

Graphical User Interfaces

Essential reading

Bishop, Judy *Java Gently*. (Addison-Wesley Publishing Company, 2001) third edition [ISBN 0-201-71050-1].
Chapter 9

Lewis, John and Loftus, William *Java Software Solutions - Foundations of Program Design*. (Addison-Wesley Publishing Company, 2000) second edition [ISBN 0-201-61271-2].
Chapter 9

Further reading

Barnes, David J *Object-oriented Programming with Java*. (Prentice Hall, 2000) [ISBN 0-13-086900-7].

The JFC Swing Tutorial. (Addison-Wesley, 1999) [ISBN 0-201-43321-4].

In this chapter, we introduce the main graphical functions of Java and reveal the overall structure of the packages.

Useful packages

Java has well equipped graphical user interfaces and essential tools and libraries. These are mainly provided in the form of classes in two packages. One is `java.awt`, i.e. Abstract Windowing Toolkit; the other is `java.swing`. They are sometime also called *the AWT package* and the *Swing package*. In fact, these packages can be so large that they may require more resources, such as a larger computer.

Java provides so many useful classes (see the section above) for graphical user interfaces for reasons. Considering the process of drawing even a simple figure on the screen, you would soon realise that it involves a lot of tasks. For example, in which area would the figure be displayed? Would the figure occupy the whole screen, or part of the screen? How would we stop displaying one figure and start to display another? How would every small component in a figure change respectively when we enlarge the figure? Without these standard packages in Java as the foundation, one can not easily build a graphical user interface in practice.

AWT provides a *framework* for structuring generic solutions to a common problem.

Package AWT

This package contains the following classes:

graphics - for drawing shapes, lines and images, fonts, colours etc.

components provide items such as buttons, text fields, menus, scroll bars

¹An event is an action taken by the user. See next chapter for details.

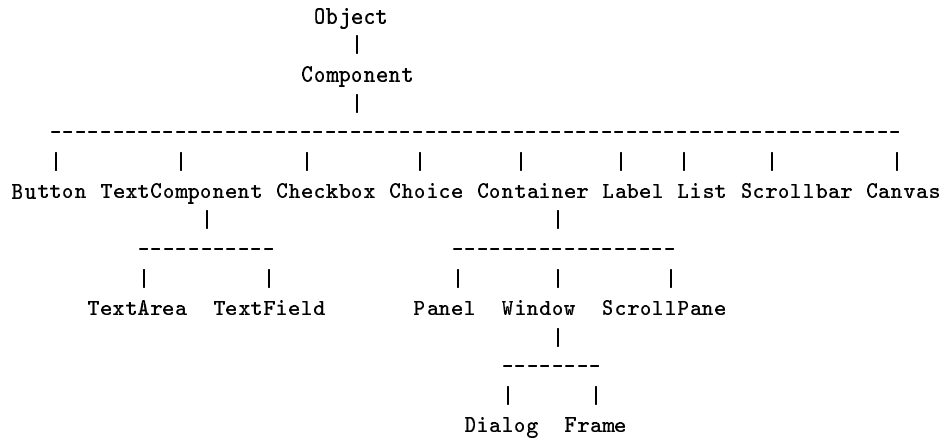
²See chapter 8 of this subject guide for details.

layout managers arrange the layout on the screen. Note there are a variety of standard layout managers, each of which places components in a slightly different ways.

event handlers, listeners and adapters java.awt.event handles external events¹

image manipulation java.awt.image incorporates images in different formats²

The AWT class hierarchy is as below:



Abstract classes

There are *four* main abstract classes:

Components
Container
MenuComponent
Graphics.

Other abstract classes include:

FontMetrics
Image
PrintJob
Toolkit.

One interface is:

LayoutManager.

Graphics in a window

We first look at how to display a figure on the screen.

From our experience before, we know that it would be no good if the whole screen is used to display one figure, for we may need to do some other things at the same time. So anything for display will be put in a window in Java. A *frame* is a free standing window that can be repositioned anywhere on the screen.

Creating a frame

Before using any built-in facilities for graphics, colour, animation or even sound, we need to create a frame by inheriting `Frame` class. This includes three routine segments of statements:

1. Import `java.awt.*`;
2. Define a new class that inherits `Frame`;
3. Initialise the frame.

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

class CLASSID extends Frame {
    CLASSID {
        setTitle ("WINDOWTITLE"); /* optional */
        setSize (WIDTH, HEIGHT);
            /* set the size of the window */
        setVisible (true);
            /* activate the drawing of the frame */
    }
    ...
        /* other statements */
new CLASSID (); /* in main */
    ...
        /* other statements */
=====
```

Adding graphics to the frame

This is implemented by redefining `paint` method either in the class that extends `Frame` (above) or in some other object of a class that extends `Canvas`.

