Object-oriented programming in Java
Course team
Developer: Dr Man Leung Wong, Lingnan University
Designer: Mr Chris Baker, OUHK
Coordinator: Dr Tak Sing Li, OUHK
Members: Dr Reggie Kwan, OUHK
Mr Tai Loon Fong, OUHK

External Course Assessor
Prof. To Yat Cheung, City University of Hong Kong

Production
ETPU Publishing Team

Acknowledgement
The OUHK gratefully acknowledges a grant of $100,000,000 from the Hong Kong Government designated for course development.

MT311 Programming Languages and Java Programming was developed in part using resources provided by this generous grant.

All rights reserved.
No part of this material may be reproduced in any form by any means without permission in writing from the President, The Open University of Hong Kong.

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong
# Contents

Overview 1  
Objectives 2  
Object-based programming 3  
Objects and classes 3  
Initializing class objects: constructors 5  
Access modifiers 6  
Class scope 8  
Creating packages 9  
Package access 11  
Using overloaded constructors 12  
Using the this reference 14  
Finalizers 14  
Static class members 15  

Object-oriented programming 18  
Inheritance 18  
Relationships between superclass objects and subclass objects 22  
Constructors and destructors in creating a subclass object 26  
Polymorphism 29  
final methods and classes 32  
Abstract superclasses and concrete classes 32  
Interface 38  

Summary 45  
Feedback to activities 46  
Solutions to self-tests 53
Overview

This unit gives an overview of object-oriented programming concepts in Java and looks at how they relate to structuring Java programs.

In object-oriented programming, problems are solved in terms of objects. Any components in the problems that are being solved are represented as an object. Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: you can create variables of a type called objects or instances and manipulate those variables through functions called methods.

This unit begins with defining what an object is. We then look at how to create an object using constructors (which act as initializers) and at how to destroy objects through garbage collectors and finalizers (which are used to clear up objects which are no longer needed). This unit also gives an overview of the concept of a reference to an object. Then we cover how a code is packaged together, and investigate the keywords package and import. Next, we learn why some parts of the code are exposed while others are hidden by introducing the keywords public, private and protected.

After looking at how objects and classes operate in Java programming, we will consider another important concept in object-oriented programming which is known as inheritance. You will learn how to create a family of types through inheritance and manipulate a family of objects. With polymorphism and dynamic binding, you will see how to make your code extensible.

This unit is a lengthy one, and should take you about three weeks to complete (or about 19 hours of study time). Object-oriented programming is a very useful concept in all programming languages for developing stand-alone applications and Internet based programs.

Please plan your time carefully. If you have any difficulties, please contact your tutor.
Objectives

By the end of Unit 2, students should be able to:

1. Explain the concepts of information hiding and encapsulation.
2. Create Java classes.
3. Control access to instance variables and methods.
4. Create class variables and class methods.
5. Appreciate the advantage of inheritance.
6. Apply polymorphism to develop extensible and maintainable software.
7. Create abstract classes.
8. Create interfaces.
Object-based programming

All objects, while being unique, are also a part of a class of objects (for example, a person is unique but is also part of the class that is called ‘human beings’). The class of human beings has a set of characteristics and behaviours in common which are different to those of the class of dogs or the class of cats. Therefore, when we first define a particular object in our program, we have to first define its class. Hence, before we declare a variable of primitive data type int, we have to first define what int is. In the case of human beings, each human being has a set of variables (such as height, age and weight) and methods (or behaviours, such as sleeping and eating). Each human has his or her own set of variables and can call the methods independently. It is not difficult to understand that if a human always calls the method eat(), the value of his or her weight variable will increase.

Since a class describes a set of objects that has identical characteristics (instance variables) and behaviour (methods), a class is really a data type because an integer number, for example, also has a set of methods and instance variables. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types (classes) specific to your needs. In this section of the unit, we will learn how to define a class and create an object from a defined class.

Objects and classes

Each object within a class has its own set of variables and methods. We can compare objects within classes to individual people in a group. For example, imagine that we have two objects, namely Mary and John, that both belong to the human class. Mary has her own set of variables: weight, height, age and phone number etc. which are different in value compared with that of John. Mary may keep some of her variables, such as age and weight, as private data (which are not accessible to other humans) but she may allow John to access other data such as her phone number. Also, Mary may like John to execute one of his behaviours, like buyingFlower() for her, while not allowing John to execute another method, Eating() for her. The relationship between Mary’s behaviour and John’s behaviour is a useful metaphor for understanding the technique of setting accessing authorities through access modifiers. This technique will be discussed in more detail later in this section of the unit.

Defining classes

In Java, new Abstract Data Types are defined using the class declaration. A class is the template for an object and it is a way of encapsulating all of the features of a particular set of objects. A class is a template for an object, just like data type int. The difference is that you have to tell Java the characteristics of the class you created but when you define a data type, int, the compiler knows what an integer is.
You define a class by using the class keyword along with the class name. The class body is marked off by curly braces just like a program block. The following is a complete Java class.

```java
public class Time1()
{
}
```

This is an empty class; it does nothing. You can, however, compile the class and create an object from it. To make Time1 a useful class, it must have some attributes (or instance variables) and methods. To declare attributes in a class is to type the field followed by the name of the attribute, like this:

```java
public class Time1
{
    int hour;
    int minute;
    int second;
}
```

The above line declares three instance variables of type integer. You can see that hour, minute and second are instance variables of the Time1 class. Now, you can define different instances in the Time1 class, each with the setting of hour, minute and second. However, how can you fix the time when it is created? This requires a special kind of method, known as a **constructor**.

A constructor is a **public** method (a method that can be accessed anywhere in a program) with the same name as the class. Constructors cannot specify return type or return values.

### Creating an object from a defined class

When you create a class you are describing how objects of that class look and how they will behave. You don’t actually get anything until you create an object of that class with `new`. How do the `new` and constructor work together to create a new instance? The following example shows you how you can create an object from the Time1 classes:

```java
Time1 aTime = new Time1();
```

The new operator creates an object as the program executes by obtaining enough memory to store an object of the type specified to the right of `new`. The process of creating new objects is also known as **creating an instance**. The operator `new` is known as the **dynamic memory allocation operator**. After allocating enough memory, the constructor is called to initialize the instance variables. The next section of this unit explains how class objects are initialized. However, before we look how this is done, please try the following self-test to check your understanding of objects and classes so far.
Self-test 2.1

1. What are the two main elements of a class?
2. How does a class relate to an object?

Initializing class objects: constructors

You should recall that Java does not allocate memory for objects when they are declared but rather when the instance is created by the keyword `new`. It makes sense that whenever an object is created, we have to do some initialization to set the values of instance variables for each object. In Java, the class designer can guarantee initialization of every object by providing a special method called a constructor. When an object is created, its members can be initialized by a constructor method. The programmer provides the constructor, which is invoked automatically each time an object of that class is instantiated.

A constructor can do anything a normal method can, but usually it is used simply to initialize variables within the object to some starting value. The name of the constructor method is the same as the name of the class in which the constructor is declared, and constructor methods cannot have return values.

When `new` is used to create a new object, the calling syntax is:

```
Ref = new ClassName ( arguments );
```

`Ref` is a reference of the appropriate object, `new` indicates that a new object is being created, `ClassName` is the type of the new object and `arguments` specify the starting values.

Notice that the difference between declaring a primitive data type, like `int` and declaring an Object type is the use of the `new` keyword. The use of the `new` keyword:

1. allocates necessary space to store the object in `ClassName`
2. invokes the constructor
3. returns a reference to the appropriate object.

The constructor, `Time1()` is defined as follows:
public class Time1 {
  int hour;
  int minute;
  int second;

  public Time1() {
    setTime(0, 0, 0);
  }

  public void setTime(int h, int m, int s) {
    hour = ((h >= 0 && h <= 24) ? h : 0);
    minute = ((m >= 0 && m <= 60) ? m : 0);
    second = ((s >= 0 && s <= 60) ? s : 0);
  }
}

As you can see, when the constructor, Time1() is called, it calls another method, setTime consequently to set the value of hour, minute and second. In general, constructors are used to initialize a class’s fields and perform tasks related to creation like performing initial calculations.

The public keyword is important because you want to be able to create an object from the class anywhere in your program. When you create an object like the above line, the corresponding constructor will be called, i.e. h, m and s are all set to zero.

  hour = ((h >= 0 && h <= 24) ? h : 0);
  minute = ((m >= 0 && m <= 60) ? m : 0);
  second = ((s >= 0 && s <= 60) ? s : 0);

You should be able to define classes in Java. To help consolidate this knowledge, there is now a self-test for you to complete.

