Inheritance and Reflection

Topics in This Chapter

Chapter 4

The preceding chapters introduced you to classes and interfaces. In this chapter, you will learn about another fundamental concept of object-oriented programming: inheritance. Inheritance is the process of creating new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods, and you can add new methods and fields.

NOTE: Instance variables and static variables are collectively called *fields*. The fields, methods, and nested classes/interfaces inside a class are collectively called its *members*.

This chapter also covers reflection, the ability to find out more about classes and their members in a running program. Reflection is a powerful feature, but it is undeniably complex. Since reflection is of greater interest to tool builders than to application programmers, you can probably glance over that part of the chapter upon first reading and come back to it later.

The key points of this chapter are:

1. A subclass can inherit or override methods from the superclass, provided they are not `private`.
2. Use the `super` keyword to invoke a superclass method or constructor.
3. A `final` method cannot be overridden; a `final` class cannot be extended.

4. An `abstract` method has no implementation; an `abstract` class cannot be instantiated.
5. A `protected` member of a superclass is accessible in a subclass method, but only when applied to objects of the same subclass. It is also accessible in its package.
6. Every class is a subclass of `Object` which provides the `toString`, `equals`, `hashCode`, and `clone` methods.
7. Each enumerated type is a subclass of `Enum` which provides instance methods `toString` and `compareTo`, and a static `valueOf` method.
8. The `Class` class provides information about a Java type, which can be a class, array, interface, primitive type, or `void`.
9. You can use a `Class` object to load resources that are placed alongside class files.
10. You can load classes from locations other than the class path by using a class loader.
11. The `ServiceLoader` class provides a mechanism for locating and selecting service implementations.
12. The reflection library enables programs to discover members of objects, access variables, and invoke methods.
13. Proxy objects dynamically implement arbitrary interfaces, routing all method invocations to a handler.

4.1 Extending a Class

Let's return to the `Employee` class that we discussed in Chapter 2. Suppose (alas) you work for a company at which managers are treated differently from other employees. Managers are, of course, just like employees in many respects. Both employees and managers are paid a salary. However, while employees are expected to complete their assigned tasks in return for their salary, managers get bonuses if they actually achieve what they are supposed to do. This is the kind of situation that can be modeled with inheritance.

4.1.1 Super- and Subclasses

Let's define a new class, `Manager`, retaining some functionality of the `Employee` class but specifying how managers are different.

```
public class Manager extends Employee {
    added fields
    added or overriding methods
}
```

The keyword `extends` indicates that you are making a new class that derives from an existing class. The existing class is called the *superclass* and the new class is called the *subclass*. In our example, the `Employee` class is the superclass and the `Manager` class is the subclass. Note that the superclass is not "superior" to its subclass. The opposite is true: Subclasses have more functionality than their superclasses. The super/sub terminology comes from set theory. The set of managers is a subset of the set of employees.

4.1.2 Defining and Inheriting Subclass Methods

Our `Manager` class has a new instance variable to store the bonus and a new method to set it:

```
public class Manager extends Employee {
    private double bonus;
    ...
    public void setBonus(double bonus) {
        this.bonus = bonus;
    }
}
```

When you have a `Manager` object, you can of course apply the `setBonus` method, as well as nonprivate methods from the `Employee` class. Those methods are *inherited*.

```
Manager boss = new Manager(...);
boss.setBonus(10000); // Defined in subclass
boss.raiseSalary(5); // Inherited from superclass
```

4.1.3 Method Overriding

Sometimes, a superclass method needs to be modified in a subclass. For example, suppose that the `getSalary` method is expected to report the total salary of an employee. Then the inherited method is not sufficient for the `Manager` class. Instead, you need to *override* the method so that it returns the sum of the base salary and the bonus.

```
public class Manager extends Employee {
    ...
    public double getSalary() { // Overrides superclass method
        return super.getSalary() + bonus;
    }
}
```

This method invokes the superclass method, which retrieves the base salary, and adds the bonus. Note that a subclass method cannot access the private instance variables of the superclass directly. That is why the `Manager.getSalary` method calls the public `Employee.getSalary` method. The `super` keyword is used for invoking a superclass method.

NOTE: Unlike `this`, `super` is not a reference to an object, but a directive to bypass dynamic method lookup (see Section 4.1.5, "Superclass Assignments," page 139) and invoke a specific method instead.

It is not required to call the superclass method when overriding a method, but it is common to do so.

When you override a method, you must be careful to match the parameter types exactly. For example, suppose that the `Employee` class has a method

```
public boolean worksFor(Employee supervisor)
```

If you override this method in the `Manager` class, you cannot change the parameter type, even though surely no manager would report to a mere employee. Suppose you defined a method

```
public class Manager extends Employee {
    ...
    public boolean worksFor(Manager supervisor) {
        ...
    }
}
```

This is simply a new method, and now `Manager` has two separate `worksFor` methods. You can protect yourself against this type of error by tagging methods that are intended to override superclass methods with the `@Override` annotation:

```
@Override public boolean worksFor(Employee supervisor)
```

If you made a mistake and are defining a new method, the compiler reports an error.

You can change the return type to a subtype when overriding a method. (In technical terms, *covariant return types* are permitted.) For example, if the `Employee` class has a method

```
public Employee getSupervisor()
```

then the `Manager` class can override it with the method

```
@Override public Manager getSupervisor()
```

CAUTION: When you override a method, the subclass method must be *at least as accessible* as the superclass method. In particular, if the superclass method is public, then the subclass method must also be declared public. It is a common error to accidentally omit the `public` modifier for the subclass method. The compiler then complains about the weaker access privilege.

4.1.4 Subclass Construction

Let us supply a constructor for the `Manager` class. Since the `Manager` constructor cannot access the private instance variables of the `Employee` class, it must initialize them through a superclass constructor.

```
public Manager(String name, double salary) {
    super(name, salary);
    bonus = 0;
}
```

Here, the keyword `super` indicates a call to the constructor of the `Employee` superclass with `name` and `salary` as arguments. The superclass constructor call must be the *first statement* in the constructor for the subclass.

If the subclass does not explicitly call any superclass constructor, the superclass must have a no-argument constructor which is implicitly called.

4.1.5 Superclass Assignments

It is legal to assign an object from a subclass to a variable whose type is a superclass, for example:

```
Manager boss = new Manager(...);
Employee empl = boss; // OK to assign to superclass variable
```

Now consider what happens when one invokes a method on the superclass variable.

```
double salary = empl.getSalary();
```

Even though the type of `empl` is `Employee`, the `Manager.getSalary` method is invoked. When invoking a method, the virtual machine looks at the actual class of the object and locates its version of the method. This process is called *dynamic method lookup*.

Why would you want to assign a `Manager` object to an `Employee` variable? It allows you to write code that works for *all* employees, be they managers or janitors or instances of another `Employee` subclass.

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...); // OK to assign to superclass variable
staff[2] = new Janitor(...);
...
double sum = 0;
for (Employee empl : staff)
    sum += empl.getSalary();
```

Thanks to dynamic method lookup, the call `empl.getSalary()` invokes the `getSalary` method belonging to the object to which `empl` refers, which may be `Employee.getSalary`, `Manager.getSalary`, and so on.

CAUTION: In Java, superclass assignment also works for arrays: You can assign a `Manager[]` array to an `Employee[]` variable. (The technical term is that Java arrays are *covariant*.) This is convenient, but it is also *unsound*—that is, a possible cause of type errors. Consider this example:

```
Manager[] bosses = new Manager[10];
Employee[] empls = bosses; // Legal in Java
empls[0] = new Employee(...); // Runtime error
```

The compiler accepts the last statement since it is generally legal to store an `Employee` in an `Employee[]` array. However, here `empls` and `bosses` reference the same `Manager[]` array, which cannot hold a lowly `Employee`. This mistake is only caught at runtime, when the virtual machine throws an `ArrayStoreException`.

4.1.6 Casts

In the preceding section, you saw how a variable `empl` of type `Employee` can refer to objects whose class is `Employee`, `Manager`, or another subclass of `Employee`. That is useful for code that deals with objects from multiple classes. There is just one drawback. You can only invoke methods that belong to the superclass. Consider, for example,

```
Employee empl = new Manager(...);
empl.setBonus(10000); // Compile-time error
```

Even though this call could succeed at runtime, it is a compile-time error. The compiler checks that you only invoke methods that exist for the receiver type. Here, `empl` is of type `Employee` and that class has no `setBonus` method.

As with interfaces, you can use the `instanceof` operator and a cast to turn a superclass reference to a subclass.

```
if (empl instanceof Manager) {
    Manager mgr = (Manager) empl;
    mgr.setBonus(10000);
}
```

4.1.7 Final Methods and Classes

When you declare a method as `final`, no subclass can override it.

```
public class Employee {
    ...
    public final String getName() {
        return name;
    }
}
```

A good example of a `final` method in the Java API is the `getClass` method of the `Object` class that you will see in Section 4.4.1, "The Class `Class`" (page 159). It does not seem wise to allow objects to lie about the class to which they belong, so this method can never be changed.

Some programmers believe that the `final` keyword is good for efficiency. This may have been true in the early days of Java, but it no longer is. Modern virtual machines will speculatively "inline" simple methods, such as the `getName` method above, even if they are not declared `final`. In the rare case when a subclass is loaded that overrides such a method, the inlining is undone.

Some programmers believe that most methods of a class should be declared `final`, and only methods specifically designed to be overridden should not be. Others find this too draconian since it prevents even harmless overriding, for example for logging or debugging purposes.

Occasionally, you may want to prevent someone from forming a subclass from one of your classes. Use the `final` modifier in the class definition to indicate this. For example, here is how to prevent others from subclassing the `Executive` class:

```
public final class Executive extends Manager {
    ...
}
```

There is a good number of final classes in the Java API, such as `String`, `LocalTime`, and `URL`.

4.1.8 Abstract Methods and Classes

A class can define a method without an implementation, forcing subclasses to implement it. Such a method, and the class containing it, are called *abstract*

and must be tagged with the `abstract` modifier. This is commonly done for very general classes, for example:

```
public abstract class Person {
    private String name;

    public Person(String name) { this.name = name; }
    public final String getName() { return name; }

    public abstract int getId();
}
```

Any class extending `Person` must either supply an implementation of the `getId` method or be itself declared as `abstract`.

Note that an abstract class can have nonabstract methods, such as the `getName` method in the preceding example.

NOTE: Unlike an interface, an abstract class can have instance variables and constructors.

It is not possible to construct an instance of an abstract class. For example, the call

```
Person p = new Person("Fred"); // Error
```

would be a compile-time error.

However, you can have a variable whose type is an abstract class, provided it contains a reference to an object of a concrete subclass. Suppose the class `Student` is declared as

```
public class Student extends Person {
    private int id;

    public Student(String name, int id) { super(name); this.id = id; }
    public int getId() { return id; }
}
```

Then you can construct a `Student` and assign it to a `Person` variable.

```
Person p = new Student("Fred", 1729); // OK, a concrete subclass
```

4.1.9 Protected Access

There are times when you want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access an instance variable of a superclass. For that, declare a class feature as `protected`.

For example, suppose the superclass `Employee` declares the instance variable `salary` as `protected` instead of `private`.

```
package com.horstmann.employees;

public class Employee {
    protected double salary;
    ...
}
```

All classes in the same package as `Employee` can access this field. Now consider a subclass from a different package:

```
package com.horstmann.managers;

import com.horstmann.employees.Employee;

public class Manager extends Employee {
    ...
    public double getSalary() {
        return salary + bonus; // OK to access protected salary variable
    }
}
```

The `Manager` class methods can peek inside the `salary` variable of `Manager` objects only, not of other `Employee` objects. This restriction is made so that you can't abuse the protected mechanism by forming subclasses just to gain access to protected features.

Of course, protected fields should be used with caution. Once provided, you cannot take them away without breaking classes that are using them.

Protected methods and constructors are more common. For example, the `clone` method of the `Object` class is protected since it is somewhat tricky to use (see Section 4.2.4, "Cloning Objects," page 151).

CAUTION: In Java, `protected` grants package-level access, and it only protects access from other packages.

4.1.10 Anonymous Subclasses

Just as you can have an anonymous class that implements an interface, you can have an anonymous class that extends a superclass. This can be handy for debugging:

```
ArrayList<String> names = new ArrayList<String>(100) {
    public void add(int index, String element) {
        super.add(index, element);
        System.out.printf("Adding %s at %d\n", element, index);
    }
};
```

The arguments in the parentheses following the superclass name are passed to the superclass constructor. Here, we construct an anonymous subclass of `ArrayList<String>` that overrides the `add` method. The instance is constructed with an initial capacity of 100.

A trick called *double brace initialization* uses the inner class syntax in a rather bizarre way. Suppose you want to construct an array list and pass it to a method:

```
ArrayList<String> friends = new ArrayList<>();
friends.add("Harry");
friends.add("Sally");
invite(friends);
```

If you won't ever need the array list again, it would be nice to make it anonymous. But then, how can you add the elements? Here is how:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Sally"); }});
```

Note the double braces. The outer braces make an anonymous subclass of `ArrayList<String>`. The inner braces are an initialization block (see Chapter 2).

I am not recommending that you use this trick outside of Java trivia contests. There are several drawbacks beyond the confusing syntax. It is inefficient, and the constructed object can behave strangely in equality tests, depending on how the `equals` method is implemented.

4.1.11 Inheritance and Default Methods

Suppose a class extends a class and implements an interface, both of which happen to have a method of the same name.

```
public interface Named {
    default String getName() { return ""; }
}

public class Person {
    ...
    public String getName() { return name; }
}

public class Student extends Person implements Named {
    ...
}
```

In this situation, the superclass implementation always wins over the interface implementation. There is no need for the subclass to resolve the conflict.

In contrast, as you saw in Chapter 3, you must resolve a conflict when the same default method is inherited from two interfaces.

The "classes win" rule ensures compatibility with Java 7. If you add default methods to an interface, it has no effect on code that worked before there were default methods.

4.1.12 Method Expressions with super

Recall from Chapter 3 that a method expression can have the form *object*::*instanceMethod*. It is also valid to use `super` instead of an object reference. The method expression

```
super::instanceMethod
```

uses `this` as the target and invokes the superclass version of the given method. Here is an artificial example that shows the mechanics:

```
public class Worker {
    public void work() {
        for (int i = 0; i < 100; i++) System.out.println("Working");
    }
}

public class ConcurrentWorker extends Worker {
    public void work() {
        Thread t = new Thread(super::work);
        t.start();
    }
}
```

The thread is constructed with a `Runnable` whose `run` method calls the `work` method of the superclass.

4.2 Object: The Cosmic Superclass

Every class in Java directly or indirectly extends the class `Object`. When a class has no explicit superclass, it implicitly extends `Object`. For example,

```
public class Employee { ... }
```

is equivalent to

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

The `Object` class defines methods that are applicable to any Java object (see Table 4-1). We will examine several of these methods in detail in the following sections.

NOTE: Arrays are classes. Therefore, it is legal to convert an array, even a primitive type array, to a reference of type `Object`.

Table 4–1 The Methods of the `java.lang.Object` Class

Method	Description
`String toString()`	Yields a string representation of this object, by default the name of the class and the hash code.
`boolean equals(Object other)`	Returns `true` if this object should be considered equal to `other`, `false` if `other` is `null` or different from `other`. By default, two objects are equal if they are identical. Instead of `obj.equals(other)`, consider the null-safe alternative `Objects.equals(obj, other)`.
`int hashCode()`	Yields a hash code for this object. Equal objects must have the same hash code. Unless overridden, the hash code is assigned in some way by the virtual machine.
`Class<?> getClass()`	Yields the `Class` object describing the class to which this object belongs.
`protected Object clone()`	Makes a copy of this object. By default, the copy is shallow.
`protected void finalize()`	This method is called when this object is reclaimed by the garbage collector. Don't override it.
`wait, notify, notifyAll`	See Chapter 10.

4.2.1 The `toString` Method

An important method in the `Object` class is the `toString` method that returns a string description of an object. For example, the `toString` method of the `Point` class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Many `toString` methods follow this format: the name of the class, followed by the instance variables enclosed in square brackets. Here is such an implementation of the `toString` method of the `Employee` class:

```
public String toString() {
    return getClass().getName() + "[name=" + name
        + ",salary=" + salary + "]";
}
```

By calling `getClass().getName()` instead of hardwiring the string `"Employee"`, this method does the right thing for subclasses as well.

In a subclass, call `super.toString()` and add the instance variables of the subclass, in a separate pair of brackets:

```
public class Manager extends Employee {
    ...
    public String toString() {
        return super.toString() + "[bonus=" + bonus + "]";
    }
}
```

Whenever an object is concatenated with a string, the Java compiler automatically invokes the `toString` method on the object. For example:

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
    // Concatenates with p.toString()
```

TIP: Instead of writing `x.toString()`, you can write `"" + x`. This expression even works if `x` is `null` or a primitive type value.

The `Object` class defines the `toString` method to print the class name and the hash code (see Section 4.2.3, "The `hashCode` Method," page 150). For example, the call

```
System.out.println(System.out)
```

produces an output that looks like `java.io.PrintStream@2f6684` since the implementor of the `PrintStream` class didn't bother to override the `toString` method.

CAUTION: Arrays inherit the `toString` method from `Object`, with the added twist that the array type is printed in an archaic format. For example, if you have the array

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

then `primes.toString()` yields a string such as `"[I@1a46e30"`. The prefix `[I` denotes an array of integers.

The remedy is to call `Arrays.toString(primes)` instead, which yields the string `"[2, 3, 5, 7, 11, 13]"`. To correctly print multidimensional arrays (that is, arrays of arrays), use `Arrays.deepToString`.

4.2.2 The equals Method

The `equals` method tests whether one object is considered equal to another. The `equals` method, as implemented in the `Object` class, determines whether two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal. For quite a few classes, nothing else is required. For example, it makes little sense to compare two `Scanner` objects for equality.

Override the `equals` method only for state-based equality testing, in which two objects are considered equal when they have the same contents. For example, the `String` class overrides `equals` to check whether two strings consist of the same characters.

CAUTION: Whenever you override the `equals` method, you *must* provide a compatible `hashCode` method as well—see Section 4.2.3, "The `hashCode` Method" (page 150).

Suppose we want to consider two objects of a class `Item` equal if their descriptions and prices match. Here is how you can implement the `equals` method:

```
public class Item {
    private String description;
    private double price;
    ...
    public boolean equals(Object otherObject) {
        // A quick test to see if the objects are identical
        if (this == otherObject) return true;

        // Must return false if the parameter is null
        if (otherObject == null) return false;
        // Check that otherObject is an Item
        if (getClass() != otherObject.getClass()) return false;
        // Test whether the instance variables have identical values
        Item other = (Item) otherObject;
        return Objects.equals(description, other.description)
            && price == other.price;
    }

    public int hashCode() { ... } // See Section 4.2.3
}
```

There are a number of routine steps that you need to go through in an `equals` method:

1. It is common for equal objects to be identical, and that test is very inexpensive.

2. Every `equals` method is required to return `false` when comparing against `null`.
3. Since the `equals` method overrides `Object.equals`, its parameter is of type `Object`, and you need to cast it to the actual type so you can look at its instance variables. Before doing that, make a type check, either with the `getClass` method or with the `instanceof` operator.
4. Finally, compare the instance variables. Use `==` for primitive types. However, for `double` values, if you are concerned about $\pm\infty$ or NaN, use `Double.equals`. For objects, use `Objects.equals`, a null-safe version of the `equals` method. The call `Objects.equals(x, y)` returns `false` if `x` is `null`, whereas `x.equals(y)` would throw an exception.

TIP: If you have instance variables that are arrays, use the static `Arrays.equals` method to check that the arrays have equal length and the corresponding array elements are equal.

When you define the `equals` method for a subclass, first call `equals` on the superclass. If that test doesn't pass, the objects can't be equal. If the instance variables of the superclass are equal, then you are ready to compare the instance variables of the subclass.

```
public class DiscountedItem extends Item {
    private double discount;
    ...
    public boolean equals(Object otherObject) {
        if (!super.equals(otherObject)) return false;
        DiscountedItem other = (DiscountedItem) otherObject;
        return discount == other.discount;
    }

    public int hashCode() { ... }
}
```

Note that the `getClass` test in the superclass fails if `otherObject` is not a `DiscountedItem`.

How should the `equals` method behave when comparing values that belong to different classes? This has been an area of some controversy. In the preceding example, the `equals` method returns `false` if the classes don't match exactly. But many programmers use an `instanceof` test instead:

```
if (!(otherObject instanceof Item)) return false;
```

This leaves open the possibility that `otherObject` can belong to a subclass. For example, you can compare an `Item` with a `DiscountedItem`.

However, that kind of comparison doesn't usually work. One of the requirements of the `equals` method is that it is *symmetric*: For non-null `x` and `y`, the calls `x.equals(y)` and `y.equals(x)` need to return the same value.

Now suppose `x` is an `Item` and `y` a `DiscountedItem`. Since `x.equals(y)` doesn't consider discounts, neither can `y.equals(x)`.

NOTE: The Java API contains over 150 implementations of `equals` methods, with a mixture of `instanceof` tests, calling `getClass`, catching a `ClassCastException`, or doing nothing at all. Check out the documentation of the `java.sql.Timestamp` class, where the implementors note with some embarrassment that the `Timestamp` class inherits from `java.util.Date`, whose `equals` method uses an `instanceof` test, and it is therefore impossible to override `equals` to be both symmetric and accurate.

There is one situation where the `instanceof` test makes sense: if the notion of equality is fixed in the superclass and never varies in a subclass. For example, this is the case if we compare employees by ID. In that case, make an `instanceof` test and declare the `equals` method as `final`.

```
public class Employee {
    private int id;
    ...
    public final boolean equals(Object otherObject) {
        if (this == otherObject) return true;
        if (!(otherObject instanceof Employee)) return false;
        Employee other = (Employee) otherObject;
        return id == other.id;
    }

    public int hashCode() { ... }
}
```

4.2.3 The hashCode Method

A *hash code* is an integer that is derived from an object. Hash codes should be scrambled—if `x` and `y` are two unequal objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different. For example, `"Mary".hashCode()` is 2390779, and `"Myra".hashCode()` is 2413819.

The `String` class uses the following algorithm to compute the hash code:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

The `hashCode` and `equals` methods must be *compatible*: If `x.equals(y)`, then it must be the case that `x.hashCode() == y.hashCode()`. As you can see, this is the case for the `String` class since strings with equal characters produce the same hash code.

The `Object.hashCode` method derives the hash code in some implementation-dependent way. It can be derived from the object's memory location, or a number (sequential or pseudorandom) that is cached with the object, or a combination of both. Since `Object.equals` tests for identical objects, the only thing that matters is that identical objects have the same hash code.

If you redefine the `equals` method, you will also need to redefine the `hashCode` method to be compatible with `equals`. If you don't, and users of your class insert objects into a hash set or hash map, they might get lost!

In your `hashCode` method, simply combine the hash codes of the instance variables. For example, here is a `hashCode` method for the `Item` class:

```
class Item {
    ...
    public int hashCode() {
        return Objects.hash(description, price);
    }
}
```

The `Objects.hash` varargs method computes the hash codes of its arguments and combines them. The method is null-safe.

If your class has instance variables that are arrays, compute their hash codes first with the static `Arrays.hashCode` method, which computes a hash code composed of the hash codes of the array elements. Pass the result to `Objects.hash`.

CAUTION: In an interface, you can never make a default method that redefines one of the methods in the `Object` class. In particular, an interface can't define a default method for `toString`, `equals`, or `hashCode`. As a consequence of the "classes win" rule (see Section 4.1.11, "Inheritance and Default Methods," page 144), such a method could never win against `Object.toString`, `Object.equals`, or `Object.hashCode`.

4.2.4 Cloning Objects

You have just seen the "big three" methods of the `Object` class that are commonly overridden: `toString`, `equals`, and `hashCode`. In this section, you will learn how to override the `clone` method. As you will see, this is complex, and it is also rarely necessary. Don't override `clone` unless you have a good reason to do so. Less than five percent of the classes in the standard Java library implement `clone`.

The purpose of the `clone` method is to make a "clone" of an object—a distinct object with the same state of the original. If you mutate one of the objects, the other stays unchanged.

```
Employee cloneOfFred = fred.clone();
cloneOfFred.raiseSalary(10); // fred unchanged
```

The `clone` method is declared as `protected` in the `Object` class, so you must override it if you want users of your class to clone instances.

The `Object.clone` method makes a *shallow copy*. It simply copies all instance variables from the original to the cloned object. That is fine if the variables are primitive or immutable. But if they aren't, then the original and the clone share mutable state, which can be a problem.

Consider a class for email messages that has a list of recipients.