```
=====
public void paint (Graphics g) {
    ... /* Calls to methods in Graphics,
            prefixed by g */
}
=====
```

A `repaint` method can also be used by the program for the same effect.

```
=====
public void repaint (Graphics g) {
    ... /* Calls to methods in Graphics,
            prefixed by g */
}
=====
```

Once we have set up a frame, many things can be done by calling any methods in graphics class in the following format:

Example 11 Drawing a flag

The following program will draw a flag on a window.

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

class FlagMaker1 extends Frame {
    FlagMaker1 () {
        add ("Center", new Flag());
        //    addWindowListener(new WindowAdapter () {
        //        public void windowClosing(WindowEvent e) {
        //            System.exit(0);
        //        }
        //    });
        setTitle("A Flag");
        setSize(300,200);
        setVisible (true);
    }

    public static void main (String [] args) {
        new FlagMaker1 ();
    }
}

class Flag extends Canvas {
    public void paint (Graphics g) {
        g.setColor (Color.black);
        g.fillRect (40, 40, 200, 40);
        g.setColor (Color.red);
        g.fillRect (40,80,200,40);
        g.setColor (Color.yellow);
        g.fillRect (40, 120, 200, 40);

        g.setColor (Color.black);
        g.drawString ("Germany", 100, 180);
    }
}
```

Note After compiling and running the program, you will find that the window with the display of the flag cannot be closed unless you remove the five comment signs (i.e. //s in the program).

Closing a window

The reason that we could not close the window in the example **Drawing a flag** is that we did not make any arrangement to deal with the user's clicking the mouse. In other words, we did not handle the event of clicking³. By removing the //s, we actually insert five lines of statements to make the window stop.

³*Events will be discussed in the next chapter.*

The statement to close a window can be as follows:

```
Example 12 addWindowListener(new WindowAdapter () {  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
});
```

Swing

⁴*This comes from a code
name for a further set of
GUI tools.*

Swing⁴ provides further tools in Java. These tools can be implemented on independent operating systems. In terms of the size of the GUI and Swing, GUI is smaller and quicker at run time. On the other hand, Swing library contains new graphical components such as split panes and progress bars, etc. The user will have greater control over the quality of the graphical components and can change the appearance dynamically.

For the interested reader, further details can be found in the books listed under **Further Reading** at the beginning of this chapter.

Inheritance and Abstraction

Essential reading

Bishop, Judy *Java Gently*. (Addison-Wesley Publishing Company, 2001) third edition [ISBN 0-201-71050-1].
Chapter 9

Winder, Russel and Roberts, Graham *Developing Java Software*. (published by John Wiley & Sons Ltd., 1998) second edition [ISBN 0-471-60696-0].
Chapters 30, 31 and 29

In this section, we discuss further issues on Inheritance as well as introducing Abstraction.

The power of classes

We have seen so far that objects can easily be created from classes. In this chapter, we shall look at another powerful function of classes which is to do with how to make use of the existing classes and how to prepare for the changes at the early stage of a software project.

In object-oriented programming, there are three powerful techniques in developing a large software package, namely:

- Composition: creating an object based on an existing class which we have seen a lot of examples in previous chapters;
- Inheritance: creating a class based on an existing class and allowing not only reuse of the existing classes but also adding more properties.
- Abstraction: creating a class containing only some ideas or the outlines but no details.

The goal of inheritance and abstraction is to:

- cut down on repetition as much as possible
- aid reuse of existing classes and form a class hierarchy.

Inheritance

Inheritance is a technique to

- concentrate on reuse of a known class;
- allow a design of additional versions of a class later.

To ease the description, we define commonly used terms and concepts below:

extend A new level of class *extends* the class above it and can have more fields and methods.

superclass is the parent (i.e. original) class.

subclass is the child class.

transitive inheritance is transitive.¹

Example 13 Suppose that we have a class **Sphere** which represents a sphere².

```
class Sphere {
    private int x=100, y=100;

    public void setX(int newX) {
        x=newX;
    }

    public void setY(int newY) {
        y=newY;
    }

    public void display(Graphics g) {
        g.drawOval(x,y,20,20);
    }
}
```

Here `g.drawOval` is the library method `drawOval`. Suppose that we now need a new class, called **Bubble**, similar to **Sphere** but to describe a bubble. The size of the bubble needs to be changeable and the location movable vertically.

Instead of writing a new class that uses the code that is already in **Sphere**, we let (the new) **Bubble** inherit variables and methods from (the old) **Sphere**. In other words, we *extend* (the old) **Sphere** to get (the new) **Bubble**. (The old) **Sphere** is the superclass of (the new) **Bubble** and (the new) **Bubble** is the subclass of (the old) **Sphere**.

