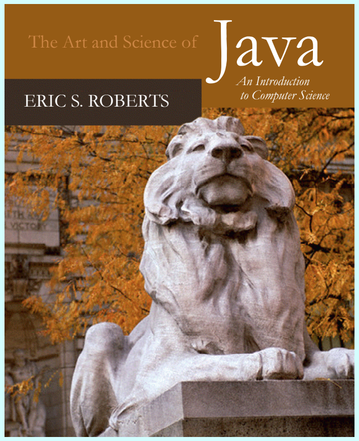
CHAPTER 5

Methods

With method and logic one can accomplish anything.

—Agatha Christie, *Poirot Investigates,* 1924



A Quick Overview of Methods

- You have been working with methods ever since you wrote your first Java program in Chapter 2. The **`run`** method in every program is just one example. Most of the programs you have seen have used other methods as well, such as **`println`** and **`setColor`**.
- At the most basic level, a **method** is a sequence of statements that has been collected together and given a name. The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the method name.
- The following terms are useful when learning about methods:
  - Invoking a method using its name is known as **calling** that method.
  - The caller can pass information to a method by using **arguments**.
  - When a method completes its operation, it **returns** to its caller.
  - A method can pass information to the caller by **returning a result**.

Methods and Information Hiding

- One of the most important advantages of methods is that they make it possible for callers to ignore the inner workings of complex operations.
- When you use a method, it is more important to know what the method does than to understand exactly how it works. The underlying details are of interest only to the programmer who implements a method. Programmers who use a method as a tool can usually ignore the implementation altogether.
- The idea that callers should be insulated from the details of method operation is the principle of **information hiding**, which is one of the cornerstones of software engineering.

Methods as Tools for Programmers

- Particularly when you are first learning about programming, it is important to keep in mind that methods are not the same as application programs, even though both provide a service that hides the underlying complexity involved in computation.
- The key difference is that an application program provides a service to a *user,* who is typically not a programmer but rather someone who happens to be sitting in front of the computer. By contrast, a method provides a service to a *programmer,* who is typically creating some kind of application.
- This distinction is particularly important when you are trying to understand how the applications-level concepts of input and output differ from the programmer-level concepts of arguments and results. Methods like **`readInt`** and **`println`** are used to communicate with the user and play no role in communicating information from one part of a program to another.

Method Calls as Expressions

- Syntactically, method calls in Java are part of the expression framework. Methods that return a value can be used as terms in an expression just like variables and constants.
- The **`Math`** class in the **`java.lang`** package defines several methods that are useful in writing mathematical expressions. Suppose, for example, that you need to compute the distance from the origin to the point (x, y), which is given by

  $$\sqrt{x^2 + y^2}$$

  You can apply the square root function by calling the **`sqrt`** method in the **`Math`** class like this:

```
double distance = Math.sqrt(x * x + y * y);
```

- Note that you need to include the name of the class along with the method name. Methods like **`Math.sqrt`** that belong to a class are called **static methods**.

Useful Methods in the `Math` Class

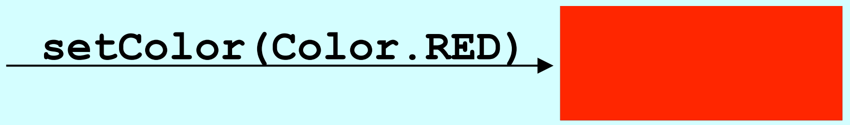
`Math.abs(`*x*`)`	Returns the absolute value of *x*
`Math.min(`*x*`, `*y*`)`	Returns the smaller of *x* and *y*
`Math.max(`*x*`, `*y*`)`	Returns the larger of *x* and *y*
`Math.sqrt(`*x*`)`	Returns the square root of *x*
`Math.log(`*x*`)`	Returns the natural logarithm of *x* ($\log_e x$)
`Math.exp(`*x*`)`	Returns the inverse logarithm of *x* (e^x)
`Math.pow(`*x*`, `*y*`)`	Returns the value of *x* raised to the *y* power (x^y)
`Math.sin(`*theta*`)`	Returns the sine of *theta,* measured in radians
`Math.cos(`*theta*`)`	Returns the cosine of *theta*
`Math.tan(`*theta*`)`	Returns the tangent of *theta*
`Math.asin(`*x*`)`	Returns the angle whose sine is *x*
`Math.acos(`*x*`)`	Returns the angle whose cosine is *x*
`Math.atan(`*x*`)`	Returns the angle whose tangent is *x*
`Math.toRadians(`*degrees*`)`	Converts an angle from degrees to radians
`Math.toDegrees(`*radians*`)`	Converts an angle from radians to degrees

