Chapter 3

Learning Object-Oriented Language Features

An *object-based language* encapsulates attributes and behaviors in objects. To be known as an *object-oriented language*, the language must also support inheritance and polymorphism. This chapter introduces you to Java's language features that support these twin pillars of object orientation. Furthermore, the chapter introduces you to interfaces, Java's ultimate abstract type mechanism.

Inheritance

Inheritance is a hierarchical relationship between entity categories in which one category inherits attributes and behaviors from at least one other category. For example, tiger inherits from animal (tiger is a kind of animal), car inherits from vehicle (car is a kind of vehicle), and checking account inherits from bank account (checking account is a kind of bank account). Animal, vehicle, and bank account are more generic categories; and tiger, car, and checking account are more specific categories.

Java supports *implementation inheritance* (class extension) by providing language features for declaring and initializing classes that are extensions of existing classes. After showing you how to use these features, this section introduces you to a special class that sits at the top of Java's class hierarchy. The section then introduces you to composition, an alternative to implementation inheritance for reusing code. Lastly, I will show you how composition can overcome problems with implementation inheritance.

> **NOTE:** Java also supports another kind of inheritance called interface inheritance. Later in this chapter, while discussing Java's interfaces language feature, I discuss interface inheritance.

Extending Classes

Java provides the reserved word extends for specifying a hierarchical relationship between two classes. For example, suppose you have a Vehicle class and want to introduce a Car class as a kind of Vehicle. Listing 3–1 uses extends to cement this relationship.

Listing 3–1. *Relating two classes via extends*

```
class Vehicle
{
   // member declarations
}
class Car extends Vehicle
{
   // member declarations
}
```

Listing 3–1 codifies a relationship that is known as an "is-a" relationship: a car is a kind of vehicle. In this relationship, Vehicle is known as the *base class*, *parent class*, or *superclass*; and Car is known as the *derived class*, *child class*, or *subclass*.

> **CAUTION:** You cannot extend a final class. For example, if you declared Vehicle as final class Vehicle, the compiler would report an error upon encountering class Car extends Vehicle. Developers declare their classes final when they do not want these classes to be subclassed (for security or other reasons).

In addition to being capable of providing its own member declarations, Car is capable of inheriting member declarations from its Vehicle superclass. As Listing 3–2 shows, inherited members become accessible to members of the Car class.

Listing 3–2. *Inheriting members*

```
class Vehicle
{
   private String make;
   private String model;
   private int year;
   Vehicle(String make, String model, int year)
   {
      this.make = make;
      this.model = model;
      this.year = year;
   }
   String getMake()
   {
      return make;
   }
   String getModel()
   {
      return model;
   }
   int getYear()
```

```
   {
      return year;
   }
}
class Car extends Vehicle
{
   private int numWheels;
   Car(String make, String model, int year, int numWheels)
   {
      super(make, model, year);
      this.numWheels = numWheels;
   }
   public static void main(String[] args)
   {
      Car car = new Car("Ford", "Fiesta", 2009, 4);
      System.out.println("Make = " + car.getMake());
      System.out.println("Model = " + car.getModel ());
      System.out.println("Year = " + car.getYear ());
      // Normally, you cannot access a private field via an object
      // reference. However, numWheels is being accessed from
      // within a method (main()) that is part of the Car class.
      System.out.println("Number of wheels = "+car.numWheels);
   }
}
```

Listing 3–2's `Vehicle` class declares `private` fields that store a vehicle's make, model, and year; a constructor that initializes these fields to passed arguments; and getter methods that retrieve these fields' values.

The `Car` subclass provides a private `numWheels` field, a constructor that initializes a `Car` object's `Vehicle` and `Car` layers, and a `main()` class method for test-driving this application.

`Car`'s constructor uses reserved word `super` to call `Vehicle`'s constructor with `Vehicle`-oriented arguments, and then initializes `Car`'s `numWheels` instance field. The `super()` call is analogous to specifying `this()` to call another constructor in the same class.

> **CAUTION:** The `super()` call can only appear in a constructor. Furthermore, it must be the first code that is specified in the constructor.
>
> If `super()` is not specified, and if the superclass does not have a noargument constructor, the compiler will report an error because the subclass constructor must call a noargument superclass constructor when `super()` is not present.

`Car`'s `main()` method creates a `Car` object, initializing this object to a specific make, model, year, and number of wheels. Four `System.out.println()` method calls subsequently output this information.

The first three `System.out.println()` method calls retrieve their pieces of information by calling the `Car` instance's inherited `getMake()`, `getModel()`, and `getYear()` methods. The final `System.out.println()` method call accesses the instance's `numWheels` field directly.

NOTE: A class whose instances cannot be modified is known as an *immutable class*. Vehicle is an example. If Car's main() method, which can directly read or write numWheels, was not present, Car would also be an example of an immutable class.

A class cannot inherit constructors, nor can it inherit private fields and methods. Car does not inherit Vehicle's constructor, nor does it inherit Vehicle's private make, model, and year fields.

A subclass can *override* (replace) an inherited method so that the subclass's version of the method is called instead. Listing 3–3 shows you that the overriding method must specify the same name, parameter list, and return type as the method being overridden.

Listing 3–3. *Overriding a method*

```
class Vehicle
{
   private String make;
   private String model;
   private int year;
   Vehicle(String make, String model, int year)
   {
      this.make = make;
      this.model = model;
      this.year = year;
   }
   void describe()
   {
      System.out.println(year + " " + make + " " + model);
   }
}
class Car extends Vehicle
{
   private int numWheels;
   Car(String make, String model, int year, int numWheels)
   {
      super(make, model, year);
   }
   void describe()
   {
      System.out.print("This car is a "); // Print without newline – see Chapter 1.
      super.describe();
   }
   public static void main(String[] args)
   {
      Car car = new Car("Ford", "Fiesta", 2009, 4);
      car.describe();
   }
}
```

Listing 3–3's Car class declares a describe() method that overrides Vehicle's describe() method to output a car-oriented description. This method uses reserved word super to call Vehicle's describe() method via super.describe();.

NOTE: You call a superclass method from the overriding subclass method by prefixing the method's name with reserved word `super` and the member access operator. If you do not do this, you end up recursively calling the subclass's overriding method.

You can also use `super` and the member access operator to access non-`private` superclass fields from subclasses that replace these fields by declaring same-named fields.

If you were to compile and run Listing 3–3, you would discover that `Car`'s overriding `describe()` method executes instead of `Vehicle`'s overridden `describe()` method, and outputs `This car is a 2009 Ford Fiesta`.

CAUTION: You cannot override a `final` method. For example, if `Vehicle`'s `describe()` method was declared as `final void describe()`, the compiler would report an error upon encountering an attempt to override this method in the `Car` class. Developers declare their methods `final` when they do not want these methods to be overridden (for security or other reasons).

Also, you cannot make an overriding method less accessible than the method it overrides. For example, if `Car`'s `describe()` method was declared as `private void describe()`, the compiler would report an error because private access is less accessible than the default package access. However, `describe()` could be made more accessible by declaring it `public`, as in `public void describe()`.

Suppose you happened to replace Listing 3–3's `describe()` method with the method shown in Listing 3–4.

Listing 3–4. *Incorrectly overriding a method*

```
void describe(String owner)
{
   System.out.print("This car, which is owned by " + owner + ", is a ");
   super.describe();
}
```

The modified `Car` class now has two `describe()` methods, the explicitly declared method in Listing 3–4 and the method inherited from `Vehicle`. Listing 3–4 does not override `Vehicle`'s `describe()` method. Instead, it overloads this method.

The Java compiler helps you detect an attempt to overload instead of override a method at compile time by letting you prefix a subclass's method header with the `@Override` annotation, as shown in Listing 3–5. (I will discuss annotations in Chapter 5.)

Listing 3–5. *Annotating an overriding method*

```
@Override void describe()
{
   System.out.print("This car is a ");
   super.describe();
}
```

Specifying @Override tells the compiler that the method overrides another method. If you overload the method instead, the compiler reports an error. Without this annotation, the compiler would not report an error because method overloading is a valid feature.

TIP: Get into the habit of prefixing overriding methods with the @Override annotation. This habit will help you detect overloading mistakes much sooner.

Chapter 2 discussed the initialization order of classes and objects, where you learned that class members are always initialized first, and in a top-down order (the same order applies to instance members). Implementation inheritance adds a couple more details:

- A superclass's class initializers always execute before a subclass's class initializers.
- A subclass's constructor always calls the superclass constructor to initialize an object's superclass layer, and then initializes the subclass layer.

Java lets you extend a single class, which is commonly referred to as *single inheritance*. However, Java does not permit you to extend multiple classes, which is known as *multiple implementation inheritance*, because it leads to ambiguities.

For example, suppose Java supported multiple implementation inheritance, and you decided to model a *tiglon* (a cross between a tiger and a lioness) via the class structure shown in Listing 3–6.

