

# Think Java

How to Think Like a Computer Scientist

2nd Edition, Version 7.1.0

# Chapter 11

## Designing Classes

Whenever you create a new class, you are creating a new object type with the same name. So way back in Section 1.3, when we created the class `Hello`, we also created an object type named `Hello`.

We didn't declare any variables with type `Hello`, and we didn't use `new` to create `Hello` objects. And it wouldn't have done much good if we had—but we could have!

In this chapter, you will learn to design classes that represent *useful* objects. Here are the main ideas:

- Again, defining a **class** creates a new object type with the same name.
- A class definition is a template for objects: it specifies what attributes the objects have and what methods can operate on them.
- Every object belongs to an object type; that is, it is an **instance** of a class.
- The `new` operator **instantiates** objects; that is, it creates new instances of a class.

Think of a class as a blueprint for a house: you can use the same blueprint to build any number of houses.

## 11.1 The Time Class

A common reason to define a new class is to encapsulate related data in an object that can be treated as a single unit. That way, we can use objects as parameters and return values, rather than passing and returning multiple values. You have already seen two types that encapsulate data in this way: `Point` and `Rectangle`.

Another example, which we will implement ourselves, is `Time`, which represents a time of day. The data encapsulated in a `Time` object includes an hour, a minute, and a number of seconds. Because every `Time` object contains these values, we define attributes to hold them.

Attributes are also called **instance variables**, because each instance has its own variables (as opposed to “class variables”, coming up in Section 12.3).

The first step is to decide what type each variable should be. It seems clear that `hour` and `minute` should be integers. Just to keep things interesting, let’s make `second` a double.

Instance variables are declared at the beginning of the class definition, outside any method. By itself, this code fragment is a legal class definition:

```
public class Time {  
    private int hour;  
    private int minute;  
    private double second;  
}
```

The `Time` class is `public`, which means that it can be used in other classes. But the instance variables are `private`, which means they can be accessed only from inside the `Time` class. If you try to read or write them from another class, you will get a compiler error.

Private instance variables help keep classes isolated from each other, so that changes in one class won’t require changes in other classes. It also simplifies what other programmers need to know to use your classes. This kind of isolation is called **information hiding**.

## 11.2 Constructors

After declaring instance variables, the next step is to define a **constructor**, which is a special method that initializes the object. The syntax for constructors is similar to that of other methods, except for the following:

- The name of the constructor is the same as the name of the class.
- Constructors have no return type (and no return value).
- The keyword `static` is omitted.

Here is an example constructor for the `Time` class:

```
public Time() {  
    this.hour = 0;  
    this.minute = 0;  
    this.second = 0.0;  
}
```

This constructor does not take any arguments. Each line initializes an instance variable to 0 (which is midnight for a `Time` object).

The name `this` is a keyword that refers to the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods. But you do not declare `this`, and you can't make an assignment to it.

A common error when writing constructors is to put a `return` statement at the end. Like `void` methods, constructors do not return values.

To create a `Time` object, you must use the `new` operator:

```
public static void main(String[] args) {  
    Time time = new Time();  
}
```

When you use `new`, Java creates the object and invokes your constructor to initialize the instance variables. When the constructor is done, `new` returns a

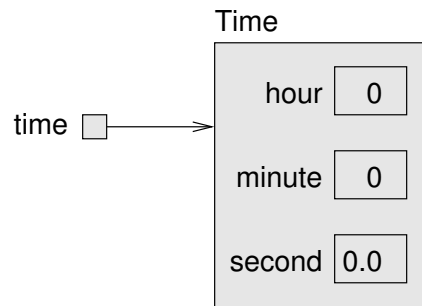


Figure 11.1: Memory diagram of a `Time` object.

reference to the new object. In this example, the reference gets assigned to the variable `time`, which has type `Time`. Figure 11.1 shows the result.

Beginners sometimes make the mistake of using `new` in the constructor:

```
public Time() {  
    new Time();           // StackOverflowError  
    this.hour = 0;  
    this.minute = 0;  
    this.second = 0.0;  
}
```

Doing so causes an infinite recursion, since `new` invokes the *same* constructor, which uses `new` again, which invokes the constructor again, and so on.

## 11.3 Value Constructors

Like other methods, constructors can be overloaded, which means you can provide multiple constructors with different parameters. Java knows which constructor to invoke by matching the arguments you provide with the parameters of the constructor.

It is common to provide both a “default constructor” that takes no arguments, like the previous one, and a “value constructor”, like this one:

```
public Time(int hour, int minute, double second) {  
    this.hour = hour;  
    this.minute = minute;  
    this.second = second;  
}
```

To invoke this constructor, you have to provide arguments to the `new` operator. The following example creates a `Time` object that represents a fraction of a second before noon:

```
Time time = new Time(11, 59, 59.9);
```

Overloading constructors provides the flexibility to create an object first and then fill in the attributes, or collect all the information before creating the object itself.

Once you get the hang of it, writing constructors gets boring. You can write them quickly just by looking at the list of instance variables. In fact, some IDEs can generate them for you.

Here is the complete class definition so far:

```
public class Time {  
    private int hour;  
    private int minute;  
    private double second;  
  
    public Time() {  
        this.hour = 0;  
        this.minute = 0;  
        this.second = 0.0;  
    }  
  
    public Time(int hour, int minute, double second) {  
        this.hour = hour;  
        this.minute = minute;  
        this.second = second;  
    }  
}
```

Notice how the second constructor declares the parameters `hour`, `minute`, and `second`. Java allows you to declare parameters (and local variables) with the same names as instance variables. They don't have to use the same names, but it's common practice.

The right side of `this.hour = hour;` refers to the parameter `hour`, since it was declared most recently. This situation is called **shadowing**, because the parameter “hides” the instance variable with the same name.

Java provides the keyword `this` so you can access instance variables, regardless of shadowing. As a result, this constructor copies the values from the parameters to the instance variables.

## 11.4 Getters and Setters

Recall that the instance variables of `Time` are `private`. We can access them from within the `Time` class, but if we try to read or write them from another class, the compiler reports an error.

A class that uses objects defined in another class is called a **client**. For example, here is a new class called `TimeClient`:

```
public class TimeClient {  
  
    public static void main(String[] args) {  
        Time time = new Time(11, 59, 59.9);  
        System.out.println(time.hour);    // compiler error  
    }  
}
```

If you compile this code, you get an error message like “hour has private access in Time”. There are three ways to solve this problem:

- Make the instance variables public.
- Provide methods to access the instance variables.
- Decide that it's not a problem and refuse to let other classes access the instance variables.

The first choice is appealing because it's simple. But here is the problem: when class *A* accesses the instance variables of class *B* directly, *A* becomes dependent on *B*. If anything in *B* changes later, it is likely that *A* will have to change too.

But if *A* uses only methods to interact with *B*, *A* and *B* are less dependent, which means that we can make changes in *B* without affecting *A* (as long as we don't change the method parameters). So we generally avoid making instance variables public.

The second option is to provide methods that access the instance variables. For example, we might want the instance variables to be “read only”; that is, code in other classes should be able to read them but not write them. We can do that by providing one method for each instance variable:

```
public int getHour() {  
    return this.hour;  
}  
  
public int getMinute() {  
    return this.minute;  
}  
  
public double getSecond() {  
    return this.second;  
}
```

Methods like these are formally called “accessors”, but more commonly referred to as **getters**. By convention, the method that gets a variable named *something* is called *getSomething*.

We can fix the compiler error in `TimeClient` by using the getter:

```
System.out.println(time.getHour());
```

If we decide that `TimeClient` should also be able to modify the instance variables of `Time`, we can provide methods to do that too:



```
public void setHour(int hour) {  
    this.hour = hour;  
}  
  
public void setMinute(int minute) {  
    this.minute = minute;  
}  
  
public void setSecond(double second) {  
    this.second = second;  
}
```

These methods are formally called “mutators”, but more commonly known as **setters**. The naming convention is similar; the method that sets something is usually called `setSomething`.

Writing getters and setters can get boring, but many IDEs can generate them for you based on the instance variables.

## 11.5 Displaying Objects

To display `Time` objects, we can write a method to display the hour, minute, and second. Using `printTime` in Section 4.4 as a starting point, we could write the following:

```
public static void printTime(Time t) {  
    System.out.print(t.hour);  
    System.out.print(":");  
    System.out.print(t.minute);  
    System.out.print(":");  
    System.out.println(t.second);  
}
```

The output of this method, given the `time` object from the first example, would be `11:59:59.9`. We can use `printf` to make the code more concise:

```
public static void printTime(Time t) {  
    System.out.printf("%02d:%02d:%04.1f\n",  
        t.hour, t.minute, t.second);  
}
```

As a reminder, you need to use `%d` with integers, and `%f` with floating-point numbers. The `02` option means “total width 2, with leading zeros if necessary”, and the `04.1` option means “total width 4, one digit after the decimal point, leading zeros if necessary”. The output is the same: `11:59:59.9`.

There’s nothing wrong with a method like `printTime`, but it is not consistent with object-oriented style. A more idiomatic solution is to provide a special method called `toString`.

## 11.6 The toString Method

Every object has a method called `toString` that returns a string representation of the object. When you display an object using `print` or `println`, Java invokes the object’s `toString` method.

By default, it simply displays the type of the object and its address in hexadecimal. So, say you create a `Time` object and display it with `println`:

```
public static void main(String[] args) {  
    Time time = new Time(11, 59, 59.9);  
    System.out.println(time);  
}
```

The output looks something like this:

```
Time@80cc7c0
```

This address can be useful for debugging, if you want to keep track of individual objects.

But you can **override** this behavior by providing your own `toString` method. For example, here is a `toString` method for `Time`:

```
public String toString() {  
    return String.format("%02d:%02d:%04.1f\n",  
        this.hour, this.minute, this.second);  
}
```

The definition does not have the keyword `static`, because it is not a static method. It is an **instance method**, so called because when you invoke it, you

invoke it on an instance of the class. Instance methods are sometimes called “non-static”; you might see this term in an error message.

The body of the method is similar to `printTime` in the previous section, with two changes:

- Inside the method, we use `this` to refer to the current instance; that is, the object the method is invoked on.
- Instead of `printf`, it uses `String.format`, which returns a formatted `String` rather than displaying it.

Now you can call `toString` directly:

```
Time time = new Time(11, 59, 59.9);  
String s = time.toString();
```

The value of `s` is the string `"11:59:59.9"`. You can also invoke `toString` indirectly by invoking `print` or `println`:

```
System.out.println(time);
```

This code displays the string `"11:59:59.9"`. Either way, when you use `this` inside `toString`, it refers to the same object as `time`.

## 11.7 The equals Method

We have seen two ways to check whether values are equal: the `==` operator and the `equals` method. With objects, you can use either one, but they are not the same:

- The `==` operator checks whether two references are **identical**; that is, whether they refer to the same object.
- The `equals` method checks whether two objects are **equivalent**; that is, whether they have the same values.

The definition of *identity* is always the same, so the `==` operator always does the same thing. But the definition of *equivalence* is different for different objects, so objects can define their own `equals` methods.

Consider the following variables and the corresponding memory diagram in Figure 11.2:

```
Time time1 = new Time(9, 30, 0.0);
Time time2 = time1;
Time time3 = new Time(9, 30, 0.0);
```

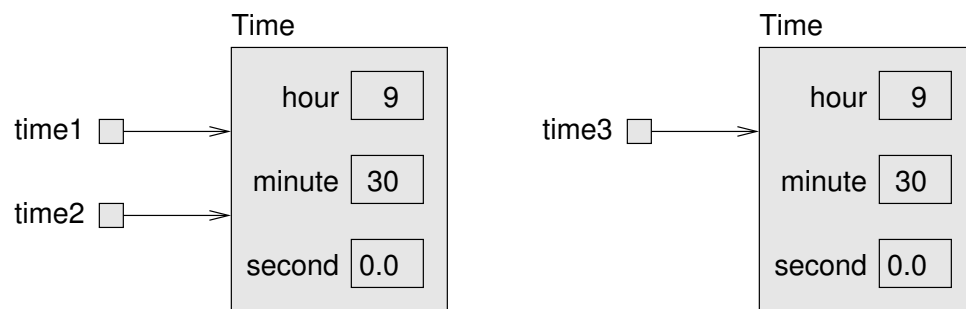


Figure 11.2: Memory diagram of three `Time` variables.

The assignment operator copies references, so `time1` and `time2` refer to the same object. Because they are identical, `time1 == time2` is true. But `time1` and `time3` refer to two different objects. Because they are not identical, `time1 == time3` is false.

By default, the `equals` method does the same thing as `==`. For `Time` objects, that's probably not what we want. For example, `time1` and `time3` represent the same time of day, so we should consider them equivalent.

We can provide an `equals` method that implements this idea:

```
public boolean equals(Time that) {
    final double DELTA = 0.001;
    return this.hour == that.hour
        && this.minute == that.minute
        && Math.abs(this.second - that.second) < DELTA;
}
```

`equals` is an instance method, so it doesn't have the keyword `static`. It uses `this` to refer to the current object, and `that` to refer to the other. `that` is *not*

a keyword, so we could have given this parameter a different name. But using `that` makes the code nicely readable.

We can invoke `equals` like this:

```
time1.equals(time3);
```

Inside the `equals` method, `this` refers to the same object as `time1`, and `that` refers to the same object as `time3`. Since their instance variables are “equal”, the result is `true`.

Because `hour` and `minute` are integers, we compare them with `==`. But `second` is a floating-point number. Because of rounding errors, it is not good to compare floating-point numbers with `==` (see Section 2.7). Instead, we check whether the difference is smaller than a threshold, `DELTA`.

Many objects have a similar notion of equivalence; that is, two objects are considered equal if their instance variables are equal. But other definitions are possible.

## 11.8 Adding Times

Suppose you are going to a movie that starts at 18:50 (that is, 6:50 PM), and the running time is 2 hours, 16 minutes. What time does the movie end? We’ll use `Time` objects to figure it out:

```
Time startTime = new Time(18, 50, 0.0);  
Time runningTime = new Time(2, 16, 0.0);
```

Here are two ways we could “add” the `Time` objects:

- Write a static method that takes two `Time` objects as parameters.
- Write an instance method that gets invoked on one object and takes the other as a parameter.

To demonstrate the difference, we’ll do both. Here is the static method:

```
public static Time add(Time t1, Time t2) {  
    Time sum = new Time();  
    sum.hour = t1.hour + t2.hour;  
    sum.minute = t1.minute + t2.minute;  
    sum.second = t1.second + t2.second;  
    return sum;  
}
```

And here's how we would invoke it:

```
Time endTime = Time.add(startTime, runningTime);
```

Here's what it looks like as an instance method:

```
public Time add(Time t2) {  
    Time sum = new Time();  
    sum.hour = this.hour + t2.hour;  
    sum.minute = this.minute + t2.minute;  
    sum.second = this.second + t2.second;  
    return sum;  
}
```

And here's how we would invoke it:

```
Time endTime = startTime.add(runningTime);
```

Notice the differences:

- The static method has the keyword `static`; the instance method does not.
- The static method has two parameters, `t1` and `t2`. The instance method has one explicit parameter, `t1`, and the implicit parameter, `this`.
- We invoked the static method with the `Time` class; we invoked the instance method with the `startTime` object.

That's all there is to it. Static methods and instance methods do the same thing, and you can convert from one to the other with just a few changes.

However, there's a problem with both of these methods; they are not correct. The result from either method is 20:66, which is not a valid time.

If `second` exceeds 59, we have to carry into the minutes column, and if `minute` exceeds 59, we have to carry into `hour`.

Here is a better version of the instance method, `add`:

```
public Time add(Time t2) {
    Time sum = new Time();
    sum.hour = this.hour + t2.hour;
    sum.minute = this.minute + t2.minute;
    sum.second = this.second + t2.second;

    if (sum.second >= 60.0) {
        sum.second -= 60.0;
        sum.minute += 1;
    }
    if (sum.minute >= 60) {
        sum.minute -= 60;
        sum.hour += 1;
    }
    if (sum.hour >= 24) {
        sum.hour -= 24;
    }
    return sum;
}
```

If `hour` exceeds 23, we subtract 24 hours, but there's no `days` attribute to carry into.

## 11.9 Vocabulary

**class:** In Chapter 1, we defined a class as a collection of related methods.

Now you know that a class is also a template for a new type of object.

**instance:** A member of a class. Every object is an instance of a class.

**instantiate:** Create a new instance of a class in the computer's memory.

**instance variable:** An attribute of an object; a non-static variable defined at the class level.

**information hiding:** The practice of making instance variables `private` to limit dependencies between classes.

**constructor:** A special method that initializes the instance variables of a newly constructed object.

**shadowing:** Occurs when a local variable or parameter has the same name as an attribute.

**client:** A class that uses objects defined in another class.

**getter:** A method that returns the value of an instance variable.

**setter:** A method that assigns a value to an instance variable.

**override:** To replace a default implementation of a method, such as `toString`.

**instance method:** A non-static method that has access to `this` and the instance variables.

**identical:** References to the same object (at the same location in memory).

**equivalent:** Objects that are equal in value, as defined by the `equals` method.

## 11.10 Exercises

The code for this chapter is in the *ch11* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise 11.1** The implementation of `increment` in this chapter is not very efficient. Can you rewrite it so it doesn't use any loops?

*Hint:* Remember the remainder operator—it works with floating-point too.

**Exercise 11.2** In the board game Scrabble, each tile contains a letter, which is used to spell words in rows and columns, and a score, which is used to determine the value of words. The point of this exercise is to practice the mechanical part of creating a new class definition:



1. Write a definition for a class named `Tile` that represents Scrabble tiles. The instance variables should include a character named `letter` and an integer named `value`.
2. Write a constructor that takes parameters named `letter` and `value`, and initializes the instance variables.
3. Write a method named `printTile` that takes a `Tile` object as a parameter and displays the instance variables in a reader-friendly format.
4. Write a `main` method that creates a `Tile` object with the letter Z and the value 10, and then uses `printTile` to display the state of the object.
5. Implement the `toString` and `equals` methods for a `Tile`.
6. Create getters and setters for each of the attributes.

**Exercise 11.3** Write a class definition for `Date`, an object type that contains three integers: `year`, `month`, and `day`. This class should provide two constructors. The first should take no parameters and initialize a default date. The second should take parameters named `year`, `month` and `day`, and use them to initialize the instance variables.

Write a `main` method that creates a new `Date` object named `birthday`. The new object should contain your birth date. You can use either constructor.

**Exercise 11.4** A “rational number” is a number that can be represented as the ratio of two integers. For example,  $2/3$  is a rational number, and you can think of 7 as a rational number with an implicit 1 in the denominator.

The purpose of this exercise is to write a class definition that includes a variety of methods, including constructors, static methods, instance methods, modifiers, and pure methods:

1. Define a class called `Rational`. A `Rational` object should have two integer instance variables that store the numerator and denominator.
2. Write a constructor that takes no arguments and sets the numerator to 0 and denominator to 1.
3. Write an instance method called `printRational` that displays a `Rational` object in a reasonable format.

4. Write a `main` method that creates a new object with type `Rational`, sets its instance variables to the values of your choice, and displays the object.
5. You now have a minimal testable program. Test it and, if necessary, debug it.
6. Write a `toString` method for `Rational` and test it using `println`.
7. Write a second constructor that takes two arguments and uses them to initialize the instance variables.
8. Write an instance method called `negate` that reverses the sign of a rational number. This method should be a modifier, so it should be `void`. Add lines to `main` to test the new method.
9. Write an instance method called `invert` that swaps the numerator and denominator. It should be a modifier. Add lines to `main` to test the new method.
10. Write an instance method called `toDouble` that converts the rational number to a `double` (floating-point number) and returns the result. This method is a pure method; it does not modify the object. As always, test the new method.
11. Write an instance method named `reduce` that reduces a rational number to its lowest terms by finding the greatest common divisor (GCD) of the numerator and denominator and dividing through. This method should be a pure method; it should not modify the instance variables of the object on which it is invoked.

*Hint:* Finding the GCD takes only a few lines of code. Search the web for “Euclidean algorithm”.
12. Write an instance method called `add` that takes a `Rational` number as an argument, adds it to `this`, and returns a new `Rational` object. There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).



# Chapter 12

## Arrays of Objects

During the next three chapters, we will develop programs that work with playing cards and decks of cards. Here is an outline of the road ahead:

- In this chapter, we define a **Card** class and write methods that work with cards and arrays of cards.
- In Chapter 13.1, we define a **Deck** class that encapsulates an array of cards, and we write methods that operate on decks.
- In Chapter 14, we introduce a way to define new classes that extend existing classes. Then we use **Card** and **Deck** to implement the game Crazy Eights.

There are 52 cards in a standard deck. Each card belongs to one of four suits and one of 13 ranks. The suits are Clubs, Diamonds, Hearts, and Spades. The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King.

If you are unfamiliar with traditional playing cards, now would be a good time to get a deck or read through [https://en.wikipedia.org/wiki/Standard\\_52-card\\_deck](https://en.wikipedia.org/wiki/Standard_52-card_deck).

## 12.1 Card Objects

If we want to define a class to represent a playing card, it is pretty clear what the instance variables should be: `rank` and `suit`. It is not as obvious what types they should be.

One possibility is a `String` containing things like `"Spade"` for suits and `"Queen"` for ranks. A problem with this choice is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By encode, we *don't* mean to encrypt or translate into a secret code. We mean to define a mapping between a sequence of numbers and the things we want to represent.

Here is a mapping for suits:

Clubs	$\mapsto$	0
Diamonds	$\mapsto$	1
Hearts	$\mapsto$	2
Spades	$\mapsto$	3

We use the mathematical symbol  $\mapsto$  to make it clear that these mappings are not part of the program. They are part of the program design, but they never appear explicitly in the code.

Each of the numerical ranks (2 through 10) maps to the corresponding integer. For the face cards, we can use the following:

Ace	$\mapsto$	1
Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

With this encoding, the class definition for the `Card` type looks like this:

```
public class Card {  
    private int rank;  
    private int suit;  
  
    public Card(int rank, int suit) {  
        this.rank = rank;  
        this.suit = suit;  
    }  
}
```

The instance variables are `private`: we can access them from inside this class, but not from other classes.

The constructor takes a parameter for each instance variable. To create a `Card` object, we use the `new` operator:

```
Card threeOfClubs = new Card(3, 0);
```

The result is a reference to a `Card` that represents the 3 of Clubs.

## 12.2 Card toString

When you create a new class, the first step is to declare the instance variables and write constructors. A good next step is to write `toString`, which is useful for debugging and incremental development.

To display `Card` objects in a way that humans can read easily, we need to “decode” the integer values as words. A natural way to do that is with an array of `Strings`. For example, we can create the array like this:

```
String[] suits = new String[4];
```

And then assign values to the elements:

```
suits[0] = "Clubs";  
suits[1] = "Diamonds";  
suits[2] = "Hearts";  
suits[3] = "Spades";
```

Or we can create the array and initialize the elements at the same time, as you saw in Section 7.3:

```
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
```

The memory diagram in Figure 12.1 shows the result. Each element of the array is a reference to a `String`.

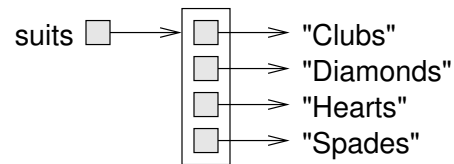


Figure 12.1: Memory diagram of an array of strings.

We also need an array to decode the ranks:

```
String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",  
                  "7", "8", "9", "10", "Jack", "Queen", "King"};
```

The zeroth element should never be used, because the only valid ranks are 1–13. We set it to `null` to indicate an unused element.

Using these arrays, we can create a meaningful `String` by using `suit` and `rank` as indexes.

```
String s = ranks[this.rank] + " of " + suits[this.suit];
```

The expression `ranks[this.rank]` means “use the instance variable `rank` from `this` object as an index into the array `ranks`.” We select the string for `this.suit` in a similar way.

Now we can wrap all the previous code in a `toString` method:

```
public String toString() {  
    String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",  
                     "7", "8", "9", "10", "Jack", "Queen", "King"};  
    String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};  
    String s = ranks[this.rank] + " of " + suits[this.suit];  
    return s;  
}
```

When we display a card, `println` automatically calls `toString`. The output of the following code is Jack of Diamonds:

```
Card card = new Card(11, 1);
System.out.println(card);
```

## 12.3 Class Variables

So far you have seen local variables, which are declared inside a method, and instance variables, which are declared in a class definition, usually before the method definitions. Now it's time to learn about **class variables**. They are shared across all instances of the class.

Like instance variables, class variables are defined in a class definition, before the method definitions. But they are identified by the keyword `static`. Here is a version of `Card` in which `RANKS` and `SUITS` are defined as class variables:

```
public class Card {

    public static final String[] RANKS = {
        null, "Ace", "2", "3", "4", "5", "6", "7",
        "8", "9", "10", "Jack", "Queen", "King"};

    public static final String[] SUITS = {
        "Clubs", "Diamonds", "Hearts", "Spades"};

    // instance variables and constructors go here

    public String toString() {
        return RANKS[this.rank] + " of " + SUITS[this.suit];
    }
}
```

Class variables are allocated when the program begins and persist until the program ends. In contrast, instance variables like `rank` and `suit` are allocated when the program creates `new` objects, and they are deleted when the object is garbage-collected (see Section 10.9).



Class variables are often used to store constant values that are needed in several places. In that case, they should also be declared as `final`. Note that whether a variable is `static` or `final` involves two separate considerations: `static` means the variable is *shared*, and `final` means the variable (or in this case, the reference) is *constant*.

Naming `static final` variables with capital letters is a common convention that makes it easier to recognize their role in the class. In the `toString` method, we refer to `SUITS` and `RANKS` as if they were local variables, but we can tell that they are class variables.

One advantage of defining `SUITS` and `RANKS` as class variables is that they don't need to be created (and garbage-collected) every time `toString` is called. They may also be needed in other methods and classes, so it's helpful to make them available everywhere.

## 12.4 The `compareTo` Method

As you saw in Section 11.7, it's helpful to create an `equals` method to test whether two objects are equivalent:

```
public boolean equals(Card that) {  
    return this.rank == that.rank  
        && this.suit == that.suit;  
}
```

It would also be nice to have a method for comparing cards, so we can tell if one is higher or lower than another. For primitive types, we can use comparison operators like `<` and `>` to compare values. But these operators don't work for object types.

For strings, Java provides a `compareTo` method, as you saw in Section 6.10. We can write our own version of `compareTo` for the classes that we define, as we did for the `equals` method.