Method Calls as Messages

- In object-oriented languages like Java, the act of calling a method is often described in terms of **sending a message** to an object. For example, the method call

  ```
  rect.setColor(Color.RED);
  ```

  is regarded metaphorically as sending a message to the **`rect`** object asking it to change its color.



- The object to which a message is sent is called the **receiver**.
- The general pattern for sending a message to an object is

  receiver`.`*name*`(`*arguments*`);`

Writing Your Own Methods

- The general form of a method definition is

```
scope  type  name(argument list) {
      statements in the method body
}
```

  where *scope* indicates who has access to the method, *type* indicates what type of value the method returns, *name* is the name of the method, and *argument list* is a list of declarations for the variables used to hold the values of each argument.

- The most common value for *scope* is **`private`**, which means that the method is available only within its own class. If other classes need access to it, *scope* should be **`public`** instead.

- If a method does not return a value, *type* should be **`void`**. Such methods are sometimes called **procedures**.

Returning Values from a Method

- You can return a value from a method by including a **`return`** statement, which is usually written as

```
return expression;
```

  where *expression* is a Java expression that specifies the value you want to return.

- As an example, the method definition

```
private double feetToInches(double feet) {
   return 12 * feet;
}
```

  converts an argument indicating a distance in feet to the equivalent number of inches, relying on the fact that there are 12 inches in a foot.

Methods Involving Control Statements

- The body of a method can contain statements of any type, including control statements. As an example, the following method uses an **`if`** statement to find the larger of two values:

```
private int max(int x, int y) {
   if (x > y) {
      return x;
   } else {
      return y;
   }
}
```

- As this example makes clear, **`return`** statements can be used at any point in the method and may appear more than once.

The `factorial` Method

- The **factorial** of a number *n* (which is usually written as *n*! in mathematics) is defined to be the product of the integers from 1 up to *n*. Thus, 5! is equal to 120, which is $1 \times 2 \times 3 \times 4 \times 5$.
- The following method definition uses a **`for`** loop to compute the factorial function:

```
private int factorial(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

Nonnumeric Methods

Methods in Java can return values of any type. The following method, for example, returns the English name of the day of the week, given a number between 0 (Sunday) and 6 (Saturday):

```
private String weekdayName(int day) {
   switch (day) {
     case 0: return "Sunday";
     case 1: return "Monday";
     case 2: return "Tuesday";
     case 3: return "Wednesday";
     case 4: return "Thursday";
     case 5: return "Friday";
     case 6: return "Saturday";
     default: return "Illegal weekday";
   }
}
```

Methods Returning Graphical Objects

- The text includes examples of methods that return graphical objects. The following method creates a filled circle centered at the point (**x**, **y**), with a radius of **r** pixels, which is filled using the specified color:

```
private GOval createFilledCircle(double x, double y,
                                 double r, Color color) {
   GOval circle = new GOval(x - r, y - r, 2 * r, 2 * r);
   circle.setFilled(true);
   circle.setColor(color);
   return circle;
}
```

- If you are creating a **GraphicsProgram** that requires many filled circles in different colors, the **createFilledCircle** method turns out to save a considerable amount of code.

Predicate Methods

- Methods that return Boolean values play an important role in programming and are called **predicate methods**.
- As an example, the following method returns `true` if the first argument is divisible by the second, and `false` otherwise:

```
private boolean isDivisibleBy(int x, int y) {
   return x % y == 0;
}
```

- Once you have defined a predicate method, you can use it just like any other Boolean value. For example, you can print the integers between 1 and 100 that are divisible by 7 as follows:

```
for (int i = 1; i <= 100; i++) {
   if (isDivisibleBy(i, 7)) {
      println(i);
   }
}
```

Using Predicate Methods Effectively

- New programmers often seem uncomfortable with Boolean values and end up writing ungainly code. For example, a beginner might write **`isDivisibleBy`** like this:

```
private boolean isDivisibleBy(int x, int y) {
   if (x % y == 0) {
      return true;
   } else {
      return false;
   }
}
```

While this code is not technically incorrect, it is inelegant enough to deserve the bug symbol.

- A similar problem occurs when novices explicitly check to see if a predicate method returns **`true`**. You should be careful to avoid such redundant tests in your own programs.

Exercise: Testing Powers of Two

- Write a predicate method called **`isPowerOfTwo`** that takes an integer **`n`** and returns **`true`** if **`n`** is a power of two, and **`false`** otherwise. The powers of 2 are 1, 2, 4, 8, 16, 32, and so forth; numbers that are less than or equal to zero cannot be powers of two.

```
private boolean isPowerOfTwo(int n) {
   if (n < 1) return false;
   while (n > 1) {
      if (n % 2 == 1) return false;
      n /= 2;
   }
   return true;
}
```

Mechanics of the Method-Calling Process

When you invoke a method, the following actions occur:

1. Java evaluates the argument expressions in the context of the calling method.
2. Java then copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a **stack frame**. This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on.
3. Java then evaluates the statements in the method body, using the new stack frame to look up the values of local variables.
4. When Java encounters a **`return`** statement, it computes the return value and substitutes that value in place of the call.
5. Java then discards the stack frame for the called method and returns to the caller, continuing from where it left off.

The Combinations Function

- To illustrate method calls, the text uses a function $C(n, k)$ that computes the **combinations** function, which is the number of ways one can select k elements from a set of n objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:

How many ways are there to select two coins?

penny + nickel
penny + dime
penny + quarter
penny + dollar

nickel + dime
nickel + quarter
nickel + dollar

dime + quarter
dime + dollar

quarter + dollar

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

- Given that you already have a **`factorial`** method, is easy to turn this formula directly into a Java method, as follows:

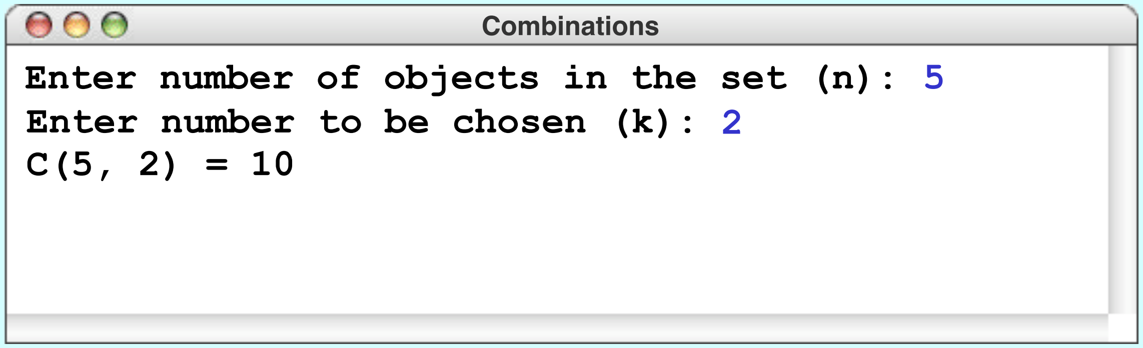
```
private int combinations(int x, int y) {
   return factorial(n) / (factorial(k) * factorial(n - k));
}
```

- The next slide simulates the operation of **`combinations`** and **`factorial`** in the context of a simple **`run`** method.

The Combinations Program