**Self-test 2.2**

1. Write a basic, empty class called MyClass.

2. Add to MyClass a string data field called myField.

3. Add to MyClass a constructor that accepts a starting value for myField as its single argument, and public methods for setting and retrieving the value of data1. Call these methods setData() and getData().

**Access modifiers**

As mentioned earlier in this section, some data and methods are kept private while others are not. To achieve this accessing authority, we use
access modifiers. The purpose of access modifiers is to find which objects have access to your object’s methods and instance variables. In general, you want to keep internal attributes private to enforce information hiding, then provide public methods to access and manipulate the data. For example, you may keep the variable weight as private data but provide a public method to update the value of weight.

Any instance variable or method declared with the member access modifier public is accessible whenever the program has access to an object of that class. Any instance variable or methods declared with the member access modifier private is accessible only by a method of that class.

If we are declaring hour, minute and second in the class Time1 as private instance variables of the class we can write the following:

```java
public class Time1 {
    private int hour;
    private int minute;
    private int second;
}
```

A user cannot manipulate these statements directly; i.e. the following statements are illegal:

```java
Time1 HongKongTime = new Time1();
HongKongTime.hour = 21;
```

A programmer can only use the public method, setTime, to modify the time.

You may refer to our textbook for a detailed description of how to implement a time abstract data type with a class. The text basically covers what we have just discussed above, but does so in greater detail. You might want to skim through the reading, rather than read it word for word.

**Reading**

Deitel and Deitel, section 8.2, pages 343–50.

**Self-test 2.3**

1. How do you create an object from a class?

2. How do you use a class that is defined in a different file than the file that accesses the class?
Class scope

We will now see how to define a simple class with its own instance variables and methods. As in procedural programming languages in which variables have an accessing scope, an accessing scope also exists in object-oriented programming languages.

Class members are accessible by all methods of that class. Outside a class’s scope, class members are referred to by using the dot notation. If a method defines a variable with the same name as an instance variable, the instance variable is hidden in the method scope. A hidden instance variable can be accessed in the method by preceding its name with the keyword this and the dot operator as in this.x.

Here is an example of how a hidden instance variable can be accessed:

```java
public class ptThis {
    private int x = 10;
    private int y = 20;
    public void Display() {
        int x = 40;
        int y = 50;
        System.out.println("(" + x + ", " + y + ")");
        System.out.println("(" + this.x + ", " + this.y + ")");
    }
}
public class testThis {
    public static void main( String args[] ) {
        ptThis pt;
        pt = new ptThis();
        pt.Display();
    }
}
```

Program Listing 2.1.

The x and y in the following statement refer to the variables defined in the method, i.e. x = 40 and y = 50.

```
System.out.println("(" + x + ", " + y + ")");
```

However, in the following line, the x and y refer to the instance variables in the class scope, i.e. x = 10 and y = 20.

```
System.out.println("(" + this.x + ", " + this.y + ")");
```

Class scope is described in the following reading from our textbook:

**Reading**

Deitel and Deitel, section 8.3, page 351.
Class scope is just like the scope of variables in different subroutines in procedural programming. The reading mentioned that the this reference can be used to access hidden instance variables, as we demonstrated in the example above.

Creating packages

Whenever you want to use a defined class in your program, the compiler must know how to locate it if they are not defined in the same file. Imagine that you want to use a class of a particular name, but more than one class exists with that particular name. Or worse, imagine that you’re writing a program, and as you’re building it you add a new class to your library that conflicts with the name of an existing class. In order to eliminate ambiguities such as these, you must tell the Java compiler exactly which classes you want by using the import keyword. import tells the compiler to bring in a package, which is a library of classes.

You may not realize that you have already been using Java packages. In Unit 1 of this course you imported packages into your source code with this code:

```java
import javax.swing.JOptionPane;
import java.text.*;
```

Each of these lines starts with the word java or javax, followed by a package name and the class you want to import. Packages in Java are actually groups of related classes. These are similar to libraries in other computer languages. The above statements are used to import certain classes from some existing classes. The asterisk means that Java should import all of the classes of the java.text package into the program you’re writing.

Packages are actually directory structures used to organize classes and interfaces. Packages also provide a mechanism for software reuse. Because each Java class is usually located in a separate source file, the grouping of classes provided by a hierarchy of packages is analogous to the grouping of files into a hierarchy of directories on your file system. All the packages in the Java API are stored in the directory java or javax. The Java compiler reinforces this analogy by requiring you to create a directory hierarchy under your class directory that exactly matches the hierarchy of the packages you have created, and to place a class into the directory with the same name and level as the package in which it’s defined.

To use packages as a mechanism for software reuse, choose a package name and add a package statement to the source code file for the reusable class definition. The package statement must be the first statement in the file (excluding comments and white space of course) as in the example below:
package com.deitel.jhtp3.ch08;
public class MyCar {
    private int seat;
    public MyCar ()
    {
        seat = 5;
    }
}

Program Listing 2.2.

Then, compile the class so it is placed in the appropriate package
directory structure. After compiling the package, a directory called com
will be created (or located) under your default class directory. com
contains a directory called jhtp3, and jhtp3 contains a subdirectory
called ch08. A file with .class extension will be created under ch08.

Once the class is compiled, the class can be imported into programs. For
example:

import com.deitel.jhtp3.ch08.MyCar;

The following reading provides a more detailed explanation of how
packages are created.

Reading

Deitel and Deitel, section 8.13, pages 379–84.

In addition to elaborating on our discussion of package creation, the
text describes a naming convention for packages on page 339. This
convention is useful if we are working in a shared environment.

Self-test 2.4

1 What is a package?

2 How do you tell Java that a source-code file uses a particular
package?

3 Create a new package using the source code in Fig. 8.3, and verify
that appropriate subdirectories and files are created.

4 Find out the physical directory of your classes directory.
**Package access**

We have learnt how to use access modifiers to set the authority of instance variables and methods. However, what happens if you give no access specifier at all? When no member access modifier is provided for a method or variable when it is defined in a class, the method or instance variable is considered to have **package access**.

Consider the following coding:

```java
public class PackageData {
    int x;    // no member access modifier specified
    // package access instance variables
    String s;  // no member access modifier specified
    // package access instance variables

    // constructor
    public PackageData ()
    {
        x = 0;
        s = "Hello";
    }
}
```

Program Listing 2.3.

You should notice that both `x` and `s` are package access instance variables. The following program tries to update the value of `x` and `s` by direct access.

```java
public class PackageDataTest {
    public static void main ( String args[] )
    {
        PackageData d = new PackageData ();
        String output;

        output = "After instantiation: \n" + d.toString();
        d.x = 77;         // changing package access data
        d.s = "Good bye"; // changing package access data
    }
}
```

Program Listing 2.4.

If a program uses multiple classes from the same package, these classes can access each other's package-access methods and data directly through a reference to an object.

The next reading provides an example of package access.

**Reading**

Deitel and Deitel, section 8.14, pages 384–86.
You can check your understanding of the reading by doing the following activity.

---

**Activity 2.1**

Every Java class has its own class file. Try to modify the coding in Fig. 8.18 by putting the two classes, PackageDataTest and PackageData in different packages. Run the program again and compare the results.

---

**Using overloaded constructors**

A constructor’s name is determined by the name of its class; hence, there can be only one constructor name. However, what happens if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way or by reading information from a file. You would then need two constructors, one that takes no arguments (the default constructor), and one that takes a **String** as an argument, which is the name of the file from which to initialize the object. Both are constructors, so they must have the same name — the name of the class. Thus, **method overloading** is needed to allow the same method name to be used with different argument types.

Methods of a class can be **overloaded**, and overloaded methods must have different signatures. Constructors can also be overloaded to provide a variety of means for initializing objects of a class. We can have as many constructors for a class as we want provided that these constructors have distinct signatures. Initializers are passed as arguments to the constructors of the classes.

For example, if the class `Time2` is defined as follows:

```java
public class Time2 {
    private int hour;
    private int minute;
    private int second;
    public Time2() {
        setTime ( 0,0,0 );
    }
    public Time2( int h )
    { setTime ( h, 0 , 0 ); }  
    public Time2 ( int h, int m )
    { setTime ( h, m , 0 ); }  
    public Time2 ( int h, int m, int s )
    { setTime ( h, m, s ); }  
```
four constructors are defined for the class `Time2`. However, only one will be invoked, depending on the parameters given.