```
public final class Message {
    private String sender;
    private ArrayList<String> recipients;
    private String text;
    ...
    public void addRecipient(String recipient) { ... };
}
```

If you make a shallow copy of a `Message` object, both the original and the clone share the `recipients` list (see Figure 4-1):

```
Message specialOffer = ...;
Message cloneOfSpecialOffer = specialOffer.clone();
```

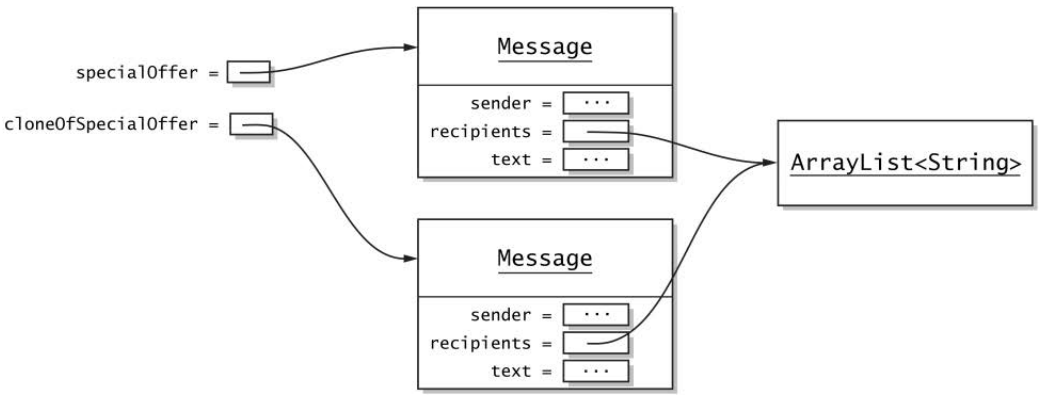

Figure 4-1 A shallow copy of an object

If either object changes the recipient list, the change is reflected in the other. Therefore, the `Message` class needs to override the `clone` method to make a *deep copy*.

It may also be that cloning is impossible or not worth the trouble. For example, it would be very challenging to clone a `Scanner` object.

In general, when you implement a class, you need to decide whether

1. You do not want to provide a `clone` method, or
2. The inherited `clone` method is acceptable, or
3. The `clone` method should make a deep copy.

For the first option, simply do nothing. Your class will inherit the `clone` method, but no user of your class will be able to call it since it is `protected`.

To choose the second option, your class must implement the `Cloneable` interface. This is an interface without any methods, called a *tagging* or *marker* interface. (Recall that the `clone` method is defined in the `Object` class.) The `Object.clone` method checks that this interface is implemented before making a shallow copy, and throws a `CloneNotSupportedException` otherwise.

You will also want to raise the scope of `clone` from `protected` to `public`, and change the return type.

Finally, you need to deal with the `CloneNotSupportedException`. This is a *checked* exception, and as you will see in Chapter 5, you must either declare or catch it. If your class is `final`, you can catch it. Otherwise, declare the exception since it is possible that a subclass might again want to throw it.

```
public class Employee implements Cloneable {
    ...
    public Employee clone() throws CloneNotSupportedException {
        return (Employee) super.clone();
    }
}
```

The cast `(Employee)` is necessary since the return type of `Object.clone` is `Object`.

The third option for implementing the `clone` method, in which a class needs to make a deep copy, is the most complex case. You don't need to use the `Object.clone` method at all. Here is a simple implementation of `Message.clone`:

```
public Message clone() {
    Message cloned = new Message(sender, text);
    cloned.recipients = new ArrayList<>(recipients);
    return cloned;
}
```

Alternatively, you can call `clone` on the superclass and the mutable instance variables.

The `ArrayList` class implements the `clone` method, yielding a shallow copy. That is, the original and cloned list share the element references. That is fine in our case since the elements are strings. If not, we would have had to clone each element as well.

However, for historical reasons, the `ArrayList.clone` method has return type `Object`. You need to use a cast.

```
cloned.recipients = (ArrayList<String>) recipients.clone(); // Warning
```

Unhappily, as you will see in Chapter 6, that cast cannot be fully checked at runtime, and you will get a warning. You can suppress the warning with an annotation, but that annotation can only be attached to a declaration (see Chapter 12). Here is the complete method implementation:

```
public Message clone() {
    try {
        Message cloned = (Message) super.clone();
        @SuppressWarnings("unchecked") ArrayList<String> clonedRecipients
            = (ArrayList<String>) recipients.clone();
        cloned.recipients = clonedRecipients;
        return cloned;
    } catch (CloneNotSupportedException ex) {
        return null; // Can't happen
    }
}
```

In this case, the `CloneNotSupportedException` cannot happen since the `Message` class is `Cloneable` and `final`, and `ArrayList.clone` does not throw the exception.

NOTE: Arrays have a public `clone` method whose return type is the same as the type of the array. For example, if `recipients` had been an array, not an array list, you could have cloned it as

```
cloned.recipients = recipients.clone(); // No cast required
```

4.3 Enumerations

You saw in Chapter 1 how to define enumerated types. Here is a typical example, defining a type with exactly four instances:

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

In the following sections, you will see how to work with enumerations.

4.3.1 Methods of Enumerations

Since each instance of an enumerated type is unique, you never need to use the `equals` method for values of enumerated types. Simply use `==` to compare them. (You can, if you like, call `equals` which makes the `==` test.)

You also don't need to provide a `toString` method. It is automatically provided to yield the name of the enumerated object—in our example, `"SMALL"`, `"MEDIUM"`, and so on.

The converse of `toString` is the static `valueOf` method that is synthesized for each enumerated type. For example, the statement

```
Size notMySize = Size.valueOf("SMALL");
```

sets `notMySize` to `Size.SMALL`. The `valueOf` method throws an exception if there is no instance with the given name.

Each enumerated type has a static `values` method that returns an array of all instances of the enumeration, in the order in which they were declared. The call

```
Size[] allValues = Size.values();
```

returns the array with elements `Size.SMALL`, `Size.MEDIUM`, and so on.

TIP: Use this method to traverse all instances of an enumerated type in an enhanced `for` loop:

```
for (Size s : Size.values()) { System.out.println(s); }
```

The `ordinal` method yields the position of an instance in the `enum` declaration, counting from zero. For example, `Size.MEDIUM.ordinal()` returns `1`. Of course, you need to be careful with this method. The values shift if new constants are inserted.

Every enumerated type `E` automatically implements `Comparable<E>`, allowing comparisons only against its own objects. The comparison is by ordinal values.

NOTE: Technically, an enumerated type `E` extends the class `Enum<E>` from which it inherits the `compareTo` method as well as the other methods described in this section. Table 4-2 shows the methods of the `Enum` class.

Table 4-2 Methods of the `java.lang.Enum<E>` Class

Method	Description
`String toString()` `String name()`	The name of this instance, as provided in the `enum` declaration. The `name` method is `final`.
`int ordinal()`	The position of this instance in the `enum` declaration.
`int compareTo(Enum<E> other)`	Compares this instance against `other` by ordinal value.
`static T valueOf(Class<T> type, String name)`	Returns the instance for a given name. Consider using the synthesized `valueOf` or `values` method of the enumeration type instead.
`Class<E> getDeclaringClass()`	Gets the class in which this instance was defined. (This differs from `getClass()` if the instance has a body.)
`int hashCode()` `protected void finalize()`	These methods call the corresponding `Object` methods and are `final`.
`protected Object clone()`	Throws a `CloneNotSupportedException`.

4.3.2 Constructors, Methods, and Fields

You can, if you like, add constructors, methods, and fields to an enumerated type. Here is an example:

```
public enum Size {
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    Size(String abbreviation) {
        this.abbreviation = abbreviation;
    }

    public String getAbbreviation() { return abbreviation; }
}
```

Each instance of the enumeration is guaranteed to be constructed exactly once.

NOTE: The constructor of an enumeration is always private. You can omit the `private` modifier, as in the preceding example. It is a syntax error to declare an enum constructor as `public` or `protected`.

4.3.3 Bodies of Instances

You can add methods to each individual `enum` instance, but they have to override methods defined in the enumeration. For example, to implement a calculator, you might do this:

```
public enum Operation {
    ADD {
        public int eval(int arg1, int arg2) { return arg1 + arg2; }
    },
    SUBTRACT {
        public int eval(int arg1, int arg2) { return arg1 - arg2; }
    },
    MULTIPLY {
        public int eval(int arg1, int arg2) { return arg1 * arg2; }
    },
    DIVIDE {
        public int eval(int arg1, int arg2) { return arg1 / arg2; }
    };

    public abstract int eval(int arg1, int arg2);
}
```

In the loop of a calculator program, one would set a variable to one of these values, depending on user input, and then invoke `eval`:

```
Operation op = ...;
int result = op.eval(first, second);
```

NOTE: Technically, each of these constants belongs to an anonymous subclass of `Operation`. Anything that you could place into an anonymous subclass body you can also add into the body of a member.

4.3.4 Static Members

It is legal for an enumeration to have static members. However, you have to be careful with construction order. The enumerated constants are constructed *before* the static members, so you cannot refer to any static members in an enumeration constructor. For example, the following would be illegal:

```
public enum Modifier {
    PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL, ABSTRACT;
    private static int maskBit = 1;
    private int mask;
    Modifier() {
        mask = maskBit; // Error—cannot access static variable in constructor
        maskBit *= 2;
    }
    ...
}
```

The remedy is to do the initialization in a static initializer:

```
public enum Modifier {
    PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL, ABSTRACT;
    private int mask;

    static {
        int maskBit = 1;
        for (Modifier m : Modifier.values()) {
            m.mask = maskBit;
            maskBit *= 2;
        }
    }
    ...
}
```

Once the constants have been constructed, static variable initializations and static initializers run in the usual top-to-bottom fashion.

NOTE: Enumerated types can be nested inside classes. Such nested enumerations are implicitly static nested classes—that is, their methods cannot reference instance variables of the enclosing class.

4.3.5 Switching on an Enumeration

You can use enumeration constants in a `switch` statement.

```
enum Operation { ADD, SUBTRACT, MULTIPLY, DIVIDE };

public static int eval(Operation op, int arg1, int arg2) {
    int result = 0;
    switch (op) {
        case ADD: result = arg1 + arg2; break;
        case SUBTRACT: result = arg1 - arg2; break;
        case MULTIPLY: result = arg1 * arg2; break;
        case DIVIDE: result = arg1 / arg2; break;
    }
    return result;
}
```

You use `ADD`, not `Operation.ADD`, inside the `switch` statement—the type is inferred from the type of the expression on which the switch is computed.

NOTE: According to the language specification, compilers are encouraged to give a warning if a switch on an enumeration is not exhaustive—that is, if there aren't cases for all constants and no `default` clause. The Oracle compiler does not produce such a warning.

TIP: If you want to refer to the instances of an enumeration by their simple name outside a `switch`, use a static import declaration. For example, with the declaration

```
import static com.horstmann.corejava.Size.*;
```

you can use `SMALL` instead of `Size.SMALL`.

4.4 Runtime Type Information and Resources

In Java, you can find out at runtime to which class a given object belongs. This is sometimes useful, for example in the implementation of the `equals` and `toString` methods. Moreover, you can find out how the class was loaded and load its associated data, called *resources*.

4.4.1 The Class Class

Suppose you have a variable of type `Object`, filled with some object reference, and you want to know more about the object, such as to which class it belongs.

The `getClass` method yields an object of a class called, not surprisingly, `Class`.

```
Object obj = ...;
Class<?> cl = obj.getClass();
```

NOTE: See Chapter 6 for an explanation of the `<?>` suffix. For now, just ignore it. But don't omit it. If you do, not only does your IDE give you an unsightly warning, but you also turn off useful type checks in expressions involving the variable.

Once you have a `Class` object, you can find out the class name:

```
System.out.println("This object is an instance of " + cl.getName());
```

Alternatively, you can get a `Class` object by using the static `Class.forName` method:

```
String className = "java.util.Scanner";
Class<?> cl = Class.forName(className);
   // An object describing the java.util.Scanner class
```

CAUTION: The `Class.forName` method, as well as many other methods used with reflection, throws checked exceptions when something goes wrong (for example, when there is no class with the given name). For now, tag the calling method with `throws ReflectiveOperationException`. You will see in Chapter 5 how to handle the exception.

The `Class.forName` method is intended for constructing `Class` objects for classes that may not be known at compile time. If you know in advance which class you want, use a *class literal* instead:

```
Class<?> cl = java.util.Scanner.class;
```

The `.class` suffix can be used to get information about other types as well:

```
Class<?> cl2 = String[].class; // Describes the array type String[]
Class<?> cl3 = Runnable.class; // Describes the Runnable interface
Class<?> cl4 = int.class; // Describes the int type
Class<?> cl5 = void.class; // Describes the void type
```

Arrays are classes in Java, but interfaces, primitive types, and `void` are not. The name `Class` is a bit unfortunate—`Type` would have been more accurate.

CAUTION: The `getName` method returns strange names for array types:

- `String[].class.getName()` returns `"[Ljava.lang.String;"`
- `int[].class.getName()` returns `"[I"`

This notation has been used since archaic times in the virtual machine. Use `getCanonicalName` instead to get names such as `"java.lang.String[]"` and `"int[]"`. You need to use the archaic notation with the `Class.forName` method if you want to generate `Class` objects for arrays.

The virtual machine manages a unique `Class` object for each type. Therefore, you can use the `==` operator to compare class objects. For example:

```
if (other.getClass() == Employee.class) ...
```

You have already seen this use of class objects in Section 4.2.2, "The `equals` Method" (page 148).

In the following sections, you will see what you can do with `Class` objects. See Table 4-3 for a summary of useful methods.

Table 4–3 Useful Methods of the `java.lang.Class<T>` Class

Method	Description
`static Class<?> forName(String className)`	Gets the `Class` object describing `className`.

`String getCanonicalName()`
`String getSimpleName()`
`String getTypeName()`
`String getName()`
`String toString()`
`String toGenericString()`
	Gets the name of this class, with various idiosyncrasies for arrays, inner classes, generic classes, and modifiers (see Exercise 8).

`Class<? super T> getSuperclass()`
`Class<?>[] getInterfaces()`
`Package getPackage()`
`int getModifiers()`
	Gets the superclass, the implemented interfaces, package, and modifiers of this class. Table 4-4 shows how to analyze the value returned by `getModifiers`.

`boolean isPrimitive()`
`boolean isArray()`
`boolean isEnum()`
`boolean isAnnotation()`
`boolean isMemberClass()`
`boolean isLocalClass()`
`boolean isAnonymousClass()`
`boolean isSynthetic()`
	Tests whether the represented type is primitive or `void`, an array, an enumeration, an annotation (see Chapter 12), nested in another class, local to a method or constructor, anonymous, or synthetic (see Section 4.5.7).

`Class<?> getComponentType()`
`Class<?> getDeclaringClass()`
`Class<?> getEnclosingClass()`
`Constructor getEnclosingConstructor()`
`Method getEnclosingMethod()`
	Gets the component type of an array, the class declaring a nested class, the class and constructor or method in which a local class is declared.

`boolean isAssignableFrom(Class<?> cls)` `boolean isInstance(Object obj)`	Tests whether the type `cls` or the class of `obj` is a subtype of this type.
`String getPackageName()`	Gets the fully qualified package name of this class or, if it is not a top-level class, its enclosing class.
`ClassLoader getClassLoader()`	Gets the class loader that loaded this class (see Section 4.4.3).
`InputStream getResourceAsStream(String path)` `URL getResource(String path)`	Loads the requested resource from the same location from which this class was loaded.

(Continues)

Table 4–3 Useful Methods of the `java.lang.Class<T>` Class *(Continued)*

Method	Description

`Field[] getFields()`
`Method[] getMethods()`
`Field getField(String name)`
`Method getMethod(String name, Class<?>... parameterTypes)`
	Gets all public fields or methods, or the specified field or method, from this class or a superclass.

`Field[] getDeclaredFields()`
`Method[] getDeclaredMethods()`
`Field getDeclaredField(String name)`
`Method getDeclaredMethod(String name, Class<?>... parameterTypes)`
	Gets all fields or methods, or the specified field or method, from this class.

`Constructor[] getConstructors()`
`Constructor[] getDeclaredConstructors()`
`Constructor getConstructor(Class<?>... parameterTypes)`
`Constructor getDeclaredConstructor(Class<?>... parameterTypes)`
	Gets all public constructors, or all constructors, or the specified public constructor, or the specified constructor, for this class.

Table 4–4 Methods of the `java.lang.reflect.Modifier` Class

Method	Description
`static String toString(int modifiers)`	Returns a string with the modifiers that correspond to the bits set in `modifiers`.
`static boolean is(Abstract|Interface|Native|Private|Protected|Public|Static|Strict|Synchronized|Volatile)(int modifiers)`	Tests the bit in the `modifiers` argument that corresponds to the modifier in the method name.

4.4.2 Loading Resources

One useful service of the `Class` class is to locate resources that your program may need, such as configuration files or images. If you place a resource into the same directory as the class file, you can open an input stream to the file like this:

```
InputStream stream = MyClass.class.getResourceAsStream("config.txt");
Scanner in = new Scanner(stream);
```

NOTE: Some legacy methods such as `Applet.getAudioClip` and the `javax.swing.ImageIcon` constructor read data from a `URL` object. In that case, you can use the `getResource` method which returns a URL to the resource.

Resources can have subdirectories which can be relative or absolute. For example, `MyClass.class.getResourceAsStream("/config/menus.txt")` locates `config/menus.txt` in the directory that contains the root of the package to which `MyClass` belongs.

If you package classes into JAR files, zip up the resources together with the class files, and they will be located as well.

4.4.3 Class Loaders

Virtual machine instructions are stored in class files. Each class file contains the instructions for a single class or interface. A class file can be located on a file system, in a JAR file, at a remote location, or it can even be dynamically constructed in memory. A *class loader* is responsible for loading the bytes and turning them into a class or interface in the virtual machine.

The virtual machine loads class files on demand, starting with the class whose `main` method is to be invoked. That class will depend on other classes, such as `java.lang.System` and `java.util.Scanner`, which will be loaded together with the classes that they depend on.

When executing a Java program, at least three class loaders are involved.

The *bootstrap class loader* loads the most fundamental Java library classes. It is a part of the virtual machine.

The *platform class loader* loads other library classes. Unlike the classes loaded with the bootstrap class loader, platform class permissions can be configured with a security policy.

The *system class loader* loads the application classes. It locates classes in the directories and JAR files on the class path and module path.

CAUTION: In previous releases of the Oracle JDK, the platform and system class loaders were instances of the `URLClassLoader` class. This is no longer the case. Some programmers used the `getURLs` method of the `URLClassLoader` to find the class path. Use `System.getProperty("java.class.path")` instead.

You can load classes from a directory or JAR file that is not already on the class path, by creating your own `URLClassLoader` instance. This is commonly done to load plugins.

```
URL[] urls = {
    new URL("file:///path/to/directory/"),
    new URL("file:///path/to/jarfile.jar")
};
String className = "com.mycompany.plugins.Entry";
try (URLClassLoader loader = new URLClassLoader(urls)) {
    Class<?> cl = Class.forName(className, true, loader);
    // Now construct an instance of cl—see Section 4.5.4
    ...
}
```

CAUTION: The second parameter in the call `Class.forName(className, true, loader)` ensures that the static initialization of the class happens after loading. You definitely want that to happen.

Do not use the `ClassLoader.loadClass` method. It does not run the static initializers.

NOTE: The `URLClassLoader` loads classes from the file system. If you want to load a class from somewhere else, you need to write your own class loader. The only method you need to implement is `findClass`, like this:

```
public class MyClassLoader extends ClassLoader {
    ...
    @Override public Class<?> findClass(String name)
            throws ClassNotFoundException {
        byte[] bytes = the bytes of the class file
        return defineClass(name, bytes, 0, bytes.length);
    }
}
```

See Chapter 14 for an example in which classes are compiled into memory and then loaded.

4.4.4 The Context Class Loader

Most of the time you don't have to worry about the class loading process. Classes are transparently loaded as they are required by other classes. However, if a method loads classes dynamically, and that method is called from

a class that itself was loaded with another class loader, then problems can arise. Here is a specific example.

1. You provide a utility class that is loaded by the system class loader, and it has a method

```
public class Util {
    Object createInstance(String className) {
        Class<?> cl = Class.forName(className);
        ...
    }
    ...
}
```

2. You load a plugin with another class loader that reads classes from a plugin JAR.
3. The plugin calls `Util.createInstance("com.mycompany.plugins.MyClass")` to instantiate a class in the plugin JAR.

The author of the plugin expects that class to be loaded. However, `Util.createInstance` uses its own class loader to execute `Class.forName`, and that class loader won't look into the plugin JAR. This phenomenon is called *classloader inversion*.

One remedy is to pass the class loader to the utility method and then to the `forName` method.

```
public class Util {
    public Object createInstance(String className, ClassLoader loader) {
        Class<?> cl = Class.forName(className, true, loader);
        ...
    }
    ...
}
```

Another strategy is to use the *context class loader* of the current thread. The main thread's context class loader is the system class loader. When a new thread is created, its context class loader is set to the creating thread's context class loader. Thus, if you don't do anything, all threads will have their context class loaders set to the system class loader. However, you can set any class loader by calling

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

The utility method can then retrieve the context class loader:

```
public class Util {
    public Object createInstance(String className) {
        Thread t = Thread.currentThread();
        ClassLoader loader = t.getContextClassLoader();
        Class<?> cl = Class.forName(className, true, loader);
        ...
    }
    ...
}
```

When invoking a method of a plugin class, the application should set the context class loader to the plugin class loader. Afterwards, it should restore the previous setting.

TIP: If you write a method that loads a class by name, don't simply use the class loader of the method's class. It is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader.

4.4.5 Service Loaders

Certain services need to be configurable when a program is assembled or deployed. One way to do this is to make different implementations of a service available, and have the program choose the most appropriate one among them. The `ServiceLoader` class makes it easy to load service implementations that conform to a common interface.

Define an interface (or, if you prefer, a superclass) with the methods that each instance of the service should provide. For example, suppose your service provides encryption.

```
package com.corejava.crypt;

public interface Cipher {
    byte[] encrypt(byte[] source, byte[] key);
    byte[] decrypt(byte[] source, byte[] key);
    int strength();
}
```

The service provider supplies one or more classes that implement this service, for example

```
package com.corejava.crypt.impl;

public class CaesarCipher implements Cipher {
    public byte[] encrypt(byte[] source, byte[] key) {
        byte[] result = new byte[source.length];
        for (int i = 0; i < source.length; i++)
            result[i] = (byte)(source[i] + key[0]);
        return result;
    }
    public byte[] decrypt(byte[] source, byte[] key) {
        return encrypt(source, new byte[] { (byte) -key[0] });
    }
    public int strength() { return 1; }
}
```

The implementing classes can be in any package, not necessarily the same package as the service interface. Each of them must have a no-argument constructor.

Now add the names of the provider classes to a UTF-8 encoded text file in a `META-INF/services` directory that a class loader can find. In our example, the file `META-INF/services/com.corejava.crypt.Cipher` would contain the line

```
com.corejava.crypt.impl.CaesarCipher
```

With this preparation done, the program initializes a service loader as follows:

```
public static ServiceLoader<Cipher> cipherLoader = ServiceLoader.load(Cipher.class);
```

This should be done just once in the program.

The `iterator` method of the service loader provides an iterator through all provided implementations of the service. (See Chapter 7 for more information about iterators.) It is easiest to use an enhanced `for` loop to traverse them. In the loop, pick an appropriate object to carry out the service.

```
public static Cipher getCipher(int minStrength) {
    for (Cipher cipher : cipherLoader) // Implicitly calls iterator
        if (cipher.strength() >= minStrength) return cipher;
    return null;
}
```

Alternatively, you can use streams (see Chapter 8) to locate the desired service. The `stream` method yields a stream of `ServiceLoader.Provider` instances. That interface has methods `type` and `get` for getting the provider class and the provider instance. If you select a provider by type, then you just call `type` and no service instances are unnecessarily instantiated. In our example, we need to get the providers since we filter the stream for ciphers that have the required strength:

```
public static Optional<Cipher> getCipher2(int minStrength) {
    return cipherLoader.stream()
        .map(ServiceLoader.Provider::get)
        .filter(c -> c.strength() >= minStrength)
        .findFirst();
}
```

If you are willing to take any implementation, simply call `findFirst`:

```
Optional<Cipher> cipher = cipherLoader.findFirst();
```

The `Optional` class is explained in Chapter 8.

4.5 Reflection

Reflection allows a program to inspect the contents of objects at runtime and to invoke arbitrary methods on them. This capability is useful for implementing tools such as object-relational mappers or GUI builders.

Since reflection is of interest mainly to tool builders, application programmers can safely skip this section and return to it as needed.

4.5.1 Enumerating Class Members

The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class. All three classes have a method called `getName` that returns the name of the member. The `Field` class has a method `getType` that returns an object, again of type `Class`, that describes the field type. The `Method` and `Constructor` classes have methods to report the types of the parameters, and the `Method` class also reports the return type.

All three of these classes also have a method called `getModifiers` that returns an integer, with various bits turned on and off, that describes the modifiers used (such as `public` or `static`). You can use static methods such as `Modifier.isPublic` and `Modifier.isStatic` to analyze the integer that `getModifiers` returns. The `Modifier.toString` returns a string of all modifiers.

The `getFields`, `getMethods`, and `getConstructors` methods of the `Class` class return arrays of the *public* fields, methods, and constructors that the class supports; this includes public inherited members. The `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` methods return arrays consisting of all fields, methods, and constructors that are declared in the class. This includes private, package, and protected members, but not members of superclasses.

The `getParameters` method of the `Executable` class, the common superclass of `Method` and `Constructor`, returns an array of `Parameter` objects describing the method parameters.

NOTE: The names of the parameters are only available at runtime if the class has been compiled with the `-parameters` flag.

For example, here is how you can print all methods of a class:

```
Class<?> cl = Class.forName(className);
while (cl != null) {
    for (Method m : cl.getDeclaredMethods()) {
        System.out.println(
            Modifier.toString(m.getModifiers()) + " " +
            m.getReturnType().getCanonicalName() + " " +
            m.getName() +
            Arrays.toString(m.getParameters()));
    }
    cl = cl.getSuperclass();
}
```

What is remarkable about this code is that it can analyze any class that the Java virtual machine can load—not just the classes that were available when the program was compiled.

CAUTION: As you will see in Chapter 15, the Java platform module system imposes significant restrictions on reflective access. By default, only classes in the same module can be analyzed through reflection. If you don't declare modules, all your classes belong to a single module, and they can all be accessed through reflection. However, the Java library classes belong to different modules, and reflective access to their non-public members is restricted.

4.5.2 Inspecting Objects

As you saw in the preceding section, you can get `Field` objects that describe the types and names of an object's fields. These `Field` objects can do more: They can access field values in objects that have the given field.

For example, here is how to enumerate the contents of all fields of an object:

```
Object obj = ...;
for (Field f : obj.getClass().getDeclaredFields()) {
    f.setAccessible(true);
    Object value = f.get(obj);
    System.out.println(f.getName() + ":" + value);
}
```

The key is the `get` method that reads the field value. If the field value is a primitive type value, a wrapper object is returned; in that case you can also call one of the methods `getInt`, `getDouble`, and so on.

NOTE: You must make private `Field` and `Method` objects "accessible" before you can use them. Calling `setAccessible(true)` "unlocks" the field or method for reflection. However, the module system or a security manager can block the request and protect objects from being accessed in this way. In that case, the `setAccessible` method throws an `InaccessibleObjectException` or `SecurityException`. Alternatively, you can call the `trySetAccessible` method which simply returns `false` if the field or method is not accessible.

CAUTION: As you will see in Chapter 15, the Java platform packages are contained in modules and their classes are protected from reflection. For example, if you call

```
Field f = String.class.getDeclaredField("value");
f.setAccessible(true);
```

an `InaccessibleObjectException` is thrown.

Once a field is accessible, you can also set it. This code will give a raise to `obj`, no matter to which class it belongs, provided that it has an accessible `salary` field of type `double` or `Double`.

```
Field f = obj.getClass().getDeclaredField("salary");
f.setAccessible(true);
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

4.5.3 Invoking Methods

Just like a `Field` object can be used to read and write fields of an object, a `Method` object can invoke the given method on an object.

```
Method m = ...;
Object result = m.invoke(obj, arg1, arg2, ...);
```

If the method is static, supply `null` for the initial argument.

To obtain a method, you can search through the array returned by `getMethods` or `getDeclaredMethods` that you saw in Section 4.5.1, "Enumerating Class Members" (page 168). Or you can call `getMethod` and supply the parameter types. For example, to get the `setName(String)` method on a `Person` object:

```
Person p = ...;
Method m = p.getClass().getMethod("setName", String.class);
m.invoke(obj, "********");
```

CAUTION: Even though `clone` is a public method of all array types, it is not reported by `getMethod` when invoked on a `Class` object describing an array.

4.5.4 Constructing Objects

To construct an object, first find the `Constructor` object and then call its `newInstance` method. For example, suppose you know that a class has a public constructor whose parameter is an `int`. Then you can construct a new instance like this:

```
Constructor constr = cl.getConstructor(int.class);
Object obj = constr.newInstance(42);
```

CAUTION: The `Class` class has a `newInstance` method to construct an object of the given class with the no-argument constructor. That method is now deprecated because it has a curious flaw. If the no-argument constructor throws a checked exception, the `newInstance` method rethrows it *even though it isn't declared*, thereby completely defeating the compile-time checking of checked exceptions. Instead, you should call `cl.getConstructor().newInstance()`. Then any exception is wrapped inside an `InvocationTargetException`.

Table 4-5 summarizes the most important methods for working with `Field`, `Method`, and `Constructor` objects.

Table 4–5 Useful Classes and Methods in the `java.lang.reflect` Package

Class	Method	Notes
`AccessibleObject`	`void setAccessible(boolean flag)` `static void setAccessible(AccessibleObject[] array, boolean flag)`	`AccessibleObject` is a superclass of `Field`, `Method`, and `Constructor`. The methods set the accessibility of this member, or the given members.

`Field`	`String getName()`
	`int getModifiers()`
	`Object get(Object obj)`
	p `get`*P*`(Object obj)`
	`void set(Object obj, Object newValue)`
	`void set`*P*`(Object obj,` *p* `newValue)`
		There is a `get` and `set` method for each primitive type *p*.

`Method`	`Object invoke(Object obj, Object... args)`	Invokes the method described by this object, passing the given arguments and returning the value that the method returns. For static methods, pass `null` for `obj`. Primitive type arguments and return values are wrapped.
`Constructor`	`Object newInstance(Object... args)`	Invokes the constructor described by this object, passing the given arguments and returning the constructed object.

`Executable`	`String getName()`
	`int getModifiers()`
	`Parameters[] getParameters()`
		`Executable` is the superclass of `Method` and `Constructor`.

`Parameter`	`boolean isNamePresent()`
	`String getName()`
	`Class<?> getType()`
		The `getName` method gets the name or a synthesized name such as `arg0` if the name is not present.

4.5.5 JavaBeans

Many object-oriented programming languages support *properties*, mapping the expression *object*.*propertyName* to a call of a getter or setter method, depending

on whether the property is read or written. Java does not have this syntax, but it has a convention in which properties correspond to getter/setter pairs. A *JavaBean* is a class with a no-argument constructor, getter/setter pairs, and any number of other methods.

The getters and setters must follow the specific pattern

```
public Type getProperty()
public void setProperty(Type newValue)
```

It is possible to have read-only and write-only properties by omitting the setter or getter.

The name of the property is the *decapitalized* form of the suffix after `get`/`set`. For example, a `getSalary`/`setSalary` pair gives rise to a property named `salary`. However, if the first *two* letters of the suffix are uppercase, then it is taken verbatim. For example, `getURL` yields a read-only property named `URL`.

NOTE: For Boolean properties, you may use either `get`*Property* or `is`*Property* for the getter, and the latter is preferred.

JavaBeans have their origin in GUI builders, and the JavaBeans specification has arcane rules that deal with property editors, property change events, and custom property discovery. These features are rarely used nowadays.

It is a good idea to use the standard classes for JavaBeans support whenever you need to work with arbitrary properties. Given a class, obtain a `BeanInfo` object like this:

```
Class<?> cl = ...;
BeanInfo info = Introspector.getBeanInfo(cl);
PropertyDescriptor[] props = info.getPropertyDescriptors();
```

For a given `PropertyDescriptor`, call `getName` and `getPropertyType` to get the name and type of the property. The `getReadMethod` and `getWriteMethod` yield `Method` objects for the getter and setter.