Some types are “totally ordered”, which means that you can compare any two values and tell which is bigger. Integers and strings are totally ordered. Other types are “unordered”, which means that there is no meaningful way

to say that one element is bigger than another. In Java, the `boolean` type is unordered; if you try to compare `true < false`, you get a compiler error.

The set of playing cards is “partially ordered”, which means that sometimes we can compare cards and sometimes not. For example, we know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

To make cards comparable, we have to decide which is more important: rank or suit. The choice is arbitrary, and it might be different for different games. But when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on. So for now, let’s say that suit is more important. With that decided, we can write `compareTo` as follows:

```
public int compareTo(Card that) {
    if (this.suit < that.suit) {
        return -1;
    }
    if (this.suit > that.suit) {
        return 1;
    }
    if (this.rank < that.rank) {
        return -1;
    }
    if (this.rank > that.rank) {
        return 1;
    }
    return 0;
}
```

`compareTo` returns `-1` if `this` is a lower card, `+1` if `this` is a higher card, and `0` if `this` and `that` are equivalent. It compares suits first. If the suits are the same, it compares ranks. If the ranks are also the same, it returns `0`.

## 12.5 Cards Are Immutable

The instance variables of `Card` are `private`, so they can’t be accessed from other classes. We can provide getters to allow other classes to read the `rank`

and suit values:

```
public int getRank() {  
    return this.rank;  
}  
  
public int getSuit() {  
    return this.suit;  
}
```

Whether or not to provide setters is a design decision. If we did, cards would be mutable, so you could transform one card into another. That is probably not a feature we want, and in general, mutable objects are more error-prone. So it might be better to make cards immutable. To do that, all we have to do is *not* provide any modifier methods (including setters).

That's easy enough, but it is not foolproof, because a fool might come along later and add a modifier. We can prevent that possibility by declaring the instance variables `final`:

```
public class Card {  
    private final int rank;  
    private final int suit;  
  
    ...  
}
```

You can initialize these variables inside a constructor, but if someone writes a method that tries to modify them, they'll get a compiler error. This kind of safeguard helps prevent future mistakes and hours of debugging.

## 12.6 Arrays of Cards

Just as you can create an array of `String` objects, you can create an array of `Card` objects. The following statement creates an array of 52 cards. Figure 12.2 shows the memory diagram for this array.

```
Card[] cards = new Card[52];
```

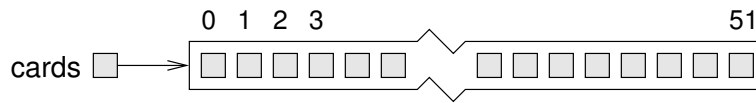


Figure 12.2: Memory diagram of an unpopulated `Card` array.

Although we call it an “array of cards”, the array contains *references* to cards; it does not contain the `Card` objects themselves. Initially the references are all `null`.

Even so, you can access the elements of the array in the usual way:

```
if (cards[0] == null) {  
    System.out.println("No card yet!");  
}
```

But if you try to access the instance variables of non-existent `Card` objects, you will get a `NullPointerException`:

```
System.out.println(cards[0].rank); // NullPointerException
```

That code won’t work until we put cards in the array. One way to populate the array is to write nested `for` loops:

```
int index = 0;  
for (int suit = 0; suit <= 3; suit++) {  
    for (int rank = 1; rank <= 13; rank++) {  
        cards[index] = new Card(rank, suit);  
        index++;  
    }  
}
```

The outer loop iterates suits from 0 to 3. For each suit, the inner loop iterates ranks from 1 to 13. Since the outer loop runs 4 times, and the inner loop runs 13 times for each suit, the body is executed 52 times.

We use a separate variable `index` to keep track of where in the array the next card should go. Figure 12.3 shows what the array looks like after the first two cards have been created.

When you work with arrays, it is convenient to have a method that displays the contents. You have seen the pattern for traversing an array several times, so the following method should be familiar:

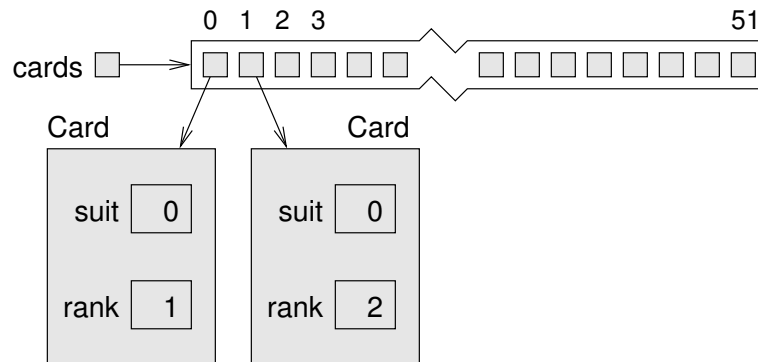


Figure 12.3: Memory diagram of a `Card` array with two cards.

```
public static void printDeck(Card[] cards) {  
    for (Card card : cards) {  
        System.out.println(card);  
    }  
}
```

Since `cards` has type `Card[]`, pronounced “card array”, an element of `cards` has type `Card`. So `println` invokes the `toString` method in the `Card` class.

Then again, we don’t have to write our own `printDeck` method. The `Arrays` class provides a `toString` method that invokes `toString` on the elements of an array and concatenates the results:

```
System.out.println(Arrays.toString(cards))
```

## 12.7 Sequential Search

The next method we’ll write is `search`, which takes an array of cards and a `Card` object as parameters. It returns the index where the `Card` appears in the array, or `-1` if it doesn’t. This version of `search` uses the algorithm in Section 7.5, which is called **sequential search**:

```
public static int search(Card[] cards, Card target) {  
    for (int i = 0; i < cards.length; i++) {  
        if (cards[i].equals(target)) {  
            return i;  
        }  
    }  
    return -1;  
}
```

The method returns as soon as it discovers the card, which means we don't have to traverse the entire array if we find the target. If we get to the end of the loop, we know the card is not in the array.

If the cards in the array are not in order, there is no way to search faster than sequential search. We have to look at every card, because otherwise we can't be certain the card we want is not there. But if the cards are in order, we can use better algorithms.

Sequential search is relatively inefficient, especially for large arrays. If you pay the price to keep the array sorted, finding elements becomes much easier.

## 12.8 Binary Search

When you look for a word in a dictionary, you don't search page by page from front to back. Since the words are in alphabetical order, you probably use a **binary search** algorithm:

1. Start on a page near the middle of the dictionary.
2. Compare a word on the page to the word you are looking for. If you find it, stop.
3. If the word on the page comes before the word you are looking for, flip to somewhere later in the dictionary and go to step 2.
4. If the word on the page comes after the word you are looking for, flip to somewhere earlier in the dictionary and go to step 2.

This algorithm is much faster than sequential search, because it rules out half of the remaining words each time you make a comparison. If at any point you find two adjacent words on the page, and your word comes between them, you can conclude that your word is not in the dictionary.

Getting back to the array of cards, we can write a faster version of `search` if we know the cards are in order:

```
public static int binarySearch(Card[] cards, Card target) {
    int low = 0;
    int high = cards.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;           // step 1
        int comp = cards[mid].compareTo(target);

        if (comp == 0) {                     // step 2
            return mid;
        } else if (comp < 0) {               // step 3
            low = mid + 1;
        } else {                             // step 4
            high = mid - 1;
        }
    }
    return -1;
}
```

First, we declare `low` and `high` variables to represent the range we are searching. Initially, we search the entire array, from 0 to `cards.length - 1`.

Inside the `while` loop, we repeat the four steps of binary search:

1. Choose an index between `low` and `high`—call it `mid`—and compare the card at `mid` to the target.
2. If you found the target, return its index (which is `mid`).
3. If the card at `mid` is lower than the target, search the range from `mid + 1` to `high`.
4. If the card at `mid` is higher than the target, search the range from `low` to `mid - 1`.

If `low` exceeds `high`, there are no cards in the range, so we terminate the loop and return `-1`.

This algorithm depends on only the `compareTo` method of the object, so we can use this code with any object type that provides `compareTo`.

## 12.9 Tracing the Code

To see how binary search works, it's helpful to add the following print statement at the beginning of the loop:

```
System.out.println(low + ", " + high);
```

Using a sorted deck of cards, we can search for the Jack of Clubs like this:

```
Card card = new Card(11, 0);  
System.out.println(binarySearch(cards, card));
```

We expect to find this card at position 10 (since the Ace of Clubs is at position 0). Here is the output of `binarySearch`:

```
0, 51  
0, 24  
0, 11  
6, 11  
9, 11  
10
```

You can see the range of cards shrinking as the `while` loop runs, until eventually index 10 is found. If we search for a card that's not in the array—like `new Card(15, 1)`, or the 15 of Diamonds—we get the following:

```
0, 51  
26, 51  
26, 37  
26, 30  
26, 27  
-1
```



Each time through the loop, we cut the distance between `low` and `high` in half. After  $k$  iterations, the number of remaining cards is  $52/2^k$ . To find the number of iterations it takes to complete, we set  $52/2^k = 1$  and solve for  $k$ . The result is  $\log_2 52$ , which is about 5.7. So we might have to look at 5 or 6 cards, as opposed to all 52 if we did a sequential search.

More generally, if the array contains  $n$  elements, binary search requires  $\log_2 n$  comparisons, and sequential search requires  $n$ . For large values of  $n$ , binary search is substantially faster.

## 12.10 Vocabulary

**encode:** To represent one set of values using another set of values by constructing a mapping between them.

**class variable:** A variable declared within a class as `static`. There is only one copy of a class variable, no matter how many objects there are.

**sequential search:** An algorithm that searches array elements, one by one, until a target value is found.

**binary search:** An algorithm that searches a sorted array by starting in the middle, comparing an element to the target, and eliminating half of the remaining elements.

## 12.11 Exercises

The code for this chapter is in the `ch12` directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise 12.1** Encapsulate the deck-building code from Section 12.6 in a method called `makeDeck` that takes no parameters and returns a fully populated array of `Cards`.

**Exercise 12.2** In some card games, Aces are ranked higher than Kings. Modify the `compareTo` method to implement this ordering.

**Exercise 12.3** In Poker a “flush” is a hand that contains five or more cards of the same suit. A hand can contain any number of cards.

1. Write a method called `suitHist` that takes an array of cards as a parameter and returns a histogram of the suits in the hand. Your solution should traverse the array only once, as in Section 7.7.
2. Write a method called `hasFlush` that takes an array of cards as a parameter and returns `true` if the hand contains a flush (and `false` otherwise).
3. A “royal flush” includes the Ace, King, Queen, Jack, and 10 (all in the same suit). Write a method called `hasRoyal` that determines whether an array of cards contains a royal flush.

**Exercise 12.4** Working with cards is more fun if you can display them on the screen. If you have not already read Appendix C about 2D graphics, you should read it before working on this exercise. In the code directory for this chapter, *ch12*, you will find the following:

- *cardset-oxymoron*  
A directory containing images of playing cards.
- *CardTable.java*  
A sample program that demonstrates how to read and display images.

*CardTable.java* demonstrates the use of a 2D array; specifically, an array of card images. The declaration looks like this:

```
private Image[] [] images;
```

The variable `images` refers to a 2D array of `Image` objects, which are defined in the `java.awt` package. Here’s the code that creates the array itself:

```
images = new Image[14][4];
```

The array has 14 rows (one for each rank, plus an unused row for rank 0) and 4 columns (one for each suit). Here’s the loop that populates the array:

```
String cardset = "cardset-oxymoron";
String suits = "cdhs";

for (int suit = 0; suit <= 3; suit++) {
    char c = suits.charAt(suit);

    for (int rank = 1; rank <= 13; rank++) {
        String s = String.format("%s/%02d%c.gif",
                                cardset, rank, c);
        images[rank][suit] = new ImageIcon(s).getImage();
    }
}
```

The variable `cardset` is the name of the directory that contains the image files. `suits` is a string that contains the single-letter abbreviations for the suits. These strings are used to assemble `s`, which contains the filename for each image. For example, when `rank=1` and `suit=2`, the value of `s` is `"cardset-oxymoron/01h.gif"`, which is an image of the Ace of Hearts.

The last line of the loop reads the image file, extracts an `Image` object, and assigns it to a location in the array, as specified by the indexes `rank` and `suit`. For example, the image of the Ace of Hearts is stored in row 1, column 2.

If you compile and run *CardTable.java*, you should see images of a deck of cards laid out on a green table. You can use this class as a starting place to implement your own card games.

As a starting place, try placing cards on the table in the starting configuration for the solitaire game Klondike ([https://en.wikipedia.org/wiki/Klondike\\_\(solitaire\)](https://en.wikipedia.org/wiki/Klondike_(solitaire))).

You can get the image for the back of the card by reading the file *back192.gif*.

# Chapter 13

## Objects of Arrays

In the previous chapter, we defined a class to represent cards and used an array of `Card` objects to represent a deck. In this chapter, we take additional steps toward object-oriented programming.

First we define a class to represent a deck of cards. Then we present algorithms for shuffling and sorting decks. Finally, we introduce `ArrayList` from the Java library and use it to represent collections of cards.

### 13.1 Decks of Cards

Here is the beginning of a `Deck` class that encapsulates an array of `Card` objects:

```
public class Deck {
    private Card[] cards;

    public Deck(int n) {
        this.cards = new Card[n];
    }

    public Card[] getCards() {
        return this.cards;
    }
}
```

The constructor initializes the instance variable with an array of *n* cards, but it doesn't create any *Card* objects. Figure 13.1 shows what a *Deck* looks like with no cards.

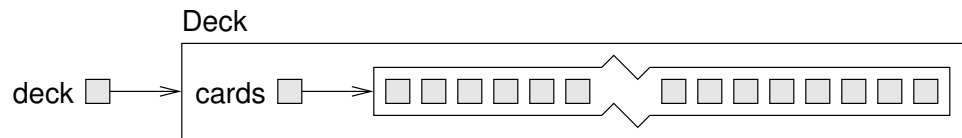


Figure 13.1: Memory diagram of an unpopulated *Deck* object.

We'll add another constructor that creates a standard 52-card array and populates it with *Card* objects:

```
public Deck() {
    this.cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            this.cards[index] = new Card(rank, suit);
            index++;
        }
    }
}
```

This method is similar to the example in Section 12.6; we just turned it into a constructor. We can use it to create a complete *Deck* like this:

```
Deck deck = new Deck();
```

Now that we have a *Deck* class, we have a logical place to put methods that pertain to decks. Looking at the methods we have written so far, one obvious candidate is *printDeck* from Section 12.6. Here's how it looks, rewritten as an instance method of *Deck*:

```
public void print() {
    for (Card card : this.cards) {
        System.out.println(card);
    }
}
```

Notice that when we transform a static method into an instance method, the code is shorter. Here's how we invoke it:

```
deck.print();
```

## 13.2 Shuffling Decks

For most card games, you have to shuffle the deck; that is, put the cards in a random order. In Section 7.6 you saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle; for example, we could divide the deck in two halves and then choose alternately from each one. Humans usually don't shuffle perfectly, so after about seven iterations, the order of the deck is pretty well randomized.

But a computer program would have the annoying property of doing a perfect shuffle every time, which is not very random. In fact, after eight perfect shuffles, you would find the deck back in the order you started in! For more on this, see [https://en.wikipedia.org/wiki/Faro\\_shuffle](https://en.wikipedia.org/wiki/Faro_shuffle).

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration, choose two cards and swap them. To outline this algorithm, we'll use a combination of Java statements and English comments. This technique is sometimes called **pseudocode**:

```
public void shuffle() {  
    for each index i {  
        // choose a random number between i and length - 1  
        // swap the ith card and the randomly-chosen card  
    }  
}
```

The nice thing about pseudocode is that it often makes clear what other methods you are going to need. In this case, we need a method that chooses a random integer in a given range and a method that takes two indexes and swaps the cards at those positions:

```
private static int randomInt(int low, int high) {  
    // return a random number between low and high,  
    // including both  
}  
  
private void swapCards(int i, int j) {  
    // swap the ith and the jth cards in the array  
}
```

Methods like `randomInt` and `swapCards` are called **helper methods**, because they help you solve parts of the problem. Helper methods are often `private`, because they are used only by methods in the class and are not needed by methods in other classes.

The process of writing pseudocode first and then writing helper methods to make it work is a kind of **top-down design** (see [https://en.wikipedia.org/wiki/Top-down\\_and\\_bottom-up\\_design](https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design)). It is an alternative to “incremental development” and “encapsulation and generalization”, the other design processes you have seen in this book.

One of the exercises at the end of the chapter asks you to write the helper methods `randomInt` and `swapCards`, and use them to implement `shuffle`.

When you do the exercise, notice that `randomInt` is a class method and `swapCards` is an instance method. Do you understand why?

## 13.3 Selection Sort

Now that we have shuffled the deck, we need a way to put it back in order. There is an algorithm for sorting that is ironically similar to the algorithm for shuffling. It’s called **selection sort**, because it works by traversing the array repeatedly and selecting the lowest (or highest) remaining card each time.

During the first iteration, we find the lowest card and swap it with the card in the zeroth position. During the  $i$ th iteration, we find the lowest card to the right of  $i$  and swap it with the  $i$ th card. Here is pseudocode for selection sort:

```
public void selectionSort() {  
    for each index i {  
        // find the lowest card at or to the right of i  
        // swap the ith card and the lowest card found  
    }  
}
```

Again, the pseudocode helps with the design of the helper methods. For this algorithm, we can reuse `swapCards` from the previous section, so we need only a method to find the lowest card; we'll call it `indexLowest`:

```
private int indexLowest(int low, int high) {  
    // find the lowest card between low and high  
}
```

One of the exercises at the end of the chapter asks you to write `indexLowest`, and then use it and `swapCards` to implement `selectionSort`.

## 13.4 Merge Sort

Selection sort is a simple algorithm, but it is not very efficient. To sort  $n$  items, it has to traverse the array  $n - 1$  times. Each traversal takes an amount of time proportional to  $n$ . The total time, therefore, is proportional to  $n^2$ .

We will develop a more efficient algorithm called **merge sort**. To sort  $n$  items, merge sort takes time proportional to  $n \log_2 n$ . That may not seem impressive, but as  $n$  gets big, the difference between  $n^2$  and  $n \log_2 n$  can be enormous.

For example,  $\log_2$  of one million is around 20. So if you had to sort a million numbers, merge sort would require 20 million steps. But selection sort would require one trillion steps!

The idea behind merge sort is this: if you have two decks, each of which has already been sorted, you can quickly merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two decks with about 10 cards each, and sort them so they are face up with the lowest cards on top. Place the decks in front of you.



2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.
3. Repeat step 2 until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. In the next few sections, we'll explain how to implement this algorithm in Java.

## 13.5 Subdecks

The first step of merge sort is to split the deck into two “subdecks”, each with about half of the cards. So we need a method that takes a deck, and a range of indexes, and returns a new deck that contains the specified subset of cards:

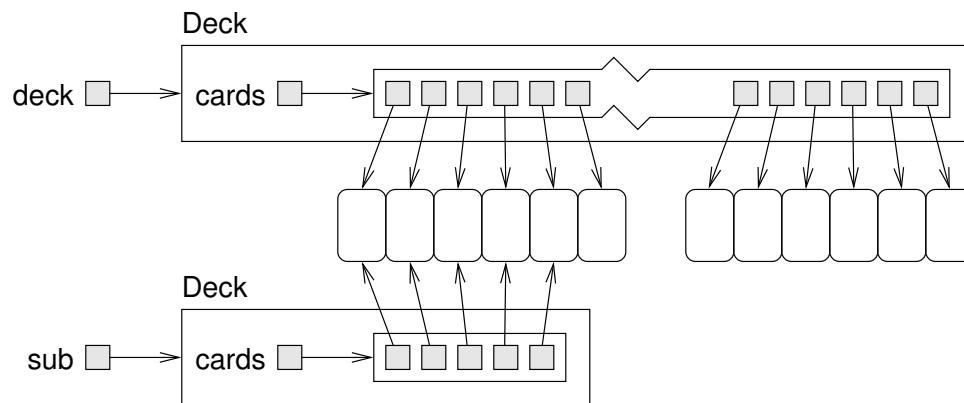
```
public Deck subdeck(int low, int high) {  
    Deck sub = new Deck(high - low + 1);  
    for (int i = 0; i < sub.cards.length; i++) {  
        sub.cards[i] = this.cards[low + i];  
    }  
    return sub;  
}
```

The first line creates an unpopulated `Deck` object that contains an array of `null` references. Inside the `for` loop, the subdeck gets populated with references to `Card` objects.

The length of the subdeck is `high - low + 1`, because both the low card and the high card are included. This sort of computation can be confusing, and forgetting the “+ 1” often leads to **off-by-one** errors. Drawing a picture is usually the best way to avoid them.

Figure 13.2 is a memory diagram of a subdeck with `low = 0` and `high = 4`. The result is a hand with five cards that are *shared* with the original deck; that is, they are aliased.

Aliasing might not be a good idea, because changes to shared cards would be reflected in multiple decks. But since `Card` objects are immutable, this kind of aliasing is not a problem. And it saves some memory because we don't create duplicate `Card` objects.

Figure 13.2: Memory diagram showing the effect of `subdeck`.

## 13.6 Merging Decks

The next helper method we need is `merge`, which takes two sorted subdecks and returns a new deck containing all cards from both decks, in order. Here's what the algorithm looks like in pseudocode, assuming the subdecks are named `d1` and `d2`:

```
private static Deck merge(Deck d1, Deck d2) {
    // create a new deck, d3, big enough for all the cards

    // use the index i to keep track of where we are at in
    // the first deck, and the index j for the second deck
    int i = 0;
    int j = 0;

    // the index k traverses the result deck
    for (int k = 0; k < d3.length; k++) {
        // if d1 is empty, use top card from d2
        // if d2 is empty, use top card from d1
        // otherwise, compare the top two cards

        // add lowest card to the new deck at k
        // increment i or j (depending on card)
    }
    // return the new deck
}
```

An exercise at the end of the chapter asks you to implement `merge`. It's a little tricky, so be sure to test it with different subdecks. Once your `merge` method is working, you can use it to write a simplified version of merge sort:

```
public Deck almostMergeSort() {  
    // divide the deck into two subdecks  
    // sort the subdecks using selectionSort  
    // merge the subdecks, return the result  
}
```

If you have working versions of `subdeck`, `selectionSort`, and `merge`, you should have no trouble getting this method working. But it is still not very efficient, because it uses `selectionSort` to sort the subdecks. We can make it more efficient if we use `mergeSort` instead, but that means we have to make it recursive!

## 13.7 Adding Recursion

To make `mergeSort` work recursively, you have to add a base case; otherwise, it repeats forever. The simplest base case is a subdeck with one card. If there is only one card, it can't be out of order, so we consider it sorted. And if it is already sorted, we can just return it.

And it will turn out to be convenient if we handle another base case, a subdeck with zero cards. By the same logic, if there are no cards, they can't be out of order. So we consider an empty deck to be sorted, and return it.

With these base cases, a recursive version of `mergeSort` looks like this:

```
public Deck mergeSort() {  
    // if the deck has 0 or 1 cards, return it  
    // otherwise, divide the deck into two subdecks  
    // sort the subdecks using mergeSort  
    // merge the subdecks  
    // return the result  
}
```

As usual, there are two ways to think about recursive programs: you can follow the flow of execution, or you can make the “leap of faith” (see Section 8.4). This example should encourage you to make the leap of faith.

When you use `selectionSort` to sort the subdecks, you don't feel compelled to follow the flow of execution. You assume it works because you already debugged it. When you make `mergeSort` recursive, you just replace one sorting algorithm with another. There is no reason to read the program differently.

Well, almost. You have to think about the base cases and make sure that you reach them. But other than that, writing the recursive version should be no problem. As an exercise at the end of this chapter, you'll have a chance to finish off this example.

## 13.8 Static Context

Figure 13.3 shows a UML class diagram for `Deck`, including the instance variable, `cards`, and the methods we have so far. In UML diagrams, `private` attributes and methods begin with a minus sign (-) and `static` methods are underlined.

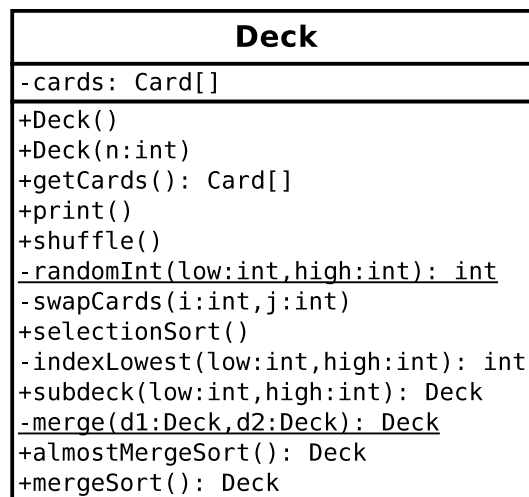


Figure 13.3: UML diagram for the `Deck` class.

The helper methods `randomInt` and `merge` are `static`, because they do not read or write any instance variables. All other methods are instance methods, because they access the instance variable, `cards`.

When you have static methods and instance methods in the same class, it is easy to get them confused.

To invoke an instance method, you need an instance:

```
Deck deck = new Deck();  
deck.print(); // correct
```

Deck with a capital D is a class, and deck with a lowercase d is an object.

Say you try to invoke print like this:

```
Deck.print(); // wrong!
```

You get a compiler error like this:

```
Non-static method print() cannot be referenced from a  
static context.
```

By “static context”, the compiler means you are trying to invoke a method in a context that requires a static method.

On the other hand, if you have a Deck object, you can use it to invoke a static method:

```
Deck deck = new Deck();  
int i = deck.randomInt(0, 51); // legal, but not good style
```

This is legal, but it is not considered good style, because someone reading this code would expect randomInt to be an instance method.

Another common error is to use this in a static method. For example, say you write something like this:

```
private static Deck merge(Deck d1, Deck d2) {  
    return this.cards; // wrong!  
}
```

You get a compiler error like this:

```
Non-static variable this cannot be referenced from a  
static context.
```

The problem is that cards is an instance variable, so it is *non-static*; therefore, you can't access it from a static method. In general, you can't use this in a static method, because a static method is not invoked on an object.

For beginners, error messages about non-static context can be confusing and frustrating. We hope this section helps.

## 13.9 Piles of Cards

Now that we have classes that represent cards and decks, let's use them to make a game. One of the simplest card games that children play is called "War" (see [https://en.wikipedia.org/wiki/War\\_\(card\\_game\)](https://en.wikipedia.org/wiki/War_(card_game))).