When you create a class like this:

```java
T1 = new Time2 ( 21, 23 );
```

since two arguments are given in the constructor, the constructor with two parameters will then be executed. In this case, hour is set to 21, minute is set to 23 and second is set to zero. The new object, `T1`, is created with time set to 21:23:00.

```java
public Time2 ( int h, int m )
{ setTime ( h, m, 0 ); }
```

You should notice that the constructor below uses the hour, minute and second values of its argument to initialize the new `Time2` object. When one object of a class has a reference to another object of the same class, the first object can access all the second object’s data and methods.

```java
public void setTime ( int h, int m, int s);
{
    hour = (( h >= 0 && h <= 24 ) ? h : 0 );
    minute = (( m >= 0 && m <= 60 ) ? m : 0 );
    second = (( s >= 0 && s <= 60 ) ? s : 0 );
}
}
```

A discussion of how to overload constructors is provided in the next reading. Fig. 8.5 (on pp. 356–357) gives a complete example of calling different constructors with different initializing parameters. Please have a look at the output of Fig. 8.5 and notice that default value, 0, will be assumed if a particular instance variable is not given.

**Reading**

Deitel and Deitel, section 8.7, pages 355–60.
Using the this reference

When you need to explicitly refer to an object from within the object's methods or you need to pass a reference to the object as an argument in a method call, you can refer to the object explicitly by using the this reference. The this keyword is implicit in the method call or variable reference. For example, inside an object that has data field MyField, the line

\[ \text{MyField} = 1; \]

is the same as the following line:

\[ \text{this.MyField} = 1; \]

Explicitly using the this reference can increase program clarity in some contexts. The example in Fig. 8.4 on pages 352–53 of our textbook demonstrates implicit and explicit use of the this reference.

Another use of this is that you can also pass a reference to the object as an argument. For example:

\[ \text{MyMethod(this)}; \]

where MyMethod is a method which accepts an object of this's type as parameter.

Finalizers

Programmers know about the importance of initialization, but often forget the importance of cleanup. When an object is no longer referenced by any object, Java performs automatic garbage collection of members to help return memory back to the system. Java reclaims the memory space using garbage collection. In fact, Java calls a destructor method, finalize, before garbage collection takes place. Why do we need a destructor method? If, for example, there is some activity that must be performed before you no longer need an object, you must perform that activity yourself. Java has no destructor or similar concept, so you must create an ordinary method to perform this cleanup. For example, suppose in the process of creating your object it draws itself on the screen. If you don’t explicitly erase its image from the screen, it might never get cleaned up. If you put some kind of erasing functionality inside finalize, then if an object is garbage-collected, the image will first be removed from the screen, but if it isn’t, the image will not be erased. Unlike the constructor methods, destructor methods have a specific name: finalize. Here is an example:

\[\text{void finalize() \{}
  \text{body of the finalize method}
\}\]
Finalize methods always have a return type of void. For example:

```java
public class Point {
    private int x;
    private int y;
    protected void finalize() {
        System.out.println("The point is destroyed");
    }
}
```

A program can call `finalize` directly just as it would any other method. However, calling `finalize` will not initiate any type of garbage collection. It is treated as any other method if called directly. When Java does garbage collection, `finalize` is still called even if it has already been called directly by the program.

The next reading describes finalizers.

**Reading**


You’ll notice that this reading doesn’t really state clearly why the finalize method is required if the garbage collection already clears the memory. Clearing the screen is just one of several possible examples that explains why the finalize method is required. You will encounter more in some other large systems later on. Other examples may include closing files and setting flags to denote the end of existence of the object.

**Static class members**

Ordinarily, when you create a class you are describing how objects of that class look and how they will behave. You don’t actually get anything until you create an object of that class with `new`, and at that point data storage is created and methods become available. But there are two situations in which this approach is not sufficient. One situation is if you want to have only one piece of storage for a particular piece of data, regardless of how many objects are created, or even if no objects are created. The other situation is if you need a method that isn’t associated with any particular object of this class. That is, you need a method that you can call even if no objects are created. You can achieve both of these effects with the `static` keyword. When you say something is `static`, it means that data or methods are not tied to any particular object instance of that class. Even if you’ve never created an object of that class you can call a static method or access a piece of static data.
Static variables exist in only one location and are globally accessible by all instances of a class. A variable cannot be changed by a subclass if it has been declared static and final. Furthermore, static variables have the same information in all instances of the class. A static variable is declared in the same way as an instance variable but has the keyword static in front of it. In the following code, the variable count is declared static and is thus the same for all instances of class Employee:

```java
// Declaration of the Employee class.
public class Employee {
    private String firstName;
    private String lastName;
    private static int count;  // # of objects instantiated

    public Employee( String fName, String lName )
    {
        firstName = fName;
        lastName = lName;
        ++count;  // increment static count of employees
        System.out.println( "Employee object constructor: " +
                            firstName + " " + lastName );
    }

    protected void finalize()
    {
        --count;  // decrement static count of employees
        System.out.println( "Employee object finalizer: " +
                            firstName + " " + lastName );
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public static int getCount() { return count; }
}

public class EmployeeTest  {
    public static void main( String args[] ) {
        System.out.println( "Employees before instantiation: " +
                            Employee.getCount() );

        Employee e1 = new Employee( "Susan", "Baker" );
        Employee e2 = new Employee( "Bob", "Jones" );
        System.out.println( "Employees after instantiation: " +
                            e1.getCount());

        e1 = null;
        e2 = null;
        System.gc();  // explicit call to garbage collector
        System.out.println( "Employees after garbage collection: " +
                            Employee.getCount() );
    }
}
```

Program Listing 2.6.
From this code, we can see that all the instances are able to check this variable for the value of count. An external class can query this variable for the same information. Its value needs to be specified only once and stored in one location because the information is the same for all instances of class Employee. Please also notice that as public class is used, the above two classes are stored in two different files.

You can also define a static method, StaticMethod in a class StaticClass with a static variable:

```java
public class StaticClass {
    static int counter = 47;
    static void StaticMethod() { counter ++; }
}
```

Since StaticMethod() is a static method, you can call it directly through its class: StaticClass.StaticMethod();

In the following reading, you will see the difference between public static and private static on page 368. The program in Fig. 8.12 demonstrates the use of a private static class variable and a public static method. Compare the source code with the sample output.

**Reading**

Deitel and Deitel, section 8.11, pages 372–77.
Object-oriented programming

We have mentioned that human beings are a class of object. We know that there are different ways to classify this class. For example, we can classify people according to geographical or racial criteria, such as whether they are Asian or European etc. Asians will have some distinct characteristics compared with Europeans, such as skin colour, eye colour etc. People may also have different behaviours such as different wedding ceremonies and birth ceremonies. However, both Asians and Europeans have some characteristics in common. For example, both have two eyes, two ears etc. The idea that different classes have shared common characteristics is explained by the concepts of superclasses, subclasses and inheritance. We can illustrate these concepts by seeing human beings as the superclass, in which we have different subclasses with varying characteristics inherited from it.

Subclasses have all the characteristics and behaviour inherited from the superclasses but have added some more characteristics and behaviours. For example, we can see that after we have created a class of human being, we could also define another class for Asians and Europeans. It would involve a lot of effort to create a brand new class which might have functionality. However, it would be easier if we could take an existing class, clone it, and then make additions and modifications to the clone. This is effectively what you do with inheritance. In this second part of Unit 2 we will explain and illustrate the process of inheritance.

After we explore the concept of inheritance, we will look at polymorphism. Polymorphism allows improved code organization and readability as well as the creation of extensible programs that can be ‘grown’ not only during the original creation of the project but also when new features are desired. Suppose we define the human class with many methods, but we never create a human in that human class; the human class is just a prototype for us to define Asians and Europeans. To achieve this function, we have two important keywords, abstract and interface. These concepts will also be discussed in this part of the unit.

Inheritance

Inheritance is one of the most important concepts in object-oriented programming, and it enables you to create a class that is similar to an existing class, but one that still has some of its own properties. To create the new class, you only have to specify how that class is different from an existing class, and inheritance gives you automatic access to the existing class.

Each class has a superclass and every class in Java is a subclass of Object. Each class can have one or more subclasses (classes below it in the hierarchy). Classes in the hierarchy inherit from classes above them in the hierarchy. The keyword extends followed by class names indicates the class from which our new class inherits an existing class, e.g. "class Asian extends Human" means Asian is a subclass inheriting
from the superclass, Human. If a class is declared without explicitly stating the superclass, Java implicitly uses class Object.