Listing 3–6. *Modeling a tiglon*

```
class Tiger
{
   void describe()
   {
      // Code that outputs a description of the tiger's appearance and behaviors.
   }
}
class Lioness
{
   void describe()
   {
      // Code that outputs a description of the lioness's appearance and behaviors.
   }
}
class Tiglon extends Tiger, Lioness
{
   // Which describe() method does Tiglon inherit?
}
```

Listing 3–6 shows an ambiguity resulting from each of Tiger and Lioness possessing a describe() method. Which of these methods does Tiglon inherit? A related ambiguity arises from same-named fields, possibly of different types. Which field is inherited?

The Ultimate Superclass

A class that does not explicitly extend another class implicitly extends Java's `Object` class (located in the `java.lang` package—I will discuss packages in the next chapter). For example, Listing 3–1's `Vehicle` class extends `Object`, whereas `Car` extends `Vehicle`.

`Object` is Java's ultimate superclass because it serves as the ancestor of every other class, but does not itself extend any other class. `Object` provides a common set of methods that other classes inherit. Table 3–1 describes these methods.

Table 3–1. *`Object`'s Methods*

Method	Description
`Object clone()`	Create and return a copy of the current object.
`boolean equals(Object obj)`	Determine if the current object is equal to the object identified by `obj`.
`void finalize()`	Finalize the current object.
`Class<?> getClass()`	Return the current object's `Class` object.
`int hashCode()`	Return the current object's hash code.
`void notify()`	Wake up one of the threads that are waiting on the current object's monitor.
`void notifyAll()`	Wake up all threads that are waiting on the current object's monitor.
`String toString()`	Return a string representation of the current object.
`void wait()`	Cause the current thread to wait on the current object's monitor until it is woken up via `notify()` or `notifyAll()`.
`void wait(long timeout)`	Cause the current thread to wait on the current object's monitor until it is woken up via `notify()` or `notifyAll()`, or until the specified `timeout` value (in milliseconds) has elapsed, whichever comes first.
`void wait (long timeout, int nanos)`	Cause the current thread to wait on the current object's monitor until it is woken up via `notify()` or `notifyAll()`, or until the specified `timeout` value (in milliseconds) plus `nanos` value (in nanoseconds) has elapsed, whichever comes first.

I will discuss `getClass()`, `notify()`, `notifyAll()`, and the `wait()` methods in Chapter 7.

Cloning

The clone() method *clones* (duplicates) an object without calling a constructor. It copies each primitive or reference field's value to its counterpart in the clone, a task known as *shallow copying* or *shallow cloning*. Listing 3–7 demonstrates this behavior.

Listing 3–7. ***Shallowly cloning an*** `Employee` ***object***

```
class Employee implements Cloneable
{
   String name;
   int age;
   Employee(String name, int age)
   {
      this.name = name;
      this.age = age;
   }
   public static void main(String[] args) throws CloneNotSupportedException
   {
      Employee e1 = new Employee("John Doe", 46);
      Employee e2 = (Employee) e1.clone();
      System.out.println(e1 == e2); // Output: false
      System.out.println(e1.name == e2.name); // Output: true
   }
}
```

Listing 3–7 declares an Employee class with name and age instance fields, and a constructor for initializing these fields. The main() method uses this constructor to initialize a new Employee object's copies of these fields to John Doe and 46.

> **NOTE:** A class must implement the Cloneable interface or its instances cannot be shallowly cloned via Object's clone() method—this method performs a runtime check to see if the class implements Cloneable. (I will discuss interfaces later in this chapter.) If a class does not implement Cloneable, clone() throws CloneNotSupportedException. (Because CloneNotSupportedException is a checked exception, it is necessary for Listing 3–7 to satisfy the compiler by appending throws CloneNotSupportedException to the main() method's header. I will discuss exceptions in the next chapter.) String is an example of a class that does not implement Cloneable; hence, String objects cannot be shallowly cloned.

After assigning the Employee object's reference to local variable e1, main() calls the clone() method on this variable to duplicate the object, and then assigns the resulting reference to variable e2. The (Employee) cast is needed because clone() returns Object.

To prove that the objects whose references were assigned to e1 and e2 are different, main() next compares these references via == and outputs the Boolean result, which happens to be false.

To prove that the Employee object was shallowly cloned, main() next compares the references in both Employee objects' name fields via == and outputs the Boolean result, which happens to be true.

NOTE: `Object`'s `clone()` method was originally specified as a `public` method, which meant that any object could be cloned from anywhere. For security reasons, this access was later changed to `protected`, which means that only code within the same package as the class whose `clone()` method is to be called, or code within a subclass of this class (regardless of package), can call `clone()`.

Shallow cloning is not always desirable because the original object and its clone refer to the same object via their equivalent reference fields. For example, each of Listing 3–7's two `Employee` objects refers to the same `String` object via its `name` field.

Although not a problem for `String`, whose instances are immutable, changing a mutable object via the clone's reference field results in the original (noncloned) object seeing the same change via its equivalent reference field.

For example, suppose you add a reference field named `hireDate` to `Employee`. This field is of type `Date` with `year`, `month`, and `day` fields. Because `Date` is mutable, you can change the contents of these fields in the `Date` instance assigned to `hireDate`.

Now suppose you plan to change the clone's date, but want to preserve the original `Employee` object's date. You cannot do this with shallow cloning because the change is also visible to the original `Employee` object.

To solve this problem, you must modify the cloning operation so that it assigns a new `Date` reference to the `Employee` clone's `hireDate` field. This task, which is known as *deep copying* or *deep cloning*, is demonstrated in Listing 3–8.

Listing 3–8. *Deeply cloning an `Employee` object*

```
class Date
{
   int year, month, day;
   Date(int year, int month, int day)
   {
      this.year = year;
      this.month = month;
      this.day = day;
   }
}
class Employee implements Cloneable
{
   String name;
   int age;
   Date hireDate;
   Employee(String name, int age, Date hireDate)
   {
      this.name = name;
      this.age = age;
      this.hireDate = hireDate;
   }
   @Override protected Object clone() throws CloneNotSupportedException
   {
      Employee emp = (Employee) super.clone();
```

```
      if (hireDate != null) // no point cloning a null object (one that does not exist)
         emp.hireDate = new Date(hireDate.year, hireDate.month, hireDate.day);
      return emp;
   }
   public static void main(String[] args) throws CloneNotSupportedException
   {
      Employee e1 = new Employee("John Doe", 46, new Date(2000, 1, 20));
      Employee e2 = (Employee) e1.clone();
      System.out.println(e1 == e2); // Output: false
      System.out.println(e1.name == e2.name); // Output: true
      System.out.println(e1.hireDate == e2.hireDate); // Output: false
      System.out.println(e2.hireDate.year + " " + e2.hireDate.month + " " +
                         e2.hireDate.day); // Output: 2000 1 20
   }
}
```

Listing 3–8 declares `Date` and `Employee` classes. The `Date` class declares `year`, `month`, and `day` fields and a constructor. (You can declare a comma-separated list of variables on one line provided that these variables all share the same type, which is `int` in this case.)

The `Employee` class overrides the `clone()` method to deeply clone the `hireDate` field. This method first calls the `Object` superclass's `clone()` method to shallowly clone the current `Employee` instance's fields, and then stores the new instance's reference in `emp`.

The `clone()` method next assigns a new `Date` instance to `emp`'s `hireDate` field, where this instance's fields are initialized to the same values as those in the original `Employee` object's `hireDate` instance.

At this point, you have an `Employee` clone with shallowly cloned `name` and `age` fields, and a deeply cloned `hireDate` field. The `clone()` method finishes by returning this `Employee` clone.

> **NOTE:** If you are not calling `Object`'s `clone()` method from an overridden `clone()` method (because you prefer to deeply clone reference fields and do your own shallow copying of non-reference fields), it is not necessary for the class containing the overridden `clone()` method to implement `Cloneable`, but it should implement this interface for consistency. `String` does not override `clone()`, so `String` objects cannot be deeply cloned.

Equality

The `==` and `!=` operators compare two primitive values (such as integers) for equality (`==`) or inequality (`!=`). These operators also compare two references to see if they refer to the same object or not. This latter comparison is known as an *identity check*.

You cannot use `==` and `!=` to determine if two objects are logically the same (or not). For example, two `Car` objects with the same field values are logically equivalent. However, `==` reports them as unequal because of their different references.

> **NOTE:** Because == and != perform the fastest possible comparisons, and because string comparisons need to be performed quickly (especially when sorting a huge number of strings), the `String` class contains special support that allows literal strings and string-valued constant expressions to be compared via == and !=. (I will discuss this support when I present `String` in Chapter 7.) The following statements demonstrate these comparisons:
>
> ```
> System.out.println("abc" == "abc"); // Output: true
> System.out.println("abc" == "a" + "bc"); // Output: true
> System.out.println("abc" == "Abc"); // Output: false
> System.out.println("abc" != "def"); // Output: true
> System.out.println("abc" == new String("abc")); // Output: false
> ```

Recognizing the need to support logical equality in addition to reference equality, Java provides an `equals()` method in the `Object` class. Because this method defaults to comparing references, you need to override `equals()` to compare object contents.

Before overriding `equals()`, make sure that this is necessary. For example, Java's `StringBuffer` class does not override `equals()`. Perhaps this class's designers did not think it necessary to determine if two `StringBuffer` objects are logically equivalent.