Initially, the deck is divided evenly into two piles, one for each player. During each round, each player takes the top card from their pile and places it, face up, in the center. Whoever has the highest-ranking card, ignoring suit, takes the two cards and adds them to the bottom of their pile. The game continues until one player has won the entire deck.

We could use the `Deck` class to represent the individual piles. However, our implementation of `Deck` uses a `Card` array, and the length of an array can't change. As the game progresses, we need to be able to add and remove cards from the piles.

We can solve this problem with an `ArrayList`, which is in the `java.util` package. An `ArrayList` is a **collection**, which is an object that contains other objects. It provides methods to add and remove elements, and it grows and shrinks automatically.

We define a new class named `Pile` to represent a pile of cards. It uses an `ArrayList` to store `Card` objects:

```
public class Pile {
    private ArrayList<Card> cards;

    public Pile() {
        this.cards = new ArrayList<Card>();
    }
}
```

When you declare an `ArrayList`, you specify the type it contains in angle brackets (<>). This declaration says that `cards` is not just an `ArrayList`; it's an `ArrayList` of `Card` objects. The constructor initializes `this.cards` with an empty `ArrayList`.

Now let's think about the methods we need to play the game. At the beginning of each round, each player draws a card from the top of their pile. So we define a method to do that:

```
public Card popCard() {  
    return this.cards.remove(0); // from the top of the pile  
}
```

`popCard` removes the `Card` at the beginning of the `ArrayList`, which we think of as the top of the pile. Because we use `ArrayList.remove`, it automatically shifts the remaining cards to fill the gap.

At the end of each round, the winner adds cards to the bottom of their pile. So we define a method to do that:

```
public void addCard(Card card) {  
    this.cards.add(card); // to the bottom of the pile  
}
```

`ArrayList` provides a method, `add`, that adds an element to the end of the collection, which we think of as the bottom of the pile.

To know when to stop the game, we have to check if one of the piles is empty. Here's a method to do that:

```
public boolean isEmpty() {  
    return this.cards.isEmpty();  
}
```

So far, these methods don't do very much; they just invoke methods on the instance variable, `cards`. Methods like these are called **wrapper methods** because they wrap one method with another.

Finally, to start the game, we need to divide the deck into two equal parts. We can do that with `subdeck` from Section 13.5 and a new method, `addDeck`:

```
public void addDeck(Deck deck) {  
    for (Card card : deck.getCards()) {  
        this.cards.add(card);  
    }  
}
```

`addDeck` takes a `Deck` object, loops through the cards, and adds them to the `Pile`. Notice that it does not remove the cards from the `Deck`, so the `Deck` and the `Pile` share cards. But that won't be a problem because cards are immutable.

## 13.10 Playing War

Now we can use `Deck` and `Pile` to implement the game. We'll start by creating a deck and shuffling:

```
Deck deck = new Deck();
deck.shuffle();
```

Then we divide the `Deck` into two piles:

```
Pile p1 = new Pile();
p1.addDeck(deck.subdeck(0, 25));

Pile p2 = new Pile();
p2.addDeck(deck.subdeck(26, 51));
```

The game itself is a loop that repeats until one of the piles is empty. At each iteration, we draw a card from each pile and compare their ranks:

```
while (!p1.isEmpty() && !p2.isEmpty()) {
    // pop a card from each pile
    Card c1 = p1.popCard();
    Card c2 = p2.popCard();

    // compare the cards
    int diff = c1.getRank() - c2.getRank();
    if (diff > 0) {
        p1.addCard(c1);
        p1.addCard(c2);
    } else if (diff < 0) {
        p2.addCard(c1);
        p2.addCard(c2);
    } else {
        // it's a tie
    }
}
```

If the two cards have the same rank, it's a tie. In that case, each player draws four more cards. Whoever has the higher fourth card takes all cards in play. If there's another tie, they draw another four cards, and so on.



One of the exercises at the end of this chapter asks you to implement the `else` block when there's a tie.

After the `while` loop ends, we display the winner based on which pile is not empty:

```
if (p2.isEmpty()) {  
    System.out.println("Player 1 wins!");  
} else {  
    System.out.println("Player 2 wins!");  
}
```

`ArrayList` provides many other methods that we didn't use for this example. Take a minute to read the documentation, which you can find by doing a web search for "Java `ArrayList`".

## 13.11 Vocabulary

**pseudocode:** A way of designing programs by writing rough drafts in a combination of English and Java.

**helper method:** A method that implements part of a more complex algorithm; often it is not particularly useful on its own.

**top-down design:** Breaking down a problem into subproblems, and solving each subproblem one at a time.

**selection sort:** A simple sorting algorithm that searches for the smallest or largest element  $n$  times.

**merge sort:** A recursive sorting algorithm that divides an array into two parts, sorts each part (using merge sort), and merges the results.

**off-by-one:** A common programming mistake that results in iterating one time too many, or too few.

**static context:** The parts of a class that run without reference to a specific instance of the class.

**collection:** A Java library class, like `ArrayList`, that represents a group of objects.

**wrapper method:** A method that calls another method without doing much additional work.

## 13.12 Exercises

The code for this chapter is in the *ch13* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise 13.1** Write a `toString` method for the `Deck` class. It should return a single string that represents the cards in the deck. When it's printed, this string should display the same results as the `print` method in Section 13.1.

*Hint:* You can use the `+` operator to concatenate strings, but that is not very efficient. Consider using `StringBuilder` instead; see Section 10.11.

**Exercise 13.2** The goal of this exercise is to implement the shuffling algorithm from this chapter.

1. In the repository for this book, you should find the file named *Deck.java*. Check that you can compile it in your environment.
2. Implement the `randomInt` method. You can use the `nextInt` method provided by `java.util.Random`, which you saw in Section 7.6.

*Hint:* To avoid creating a `Random` object every time `randomInt` is invoked, consider defining a class variable.

3. Write a `swapCards` method that takes two indexes and swaps the cards at the given locations.
4. Fill in the `shuffle` method by using the algorithm in Section 13.2.

**Exercise 13.3** The goal of this exercise is to implement the sorting algorithms from this chapter. Use the *Deck.java* file from the previous exercise or create a new one from scratch.

1. Implement the `indexLowest` method. Use the `Card.compareTo` method to find the lowest card in a given range of the deck, from `lowIndex` to `highIndex`, including both.

2. Fill in `selectionSort` by using the algorithm in Section 13.3.
3. Using the pseudocode in Section 13.4, implement the `merge` method. The best way to test it is to build and shuffle a deck. Then use `subdeck` to form two small subdecks, and use selection sort to sort them. Finally, pass the two halves to `merge` and see if it works.
4. Fill in `almostMergeSort`, which divides the deck in half, then uses `selectionSort` to sort the two halves, and uses `merge` to create a new, sorted deck. You should be able to reuse code from the previous step.
5. Implement `mergeSort` recursively. Remember that `selectionSort` is `void` and `mergeSort` returns a new `Deck`, which means that they get invoked differently:

```
deck.selectionSort();    // modifies an existing deck
deck = deck.mergeSort(); // replaces old deck with new
```

**Exercise 13.4** You can learn more about the sorting algorithms presented in this chapter at <https://www.toptal.com/developers/sorting-algorithms>. This site provides explanations of the algorithms, along with animations that show how they work. It also includes an analysis of their efficiency.

For example, “insertion sort” is an algorithm that inserts elements into place, one at a time. Read about it on the website and play the animations. Then write a method named `insertionSort` that implements this algorithm.

One goal of this exercise is to practice top-down design. Your solution should use a helper method, named `insert`, that implements the inner loop of the algorithm. `insertionSort` should invoke this method  $n - 1$  times.

**Exercise 13.5** Find and open the file *War.java* in the repository. The `main` method contains all the code from the last section of this chapter. Check that you can compile and run this code before proceeding.

The program is incomplete; it does not handle the case when two cards have the same rank. Finish implementing the `main` method, beginning at the line that says: `// it's a tie.`

When there's a tie, draw three cards from each pile and store them in a collection, along with the original two. Then draw one more card from each pile and compare them. Whoever wins the tie takes all ten of these cards.

If one pile does not have at least four cards, the game ends immediately. If a tie ends with a tie, draw three more cards, and so on.

Notice that this program depends on `Deck.shuffle`, so you might have to do Exercise 13.2 first.



# Chapter 14

## Extending Classes

In this chapter, we present a more comprehensive example of object-oriented programming.

*Crazy Eights* is a classic card game for two or more players. The main objective is to be the first player to get rid of all your cards. Here's how to play:

- Deal five or more cards to each player, and then deal one card face up to create the “discard pile”. Place the remaining cards face down to create the “draw pile”.
- Each player takes turns placing a single card on the discard pile. The card must match the rank or suit of the previously played card, or be an eight, which is a “wild card”.
- When players don't have a matching card or an eight, they must draw new cards until they get one.
- If the draw pile ever runs out, the discard pile is shuffled (except the top card) and becomes the new draw pile.
- As soon as a player has no cards, the game ends, and all other players score penalty points for their remaining cards. Eights are worth 20, face cards are worth 10, and all others are worth their rank.

You can read [https://en.wikipedia.org/wiki/Crazy\\_Eights](https://en.wikipedia.org/wiki/Crazy_Eights) for more details, but we have enough to get started.

## 14.1 CardCollection

To implement Crazy Eights, we need to represent a deck of cards, a discard pile, a draw pile, and a hand for each player. And we need to be able to deal, draw, and discard cards.

The `Deck` and `Pile` classes from the previous chapter meet some of these requirements. But unless we make some changes, neither of them represents a hand of cards very well.

Furthermore, `Deck` and `Pile` are essentially two versions of the same code: one based on arrays, and the other based on `ArrayList`. It would be helpful to combine their features into one class that meets the needs of both.

We will define a class named `CardCollection` and add the code we want one step at a time. Since this class will represent different piles and hands of cards, we'll add a `label` attribute to tell them apart:

```
public class CardCollection {  
  
    private String label;  
    private ArrayList<Card> cards;  
  
    public CardCollection(String label) {  
        this.label = label;  
        this.cards = new ArrayList<Card>();  
    }  
}
```

As with the `Pile` class, we need a way to add cards to the collection. Here is the `addCard` method from the previous chapter:

```
public void addCard(Card card) {  
    this.cards.add(card);  
}
```

Until now, we have used `this` explicitly to make it easy to identify attributes. Inside `addCard` and other instance methods, you can access instance variables without using the keyword `this`. So from here on, we will drop it:

```
public void addCard(Card card) {  
    cards.add(card);  
}
```

We also need to be able to remove cards from the collection. The following method takes an index, removes the card at that location, and shifts the following cards left to fill the gap:

```
public Card popCard(int i) {  
    return cards.remove(i);  
}
```

If we are dealing cards from a shuffled deck, we don't care which card gets removed. It is most efficient to choose the last one, so we don't have to shift any cards left. Here is an overloaded version of `popCard` that removes and returns the last card:

```
public Card popCard() {  
    int i = cards.size() - 1;    // from the end of the list  
    return popCard(i);  
}
```

`CardCollection` also provides `isEmpty`, which returns `true` if there are no cards left, and `size`, which returns the number of cards:

```
public boolean isEmpty() {  
    return cards.isEmpty();  
}
```

```
public int size() {  
    return cards.size();  
}
```

To access the elements of an `ArrayList`, you can't use the array `[]` operator. Instead, you have to use the methods `get` and `set`. Here is a wrapper for `get`:

```
public Card getCard(int i) {  
    return cards.get(i);  
}
```

`lastCard` gets the last card (but doesn't remove it):



```
public Card lastCard() {  
    int i = cards.size() - 1;  
    return cards.get(i);  
}
```

In order to control the ways card collections are modified, we don't provide a wrapper for `set`. The only modifiers we provide are the two versions of `popCard` and the following version of `swapCards`:

```
public void swapCards(int i, int j) {  
    Card temp = cards.get(i);  
    cards.set(i, cards.get(j));  
    cards.set(j, temp);  
}
```

Finally, we use `swapCards` to implement `shuffle`, which we described in Section 13.2:

```
public void shuffle() {  
    Random random = new Random();  
    for (int i = cards.size() - 1; i > 0; i--) {  
        int j = random.nextInt(i + 1);  
        swapCards(i, j);  
    }  
}
```

## 14.2 Inheritance

At this point, we have a class that represents a collection of cards. It provides functionality common to decks of cards, piles of cards, hands of cards, and potentially other collections.

However, each kind of collection will be slightly different. Rather than add every possible feature to `CardCollection`, we can use **inheritance** to define subclasses. A **subclass** is a class that “extends” an existing class; that is, it has the attributes and methods of the existing class, plus more.

Here is the complete definition of our new and improved `Deck` class:

```
public class Deck extends CardCollection {  
  
    public Deck(String label) {  
        super(label);  
        for (int suit = 0; suit <= 3; suit++) {  
            for (int rank = 1; rank <= 13; rank++) {  
                addCard(new Card(rank, suit));  
            }  
        }  
    }  
}
```

The first line uses the keyword `extends` to indicate that `Deck` extends the class `CardCollection`. That means a `Deck` object has the same instance variables and methods as a `CardCollection`. Another way to say the same thing is that `Deck` “inherits from” `CardCollection`. We could also say that `CardCollection` is a **superclass**, and `Deck` is one of its subclasses.

In Java, classes may extend only one superclass. Classes that do not specify a superclass with `extends` automatically inherit from `java.lang.Object`. So in this example, `Deck` extends `CardCollection`, which in turn extends `Object`. The `Object` class provides the default `equals` and `toString` methods, among other things.

Constructors are *not* inherited, but all other `public` attributes and methods are. The only additional method in `Deck`, at least for now, is a constructor. So you can create a `Deck` object like this:

```
Deck deck = new Deck("Deck");
```

The first line of the constructor uses `super`, which is a keyword that refers to the superclass of the current class. When `super` is used as a method, as in this example, it invokes the constructor of the superclass.

So in this case, `super` invokes the `CardCollection` constructor, which initializes the attributes `label` and `cards`. When it returns, the `Deck` constructor resumes and populates the (empty) `ArrayList` with `Card` objects.

That’s it for the `Deck` class. Next we need a way to represent a hand, which is the collection of cards held by a player, and a pile, which is a collection of

cards on the table. We could define two classes, one for hands and one for piles, but there is not much difference between them. So we'll use one class, called `Hand`, for both hands and piles. Here's what the definition looks like:

```
public class Hand extends CardCollection {

    public Hand(String label) {
        super(label);
    }

    public void display() {
        System.out.println(getLabel() + ": ");
        for (int i = 0; i < size(); i++) {
            System.out.println(getCard(i));
        }
        System.out.println();
    }
}
```

Like `Deck`, the `Hand` class extends `CardCollection`. So it inherits methods like `getLabel`, `size`, and `getCard`, which are used in `display`. `Hand` also provides a constructor, which invokes the constructor of `CardCollection`.

In summary, a `Deck` is just like a `CardCollection`, but it provides a different constructor. And a `Hand` is just like a `CardCollection`, but it provides an additional method, `display`.

## 14.3 Dealing Cards

To begin the game, we need to deal cards to each of the players. And during the game, we need to move cards between hands and piles. If we add the following method to `CardCollection`, it can meet both of these requirements:

```
public void deal(CardCollection that, int n) {
    for (int i = 0; i < n; i++) {
        Card card = popCard();
        that.addCard(card);
    }
}
```

The `deal` method removes cards from the collection it is invoked on, `this`, and adds them to the collection it gets as a parameter, `that`. The second parameter, `n`, is the number of cards to deal. We will use this method to implement `dealAll`, which deals (or moves) all of the remaining cards:

```
public void dealAll(CardCollection that) {  
    int n = cards.size();  
    deal(that, n);  
}
```

At this point, we can create a `Deck` and start dealing cards. Here's a simple example that deals five cards to a hand, and deals the rest into a draw pile:

```
Deck deck = new Deck("Deck");  
deck.shuffle();  
  
Hand hand = new Hand("Hand");  
deck.deal(hand, 5);  
hand.display();  
  
Hand drawPile = new Hand("Draw Pile");  
deck.dealAll(drawPile);  
System.out.printf("Draw Pile has %d cards.\n",  
    drawPile.size());
```

Because the deck is shuffled randomly, you should get a different hand each time you run this example. The output will look something like this:

```
Hand:  
5 of Diamonds  
Ace of Hearts  
6 of Clubs  
6 of Diamonds  
2 of Clubs  
  
Draw Pile has 47 cards.
```

If you are a careful reader, you might notice something strange about this example. Take another look at the definition of `deal`. Notice that the first parameter is supposed to be a `CardCollection`. But we invoked it like this:

```
Hand hand = new Hand("Hand");  
deck.deal(hand, 5);
```

The argument is a `Hand`, not a `CardCollection`. So why is this example legal?

It's because `Hand` is a subclass of `CardCollection`, so a `Hand` object is also considered to be a `CardCollection` object. If a method expects a `CardCollection`, you can give it a `Hand`, a `Deck`, or a `CardCollection`.

But it doesn't work the other way around: not every `CardCollection` is a `Hand`, so if a method expects a `Hand`, you have to give it a `Hand`, not a `CardCollection` or a `Deck`.

If it seems strange that an object can belong to more than one type, remember that this happens in real life too. Every cat is also a mammal, and every mammal is also an animal. But not every animal is a mammal, and not every mammal is a cat.

## 14.4 The Player Class

The `Deck` and `Hand` classes we have defined so far could be used for any card game; we have not yet implemented any of the rules specific to Crazy Eights. And that's probably a good thing, since it makes it easy to reuse these classes if we want to make another game in the future.

But now it's time to implement the rules. We'll use two classes: `Player`, which encapsulates player strategy, and `Eights`, which creates and maintains the state of the game. Here is the beginning of the `Player` definition:

```
public class Player {  
  
    private String name;  
    private Hand hand;  
  
    public Player(String name) {  
        this.name = name;  
        this.hand = new Hand(name);  
    }  
}
```

A `Player` has two `private` attributes: a name and a hand. The constructor takes the player's name as a string and saves it in an instance variable. In this example, we have to use `this` to distinguish between the instance variable and the parameter with the same name.

The primary method that `Player` provides is `play`, which decides which card to discard during each turn:

```
public Card play(Eights eights, Card prev) {
    Card card = searchForMatch(prev);
    if (card == null) {
        card = drawForMatch(eights, prev);
    }
    return card;
}
```

The first parameter is a reference to the `Eights` object that encapsulates the state of the game (coming up in the next section). The second parameter, `prev`, is the card on top of the discard pile.

`play` invokes two helper methods: `searchForMatch` and `drawForMatch`. Since we have not written them yet, this is an example of top-down design.

Here's `searchForMatch`, which looks in the player's hand for a card that matches the previously played card:

```
public Card searchForMatch(Card prev) {
    for (int i = 0; i < hand.size(); i++) {
        Card card = hand.getCard(i);
        if (cardMatches(card, prev)) {
            return hand.popCard(i);
        }
    }
    return null;
}
```

The strategy is pretty simple: the `for` loop searches for the first card that's legal to play and returns it. If there are no cards that match, it returns `null`. In that case, we have to draw cards until we get a match, which is what `drawForMatch` does:

```
public Card drawForMatch(Eights eights, Card prev) {
    while (true) {
        Card card = eights.drawCard();
        System.out.println(name + " draws " + card);
        if (cardMatches(card, prev)) {
            return card;
        }
        hand.addCard(card);
    }
}
```

The `while` loop runs until it finds a match (we'll assume for now that it always finds one). The loop uses the `Eights` object to draw a card. If it matches, `drawForMatch` returns the card. Otherwise it adds the card to the player's hand and repeats.

Both `searchForMatch` and `drawForMatch` use `cardMatches`, which is a static method, also defined in `Player`. This method is a straightforward translation of the rules of the game:

```
public static boolean cardMatches(Card card1, Card card2) {
    return card1.getSuit() == card2.getSuit()
        || card1.getRank() == card2.getRank()
        || card1.getRank() == 8;
}
```

Finally, `Player` provides a `score` method, which computes penalty points for cards left in a player's hand at the end of the game.

## 14.5 The Eights Class

In Section 13.2, we introduced top-down design. In this way of developing programs, we identify high-level goals, like shuffling a deck, and break them into smaller problems, like choosing a random element or swapping two elements.

In this section, we present **bottom-up design**, which goes the other way around: first we identify simple pieces we need and then we assemble them into more-complex algorithms.

Looking at the rules of Crazy Eights, we can identify some of the methods we'll need:

- Create the deck, the players, and the discard and draw piles. Deal the cards and set up the game. (**Eights** constructor)
- Check whether the game is over. (**isDone**)
- If the draw pile is empty, shuffle the discard pile and move the cards into the draw pile. (**reshuffle**)
- Draw a card, reshuffling the discard pile if necessary. (**drawCard**)
- Keep track of whose turn it is, and switch from one player to the next. (**nextPlayer**)
- Display the state of the game, and wait for the user before running the next turn. (**displayState**)

Now we can start implementing the pieces. Here is the beginning of the class definition for **Eights**, which encapsulates the state of the game:

```
public class Eights {  
  
    private Player one;  
    private Player two;  
    private Hand drawPile;  
    private Hand discardPile;  
    private Scanner in;
```

In this version, there are always two players. One of the exercises at the end of the chapter asks you to modify this code to handle more players. The **Eights** class also includes a draw pile, a discard pile, and a **Scanner**, which we will use to prompt the user after each turn.

The constructor for **Eights** initializes the instance variables and deals the cards, similar to Section 14.3. The next piece we'll need is a method that checks whether the game is over. If either hand is empty, we're done:

```
public boolean isDone() {  
    return one.getHand().isEmpty() || two.getHand().isEmpty();  
}
```



When the draw pile is empty, we have to shuffle the discard pile. Here is a method for that:

```
public void reshuffle() {  
    Card prev = discardPile.popCard();  
    discardPile.dealAll(drawPile);  
    discardPile.addCard(prev);  
    drawPile.shuffle();  
}
```

The first line saves the top card from `discardPile`. The next line transfers the rest of the cards to `drawPile`. Then we put the saved card back into `discardPile` and shuffle `drawPile`. We can use `reshuffle` as part of the `draw` method:

```
public Card drawCard() {  
    if (drawPile.isEmpty()) {  
        reshuffle();  
    }  
    return drawPile.popCard();  
}
```

The `nextPlayer` method takes the current player as a parameter and returns the player who should go next:

```
public Player nextPlayer(Player current) {  
    if (current == one) {  
        return two;  
    } else {  
        return one;  
    }  
}
```

The last method from our bottom-up design is `displayState`. It displays the hand of each player, the contents of the discard pile, and the number of cards in the draw pile. Finally, it waits for the user to press the **Enter** key:

```
public void displayState() {
    one.display();
    two.display();
    discardPile.display();
    System.out.println("Draw pile:");
    System.out.println(drawPile.size() + " cards");
    in.nextLine();
}
```

Using these pieces, we can write `takeTurn`, which executes one player's turn. It reads the top card off the discard pile and passes it to `player.play`, which you saw in the previous section. The result is the card the player chose, which is added to the discard pile:

```
public void takeTurn(Player player) {
    Card prev = discardPile.lastCard();
    Card next = player.play(this, prev);
    discardPile.addCard(next);

    System.out.println(player.getName() + " plays " + next);
    System.out.println();
}
```

Finally, we use `takeTurn` and the other methods to write `playGame`:

```
public void playGame() {
    Player player = one;

    // keep playing until there's a winner
    while (!isDone()) {
        displayState();
        takeTurn(player);
        player = nextPlayer(player);
    }

    // display the final score
    one.displayScore();
    two.displayScore();
}
```

Done! The result of bottom-up design is similar to top-down: we have a high-level method that calls helper methods. The difference is the development process we used to arrive at this solution.

## 14.6 Class Relationships

This chapter demonstrates two common relationships between classes:

**composition:** Instances of one class contain references to instances of another class. For example, an instance of `Eights` contains references to two `Player` objects, two `Hand` objects, and a `Scanner`.

**inheritance:** One class extends another class. For example, `Hand` extends `CardCollection`, so every instance of `Hand` is also a `CardCollection`.

Composition is also known as a **HAS-A** relationship, as in “`Eights` has a `Scanner`”. Inheritance is also known as an **IS-A** relationship, as in “`Hand` is a `CardCollection`”. This vocabulary provides a concise way to talk about an object-oriented design.

There is also a standard way to represent these relationships graphically in UML class diagrams. As you saw in Section 10.7, the UML representation of a class is a box with three sections: the class name, the attributes, and the methods. The latter two sections are optional when showing relationships.

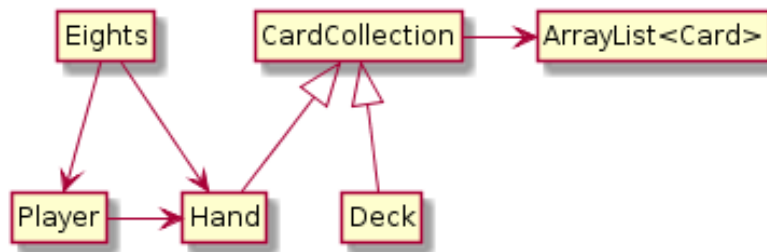


Figure 14.1: UML diagram for the classes in this chapter.

Relationships between classes are represented by arrows: composition arrows have a standard arrow head, and inheritance arrows have a hollow triangle head (usually pointing up). Figure 14.1 shows the classes defined in this chapter and the relationships among them.

UML is an international standard, so almost any software engineer in the world could look at this diagram and understand our design. And class diagrams are only one of many graphical representations defined in the UML standard.

## 14.7 Vocabulary

**inheritance:** The ability to define a new class that has the same instance variables and methods of an existing class.

**subclass:** A class that inherits from, or extends, an existing class.

**superclass:** An existing class that is extended by another class.

**bottom-up design:** A way of developing programs by identifying simple pieces, implementing them first, and then assembling them into more-complex algorithms.

**HAS-A:** A relationship between two classes in which one class “has” an instance of another class as one of its attributes.

**IS-A:** A relationship between two classes in which one class extends another class; the subclass “is” an instance of the superclass.

## 14.8 Exercises

The code for this chapter is in the *ch14* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise 14.1** Design a better strategy for the `Player.play` method. For example, if there are multiple cards you can play, and one of them is an 8, you might want to play the 8.

Think of other ways you can minimize penalty points, such as playing the highest-ranking cards first. Write a new class that extends `Player` and overrides `play` to implement your strategy.

**Exercise 14.2** Write a loop that plays the game 100 times and keeps track of how many times each player wins. If you implemented multiple strategies in the previous exercise, you can play them against each other to evaluate which one works best.

*Hint:* Design a **Genius** class that extends **Player** and overrides the **play** method, and then replace one of the players with a **Genius** object.