For example, you have a class defined as follows:

```java
public class Car {
    private boolean running = true;
    private String owner;
    private int speed;
    // and other private data
    public brake() {
        // body of method
    }
    public boolean doorLock () {
        // body of method
    }
    public void startEngine () {
        running = true;
    // other public methods
}
}
```

Program Listing 2.7.

In this base Car class, a car is defined by its private data, like owner, speed, running etc. These data are generic to every type of car. These data fields can be manipulated by the public methods like brake(), doorLock(), startEngine() etc. The data fields and methods are all encapsulated inside the class. Moreover, the data fields are private to the class, meaning that they cannot be directly accessed from outside of the class. Only the class’s public methods can access the data fields. In this way, users cannot update the private data directly but this is possible with the public method. This shows how encapsulation works.

Now, suppose you want to create a safer car. You want to ensure that the door is locked before the engine starts. Class SafeCar inherits Car.

```java
public class SafeCar extends Car {
    private boolean running = true;
    public void startEngine () {
        if (  doorLocked() )
            running = true;
        else
            showError ()
    }
}
```

Program Listing 2.8.

You will find that the new class is much smaller than the superclass as it implicitly inherits all the data fields and methods (except private data and methods) from the superclass, Car. Besides startEngine(), it also has all the methods like brake(), doorLocked() etc. This is an example of inheritance.

Let us now discuss a slightly more complicated example. Suppose you need a program that maintains a database of all the employees in a
company. The company has several different types of employees: regular employees, salespersons, managers, temporary employees, etc., and your program must be able to handle all of them.

You should define a data type called employee that has fields for name, birthday, social security number, and other characteristics.

However, each type of employee requires slightly different information. For example, a regular employee’s salary is based on an hourly wage and the number of hours worked, while a salesperson’s salary also includes a commission on the number of sales made, and a manager’s salary is a fixed amount per week.

You can define a class called Employee that describes the common characteristics of all employees. For example:

```java
public class Employee {
    private String name;
    public Employee() { name = new String(); }
    public Employee(String empName) {
        name = new String( empName );
    }
    public String getName() {
        return name;
    }
    public double computePay() {
        return 0.0;
    }
}

public class testEmployee {
    public static void main( String args[] ) {
        Employee emp = new Employee("Wong Tai Man");

        System.out.println( "The name is " + emp.getName());

        System.out.println( "The pay is " + emp.computePay());
    }
}
```

Program Listing 2.9.

Next, you can define a WageEmployee class that describes a particular type of employee. These employees have the characteristics common to all employees, plus some additional ones. You can make WageEmployee inherit from Employee with the following syntax:

```java
public class WageEmployee extends Employee {
    // inherits from Employee class
    private double wage;
    private double hours;
}
```
```java
public WageEmployee() {
    // implicit call to superclass constructor occurs
    setWage(0.0);
    setHours(0.0);
}

public WageEmployee( String empName,
        double wg, double hrs ) {
    super( empName );
    setWage(wg);
    setHours(hrs);
}

public void setWage(double wg)
{ wage = wg; }
public void setHours(double hrs)
{ hours = hrs; }
public double computePay() {
    return wage * hours;
}
}

public class testWageEmployee {
    public static void main( String args[] ) {
        WageEmployee emp = new WageEmployee(
                "Wong Man Leung",100, 40);

        System.out.println("The name is "+
                emp.getName());

        System.out.println("The pay is "+
                emp.computePay());
    }
}
```

Program Listing 2.10.

`WageEmployee` is a ‘derived class/subclass’, and `Employee` is its ‘base class/superclass’. Each instance of `WageEmployee` contains all of `Employee`’s instance variables, in addition to its own.

The following reading gives you an overview of inheritance, and introduces some new terms such as dynamic binding, polymorphism and protected. Details of these terms will be discussed in the following subsections. The relation between superclass and subclass is well illustrated in Fig. 9.2 and Fig. 9.3. As you read, make sure you understand the is an relationship on page 402.

**Reading**

We have stated previously that there are two access modifiers, namely private and public. A superclass’s public members are accessible anywhere the program has a reference to that superclass type or one of its subclass types. A superclass’s private members are accessible only in methods of that superclass. A method / variable is package access if no access modifier is given. There is also another access method, known as protected members. Protected members may be accessed by all subclasses of the current class, but are not visible to classes outside of the current package. We can therefore say that protected is a fourth kind of access modifier. Together with private, public and package access, it can achieve different levels of accessibility. These different levels of accessibility are summarized in the table below.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>All classes</td>
</tr>
<tr>
<td>private</td>
<td>No other class except the private class itself</td>
</tr>
<tr>
<td>protected</td>
<td>Subclasses included in the same package</td>
</tr>
<tr>
<td>&lt;No modifier given&gt;</td>
<td>Classes in the same package</td>
</tr>
</tbody>
</table>

If you are interested in finding out more about protected members, please refer to the next reading from your text. The activity which follows this reading should consolidate your understanding of the different levels of accessibility.

**Reading**


**Activity 2.2**

Create a class with public, private, protected, and package access data members and method members. Create a subclass that inherits the superclass. Create an object of this subclass and see what kind of compiler messages you get when you try to access all the superclass’s members. Be aware that classes in the same directory are part of the package access.

**Relationships between superclass objects and subclass objects**

An object of a subclass can be treated as an object of its superclass. When an object is created, the corresponding constructor is called. In fact, every superclass constructor is required to call its direct superclass’s constructor as the first task either implicitly or explicitly. If no explicit call of the superclass constructor is specified, the default constructor is called. Consider the following example, which shows three levels of inheritance:
public class LevelOne {
    LevelOne() {
        System.out.println("LevelOne constructor");
    }
}

public class LevelTwo extends LevelOne {
    LevelTwo() {
        System.out.println("LevelTwo constructor");
    }
}

public class LevelThree extends LevelTwo {
    LevelThree() {
        System.out.println("LevelThree constructor");
    }
    public static void main(String[] args) {
        LevelThree x = new LevelThree();
    }
}

Program Listing 2.11.

The output for this program shows the implicit automatic calls:

LevelOne constructor
LevelTwo constructor
LevelThree constructor

You can see that the construction happens from the base ‘outward’, so the base class is initialized before the derived-class constructors can access it.

Now, consider another example shown below:

class OUStudent {
    private String name;
    private String studentID;
    public OUStudent(String n, String id) {
        name = n;
        id = studentID;
    }
    public String getName() { return name; }
    public String getStudentID() { return studentID; }
}

class OUUndergraduate extends OUStudent {
    private String major;
    public OUUndergraduate(String n, String id, String m) {
        name = n;
        studentID = id;
        major = m;
    }
}

Program Listing 2.12.
OUStudent is a class for OUHK students. OУUndergraduate is a class for undergraduate students of OUHK. Since undergraduate students of the OUHK are also OUHK students, OУUndergraduate should be a subclass of OУStudent.

The code in 2.12 would not even compile. It has the following errors:

- although OУUndergraduate does inherit the two attributes, namely name and studentID from OУStudent, it cannot access them because they are all private to OУStudent. Therefore, the first two statements of the constructor of OУUndergraduate are both invalid;

- In the constructor of OУUndergraduate, there is no reference to the constructor of its superclass, therefore, the default one would be called. However, in OУStudent, there is no such default constructor which accepts no parameters.

In order to fix the above problems, we must invoke the constructor of OУStudent explicitly as shown below:

```java
class OУStudent {
    private String name;
    private String studentID;
    public OУStudent(String n, String id) {
        name=n;
        id=studentID;
    }
    public String getName() {return name;}
    public String getStudentID() {return studentID;}
}

class OУUndergraduate extends OУStudent {
    private String major;
    public OУUndergraduate(String n, String id, String m) {
        super(n,id);
        major=m;
    }
}
```

Program listing 2.13

In the constructor of OУUndergraduate, the super keyword is used to refer to the superclass. Thus, super(n, id) is to invoke the constructor of its superclass which accepts two parameters.