You cannot override `equals()` with arbitrary code. Doing so will probably prove disastrous to your applications. Instead, you need to adhere to the contract that is specified in the Java documentation for this method, and which I present next.

The `equals()` method implements an equivalence relation on nonnull object references:

- *It is reflexive*: For any nonnull reference value *x*, *x*`.equals(`*x*`)` returns true.
- *It is symmetric*: For any nonnull reference values *x* and *y*, *x*`.equals(`*y*`)` returns true if and only if *y*`.equals(`*x*`)` returns true.
- *It is transitive*: For any nonnull reference values *x*, *y*, and *z*, if *x*`.equals(`*y*`)` returns true and *y*`.equals(`*z*`)` returns true, then *x*`.equals(`*z*`)` returns true.
- *It is consistent*: For any nonnull reference values *x* and *y*, multiple invocations of *x*`.equals(`*y*`)` consistently return true or consistently return false, provided no information used in `equals()` comparisons on the objects is modified.
- For any nonnull reference value *x*, *x*`.equals(null)` returns false.

Although this contract probably looks somewhat intimidating, it is not that difficult to satisfy. For proof, take a look at the implementation of the `equals()` method in Listing 3–9's `Point` class.

Listing 3–9. *Logically comparing `Point` objects*

```
class Point
{
   private int x, y;
   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
   @Override public boolean equals(Object o)
   {
      if (!(o instanceof Point))
         return false;
      Point p = (Point) o;
      return p.x == x && p.y == y;
   }
   public static void main(String[] args)
   {
      Point p1 = new Point(10, 20);
      Point p2 = new Point(20, 30);
      Point p3 = new Point(10, 20);
      // Test reflexivity
      System.out.println(p1.equals(p1)); // Output: true
      // Test symmetry
      System.out.println(p1.equals(p2)); // Output: false
      System.out.println(p2.equals(p1)); // Output: false
      // Test transitivity
      System.out.println(p2.equals(p3)); // Output: false
      System.out.println(p1.equals(p3)); // Output: true
      // Test nullability
      System.out.println(p1.equals(null)); // Output: false
      // Extra test to further prove the instanceof operator's usefulness.
      System.out.println(p1.equals("abc")); // Output: false
   }
}
```

Listing 3–9's overriding `equals()` method begins with an if statement that uses the `instanceof` operator to determine if the argument passed to parameter `o` is an instance of the `Point` class. If not, the if statement executes `return false;`.

The `o instanceof Point` expression satisfies the last portion of the contract: For any nonnull reference value *x*, *`x`*`.equals(null)` returns false. Because null is not an instance of any class, passing this value to `equals()` causes the expression to evaluate to false.

The `o instanceof Point` expression also prevents a `ClassCastException` instance from being thrown via expression `(Point) o` in the event that you pass an object other than a `Point` object to `equals()`. (I will discuss exceptions in the next chapter.)

Following the cast, the contract's reflexivity, symmetry, and transitivity requirements are met by only allowing `Points` to be compared with other `Points`, via expression `p.x == x && p.y == y`.

The final contract requirement, consistency, is met by making sure that the `equals()` method is deterministic. In other words, this method does not rely on any field value that could change from method call to method call.

> **TIP:** You can optimize the performance of a time-consuming `equals()` method by first using `==` to determine if o's reference identifies the current object. Simply specify `if (o == this) return true;` as the `equals()` method's first statement. This optimization is not necessary in Listing 3–9's `equals()` method, which has satisfactory performance.

It is important to always override the `hashCode()` method when overriding `equals()`. I did not do so in Listing 3–9 because I have yet to formally introduce `hashCode()`.

Finalization

Finalization refers to cleanup. The `finalize()` method's Java documentation states that `finalize()` is "called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize()` method to dispose of system resources or to perform other cleanup."

`Object`'s version of `finalize()` does nothing; you must override this method with any needed cleanup code. Because the virtual machine might never call `finalize()` before an application terminates, you should provide an explicit cleanup method, and have `finalize()` call this method as a safety net in case the method is not otherwise called.

> **CAUTION:** Never depend on `finalize()` for releasing limited resources such as graphics contexts or file descriptors. For example, if an application object opens files, expecting that its `finalize()` method will close them, the application might find itself unable to open additional files when a tardy virtual machine is slow to call `finalize()`. What makes this problem worse is that `finalize()` might be called more frequently on another virtual machine, resulting in this too-many-open-files problem not revealing itself. The developer might falsely believe that the application behaves consistently across different virtual machines.

If you decide to override `finalize()`, your object's subclass layer must give its superclass layer an opportunity to perform finalization. You can accomplish this task by specifying `super.finalize();` as the last statement in your method, which Listing 3–10 demonstrates.

Listing 3–10. ***A properly coded `finalize()` method for a subclass***

```
protected void finalize() throws Throwable
{
   try
   {
      // Perform subclass cleanup.
   }
   finally
   {
      super.finalize();
   }
}
```

Listing 3–10's `finalize()` declaration appends `throws Throwable` to the method header because the cleanup code might throw an exception. If an exception is thrown, execution leaves the method and, in the absence of try-finally, `super.finalize();` never executes. (I will discuss exceptions and try-finally in Chapter 4.)

To guard against this possibility, the subclass's cleanup code executes in a compound statement that follows reserved word `try`. If an exception is thrown, Java's exception-handling logic executes the compound statement following the `finally` reserved word, and `super.finalize();` executes the superclass's `finalize()` method.

Hash Codes

The `hashCode()` method returns a 32-bit integer that identifies the current object's *hash code*, a small value that results from applying a mathematical function to a potentially large amount of data. The calculation of this value is known as *hashing*.

You must override `hashCode()` when overriding `equals()`, and in accordance with the following contract, which is specified in `hashCode()`'s Java documentation:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals(Object)` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects might improve the performance of hash tables.

Fail to obey this contract and your class's instances will not work properly with Java's hash-based collections, such as `HashMap`. (I will discuss collections in Chapter 8.)

If you override `equals()` but not `hashCode()`, you most importantly violate the second item in the contract: The hash codes of equal objects must also be equal. This violation can lead to serious consequences, as demonstrated in Listing 3–11.

Listing 3–11. *The problem of not overriding* `hashCode()`

```
java.util. Map map = new java.util.HashMap();
map.put(p1, "first point");
System.out.println(map.get(p1)); // Output: first point
System.out.println(map.get(new Point(10, 20))); // Output: null
```

Assume that Listing 3–11's statements are appended to Listing 3–9's `main()` method. After `main()` creates its `Point` objects and calls its `System.out.println()` methods, it executes Listing 3–11's statements, which perform the following tasks:

- The first statement instantiates the `HashMap` class, which is located in the `java.util` package. (I will discuss packages in the next chapter.)
- The second statement calls `HashMap`'s `put()` method to store Listing 3–9's `p1` object key and the `"first point"` value in the hashmap.
- The third statement retrieves the value of the hashmap entry whose `Point` key is logically equal to `p1` via `HashMap`'s `get()` method.
- The fourth statement is equivalent to the third statement, but returns the null reference instead of `"first point"`.

Although objects `p1` and `Point(10, 20)` are logically equivalent, these objects have different hash codes, resulting in each object referring to a different entry in the hashmap. If an object is not stored (via `put()`) in that entry, `get()` returns null.

Correcting this problem requires that `hashCode()` be overridden in order to return the same integer value for logically equivalent objects. I will show you how to accomplish this task when I discuss `HashMap` in Chapter 8.

String Representation

The `toString()` method returns a string-based representation of the current object. This representation defaults to the object's class name, followed by the `@` symbol, followed by a hexadecimal representation of the object's hash code.

For example, if you were to execute `System.out.println(p1);` to output Listing 3–9's `p1` object, you would see a line of output similar to `Point@3e25a5`. (`System.out.println()` calls `p1`'s inherited `toString()` method behind the scenes.)

You should strive to override `toString()` so that it returns a concise but meaningful description of the object. For example, you might declare, in Listing 3–9's `Point` class, a `toString()` method that is similar to Listing 3–12's `toString()` method.

Listing 3–12. *Returning a meaningful string-based representation of a* `Point` *object*

```
public String toString()
{
   return "(" + x + ", " + y + ")";
}
```

This time, executing `System.out.println(p1);` results in more meaningful output, such as `(10, 20)`.

Composition

Implementation inheritance and composition offer two different approaches to reusing code. As you have learned, implementation inheritance is concerned with extending a class with a new class, which is based upon an “is-a” relationship between them: a `Car` is a `Vehicle`, for example.

On the other hand, *composition* is concerned with composing classes out of other classes, which is based upon a “has-a” relationship between them. For example, a `Car` has an `Engine`, `Wheels`, and a `SteeringWheel`.

You have already seen examples of composition in Chapter 2 and this chapter. For example, Chapter 2’s `CheckingAccount` class included a `String owner` field. Listing 3–13’s `Car` class provides another example of composition.