Unfortunately, there is no method to get the descriptor for a given property name, so you'll have to traverse the array of descriptors:

```
String propertyName = ...;
Object propertyValue = null;
for (PropertyDescriptor prop : props) {
    if (prop.getName().equals(propertyName))
        propertyValue = prop.getReadMethod().invoke(obj);
}
```

4.5.6 Working with Arrays

The `isArray` method checks whether a given `Class` object represents an array. If so, the `getComponentType` method yields the `Class` describing the type of the array elements. For further analysis, or to create arrays dynamically, use the `Array` class in the `java.lang.reflect` package. Table 4-6 shows its methods.

Table 4–6 Methods of the `java.lang.reflect.Array` Class

Method	Description

`static Object get(Object array, int index)`
`static` *p* `get`*P*`(Object array, int index)`
`static void set(Object array, int index, Object newValue)`
`static void set`*P*`(Object array, int index,` *p* `newValue)`
	Gets or sets an element of the array at the given index, where *p* is a primitive type.

`static int getLength(Object array)`	Gets the length of the given array.
`static Object newInstance(Class<?> componentType, int length)` `static Object newInstance(Class<?> componentType, int[] lengths)`	Returns a new array of the given component type with the given dimensions.

As an exercise, let us implement the `copyOf` method in the `Arrays` class. Recall how this method can be used to grow an array that has become full.

```
Person[] friends = new Person[100];
...
// Array is full
friends = Arrays.copyOf(friends, 2 * friends.length);
```

How can one write such a generic method? Here is a first attempt:

```
public static Object[] badCopyOf(Object[] array, int newLength) { // Not useful
    Object[] newArray = new Object[newLength];
    for (int i = 0; i < Math.min(array.length, newLength); i++)
        newArray[i] = array[i];
    return newArray;
}
```

However, there is a problem with actually using the resulting array. The type of array that this method returns is `Object[]`. An array of objects cannot be cast to a `Person[]` array. The point is, as we mentioned earlier, that a Java array remembers the type of its elements—that is, the type used in the `new` expression that created it. It is legal to cast a `Person[]` array temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into a `Person[]` array.

In order to make a new array of the same type as the original array, you need the `newInstance` method of the `Array` class. Supply the component type and the desired length:

```
public static Object goodCopyOf(Object array, int newLength) {
    Class<?> cl = array.getClass();
    if (!cl.isArray()) return null;
    Class<?> componentType = cl.getComponentType();
    int length = Array.getLength(array);
    Object newArray = Array.newInstance(componentType, newLength);
    for (int i = 0; i < Math.min(length, newLength); i++)
        Array.set(newArray, i, Array.get(array,  i));
    return newArray;
}
```

Note that this `copyOf` method can be used to grow arrays of any type, not just arrays of objects.

```
int[] primes = { 2, 3, 5, 7, 11 };
primes = (int[]) goodCopyOf(primes, 10);
```

The parameter type of `goodCopyOf` is `Object`, not `Object[]`. An `int[]` is an `Object` but not an array of objects.

4.5.7 Proxies

The `Proxy` class can create, at runtime, new classes that implement a given interface or set of interfaces. Such proxies are only necessary when you don't yet know at compile time which interfaces you need to implement.

A proxy class has all methods required by the specified interfaces, and all methods defined in the `Object` class (`toString`, `equals`, and so on). However, since you cannot define new code for these methods at runtime, you supply an *invocation handler*, an object of a class that implements the `InvocationHandler` interface. That interface has a single method:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Whenever a method is called on the proxy object, the `invoke` method of the invocation handler gets called, with the `Method` object and parameters of the original call. The invocation handler must then figure out how to handle the call. There are many possible actions an invocation handler might take, such as routing calls to remote servers or tracing calls for debugging purposes.

To create a proxy object, use the `newProxyInstance` method of the `Proxy` class. The method has three parameters:

- A class loader (see Section 4.4.3, "Class Loaders," page 163), or `null` to use the default class loader

- An array of `Class` objects, one for each interface to be implemented
- The invocation handler

To show the mechanics of proxies, here is an example where an array is populated with proxies for `Integer` objects, forwarding calls to the original objects after printing trace messages:

```
Object[] values = new Object[1000];

for (int i = 0; i < values.length; i++) {
    Object value = new Integer(i);
    values[i] = Proxy.newProxyInstance(
        null,
        value.getClass().getInterfaces(),
        // Lambda expression for invocation handler
        (Object proxy, Method m, Object[] margs) -> {
            System.out.println(value + "." + m.getName() + Arrays.toString(margs));
            return m.invoke(value, margs);
        });
}
```

When calling

```
Arrays.binarySearch(values, new Integer(500));
```

the following output is produced:

```
499.compareTo[500]
749.compareTo[500]
624.compareTo[500]
561.compareTo[500]
530.compareTo[500]
514.compareTo[500]
506.compareTo[500]
502.compareTo[500]
500.compareTo[500]
```

You can see how the binary search algorithm homes in on the key by cutting the search interval in half in every step.

The point is that the `compareTo` method is invoked through the proxy, even though this was not explicitly mentioned in the code. All methods in any interfaces implemented by `Integer` are proxied.

CAUTION: When the invocation handler is called with a method call that has no parameters, the argument array is `null`, not an `Object[]` array of length `0`. That is utterly reprehensible and not something you should do in your own code.

Exercises

1. Define a class `Point` with a constructor `public Point(double x, double y)` and accessor methods `getX`, `getY`. Define a subclass `LabeledPoint` with a constructor `public LabeledPoint(String label, double x, double y)` and an accessor method `getLabel`.
2. Define `toString`, `equals`, and `hashCode` methods for the classes of the preceding exercise.
3. Make the instance variables `x` and `y` of the `Point` class in Exercise 1 `protected`. Show that the `LabeledPoint` class can access these variables only in `LabeledPoint` instances.
4. Define an abstract class `Shape` with an instance variable of class `Point`, a constructor, a concrete method `public void moveBy(double dx, double dy)` that moves the point by the given amount, and an abstract method `public Point getCenter()`. Provide concrete subclasses `Circle`, `Rectangle`, `Line` with constructors `public Circle(Point center, double radius)`, `public Rectangle(Point topLeft, double width, double height)`, and `public Line(Point from, Point to)`.
5. Define `clone` methods for the classes of the preceding exercise.
6. Suppose that in Section 4.2.2, "The `equals` Method" (page 148), the `Item.equals` method uses an `instanceof` test. Implement `DiscountedItem.equals` so that it compares only the superclass if `otherObject` is an `Item`, but also includes the discount if it is a `DiscountedItem`. Show that this method preserves symmetry but fails to be *transitive*—that is, find a combination of items and discounted items so that `x.equals(y)` and `y.equals(z)`, but not `x.equals(z)`.
7. Define an enumeration type for the eight combinations of primary colors `BLACK`, `RED`, `BLUE`, `GREEN`, `CYAN`, `MAGENTA`, `YELLOW`, `WHITE` with methods `getRed`, `getGreen`, and `getBlue`.
8. The `Class` class has six methods that yield a string representation of the type represented by the `Class` object. How do they differ when applied to arrays, generic types, inner classes, and primitive types?
9. Write a "universal" `toString` method that uses reflection to yield a string with all instance variables of an object. Extra credit if you can handle cyclic references.
10. Use the `MethodPrinter` program in Section 4.5.1,"Enumerating Class Members" (page 168) to enumerate all methods of the `int[]` class. Extra credit if you can identify the one method (discussed in this chapter) that is wrongly described.

11. Write the "Hello, World" program, using reflection to look up the `out` field of `java.lang.System` and using `invoke` to call the `println` method.
12. Measure the performance difference between a regular method call and a method call via reflection.
13. Write a method that prints a table of values for any `Method` representing a static method with a parameter of type `double` or `Double`. Besides the `Method` object, accept a lower bound, upper bound, and step size. Demonstrate your method by printing tables for `Math.sqrt` and `Double.toHexString`. Repeat, using a `DoubleFunction<Object>` instead of a `Method` (see Section 3.6.2, "Choosing a Functional Interface," page 120). Contrast the safety, efficiency, and convenience of both approaches.