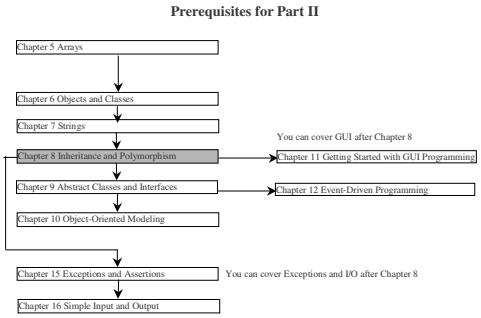
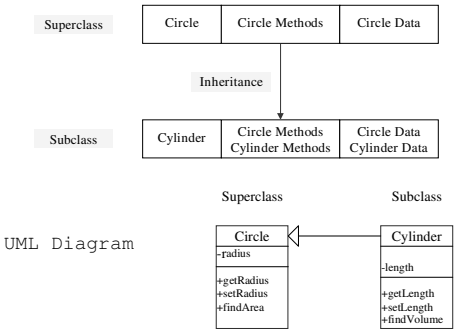
Chapter 8 Inheritance and Polymorphism



Objectives

- To develop a subclass from a superclass through inheritance (§8.2).
- To invoke the superclass's constructors and methods using the super keyword (§8.3).
- To override methods in the subclass (§8.4).
- To explore the useful methods (equals(Object), hashCode(), toString(), finalize(), clone(), and getClass()) in the Object class (§8.5, §8.11 Optional).
- To comprehend polymorphism, dynamic binding, and generic programming (§8.6).
- To describe casting and explain why explicit downcasting is necessary (§8.7).
- To understand the effect of hiding data fields and static methods (§8.8 Optional).
- To restrict access to data and methods using the protected visibility modifier (§8.9).
- To declare constants, unmodifiable methods, and nonextendable class using the final modifier (§8.10).
- To initialize data using initialization blocks and distinguish between instance initialization and static initialization blocks (§8.12 Optioanl).

Superclasses and Subclasses



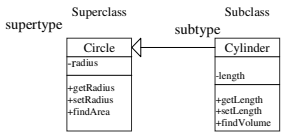
UML Diagram


```
// Cylinder.java: Class definition for describing Cylinder
public class Cylinder extends Circle {
  private double length = 1;

  /** Return length */
  public double getLength() {
    return length;
  }

  /** Set length */
  public void setLength(double length) {
    this.length = length;
  }

  /** Return the volume of this cylinder */
  public double findVolume() {
    return findArea() * length;
  }
}
```




```
Cylinder cylinder = new Cylinder();
System.out.println("The length is " +
  cylinder.getLength());
System.out.println("The radius is " +
  cylinder.getRadius());
System.out.println("The volume of the cylinder is " +
  cylinder.findVolume());
System.out.println("The area of the circle is " +
  cylinder.findArea());
```

The output is

```
The length is 1.0
The radius is 1.0
The volume of the cylinder is 3.14159
The area of the circle is 3.14159
```


Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

NOTE

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public Cylinder() {
}
```

is equivalent to

```
public Cylinder() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```


Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called *constructor chaining*.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("Invoke Employee's overloaded constructor");
    System.out.println("Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Person's no-arg constructor is invoked");
  }
}
```


Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```


Declaring a Subclass

A subclass extends properties and methods from the superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
// Cylinder.java: New cylinder class that overrides the findArea()
// method defined in the circle class.
public class Cylinder extends Circle {

 /** Return the surface area of this cylinder. The formula is
  * 2 * circle area + cylinder body area
  */
  public double findArea() {
    return 2 * super.findArea() + 2 * getRadius() * Math.PI * length;
  }

  // Other methods are omitted
}
```


NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

NOTE

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

The `Object` Class

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```


The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Cylinder myCylinder = new Cylinder(5.0, 2.0);

System.out.println(myCylinder.toString());
```

The code displays something like Cylinder@15037e5. This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

Polymorphism, Dynamic Binding and Generic Programming

```
public class Test {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

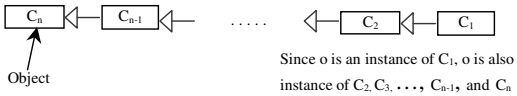
Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . . , C_{n-1}, and C_n

Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime. See Review Questions 8.7 and 8.8.

Generic Programming

```
public class Test {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compilation error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```


Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Cylinder myCylinder = (Cylinder)myCircle;

Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Circle myCircle = new Circle();

if (myCircle instanceof Cylinder) {
  Cylinder myCylinder = (Cylinder)myCircle;
  ...
}
```


TIP

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

Example 8.1 *Demonstrating Polymorphism and Casting*

This example creates two geometric objects: a circle, and a cylinder, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and perimeter if the object is a circle, and displays area and volume if the object is a cylinder.

TestPolymorphismCasting Run

Hiding Fields and Static Methods (Optional)

You can override an instance method, but you cannot override a field (instance or static) or a static method. If you declare a field or a static method in a subclass with the same name as one in the superclass, the one in the superclass is hidden, but it still exists. The two fields or static methods are independent. You can reference the hidden field or static method using the super keyword in the subclass. The hidden field or method can also be accessed via a reference variable of the superclass's type.

Hiding Fields and Static Methods, cont.

When invoking an instance method from a reference variable, the actual class of the object referenced by the variable decides which implementation of the method is used at runtime. When accessing a field or a static method, the declared type of the reference variable decides which method is used at compilation time.

See the example in the book.

The `protected` Modifier

- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public

Visibility increases →

`private, none (if no modifier is used), protected, public`

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers

A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

The equals() and hashCode() Methods in the Object Class

- The equals() method compares the contents of two objects.
- The hashCode() method returns the hash code of the object. Hash code is an integer, which can be used to store the object in a hash set so that it can be located quickly.

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```


NOTE

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

The hashCode() method

Invoking hashCode() on an object returns the hash code of the object. Hash code is an integer, which can be used to store the object in a hash set so that it can be located quickly. Hash sets will be introduced in Chapter 18, "Java Collections Framework." The hashCode implemented in the Object class returns the internal memory address of the object in hexadecimal. Your class should override the hashCode method whenever the equals method is overridden. By contract, if two objects are equal, their hash codes must be same.

The finalize, clone, and getClass Methods

- The finalize method is invoked by the garbage collector on an object when the object becomes garbage.
- The clone() method copies an object.
- The getClass() method returns an instance of the java.lang.Class class, which contains the information about the class for the object. Before an object is created, its defining class is loaded and the JVM automatically creates an instance of java.lang.Class for the class. From this instance, you can discover the information about the class at runtime.

Initialization Block

Initialization blocks can be used to initialize objects along with the constructors. An initialization block is a block of statements enclosed inside a pair of braces. An initialization block appears within the class declaration, but not inside methods or constructors. It is executed as if it were placed at the beginning of every constructor in the class.

```
public class Book {
  private static int numOfObjects;
  private String title
  private int id;

  public Book(String title) {
    this.title = title;
  }

  public Book(int id) {
    this.id = id;
  }

  {
    numOfObjects++;
  }
}
```

Equivalent

```
public class Book {
  private static int numOfObjects;
  private String title;
  private int id;

  public Book(String title) {
    numOfObjects++;
    this.title = title;
  }

  public Book(int id) {
    numOfObjects++;
    this.id = id;
  }
}
```


Initialization Block

```
public class Book {

  {
    numOfObjects++;
  }
}
```

Static Initialization Block

A static initialization block is much like a nonstatic initialization block except that it is declared static, can only refer to static members of the class, and is invoked when the class is loaded. The JVM loads a class when it is needed. A superclass is loaded before its subclasses.

Static Initialization Block

```
class A extends B {
static {
    System.out.println("A's static initialization block " +
      "is invoked");
  }
}

class B {
static {
    System.out.println("B's static initialization block " +
      "is invoked");
  }
}
```