Listing 3–13. *A `Car` class whose instances are composed of other objects*

```
class Car extends Vehicle
{
   private Engine engine;
   private Wheel[] wheels;
   private SteeringWheel steeringWheel;
}
```

Listing 3–13 demonstrates that composition and implementation inheritance are not mutually exclusive. Although not shown, `Car` inherits various members from its `Vehicle` superclass, in addition to providing its own `engine`, `wheels`, and `steeringwheel` fields.

The Trouble with Implementation Inheritance

Implementation inheritance is potentially dangerous, especially when the developer does not have complete control over the superclass, or when the superclass is not designed and documented with extension in mind.

The problem is that implementation inheritance breaks encapsulation. The subclass relies on implementation details in the superclass. If these details change in a new version of the superclass, the subclass might break, even if the subclass is not touched.

For example, suppose you have purchased a library of Java classes, and one of these classes describes an appointment calendar. Although you do not have access to this class’s source code, assume that Listing 3–14 describes part of its code.

Listing 3–14. *An appointment calendar class*

```
public class ApptCalendar
{
   private final static int MAX_APPT = 1000;
   private Appt[] appts;
   private int size;
```

```
   public ApptCalendar()
   {
      appts = new Appt[MAX_APPT];
      size = 0; // redundant because field automatically initialized to 0
                // adds clarity, however
   }
   public void addAppt(Appt appt)
   {
      if (size == appts.length)
         return; // array is full
      appts[size++] = appt;
   }
   public void addAppts(Appt[] appts)
   {
      for (int i = 0; i < appts.length; i++)
         addAppt(appts[i]);
   }
}
```

Listing 3–14's `ApptCalendar` class stores an array of appointments, with each appointment described by an `Appt` instance. For this discussion, the details of `Appt` are irrelevant.

Suppose you want to log each appointment in a file. Because a logging capability is not provided, you extend `ApptCalendar` with Listing 3–15's `LoggingApptCalendar` class, which adds logging behavior in overriding `addAppt()` and `addAppts()` methods.

Listing 3–15. *Extending the appointment calendar class*

```
public class LoggingApptCalendar extends ApptCalendar
{
   // A constructor is not necessary because the Java compiler will add a
   // noargument constructor that calls the superclass's noargument
   // constructor by default.
   @Override public void addAppt(Appt appt)
   {
      Logger.log(appt.toString());
      super.addAppt(appt);
   }
   @Override public void addAppts(Appt[] appts)
   {
      for (int i = 0; i < appts.length; i++)
         Logger.log(appts[i].toString());
      super.addAppts(appts);
   }
}
```

Listing 3–15's `LoggingApptCalendar` class relies on a `Logger` class whose `log()` class method logs a string to a file (the details are unimportant). Notice the use of `toString()` to convert an `Appt` object to a `String` object, which is then passed to `log()`.

Although this class looks okay, it does not work as you might expect. Suppose you instantiate this class and add a few `Appt` instances to this instance via `addAppts()`, as demonstrated in Listing 3–16.

Listing 3–16. *Demonstrating the logging appointment calendar*

```
LoggingApptCalendar lapptc = new LoggingApptCalendar();
lapptc.addAppts(new Appt[] {new Appt(), new Appt(), new Appt()});
```

If you also add a `System.out.println()` method call to `Logger`'s `log()` method, to output this method's argument, you will discover that `log()` outputs a total of six messages; each of the expected three messages (one per `Appt` object) is duplicated.

When `LoggingApptCalendar`'s `addAppts()` method is called, it first calls `Logger.log()` for each `Appt` instance in the `appts` array that is passed to `addAppts()`. This method then calls `ApptCalendar`'s `addAppts()` method via `super.addAppt(appt);`.

`ApptCalendar`'s `addAppts()` method calls `LoggingApptCalendar`'s overriding `addAppt()` method for each `Appt` instance in its `appts` array argument. `addAppt()` calls `Logger.log()` to log its `appt` argument, and you end up with three additional logged messages.

If you did not override the `addAppts()` method, this problem would go away. However, the subclass would be tied to an implementation detail: `ApptCalendar`'s `addAppts()` method calls `addAppt()`.

It is not a good idea to rely on an implementation detail when the detail is not documented. (I previously stated that you do not have access to `ApptCalendar`'s source code.) When a detail is not documented, it can change in a new version of the class.

Because a base class change can break a subclass, this problem is known as the *fragile base class problem*. A related cause of fragility that also has to do with overriding methods occurs when new methods are added to a superclass in a subsequent release.

For example, suppose a new version of the library introduces a new `public void addAppt(Appt appt, boolean unique)` method into the `ApptCalendar` class. This method adds the `appt` instance to the calendar when `unique` is false, and, when `unique` is true, adds the `appt` instance only if it has not previously been added.

Because this method has been added after the `LoggingApptCalendar` class was created, `LoggingApptCalendar` does not override the new `addAppt()` method with a call to `Logger.log()`. As a result, `Appt` instances passed to the new `addAppt()` method are not logged.

Here is another problem: You introduce a method into the subclass that is not also in the superclass. A new version of the superclass presents a new method that matches the subclass method signature and return type. Your subclass method now overrides the superclass method, and probably does not fulfill the superclass method's contract.

There is a way to make these problems disappear. Instead of extending the superclass, create a private field in a new class, and have this field reference an instance of the superclass. This task demonstrates composition because you are forming a has-a relationship between the new class and the superclass.

Additionally, have each of the new class's instance methods call the corresponding superclass method via the superclass instance that was saved in the private field, and also return the called method's return value. This task is known as *forwarding*, and the new methods are known as *forwarding methods*.

Listing 3–17 presents an improved `LoggingApptCalendar` class that uses composition and forwarding to forever eliminate the fragile base class problem and the additional problem of unanticipated method overriding.

Listing 3–17. *A composed logging appointment calendar class*

```
public class LoggingApptCalendar
{
   private ApptCalendar apptCal;
   public LoggingApptCalendar(ApptCalendar apptCal)
   {
      this.apptCal = apptCal;
   }
   public void addAppt(Appt appt)
   {
      Logger.log(appt.toString());
      apptCal.addAppt(appt);
   }
   public void addAppts(Appt[] appts)
   {
      for (int i = 0; i < appts.length; i++)
         Logger.log(appts[i].toString());
      apptCal.addAppts(appts);
   }
}
```

Listing 3–17's `LoggingApptCalendar` class does not depend upon implementation details of the `ApptCalendar` class. You can add new methods to `ApptCalendar` and they will not break `LoggingApptCalendar`.

> **NOTE:** `LoggingApptCalendar` is an example of a *wrapper class*, a class whose instances wrap other instances. Each `LoggingApptCalendar` instance wraps an `ApptCalendar` instance.
>
> `LoggingApptCalendar` is also an example of the *Decorator design pattern*, which is presented on page 175 of *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; ISBN: 0201633612).

When should you extend a class and when should you use a wrapper class? Extend a class when an is-a relationship exists between the superclass and the subclass, and either you have control over the superclass or the superclass has been designed and documented for class extension. Otherwise, use a wrapper class.

What does "design and document for class extension" mean? Design means provide protected methods that hook into the class's inner workings (to support writing efficient subclasses), and ensure that constructors and the `clone()` method never call overridable methods. Document means clearly state the impact of overriding methods.

CAUTION: Wrapper classes should not be used in a *callback framework*, an object framework in which an object passes its own reference to another object (via `this`) so that the latter object can call the former object's methods at a later time. This "calling back to the former object's method" is known as a *callback*. Because the wrapped object does not know of its wrapper class, it passes only its reference (via `this`), and resulting callbacks do not involve the wrapper class's methods.

Polymorphism

Polymorphism is the ability to change forms. Examples of polymorphism abound in nature. For example, water is naturally a liquid, but it changes to a solid when frozen, and it changes to a gas when heated to its boiling point.

Java supports several kinds of polymorphism:

- *Coercion*: An operation serves multiple types through implicit type conversion. For example, division lets you divide an integer by another integer, or divide a floating-point value by another floating-point value. If one operand is an integer and the other operand is a floating-point value, the compiler *coerces* (implicitly converts) the integer to a floating-point value, to prevent a type error. (There is no division operation that supports an integer operand and a floating-point operand.) Passing a subclass object reference to a method's superclass parameter is another example of coercion polymorphism. The compiler coerces the subclass type to the superclass type, to restrict operations to those of the superclass.
- *Overloading*: The same operator symbol or method name can be used in different contexts. For example, + can be used to perform integer division, floating-point division, or string concatenation, depending on the types of its operands. Also, multiple methods having the same name can appear in a class (through declaration and/or inheritance).
- *Parametric*: Within a class declaration, a field name can associate with different types and a method name can associate with different parameter and return types. The field and method can then take on different types in each class instance. For example, a field might be of type `Integer` and a method might return an `Integer` in one class instance, and the same field might be of type `String` and the same method might return a `String` in another class instance. Java supports parametric polymorphism via generics, which I will discuss in Chapter 5.