Now assume that a general OUHK student can borrow three books from the OUHK library. However, an OUHK undergraduate can borrow six books. We add a method called maxBorrowing which returns the number of books a student can borrow, we have to modify program listing 2.13 to that shown in program listing 2.14:
class OUStudent {
    private String name;
    private String studentID;
    public OUStudent(String n, String id) {
        name=n;
        id=studentID;
    }
    public String getName() {return name;}
    public String getStudentID() {return studentID;}
    public int maxBorrowing() {return 3;}
}

class OUUndergraduate extends OUStudent {
    private String major;
    public OUUndergraduate(String n, String id, String m) {
        super(n,id);
        major=m;
    }
    public int maxBorrowing() {return 6;}
    public String getMajor() {return major;}
}

Program listing 2.14

In the above code, both OUStudent and OUUndergraduate have the method maxBorrowing defined and the two methods have the same signature, i.e., both accept no parameters. We say that the maxBorrowing method of OUUndergraduate overrides the definition of the maxBorrowing method of OUStudent. If the two methods have different signatures, then this is the overloading of the same method name, not overriding.

The following reading give you another example of constructors of subclasses:

Reading


In Java, a variable of a base class can be used to referred to any instance of its subclasses. For example, a variable of type OUStudent can be used to refer to an OUUndergraduate student. Then, if the maxBorrowing method of the variable is called, which version would be involved? The answer is OUUndergraduate although the variable is OUStudent. This is called polymorphism and will be presented in detail in a later section.
Self-test 2.5

1. What is inheritance and how does it help you create new classes quickly?

2. What is a subclass and a superclass?

3. How do you create a subclass?

4. How do you override a method inherited from a superclass?

Activity 2.3

1. Create a class, A with default constructors (empty argument lists) that contain a print statement. Inherit a new class called C from A. Do not create a constructor for C. Create an object of class C and observe the result.

2. Modify Question 1 so that A has a constructor with arguments instead of default constructors. Write a constructor for C with a print statement and perform all initialization within C’s constructor and observe the result.

Constructors and destructors in creating a subclass object

You must agree that a person belonging to the class European must also belong to the class Human. When we create a European, besides initializing the instance variable characteristics to European, we also need to initialize the variables of human, i.e. we need to call the superclass’s constructor.

An explicit call to the superclass’s constructor (via the super reference) can be provided as the first statement in the subclass constructor. Otherwise, the subclass constructor will call the superclass default constructor (or no-argument constructor) implicitly. If the classes in your class hierarchy define finalize methods, the subclass finalize method should (if overriding finalize) invoke the superclass finalize method (as its last action) to ensure that all parts of an object are finalized properly when the garbage collector reclaims the memory for the object.

```java
public class point extends object {
    protected int x, y; // coordinates of the point
    // no-argument constructor
    public point() {
        x = 0;
    }
```
y = 0;
System.out.println ("Point constructor: " + this);
}

// constructor
public point ( int a, int b)
{ x = a;
y = b;
 System.out.println ("Point constructor: " + this);
}

// finalizer
protected void finalize()
{ System.out.println ("Point destructor: " + this);  }

// convert the point into a string representation
public String toString()
{ return "[" + x + ", " + y + "]";
}

public class circle extends Point
{
protected double radius;

// no-argument constructor
public Circle()
{ // implicit call to superclass constructor here
 radius = 0;
 System.out.println ("Circle constructor: " + this);
}

// constructor
public Circle( double r, int a, int b )
{ super ( a, b ); // call the superclass constructor
   // should be the first sentence
   radius = r;
   System.out.println ("Circle constructor: " + this);
}

// finalizer
protected void finalize ()
{ system.out.println ("Circle finalizer : " + this);
super.finalize (); // call the superclass finalize
   // should be the last statement
}

// convert the circle to a string
public String toString()
{ return "Center = " + super.toString() + ";Radius = " + radius;  }
}

public class test
{
public static void main ( string args[] )
{
    Circle circle1, circle2;
The order in which constructor and finalizer are called:

Point constructor: Center = [72, 29]; Radius = 0.0
Circle constructor: Center = [72, 29]; Radius = 4.5
Point constructor: Center = [5, 5]; Radius = 0.0
Circle constructor: Center = [5, 5]; Radius = 10.0
Point finalizer: Center = [72, 29]; Radius = 4.5
Circle finalizer: Center = [72, 29]; Radius = 4.5
Point finalizer: Center = [5, 5]; Radius = 10.0
Circle finalizer: Center = [5, 5]; Radius = 10.0

When circle1 and circle2 are set to null, each of these objects is no longer needed and Java marks the memory occupied for garbage collection. Java guarantees that before the garbage collection runs to reclaim the space of each of these objects, the finalize method for each object will be called. Both Circle and Point finalize methods are called when each Circle object is garbage collected.

The following section of your textbook explains the sequence of calling constructors and finalizers of a superclass when an object in a subclass is being created. Make sure you’re able to see how the sample output of the program listing in Fig. 9.5 is reached.

Reading

Deitel and Deitel, section 9.6, pages 427–432.

Activity 2.4

Create a 3-level inheritance hierarchy. Each class in the hierarchy should have a finalize( ) method and a constructor method with a print statement inside, and it should properly call the base-class version of finalize( ).
Polymorphism

By using polymorphism, it is possible to design and implement systems that are more easily extensible. Programs can be written to process — as superclass objects — objects of all existing classes in a hierarchy. Classes that do not exist during program development can be added with little or no modification to the generic part of the program. The only parts of a program that need modification are those parts that require direct knowledge of the particular class that is added to the hierarchy. For example in the coding segment, after defining the class Employee, we can extend the program by introducing new classes, SalaryEmployee and boss without modifying the coding for Employee:

```java
public class SalaryEmployee extends Employee {
    private double salary;
    public SalaryEmployee() {
        salary = 10000;
    }

    public SalaryEmployee(String name, double sal) {
        super(name);
        salary = sal;
    }

    public double computePay() {
        return salary;
    }
}

public class Boss extends Employee {
    private double weeklySalary;
    public Boss(String first, String last, double s) {
        super(first, second);
        setWeeklySalary(s);
    }

    // set the Boss’s salary
    public void setWeeklySalary(double s) {
        weeklySalary = (s > 0 ? s : 0);
    }

    public double computePay() { return weeklySalary * 4; }
}

public class testPoly {
    public static void main(String args[]) {
        Employee emp1 = new Employee("Wong Man Leung");
        WageEmployee emp2 = new WageEmployee("Wong Tai Man", 200, 40);
        SalaryEmployee emp3 = new SalaryEmployee("Wong Man", 40000);
    }
}
```
Employee emp;

emp = emp1;
System.out.println(emp.getName() + " " +
    emp.computePay());
emp = emp2;
System.out.println(emp.getName() + " " +
    emp.computePay());
emp = emp3;
System.out.println(emp.getName() + " " +
    emp.computePay());
}
}

Program Listing 2.16.

After extending the coding by adding new classes, Boss and SalaryEmployee, to the hierarchy, the generic part of the program is not affected.

With **polymorphism**, you can create new objects that perform the same functions as the base object but which perform one or more of these functions in a different way. For example, you may have a shape object that draws a circle on the screen. By using polymorphism, you can create a shape object that draws a rectangle instead. You do this by creating a new version of the method that draws the shape on the screen. Both the old circle-drawing and the new rectangle-drawing method have the same name, such as drawShape(), but accomplish the drawing in a different way.

One of the means by which we apply traditional procedural programming languages in dealing with different types of objects is the switch statement. For example, assume that at the OUHK, the number of books you can borrow are:

- undergraduate students: 6
- postgraduate students: 12
- all others: 3

In procedural languages, you may have the code shown in program listing 2.17.

```java
int maxBorrowing(Student student){
    switch (student.type) {
        case OUUNDERGRADATE: return 6;
        case OUGRADUATE: return 12;
        default: return 3;
    }
}
```

Program Listing 2.17.

This type of code has the following disadvantages:

- The programmer has the responsibility of making sure that the type field of Student is correct.
• The programmer has the responsibility of making sure that all cases have been listed in the switch statement.

• If there are any changes to the types of students, for example, a new type is added, the programmer has to update the corresponding switch statement to reflect this.

However, in object oriented languages like Java, you can change the behaviour of subclasses by overriding a method. In an earlier example of OUStudent, you saw that we can override the maxBorrowing method in OUUndergraduate to return a value of 6. When the maxBorrowing method of an OUStudent is invoked, the real version to be invoked would be determined by the actual type of OUStudent. If it is actually an OUUndergraduate, then the method of OUUndergraduate would be invoked. Such decisions are made during runtime. Therefore, this is called dynamic binding, i.e., the version of maxBorrowing to be invoked is determined dynamically.