**Exercise 14.3** One limitation of the program we wrote in this chapter is that it handles only two players. Modify the **Eights** class to create an **ArrayList** of players, and modify **nextPlayer** to select the next player.

**Exercise 14.4** When we designed the program for this chapter, we tried to minimize the number of classes. As a result, we ended up with a few awkward methods. For example, **cardMatches** is a static method in **Player**, but it would be more natural if it were an instance method in **Card**.

The problem is that **Card** is supposed to be useful for any card game, not just Crazy Eights. You can solve this problem by adding a new class, **EightsCard**, that extends **Card** and provides a method, **match**, that checks whether two cards match according to the rules of Crazy Eights.

At the same time, you could create a new class, **EightsHand**, that extends **Hand** and provides a method, **scoreHand**, that adds up the scores of the cards in the hand. And while you're at it, you could add a method named **scoreCard** to **EightsCard**.

Whether or not you actually make these changes, draw a UML class diagram that shows this alternative object hierarchy.

# Chapter 15

## Arrays of Arrays

The last three chapters of this book use 2D graphics to illustrate more advanced object-oriented concepts. If you haven't yet read Appendix C, you might want to read it now and become familiar with the `Canvas`, `Color`, and `Graphics` classes from the `java.awt` package. In this chapter, we use these classes to draw images and animations, and to run graphical simulations.

### 15.1 Conway's Game of Life

The Game of Life, or GoL for short, was developed by John Conway and popularized in 1970 in Martin Gardner's column in *Scientific American*. Conway calls it a “zero-player game” because no players are needed to choose strategies or make decisions. After you set up the initial conditions, you watch the game play itself. That turns out to be more interesting than it sounds; you can read about it at [https://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway's_Game_of_Life).

The game board is a 2D grid of square cells. Each cell is either “alive” or “dead”; the color of the cell indicates its state. Figure 15.1 shows an example grid configuration; the five black cells are alive.

The game proceeds in time steps, during which each cell interacts with its neighbors in the eight adjacent cells. At each time step, the following rules are applied:

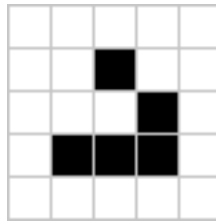


Figure 15.1: A “Glider” in the Game of Life.

- A live cell with fewer than two live neighbors dies, as if by underpopulation.
- A live cell with more than three live neighbors dies, as if by overpopulation.
- A dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Notice some consequences of these rules. If you start with a single live cell, it dies. If all cells are dead, no cells come to life. But if you have four cells in a square, they keep each other alive, so that’s a “stable” configuration.

Another initial configuration is shown in Figure 15.2. If you start with three horizontal cells, the center cell lives, the left and right cells die, and the top and bottom cells come to life. The result after the first time step is three vertical cells.

During the next time step, the center cell lives, the top and bottom cells die, and the left and right cells come to life. The result is three horizontal cells, so we’re back where we started, and the cycle repeats forever.

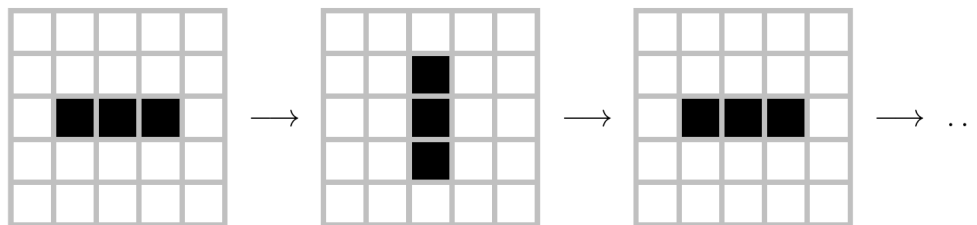


Figure 15.2: A “Blinker” in the Game of Life.

Patterns like this are called “periodic”, because they repeat after a period of two or more time steps. But they are also considered stable, because the total number of live cells doesn’t grow over time.

Most simple starting configurations either die out quickly or reach a stable configuration. But there are a few starting conditions that display remarkable complexity. One of those is the R-pentomino (<https://www.conwaylife.com/wiki/R-pentomino>), which starts with only five cells, runs for 1,103 time steps, and ends in a stable configuration with 116 live cells.

In the following sections, we'll implement the Game of Life in Java. We'll first implement the cells, then the grid of cells, and finally the game itself.

## 15.2 The Cell Class

When drawing a cell, we'll need to know its location on the screen and size in pixels. To represent the location, we use the **x** and **y** coordinates of the upper-left corner. And to represent the size, we use an integer, **size**.

To represent the state of a cell, we use an integer, **state**, which is 0 for dead cells and 1 for live cells. We could use a **boolean** instead, but it's good practice to design classes to be reusable (e.g., for other games that have more states).

Here is a **Cell** class that declares these instance variables:

```
public class Cell {  
    private final int x;  
    private final int y;  
    private final int size;  
    private int state;  
}
```

Notice that **x**, **y**, and **size** are constants. Once the cell is created, we don't want it to move or change size. But **state** can and should change, so it is not a constant.

The next step is to write a constructor. Here's one that takes **x**, **y**, and **size** as parameters, and sets **state** to a default value:

```
public Cell(int x, int y, int size) {  
    this.x = x;  
    this.y = y;  
    this.size = size;  
    this.state = 0;  
}
```



The following method draws a cell. Like the `paint` method in Appendix C, it takes a graphics context as a parameter:

```
public static final Color[] COLORS = {Color.WHITE, Color.BLACK};

public void draw(Graphics g) {
    g.setColor(COLORS[state]);
    g.fillRect(x + 1, y + 1, size - 1, size - 1);
    g.setColor(Color.LIGHT_GRAY);
    g.drawRect(x, y, size, size);
}
```

The `draw` method uses the state of the cell to select a color from an array of `Color` objects. Then it uses `fillRect` to draw the center of the cell and `drawRect` to draw a light-gray border.

We also need methods to get and set the cell's state. We could just provide `getState` and `setState`, but the code will be more readable if we provide methods customized for the Game of Life:

```
public boolean isOff() {
    return state == 0;
}

public boolean isOn() {
    return state == 1;
}

public void turnOff() {
    state = 0;
}

public void turnOn() {
    state = 1;
}
```

## 15.3 Two-Dimensional Arrays

To represent a grid of cells, we can use a **multidimensional array**. To create a 2D array, we specify the number of rows and columns:

```
int rows = 4;
int cols = 3;
Cell[] [] array = new Cell[rows][cols];
```

The result is an array with four rows and three columns. Initially, the elements of the array are `null`. We can fill the array with `Cell` objects like this:

```
for (int r = 0; r < rows; r++) {
    int y = r * size;
    for (int c = 0; c < cols; c++) {
        int x = c * size;
        array[r][c] = new Cell(x, y, size);
    }
}
```

The loop variables `r` and `c` are the row and column indexes of the cells. The variables `x` and `y` are the coordinates, respectively. For example, if `size` is 10 pixels, the cell at index (1, 2) would be at coordinates (10, 20) on the screen.

In Java, a 2D array is really an array of arrays. You can think of it as an array of rows, where each row is an array. Figure 15.3 shows what it looks like.

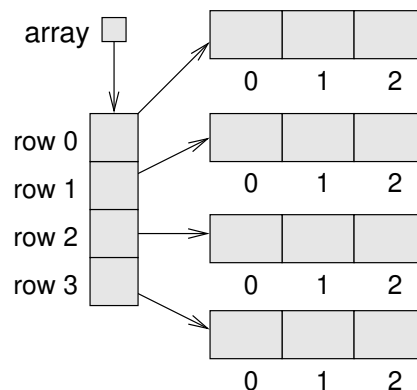


Figure 15.3: Storing rows and columns with a 2D array.

When we write `array[r][c]`, Java uses the first index to select a row and the second index to select an element from the row. This way of representing 2D data is known as **row-major order**.

## 15.4 The GridCanvas Class

Now that we have a `Cell` class and a way to represent a 2D array of cells, we can write a class to represent a grid of cells. We encapsulate the code from the previous section and generalize it to construct a grid with any number of rows and columns:

```
public class GridCanvas extends Canvas {
    private Cell[][] array;

    public GridCanvas(int rows, int cols, int size) {
        array = new Cell[rows][cols];
        for (int r = 0; r < rows; r++) {
            int y = r * size;
            for (int c = 0; c < cols; c++) {
                int x = c * size;
                array[r][c] = new Cell(x, y, size);
            }
        }

        // set the canvas size
        setSize(cols * size, rows * size);
    }
}
```

Using vocabulary from the previous chapter, `GridCanvas` “is a” `Canvas` that “has a” 2D array of cells. By extending the `Canvas` class from `java.awt`, we inherit methods for drawing graphics on the screen.

In fact, the code is surprisingly straightforward: to draw the grid, we simply draw each cell. We use nested `for` loops to traverse the 2D array:

```
public void draw(Graphics g) {
    for (Cell[] row : array) {
        for (Cell cell : row) {
            cell.draw(g);
        }
    }
}
```

The outer loop traverses the rows; the inner loop traverses the cells in each row. You can almost read this method in English: “For each **row** in the **array**, and for each **cell** in the **row**, draw the **cell** in the graphics context.” Each cell contains its coordinates and size, so it knows how to draw itself.

Classes that extend **Canvas** are supposed to provide a method called **paint** that “paints” the contents of the **Canvas**. It gets invoked when the **Canvas** is created and anytime it needs to be redrawn; for example, when its window is moved or resized.

Here’s the **paint** method for **GridCanvas**. When the window management system calls **paint**, **paint** calls **draw**, which draws the cells:

```
public void paint(Graphics g) {  
    draw(g);  
}
```

## 15.5 Other Grid Methods

In addition to **draw** and **paint**, the **GridCanvas** class provides methods for working with the grid itself. **numRows** and **numCols** return the number of rows and columns. We can get this information from the 2D array, using **length**:

```
public int numRows() {  
    return array.length;  
}  
  
public int numCols() {  
    return array[0].length;  
}
```

Because we are using row-major order, the 2D array is an array of rows. **numRows** simply returns the length of the rows array. **numCols** returns the length of the first row, which is the number of columns. Since the rows all have the same length, we have to check only one.

**GridCanvas** also provides a method that gets the **Cell** at a given location, and for convenience when starting the game, a method that turns on the **Cell** at a given location.

```
public Cell getCell(int r, int c) {  
    return array[r][c];  
}  
  
public void turnOn(int r, int c) {  
    array[r][c].turnOn();  
}
```

## 15.6 Starting the Game

Now we're ready to implement the game. To encapsulate the rules of GoL, we define a class named `Conway`. The `Conway` class “has a” `GridCanvas` that represents the state of the game.

This constructor makes a `GridCanvas` with 5 rows and 10 columns, with cells that are 20 pixels wide and high. It then sets up the initial conditions:

```
public class Conway {  
    private GridCanvas grid;  
  
    public Conway() {  
        grid = new GridCanvas(5, 10, 20);  
        grid.turnOn(2, 1);  
        grid.turnOn(2, 2);  
        grid.turnOn(2, 3);  
        grid.turnOn(1, 7);  
        grid.turnOn(2, 7);  
        grid.turnOn(3, 7);  
    }  
}
```

Before we implement the rest of the game, we'll write a `main` method that creates a `Conway` object and displays it. We can use this method to test `Cell` and `GridCanvas`, and to develop the other methods we need:

```
public static void main(String[] args) {  
    String title = "Conway's Game of Life";  
    Conway game = new Conway();  
    JFrame frame = new JFrame(title);  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    frame.setResizable(false);  
    frame.add(game.grid);  
    frame.pack();  
    frame.setVisible(true);  
    game.mainloop();  
}
```

After constructing the `game` object, `main` constructs a `JFrame`, which creates a window on the screen. The `JFrame` is configured<sup>1</sup> to exit the program when closed. Resizing the window is disabled.

`main` then adds the `GridCanvas` inside the frame, resizes (“packs”) the frame to fit the canvas, and makes the frame visible. Figure 15.4 shows the result.

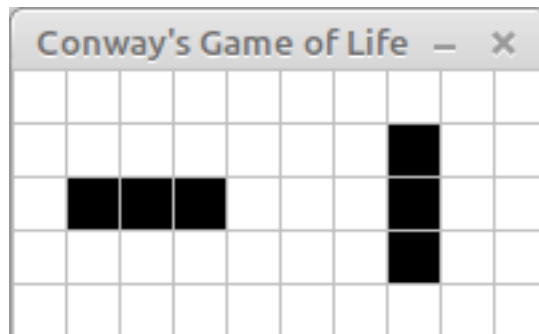


Figure 15.4: Screenshot of the initial Conway application.

## 15.7 The Simulation Loop

At the end of `main`, we call `mainloop`, which uses a `while` loop to simulate the time steps of the Game of Life. Here’s a rough draft of this method:

---

<sup>1</sup> We are using `JFrame` (in `javax.swing`) instead of `Frame` (in `java.awt`) for simplicity. `Frame` does not provide a default close operation; it requires you to implement a method to be called when the user closes the window.

```
private void mainloop() {  
    while (true) {  
        this.update();  
        grid.repaint();  
        Thread.sleep(500);    // compiler error  
    }  
}
```

During each time step, we update the state of the game and repaint the `grid`. We will present the `update` method in Section 15.10.

`repaint` comes from the `Canvas` class. By default, it calls the `paint` method we provided, which calls `draw`. The reason we use it here is that `repaint` does not require a `Graphics` object as a parameter.

`Thread.sleep(500)` causes the program to “sleep” for 500 milliseconds, or a half second. Otherwise, the program would run so fast we would not be able to see the animation.

There’s just one problem: compiling this code results in the error “unreported exception `InterruptedException`”. This message means we need to do some exception handling.

## 15.8 Exception Handling

So far, the only exceptions you have seen are run-time errors like “array index out of bounds” and “null pointer”. When one of these exceptions occurs, Java displays a message and ends the program.

If you don’t want the program to end, you can handle exceptions with a `try-catch` statement. The syntax is similar to an `if-else` statement, and the logic is, too. Here’s what it looks like:

```
try {  
    Thread.sleep(500);  
} catch (InterruptedException e) {  
    // do nothing  
}
```

First, Java runs the code in the try block, which calls `Thread.sleep` in this example. If an `InterruptedException` occurs during the try block, Java executes the catch block. In this example, the catch block contains a comment, so it doesn't do anything.

If a different exception occurs during the try block, Java does whatever it would do otherwise, which is probably to display a message and end the program. If no exceptions occur during the try block, the catch block doesn't run and the program continues.

In this example, the effect of the `try-catch` statement is to ignore an “interrupted” exception if it occurs. As an alternative, we could use the catch block to display a customized message, end the program, or handle the exception in whatever way is appropriate. For example, if user input causes an exception, we could catch the exception and prompt the user to try again later.

There's more to learn about exception handling. You can read about exceptions in the Java tutorials (see <https://thinkjava.org/exceptions>).

## 15.9 Counting Neighbors

Now that you know about `try` and `catch`, we can use them to implement a useful method in `GridCanvas`. Part of the GoL logic is to count the number of live neighbors. Most cells have eight neighbors, as shown in Figure 15.5.

However, cells on the edges and in the corners have fewer neighbors. If we try to count all possible neighbors, we'll go out of bounds. The following method uses a `try-catch` statement to deal with these special cases:

```
public int test(int r, int c) {
    try {
        if (array[r][c].isOn()) {
            return 1;
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        // cell doesn't exist
    }
    return 0;
}
```



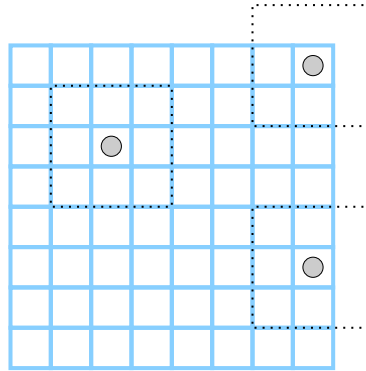


Figure 15.5: Cells in the interior of the grid have eight neighbors. Cells in the corners and along the edges have fewer neighbors.

The `test` method takes a row index, `r`, and a column index, `c`. It tries to look up the `Cell` at that location. If both indexes are in bounds, the `Cell` exists. In that case, `test` returns 1 if the `Cell` is on. Otherwise, it skips the catch block and returns 0.

If either index is out of bounds, the array lookup throws an exception, but the catch block ignores it. Then `test` resumes and returns 0. So the non-existent cells around the perimeter are considered to be off.

Now we can use `test` to implement `countAlive`, which takes a grid location, `(r, c)`, and returns the number of live neighbors surrounding that location:

```
private int countAlive(int r, int c) {
    int count = 0;
    count += grid.test(r - 1, c - 1);
    count += grid.test(r - 1, c);
    count += grid.test(r - 1, c + 1);
    count += grid.test(r, c - 1);
    count += grid.test(r, c + 1);
    count += grid.test(r + 1, c - 1);
    count += grid.test(r + 1, c);
    count += grid.test(r + 1, c + 1);
    return count;
}
```

Because `test` handles “out of bounds” exceptions, `countAlive` works for any values of `r` and `c`.

## 15.10 Updating the Grid

Now we are ready to write `update`, which gets invoked each time through the simulation loop. It uses the GoL rules to compute the state of the grid after the next time step:

```
public void update() {  
    int[] [] counts = countNeighbors();  
    updateGrid(counts);  
}
```

The rules of GoL specify that you have to update the cells “simultaneously”; that is, you have to count the neighbors for all cells before you can update any of them.

We do that by traversing the grid twice: first, `countNeighbors` counts the live neighbors for each cell and puts the results in an array named `counts`; second, `updateGrid` updates the cells. Here’s `countNeighbors`:

```
private int[] [] countNeighbors() {  
    int rows = grid.numRows();  
    int cols = grid.numCols();  
  
    int[] [] counts = new int[rows][cols];  
    for (int r = 0; r < rows; r++) {  
        for (int c = 0; c < cols; c++) {  
            counts[r][c] = countAlive(r, c);  
        }  
    }  
    return counts;  
}
```

`countNeighbors` traverses the cells in the grid and uses `countAlive` from the previous section to count the neighbors. The return value is a 2D array of integers with the same size as `grid`. Figure 15.6 illustrates an example.

In contrast to the `draw` method of `GridCanvas`, which uses enhanced `for` loops, `countNeighbors` uses standard `for` loops. The reason is that, in this example, we need the indexes `r` and `c` to store the neighbor counts.



helper methods not intended to be invoked from outside the class. `updateCell` is also `static`, because it does not depend on `grid`.

Now our implementation of the Game of Life is complete. We think it's pretty fun, and we hope you agree. But more importantly, this example is meant to demonstrate the use of 2D arrays and an object-oriented design that's a little more substantial than in previous chapters.

## 15.11 Vocabulary

**multidimensional array:** An array with more than one dimension; a 2D array is an “array of arrays”.

**row-major order:** Storing data in a 2D array, first by rows and then by columns.

## 15.12 Exercises

The code for this chapter is in the *ch15* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise 15.1** In `GridCanvas`, write a method named `countOn` that returns the total number of cells that are “on”. This method can be used, for example, to track the population in Game of Life over time.

**Exercise 15.2** In our version of the Game of Life, the grid has a finite size. As a result, moving objects such as Gliders either crash into the wall or go out of bounds.

An interesting variation of the Game of Life is a “toroidal” grid, meaning that the cells “wrap around” on the edges. Modify the `test` method of `GridCanvas` so that the coordinates `r` and `c` map to the opposite side of the grid if they are too low or too high.

Run your code with a Glider (see Figure 15.1) to see if it works. You can initialize the Glider by modifying the constructor in the `Conway` class, or by reading it from a file (see the next exercise).

**Exercise 15.3** The LifeWiki site (<https://conwaylife.com/wiki/>) has a fascinating collection of patterns for the Game of Life. These patterns are stored in a file format that is easy to read, in files with the suffix “.cells”.

For example, here is an 8 x 10 grid with a Glider near the upper-left corner:

```
!Name: Glider
.....
..0.....
...0.....
.000.....
.....
.....
.....
.....
```

Lines that begin with ! are comments and should be ignored. The rest of the file describes the grid, row by row. A period represents a dead cell, and an uppercase 0 represents a live cell. See <https://conwaylife.com/wiki/Plaintext> for more examples.

1. Create a plain text file with the contents shown above, and save the file as *glider.cells* in the same directory as your code.
2. Define a constructor for the `Conway` class that takes a string representing the name (or path) of a “.cells” file. Here is a starting point:

```
public Conway(String path) {
    File file = new File(path);
    Scanner scan = new Scanner(file);
}
```

3. Modify the `main` method to invoke the constructor as follows:

```
Conway game = new Conway("glider.cells");
```

4. Handle the `FileNotFoundException` that may be thrown when creating a `Scanner` for a `File` by invoking `printStackTrace` on the exception object and calling `System.exit()` with a status of 1, indicating an error.
5. Continue implementing the constructor by reading all non-comment lines into an `ArrayList` via `hasNextLine` and `nextLine` of the `Scanner`.

6. Determine the number of rows and columns of the grid by examining the `ArrayList` contents.
7. Create and initialize a `GridCanvas` based on the `ArrayList`.

Once your constructor is working, you will be able to run many of the patterns on the LifeWiki. You might want to add a margin of empty cells around the initial pattern, to give it room to grow.

**Exercise 15.4** Some files on the LifeWiki use “run-length encoding” (RLE) instead of plain text. The basic idea of RLE is to describe the number of dead and alive cells, rather than type out each individual cell.

For example, *glider.cells* from the previous exercise could be represented this way with RLE:

```
#C Name: Glider
x = 10, y = 8
$2bo$3bo$b3o!
```

The first line specifies `x` (the number of columns) and `y` (the number of rows). Subsequent lines consist of the letters `b` (dead), `o` (alive), and `$` (end of line), optionally preceded by a count. The pattern ends with `!`, after which any remaining file contents are ignored.

Lines beginning with `#` have special meaning and are not part of the pattern. For example, `#C` is a comment line. You can read more about RLE format on <https://conwaylife.com/wiki/RLE>.

1. Create a plain text file with the preceding contents, and save the file as *glider.rle* in the same directory as your code.
2. Modify your constructor from the previous exercise to check the last three characters of the `path`. If they are `"rle"`, then you will need to process the file as RLE. Otherwise, assume the file is in “.cells” format.
3. In the end, your constructor should be able to read and initialize grids in both formats. Test your constructor by modifying the `main` method to read different files.



# Chapter 16

## Reusing Classes

In Chapter 15, we developed classes to implement Conway’s Game of Life. We can reuse the `Cell` and `GridCanvas` classes to implement other simulations. One of the most interesting zero-player games is *Langton’s Ant*, which models an “ant” that walks around a grid. The ant follows only two simple rules:

1. If the ant is on a white cell, it turns to the right, makes the cell black, and moves forward.
2. If the ant is on a black cell, it turns to the left, makes the cell white, and moves forward.

Because the rules are simple, you might expect the ant to do something simple, like make a square or repeat a simple pattern. But starting on a grid with all white cells, the ant makes more than 10,000 steps in a seemingly random pattern before it settles into a repeating loop of 104 steps. You can read more about it at [https://en.wikipedia.org/wiki/Langton’s\\_ant](https://en.wikipedia.org/wiki/Langton's_ant).

In this chapter, we present an implementation of Langton’s Ant and use it to demonstrate more advanced object-oriented techniques.

### 16.1 Langton’s Ant

We begin by defining a `Langton` class that has a grid and information about the ant. The constructor takes the grid dimensions as parameters:



```
public class Langton {
    private GridCanvas grid;
    private int xpos;
    private int ypos;
    private int head; // 0=North, 1=East, 2=South, 3=West

    public Langton(int rows, int cols) {
        grid = new GridCanvas(rows, cols, 10);
        xpos = rows / 2;
        ypos = cols / 2;
        head = 0;
    }
}
```

`grid` is a `GridCanvas` object, which represents the state of the cells. `xpos` and `ypos` are the coordinates of the ant, and `head` is the “heading” of the ant; that is, which direction it is facing. `head` is an integer with four possible values, where 0 means the ant is facing “north” (i.e., toward the top of the screen), 1 means “east”, etc.

Here’s an `update` method that implements the rules for Langton’s Ant:

```
public void update() {
    flipCell();
    moveAnt();
}
```

The `flipCell` method gets the `Cell` at the ant’s location, figures out which way to turn, and changes the state of the cell:

```
private void flipCell() {
    Cell cell = grid.getCell(xpos, ypos);
    if (cell.isOff()) {
        head = (head + 1) % 4;    // turn right
        cell.turnOn();
    } else {
        head = (head + 3) % 4;    // turn left
        cell.turnOff();
    }
}
```

We use the remainder operator, %, to make `head` wrap around: if `head` is 3 and we turn right, it wraps around to 0; if `head` is 0 and we turn left, it wraps around to 3.

Notice that to turn right, we add 1 to `head`. To turn left, we could subtract 1, but `-1 % 4` is `-1` in Java. So we add 3 instead, since one left turn is the same as three right turns.

The `moveAnt` method moves the ant forward one square, using `head` to determine which way is forward:

```
private void moveAnt() {
    if (head == 0) {
        ypos -= 1;
    } else if (head == 1) {
        xpos += 1;
    } else if (head == 2) {
        ypos += 1;
    } else {
        xpos -= 1;
    }
}
```

Here is the main method we use to create and display the `Langton` object:

```
public static void main(String[] args) {
    String title = "Langton's Ant";
    Langton game = new Langton(61, 61);
    JFrame frame = new JFrame(title);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.add(game.grid);
    frame.pack();
    frame.setVisible(true);
    game.mainloop();
}
```

Most of this code is the same as the `main` we used to create and run `Conway`, in Section 15.6. It creates and configures a `JFrame` and runs `mainloop`.

And that's everything! If you run this code with a grid size of 61 x 61 or larger, you will see the ant eventually settle into a repeating pattern.

Because we designed `Cell` and `GridCanvas` to be reusable, we didn't have to modify them at all. However, we now have two copies of `main` and `mainloop`—one in `Conway`, and one in `Langton`.

## 16.2 Refactoring

Whenever you see repeated code like `main`, you should think about ways to remove it. In Chapter 14, we used inheritance to eliminate repeated code. We'll do something similar with `Conway` and `Langton`.

First, we define a superclass named `Automaton`, in which we will put the code that `Conway` and `Langton` have in common:

```
public class Automaton {
    private GridCanvas grid;

    public void run(String title, int rate) {
        JFrame frame = new JFrame(title);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(false);
        frame.add(this.grid);
        frame.pack();
        frame.setVisible(true);
        this.mainloop(rate);
    }
}
```

`Automaton` declares `grid` as an instance variable, so every `Automaton` “has a” `GridCanvas`. It also provides `run`, which contains the code that creates and configures the `JFrame`.