- *Subtype*: A type can serve as another type's subtype. When a subtype instance appears in a supertype context, executing a supertype operation on the subtype instance results in the subtype's version of that operation executing. For example, suppose that Circle is a subclass of Point, and that both classes contain a draw() method. Assigning a Circle instance to a variable of type Point, and then calling the draw() method via this variable, results in Circle's draw() method being called.

Many developers do not regard coercion and overloading as valid kinds of polymorphism. They see coercion and overloading as nothing more than type conversions and *syntactic sugar* (syntax that simplifies a language, making it "sweeter" to use). In contrast, parametric and subtype are regarded as valid polymorphisms.

This section focuses on subtype polymorphism by first examining upcasting and late binding. The section then introduces you to abstract classes and abstract methods, downcasting and runtime type identification, and covariant return types.

Upcasting and Late Binding

Listing 3–9's Point class represents a point as an x-y pair. Because a circle (in this example) is an x-y pair denoting its center, and has a radius denoting its extent, you can extend Point with a Circle class that introduces a radius field. Check out Listing 3–18.

Listing 3–18. *A Circle class extending the Point class*

```
class Circle extends Point
{
   private int radius;
   Circle(int x, int y, int radius)
   {
      super(x, y);
      this.radius = radius;
   }
   int getRadius()
   {
      return radius;
   }
}
```

The fact that Circle is really a Point with a radius implies that you can treat a Circle instance as if it was a Point instance. Accomplish this task by assigning the Circle instance to a Point variable, as demonstrated in Listing 3–19.

Listing 3–19. *Upcasting from Circle to Point*

```
Circle c = new Circle(10, 20, 30);
Point p = c;
```

The cast operator is not needed to convert from Circle to Point because access to a Circle instance via Point's interface is legal. After all, a Circle is at least a Point. This assignment is known as *upcasting* because you are implicitly casting up the type hierarchy (from the Circle subclass to the Point superclass).

After upcasting `Circle` to `Point`, you cannot call `Circle`'s `getRadius()` method because this method is not part of `Point`'s interface. Losing access to subtype features after narrowing it to a superclass seems useless, but is necessary for achieving subtype polymorphism.

In addition to upcasting the subclass instance to a variable of the superclass type, subtype polymorphism involves declaring a method in the superclass and overriding this method in the subclass.

For example, suppose `Point` and `Circle` are to be part of a graphics application, and you need to introduce a `draw()` method into each class to draw a point and a circle, respectively. You end with the class structure shown in Listing 3–20.

Listing 3–20. ***Declaring a graphics application's `Point` and `Circle` classes***

```
class Point
{
   private int x, y;
   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
   @Override public String toString()
   {
      return "(" + x + ", " + y + ")";
   }
   void draw()
   {
      System.out.println("Point drawn at " + toString ());
   }
}
class Circle extends Point
{
   private int radius;
   Circle(int x, int y, int radius)
   {
      super(x, y);
      this.radius = radius;
   }
   int getRadius()
   {
      return radius;
   }
   @Override public String toString()
   {
      return "" + radius;
   }
```

```
   @Override void draw()
   {
      System.out.println("Circle drawn at " + super.toString() +
                         " with radius " + toString());
   }
}
```

Although the `draw()` methods will ultimately draw graphics shapes, simulating their behaviors via `System.out.println()` method calls is sufficient during the early testing phase of the graphics application.

Now that you have temporarily finished with `Point` and `Circle`, you want to test their `draw()` methods in a simulated version of the graphics application. To achieve this objective, you write Listing 3–21's `Graphics` class.

Listing 3–21. *A `Graphics` class for testing `Point`'s and `Circle`'s `draw()` methods*

```
class Graphics
{
   public static void main(String[] args)
   {
      Point[] points = new Point[] {new Point(10, 20), new Circle(10, 20, 30)};
      for (int i = 0; i < points.length; i++)
         points[i].draw();
   }
}
```

Listing 3–21's `main()` method first declares an array of `Points`. Upcasting is demonstrated by first having the array's initializer instantiate the `Circle` class, and then by assigning this instance's reference to the second element in the `points` array.

Moving on, `main()` uses a for loop to call each `Point` element's `draw()` method. Because the first iteration calls `Point`'s `draw()` method, whereas the second iteration calls `Circle`'s `draw()` method, you observe the following output:

```
Point drawn at (10, 20)
Circle drawn at (10, 20) with radius 30
```

How does Java "know" that it must call `Circle`'s `draw()` method on the second loop iteration? Should it not call `Point`'s `draw()` method because `Circle` is being treated as a `Point` thanks to the upcast?

At compile time, the compiler does not know which method to call. All it can do is verify that a method exists in the superclass, and verify that the method call's arguments list and return type match the superclass's method declaration.

In lieu of knowing which method to call, the compiler inserts an instruction into the compiled code that, at runtime, fetches and uses whatever reference is in `points[1]` to call the correct `draw()` method. This task is known as *late binding*.

Late binding is used for calls to non-`final` instance methods. For all other method calls, the compiler knows which method to call, and inserts an instruction into the compiled code that calls the method associated with the variable's type (not its value). This task is known as *early binding*.

Abstract Classes and Abstract Methods

Suppose new requirements dictate that your graphics application must include a Rectangle class. Furthermore, this class must include a draw() method, and this method must be tested in a manner similar to that shown in Listing 3–21's Graphics class.

In contrast to Circle, which is a Point with a radius, it does not make sense to think of a Rectangle as a being a Point with a width and height. Rather, a Rectangle instance would probably be composed of a Point (indicating its origin) and a width and height.

Because circles, points, and rectangles are examples of shapes, it makes more sense to declare a Shape class with its own draw() method than to specify class Rectangle extends Point. Listing 3–22 presents Shape's declaration.

Listing 3–22. *Declaring a Shape class*

```
class Shape
{
   void draw() {}
}
```

You can now refactor Point to extend Listing 3–22's Shape class, leave Circle as is, and introduce a Rectangle class that extends Shape. You can then refactor Listing 3–21's Graphics class's main() method to take Shape into account. Check out Listing 3–23.

Listing 3–23. *A new main() method for the Graphics class takes Shape into account*

```
public static void main(String[] args)
{
   Shape[] shapes = new Shape[] {new Point(10, 20), new Circle(10, 20, 30),
                                 new Rectangle(20, 30, 15, 25)};
   for (int i = 0; i < shapes.length; i++)
      shapes[i].draw();
}
```

Because Point and Rectangle directly extend Shape, and because Circle indirectly extends Shape by extending Point, Listing 3–23's main() method will call the appropriate subclass's draw() method in response to shapes[i].draw();.

Although the introduction of Shape makes our code more flexible, there is a problem. What is to stop us from instantiating Shape and adding this meaningless instance to the shapes array, as Listing 3–24 demonstrates?

Listing 3–24. *A useless instantiation*

```
Shape[] shapes = new Shape[] {new Point(10, 20), new Circle(10, 20, 30),
                              new Rectangle(20, 30, 15, 25), new Shape()};
```

What does it mean to instantiate Shape? Because this class describes an abstract concept, what does it mean to draw a generic shape? Fortunately, Java provides a solution to this problem, which is demonstrated in Listing 3–25.

Listing 3–25. *Abstracting the Shape class*

```
abstract class Shape
{
   abstract void draw(); // semicolon is required
}
```

Listing 3–25 uses Java's `abstract` reserved word to declare a class that cannot be instantiated. The compiler reports an error should you try to instantiate this class.

> **TIP:** Get into the habit of declaring classes that describe generic categories (such as shape, animal, vehicle, and account) `abstract`. This way, you will not inadvertently instantiate them.

The `abstract` reserved word is also used to declare a method without a body. The `draw()` method does not need a body because it cannot draw an abstract shape.

> **CAUTION:** The compiler reports an error if you attempt to declare a class that is both abstract and final. For example, `abstract final class Shape` is an error because an abstract class cannot be instantiated and a final class cannot be extended.
>
> The compiler also reports an error if you declare a method to be abstract but do not declare its class to be abstract. For example, removing `abstract` from the `Shape` class's header in Listing 3–25 results in an error. This removal is an error because a non-`abstract` (concrete) class cannot be instantiated if it contains an abstract method.
>
> When you extend an abstract class, the extending class must override all of the abstract class's abstract methods, or else the extending class must itself be declared to be abstract; otherwise, the compiler will report an error.

An abstract class can contain non-`abstract` methods in addition to or instead of `abstract` methods. For example, Listing 3–2's `Vehicle` class could have been declared `abstract`. The constructor would still be present, to initialize private fields, even though you could not instantiate the resulting class.

Downcasting and Runtime Type Identification

Moving up the type hierarchy via upcasting results in loss of access to subtype features. For example, assigning a `Circle` instance to `Point` variable `p` means that you cannot use `p` to call `Circle`'s `getRadius()` method.

However, it is possible to once again access the `Circle` instance's `getRadius()` method by performing an explicit cast operation; for example, `Circle c = (Circle) p;`. This assignment is known as *downcasting* because you are explicitly moving down the type hierarchy (from the `Point` superclass to the `Circle` subclass).

Although an upcast is always safe (the superclass's interface is a subset of the subclass's interface), the same cannot be said of a downcast. Listing 3–26 shows you what kind of trouble you can get into when downcasting is used incorrectly.

Listing 3–26. *The trouble with downcasting*

```
class A
{
}
class B extends A
{
   void d() {}
}
class C
{
   public static void main(String[] args)
   {
      A a = new A();
      B b = (B) a;
      b.d();
   }
}
```

Listing 3–26 presents a class hierarchy consisting of a superclass named A and a subclass named B. Although A does not declare any members, B declares a single d() method.

A third class named C provides a main() method that first instantiates A, and then tries to downcast this instance to B and assign the result to variable b. The compiler will not complain because downcasting from a superclass to a subclass in the same type hierarchy is legal.

However, if the assignment is allowed, the application will undoubtedly crash when it tries to execute b.d();. The crash happens because the virtual machine will attempt to call a method that does not exist—class A does not have a d() method.

Fortunately, this scenario will never happen because the virtual machine verifies that the cast is legal. Because it detects that A does not have a d() method, it does not permit the cast by throwing an instance of the ClassCastException class.

The virtual machine's cast verification illustrates *runtime type identification* (or RTTI, for short). Cast verification performs RTTI by examining the type of the cast operator's operand to see if the cast should be allowed. Clearly, the cast should not be allowed.

A second form of RTTI involves the instanceof operator. This operator checks the left operand to see if it is an instance of the right operand, and returns true if this is the case. Listing 3–27 introduces instanceof to Listing 3–26 to prevent the ClassCastException.

Listing 3–27. *Preventing a ClassCastException*

```
if(a instanceof B)
{
   B b = (B) a;
   b.d();
}
```

The instanceof operator detects that variable a's instance was not created from B and returns false to indicate this fact. As a result, the code that performs the illegal cast will not execute. (Overuse of instanceof probably indicates poor software design.)

Because a subtype is a kind of supertype, instanceof will return true when its left operand is a subtype instance or a supertype instance of its right operand supertype. Listing 3–28 provides a demonstration.

Listing 3–28. *Subtype and supertype instances of a supertype*

```
A a = new A();
B b = new B();
System.out.println(b instanceof A); // Output: true
System.out.println(a instanceof A); // Output: true
```

Listing 3–28, which assumes the class structure shown in Listing 3–26, instantiates superclass A and subclass B. The first System.out.println() method call outputs true because b's reference identifies an instance of a subclass of A; the second System.out.println() method call outputs true because a's reference identifies an instance of superclass A.

So far, you have encountered two forms of RTTI. Java also supports a third form that is known as reflection. I will introduce you to this form of RTTI when I cover reflection in Chapter 7.

Covariant Return Types

A *covariant return type* is a method return type that, in the superclass's method declaration, is the supertype of the return type in the subclass's overriding method declaration. Listing 3–29 provides a demonstration of this language feature.

Listing 3–29. *A demonstration of covariant return types*

```
class Zip
{
   ZipFile getArchive(String name) throws IOException
   {
      return new ZipFile(name); // ZipFile is located in the java.util.zip package
   }
}
class Jar extends Zip
{
   @Override JarFile getArchive(String name) throws IOException
   {
      return new JarFile(name); // JarFile is located in the java.util.jar package
   }
}
class Archive
{
   public static void main(String[] args) throws IOException
   {
      if (args.length == 2 && args[0].equals("-zip"))
      {
         ZipFile zf = new Zip().getArchive(args[1]);
      }
      else
      if (args.length == 2 && args[0].equals("-jar"))
      {
         JarFile jf = new Jar().getArchive(args[1]);
```

```
      }
   }
}
```

Listing 3–29 declares a `Zip` superclass and a `Jar` subclass; each class declares a `getArchive()` method. `Zip`'s method has its return type set to `ZipFile`, whereas `Jar`'s overriding method has its return type set to `JarFile`, a subclass of `ZipFile`.

Covariant return types minimize upcasting and downcasting. For example, `Jar`'s `getArchive()` method does not need to upcast its `JarFile` instance to its `JarFile` return type. Furthermore, this instance does not need to be downcast to `JarFile` when assigning to variable `jf`.

In the absence of covariant return types, you would end up with Listing 3–30.

Listing 3–30. ***Upcasting and downcasting in the absence of covariant return types***

```
class Zip
{
   ZipFile getArchive(String name) throws IOException
   {
      return new ZipFile(name);
   }
}
class Jar extends Zip
{
   @Override ZipFile getArchive(String name) throws IOException
   {
      return new JarFile(name);
   }
}
class Archive2
{
   public static void main(String[] args) throws IOException
   {
      if (args.length == 2 && args[0].equals("-zip"))
      {
         ZipFile zf = new Zip().getArchive(args[1]);
      }
      else
      if (args.length == 2 && args[0].equals("-jar"))
      {
         JarFile jf = (JarFile) new Jar().getArchive(args[1]);
      }
   }
}
```

In Listing 3–30, the first bolded code reveals an upcast from `JarFile` to `ZipFile`, and the second bolded code uses the required `(JarFile)` cast operator to downcast from `ZipFile` to `jf`, which is of type `JarFile`.

Interfaces

In Chapter 2, I stated that every class *X* exposes an *interface*, which is a protocol or contract consisting of constructors, methods, and (possibly) fields that are made available to objects created from other classes for use in creating and communicating with *X*'s objects.

> **NOTE:** A *contract* is an agreement between two parties. In this case, those parties are a class and *clients* (external constructors, methods, class initializers, and instance initializers) that communicate with the class's instances by calling constructors and methods, and by accessing fields (typically `public static final` fields, or constants). The essence of the contract is that the class promises to not change its interface, which would break clients that depend upon the interface.

Java formalizes the interface concept by providing reserved word `interface`, which is used to introduce a type without implementation. Java also provides language features to declare, implement, and extend interfaces. After looking at interface declaration, implementation, and extension, this section explains the rationale for using interfaces.

Declaring Interfaces

An interface declaration consists of a header followed by a body. At minimum, the header consists of reserved word `interface` followed by a name that identifies the interface. The body starts with an open brace character and ends with a close brace. Sandwiched between these delimiters are constant and method header declarations. Consider Listing 3–31.

Listing 3–31. *Declaring a `Drawable` interface*

```
interface Drawable
{
   int RED = 1;   // For simplicity, integer constants are used. These constants are
   int GREEN = 2; // not that descriptive, as you will see.
   int BLUE = 3;
   int BLACK = 4;
   void draw(int color);
}
```

Listing 3–31 declares an interface named `Drawable`. By convention, an interface's name begins with an uppercase letter. Furthermore, the first letter of each subsequent word in a multiword interface name is capitalized.

> **NOTE:** Many interface names end with the `able` suffix. For example, the Java's standard class library includes interfaces named `Adjustable`, `Callable`, `Comparable`, `Cloneable`, `Iterable`, `Runnable`, and `Serializable`. It is not mandatory to use this suffix. For example, the standard class library also provides interfaces named `CharSequence`, `Collection`, `Composite`, `Executor`, `Future`, `Iterator`, `List`, `Map`, and `Set`.

`Drawable` declares four fields that identify color constants. `Drawable` also declares a `draw()` method that must be called with one of these constants to specify the color used to draw something.

> **NOTE:** As with a class declaration, you can precede `interface` with `public`, to make your interface accessible to code outside of its package. (I will discuss packages in the next chapter). Otherwise, the interface is only accessible to other types in its package.
>
> You can also precede `interface` with `abstract`, to emphasize that an interface is abstract. Because an interface is already abstract, it is redundant to specify `abstract` in the interface's declaration.
>
> An interface's fields are implicitly declared `public`, `static`, and `final`. It is therefore redundant to declare them with these reserved words. Because these fields are constants, they must be explicitly initialized; otherwise, the compiler reports an error.
>
> An interface's methods are implicitly declared `public` and `abstract`. Therefore, it is redundant to declare them with these reserved words. Because these methods must be instance methods, do not declare them `static` or the compiler will report errors.

`Drawable` identifies a type that specifies what to do (draw something) but not how to do it. Implementation details are left up to classes that implement this interface. Instances of such classes are known as *drawables* because they know how to draw themselves.

> **NOTE:** An interface that declares no members is known as a *marker interface* or a *tagging interface*. It associates metadata with a class. For example, the `Cloneable` marker/tagging interface states that instances of its implementing class can be shallowly cloned.
>
> RTTI is used to detect that an object's class implements a marker/tagging interface. For example, when `Object`'s `clone()` method detects, via RTTI, that the calling instance's class implements `Cloneable`, it shallowly clones the object.

Implementing Interfaces

By itself, an interface is useless. To be of any benefit to an application, the interface needs to be implemented by a class. Java provides the `implements` reserved word for this task. This reserved word is demonstrated in Listing 3–32.