Consider the code in listing 2.18.

```
...
OUStudent aStudent=new OUUndergraduate("Chan Tai Man","9812345","Applied Computing");
System.out.println("The no of books that can be borrowed is "+aStudent.maxBorrowing());
...
```

Program listing 2.18.

Although aStudent is of type OUStudent, it actually refers to an OUUndergraduate. Therefore, when the maxBorrowing method of aStudent is invoked, the run time environment would invoke the OUUndergraduate version of maxBorrowing and return the value of 6. Note that such a decision cannot be made during compilation time because it is difficult, if not impossible, for the compiler to determine that aStudent actually refers to an OUUndergraduate.

The following reading gives you further examples in polymorphism and dynamic method binding.

**Reading**


**Activity 2.5**

Create a base class with two methods. In the first method, call the second method. Inherit a class and override the second method. Create an object of the derived class, upcast it to the base type, and call the first method. Describe what will happen.
**final methods and classes**

The `final` modifier indicates that an object is fixed and cannot be changed. When you use this modifier with a class-level object, it means that the class can never have subclasses. When you apply this modifier to a method, the method can never be overridden. *All methods in a final class are implicitly final.* When you apply this modifier to a variable, the value of the variable remains constant. Here is how you can use the `final` modifier:

```java
public class notChange {
    // final instance variable
    final int constnatValue = 21;

    // final method
    final int nonOverrideMethod(int a) {
    }
}
```

Program Listing 2.19.

Why do you need to prevent a method from being overridden? Imagine, for example, that a class corresponds to a door lock. The class also has a method which corresponds to a function for unlocking the door. The method should ask for a password and return `true` if it is correct. If this method is not final, then any user can simply override this method to just return `true` without the need of a password.

Try to verify a `final` method and class by completing the following activity.

---

**Activity 2.6**

1. Create a class with a final method. Inherit from that class and attempt to override that method.

2. Declare a final class and try to inherit from a final class.

---

**Abstract superclasses and concrete classes**

*Abstract classes* are classes for which the programmer never intends to instantiate any objects. The purpose of an abstract class is to provide an appropriate superclass from which other classes may inherit interface and/or implementation. We do not have to provide the implementation for an abstract method when it is declared.

Classes from which objects can be instantiated are called *concrete classes.*
// Abstract base class Employee

public abstract class Employee {
    private String firstName;
    private String lastName;

    // Constructor
    public Employee(String first, String last) {
        firstName = first;
        lastName = last;
    }

    // Return the first name
    public String getFirstName() { return firstName; }

    // Return the last name
    public String getLastName() { return lastName; }

    public String toString() {
        return firstName + firstName + lastName;
    }

    // Abstract method that must be implemented for each
    // derived class of Employee from which objects
    // are instantiated.
    abstract double earnings();
}

Program Listing 2.20.

The above coding defines an abstract class, Employee. Since no objects of abstract can be instantiated from abstract classes, they are always known as abstract superclasses.

The subclasses of Employee — Boss, CommissionWorker, PieceWorker and HourlyWorker — are defined in the following few pages. Each subclass is defined final as they are not supposed to have any subclass.

The reason for declaring earnings as an abstract method is that we still do not know how to calculate the earnings of this generic employee. With an abstract method, its actual implementation has to be delayed to the derived classes. Therefore, there is no implementation of the method in Employee.

// Boss class derived from Employee

public final class Boss extends Employee {
    private double weeklySalary;

    // Constructor for class Boss
    public Boss(String first, String last, double s) {
        super(first, last);
        // call superclass constructor
    }
}
setWeeklySalary( s );
}

// Set the Boss’s salary
public void setWeeklySalary( double s )
{ weeklySalary = ( s > 0 ? s : 0 ); }

// Get the Boss’s pay
public double earnings() { return weeklySalary; }

// Print the Boss’s name
public String toString()
{
    return "Boss: " + super.toString();
}
}

// CommissionWorker class derived from Employee
public final class CommissionWorker extends Employee {
    private double salary;    // base salary per week
    private double commission;  // amount per item sold
    private int quantity;     // total items sold for week

    // Constructor for class CommissionWorker
    public CommissionWorker( String first, String last,
            double s, double c, int q)
    {
        super( first, last );
        // call superclass constructor
        setSalary( s );
        setCommission( c );
        setQuantity( q );
    }

    // Set CommissionWorker’s weekly base salary
    public void setSalary( double s )
    { salary = ( s > 0 ? s : 0 ); }

    // Set CommissionWorker’s commission
    public void setCommission( double c )
    { commission = ( c > 0 ? c : 0 ); }

    // Set CommissionWorker’s quantity sold
    public void setQuantity( int q )
    { quantity = ( q > 0 ? q : 0 ); }

    // Determine CommissionWorker’s earnings
    public double earnings()
    { return salary + commission * quantity; }

    // Print the CommissionWorker’s name
public String toString()
{
    return "Commission worker: " + super.toString();
}
}

// PieceWorker class derived from Employee

public final class PieceWorker extends Employee {
    private double wagePerPiece; // wage per piece
    output
    private int quantity; // output for week

    // Constructor for class PieceWorker
    public PieceWorker( String first, String last,
                        double w, int q )
    {
        super( first, last );
        // call superclass constructor
        setWage( w );
        setQuantity( q );
    }

    // Set the wage
    public void setWage( double w )
    { wagePerPiece = ( w > 0 ? w : 0 ); }

    // Set the number of items output
    public void setQuantity( int q )
    { quantity = ( q > 0 ? q : 0 ); }

    // Determine the PieceWorker’s earnings
    public double earnings()
    { return quantity * wagePerPiece; }

    public String toString()
    {
        return "Piece worker: " + super.toString();
    }
}

// Definition of class HourlyWorker

public final class HourlyWorker extends Employee {
    private double wage; // wage per hour
    private double hours; // hours worked for week

    // Constructor for class HourlyWorker
    public HourlyWorker( String first, String last,
                          double w, double h )
    {
```java
{  
    super( first, last ); // call superclass?
    constructor
    setWage( w );
    setHours( h );
}

// Set the wage
public void setWage( double w )
{  
    wage = ( w > 0 ? w : 0 );
}

// Set the hours worked
public void setHours( double h )
{  
    hours = ( h >= 0 && h < 168 ? h : 0 );
}

// Get the HourlyWorker’s pay
public double earnings() { return wage * hours; }

public String toString()
{
    return ”Hourly worker: “ + super.toString();
}
}

import javax.swing.JOptionPane;
import java.text.DecimalFormat;

public class Test {
    public static void main( String args[] )
    {
        Employee ref; // superclass reference
        String output = “”;

        Boss b = new Boss( ”John”, ”Smith”, 800.00 );

        PieceWorker p =
            new PieceWorker( ”Bob”, ”Lewis”, 2.5, 200 );

        HourlyWorker h =
            new HourlyWorker( ”Karen”, ”Price”, 13.75, 40 );

        DecimalFormat precision2 = new DecimalFormat( ”#.00” );

        ref = b;
        // superclass reference to subclass object
        output += ref.toString() + ” earned $” +
            precision2.format( ref.earnings() ) +
            ”\n” +
        b.toString() + ” earned $” +
            precision2.format( b.earnings() ) +
            ”\n”;  

        ref = p;  
```
Program Listing 2.21.

This is a long example, but don’t worry; its idea is simple. First, it defines an abstract class, Employee. You should pay attention to the keyword abstract in the class.

Four subclasses, Boss, CommissionWorker, PieceWorker and HourlyWorker are then defined, inherited from Employee. You should notice that the definitions of these four subclasses are slightly different, as they have different declarations of methods and different instance variables. However, you should notice that, even though they have different instance variables, they all have these two private variables as these are inherited from Employee.

private String firstName;
private String lastName;

You may also notice that all the classes, including the superclass and the subclasses have earnings() and toString() methods. This is an example of overriding a superclass’s method when defining subclasses. Which method is referred to when we call earnings()?

Referring to the above coding, in the line ref = b, it assigns to superclass Employee reference ref, a reference to the subclass Boss object to which b refers. The following method calls

ref = b;
   // superclass reference to subclass object
output += ref.toString() + " earned $" +
         precision2.format( ref.earnings() ) +
         "\n" +
p.toString() + " earned $" +
         precision2.format( p.earnings() ) +
         "\n";

ref = h;
   // superclass reference to subclass object
output += ref.toString() + " earned $" +
         precision2.format( ref.earnings() ) +
         "\n" +
h.toString() + " earned $" +
         precision2.format( h.earnings() ) +
         "\n";

JOptionPane.showMessageDialog( null, output,
   "Demonstrating Polymorphism",
   JOptionPane.INFORMATION_MESSAGE );
System.exit(0);
ref.toString() and ref.earnings() invoke the toString method of Boss as Java is always capable of determining the type of the referenced object before invoking a method. This method call is an example of dynamic method binding.

```java
b.toString() + " earned $" +
precision2.format( b.earnings() ) + "\n";
```

The method call b.toString() and b.earnings() explicitly invokes the Boss version of method toString and earnings. These lines show that the method was invoked as expected.

An example of how abstract methods and polymorphism are used to perform payroll calculation based on the type of employee is illustrated in the next reading. In this example, you should be able to also identify inheritance, overloaded constructors, overriding methods and dynamic binding. Please make sure you understand how these concepts apply to the example before you continue.

**Reading**


On pages 449–55 (Fig. 10.6–10.10), it reexamines the Point, Circle and Cylinder hierarchy concept presented on page 448 (Fig 10.4). What is new in this example is that it uses the hierarchy with abstract superclass Shape and includes an additional abstract method, getName in the superclass Shape. When class Point is derived from Shape, it has to implement the abstract method, getName in the superclass, otherwise, class Point would be an abstract class.

**Activity 2.7**

Create a class as abstract without including any abstract methods, and verify that you cannot create any instances of that class.

**Self-test 2.6**

Distinguish between non-abstract and abstract methods.

**Interface**

An interface is very much like a class. However, there is one important difference between an interface and a class. None of the methods
declared in an interface are implemented in the interface itself. Instead, these methods must be implemented in any class that uses the interface. In short, interfaces describe behaviours but do not detail how those behaviours will be carried out.

Interfaces are so much like classes, in fact, that they are declared in almost exactly the same way. You just replace the class keyword with the interface keyword and an interface definition contains a set of public abstract methods. Interface declarations have the syntax:

```java
public interface NameofInterface extends InterfaceList.
```

Making your interface public allows classes and objects outside the given package to implement it, so it is optional. It is also optional for an interface to extend an existing interface. For example:

```java
interface MyInterface
{
}
```

The above coding presents a complete interface, meaning that it can be compiled, after which other Java programs can reference it. An interface is compiled in exactly the same way as a class. The interface’s source code should be saved in a file with the .java extension. Then you should use the Java compiler to compile the source code into a byte-code file which will have the .class extension.

Next, we will discuss how to define methods to an interface.

The following is an interface:

```java
package MyPackages.Display;
public interface MyInterface
{
    public abstract String showText();
}
```

Program Listing 2.22.

The first line specifies that this interface is to be part of the MyPackages.Display package. The second line declares the interface. The difference between a class and an interface is that although the ShowText() method is declared, it doesn’t actually implement the method. That is, you can see from the interface that showText() is supposed to return a String object, but there isn’t a clue as to how that String object is created or what it contains. Those details are left up to any class that decides to implement the interface. The syntax of declaring a method in an interface is similar to that in a method in a class but without method bodies. For example, the line

```java
public abstract String showText();
```

is a complete method in an interface.
If the class leaves one method in the interface undefined, the class becomes an *abstract* class, and must be declared *abstract* in the first line of its class definition and is thus unable to instantiate objects.

After compiling the interface, you can implement it in a class. This means not only telling Java that you’ll be using the interface, but also implementing the interface within the new class. That new class *must* define every method in the interface with the number of arguments and the return type specified in the interface definition. The following coding of a new object, called *InterfaceApplet*, shows how to implement the *MyInterface* interface. Notice how the listing uses the *implements* keyword to tell Java that the applet will be implementing the *MyInterface* interface.

```java
import MyPackages.Display.MyInterface;

public class InterfaceApplet implements MyInterface {
    public String showText() {
        return "Display Text";
    }
}
```

Program Listing 2.23.

The following shows an example of implementing an interface.

```java
// Definition of interface Shape

interface Shape {
    public abstract double area();
    public abstract double volume();
    public abstract String getName();
}

// Definition of class Point

public class Point extends Object implements Shape {
    protected int x, y; // coordinates of the Point

    // no-argument constructor
    public Point() { setPoint(0, 0); }

    // constructor
    public Point(int a, int b) { setPoint(a, b); }

    // Set x and y coordinates of Point
    public void setPoint(int a, int b) {
        x = a;
        y = b;
    }
}
```
// get x coordinate
public int getX() { return x; }

// get y coordinate
public int getY() { return y; }

// convert the point into a String representation
public String toString()
    { return "[" + x + ", " + y + "]"; }

// return the area
public double area() { return 0.0; }

// return the volume
public double volume() { return 0.0; }

// return the class name
public String getName() { return "Point"; }
}

Program Listing 2.24.

The following line,

public class Point extends Object implements Shape

indicates that class Point extends class Object and implements interface Shape. Class Point provides definitions of all three methods in the interface.

// Definition of class Circle
public class Circle extends Point {
  // inherits from Point
  protected double radius;

  // no-argument constructor
  public Circle()
  {
    // implicit call to superclass constructor here
    setRadius( 0 );
  }

  // Constructor
  public Circle( double r, int a, int b )
  {
    super( a, b ); // call the superclass constructor
    setRadius( r );
  }

  // Set radius of Circle
  public void setRadius( double r )
  { radius = ( r >= 0 ? r : 0 ); }
}
// Get radius of Circle
public double getRadius() { return radius; }

// Calculate area of Circle
public double area() {
    return Math.PI * radius * radius;
}

// convert the Circle to a String
public String toString()
    { return "Center = " + super.toString() + "; Radius = " + radius; }

// return the class name
public String getName() { return "Circle"; }
}

// Definition of class Cylinder
public class Cylinder extends Circle {
    protected double height; // height of Cylinder

    // no-argument constructor
    public Cylinder()
    {
        // implicit call to superclass constructor here
        setHeight( 0 );
    }

    // constructor
    public Cylinder( double h, double r, int a, int b )
    {
        super( r, a, b ); // call superclass constructor
        setHeight( h );
    }

    // Set height of Cylinder
    public void setHeight( double h )
    { height = ( h >= 0 ? h : 0 ); }

    // Get height of Cylinder
    public double getHeight() { return height; }

    // Calculate area of Cylinder (i.e., surface area)
    public double area() {
        return 2 * super.area() +
            2 * Math.PI * radius * height;
    }

    // Calculate volume of Cylinder
    public double volume() { return super.area() * height; }

    // Convert a Cylinder to a String
public String toString() {
    return super.toString() + "; Height = " + height;
}

// Return the class name
public String getName() { return "Cylinder"; }
}

Program Listing 2.25.

Any Point object is a Shape. In fact, objects of any class that extends Point are also Shape objects.

// Driver for point, circle, cylinder hierarchy
import javax.swing.JOptionPane;
import java.text.DecimalFormat;

public class Test {

    public static void main ( String args[])
    {
        point = new Point( 7, 11 );
circle = new Circle( 3.5, 22, 8 );
cylinder = new Cylinder( 10, 3.3, 10, 10 );

arrayOfShapes = new Shape[3];

    // aim arrayOfShapes[0] at subclass Point object
    // aim arrayOfShapes[1] at subclass Circle object
    // aim arrayOfShapes[2] at subclass Cylinder object
arrayOfShapes[0] = point;
arrayOfShapes[1] = circle;

String output =
point.getName() + " : " + point.toString() + "\n" +
circle.getName() + " : " + circle.toString() + "\n" +
cylinder.getName() + " : "+cylinder.toString() + "\n";

DecimalFormat precision2 = new DecimalFormat("
#0.00" );

// Loop through arrayOfShapes and print the name, // area, and volume of each object.
for ( int i = 0; i < arrayOfShapes.length; i++ ) {
    output += "\n\n" +
arrayOfShapes[ i ].getName() + " : " +
arrayOfShapes[ i ].toString() +
"\nArea = " +
precision2.format( arrayOfShapes[i].area()) +
"\nVolume = " +
precision2.format( arrayOfShapes[i].volume());
}
JOptionPane.showMessageDialog (null, output,
"Demonstrating Polymorphism",
JOptionPane.INFORMATION_MESSAGE);

System.exit(0);
}
}

Program Listing 2.26.

You may wonder why it is necessary to have interfaces. One benefit of using interfaces is that a class can implement as many interfaces as it needs in addition to extending a class. With multiple interfaces, you can create a new class that inherits the data fields and methods from multiple classes. Even though Java does not support multiple inheritance, you are still able to declare a set of behaviours that can be inherited by one or more classes. To implement multiple interfaces, just list the interfaces after the implements keyword, separating each interface name from the others with a comma. Just like implementing a single interface, it is necessary to implement in the new class all the methods declared in all the interfaces that are being implemented.

Interfaces may contain public final static data. These data will then be considered as constants that can be used in many class definitions. The following reading describes and illustrates a case study of how interfaces can be created and used.

**Reading**

Deitel and Deitel, section 10.8, pages 467–72.

Interface can act in place of abstract. However, with interface it can achieve the purpose of multiple-inheritance, which is not supported in Java.

**Self-test 2.7**

1. How do you add a class or interface to a package?

2. How do you tell Java that the class you’re creating implements a particular interface?

3. What is the biggest difference between an interface and a class?
Summary

In this unit, you have learned a lot about object-oriented programming in Java. Constructors and finalizers help to create and destroy an object. Access modifiers help in manipulating encapsulation. Inheritance allows you to create a new type from existing types.

Inheritance is helpful for rapid project development as you can save lots of effort through code reuse. However, you must have a good class hierarchy design before you start writing your first line of code. In a good hierarchy design, each class:

- should have a specific use;
- should not include too much functionality that is not likely to be reused; and
- can be used without adding functionality.
**Feedback to activities**

**Activity 2.1**

The modified programs are given as follows:

```java
package PackageA;
public class PackageData {
    int x; // package access instance variable
    String s; // package access instance variable
    // constructor
    public PackageData()
    {
        x = 0;
        s = "Hello";
    }
    public String toString()
    {
        return "x: " + x + "  s: " + s;
    }
}
package PackageB;
import javax.swing.JOptionPane;
import PackageA.PackageData;
public class PackageDataTest {
    public static void main( String args[] )
    {
        PackageData d = new PackageData();
        String output;
        output = "After instantiation:
        " + d.toString();
        d.x = 77; // changing package access data
        d.s = "Good bye"; // changing package access data
        output += "\nAfter changing values:
        " + d.toString();
        JOptionPane.showMessageDialog( null, output,
        "Demonstrating Package Access",
        JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    }
}
```

Program Listing 2.27.

The programs cannot be compiled successfully because the instance variables x and s cannot be accessed from outside the package PackageA.
Activity 2.2

Consider the following program:

```java
class superclass {
    public int publicData;
    private int privateData;
    protected int protectedData;
    int packageData;

    public void publicMethod() {
        System.out.println("publicMethod");
    }
    private void privateMethod() {
        System.out.println("privateMethod");
    }
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
    void packageMethod() {
        System.out.println("packageMethod");
    }
}

class subclass extends superclass {}

public class activity82 {
    public static void main(String args[]) {
        subclass c = new subclass();

        c.publicData = 1;
        c.privateData = 2;
        c.protectedData = 3;
        c.packageData = 4;
        c.publicMethod();
        c.privateMethod();
        c.protectedMethod();
        c.packageMethod();
    }
}
```

Program Listing 2.28.

The program cannot be compiled successfully because

- `c.privateData` and `c.privateMethod()` are private in the class `superclass` and they cannot be accessed outside the class `superclass`. 
Activity 2.3

1 Consider the following program:

```java
class A {
    public A() {
        System.out.println("Object of class A is created");
    }
}

class C extends A {
}

class test {
    public static void main(String args[]) {
        C refC = new C();
        System.out.println("In Test");
    }
}
```

Program Listing 2.29.

The program will display the following output when it is executed:

Object of class A is created
In Test

2 Consider the following program:

```java
class A {
    protected int x;
    public A(int i) {
        System.out.println("Object of class A is created");
    }
}

class C extends A {
    public C(int i) {
        x = i;
        System.out.println("Object of class C is created");
    }
}

class test {
    public static void main(String args[]) {
        C refC = new C(10);
        System.out.println("The value is "+ refC.x);
    }
}
```

Program Listing 2.30.

This program cannot be compiled successfully because there is no default constructor in class A.
Activity 2.4

```java
class A {
    public A() {
        System.out.println("Constructor of Class A");
    }
    protected void finalize() {
        System.out.println("Finalizer of Class A");
    }
}

class B extends A {
    public B() {
        System.out.println("Constructor of Class B");
    }
    protected void finalize() {
        System.out.println("Finalizer of Class B");
    }
}

class C extends B {
    public C() {
        System.out.println("Constructor of Class C");
    }
    protected void finalize() {
        System.out.println("Finalizer of Class C");
    }
}

class test {
    public static void main(String args[]) {
        C ref = new C();
        ref = null;
        System.gc();
    }
}
```

Program Listing 2.31.
Activity 2.5

Consider the following program:

```java
class A {
    protected void method1() {
        System.out.println("In method1 of class A");
        method2();
    }
    protected void method2() {
        System.out.println("In method2 of class A");
    }
}

class B extends A {
    protected void method2() {
        System.out.println("In method2 of class B");
    }
}

class test {
    public static void main(String args[]) {
        B refB = new B();
        A refA;

        refA = refB;
        refA.method1();
    }
}
```

Program Listing 2.32.

The program will display the following output when it is executed:

In method1 of class A
In method2 of class B
Activity 2.6

1. Consider the following program:

```java
class Employee {
    public double salary;
    protected final void setSalary(double x) {
        salary = x;
    }
}

class Manager extends Employee {
    protected void setSalary(double x) {
        salary = 1.5 * x;
    }
}

class test {
    public static void main(String args[]) {
        Manager m = new Manager();
        m.setSalary(20000.0);
    }
}
```

Program Listing 2.33.

This program cannot be compiled successfully because the final method setSalary in class Employee cannot be overridden by method setSalary in class Manager.

2. Consider the following program:

```java
final class Employee {
    public double salary;
    protected void setSalary(double x) {
        salary = x;
    }
}

class Manager extends Employee {
}

class test {
    public static void main(String args[]) {
        Manager m = new Manager();
        m.setSalary(20000.0);
    }
}
```

Program Listing 2.34.

This program cannot be compiled successfully because it is illegal to inherit from final class Employee.
Activity 2.7

Consider the following program:

```java
abstract class Employee {
    public double salary;
    protected void setSalary(double x) {
        salary = x;
    }
}

class test {
    public static void main(String args[]) {
        Employee m = new Employee();
        m.setSalary(20000.0);
    }
}
```

Program Listing 2.35.

The program cannot be compiled successfully because it is illegal to instantiate an object from abstract class Employee.
Solutions to self-tests

Self-test 2.1

1. The two main elements of a class are data fields and the methods that operate on the data fields.

2. A class is like a template or blueprint for an object. An object is an instance of the class.

Self-test 2.2

1. public class MyClass {

2.   }

3. public class MyClass {
   int myField;

4. public MyClass (int value ) {
   myField = value;
}

5. public void setData ( int value ) {
   myField = value ;
}

6. public int getData () {
   return myField;
}

Program Listing 2.36.

Self-test 2.3

1. To create an object of a class, you use the new operator followed by a call to the class’s constructor.

2. To use a class that’s defined in a different file, you place the import keyword, followed by the class’s name, at the top of the file that needs access to the class. You must also be sure that the compiler can find the class, usually by placing the class files all in the same directory.

Self-test 2.4

1. A package is a group of related classes and interfaces.

2. To tell Java that a class uses a particular package, you use the import keyword followed by the full name of the package.
**Self-test 2.5**

1. Using inheritance, a new class (subclass) derived from a base class (superclass) inherits the data fields and methods defined in the base class. The programmer then only needs to add whatever additional functionality is required by the new class.

2. A subclass is a class that’s been derived from another class using the `extends` keyword. A superclass is the class from which a subclass is derived. That is, the terms ‘subclass’ and ‘base class’ are equivalent.

3. You create a subclass by using the `extends` keyword like this:

```java
class SubClass extends SuperClass
{
}
```

4. To override a method, you provide, in your subclass, a method with exactly the same name, return type, and arguments as the method of the superclass you want to override. Then, Java will call your class’s version of the method rather than the superclass’s version.

**Self-test 2.6**

An abstract method is one that does not have implementation while a non-abstract method is one with implementation.

**Self-test 2.7**

1. To tell Java that a class implements a particular interface, add the `implements` keyword to the class’s declaration line, followed by the name of the interface.

2. The biggest difference between an interface and a class is that an interface declares, but never implements, its methods. An interface’s methods must be defined in any class that implements the interface.

3. Interfaces and classes are similar in that they are declared in almost exactly the same way.