The (new) **Bubble** can be:

```
Example 14 class Bubble extends Sphere {
    private int radius=10;

    public void setRadius(int newRadius) {
        radius=newRadius;
    }

    public void display(Graphics g) {
        g.drawOval(x,y,2*radius, 2*radius);
    }
}
```

Note The old class **Sphere** describes objects that do not move and whose size cannot change. In addition to the public variables and all the public methods,

¹ This is to say - if class *A* inherits class *B* and *B* inherits class *C*, then *A* inherits *C*.

² Note: you need to add a main method and import `java.awt.*` if you want to try this example on the machine.

the new class `Bubble` has also declared a variable and a method. Finally, a new version of the method `display` is included in the new class `Bubble`. The new version replaces the old version. In this case, we say that the new version *overrides* the old version.

Multilevel operations

The operations among the subclass objects and the superclass objects are called *multilevel operations*. There are rules for multilevel operations in Java. Here is a summary of the operations for *assignment*:

- A subclass object can be assigned to an object of its superclass
- A superclass object can be assigned to a subclass with *cast*.

Techniques in working in a hierarchy

Some terms are commonly used in describing inheritance. They are briefly summarised here:

Superconstructing A subclass can initialise variables

Shadowing variables A subclass can replace data with its own version

Overriding methods A subclass can supply its own version of a method already in a superclass

Dynamic binding Overriding uses *dynamic binding* to find the correct method.

Abstraction

In programming design, especially a large programming design, we often come across the situation at a stage where we may

- not be sure about some details
- not know some details
- choose to decide some details later.

but we know this part should be included in the program, and we know the relationship of this part to other parts of the program. Java provides two particular kinds of classes called *abstract class* and *interface* which allow program designers to ignore certain details in the class and to leave the details to be filled in at a later stage. This technique is called *abstraction*.

Abstraction allows the designer to concentrate on the essentials of a class, in other words:

- what a class or method does
- the behaviour and interface of one kind to the world
- all the details to be filled later.

Two ways to achieve abstraction are to define an

- interface
- abstract classes.

Abstraction through interfaces

Java provides a special kind of class called *interface* that defines specifications of a set of methods.

1. Interface declaration

```
=====
interface INTERFACENAME {
    ... // method specifications
=====
}
```

2. Interface implementation

```
=====
class CLASSNAME implements INTERFACENAME {
    ... // bodies for the interface methods
    ... // own data and methods
=====
}
```

Example 15 Machines that move.

We could define typical operations used on simple movable machines as an interface below:

```
interface Movable {
    boolean start ();
    void stop ();
    boolean turn (int degrees);
    double fuelRemaining ();
    boolean changeSpeed (double kmPerHour);
}
```

Later on, we can implement the interface for a plane as a movable machine as follows:

```
class Planes implements Movable {
    boolean start () {
        ... // starting actions
    }

    void stop () {
        ... // stop actions
    }

    boolean turn (int degrees) {
        ... // turning actions
    }

    double fuelRemaining () {
        ... // return the amount of plan fuel remaining
    }

    boolean changeSpeed (double kmPerHour) {
        ... // accelerate or decelerate if kmPerHour is negative
    }
}
```

Abstract methods and classes

Abstract methods and classes in Java are convenient and useful in putting different chunks of programs together in large software systems.

Abstract methods provide *place holders* for methods mentioned at one level but implemented at other levels lower down.

An abstract class has at least one abstract method.

Example 16 Define a class to compute the area of two dimensional graphical objects, including circles and rectangles.

We know, for anything to be displayed on the screen, that x- and y-coordinates on the screen have to be provided, but the method to compute the area depends on the shape of the objects. We could, therefore, write general purpose methods of taking x- and y- data, but define the method `computeArea` as an abstract method to leave the details until the specific shape of the object is specified. The code looks like the following:

```

public abstract class Shape {
    protected int x, y, width, height;

    public void setX(int newX) {
        x=newX;
    }

    public void setY(int newY) {
        y=newY;
    }

    public void setWidth(int newWidth) {
        width = newWidth;
    }

    public void setHeight(int newHeight) {
        height = newHeight;
    }
    public abstract float computeArea();
}

```

Later we will define the abstract class as follows. The class `Rectangle` inherits from the class `Shape`, overriding the method `computeArea`:

```

public class Rectangle extends Shape {
    public float computeArea() {
        return height*width;
    }
}

```

Serialization

Serialization is a special facility used in Java to save data in a file. An object can be converted into a simple stream of bytes to save in a file. The object can be reconstructed correctly later when it is read in from a file.

Although it is important, Serialization is not required for this course. The reader is recommended to read section 9.5 on page 373 if interested.

Learning outcomes

Having read this chapter and consulted related material you should:

- understand the basic philosophy of abstraction and inheritance
- appreciate the power of inheritance, interfaces and abstract classes
- have learnt the use of abstract methods and classes.