The `run` method takes two parameters: the window `title` and the frame `rate`; that is, the number of time steps to show per second. It uses `title` when creating the `JFrame`, and it passes `rate` to `mainloop`:

```
private void mainloop(int rate) {
    while (true) {

        // update the drawing
        this.update();
        grid.repaint();

        // delay the simulation
        try {
            Thread.sleep(1000 / rate);
        } catch (InterruptedException e) {
            // do nothing
        }
    }
}
```

`mainloop` contains the code you first saw in Section 15.7. It runs a `while` loop forever (or until the window closes). Each time through the loop, it runs `update` to update `grid` and then `repaint` to redraw the grid.

Then it calls `Thread.sleep` with a delay that depends on `rate`. For example, if `rate` is 2, we should draw two frames per second, so the delay is a half second, or 500 milliseconds.

This process of reorganizing existing code, without changing its behavior, is known as **refactoring**. We're almost finished; we just need to redesign `Conway` and `Langton` to extend `Automaton`.

## 16.3 Abstract Classes

If we were not planning to implement any other zero-person games, we could leave well enough alone. But there are a few problems with the current design:

- The `grid` attribute is `private`, making it inaccessible in `Conway` and `Langton`. We could make it `public`, but then other (unrelated) classes would have access to it as well.
- The `Automaton` class has no constructors, and even if it did, there would be no reason to create an instance of this class.

- The `Automaton` class does not provide an implementation of `update`. In order to work properly, subclasses need to provide one.

Java provides language features to solve these problems:

- We can make the `grid` attribute `protected`, which means it's accessible to subclasses but not other classes.
- We can make the class `abstract`, which means it cannot be instantiated. If you attempt to create an object for an abstract class, you will get a compiler error.
- We can declare `update` as an `abstract` method, meaning that it must be overridden in subclasses. If the subclass does not override an abstract method, you will get a compiler error.

Here's what `Automaton` looks like as an abstract class (using the methods `mainloop` and `run` from Section 16.2):

```
public abstract class Automaton {
    protected GridCanvas grid;

    public abstract void update();

    private void mainloop(int rate) {
        // this method invokes update
    }

    public void run(String title, int rate) {
        // this method invokes mainloop
    }
}
```

Notice that the `update` method has no body. The declaration specifies the name, arguments, and return type. But it does not provide an implementation, because it is an abstract method.

Notice also the word `abstract` on the first line, which declares that `Automaton` is an abstract class. In order to have any abstract methods, a class must be declared as abstract.

Any class that extends `Automaton` must provide an implementation of `update`; the declaration here allows the compiler to check.

Here's what `Conway` looks like as a subclass of `Automaton`:

```
public class Conway extends Automaton {  
  
    // same methods as before, except mainloop is removed  
  
    public static void main(String[] args) {  
        String title = "Conway's Game of Life";  
        Conway game = new Conway();  
        game.run(title, 2);  
    }  
}
```

`Conway` extends `Automaton`, so it inherits the `protected` instance variable `grid` and the methods `mainloop` and `run`. But because `Automaton` is abstract, `Conway` has to provide `update` and a constructor (which it has already).

Abstract classes are essentially “incomplete” class definitions that specify methods to be implemented by subclasses. But they also provide attributes and methods to be inherited, thus eliminating repeated code.

## 16.4 UML Diagram

At the beginning of the chapter, we had three classes: `Cell`, `GridCanvas`, and `Conway`. We then developed `Langton`, which had almost the same `main` and `mainloop` methods as `Conway`. So we refactored the code and created `Automaton`. Figure 16.1 summarizes the final design.

The diagram shows three examples of inheritance: `Conway` is an `Automaton`, `Langton` is an `Automaton`, and `GridCanvas` is a `Canvas`. It also shows two examples of composition: `Automaton` has a `GridCanvas`, and `GridCanvas` has a 2D array of `Cells`.

The diagram also shows that `Automaton` uses `JFrame`, `GridCanvas` uses `Graphics`, and `Cell` uses `Graphics` and `Color`.

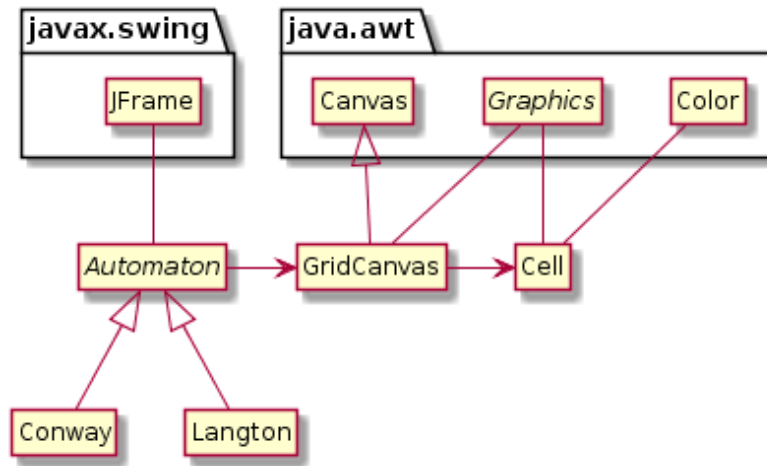


Figure 16.1: UML class diagram of Conway and Langton applications.

*Automaton* is in italics to indicate that it is an abstract class. As it happens, *Graphics* is an abstract class, too.

Conway and Langton are **concrete classes**, because they provide an implementation for all of their methods. In particular, they implement the **update** method that was declared **abstract** in *Automaton*.

One of the challenges of object-oriented programming is keeping track of a large number of classes and the relationships between them. UML class diagrams can help.

## 16.5 Vocabulary

**refactor:** To restructure or reorganize existing source code without changing its behavior.

**abstract class:** A class that is declared as **abstract**; it cannot be instantiated, and it may (or may not) include abstract methods.

**concrete class:** A class that is *not* declared as **abstract**; each of its methods must have an implementation.

## 16.6 Exercises

The code for this chapter is in the *ch16* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise 16.1** The last section of this chapter introduced **Automaton** as an abstract class and rewrote **Conway** as a subclass of **Automaton**. Now it's your turn: rewrite **Langton** as a subclass of **Automaton**, removing the code that's no longer needed.

**Exercise 16.2** Mathematically speaking, Game of Life and Langton's Ant are *cellular automata*. "Cellular" means it has cells, and "automaton" means it runs itself. See [https://en.wikipedia.org/wiki/Cellular\\_automaton](https://en.wikipedia.org/wiki/Cellular_automaton) for more discussion.

Implement another cellular automaton of your choice. You may have to modify **Cell** and/or **GridCanvas**, in addition to extending **Automaton**. For example, Brian's Brain ([https://en.wikipedia.org/wiki/Brian's\\_Brain](https://en.wikipedia.org/wiki/Brian's_Brain)) requires three states: "on", "dying", and "off".





# Chapter 17

## Advanced Topics

When we first looked at inheritance in Chapter 14, our purpose was to avoid duplicating code. We noticed that “decks of cards” and “hands of cards” had common functionality, and we designed a `CardCollection` class to provide it. This technique is an example of **generalization**. By generalizing the code, we were able to reuse it in the `Deck` and `Hand` classes.

In Chapter 15, we looked at inheritance from a different point of view. When designing `GridCanvas` to represent a grid of cells, we extended `Canvas` and overrode its `paint` method. This design is an example of **specialization**. Using the code provided by `Canvas`, we created a specialized subclass with minimal additional code.

We didn’t write the code for `Canvas`; it’s part of the Java library. But we were able to customize it for our own purposes. In fact, the `Canvas` class was explicitly designed to be extended.

In this chapter, we’ll explore the concept of inheritance more fully and present event-driven programming. We’ll continue to develop graphical simulations as a running example, but this time in varying shapes and colors!

### 17.1 Polygon Objects

The word polygon means “many angles”; the most basic polygons are triangles (three angles), rectangles (four angles), pentagons (five angles), and so forth.

Polygons are an important part of computer graphics because they are used to compose more complex images.

Java provides a `Polygon` class (in `java.awt`) that we can use to represent and draw polygons. The following code creates an empty `Polygon` and adds three points, forming a triangle:

```
Polygon p = new Polygon();
p.addPoint(57, 110);
p.addPoint(100, 35);
p.addPoint(143, 110);
```

Internally, `Polygon` objects have three attributes:

- `public int npoints;`     `// total number of points`
- `public int[] xpoints;`   `// array of X coordinates`
- `public int[] ypoints;`   `// array of Y coordinates`

When a `Polygon` is created, `npoints` is 0 and the two arrays are initialized with length 4. As points are added, `npoints` is incremented. If `npoints` exceeds the length of the arrays, larger arrays are created, and the previous values are copied over (similar to how `ArrayList` works).

The `Polygon` class provides many useful methods, like `contains`, `intersects`, and `translate`. We'll get to those later, but first we're going to do some specialization.

## 17.2 Adding Color

Specialization is useful for adding new features to an existing class, especially when you can't (or don't want to) change its design. For example, we can extend the `Polygon` class by adding a `draw` method and a `Color` attribute:

```
public class DrawablePolygon extends Polygon {
    protected Color color;

    public DrawablePolygon() {
        super();
        color = Color.GRAY;
    }

    public void draw(Graphics g) {
        g.setColor(color);
        g.fillPolygon(this);
    }
}
```

As a reminder, constructors are not inherited when you extend a class. If you don't define a constructor, the compiler will generate one that does nothing.

The constructor for `DrawablePolygon` uses `super` to invoke the constructor for `Polygon`, which initializes the attributes `npoints`, `xpoints`, and `ypoints`. Then `DrawablePolygon` initializes the `color` attribute to `GRAY`.

`DrawablePolygon` has the same attributes and methods that `Polygon` has, so you can use `addPoint` as before, or you can directly access `npoints`, `xpoints`, and `ypoints` (since they are `public`). You can also use methods like `contains`, `intersects`, and `translate`.

The following code creates a `DrawablePolygon` with the same points as in the previous section and sets its color to `GREEN`:

```
DrawablePolygon p = new DrawablePolygon();
p.addPoint(57, 110);
p.addPoint(100, 35);
p.addPoint(143, 110);
p.color = Color.GREEN;
```

## 17.3 Regular Polygons

In mathematics, a regular polygon has all sides the same length and all angles equal in measure. Regular polygons are a special case of polygons, so we will use specialization to define a class for them.

We could extend the `Polygon` class, as we did in the previous section. But then we would not have the `Color` functionality we just added. So we will make `RegularPolygon` extend `DrawablePolygon`.

To construct a `RegularPolygon`, we specify the number of sides, the radius (distance from the center to a vertex), and the color. For example:

```
RegularPolygon rp = new RegularPolygon(6, 50, Color.BLUE);
```

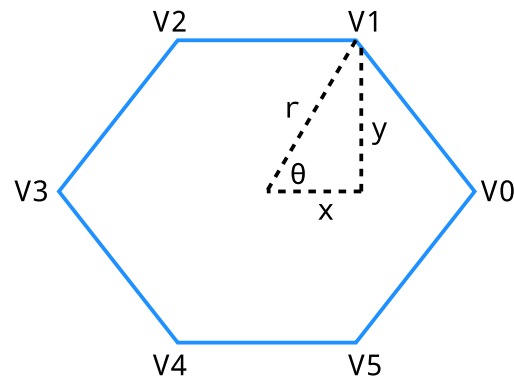


Figure 17.1: Determining the  $x$  and  $y$  coordinates of vertex  $V1$ , given the radius  $r$  and angle  $\theta$ . The center of the polygon is at the origin  $(0,0)$ .

The constructor uses trigonometry to find the coordinates of each vertex. Figure 17.1 illustrates the process. The number of sides ( $n = 6$ ) and the radius ( $r = 50$ ) are given as parameters.

- Imagine a clock hand starting at  $V0$  and rotating counterclockwise to  $V1$ ,  $V2$ , and so forth. In Figure 17.1, the hand is currently at  $V1$ .
- The angle  $\theta$  is  $2\pi/n$ , since there are  $2\pi$  radians in a circle. In other words, we are dividing the rotation of the clock hand into  $n$  equal angles.
- By definition,  $\cos(\theta) = x/r$  and  $\sin(\theta) = y/r$ . Therefore,  $x = r \cos(\theta)$  and  $y = r \sin(\theta)$ .
- We can determine the other  $(x,y)$  coordinates by multiplying  $\theta$  by  $i$ , where  $i$  is the vertex number.

Here is the constructor for `RegularPolygon`:

```
public RegularPolygon(int nsides, int radius, Color color) {  
  
    // initialize DrawablePolygon attributes  
    this.npoints = nsides;  
    this.xpoints = new int[nsides];  
    this.ypoints = new int[nsides];  
    this.color = color;  
  
    // the amount to rotate for each vertex (in radians)  
    double theta = 2.0 * Math.PI / nsides;  
  
    // compute x and y coordinates, centered at the origin  
    for (int i = 0; i < nsides; i++) {  
        double x = radius * Math.cos(i * theta);  
        double y = radius * Math.sin(i * theta);  
        xpoints[i] = (int) Math.round(x);  
        ypoints[i] = (int) Math.round(y);  
    }  
}
```

This constructor initializes all four `DrawablePolygon` attributes, so it doesn't have to invoke `super()`.

It initializes `xpoints` and `ypoints` by creating arrays of integer coordinates. Inside the `for` loop, it uses `Math.sin` and `Math.cos` (see Section 4.6) to compute the coordinates of the vertices as floating-point numbers. Then it rounds them off to integers and stores them in the arrays.

When we construct a `RegularPolygon`, the vertices are centered at the point  $(0, 0)$ . If we want the center of the polygon to be somewhere else, we can use `translate`, which we inherit from `Polygon`:

```
RegularPolygon rp = new RegularPolygon(6, 50, Color.BLUE);  
rp.translate(100, 100);
```

The result is a six-sided polygon with radius 50 centered at the point  $(100, 100)$ .

## 17.4 More Constructors

Classes in the Java library often have more than one constructor for convenience. We can do the same with `RegularPolygon`. For example, we can make the `color` parameter optional by defining a second constructor:

```
public RegularPolygon(int nsides, int radius) {  
    this(nsides, radius, Color.GRAY);  
}
```

The keyword `this`, when used in a constructor, invokes another constructor in the same class. It has a similar syntax as the keyword `super`, which invokes a constructor in the superclass.

Similarly, we could make the `radius` parameter optional too:

```
public RegularPolygon(int nsides) {  
    this(nsides, 50);  
}
```

Now, suppose we invoke the `RegularPolygon` constructor like this:

```
RegularPolygon rp = new RegularPolygon(6);
```

Because we provide only one integer argument, Java calls the third constructor, which calls the second one, which calls the first one. The result is a `RegularPolygon` with the specified value of `nsides`, 6, the default value of `radius`, 50, and the default color, `GRAY`.

When writing constructors, it's a good idea to validate the values you get as arguments. Doing so prevents run-time errors later in the program, which makes the code easier to debug.

For `RegularPolygon`, the number of sides should be at least three, the radius should be greater than zero, and the color should not be `null`. We can add the following lines to the first constructor:

```
public RegularPolygon(int nsides, int radius, Color color) {  
  
    // validate the arguments  
    if (nsides < 3) {  
        throw new IllegalArgumentException("invalid nsides");  
    }  
    if (radius <= 0) {  
        throw new IllegalArgumentException("invalid radius");  
    }  
    if (color == null) {  
        throw new NullPointerException("invalid color");  
    }  
  
    // the rest of the method is omitted  
}
```

In this example, we `throw` an exception to indicate that one of the arguments is invalid. By default, these exceptions terminate the program and display an error message along with the stack trace.

Because we added this code to the most general constructor, we don't have to add it to the others.

## 17.5 An Initial Drawing

Now that we have `DrawablePolygon` and `RegularPolygon`, let's take them for a test drive. We'll need a `Canvas` for drawing them, so we define a new class, `Drawing`, that extends `Canvas`:

```
public class Drawing extends Canvas {  
    private ArrayList<DrawablePolygon> list;  
  
    public Drawing(int width, int height) {  
        setSize(width, height);  
        setBackground(Color.WHITE);  
        list = new ArrayList<DrawablePolygon>();  
    }  
}
```



```
public void add(DrawablePolygon dp) {
    list.add(dp);
}

public void paint(Graphics g) {
    for (DrawablePolygon dp : list) {
        dp.draw(g);
    }
}
}
```

The `Drawing` class has an `ArrayList` of `DrawablePolygon` objects. When we create a `Drawing` object, the list is initially empty. The `add` method takes a `DrawablePolygon` and adds it to the list.

`Drawing` overrides the `paint` method that it inherits from `Canvas`. `paint` loops through the list of `DrawablePolygon` objects and invokes `draw` on each one.

Here is an example that creates three `RegularPolygon` objects and draws them. Figure 17.2 shows the result.

```
public static void main(String[] args) {

    // create some regular polygons
    DrawablePolygon p1 = new RegularPolygon(3, 50, Color.GREEN);
    DrawablePolygon p2 = new RegularPolygon(6, 50, Color.ORANGE);
    DrawablePolygon p3 = new RegularPolygon(360, 50, Color.BLUE);

    // move them out of the corner
    p1.translate(100, 80);
    p2.translate(250, 120);
    p3.translate(400, 160);

    // create drawing, add polygons
    Drawing drawing = new Drawing(500, 250);
    drawing.add(p1);
    drawing.add(p2);
    drawing.add(p3);
}
```

```
// set up the window frame
JFrame frame = new JFrame("Drawing");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(drawing);
frame.pack();
frame.setVisible(true);
}
```

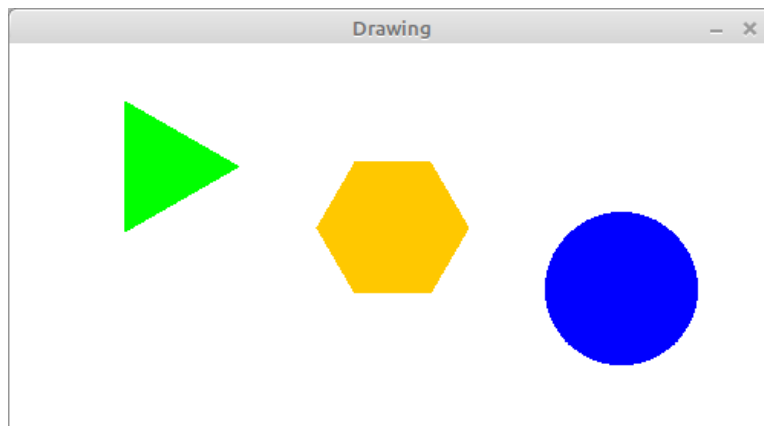


Figure 17.2: Initial drawing of three `RegularPolygon` objects.

The first block of code creates `RegularPolygon` objects with 3, 6, and 360 sides. As you can see, a polygon with 360 sides is a pretty good approximation of a circle.

The second block of code translates the polygons to different locations. The third block of code creates the `Drawing` and adds the polygons to it. And the fourth block of code creates a `JFrame`, adds the `Drawing` to it, and displays the result.

Most of these pieces should be familiar, but one part of this program might surprise you. When we create the `RegularPolygon` objects, we assign them to `DrawablePolygon` variables. It might not be obvious why that's legal.

`RegularPolygon` extends `DrawablePolygon`, so every `RegularPolygon` object is also a `DrawablePolygon`. The parameter of `Drawing.add` has to be a `DrawablePolygon`, but it can be any type of `DrawablePolygon`, including `RegularPolygon` and other subclasses.

This design is an example of **polymorphism**, a fancy word that means “having many forms”. `Drawing.add` is a polymorphic method, because the parameter can be one of many types. And the `ArrayList` in `Drawing` is a polymorphic data structure, because the elements can be different types.

## 17.6 Blinking Polygons

At this point, we have a simple program that draws polygons; we can make it more fun by adding animation. Chapter 15 introduced the idea of simulating time steps. Here’s a loop that runs the animation:

```
while (true) {
    drawing.step();
    try {
        Thread.sleep(1000 / 30);
    } catch (InterruptedException e) {
        // do nothing
    }
}
```

Each time through the loop, we call `step` to update the `Drawing`. Then we sleep with a delay calculated to update about 30 times per second.

Here’s what the `step` method of `Drawing` looks like:

```
public void step() {
    for (DrawablePolygon dp : list) {
        dp.step();
    }
    repaint();
}
```

It invokes `step` on each `DrawablePolygon` in the list and then repaints (clears and redraws) the canvas.

In order for this code to compile, we need `DrawablePolygon` to provide a `step` method. Here’s a version that doesn’t do anything; we’ll override it in subclasses:

```
public void step() {  
    // do nothing  
}
```

Now let's design a new type of polygon that blinks. We'll define a class named `BlinkingPolygon` that extends `RegularPolygon` and adds two more attributes: `visible`, which indicates whether the polygon is visible, and `count`, which counts the number of time steps since the last blink:

```
public class BlinkingPolygon extends RegularPolygon {  
    protected boolean visible;  
    protected int count;  
  
    public BlinkingPolygon(int nsides, int radius, Color c) {  
        super(nsides, radius, c);  
        visible = true;  
        count = 0;  
    }  
  
    public void draw(Graphics g) {  
        if (visible) {  
            super.draw(g);  
        }  
    }  
  
    public void step() {  
        count++;  
        if (count == 10) {  
            visible = !visible;  
            count = 0;  
        }  
    }  
}
```

The constructor uses `super` to call the `RegularPolygon` constructor. Then it initializes `visible` and `count`. Initially, the `BlinkingPolygon` is visible.

The `draw` method draws the polygon only if it is visible. It uses `super` to call `draw` in the parent class. But the parent class is `RegularPolygon`, which

does not provide a `draw` method. In this case, `super` invokes `draw` from the `DrawablePolygon` class.

The `step` method increments `count`. Every 10 time steps, it toggles `visible` and resets `count` to 0.

## 17.7 Interfaces

You might be getting tired of polygons at this point. Can't we draw anything else? Of course we can, but `Drawing` is currently based on `DrawablePolygon`. To draw other types of objects, we have to generalize the code.

The `Drawing` class does essentially three things: (1) it maintains a list of objects, (2) it invokes the `draw` method on each object, and (3) it invokes the `step` method on each object.

So here's one way we could make the code more general:

1. Define a new superclass, which we call `Actor`, that provides the two methods needed by `Drawing`:

```
public class Actor {
    public void draw(Graphics g) {
        // do nothing
    }
    public void step() {
        // do nothing
    }
}
```

2. In the `Drawing` class, replace `DrawablePolygon` with `Actor`.
3. Any class that we want to draw must now extend `Actor`.

There's just one problem: `DrawablePolygon` already extends `Polygon`, and classes can extend only one superclass. Also, the `Actor` class seems pointless, since the methods it defines don't do anything.

Java provides another mechanism for inheritance that solves these problems. We can define `Actor` as an `interface` instead of a `class`, like this:

```
public interface Actor {  
    void draw(Graphics g);  
    void step();  
}
```

Like a class definition, an **interface** definition contains methods. But it contains only the declarations of the methods, not their implementations.

Like an abstract class, an interface specifies methods that must be provided by subclasses. The difference is that an abstract class can implement some methods; an interface cannot.

All interface methods are **public** by default, since they are intended to be used by other classes. So there is no need to declare them as **public**.

To inherit from an interface, you use the keyword **implements** instead of **extends**. Here's a version of `DrawablePolygon` that extends `Polygon` and implements `Actor`. So it inherits methods from `Polygon`, and it is required to provide the methods in `Actor`; namely `draw` and `step`:

```
public class DrawablePolygon extends Polygon implements Actor {  
    // rest of the class omitted  
}
```

In terms of inheritance, `DrawablePolygon` is both a `Polygon` and an `Actor`. So the following assignments are legal:

```
Polygon p1 = new DrawablePolygon();  
Actor a2 = new DrawablePolygon();
```

And the same is true for subclasses of `DrawablePolygon`; these assignments are legal too:

```
Polygon p2 = new RegularPolygon(5, 50, Color.YELLOW);  
Actor a2 = new RegularPolygon(5, 50, Color.YELLOW);
```

Interfaces are another example of polymorphism. `a1` and `a2` are the same type of variable, but they refer to objects with different types. And similarly with `p1` and `p2`.

Classes may extend only one superclass, but they may implement as many interfaces as needed. Java library classes often implement multiple interfaces.

## 17.8 Event Listeners

Now that our `Drawing` is based on `Actor` instead of `DrawablePolygon`, we can draw other types of graphics. Here is the beginning of a class that reads an image from a file and shows the image moving across the canvas. The class is called `Sprite` because a moving image is sometimes called a **sprite**, in the context of computer graphics:

```
public class Sprite implements Actor, KeyListener {
    private int xpos;
    private int ypos;
    private int dx;
    private int dy;
    private Image image;

    public Sprite(String path, int xpos, int ypos) {
        this.xpos = xpos;
        this.ypos = ypos;
        try {
            this.image = ImageIO.read(new File(path));
        } catch (IOException exc) {
            exc.printStackTrace();
        }
    }
}
```

The instance variables `xpos` and `ypos` represent the location of the sprite. `dx` and `dy` represent the velocity of the sprite in the  $x$  and  $y$  directions.

The constructor takes as parameters the name of a file and the initial position. It uses `ImageIO`, from the `javax.imageio` package, to read the file. If an error occurs during reading, an `IOException` is caught, and the program displays the stack trace for debugging.

`Sprite` implements two interfaces: `Actor` and `KeyListener`. `Actor` requires that we provide `draw` and `step` methods:

```
public void draw(Graphics g) {
    g.drawImage(image, xpos, ypos, null);
}

public void step() {
    xpos += dx;
    ypos += dy;
}
```

The `draw` method draws the image at the sprite's current position. The `step` method changes the position based on `dx` and `dy`, which are initially zero.

`KeyListener` is an interface for receiving keyboard events, which means we can detect and respond to key presses. A class that implements `KeyListener` has to provide the following methods:

`void keyPressed(KeyEvent e)`

Invoked when a key has been “pressed”. This method is invoked repeatedly while a key is being held down.

`void keyReleased(KeyEvent e)`

Invoked when a key has been “released”, meaning it is no longer down.

`void keyTyped(KeyEvent e)`

Invoked when a key has been “typed”, which generally means it has been both pressed and released.

These methods get invoked when the user presses and releases *any* key. They take a `KeyEvent` object as a parameter, which specifies which key was pressed, released, or typed.