```
public void run() {
   int n = readInt("Enter number of objects in the set (n): ");
   int k = readInt("Enter number to be chosen (k): ");
   println("C(" + n + ", " + k + ") = " + combinations(n, k) );
}
```

n	k
5	2

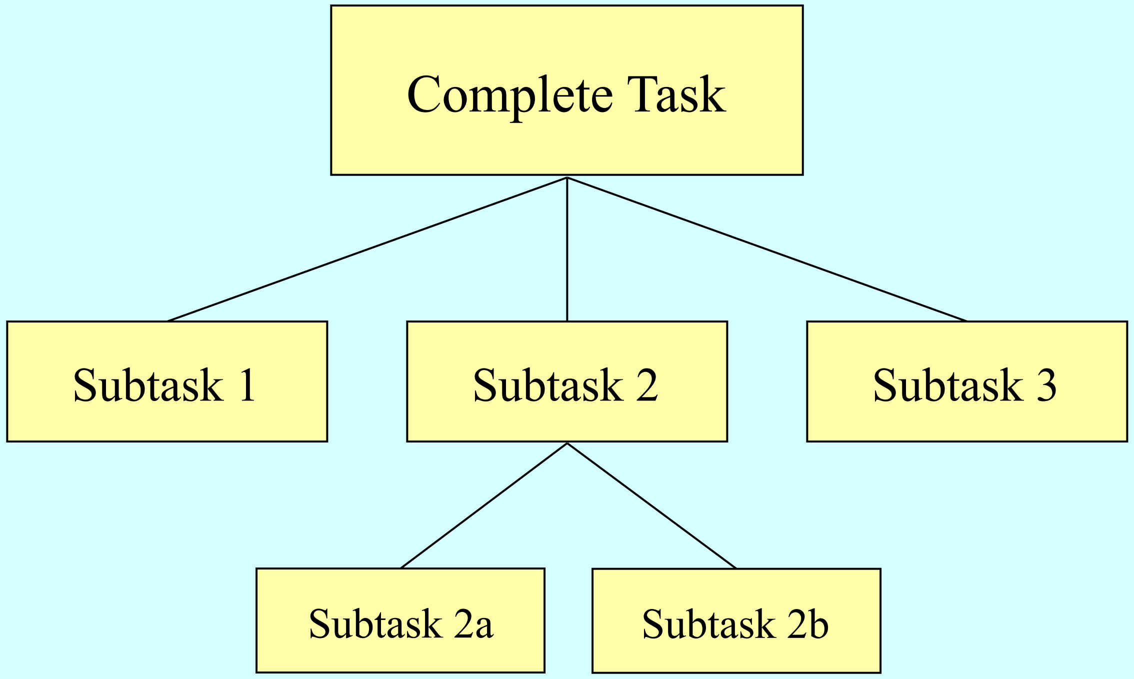


```
Enter number of objects in the set (n): 5
Enter number to be chosen (k): 2
C(5, 2) = 10
```

skip simulation

Decomposition

One of the most important advantages of methods is that they make it possible to break a large task down into successively simpler pieces. This process is called **decomposition**.



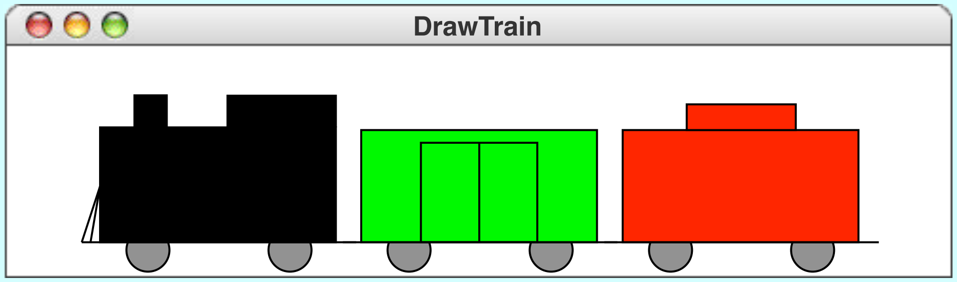
Once you have completed the decomposition, you can then write a method to implement each subtask.

Choosing a Decomposition Strategy

- One of the most subtle aspects of programming is the process of deciding how to decompose large tasks into smaller ones.
- In most cases, the best decomposition strategy for a program follows the structure of the real-world problem that program is intended to solve. If the problem seems to have natural subdivisions, those subdivisions usually provide a useful basis for designing the program decomposition.
- Each subtask in the decomposition should perform a function that is easy to name and describe.
- One of the primary goals of decomposition is to simplify the programming process. A good decomposition strategy must therefore limit the spread of complexity. As a general rule, each level in the decomposition should take responsibility for certain details and avoid having those details percolate up to higher levels.

Drawing a Train

- As its primary illustration of decomposition, the text uses the problem of writing a **`GraphicsProgram`** to draw a train:



- Although it would be possible to write a single **`run`** method to draw the necessary graphical objects, such a program would be very difficult to read. Fortunately, the problem has a natural decomposition, at least at the first level:

```
public void run() {
      Draw the engine.
      Draw the boxcar.
      Draw the caboose.
}
```

Using Pseudocode

- Although the **`run`** method

```
public void run() {
      Draw the engine.
      Draw the boxcar.
      Draw the caboose.
}
```

  suggests the decomposition for the **`DrawTrain`** program, it is not yet legal Java code, but instead a combination of English and Java that sketches the emerging outline of the solution. Such informal descriptions are called **pseudocode**.

- It is important to remember that you don't need to implement each of the steps before you can turn this pseudocode into legal Java. Each of the English lines will simply be a method call. All you need to do at this point is give each method a name and decide what arguments those methods need.

Arguments *vs.* Named Constants

- In graphical programs like the **`DrawTrain`** example, there are two primary strategies for providing the individual methods with the information they need to draw the right picture, such as the sizes and locations of the individual objects:
  - You can use named constants to define the parameters of the picture.
  - You can pass this information as arguments to each method.
- Each of these strategies has advantages and disadvantages. Using named constants is easy but relatively inflexible. If you define constants to specify the location of the boxcar, you can only draw a boxcar at that location. Using arguments is more cumbersome but makes it easier to change such values.
- What you want to do is find an appropriate tradeoff between the two approaches. The text recommends these guidelines:
  - Use arguments when callers will want to supply different values.
  - Use named constants when callers will be satisfied with a single value.

Parameters for Drawing Train Cars

- The **`DrawTrain`** program in the text makes the following assumptions:
  - The caller will always want to supply the location of each car.
  - All train cars are the same size and have the same basic structure.
  - Engines are always black.
  - Boxcars come in many colors, which means the caller must supply it.
  - Cabooses are always red.
- These assumptions imply that the headers for **`drawEngine`**, **`drawBoxcar`**, and **`drawCaboose`** will look like this:

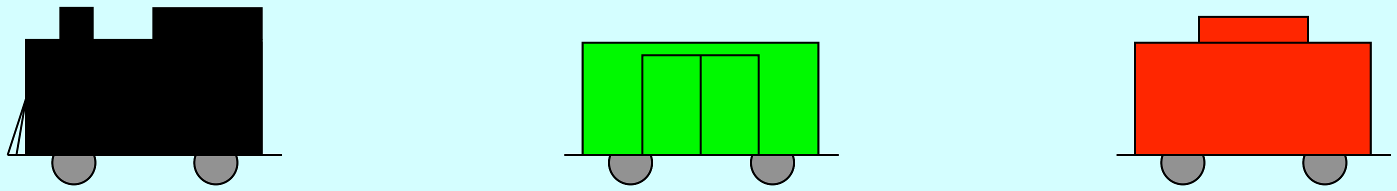
```
private void drawEngine(double x, double y)

private void drawBoxcar(double x, double y, Color color)

