` and `FileInputStream`.
2. Use the method `readObject` in `ObjectInputStream` to read objects. The objects are assigned to variables of the appropriate type. (It is necessary to cast each object before assignment.)
3. Objects must be read back in the order in which they were written.

The `Serializable` interface contains no methods, so it is just a 'marker' to inform the system that the corresponding class is Serializable. Objects of type `Department` and `StaffMember` can be serialized by simply declaring them to be `Serializable`. This is because they contain instance fields, each of which is defined to be `Serializable`. We do this as below.

```
import java.io.Serializable;
public class StaffMember implements Serializable {
  // fields and methods of StaffMember
}
import java.io.Serializable;
public class Department implements Serializable {
  // fields and methods of Department
}
```

The two StaffMember objects and the Department object can be serialized as below.

```
FileOutputStream file = new FileOutputStream("objectData");
ObjectOutputStream output = new ObjectOutputStream(file);
// Create the StaffMember and Department objects
output.writeObject(departmentObject);
```

Since departmentObject contains references to the two staff members, storing it results in the serialization of the staff members as well.

Deserialization can be done as follows:

```
FileInputStream file = new FileInputStream("objectData");
ObjectInputStream input = new ObjectInputStream(file);
Department aDepartment = (Department) input.readObject();
Staffmember member1 = (StaffMember) aDepartment.employees(0);
Staffmember member2 = (StaffMember) aDepartment.employees(1);
```

Although the above code looks simple enough, things do not always work out as easily as might be implied. A major point is that it may not make sense to serialise certain objects. Fields defined as static are not automatically serialized by the Java serialization mechanism although application code may explicitly serialise them via its own code. There are also many Java library classes that are *not* serializable. An example would be the abstract class java.awt.Graphics discussed in the last section. Instances of concrete subclasses of Graphics such as Graphics2D are created by the system and supplied to application programs for drawing on the screen. A Graphics object can be thought of as a collection of the brush (or pen), colour palette, font, etc. Every time the program needs to redraw the screen, the system supplies it with a new Graphics object. Once the program completes the drawing operation, the object is no longer applicable, and a new one will be supplied for a subsequent rendering.

These and some subtle issues related to serialization will be discussed in Chap. 7.

4.7 Discussion and Further Reading

While many of the topics discussed in this chapter may seem very specific to the Java language, we would like to state that they are really Java implementations of well-established concepts in the literature. A sound understanding of the concepts presented here would help the reader learn equivalent technologies in other languages and systems such as C# and .net. It seems that many object-oriented languages

such as Java and C# tend to borrow ideas from each other. So studying a popular object-oriented language well usually helps in understanding the features of another.

Like most other features in the Java language, the sheer size of the GUI library (packages and associated classes) can be intimidating to someone new to this style of programming. The best strategy to understanding the system is to master the basic principles: creation of a window, adding widgets, techniques of using the layout managers, processing events and so on. There are just too many classes and the reader will probably learn quickly that attempts to memorise the methods and their signatures are usually futile.

The members of packages such as `java.awt`, `javax.swing`, etc., is a collection of abstract and concrete classes that collaborate to provide the ability to produce a window on the screen. The window has no specific application-related capability because it is something to be determined, designed and implemented by users and application software designers and implementers. The idea is that the JDK classes themselves form a reusable design that the application development community may adapt as it deems fit. Such a reusable collection of classes is called a **framework**.

In the same vein, we would also like to emphasise the importance of being productive: in the software engineering arena, this translates to being able to gain a good understanding of the problem to be solved and provide speedy solutions. Since technology changes fairly quickly, it is important to understand the general principles behind specific features available in a language and, at the same time, be an effective toolsmith, which means the ability to use available tools to craft solutions rather than 'reinvent the wheel'.

As alluded to in the footnotes, the technique of serialization can be applied for purposes other than storing objects on the disk. Notice that in serialization we write objects to an `ObjectOutputStream` object, producing in effect a sequence of bits that represent one or more objects. We stored the serialized version of the objects on disk by directing the stream (`ObjectOutputStream` object) to a `FileOutputStream` object. Instead, we could transfer these bits to any Java Virtual Machine (JVM), perhaps even over a network to a JVM running on a geographically distant site. This technique is employed for implementing distributed object-oriented systems and the corresponding technology is called **Remote Method Invocation (RMI)**. We will discuss RMI in Chap. 12.