We can use these methods to design a simple animation using the arrow keys. When the user presses up or down, the sprite will move up or down. When the user presses left or right, the sprite will move left or right.

Here's an implementation of `keyPressed` that uses a `switch` statement to test which arrow key was pressed and sets `dx` or `dy` accordingly. (There is no `default` branch, so we ignore all other keys.)



```
public void keyPressed(KeyEvent e) {  
    switch (e.getKeyCode()) {  
        case KeyEvent.VK_UP:  
            dy = -5;  
            break;  
        case KeyEvent.VK_DOWN:  
            dy = +5;  
            break;  
        case KeyEvent.VK_LEFT:  
            dx = -5;  
            break;  
        case KeyEvent.VK_RIGHT:  
            dx = +5;  
            break;  
    }  
}
```

The values of `dx` and `dy` determine how much the sprite moves each time `step` is invoked. While the user holds down an arrow key, the sprite will move at a constant speed.

Here's an implementation of `keyReleased` that runs when the user releases the key:

```
public void keyReleased(KeyEvent e) {  
    switch (e.getKeyCode()) {  
        case KeyEvent.VK_UP:  
        case KeyEvent.VK_DOWN:  
            dy = 0;  
            break;  
        case KeyEvent.VK_LEFT:  
        case KeyEvent.VK_RIGHT:  
            dx = 0;  
            break;  
    }  
}
```

When the user releases the key, `keyReleased` sets `dx` or `dy` to 0, so the sprite stops moving in that direction.

We don't need the `keyTyped` method for this example, but it's required by the interface; if we don't provide one, the compiler will complain. So we provide an implementation that does nothing:

```
public void keyTyped(KeyEvent e) {  
    // do nothing  
}
```

Now, here's the code we need to create a `Sprite`, add it to a `Drawing`, and configure it as a `KeyListener`:

```
Sprite sprite = new Sprite("face-smile.png", 25, 150);  
drawing.add(sprite);  
drawing.addKeyListener(sprite);  
drawing.setFocusable(true);
```

Recall that the `add` method is one that we wrote in Section 17.5. It adds an `Actor` to the list of objects to be drawn.

The `addKeyListener` method is inherited from `Canvas`. It adds a `KeyListener` to the list of objects that will receive key events.

In graphical applications, key events are sent to components only when they have the keyboard focus. The `setFocusable` method ensures that `drawing` will have the focus initially, without the user having to click it first.

## 17.9 Timers

Now that you know about interfaces and events, we can show you a better way to create animations. Previously, we implemented the animation loop by using `while (true)` and `Thread.sleep`. Java provides a `Timer` class (in `javax.swing`) that encapsulates this behavior.

A `Timer` is useful for executing code at regular intervals. The constructor for `Timer` takes two parameters:

- `int` delay // milliseconds between events
- `ActionListener` listener // for handling timer events

The `ActionListener` interface requires only one method, `actionPerformed`. This is the method the `Timer` invokes after the given delay.

Using a `Timer`, we can reorganize the code in `main` by defining a class that implements `ActionListener`:

```
public class VideoGame implements ActionListener {
    private Drawing drawing;

    public VideoGame() {
        Sprite sprite = new Sprite("face-smile.png", 50, 50);
        drawing = new Drawing(800, 600);
        drawing.add(sprite);
        drawing.addKeyListener(sprite);
        drawing.setFocusable(true);

        JFrame frame = new JFrame("Video Game");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(drawing);
        frame.pack();
        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        drawing.step();
    }

    public static void main(String[] args) {
        VideoGame game = new VideoGame();
        Timer timer = new Timer(33, game);
        timer.start();
    }
}
```

The `main` method constructs a `VideoGame` object, which creates a `Sprite`, a `Drawing`, and a `JFrame`. Then it constructs a `Timer` object and starts the timer. Every 33 milliseconds, the `Timer` invokes `actionPerformed`, which invokes `step` on the `Drawing`.

`Drawing.step` invokes `step` on all of its `Actor` objects, which causes them

to update their position, color, or other aspects of their appearance. The `Drawing.step` then repaints the `Canvas`, and the time step is done.

At this point, you have all of the elements you need to write your own video games. In the exercises at the end of this chapter, we have some suggestions for getting started.

We hope this final chapter has been a helpful summary of topics presented throughout the book, including input and output, decisions and loops, classes and methods, arrays and objects, inheritance, and graphics. Congratulations on making it to the end!

## 17.10 Vocabulary

**generalization:** The process of extracting common code from two or more classes and moving it into a superclass.

**specialization:** Extending a class to add new attributes or methods, or to modify existing behavior.

**polymorphism:** A language feature that allows objects to be assigned to variables of related types.

**sprite:** A computer graphic that may be moved or otherwise manipulated on the screen.

## 17.11 Exercises

The code for this chapter is in the `ch17` directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

The following exercises give you a chance to practice using the features in this chapter by extending the example code.

**Exercise 17.1** The `Polygon` class does not provide a `toString` method; it inherits the default `toString` from `java.lang.Object`, which includes only the class's name and memory location. Write a more useful `toString` method for `DrawablePolygon` that includes its  $(x, y)$  points.

**Exercise 17.2** Write a class `MovingPolygon` that extends `RegularPolygon` and implements `Actor`. It should have instance variables `posx` and `posy` that specify its position, and `dx` and `dy` that specify its velocity (and direction). During each time step, it should update its position. If it gets to the edge of the `Drawing`, it should reverse direction by changing the sign of `dx` or `dy`.

**Exercise 17.3** Modify the `VideoGame` class so it displays a `Sprite` and a `MovingPolygon` (from the previous exercise). Add code that detects collisions between `Actor` objects in the same `Drawing`, and invoke a method on both objects when they collide.

*Hint:* You might want to add a method to the `Actor` interface, guaranteeing that all `Actor` objects know how to handle collisions.

**Exercise 17.4** Java provides other event listeners that you can implement to make your programs interactive. For example, the interfaces `MouseListener`, `MouseMotionListener`, and `MouseWheelListener` allow you to respond to mouse input. Use the `MouseListener` interface to implement an `Actor` that can respond to mouse clicks.

# Appendix A

## Tools

The steps for compiling, running, and debugging Java code depend on your development environment and operating system. We avoided putting these details in the main text, because they can be distracting.

Instead, we provide this appendix with a brief introduction to DrJava—an **integrated development environment** (IDE) that is helpful for beginners—and other development tools, including Checkstyle for code quality and JUnit for testing.

### A.1 Installing DrJava

The easiest way to start programming in Java is to use a website that compiles and runs Java code in the browser. Examples include <https://repl.it/>, <https://trinket.io/>, <https://jdoodle.com/>, and others.

If you are unable to install software on your computer (which is often the case in public schools and Internet cafés), you can use these online development environments for almost everything in this book.

But if you want to compile and run Java programs on your own computer, you will need the following:

- The **Java Development Kit** (JDK), which includes the compiler, the **Java Virtual Machine** (JVM) that interprets the compiled byte code, and other tools such as Javadoc.
- A **text editor** such as Atom, Notepad++, or Sublime Text, and/or an IDE such as DrJava, Eclipse, jGrasp, or NetBeans.

The JDK we recommend is OpenJDK, an open source implementation of Java SE (Standard Edition). The IDE we recommend is DrJava, which is an open source development environment written in Java (see Figure A.1).

To install OpenJDK, visit <https://adoptopenjdk.net>. Download and run the installer for your operating system.

To install DrJava, visit <http://drjava.org/> and download the **JAR** file. We recommend that you save it to your **Desktop** folder or another convenient location. Simply double-click the JAR file to run DrJava. Refer to the DrJava documentation (<http://drjava.org/docs/quickstart/>) for more details.

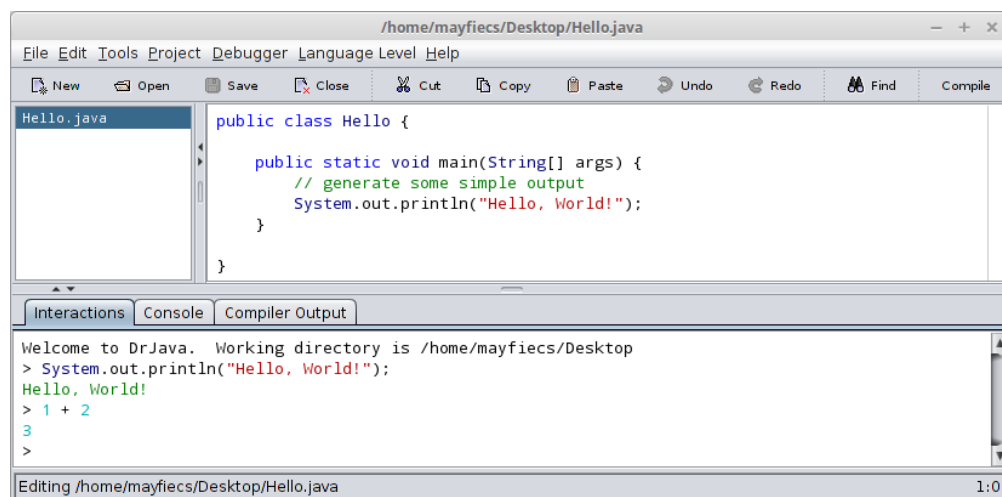


Figure A.1: DrJava editing the Hello World program.

When running DrJava for the first time, we recommend you change three settings from the **Edit > Preferences** menu under **Miscellaneous**: set the **Indent Level** to 4, check the **Automatically Close Block Comments** box, and uncheck the **Keep Emacs-style Backup Files** box.

## A.2 DrJava Interactions

One of the most useful features of DrJava is the “Interactions” pane at the bottom of the window. It provides the ability to try out code quickly, without having to write a class definition and save/compile/run the program. Figure A.2 shows an example.

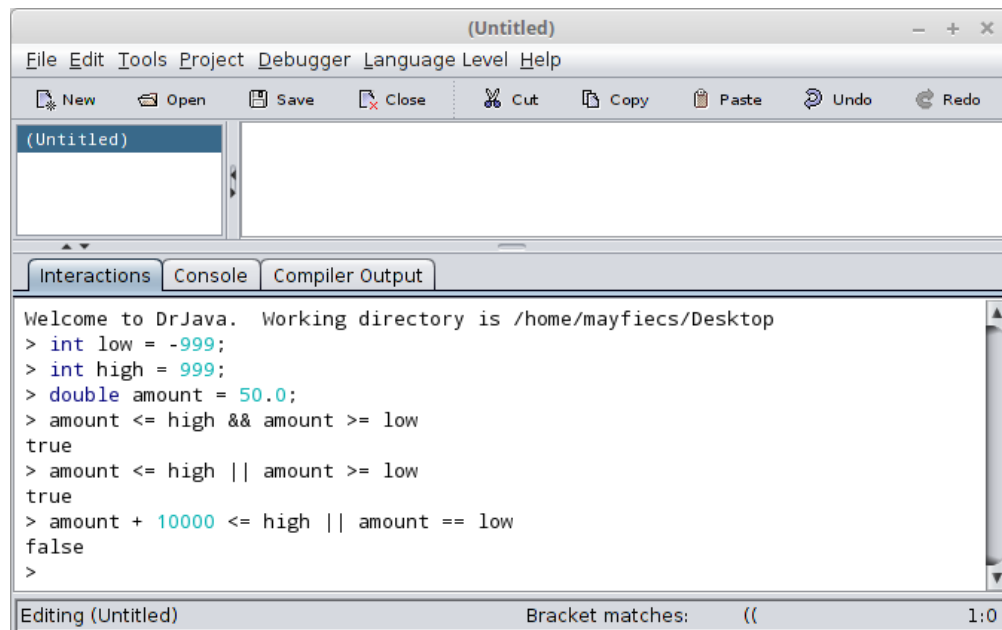


Figure A.2: The Interactions pane in DrJava.

There is one subtle detail to note when using the Interactions pane. If you don't end an expression (or statement) with a semicolon, DrJava automatically displays its value. Notice in Figure A.2 that the variable declarations end with semicolons, but the logic expressions in the following lines do not. This feature saves you from having to type `System.out.println` every time.

What's nice about this feature is that you don't have to create a new class, declare a `main` method, write arbitrary expressions inside `System.out.println` statements, save the source file, and get all of your code to compile in advance. Also, you can press the up/down arrows on the keyboard to repeat previous commands and experiment with incremental differences.

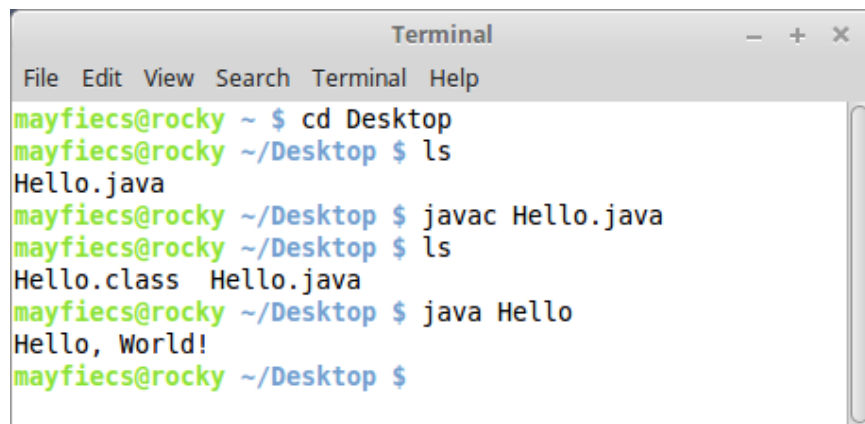


## A.3 Command-Line Interface

One of the most powerful and useful skills you can learn is how to use the **command-line interface**, also called the “terminal”. The command line is a direct interface to the operating system. It allows you to run programs, manage files and directories, and monitor system resources. Many advanced tools, both for software development and general-purpose computing, are available only at the command line.

Many good tutorials are available online for learning the command line for your operating system; just search the web for “command line tutorial”. On Unix systems like Linux and macOS, you can get started with just four commands: change the working directory (`cd`), list directory contents (`ls`), compile Java programs (`javac`), and run Java programs (`java`).

Figure A.3 shows an example in which the *Hello.java* source file is stored in the *Desktop* directory. After changing to that location and listing the files, we use the `javac` command to compile *Hello.java*. Running `ls` again, we see that the compiler generated a new file, *Hello.class*, which contains the byte code. We run the program by using the `java` command, which displays the output on the following line.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows a series of commands and their outputs. The user is at a prompt "mayfiecs@rocky ~ \$". They enter "cd Desktop", and the prompt changes to "mayfiecs@rocky ~/Desktop \$". They enter "ls", and the output is "Hello.java". They enter "javac Hello.java", and the prompt changes to "mayfiecs@rocky ~/Desktop \$". They enter "ls", and the output is "Hello.class Hello.java". They enter "java Hello", and the output is "Hello, World!". The prompt then changes to "mayfiecs@rocky ~/Desktop \$".

```
Terminal
File Edit View Search Terminal Help
mayfiecs@rocky ~ $ cd Desktop
mayfiecs@rocky ~/Desktop $ ls
Hello.java
mayfiecs@rocky ~/Desktop $ javac Hello.java
mayfiecs@rocky ~/Desktop $ ls
Hello.class Hello.java
mayfiecs@rocky ~/Desktop $ java Hello
Hello, World!
mayfiecs@rocky ~/Desktop $
```

Figure A.3: Compiling and running *Hello.java* from the command line.

Note that the `javac` command requires a *filename* (or multiple source files separated by spaces), whereas the `java` command requires a single *class name*. If you use DrJava, it runs these commands for you behind the scenes and displays the output in the Interactions pane.

Taking time to learn this efficient and elegant way of interacting with the operating system will make you more productive. People who don't use the command line don't know what they're missing.

## A.4 Command-Line Testing

As described in Section 1.9, it's more effective to program and debug your code little by little than to attempt writing everything all at once. And after you've completed programming an algorithm, it's important to test that it works correctly on a variety of inputs.

Throughout the book, we illustrate techniques for testing your programs. Most, if not all, testing is based on a simple idea: does the program do what we expect it to do? For simple programs, it's not difficult to run them several times and see what happens. But at some point, you will get tired of typing the same test cases over and over.

We can automate the process of entering input and comparing *expected output* with *actual output* using the command line. The basic idea is to store the test cases in plain text files and trick Java into thinking they are coming from the keyboard. Here are step-by-step instructions:

1. Make sure you can compile and run the *Convert.java* example in the *ch03* directory of *ThinkJavaCode2*. (See page xviii for instructions on how to download the repository.)
2. In the same directory as *Convert.java*, create a plain text file named *test.in* (“in” is for “input”). Enter the following line and save the file:

```
193.04
```

3. Create a second plain text file named *test.exp* (“exp” is for “expected”). Enter the following line and save the file:

```
193.04 cm = 6 ft, 4 in
```

4. Open a terminal, and change to the directory with these files. Run the following command to test the program:

```
java Convert < test.in > test.out
```

On the command line, `<` and `>` are **redirection operators**. The first one redirects the contents of *test.in* to `System.in`, as if it were entered from the keyboard. The second one redirects the contents of `System.out` to a new file *test.out*, much like a screen capture. In other words, the *test.out* file contains the output of your program.

By the way, it's perfectly okay to compile your programs in DrJava (or another environment) and run them from the command line. Knowing both techniques allows you to use the right tool for the job.

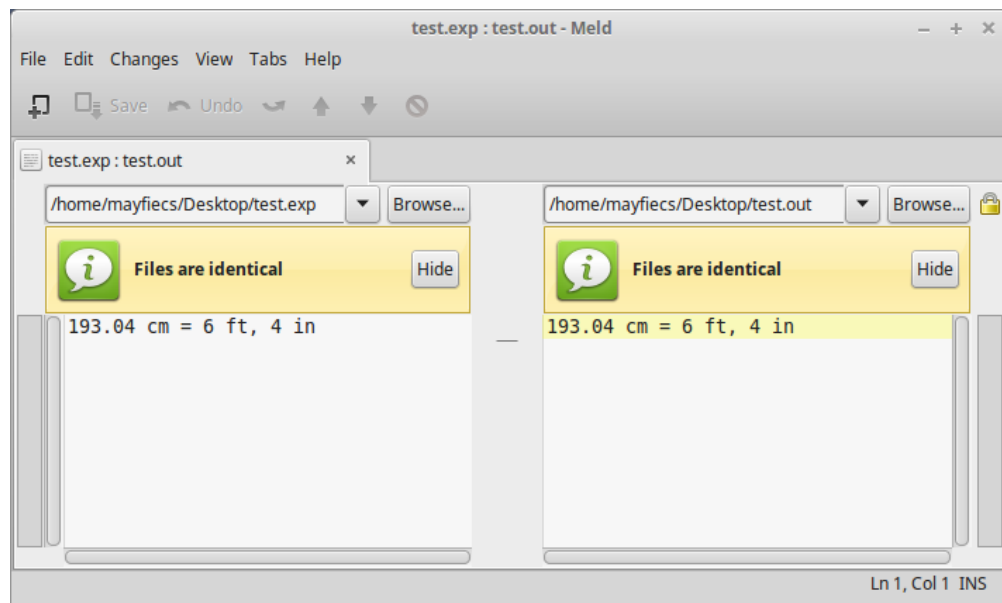


Figure A.4: Using `meld` to compare expected output with the actual output.

At this point, we just need to compare the contents `test.out` with `test.exp`. If the files are the same, then the program outputted what we expected it to output. If not, then we found a bug, and we can use the output to begin debugging our program. Fortunately, there's a simple way to compare files on the command line:

```
diff test.exp test.out
```

The `diff` utility summarizes the differences between two files. If there are no differences, it displays nothing, which in our case is what we want. If the expected output differs from the actual output, we need to continue debugging. Usually, the program is at fault, and `diff` provides some insight about what

is broken. But there's also a chance that we have a correct program and the expected output is wrong.

Interpreting the results from `diff` can be confusing, but fortunately many graphical tools can show the differences between two files. For example, on Windows you can install `WinMerge`, on macOS you can use `opendiff` (which comes with Xcode), and on Linux there's `meld`, shown in Figure A.4.

Regardless of what tool you use, the goal is the same. Debug your program until the actual output is *identical* to the expected output.

## A.5 Running Checkstyle

Checkstyle is a command-line tool that can be used to determine if your source code follows a set of style rules. It also checks for common programming mistakes, such as class and method design problems.

You can download the latest version as a JAR file from <https://checkstyle.sourceforge.io/>. To run Checkstyle, move (or copy) the JAR file to the same directory as your program. Open a terminal in that location, and run the following command:

```
java -jar checkstyle-*-all.jar -c /google_checks.xml *.java
```

The `*` characters are **wildcards** that match whatever version of Checkstyle you have and whatever Java source files are present. The output indicates the file and line number of each problem. This example refers to a method beginning on line 93, column 5 of *Hello.java*:

```
Hello.java:93:5: Missing a Javadoc comment
```

The file `/google_checks.xml` is inside the JAR file and represents most of Google's style rules. You can alternatively use `/sun_checks.xml` or provide your own configuration file. See Checkstyle's website for more information.

If you apply Checkstyle to your source code often, you will likely internalize good style habits over time. But there are limits to what automatic style checkers can do. In particular, they can't evaluate the *quality* of your comments, the *meaning* of your variable names, or the *structure* of your algorithms.

Good comments make it easier for experienced developers to identify errors in your code. Good variable names communicate the intent of your program and how the data is organized. And good programs are designed to be efficient and demonstrably correct.

## A.6 Tracing with a Debugger

A great way to visualize the flow of execution, including how parameters and arguments work, is to use a **debugger**. Most debuggers make it possible to do the following:

- Set a **breakpoint**, a line where you want the program to pause.
- Step through the code one line at a time and watch what it does.
- Check the values of variables and see when and how they change.

For example, open any program in DrJava and move the cursor to the first line of `main`. Press `Ctrl+B` to toggle a breakpoint on the current line; it should now be highlighted in red. Press `Ctrl+Shift+D` to turn on “Debug Mode”; a new pane should appear at the bottom of the window. These commands are also available from the **Debugger** menu, in case you forget the shortcut keys.

When you run the program, execution pauses at the first breakpoint. The debugging pane displays the **call stack**, with the current method on top of the stack, as shown in Figure A.5. You might be surprised to see how many methods were called before the `main` method!

To the right are several buttons that allow you to step through the code at your own pace. You can also click **Automatic Trace** to watch DrJava run your code one line at a time.

Using a debugger is like having the computer proofread your code out loud. When the program is paused, you can examine (or even change) the value of any variable by using the Interactions pane.

Tracing allows you to follow the flow of execution and see how data passes from one method to another. You might expect the code do one thing, but

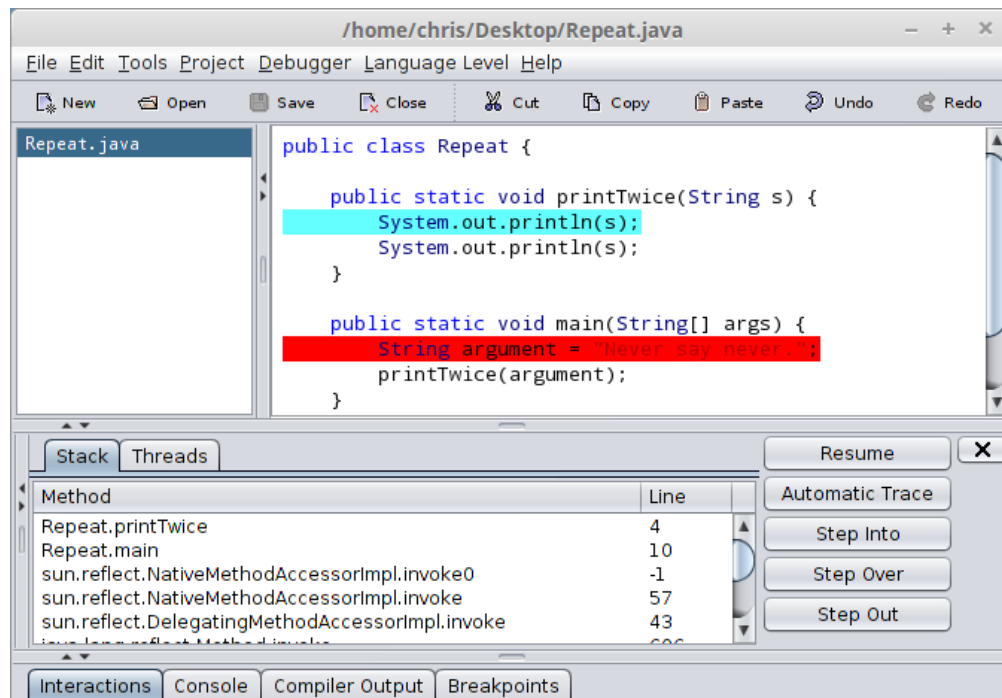


Figure A.5: The DrJava debugger. Execution is currently paused on the first line of `printTwice`. There is a breakpoint on the first line of `main`.

then the debugger shows it doing something else. At that moment, you gain insight about what may be wrong with the code.

You can edit your code while debugging it, but we don't recommend it. If you add or delete multiple lines of code while the program is paused, the results can be confusing.

See <http://drjava.org/docs/user/ch09.html> for more information about using the debugger feature of DrJava.

## A.7 Testing with JUnit

When beginners start writing methods, they usually test them by invoking them from `main` and checking the results by hand. For example, to test `fibonacci` from Section 8.4, we could write this:

```
public static void main(String[] args) {
    if (fibonacci(1) != 1) {
        System.err.println("fibonacci(1) is incorrect");
    }
    if (fibonacci(2) != 1) {
        System.err.println("fibonacci(2) is incorrect");
    }
    if (fibonacci(3) != 2) {
        System.err.println("fibonacci(3) is incorrect");
    }
}
```

This test code is self-explanatory, but it's longer than it needs to be, and it doesn't scale very well. In addition, the error messages provide limited information. For cases where we know the right answer, we can do better by writing **unit tests**.

JUnit (<https://junit.org/>) is a common testing tool for Java programs. To use it, you have to create a test class that contains test methods.

For example, suppose that the `fibonacci` method belongs to a class named `Series`. Here is a corresponding JUnit<sup>1</sup> test class and test method:

```
import junit.framework.TestCase;

public class SeriesTest extends TestCase {

    public void testFibonacci() {
        assertEquals(1, Series.fibonacci(1));
        assertEquals(1, Series.fibonacci(2));
        assertEquals(2, Series.fibonacci(3));
    }
}
```

This example uses the keyword `extends`, which indicates that the new class, `SeriesTest`, is based on an existing class, `TestCase`. The `TestCase` class is imported from the package `junit.framework`.