private void drawCaboose(double x, double y)
```

Looking for Common Features

- Another useful strategy in choosing a decomposition is to look for features that are shared among several different parts of a program. Such common features can be implemented by a single method.
- In the **`DrawTrain`** program, every train car has a common structure that consists of the frame for the car, the wheels on which it runs, and a connector to link it to its neighbor.
  - The engine is black and adds a smokestack, cab, and cowcatcher.
  - The boxcar is colored as specified by the caller and adds doors.
  - The caboose is red and adds a cupola.



- You can use a single **`drawCarFrame`** method to draw the common parts of each car, as described in the text.

Algorithmic Methods

- Methods are important in programming because they provide a structure in which to express algorithms. Algorithms are abstract expressions of a solution strategy. Implementing an algorithm as a method makes that abstract strategy concrete.
- Algorithms for solving a particular problem can vary widely in their efficiency. It makes sense to think carefully when you are choosing an algorithm because making a bad choice can be extremely costly.
- Section 5.5 in the text looks at two algorithms for computing the **greatest common divisor** of the integers *x* and *y,* which is defined to be the largest integer that divides evenly into both.

Brute-Force Approaches

- One strategy for computing the greatest common divisor is to count backwards from the smaller value until you find one that divides evenly into both. The code looks like this:

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

- This algorithm must terminate for positive values of **`x`** and **`y`** because the value of **`guess`** will eventually reach 1. At that point, **`guess`** must be the greatest common divisor because the **`while`** loop will have already tested all larger ones.
- Trying every possibility is called a **brute-force strategy**.

Euclid's Algorithm

- If you use the brute-force approach to compute the greatest common divisor of 1000005 and 1000000, the program will take almost a million steps to tell you the answer is 5.
- You can get the answer much more quickly if you use a better algorithm. The mathematician Euclid of Alexandria described a more efficient algorithm 23 centuries ago, which looks like this:

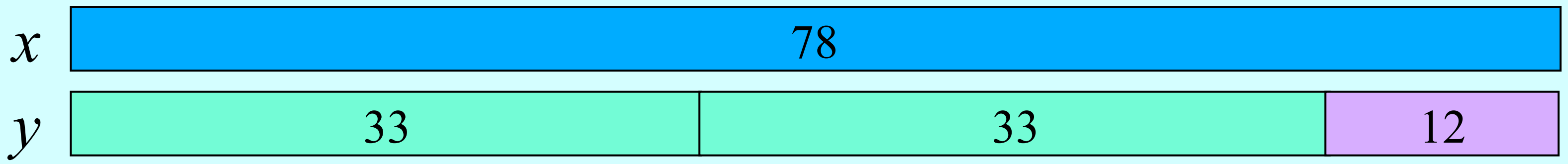
```
public int gcd(int x, int y) {
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

How Euclid's Algorithm Works

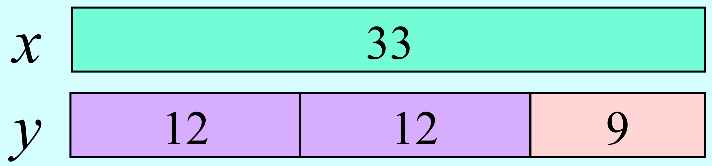
- If you use Euclid's algorithm on 1000005 and 1000000, you get the correct answer in just two steps, which is much better than the million steps required by brute force.
- Euclid's great insight was that the greatest common divisor of x and y must also be the greatest common divisor of y and the remainder of x divided by y. He was, moreover, able to prove this proposition in Book VII of his *Elements*.
- It is easy to see how Euclid's algorithm works if you think about the problem geometrically, as Euclid did. The next slide works through the steps in the calculation when x is 78 and y is 33.

An Illustration of Euclid's Algorithm

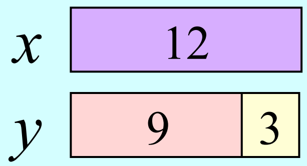
Step 1: Compute the remainder of 78 divided by 33:



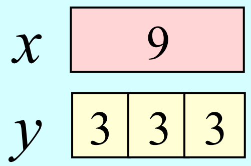
Step 2: Compute the remainder of 33 divided by 12:



Step 3: Compute the remainder of 12 divided by 9:



Step 4: Compute the remainder of 9 divided by 3:



Because there is no remainder, the answer is 3:

The End