Projects

1. Consider the following interface:

```
import java.util.*;
public interface Deque {
  public boolean addAtTail(Object value);
  public Object removeElementAtTail();
  public Object getElementAtTail();
  public boolean addAtHead(Object value);
  public Object removeElementAtHead();
```

```
    public Object getElementAtHead();
    public int size();
    public void clear();
    public Iterator iterator();
}
```

The interface represents a double-ended queue in which members can be added and removed at either end. The method names should convey the semantics of the operations. Implement the interface using the class `java.util.LinkedList`.

2. This project requires you to explore the Java GUI framework on your own, determine the appropriate classes to use, and write two Java classes that create a GUI program. The first class `CourseProcessor` is the GUI interface, and the second class `Course` stores information about a single course.
   The program accepts and stores information about courses offered in six departments: Computer Science, Mathematics, Chemistry, Physics, Botany, and Zoology.
   The user can do the following.
   (a) Enter information about a course by selecting a department name from a combo box, typing in the course number, name, number of credits and then pressing the enter button. The interface checks that the entries are non-empty (display error message otherwise) and then creates a `Course` object using the information and then stores the object in a `java.util.Vector` object.
   (b) Ask to list all courses by clicking on a button labelled display (all). All the objects in the `Vector` object are displayed. There is a scrollbar that allows viewing records that cannot be displayed in the given space. Also, note the department codes such as CS and MATH inserted by the program.
   (c) Ask to list courses of a given department by clicking on a button labelled display (dept.). Courses for the selected department (via the combo box) in the `Vector` object are displayed.
   (d) Quit instantly by clicking on the window's 'close' button, or close (after a confirm dialog) via an 'exit' button within the frame.
   Department codes Store the codes associated with departments in static arrays in the class `Course`. This mapping should not be duplicated and should be used consistently and reliably within your code. The codes are given below.

Computer Science	CS
Physics	PHY
Chemistry	CHEM
Mathematics	MATH
Botany	BOT
Zoology	ZOO

3. Write a program that draws shapes that look like houses. A house is made up of a rectangle, on top of which is placed a triangle. You need to write two classes: one that represents a single house and another that creates and draws the houses. It should be possible to specify the size of the house. You may make assumptions on the relationships between the length and height of a house.
4. Write a Java program that accepts the names of a set of files while it is started from the command line and processes these files as specified below.

   (a) The user must supply at least two file names. Thus, the following are among the infinite number of valid commands.

   ```
   java Processor infile outfile
   java Processor infile1 infile2 outfile
   java Processor infilea infileb infilec myoutfile
   ```

   The following are invalid.

   ```
   java Processor
   java Processor fileName
   ```

   For invalid commands, the program prints an error message and terminates.

   (b) The very last parameter is the name of an output file. All the other file names are input files. All are pure text files (no binary data) and you may assume that the output file is also not specified as an input file. Thus, you do not have to worry about situations such as

   ```
   java Processor infilea infileb infilec infileb
   ```

   (c) Each of the input files is read and copied to the output. The files are opened and read in the specified order.
   (d) If one of the input files is missing, it is skipped and the next input file, if any, is processed.
   (e) If there is a problem opening the output file, the program displays an error message and exits.
   (f) Each line of each input file is read and copied to the output file. Just prior to copying a line to the output, the name of the input file, the line number in the input file and the line number in the output file are all given as output.

4.8 Exercises

1. Take a look at the package `java.util.*` and the documentation for the classes `Vector`, `LinkedList` and `ArrayList`. Compare them for their features, the interfaces they implement and the class hierarchy that leads to each of them.

2. The following code attempts to write an instance of `C1` followed by an instance of `C2` onto disk and recreate the objects by reading the data from disk. However, there are errors in the code. Correct them.

```
import java.io.Serializable;
public class C1 implements Serializable {
}
import java.io.Serializable;
public class C2 {
}

import java.io.*;
public class C3 {
public static void main(String[] s) {
    FileOutputStream fos = new FileOutputStream("f1");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    C1 c1 = new C1();
    C2 c2 = new C2();
    oos.writeObject(c1);
    oos.writeObject(c2);
    FileInputStream fis = new FileInputStream("f1");
    ObjectInputStream ois = new ObjectInputStream(fis);
    C2 anotherC2 = (C2) ois.readObject();
    C1 anotherC1 = (C1) ois.readObject();
    }
}
```

3. The Java compiler flags an error if a checked exception is not caught. Study Java documentation to see how Java determines whether a certain exception is a checked exception or not.
4. An example of an unchecked exception is `NumberFormatException`. Come up with an example where it is advisable to catch exceptions of this type.