Listing 3–32. *Implementing the* `Drawable` *interface*

```
class Point implements Drawable
{
   private int x, y;
   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
   @Override public String toString()
   {
      return "(" + x + ", " + y + ")";
   }
   @Override public void draw(int color)
   {
      System.out.println("Point drawn at " + toString () + " in color " + color);
   }
}
class Circle extends Point implements Drawable
{
   private int radius;
   Circle(int x, int y, int radius)
   {
      super(x, y);
      this.radius = radius;
   }
   int getRadius()
   {
      return radius;
   }
   @Override public String toString()
   {
      return "" + radius;
   }
   @Override public void draw(int color)
   {
      System.out.println("Circle drawn at " + super.toString() +
                         " with radius " + toString() + " in color " + color);
   }
}
```

Listing 3–32 retrofits Listing 3–20's class hierarchy to take advantage of Listing 3–31's `Drawable` interface. You will notice that each of classes `Point` and `Circle` implements this interface by attaching the `implements Drawable` clause to its class header.

To implement an interface, the class must specify, for each interface method header, a method whose header has the same signature and return type as the interface's method header, and a code body to go with the method header.

> **CAUTION:** When implementing a method, do not forget that the interface's methods are implicitly declared `public`. If you forget to include `public` in the implemented method's declaration, the compiler will report an error because you are attempting to assign weaker access to the implemented method.

When a class implements an interface, the class inherits the interface's constants and method headers, and overrides the method headers by providing implementations (hence the `@Override` annotation). This is known as *interface inheritance*.

It turns out that `Circle`'s header does not need the `implements Drawable` clause. If this clause is not present, `Circle` inherits `Point`'s `draw()` method, and is still considered to be a `Drawable`, whether it overrides this method or not.

An interface specifies a type whose data values are the objects whose classes implement the interface, and whose behaviors are those specified by the interface. This fact implies that you can assign an object's reference to a variable of the interface type, provided that the object's class implements the interface. Listing 3–33 provides a demonstration.

Listing 3–33. *Exercising the* `Drawable` *interface*

```
public static void main(String[] args)
{
   Drawable[] drawables = new Drawable[] {new Point(10, 20), new Circle(10, 20, 30)};
   for (int i = 0; i < drawables.length; i++)
      drawables[i].draw(Drawable.RED);
}
```

Because `Point` and `Circle` instances are drawables by virtue of these classes implementing the `Drawable` interface, it is legal to assign `Point` and `Circle` instance references to variables (including array elements) of type `Drawable`.

When you run this method, it generates the following output:

```
Point drawn at (10, 20) in color 1
Circle drawn at (10, 20) with radius 30 in color 1
```

Listing 3–31's `Drawable` interface is useful for drawing a shape's outline. Suppose you also need to fill a shape's interior. You might attempt to satisfy this requirement by declaring Listing 3–34's `Fillable` interface.

Listing 3–34. *Declaring a `Fillable` interface*

```
interface Fillable
{
   int RED = 1;
   int GREEN = 2;
   int BLUE = 3;
   int BLACK = 4;
   void fill(int color);
}
```

You can declare that the `Point` and `Circle` classes implement both interfaces by specifying `class Point implements Drawable, Fillable` and `class Circle implements Drawable, Fillable`.

> **TIP:** You can list as many interfaces as you need to implement by specifying a comma-separated list of interface names after `implements`.

Implementing multiple interfaces can lead to name collisions, and the compiler will report errors. For example, suppose that you attempt to compile Listing 3–35's interface and class declarations.

Listing 3–35. *Colliding interfaces*

```
interface A
{
   int X = 1;
   void foo();
}
interface B
{
   int X = 1;
   int foo();
}
class C implements A, B
{
   public void foo();
   public int foo() { return X; }
}
```

Each of interfaces `A` and `B` declares a constant named `X`. Despite each constant having the same type and value, the compiler will report an error when it encounters `X` in `C`'s second `foo()` method because it does not know which `X` is being inherited.

Speaking of `foo()`, the compiler reports an error when it encounters `C`'s second `foo()` declaration because `foo()` has already been declared. You cannot overload a method by changing only its return type.

The compiler will probably report additional errors. For example, the Java version 6 update 16 compiler has this to say when told to compile Listing 3–35:

```
X.java:14: foo() is already defined in C
   public int foo() { return X; }
              ^
```

```
X.java:11: C is not abstract and does not override abstract method foo() in B
class C implements A, B
^
X.java:13: foo() in C cannot implement foo() in B; attempting to use incompatible↩
 return type
found   : void
required: int
   public void foo();
               ^
X.java:14: reference to X is ambiguous, both variable X in A and variable X in B match
   public int foo() { return X; }
                             ^
4 errors
```

Extending Interfaces

Just as a subclass can extend a superclass via reserved word `extends`, you can use this reserved word to have a subinterface extend a superinterface. This is known as *interface inheritance*.

For example, the duplicate color constants in `Drawable` and `Fillable` lead to name collisions when you specify their names by themselves in an implementing class. To avoid these name collisions, prefix a name with its interface name and the member access operator, or place these constants in their own interface, and have `Drawable` and `Fillable` extend this interface, as demonstrated in Listing 3–36.

Listing 3–36. ***Extending the `Colors` interface***

```
interface Colors
{
   int RED = 1;
   int GREEN = 2;
   int BLUE = 3;
   int BLACK = 4;
}
interface Drawable extends Colors
{
   void draw(int color);
}
interface Fillable extends Colors
{
   void fill(int color);
}
```

The fact that `Drawable` and `Fillable` each inherit constants from `Colors` is not a problem for the compiler. There is only a single copy of these constants (in `Colors`) and no possibility of a name collision, and so the compiler is satisfied.

If a class can implement multiple interfaces by declaring a comma-separated list of interface names after `implements`, it seems that an interface should be able to extend multiple interfaces in a similar way. This feature is demonstrated in Listing 3–37.

Listing 3–37. *Extending a pair of interfaces*

```
interface A
{
   int X = 1;
}
interface B
{
   double X = 2.0;
}
interface C extends A, B
{
}
```

Listing 3–37 will compile even though `C` inherits two same-named constants `X` with different return types and initializers. However, if you implement `C` and then try to access `X`, as in Listing 3–38, you will run into a name collision.

Listing 3–38. *Discovering a name collision*

```
class D implements C
{
   public void output()
   {
      System.out.println(X); // Which X is accessed?
   }
}
```

Suppose you introduce a `void foo();` method header declaration into interface `A`, and an `int foo();` method header declaration into interface `B`. This time, the compiler will report an error when you attempt to compile the modified Listing 3–37.

Why Use Interfaces?

Now that the mechanics of declaring, implementing, and extending interfaces are out of the way, we can focus on the rationale for using them. Unfortunately, newcomers to Java's interfaces feature are often told that this feature was created as a workaround to Java's lack of support for multiple implementation inheritance. While interfaces are useful in this capacity, this is not their reason for existence. Instead, **Java's interfaces feature was created to give developers the utmost flexibility in designing their applications, by decoupling interface from implementation.**

If you are an adherent to *agile software development* (a group of software development methodologies based on iterative development that emphasizes keeping code simple, testing frequently, and delivering functional pieces of the application as soon as they are deliverable), you know the importance of flexible coding. You know that you cannot afford to tie your code to a specific implementation because a change in requirements for the next iteration could result in a new implementation, and you might find yourself rewriting significant amounts of code, which wastes time and slows development.

Interfaces help you achieve flexibility by decoupling interface from implementation. For example, Listing 3–23's `main()` method creates an array of objects from classes that subclass the `Shape` class, and then iterates over these objects, calling each object's `draw()` method. The only objects that can be drawn are those that subclass `Shape`.

Suppose you also have a hierarchy of classes that model resistors, transistors, and other electronic components. Each component has its own symbol that allows the component to be shown in a schematic diagram of an electronic circuit. Perhaps you want to add a drawing capability to each class that draws that component's symbol.

You might consider specifying `Shape` as the superclass of the electronic component class hierarchy. However, electronic components are not shapes so it makes no sense to place these classes in a class hierarchy rooted in `Shape`.

However, you can make each component class implement the `Drawable` interface, which lets you add expressions that instantiate these classes to Listing 3–33's `drawables` array (so you can draw their symbols). This is legal because these instances are drawables.

Wherever possible, you should strive to specify interfaces instead of classes in your code, to keep your code adaptable to change. This is especially true when working with Java's collections framework, which I will discuss at length in Chapter 8.

For now, consider a simple example that consists of the collections framework's `List` interface, and its `ArrayList` and `LinkedList` classes. Listing 3–39 shows you an example of inflexible code based on the `ArrayList` class.

Listing 3–39. *Hardwiring the `ArrayList` class into source code*

```
ArrayList<String> arrayList = new ArrayList<String>();
void dump(ArrayList<String> arrayList)
{
   // suitable code to dump out the arrayList
}
```

Listing 3–39 uses the generics-based parameterized type language feature (which I will discuss in Chapter 5) to identify the kind of objects stored in an `ArrayList` instance. In this example, `String` objects are stored.

Listing 3–39 is inflexible because it hardwires the `ArrayList` class into multiple locations. This hardwiring focuses the developer into thinking specifically about array lists instead of generically about lists.

Lack of focus is problematic when a requirements change, or perhaps a performance issue brought about by *profiling* (analyzing a running application to check its performance), suggests that the developer should have used `LinkedList`.

Listing 3–39 only requires a minimal number of changes to satisfy the new requirement. In contrast, a larger code base might need many more changes. Although you only need to change `ArrayList` to `LinkedList`, to satisfy the compiler, consider changing `arrayList` to `linkedList`, to keep *semantics* (meaning) clear—you might have to change multiple occurrences of names that refer to an `ArrayList` instance throughout the source code.

The developer is bound to lose time while refactoring the code to adapt to `LinkedList`. Instead, time could have been saved by writing Listing 3–39 to use the equivalent of constants. In other words, Listing 3–39 could have been written to rely on interfaces, and to only specify `ArrayList` in one place. Listing 3–40 shows you what the resulting code would look like.

Listing 3–40. *Using `List` to minimize referrals to the `ArrayList` implementation class*

```
List<String> list = new ArrayList<String>();
void dump(List<String> list)
{
   // suitable code to dump out the list
}
```

Listing 3–40 is much more flexible than Listing 3–39. If a requirements or profiling change suggests that `LinkedList` should be used instead of `ArrayList`, simply replace `Array` with `Linked` and you are done. You do not even have to change the parameter name.

> **NOTE:** Java provides interfaces and abstract classes for describing *abstract types* (types that cannot be instantiated). Abstract types represent abstract concepts (drawable and shape, for example), and instances of such types would be meaningless.
>
> Interfaces promote flexibility through lack of implementation—`Drawable` and `List` illustrate this flexibility. They are not tied to any single class hierarchy, but can be implemented by any class in any hierarchy.
>
> Abstract classes support implementation, but can be genuinely abstract (Listing 3–25's abstract `Shape` class, for example). However, they are limited to appearing in the upper levels of class hierarchies.
>
> Interfaces and abstract classes can be used together. For example, the collections framework provides `List`, `Map`, and `Set` interfaces; and `AbstractList`, `AbstractMap`, and `AbstractSet` abstract classes that provide skeletal implementations of these interfaces.
>
> The skeletal implementations make it easy for you to create your own interface implementations, to address your unique requirements. If they do not meet your needs, you can optionally have your class directly implement the appropriate interface.

EXERCISES

The following exercises are designed to test your understanding of Java's object-oriented language features:

1. What is implementation inheritance?
2. How does Java support implementation inheritance?
3. Can a subclass have two or more superclasses?

4. How do you prevent a class from being subclassed?
5. True or false: The `super()` call can appear in any method.
6. If a superclass declares a constructor with one or more parameters, and if a subclass constructor does not use `super()` to call that constructor, why does the compiler report an error?
7. What is an immutable class?
8. True or false: A class can inherit constructors.
9. What does it mean to override a method?
10. What is required to call a superclass method from its overriding subclass method?
11. How do you prevent a method from being overridden?
12. Why can you not make an overriding subclass method less accessible than the superclass method it is overriding?
13. How do you tell the compiler that a method overrides another method?
14. Why does Java not support multiple implementation inheritance?
15. What is the name of Java's ultimate superclass?
16. What is the purpose of the `clone()` method?
17. When does `Object`'s `clone()` method throw `CloneNotSupportedException`?
18. Explain the difference between shallow copying and deep copying.
19. Can the == operator be used to determine if two objects are logically equivalent? Why or why not?
20. What does `Object`'s `equals()` method accomplish?
21. Does expression `"abc" == "a" + "bc"` return true or false?
22. How can you optimize a time-consuming `equals()` method?
23. What is the purpose of the `finalize()` method?
24. Should you rely on `finalize()` for closing open files? Why or why not?
25. What is a hash code?
26. True or false: You should override the `hashCode()` method whenever your override the `equals()` method.
27. What does `Object`'s `toString()` method return?
28. Why should you override `toString()`?
29. What is composition?
30. True or false: Composition is used to implement is-a relationships and implementation inheritance is used to describe has-a relationships.
31. Identify the fundamental problem of implementation inheritance. How do you fix this problem?

32. What is subtype polymorphism?
33. How is subtype polymorphism accomplished?
34. Why would you use abstract classes and abstract methods?
35. Can an abstract class contain concrete methods?
36. What is the purpose of downcasting?
37. List the three forms of RTTI.
38. What is a covariant return type?
39. How do you formally declare an interface?
40. True or false: You can precede an interface declaration with the `abstract` reserved word.
41. What is a marker interface?
42. What is interface inheritance?
43. How do you implement an interface?
44. What problem might you encounter when you implement multiple interfaces?
45. How do you form a hierarchy of interfaces?
46. Why is Java's interfaces feature so important?
47. What do interfaces and abstract classes accomplish?
48. How do interfaces and abstract classes differ?
49. Model part of an animal hierarchy by declaring `Animal`, `Bird`, `Fish`, `AmericanRobin`, `DomesticCanary`, `RainbowTrout`, and `SockeyeSalmon` classes:
    - `Animal` is `public` and `abstract`, declares `private String`-based `kind` and `appearance` fields, declares a `public` constructor that initializes these fields to passed-in arguments, declares `public` and `abstract` `eat()` and `move()` methods that take no arguments and whose return type is `void`, and overrides the `toString()` method to output the contents of `kind` and `appearance`.
    - `Bird` is `public` and `abstract`, extends `Animal`, declares a `public` constructor that passes its `kind` and `appearance` parameter values to its superclass constructor, overrides its `eat()` method to output `eats seeds and insects` (via `System.out.println()`), and overrides its `move()` method to output `flies through the air`.
    - `Fish` is `public` and `abstract`, extends `Animal`, declares a `public` constructor that passes its `kind` and `appearance` parameter values to its superclass constructor, overrides its `eat()` method to output `eats krill, algae, and insects`, and overrides its `move()` method to output `swims through the water`.

- AmericanRobin is public, extends Bird, and declares a public noargument constructor that passes "americanrobin" and "red breast" to its superclass constructor.
- DomesticCanary is public, extends Bird, and declares a public noargument constructor that passes "domesticcanary" and "yellow, orange, black, brown, white, red" to its superclass constructor.
- RainbowTrout is public, extends Fish, and declares a public noargument constructor that passes "rainbowtrout" and "bands of brilliant speckled multicolored stripes running nearly the whole length of its body" to its superclass constructor.
- SockeyeSalmon is public, extends Fish, and declares a public noargument constructor that passes "sockeyesalmon" and "bright red with a green head" to its superclass constructor.

For brevity, I have omitted from the Animal hierarchy abstract Robin, Canary, Trout, and Salmon classes that generalize robins, canaries, trout, and salmon. Perhaps you might want to include these classes in the hierarchy.

Although this exercise illustrates the accurate modeling of a natural scenario using inheritance, it also reveals the potential for class explosion—too many classes may be introduced to model a scenario, and it might be difficult to maintain all of these classes. Keep this in mind when modeling with inheritance.

50. Continuing from the previous exercise, declare an Animals class with a main() method. This method first declares an animals array that is initialized to AmericanRobin, RainbowTrout, DomesticCanary, and SockeyeSalmon objects. The method then iterates over this array, first outputting animals[i] (which causes toString() to be called), and then calling each object's eat() and move() methods (demonstrating subtype polymorphism).

51. Continuing from the previous exercise, declare a public Countable interface with a String getID() method. Modify Animal to implement Countable and have this method return kind's value. Modify Animals to initialize the animals array to AmericanRobin, RainbowTrout, DomesticCanary, SockeyeSalmon, RainbowTrout, and AmericanRobin objects. Also, introduce code that computes a census of each kind of animal. This code will use the Census class that is declared in Listing 3–41.

Listing 3–41. *The Census class stores census data on four kinds of animals*

```
public class Census
{
   public final static int SIZE = 4;
   private String[] IDs;
   private int[] counts;
   public Census()
   {
      IDs = new String[SIZE];
      counts = new int[SIZE];
   }
   public String get(int index)
   {
      return IDs[index] + " " + counts[index];
```

```
   }
   public void update(String ID)
   {
      for (int i = 0; i < IDs.length; i++)
      {
         // If ID not already stored in the IDs array (which is indicated by
         // the first null entry that is found), store ID in this array, and
         // also assign 1 to the associated element in the counts array, to
         // initialize the census for that ID.
         if (IDs[i] == null)
         {
            IDs[i] = ID;
            counts[i] = 1;
            return;
         }

         // If a matching ID is found, increment the associated element in
         // the counts array to update the census for that ID.
         if (IDs[i].equals(ID))
         {
            counts[i]++;
            return;
         }
      }
   }
}
```

Summary

An understanding of Java's fundamental language features must take inheritance and polymorphism into account. Java supports two forms of inheritance: implementation via class extension, and interface via interface implementation or interface extension.

Java supports four kinds of polymorphism: coercion, overloading, parametric, and subtype. Subtype polymorphism is used to invoke subclass methods via references to subclass objects that are stored in variables of the superclass type.

Java's interfaces feature is essential for writing extremely flexible code. It achieves this flexibility by decoupling interface from implementation. Classes that implement an interface provide their own implementations.

You now have enough language knowledge to write interesting Java applications, but Java's advanced language features related to nested types, packages, static imports, and exceptions help simplify this task. Chapter 4 focuses on these feature categories.