---

<sup>1</sup>This code is based on JUnit 3, which has been superseded but is the default version supported by DrJava.

The names in this example follow convention: if the name of your class is `Something`, the name of the test class should be `SomethingTest`. And if there is a method in `Something` named `someMethod`, there should be a method in `SomethingTest` named `testSomeMethod`.

Many development environments can generate test classes and test methods automatically. In DrJava, you can select **New JUnit Test Case** from the File menu to generate an empty test class.

`assertEquals` is provided by the `TestCase` class. It takes two arguments and checks whether they are equal. If so, it does nothing; otherwise, it displays a detailed error message. The first argument is the *expected value*, which we consider correct, and the second argument is the *actual value* we want to check. If they are not equal, the test fails.

Using `assertEquals` is more concise than writing your own `if` statements and `System.err` messages. JUnit provides additional assert methods, such as `assertNull`, `assertSame`, and `assertTrue`, which can be used to design a variety of tests.

To run JUnit directly from DrJava, click the **Test** button on the toolbar. If all your test methods pass, you will see a green bar in the lower-right corner. Otherwise, DrJava will take you directly to the first assertion that failed.

## A.8 Vocabulary

**IDE:** An “integrated development environment” that includes tools for editing, compiling, and debugging programs.

**JDK:** The “Java Development Kit”, which contains the compiler, Javadoc, and other tools.

**JVM:** The “Java Virtual Machine”, which interprets the compiled byte code.

**text editor:** A program that edits plain text files, the format used by most programming languages.

**JAR:** A “Java Archive”, which is essentially a ZIP file containing classes and other resources.



**command-line interface:** A means of interacting with the computer by issuing commands in the form of successive lines of text.

**redirection operator:** A command-line feature that substitutes `System.in` and/or `System.out` with a plain text file.

**wildcard:** A command-line feature that allows you to specify a pattern of filenames by using the `*` character.

**debugger:** A tool that allows you to run one statement at a time and see the contents of variables.

**breakpoint:** A line of code at which the debugger will pause a running program.

**call stack:** The history of method calls and where to resume execution after each method returns.

**unit test:** Code that exercises a single method of a program, testing for correctness and/or efficiency.

# Appendix B

## Javadoc

Java programs have three types of comments:

**End-of-line comments:** These start with `//` and generally contain short phrases that explain specific lines of code.

**Multiline comments:** These start with `/*` and end with `*/`, and are typically used for copyright statements.

**Documentation comments:** These start with `/**` and end with `*/`, and describe what each class and method does.

End-of-line and multiline comments are written primarily for yourself. They help you remember specific details about your source code. Documentation comments, on the other hand, are written for others. They explain how to use your classes and methods in other programs.

A nice feature of the Java language is the ability to embed documentation in the source code itself. That way, you can write it as you go, and as things change, it is easier to keep the documentation consistent with the code.

You can extract documentation from your source code, and generate well-formatted HTML pages, using a tool called **Javadoc**. This tool is included with the JDK, and it is widely used. In fact, the official documentation for the Java library (<https://thinkjava.org/apidoc>) is generated by Javadoc.

## B.1 Reading Documentation

As an example, let's look at the documentation for `Scanner`, a class we first used in Section 3.2. You can find the documentation quickly by doing a web search for “Java Scanner”. Figure B.1 shows a screenshot of the page.

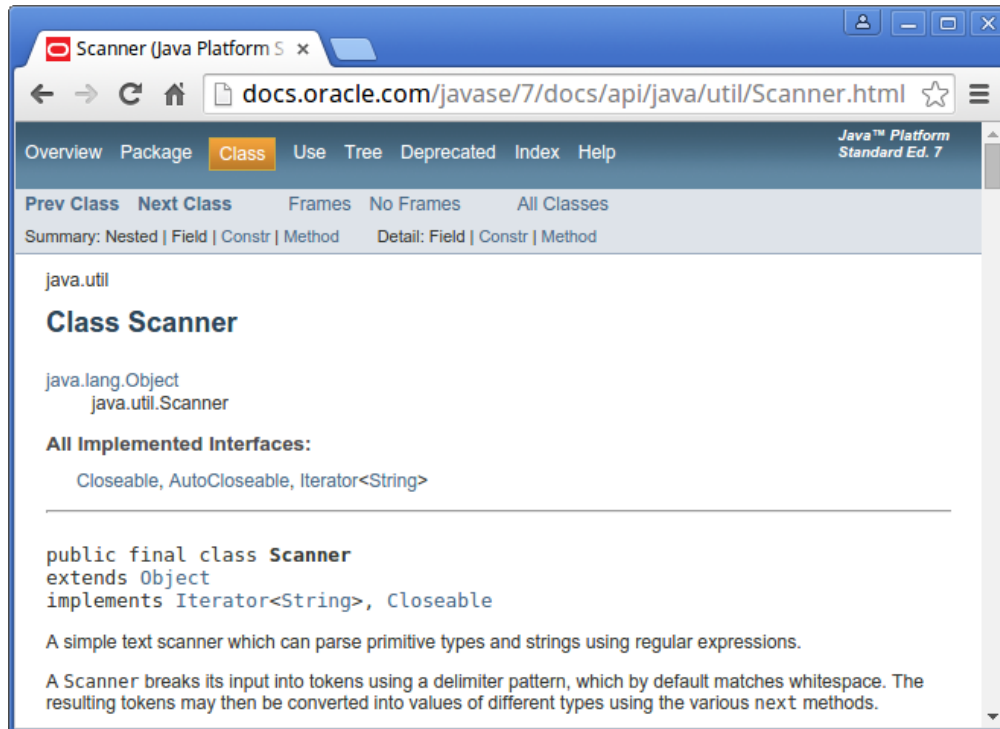


Figure B.1: The documentation for `Scanner`.

Documentation for other classes uses a similar format. The first line is the package that contains the class, such as `java.util`. The second line is the name of the class. The “All Implemented Interfaces” section lists some of the functionality a `Scanner` has.

The next section of the documentation is a narrative that explains the purpose of the class and includes examples of how to use it. This text can be difficult to read, because it may use terms you have not yet learned. But the examples are often very useful. A good way to get started with a new class is to paste the examples into a test file and see if you can compile and run them.

One of the examples shows how you can use a `Scanner` to read input from a `String` instead of `System.in`:

```
String input = "1 fish 2 fish red fish blue fish";  
Scanner s = new Scanner(input);
```

After the narrative, code examples, and other details, you will find the following tables:

**Constructor Summary:** Ways of creating, or constructing, a `Scanner`.

**Method Summary:** The list of methods that the `Scanner` class provides.

**Constructor Detail:** More information about how to create a `Scanner`.

**Method Detail:** More information about each method.

For example, here is the summary information for `nextInt`:

```
public int nextInt()  
Scans the next token of the input as an int.
```

The first line is the method's **signature**, which specifies the name of the method, its parameters (none), and the type it returns (`int`). The next line is a short description of what it does.

The “Method Detail” explains more:

```
public int nextInt()  
Scans the next token of the input as an int.
```

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:  
the int scanned from the input

Throws:  
`InputMismatchException` - if the next token does not match  
the Integer regular expression, or is out of range  
`NoSuchElementException` - if input is exhausted  
`IllegalStateException` - if this scanner is closed

The “Returns” section describes the result when the method succeeds. In contrast, the “Throws” section describes possible errors and exceptions that may occur. Exceptions are said to be thrown, like a referee throwing a flag, or like a toddler throwing a fit.

It might take you some time to get comfortable reading documentation and learning which parts to ignore. But it’s worth the effort. Knowing what’s available in the library helps you avoid reinventing the wheel. And a little bit of documentation can save you a lot of debugging.

## B.2 Writing Documentation

As you benefit from reading good documentation, you should “pay it forward” by writing good documentation.

Javadoc scans your source files looking for documentation comments, also known as “Javadoc comments”. They begin with `/**` (two stars) and end with `*/` (one star). Anything in between is considered part of the documentation.

Here’s a class definition with two Javadoc comments, one for the `Goodbye` class and one for the `main` method:

```
/**
 * Example program that demonstrates print vs println.
 */
public class Goodbye {

    /**
     * Prints a greeting.
     */
    public static void main(String[] args) {
        System.out.print("Goodbye, "); // note the space
        System.out.println("cruel world");
    }
}
```

The class comment explains the purpose of the class. The method comment explains what the method does.

Notice that this example also has an end-of-line comment (`/**`). In general, these comments are short phrases that help explain complex parts of a program. They are intended for other programmers reading and maintaining the source code.

In contrast, Javadoc comments are longer, usually complete sentences. They explain what each method does, but they omit details about how the method works. And they are intended for people who will use the methods without looking at the source code.

Appropriate comments and documentation are essential for making source code readable. And remember that the person most likely to read your code in the future, and appreciate good documentation, is you.

## B.3 Javadoc Tags

It's generally a good idea to document each class and method, so that other programmers can understand what they do without having to read the code.

To organize the documentation into sections, Javadoc supports optional **tags** that begin with the at sign (`@`). For example, we can use `@author` and `@version` to provide information about the class:

```
/**
 * Utility class for extracting digits from integers.
 *
 * @author Chris Mayfield
 * @version 1.0
 */
public class DigitUtil {
```

Documentation comments should begin with a **description** of the class or method, followed by the tags. These two sections are separated by a blank line (not counting the `*`).

For methods, we can use `@param` and `@return` to provide information about parameters and return values:

```
/**
 * Tests whether x is a single digit integer.
 *
 * @param x the integer to test
 * @return true if x has one digit, false otherwise
 */
public static boolean isSingleDigit(int x) {
```

Figure B.2 shows part of the resulting HTML page generated by Javadoc. Notice the relationship between the Javadoc comment (in the source code) and the resulting documentation (in the HTML page).

isSingleDigit
<pre>public static boolean isSingleDigit(int x)</pre>
Tests whether x is a single digit integer.
<b>Parameters:</b>
x - the integer to test
<b>Returns:</b>
true if x has one digit, false otherwise

Figure B.2: HTML documentation for `isSingleDigit`.

When writing parameter comments, do not include a hyphen (-) after the `@param` tag. Otherwise, you will have two hyphens in the resulting HTML documentation.

Notice also that the `@return` tag should not specify the type of the method. Comments like `@return boolean` are not useful, because you already know the return type from the method's signature.

Methods with multiple parameters should have separate `@param` tags that describe each one. Void methods should have no `@return` tag, since they do not return a value. Each tag should be on its own line in the source code.

## B.4 Example Source File

Now let's take a look at a more complete example. The code for this section is in the *appb* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository.

Professional-grade source files often begin with a copyright statement. This text spans multiple lines, but it is not part of the documentation. So we use a multiline comment (*/\**) rather than a documentation comment (*/\*\**). Our example source file, *Convert.java*, includes the MIT License (<https://opensource.org/licenses/MIT>):

```
/*
 * Copyright (c) 2019 Allen Downey and Chris Mayfield
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
```

Import statements generally follow the copyright text. After that, we can define the class itself and begin writing the documentation (*/\*\**):

```
import java.util.Scanner;

/**
 * Methods for converting to/from the metric system.
 *
 * @author Allen Downey
 * @author Chris Mayfield
 * @version 6.1.5
 */
public class Convert {
```



A common mistake that beginners make is to put `import` statements between the documentation and the `public class` line. Doing so separates the documentation from the class itself. To avoid this issue, always make the end of the comment (the `*/`) “touch” the word `public`.

This class has two constants and three methods. The constants are self-explanatory, so there is no need to write documentation for them:

```
public static final double CM_PER_INCH = 2.54;

public static final int IN_PER_FOOT = 12;
```

The methods, on the other hand, could use some explanation. Each documentation comment includes a description, followed by a blank line, followed by a `@param` tag for each parameter, followed by a `@return` tag:

```
/**
 * Converts a measurement in centimeters to inches.
 *
 * @param cm length in centimeters
 * @return length in inches
 */
public static double toImperial(double cm) {
    return cm / CM_PER_INCH;
}

/**
 * Converts a length in feet and inches to centimeters.
 *
 * @param feet how many feet
 * @param inches how many inches
 * @return length in centimeters
 */
public static double toMetric(int feet, int inches) {
    int total = feet * IN_PER_FOOT + inches;
    return total * CM_PER_INCH;
}
```

The main method has a similar documentation comment, except there is no `@return` tag since the method is `void`:

```
/**
 * Tests the conversion methods.
 *
 * @param args command-line arguments
 */
public static void main(String[] args) {
    double cm, result;
    int feet, inches;
    Scanner in = new Scanner(System.in);

    // test the Imperial conversion
    System.out.print("Exactly how many cm? ");
    cm = in.nextDouble();
    result = toImperial(cm);
    System.out.printf("That's %.2f inches\n", result);
    System.out.println();

    // test the Metric conversion
    System.out.print("Now how many feet? ");
    feet = in.nextInt();
    System.out.print("And how many inches? ");
    inches = in.nextInt();
    result = toMetric(feet, inches);
    System.out.printf("That's %.2f cm\n", result);
}
```

Here are two ways you can run the Javadoc tool on this example program:

- From the command line, go to the location for *Convert.java*. The `-d` option of `javadoc` indicates where to generate the HTML files:

```
javadoc -d doc Convert.java
```

- From DrJava, click the **Javadoc** button on the toolbar. The IDE will then prompt you for a location to generate the HTML files.

For more examples of what you can do with Javadoc comments, see the source code of any Java library class (e.g., *Scanner.java*). Section 10.6 explains how to find the source files for the Java library on your computer.

## B.5 Vocabulary

**documentation:** Comments that describe the technical operation of a class or method.

**Javadoc:** A tool that reads Java source code and generates documentation in HTML format.

**signature:** The first line of a method that defines its name, return type, and parameters.

**tag:** A label that begins with an at sign (@) and is used by Javadoc to organize documentation into sections.

**description:** The first line of a documentation comment that explains what the class/method does.

# Appendix C

## Graphics

The Java library includes the package `java.awt` for drawing 2D graphics. **AWT** stands for “Abstract Window Toolkit”. We are only going to scratch the surface of graphics programming. You can read more about it in the Java tutorials (see <https://thinkjava.org/java2d>).

### C.1 Creating Graphics

There are several ways to create graphics in Java; the simplest way is to use `java.awt.Canvas` and `java.awt.Graphics`. A `Canvas` is a blank rectangular area of the screen onto which the application can draw. The `Graphics` class provides basic drawing methods such as `drawLine`, `drawRect`, and `drawString`.

Here is an example program that draws a circle by using the `fillOval` method:

```
import java.awt.Canvas;
import java.awt.Graphics;
import javax.swing.JFrame;

public class Drawing extends Canvas {
```

```
public static void main(String[] args) {
    JFrame frame = new JFrame("My Drawing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Drawing drawing = new Drawing();
    drawing.setSize(400, 400);
    frame.add(drawing);
    frame.pack();
    frame.setVisible(true);
}

public void paint(Graphics g) {
    g.fillOval(100, 100, 200, 200);
}
}
```

The `Drawing` class extends `Canvas`, so it has all the methods provided by `Canvas`, including `setSize`. You can read about the other methods in the documentation, which you can find by doing a web search for “Java Canvas”.

In the `main` method, we do the following:

1. Create a `JFrame` object, which is the window that will contain the canvas.
2. Create a `Drawing` object (which is the canvas), set its width and height, and add it to the frame.
3. Pack the frame (resize it) to fit the canvas, and display it on the screen.

Once the frame is visible, the `paint` method is called whenever the canvas needs to be drawn; for example, when the window is moved or resized. If you run this code, you should see a black circle on a gray background.

The application doesn’t end after the `main` method returns; instead, it waits for the `JFrame` to close. When the `JFrame` closes, it calls `System.exit`, which ends the program.

## C.2 Graphics Methods

You are probably used to Cartesian **coordinates**, where  $x$  and  $y$  values can be positive or negative. In contrast, Java uses a coordinate system where the

origin is in the upper-left corner. That way,  $x$  and  $y$  can always be positive integers. Figure C.1 shows these coordinate systems side by side.

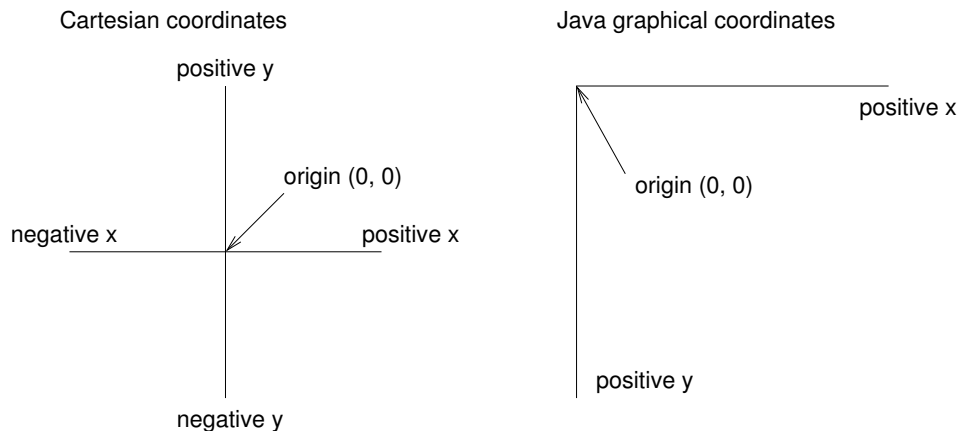


Figure C.1: The difference between Cartesian coordinates and Java graphical coordinates.

Graphical coordinates are measured in **pixels**; each pixel corresponds to a dot on the screen.

To draw on the canvas, you invoke methods on a **Graphics** object. You don't have to create the **Graphics** object; it gets created when you create the **Canvas**, and it gets passed as an argument to **paint**.

The previous example used **fillOval**, which has the following signature:

```
/**
 * Fills an oval bounded by the specified rectangle with
 * the current color.
 */
public void fillOval(int x, int y, int width, int height)
```

The four parameters specify a **bounding box**, which is the rectangle in which the oval is drawn. **x** and **y** specify the location of the upper-left corner of the bounding box. The bounding box itself is not drawn (see Figure C.2).

To choose the color of a shape, invoke **setColor** on the **Graphics** object:

```
g.setColor(Color.RED);
```

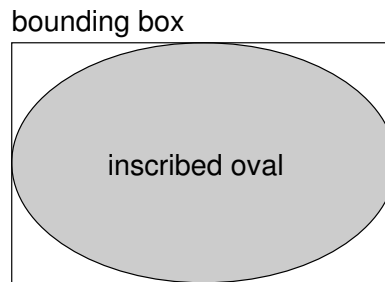


Figure C.2: An oval inside its bounding box.

The `setColor` method determines the color of everything that gets drawn afterward. `Color.red` is a constant provided by the `Color` class; to use it, you have to `import java.awt.Color`. Other colors include the following:

BLACK	BLUE	CYAN	DARKGRAY	GRAY	LIGHTGRAY
GREEN	MAGENTA	ORANGE	PINK	WHITE	YELLOW

You can create your own colors by specifying the red, green, and blue (**RGB**) components. For example:

```
Color purple = new Color(128, 0, 128);
```

Each value is an integer in the range 0 (darkest) to 255 (lightest). The color (0, 0, 0) is black, and (255, 255, 255) is white.

You can set the background color of the `Canvas` by invoking `setBackground`:

```
canvas.setBackground(Color.WHITE);
```

## C.3 Example Drawing

Suppose we want to draw a “Hidden Mickey”, which is an icon that represents Mickey Mouse (see [https://en.wikipedia.org/wiki/Hidden\\_Mickey](https://en.wikipedia.org/wiki/Hidden_Mickey)). We can use the oval we just drew as the face, and then add two ears. To make the code more readable, let’s use `Rectangle` objects to represent bounding boxes.

Here’s a method that takes a `Rectangle` and invokes `fillOval`:

```
public void boxOval(Graphics g, Rectangle bb) {  
    g.fillOval(bb.x, bb.y, bb.width, bb.height);  
}
```

And here's a method that draws Mickey Mouse:

```
public void mickey(Graphics g, Rectangle bb) {  
    boxOval(g, bb);  
  
    int hx = bb.width / 2;  
    int hy = bb.height / 2;  
    Rectangle half = new Rectangle(bb.x, bb.y, hx, hy);  
  
    half.translate(-hx / 2, -hy / 2);  
    boxOval(g, half);  
  
    half.translate(hx * 2, 0);  
    boxOval(g, half);  
}
```

The first line draws the face. The next three lines create a smaller rectangle for the ears. We **translate** the rectangle up and left for the first ear, then to the right for the second ear. The result is shown in Figure C.3.

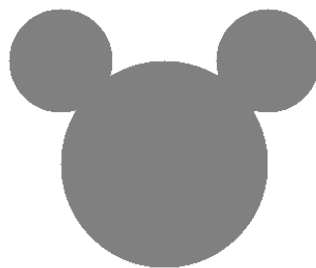


Figure C.3: A Hidden Mickey drawn using Java graphics.

You can read more about **Rectangle** and **translate** in Chapter 10. See the exercises at the end of this appendix for more example drawings.



## C.4 Vocabulary

**AWT:** The “Abstract Window Toolkit”, a Java package for creating graphical user interfaces.

**coordinate:** A value that specifies a location in a 2D graphical window.

**pixel:** The unit in which coordinates are measured.

**bounding box:** A way to specify the coordinates of a rectangular area.

**RGB:** A color model based on adding red, green, and blue light.

## C.5 Exercises

The code for this chapter is in the *appc* directory of *ThinkJavaCode2*. See page xviii for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

**Exercise C.1** Draw the flag of Japan: a red circle on a white background that is wider than it is tall.

**Exercise C.2** Modify *Mickey.java* to draw ears on the ears, and ears on those ears, and more ears all the way down until the smallest ears are only 3 pixels wide. The result should look like Figure C.4. *Hint:* You should have to add or modify only a few lines of code.

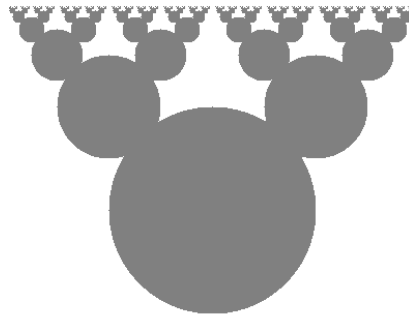
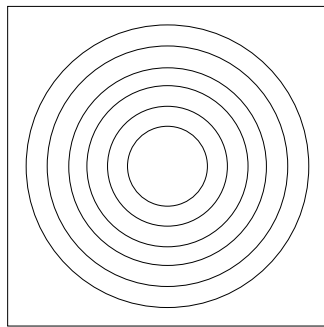


Figure C.4: A recursive shape we call “Mickey Moose”.

**Exercise C.3** In this exercise, you will draw “Moiré patterns” that seem to shift around as you move. For an explanation of what is going on, see [https://en.wikipedia.org/wiki/Moire\\_pattern](https://en.wikipedia.org/wiki/Moire_pattern).

1. Open *Moire.java* and read the `paint` method. Draw a sketch of what you expect it to do. Now run it. Did you get what you expected?
2. Modify the program so that the space between the circles is larger or smaller. See what happens to the image.
3. Modify the program so that the circles are drawn in the center of the screen and concentric, as in Figure C.5 (left). The distance between the circles should be small enough that the Moiré interference is apparent.

Concentric circles



Radial Moire pattern

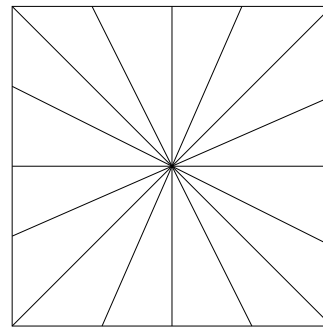


Figure C.5: Graphical patterns that can exhibit Moiré interference.

4. Write a method named `radial` that draws a radial set of line segments as shown in Figure C.5 (right), but they should be close enough together to create a Moiré pattern.
5. Just about any kind of graphical pattern can generate Moiré-like interference patterns. Play around and see what you can create.



# Appendix D

## Debugging

Although there are debugging suggestions throughout the book, we thought it would be useful to say more in an appendix. If you are having a hard time debugging, you might want to review this appendix from time to time.

The best debugging strategy depends on what kind of error you have:

- **Compile-time errors** indicate that there is something wrong with the syntax of the program. Example: omitting the semicolon at the end of a statement.
- **Run-time errors** are produced if something goes wrong while the program is running. Example: an infinite recursion eventually causes a `StackOverflowError`.
- **Logic errors** cause the program to do the wrong thing. Example: an expression may not be evaluated in the order you expect.

The following sections are organized by error type; some techniques are useful for more than one type.

### D.1 Compile-Time Errors

The best kind of debugging is the kind you don't have to do because you avoid making errors in the first place. Incremental development, which we presented

in Section 4.9, can help. The key is to start with a working program and add small amounts of code at a time. When there is an error, you will have a pretty good idea of where it is.

Nevertheless, you might find yourself in one of the following situations. For each situation, we have some suggestions about how to proceed.

### **The compiler is spewing error messages.**

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it often gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it reports spurious errors.

Only the first error message is truly reliable. We suggest that you fix only one error at a time and then recompile the program. You may find that one semicolon or brace “fixes” 100 errors.

### **I'm getting a weird compiler message, and it won't go away.**

First of all, read the error message carefully. It may be written in terse jargon, but often there is a carefully hidden kernel of information.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally, the error will be prior to the location of the error message, but in some cases it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition itself.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Make sure the program is indented properly; that makes it easier to spot syntax errors.

Now, start looking for common syntax errors:

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.
2. Remember that uppercase letters are not the same as lowercase letters.
3. Check for semicolons at the end of statements (and no semicolons after curly braces).
4. Make sure that any strings in the code have matching quotation marks. Make sure that you use double quotes for strings, and single quotes for characters.
5. For each assignment statement, make sure that the type on the left is the same as the type on the right. Make sure that the expression on the left is a variable name or something else that you can assign a value to (like an element of an array).
6. For each method invocation, make sure that the arguments you provide are in the right order and have the right type, and that the object you are invoking the method on is the right type.
7. If you are invoking a value method, make sure you are doing something with the result. If you are invoking a void method, make sure you are *not* trying to do something with the result.
8. If you are invoking an instance method, make sure you are invoking it on an object with the right type. If you are invoking a static method from outside the class where it is defined, make sure you specify the class name (using dot notation).
9. Inside an instance method, you can refer to the instance variables without specifying an object. If you try that in a static method—with or without `this`—you get a message like “non-static variable x cannot be referenced from a static context.”

If nothing works, move on to the next section...

## I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your development environment to make sure the program you are editing is the program the compiler is compiling.

This situation is often the result of having multiple copies of the same program. You might be editing one version of the file but compiling a different version.

If you are not sure, try putting an obvious and deliberate syntax error right at the beginning of the program. Now compile again. If the compiler doesn't find the new error, there is probably something wrong with the way you set up the development environment.

If you have examined the code thoroughly, and you are sure the compiler is compiling the right source file, it is time for desperate measures—**debugging by bisection**:

- Make a backup of the file you are working on. If you are working on *Bob.java*, make a copy called *Bob.java.old*.
- Delete about half the code from *Bob.java*. Try compiling again.
  - If the program compiles now, you know the error is in the code you deleted. Bring back about half of what you deleted and repeat.
  - If the program still doesn't compile, the error must be in the code that remains. Delete about half of the remaining code and repeat.
- Once you have found and fixed the error, start bringing back the code you deleted, a little bit at a time.

This process is ugly, but it goes faster than you might think and is very reliable. It works for other programming languages too!

## I did what the compiler told me to do, but it still doesn't work.

Some error messages come with tidbits of advice, like “class Golfer must be declared abstract. It does not define `compareTo(java.lang.Object)` from

interface java.lang.Comparable.” It sounds like the compiler is telling you to declare `Golfer` as an `abstract` class, and if you are reading this book, you probably don’t know what that is or how to do it.

Fortunately, the compiler is wrong. The solution in this case is to make sure `Golfer` has a method called `compareTo` that takes an `Object` as a parameter.

Don’t let the compiler lead you by the nose. Error messages give you evidence that something is wrong, but the remedies they suggest are unreliable.

## D.2 Run-Time Errors

It’s not always clear what causes a run-time error, but you can often figure things out by adding print statements to your program.

### My program hangs.

If a program stops and seems to be doing nothing, we say it is “hanging”. Often that means it is caught in an infinite loop or an infinite recursion.

- If you suspect that a particular loop is the problem, add a print statement immediately before the loop that says `"entering the loop"` and another immediately after that says `"exiting the loop"`.

Run the program. If you get the first message and not the second, you know where the program is getting stuck. Go to the section titled “Infinite loop”.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a `StackOverflowError`. If that happens, go to the section titled “Infinite recursion”.

If you are not getting a `StackOverflowError`, but you suspect there is a problem with a recursive method, you can still use the techniques in the infinite recursion section.

- If neither of the previous suggestions helps, you might not understand the flow of execution in your program. Go to the section titled “Flow of execution”.



## Infinite loop

If you think you have an infinite loop and you know which loop it is, add a print statement at the end of the loop that displays the values of the variables in the condition, and the value of the condition.

For example:

```
while (x > 0 && y < 0) {  
    // do something to x  
    // do something to y  
  
    System.out.println("x: " + x);  
    System.out.println("y: " + y);  
    System.out.println("condition: " + (x > 0 && y < 0));  
}
```

Now when you run the program, you see three lines of output for each time through the loop. The last time through the loop, the condition should be **false**. If the loop keeps going, you will see the values of `x` and `y`, and you might figure out why they are not getting updated correctly.

## Infinite recursion

Most of the time, an infinite recursion will cause the program to throw a **StackOverflowError**. But if the program is slow, it may take a long time to fill the stack.

If you know which method is causing an infinite recursion, check that there is a base case. There should be a condition that makes the method return without making a recursive invocation. If not, you need to rethink the algorithm and identify a base case.

If there is a base case, but the program doesn't seem to be reaching it, add a print statement at the beginning of the method that displays the parameters.

Now when you run the program, you see a few lines of output every time the method is invoked, and you can see the values of the parameters. If the parameters are not moving toward the base case, you might see why not.

### Flow of execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each method with a message like `"entering method foo"`, where `foo` is the name of the method. Now when you run the program, it displays a trace of each method as it is invoked.

You can also display the arguments each method receives. When you run the program, check whether the values are reasonable, and check for one of the most common errors—providing arguments in the wrong order.

### When I run the program, I get an exception.

When an exception occurs, Java displays a message that includes the name of the exception, the line of the program where the exception occurred, and a stack trace. The stack trace includes the method that was running, the method that invoked it, the method that invoked that one, and so on.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened:

#### **NullPointerException:**

You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out which variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an array type, its elements are initially `null` until you assign a value to them. For example, this code causes a `NullPointerException`:

```
int[] array = new Point[5];  
System.out.println(array[0].x);
```

#### **ArrayIndexOutOfBoundsException:**

The index you are using to access an array is either negative or greater than `array.length - 1`. If you can find the site where the problem is, add a print statement immediately before it to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backward through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing. If either one is a parameter, go to the place where the method is invoked and see where the values are coming from.

**StackOverflowError:**

See “Infinite recursion” on page 334.

**FileNotFoundException:**

This means Java didn’t find the file it was looking for. If you are using a project-based development environment like Eclipse, you might have to import the file into the project. Otherwise, make sure the file exists and that the path is correct. This problem depends on your filesystem, so it can be hard to track down.

**ArithmeticException:**

Something went wrong during an arithmetic operation; for example, division by zero.

**I added so many print statements I get inundated with output.**

One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: either simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren’t helping, or combine them, or format the output so it is easier to understand. As you develop a program, you should write code to generate concise, informative traces of what the program is doing.

To simplify the program, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the error.

Also, clean up the code. Remove unnecessary or experimental parts, and reorganize the program to make it easier to read. For example, if you suspect

that the error is in a deeply nested part of the program, rewrite that part with a simpler structure. If you suspect a large method, split it into smaller methods and test them separately.

The process of finding the minimal test case often leads you to the bug. For example, if you find that a program works when the array has an even number of elements, but not when it has an odd number, that gives you a clue about what is going on.

Reorganizing the program can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

## D.3 Logic Errors

### My program doesn't work.

Logic errors are hard to find because the compiler and interpreter provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the code and the behavior you get. You need a hypothesis about what the program is actually doing. Here are some questions to ask yourself:

- Is there something the program was supposed to do that doesn't seem to be happening? Find the section of the code that performs that function, and make sure it is executing when you think it should. See "Flow of execution" on page 335.
- Is something happening that shouldn't? Find code in your program that performs that function, and see if it is executing when it shouldn't.
- Is a section of code producing an unexpected effect? Make sure you understand the code, especially if it invokes methods in the Java library. Read the documentation for those methods, and try them out with simple test cases. They might not do what you think they do.

To program, you need a mental model of what your code does. If it doesn't do what you expect, the problem might not actually be the program; it might be in your head.

The best way to correct your mental model is to break the program into components (usually the classes and methods) and test them independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Here are some common logic errors to check for:

- Remember that integer division always rounds toward zero. If you want fractions, use `double`. More generally, use integers for countable things and floating-point numbers for measurable things.
- Floating-point numbers are only approximate, so don't rely on them to be perfectly accurate. You should probably never use the `==` operator with `doubles`. Instead of writing `if (d == 1.23)`, do something like `if (Math.abs(d - 1.23) < .000001)`.
- When you apply the equality operator (`==`) to objects, it checks whether they are identical. If you meant to check equivalence, you should use the `equals` method instead.
- By default for user-defined types, `equals` checks identity. If you want a different notion of equivalence, you have to override it.
- Inheritance can lead to subtle logic errors, because you can run inherited code without realizing it. See “Flow of execution” on page 335.

**I've got a big, hairy expression and it doesn't do what I expect.**

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables:

```
rect.translate((int) Math.round(0.5 * rect.getWidth()),  
              (int) Math.round(0.5 * rect.getHeight()));
```

This example can be rewritten as follows:

```
double halfWidth = 0.5 * rect.getWidth();
double halfHeight = 0.5 * rect.getHeight();
int dx = (int) Math.round(halfWidth);
int dy = (int) Math.round(halfHeight);
rect.translate(dx, dy);
```

The second version is easier to read, partly because the variable names provide additional documentation. It's also easier to debug, because you can check the types of the temporary variables and display their values.

Another problem that can occur with big expressions is that the order of operations may not be what you expect. For example, to evaluate  $\frac{x}{2\pi}$ , you might write this:

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and they are evaluated from left to right. This code computes  $\frac{x}{2}\pi$ .

If you are not sure of the order of operations, check the documentation, or use parentheses to make it explicit.

```
double y = x / (2 * Math.PI);
```

This version is correct, and more readable for other people who haven't memorized the order of operations.

## My method doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to display the value before returning:

```
public Rectangle intersection(Rectangle a, Rectangle b) {
    return new Rectangle(
        Math.min(a.x, b.x), Math.min(a.y, b.y),
        Math.max(a.x + a.width, b.x + b.width)
            - Math.min(a.x, b.x)
        Math.max(a.y + a.height, b.y + b.height)
            - Math.min(a.y, b.y));
}
```

Instead of writing everything in one statement, use temporary variables:

```
public Rectangle intersection(Rectangle a, Rectangle b) {  
    int x1 = Math.min(a.x, b.x);  
    int y1 = Math.min(a.y, b.y);  
    int x2 = Math.max(a.x + a.width, b.x + b.width);  
    int y2 = Math.max(a.y + a.height, b.y + b.height);  
    Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);  
    return rect;  
}
```

Now you have the opportunity to display any of the intermediate variables before returning. And by reusing `x1` and `y1`, you made the code smaller too.

## My print statement isn't doing anything.

If you use the `println` method, the output is displayed immediately, but if you use `print` (at least in some environments), the output gets stored without being displayed until the next newline. If the program terminates without displaying a newline, you may never see the stored output. If you suspect that this is happening, change some or all of the `print` statements to `println`.

## I'm really, really stuck and I need help.

First, get away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and rage.
- Superstitious beliefs (“the computer hates me”) and magical thinking (“the program works only when I wear my hat backward”).
- Sour grapes (“this program is lame anyway”).

If you suffer from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. People often find bugs when they let their mind wander. Good places to find bugs are buses, showers, and bed.

## No, I really need help.

It happens. Even the best programmers get stuck. Sometimes you need another pair of eyes. Before you bring someone else in, make sure you have tried the techniques described in this appendix.

Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, give them the information they need:

- What kind of bug is it? Compile-time, run-time, or logic?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the test case that fails?
- If the bug occurs at compile time or run time, what is the error message, and what part of the program does it indicate?
- What have you tried, and what have you learned?

By the time you explain the problem to someone, you might see the answer. This phenomenon is so common that some people recommend a debugging technique called “rubber ducking”. Here’s how it works:

1. Buy a standard-issue rubber duck.
2. When you are really stuck on a problem, put the rubber duck on the desk in front of you and say, “Rubber duck, I am stuck on a problem. Here’s what’s happening...”
3. Explain the problem to the rubber duck.
4. Discover the solution.
5. Thank the rubber duck.

We’re not kidding, it works! See [https://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging).



## I found the bug!

When you find the bug, the way to fix it is usually obvious. But not always. Sometimes what seems to be a bug is really an indication that you don't understand the program, or your algorithm contains an error. In these cases, you might have to rethink the algorithm or adjust your mental model. Take some time away from the computer to think, work through test cases by hand, or draw diagrams to represent the computation.

After you fix the bug, don't just start in making new errors. Take a minute to think about what kind of bug it was, why you made the error, how the error manifested itself, and what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly. Or even better, you will learn to avoid that type of bug for good.

# Index

## Symbols

( ) parentheses, 26  
; semicolon, 4, 27  
= assignment operator, 18  
== equals operator, 71, 192  
[ ] square brackets, 108  
% remainder operator, 42  
<> angle brackets, 227  
{ } curly braces, 4, 73

## A

abecedarian, 105  
abstract class, 274, 276  
accessor, 189  
accumulator, 114, 122  
addition  
    integer, 22  
    string, 25  
    time, 194  
address, 33, 46, 191  
algorithm, 10, 13  
alias, 112, 122  
aliasing, 173, 222  
allocate, 108, 122  
anagram, 125  
and operator, 77  
angle brackets, 227  
args, 152

argument, 54, 59, 65  
ArithmeticException, 28, 336  
array, 108, 121  
    2D, 215  
    copying, 111  
    element, 109  
    index, 109  
    length, 112  
    of cards, 217  
    of objects, 208  
    of strings, 203  
    printing, 110  
ArrayIndexOutOfBoundsException,  
    110, 335  
ArrayList, 227, 236  
Arrays class, 111, 112  
assignment, 18, 30, 71  
attribute, 168, 180  
automatic conversion, 24, 41, 55, 72  
AWT, 167, 321, 326

## B

base case, 130, 141  
BigInteger, 156  
binary, 135, 141  
binary search, 211, 214  
block, 72, 84  
Boole, George, 71

boolean, 71, 80, 84  
bottom-up design, 244, 249  
bounding box, 323, 326  
brackets  
    angle, 227  
    square, 108  
branch, 72, 84  
breakpoint, 306, 310  
bug, 10, 13  
byte code, 6, 13

## C

call stack, 130, 306, 310  
camel case, 52  
Canvas, 321  
Card, 202  
CardCollection, 236  
case-sensitive, 4, 18, 52, 101  
catch, 260  
chaining, 74, 84  
char, 7, 95  
Character, 151, 154  
charAt, 95  
Checkstyle, 305  
class, 4, 12, 196  
    definition, 183  
    relationships, 248  
    utility, 34  
    wrapper, 151  
class diagram, 174, 180, 225, 248  
class variable, 205, 214  
client, 188, 197  
CodingBat, 138  
collection, 227, 230  
Color, 323  
command-line interface, 152, 302,  
    310, 319  
comment, 12, 84

    documentation, 311, 314  
    end-of-line, 4, 311  
    multiline, 311  
compareTo, 101, 207  
comparison operator, 71  
compile, 5, 13, 330  
compile-time error, 27, 30, 329  
complete ordering, 206  
composition, 59, 65, 248  
computer science, 10, 13  
concatenate, 25, 30, 161  
concrete class, 276  
conditional statement, 72, 84  
constant, 38, 47  
constructor, 185, 197, 203, 217, 222  
    value, 186  
Convert.java, 43, 317  
coordinate, 167, 322, 326  
countdown, 127  
counter, 116  
CPU, 2  
Crazy Eights, 235

## D

De Morgan's laws, 78, 84  
debugger, 306, 310  
debugging, 10, 13, 329  
    by bisection, 332  
    experimental, 11  
    rubber duck, 341  
Deck, 217  
declaration, 17, 30, 167  
decrement, 91, 103  
degrees, 58  
dependent, 189  
description, 315, 320  
design process, 62, 157, 161, 220,  
    244, 273

deterministic, 115, 122  
diagram  
    class, 174, 225, 248  
    memory, 20, 99, 112, 147, 168,  
        186, 193, 204, 217  
    stack, 57, 129, 132, 137  
divisible, 43  
division  
    floating-point, 23  
    integer, 22, 24  
documentation, 44, 174, 312, 320  
    Javadoc comments, 314  
    Javadoc tags, 315  
dot notation, 168, 180  
Double, 151  
double, 23  
doubloon, 105  
Doubloon.java, 120  
DrJava, 300

## E

Echo.java, 35  
efficiency, 117, 163, 211, 221, 231,  
    237, 295  
Eights, 245  
element, 108, 109, 121  
empty array, 153, 161  
empty string, 98, 103  
encapsulate, 157, 161  
encapsulation, 165, 171  
    and generalization, 157  
encode, 202, 214  
enhanced for loop, 118, 122  
equals, 100, 192, 193  
equivalent, 192, 197, 206  
error  
    compile-time, 27, 329  
    logic, 29, 329, 338

    message, 11, 27, 330  
    rounding, 24  
    run-time, 28, 329  
    syntax, 331  
escape sequence, 9, 13, 95  
exception, 28, 329, 335  
    Arithmetic, 28  
    ArrayIndexOutOfBoundsException, 110  
    InputMismatch, 82  
    Interrupted, 260  
    MissingFormatArgument, 40  
    NegativeArraySize, 108  
    NullPointerException, 149, 154, 209  
    NumberFormat, 152  
    StackOverflow, 130  
    StringIndexOutOfBoundsException, 97,  
        155  
executable, 5, 13  
experimental debugging, 11  
expression, 22, 30, 59, 79  
    big and hairy, 338  
extends, 238, 239  
extract digits, 43

## F

factorial, 130, 141, 162  
fibonacci, 134  
FileNotFoundException, 336  
final, 38, 205, 208  
flag, 79, 84  
floating-point, 23, 30  
flow of execution, 53, 64, 335  
for, 92  
    enhanced, 118  
format specifier, 39, 47, 94  
format string, 39, 47, 190  
frame, 57, 65

**G**

garbage collection, 176, 180, 205  
generalization, 165, 171, 279, 297  
generalize, 158, 161  
getter, 189, 197  
GitHub, xviii  
Goodbye.java, 7  
Google style, 8  
Graphics, 256, 321  
Greenfield, Larry, 11  
GuessStarter.java, 49

**H**

hacker, 81, 85, 154  
Hand, 240  
hanging, 333  
hardware, 2, 12  
HAS-A, 248, 249, 256, 275  
Hello.java, 3  
helper method, 219, 230  
hexadecimal, 33, 40, 191  
high-level language, 5, 12  
histogram, 116, 122, 215  
HTML, 174, 311, 316

**I**

IDE, 299, 309  
identical, 192, 197  
if statement, 72  
ImageIO, 292  
immutable, 150, 161, 177, 208  
import statement, 35, 47  
increment, 91, 102  
incremental development, 62, 65  
independent, 189  
index, 103, 109, 121, 209  
indexOf, 98  
infinite loop, 90, 102, 333

infinite recursion, 186, 334  
information hiding, 184, 197  
inheritance, 238, 248, 249, 279, 290  
initialize, 19, 30, 79  
inner loop, 94  
InputMismatchException, 82  
instance, 183, 196  
instance method, 191, 195, 197  
instance variable, 184, 196  
instantiate, 183, 196  
Integer, 151  
integer division, 22, 24  
interactions, 301  
interface, 291  
interpret, 5, 13  
InterruptedException, 260  
invoke, 51, 64  
IS-A, 248, 249, 256, 275, 287  
iteration, 92, 97, 103  
iterative, 127, 140

**J**

JAR, 300, 309  
Java Tutor, 57, 140  
java.awt, 167, 321  
java.io, 33  
java.lang, 35  
java.math, 156  
java.util, 34  
javac, 6, 154, 302  
Javadoc, 174, 300, 311, 316, 320  
javax.imageio, 292  
JDK, 300, 309, 311  
JFrame, 259, 322  
JVM, 6, 300, 309

**K**

keyword, 18, 30, 185

**L**

- language
  - elements, 36
  - high-level, 5
  - low-level, 5
- leap of faith, 133, 141, 224
- length
  - array, 112
  - string, 97
- library, 34, 46, 173
- Linux, 11
- literal, 38, 47
- local variable, 55, 65
- Logarithm.java, 83
- logic error, 29, 31, 329, 338
- logical operator, 77, 84, 206
- long, 58
- loop, 90, 102
  - for, 92
  - infinite, 90
  - nested, 94, 209
  - search, 210
  - while, 89
- loop body, 90, 102
- loop variable, 94, 103, 110
- low-level language, 5, 12

**M**

- magic number, 38, 47
- main, 4, 52
- map to, 202
- Math, 58
- memory, 2, 12
- memory diagram, 20, 30, 99, 112, 147, 168, 186, 193, 204, 217
- mental model, 57, 338
- merge, 223
- merge sort, 221, 230

- method, 4, 12
  - abstract, 274
  - accessor, 189
  - boolean, 80
  - constructor, 185
  - equals, 192, 193
  - getter, 189
  - helper, 219
  - instance, 191, 195
  - modifier, 238
  - mutator, 190
  - parameters, 55
  - setter, 190
  - static, 194
  - toString, 191
  - value, 60
- Mickey Mouse, 324
- MissingFormatArgumentException, 40
- modifier method, 238
- modulo, 42, 47
- modulus, 43, 47
- multidimensional array, 254, 265
- mutable, 171, 177, 180
- mutator, 190

**N**

- NaN, 85
- NegativeArraySizeException, 108
- neighbor, 251, 261
- nested, 336
  - arrays, 256
  - conditions, 75
  - loops, 94
- nesting, 74, 84, 209
- new, 35, 49, 108, 167, 185
- newline, 7, 13, 128
- NewLine.java, 52

- next
  - Scanner, 82
- nextInt
  - Random, 115
  - Scanner, 37
- nondeterministic, 115, 122
- not operator, 77
- null, 149, 208
- NullPointerException, 149, 154, 335
- NumberFormatException, 152

## O

- object, 147, 161
  - array of, 208
  - as parameter, 169
  - displaying, 191
  - mutable, 171
  - type, 183
- Object class, 239
- object code, 5, 13
- object-oriented, 147, 160, 172, 248, 276
- off-by-one, 222, 230
- operand, 22, 30
- operator, 22, 30
  - assignment, 71
  - cast, 42
  - logical, 77, 206
  - new, 35, 49, 108, 167, 185
  - redirection, 304
  - relational, 71
  - remainder, 42
  - string, 25
- or operator, 77
- order of operations, 26, 30, 339
- ordering, 206
- outer loop, 94

- overload, 186, 222
- overloaded, 99, 103
- override, 191, 197, 249

## P

- package, 33, 46
- paint, 322
- palindrome, 145
- param tag, 315
- parameter, 54, 65, 169
  - multiple, 55
- parameter passing, 54, 65
- parentheses, 26
- parse, 28, 31, 152, 161
- partial ordering, 206
- pi, 58
- Pile, 227
- pixel, 323, 326
- Player, 242
- Point, 167
- polymorphism, 287, 291, 297
- portable, 5, 13
- precedence, 26, 339
- primitive, 147, 161
- print, 7, 191
- print statement, 3, 12, 190, 336, 340
- printArray, 111
- printDeck, 209, 218
- printf, 39, 101, 190
- println, 4
- printPoint, 169
- printTime, 190
- PrintTime.java, 56
- PrintTwice.java, 54
- private, 174, 184, 188
- problem solving, 1, 12
- processor, 2, 12
- program, 2, 12

programming, 3, 12  
prompt, 37, 47  
protected, 274  
pseudocode, 219, 230  
pseudorandom, 115, 122  
public, 51

## Q

quote mark, 7, 95, 331

## R

radians, 58  
RAM, 2  
Random, 115  
randomInt, 220  
rank, 201  
rational number, 198  
Rectangle, 170  
recursion, 130  
    infinite, 186, 334  
recursive, 127, 140  
redirection operator, 304, 310  
reduce, 114, 116, 122  
refactor, 273, 276  
reference, 109, 122, 167, 172, 204,  
    222  
relational operator, 71, 84  
remainder, 42  
replace, 150  
repository, xviii  
return, 61, 83, 170  
    inside loop, 211  
return statement, 339  
return tag, 315  
return type, 60, 65  
return value, 60, 65  
RGB, 324, 326  
rounding error, 24, 30

row-major order, 255, 265  
rubber duck, 341  
run-time error, 28, 31, 82, 209, 329

## S

scaffolding, 63, 65  
Scanner, 34  
scope, 57, 65, 175  
Scrabble, 105, 197  
search, 114, 122  
selection sort, 220, 230  
semicolon, 4, 27  
sequential search, 210, 214  
setter, 190, 197  
shadowing, 188, 197  
short circuit, 78, 84, 155  
shuffle, 219  
signature, 313, 320  
sleep, 260  
sort  
    merge, 221  
    selection, 220  
source code, 5, 13, 173  
specialization, 279, 297  
sprite, 292, 297  
src.zip, 173  
stable configuration, 252  
stack diagram, 57, 65, 129, 132, 137  
stack trace, 41, 47, 335  
StackOverflowError, 130, 186, 333,  
    336  
state, 20, 30, 45, 129, 242, 253, 270  
statement, 3, 12  
    assignment, 18  
    catch, 260  
    conditional, 72  
    declaration, 17, 167  
    else, 72



- for, 92
- if, 72
- import, 35
- initialization, 79
- print, 7, 191, 336, 340
- return, 61, 83, 170, 211, 339
- switch, 75
- throw, 285
- try, 260
- while, 89
- static, 185, 194, 205
- static context, 226, 230
- string, 7, 13
  - array of, 203
  - comparing, 100, 148
  - format, 102, 192
  - length, 97
  - operator, 25
  - reference to, 204
- String class, 167
- StringBuilder, 179, 231
- StringIndexOutOfBoundsException,
  - 97, 155
- stub, 62, 65
- style guide, 8
- subclass, 238, 249
- subdeck, 222
- substring, 99
- suit, 201
- superclass, 239, 249
- Surprise.java, 177
- swapCards, 220
- switch statement, 75
- syntax, 329
- syntax errors, 331
- System.err, 82, 309
- System.in, 34, 147, 304
- System.out, 33, 147, 304

## T

- table
  - two-dimensional, 157
- tag, 315, 320
- temporary variable, 61, 65, 339
- terminal, 302
- testing, 63, 82, 232, 303
- text editor, 300, 309
- this, 176, 185, 226, 236
- Thread.sleep, 260
- throw, 285
- Time, 184
  - addition, 194
- toCharArray, 120
- token, 36, 47
- toLowerCase, 150
- top-down design, 220, 230, 244
- Torvalds, Linus, 11
- toString, 169, 191
- toUpperCase, 150
- tracing, 57, 213, 306, 335
- traversal, 113, 122, 210, 256
- try, 260
- type, 30
  - array, 108
  - boolean, 71, 79
  - char, 17, 95
  - double, 23
  - int, 17
  - long, 25, 58
  - object, 183
  - String, 7, 17, 167
  - void, 51
- type cast, 42, 47

## U

- UML, 174, 180, 225, 248
- Unicode, 96, 103

unit test, 307, 310  
utility class, 34, 111

## V

validate, 81, 85, 154  
value, 17, 30  
value constructor, 186  
value method, 60  
variable, 17, 29

- instance, 184
- local, 55
- loop, 94, 110

- private, 174, 184, 188
- static, 205
- temporary, 61, 339

virtual machine, 6, 13, 300  
void, 51, 60, 64

## W

War (card game), 227  
while, 89  
wildcard, 305, 310  
wrapper class, 151, 161  
wrapper method, 231

# About the Authors

**Allen Downey** is a professor of computer science at Olin College of Engineering. He has taught computer science at Wellesley College, Colby College, and UC Berkeley. He has a PhD in computer science from UC Berkeley and master's and bachelor's degrees from MIT. Allen is the creator of the best-selling Think series for O'Reilly, which includes *Think Python*, *Think Complexity*, *Think DSP*, and *Think Bayes*.

**Chris Mayfield** is an associate professor of computer science at James Madison University, with a research focus on CS education and professional development. He has a PhD in computer science from Purdue University and bachelor's degrees in CS and German from the University of Utah.