Pearson New International Edition

The Art and Science of Java

An Introduction to Computer Science

Eric S. Roberts

First Edition

ALWAYS LEARNING

PEARSON

Pearson New International Edition

The Art and Science of Java
An Introduction to Computer Science
Eric S. Roberts
First Edition

PEARSON

As the header lines indicate, the **drawEngine**, **drawBoxcar**, and **drawCaboose** methods take an **x** and a **y** parameter to indicate the location of the car on the canvas. Callers that use these methods need to know what those coordinates mean in terms of where the car is drawn relative to the point (**x**, **y**). There are many possible interpretations, and part of your job as programmer is to choose an appropriate design. One possibility is to adopt the convention used by the **acm.graphics** package and define the point (**x**, **y**) to be the upper left corner of a car. In the absence of a compelling reason to the contrary, choosing a model that matches the existing library packages is a good idea. In this case, however, there may indeed be a compelling reason. The location of the upper left corner of a train car depends on how tall the car is. In this example, the engine and the caboose have graphical figures on their top side that make them taller than the boxcar. Thus, to calculate the *y* coordinate of the top of a car, you would need to know the type of car. On the other hand, it is easy to calculate the *y* coordinate of the bottom of the car because all the cars are sitting on a track. Because all cars have a common baseline, it makes sense to let the **x** parameter indicate the left edge of the connector that extends from the car and to let the **y** parameter indicate the level of the track. This interpretation is therefore similar to the one that the **acm.graphics** package uses for the **GLabel** class. Because each letter rests on a horizontal baseline, the *y* coordinate in the **GLabel** class indicates the position of the baseline.

Designing from the top down

Now that you have decided on the method headers for the **drawEngine**, **drawBoxcar**, and **drawCaboose** methods, it might seem as if the logical next step would be to go ahead and implement them. Although doing so is indeed an option, it is usually better to complete the code at each level of decomposition before moving on to the next. By doing so, you can convince yourself that you have chosen the right decomposition. If you have left out a subtask or failed to include enough flexibility in the arguments to the methods, you will discover the problem when you try to code the highest level of the program. The principle of top-down design suggests that you should start with the **run** method and then work your way down toward the details.

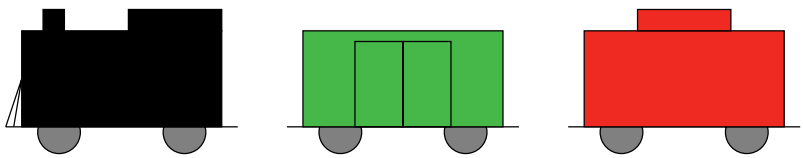
The decomposition of the problem into subsidiary methods makes the **run** method reasonably straightforward. You know that the body of the **run** method will include calls to **drawEngine**, **drawBoxcar**, and **drawCaboose**. The only remaining issue is figuring out what values to supply for the coordinates in each call. If you look at picture in the sample run, you will see that the train is resting on the bottom of the canvas. The *y* coordinate of the track is therefore equal to the *y* coordinate of the bottom of the canvas; because Java coordinate values increase as you move downward, the *y* coordinate of the bottom of the canvas is simply the height of the canvas, which you can get by calling the **getHeight** method. Finding the *x* coordinate of each car requires a little more calculation. In this example, the entire train is centered in the window. Thus, calculating the *x* coordinate for the engine requires figuring out how long the entire train is and then subtracting half that length from the coordinates of the center of the canvas. Each subsequent car begins

at a point whose *x* coordinate is shifted rightward by the width of a car and the width of its connector. Expressing these calculations in Java gives rise to the following **run** method:

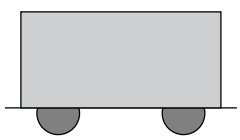
```
public void run() {
   double trainWidth = 3 * CAR_WIDTH + 4 * CONNECTOR;
   double x = (getWidth() - trainWidth) / 2;
   double y = getHeight();
   double dx = CAR_WIDTH + CONNECTOR;
   drawEngine(x, y);
   drawBoxcar(x + dx, y, Color.GREEN);
   drawCaboose(x + 2 * dx, y);
}
```

Looking for common features

Now that the **run** method is out of the way, you can turn your attention to the methods that draw the individual cars. Although one is often tempted to run to the keyboard and start typing in code, there is an enormous advantage in taking the time to think about the problem. When you choose a decomposition, one of the things you should look for is subtasks that come up in more than one part of the problem. To see how that strategy might apply in the current problem, it's worth taking another look at the three different types of cars:



If you look carefully at the diagrams of these three cars, you will see that they share a number of common features. The wheels are the same, as are the connectors that link the car to its neighbors. In fact, the body of the car itself is the same except for the color. Each type of car, therefore, shares a common framework that looks like this:



If you color the gray part of this picture with the appropriate color, you can use it as the foundation for any of the three car types. For the engine, you need to add a smokestack, a cab, and a cowcatcher. For the boxcar, you need to add doors. For the caboose, you need to add a cupola. In each case, most of the work of drawing the car can be shared by defining a method that draws the basic framework of the car. That method—which itself decomposes into a method that draws each of the wheels—looks like this:

```
private void drawCarFrame(double x, double y, Color color) {
   double x0 = x + CONNECTOR;
   double y0 = y - CAR_BASELINE;
   double top = y0 - CAR_HEIGHT;
   add(new GLine(x, y0, x + CAR_WIDTH + 2 * CONNECTOR, y0));
   drawWheel(x0 + WHEEL_INSET, y - WHEEL_RADIUS);
   drawWheel(x0 + CAR_WIDTH - WHEEL_INSET, y - WHEEL_RADIUS);
   GRect r = new GRect(x0, top, CAR_WIDTH, CAR_HEIGHT);
   r.setFilled(true);
   r.setFillColor(color);
   add(r);
}

private void drawWheel(double x, double y) {
   double r = WHEEL_RADIUS;
   GOval wheel = new GOval(x - r, y - r, 2 * r, 2 * r);
   wheel.setFilled(true);
   wheel.setFillColor(Color.GRAY);
   add(wheel);
}
```

Completing the decomposition

Once you have a tool for creating the framework of an individual car, you can complete the **drawTrain** program by filling in the definitions of the **drawEngine**, **drawBoxcar**, and **drawCaboose** methods. Some of the methods are easy enough to code without further decomposition. Here, for example, is an implementation of **drawBoxcar** that draws the background frame and then adds the two door panels:

```
private void drawBoxcar(double x, double y, Color color) {
   drawCarFrame(x, y, color);
   double xRight = x + CONNECTOR + CAR_WIDTH / 2;
   double xLeft = xRight - DOOR_WIDTH;
   double yDoor = y - CAR_BASELINE - DOOR_HEIGHT;
   add(new GRect(xLeft, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
   add(new GRect(xRight, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
}
```

You might, however, choose to decompose some of these methods further. For example, you could decide to break up the diagram of the train engine into its component parts and then code **drawEngine** like this:

```
private void drawEngine(double x, double y) {
   drawCarFrame(x, y, Color.BLACK);
   drawSmokestack(x, y);
   drawCab(x, y);
   drawCowcatcher(x, y);
}
```

Private methods that tackle smaller parts of a problem, such as **drawSmokestack**, **drawCab**, and **drawCowcatcher** in this example, are often called **helper methods.**

If you adopt this decomposition strategy, you will have to write implementations for these helper methods along with the as-yet-unimplemented **drawCaboose**. You'll have a chance to finish the decomposition in exercise 8.

While stepwise refinement is a critically important skill, object-oriented languages offer another approach to simplifying a problem. Instead of dividing a program into methods that implement successively simpler subtasks, you can usually accomplish the same goal by defining a hierarchy of classes that reflects the same decomposition strategy. When you are programming in Java, that strategy usually has distinct advantages. In Chapter 9, you'll have a chance to reimplement the **`DrawTrain`** program by defining a class hierarchy in which each of the specific car types—engines, boxcars, and cabooses—is a subclass of a more general class that encompasses each of the individual types.

5.5 Algorithmic methods

In addition to their role as a tool for managing complex programs, methods are important to programming because they provide a basis for the implementation of algorithms, which were introduced briefly in Chapter 1. An algorithm is an abstract strategy; writing a method is the conventional way to express that algorithm in the context of a programming language. Thus, when you want to implement an algorithm as part of a program, you typically write a method—which may in turn call other methods to handle part of its work—to carry out that algorithm.

Although you have seen several algorithms implemented in the context of the sample programs, you have not had a chance to focus on the nature of the algorithmic process itself. Most of the programming problems you have seen so far are simple enough that the appropriate solution technique springs immediately to mind. As problems become more complex, however, their solutions require more thought, and you will need to consider more than one strategy before writing the final program.

As an illustration of how algorithmic strategies take shape, the sections that follow consider two solutions to a problem from classical mathematics, which is to find the greatest common divisor of two integers. Given two integers *x* and *y,* the **greatest common divisor** (or **gcd** for short) is the largest integer that divides evenly into both. For example, the gcd of 49 and 35 is 7, the gcd of 6 and 18 is 6, and the gcd of 32 and 33 is 1.

Suppose that you have been asked to write a method that accepts the integers *x* and *y* as input and returns their greatest common divisor. From the caller's point of view, what you want is a method **`gcd(x, y)`** that takes two integers as arguments and returns another integer that is their greatest common divisor. The header line for this method is therefore

```
public int gcd(int x, int y)
```

How might you go about designing an algorithm to perform this calculation?

The "brute force" approach

In many ways, the most obvious approach to calculating the gcd is simply to try every possibility. To start, you simply "guess" that **`gcd(x, y)`** is the smaller of **`x`**

and **y**, because any larger value could not possibly divide evenly into a smaller number. You then proceed by dividing **x** and **y** by your guess and seeing if it divides evenly into both. If it does, you have the answer; if not, you subtract 1 from your guess and try again. A strategy that tries every possibility is often called a **brute force approach.**

The brute-force approach to calculating the **gcd** function looks like this in Java:

```
public int gcd(int x, int y) {
    int guess = Math.min(x, y);
    while (x % guess != 0 || y % guess != 0) {
        guess--;
    }
    return guess;
}
```

Before you decide that this implementation is in fact a valid algorithm for computing the **gcd** function, you need to ask yourself several questions about the code. Will the brute-force implementation of **gcd** always give the correct answer? Will it always terminate, or might the method continue forever?

To see that the program gives the correct answer, you need to look at the condition in the **while** loop

```
x % guess != 0 || y % guess != 0
```

As always, the **while** condition indicates under what circumstances the loop will continue. To find out what condition causes the loop to terminate, you have to negate the **while** condition. Negating a condition involving **&&** or **||** can be tricky unless you remember how to apply De Morgan's law, which was introduced in the section on "Logical operators" in Chapter 4. De Morgan's law indicates that the following condition must hold when the **while** loop exits:

```
x % guess == 0 && y % guess == 0
```

From this condition, you can see immediately that the final value of **guess** is certainly a common divisor. To recognize that it is in fact the greatest common divisor, you have to think about the strategy embodied in the **while** loop. The critical factor to notice in the strategy is that the program counts *backward* through all the possibilities. The greatest common divisor can never be larger than **x** (or **y**, for that matter), and the brute-force search therefore begins with that value. If the program ever gets out of the **while** loop, it must have already tried each value between **x** and the current value of **guess**. Thus, if there were a larger value that divided evenly into both **x** and **y**, the program would already have found it in an earlier iteration of the **while** loop.

To recognize that the method terminates, the key insight is that the value of **guess** must eventually reach 1, even if no larger common divisor is found. At this point, the **while** loop will surely terminate, because 1 will divide evenly into both **x** and **y**, no matter what values those variables have.

Euclid's algorithm

Brute force is not, however, the only effective strategy. Although brute-force algorithms have their place in other contexts, they are a poor choice for the **gcd** function if you are concerned about efficiency. For example, if you call the method with the integers 1,000,005 and 1,000,000, the brute-force algorithm will run through the body of the **while** loop almost a million times before it comes up with 5—an answer that you can easily determine just by thinking about the two numbers.

What you need to find is an algorithm that is guaranteed to terminate with the correct answer but that requires fewer steps than the brute-force approach. This is where cleverness and a clear understanding of the problem pay off. Fortunately, the necessary creative insight has already been supplied by the Greek mathematician Euclid, whose *Elements* (book 7, proposition II) contains an elegant solution to this problem. In modern English, Euclid's algorithm can be described as follows:

1. Divide **x** by **y** and compute the remainder; call that remainder **r**.
2. If **r** is zero, the procedure is complete, and the answer is **y**.
3. If **r** is not zero, set **x** equal to the old value of **y**, set **y** equal to **r**, and repeat the entire process.

You can easily translate this algorithmic description into the following Java code:

```
int gcd(int x, int y) {
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

This implementation of the **gcd** method also correctly finds the greatest common divisor of two integers. It differs from the brute-force implementation in two respects. On the one hand, it computes the result much more quickly. On the other, it is more difficult to prove correct.

Defending the correctness of Euclid's algorithm

Although a formal proof of correctness for Euclid's algorithm is beyond the scope of this book, you can easily get a feel for how the algorithm works by adopting the mental model of mathematics the Greeks used. In Greek mathematics, geometry held center stage, and numbers were thought of as distances. For example, when Euclid set out to find the greatest common divisor of two whole numbers, such as 55 and 15, he framed the problem as one of finding the longest measuring stick that could be used to mark off each of the two distances involved. Thus, you can visualize the specific problem by starting out with two sticks, one 55 units long and one 15 units long, as follows: