CHAPTER 9
METHODS
A method is a named block of code that can be executed easily using the method's name. This is one approach for decomposing complex programming tasks into smaller, more manageable subtasks. In addition, modularity can help to reduce or eliminate redundancies in your code. Finally, once a method is written and debugged, it can generally be reused more easily.
FLOW OF EXECUTION

Methods are also a way to control the flow of execution in an app (i.e., like the branching and looping structures discussed in Chapter 8). When a method is used (we say “called” or “invoked”), the flow of execution jumps to the method. It’s kind of like a “detour” you take when driving.

The flowchart visually depicts the flow of execution (in purple) as it jumps to the invoked method and then returns when complete. The method invocation in the center of the flowchart detours the flow of execution to the method at the right and then resumes from where it was called.
METHOD SYNTAX

This first line, up until the opening brace, is called the method signature, and defines four important details of the method:

The optional access modifier, is either public, private, or protected (discussed later). If left blank, then default access is used.

The required return type is either void (nothing returned by the method) or a valid Java data type or class (a value of which is returned at the completion of the method).

The required method name follows the return type and precedes the parentheses and must be a valid Java identifier.

The optional parameter list declares what data must be passed to the method (may be left empty).
Syntax: Java Methods

```
[access modifier] [return type] methodName([parameter list])
{
    // method body
}
```

The access modifier is first.
The return type is second.
The parameter list is enclosed in parentheses.
The body of the method is enclosed in curly braces {}.
The method name, as you can see, directly precedes the parentheses.
YOU’VE ALREADY BEEN USING METHODS

Surprise! You’ve actually been writing methods since the very first program you created. Let’s cast our minds back, back, back in time to “Hello.java”, which contains just two methods: `main()` and `Hello()`:

```java
// Hello World
public static void main(String[] args) {
    Hello app = new Hello();
}

Hello() {
    System.out.println("Hello World!");
}
```

The `main()` method (one is required in all Java apps) is `public` and `void` (both requirements). This method is run once, when the app is loaded. Its job is to set the app up. In this case, it is calling the `Hello()` method.
CONSTRUCTOR METHOD

Hello() is a special method called a constructor, which has a somewhat different format than other types of methods (specifically, it has no “return type”, which we’ll discuss momentarily, and also starts with a capital letter).

Constructors are optional in many cases, but can be useful. You’ll use application constructors to set up the UI, placing widgets where you want them and attaching them to listeners.

If present, the constructor runs once, when a new object is instantiated from this class.

We’ll cover constructors in more detail in Chapter 11.
FOUR TYPES OF METHODS

There are two major considerations when designing a method: will the method accept parameters and will the method return any data? Since both of these considerations have two possible options, we end up with a 2x2 grid that defines the four types of methods:

<table>
<thead>
<tr>
<th>Does not accept parameters</th>
<th>Does not return a result (void)</th>
<th>Returns a result (non void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not accept parameters</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Accepts parameters</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Each of these types has a distinct form and use. Syntactically, they look fairly similar.
# FOUR EXAMPLES

Syntactically, they look fairly similar:

<table>
<thead>
<tr>
<th>Does not accept parameters</th>
<th>Does not return a result (void)</th>
<th>Returns a result (non void)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void method1() {</code></td>
<td><code>System.out.print(&quot;Hi&quot;);</code></td>
<td><code>String method2() {</code></td>
</tr>
<tr>
<td><code>    }</code></td>
<td></td>
<td><code>    return &quot;there&quot;;</code></td>
</tr>
<tr>
<td><code>void method3(String name) {</code></td>
<td><code>System.out.print( name );</code></td>
<td>`}</td>
</tr>
<tr>
<td><code>    }</code></td>
<td></td>
<td><code>String method4(String</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>    param1, String param2) {</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>    return param1 + param2;</code></td>
</tr>
</tbody>
</table>

These don’t do anything particularly useful. If you really wanted to, you can use them to display a simple greeting (see [SimpleOneOfEach in the chapter resources](#)).
CALLING YOUR METHOD

You call a method by invoking its name and passing any parameter data. If there are no parameters, you still must type the empty parentheses.

Void and non-void methods are called differently: non-void methods are inserted into statements as part of an expression while void methods are complete statements already.

This may sound more complex than it really is.

The way you call any method is defined right in the method signature. If the method requires parameters, they will be listed in the signature. If a value is returned, its data type is also declared in the signature. If you want to capture the returned value, you’ll need to put it in a variable of the correct data type.
See how the **void** methods 1 and 3 are simply called? They just run and do whatever task they are coded to do.

Notice how the **non-void methods** 2 and 4 are inserted into a valid Java expression? These methods **return data back** that needs to be captured by the calling expression.

See that methods 1 and 2 take no parameters, so the parentheses are present, but empty.

Finally, see that methods 3 and 4 require parameters, so these method calls includes the required values.
ACTIVITY 1

Enter the previous 4 methods and make an app that says “hi” to you at the console.

Your app doesn’t need a UI, so there’s no need for any widgets or event-handling.
EXAMPLE: DOUBLOON METHODS

This simple app generates random account balances for 6 pirate customers. Based on these balances, each pirate customer is assigned a membership level, with “gold” being the top class. Every pirate covets this designation.

To generate this awesome report, a variety of methods are used. Let’s take a look at examples of each.
PARAMETERS

Parameters are defined in the method signature after the method name, inside parentheses, by listing the data types and parameter names that are needed (separating multiple parameters with commas).

Here's an example of a method that accepts parameters:

```java
void addSummaryData(int[] customerDoubloons) { /* method body */ }
```

See that `addSummaryData()` accepts one parameter: an array of integers to be called `customerDoubloons` inside the method. Whatever data is passed to this method will be used internally by referencing the variable name(s) declared (in this case, `customerDoubloons`).
On the other hand, methods may be written which accept **no parameters**, although they **still must have the parentheses after the method name** (they will just be empty).

**For example:** Here’s an example of a method that accepts no parameters:

```java
int getBalance( ) { /* method body */ }
```

This method **looks up the current balance** (by generating a random number) and **returns it as an integer**. It doesn’t take any parameter data, so the parentheses are empty.
NOTES

When using a method, you must provide exactly the parameters that it is expecting, or no parameters at all, based on how you defined the method.

Moreover, you must provide the parameter data in the order and data type that the method is expecting (again, based on how you defined the method).

If you provide invalid parameters, you will see a compiler error message that says so.
VOID METHODS (NOTHING RETURNED)

The return type “void” means that the method does not send any sort of value back (i.e., return a result of some sort) to the expression in which it was invoked.

As it is in the DoubloonMethods app for the `drawLine()` method:

```
void drawLine(int style) { /* method body */ }
```

When calling a void method, the method call itself is a complete Java statement. So, for example, to draw a plain line in the report text area, it is sufficient to do this:

```
drawLine(1);
```
RETURNING RESULTS (NON VOID)

For methods that *do return a result* of some kind, the *data type* (e.g., int, float, double, String) returned must be specified in place of the void declaration.

For example, the `getBalance()` method discussed above returns an integer value:

```java
int getBalance() {
    // ... }
```

That is, this method calculates an integer result and then sends that result back to the calling line of code, where it takes the place of the method call in the expression.

It might be called like this:

```java
int bal = getBalance();
```
And, if the result that the method found was 30, then that value is plugged into the expression, where it becomes:

```java
int bal = 30;
```

In this case, the result of the method call is assigned to the integer variable bal. Methods that return a value must be non-void and must use the `return` statement to send the value back. For example, in the `getBalance()` method, the simulated “account balance” is simply the result of a bit of random math:

```java
int getBalance() {
    return (int) (40*Math.random());
}
```
RETURNING RESULTS III

Note that, as you are typing this kind of method in your IDE, as soon as you make a method’s return type non-void (i.e., a data type), the IDE will show an error that essentially says “you haven’t returned anything”.

Of course, you aren’t done typing yet! As soon as you enter the return statement followed by a literal or variable of the correct data type, the error will go away.
ACCESS MODIFIERS

You may have noticed the use of the terms “public” and “private” for some methods, variables, or classes. These are called access modifiers and play an important role in object oriented programming.

Access modifiers determine the visibility of program entities outside of the class or object they are in.

Public entities (classes, objects, methods, or variables) are accessible to objects instantiated from other classes, even when the other classes are in a different package.

Private entities, on the other hand, can only be used by objects internally. Private is the most restrictive access modifier.
If you need to access these entities from another class, you’ll need to make them public.

For example, if you had 2 fictional classes A and B, each with an integer variable “val”, it might look like the figure to the right.

The difference between A and B is that val is public in A and private in B. So, your app can create objects of each, but can’t access the val variable in the object b (because it is private). These modifiers enable or disable access to methods in the same manner.
THAT WHICH MUST BE DONE IN PUBLIC

There are several program entities that must be public. Such entities are typically used directly by the Java VM or would be useless otherwise.

Here’s the short list:

* “Top-level” classes (a class declared outside of any containing class), including the one that contains the main( ) method (i.e., your app).
* The main( ) method itself.
* All event handlers.
* The get( ) and set( ) methods (also called the “getters and setters”).
METHOD OVERLOADING

Your apps may contain methods with the same name, as long as the methods accept different parameters (different data types, different number of parameters, or different ordering of the data types).

This is called method overloading. Overloading is useful because it can provide the developer with more flexibility with regard to how you call a method.

Why would you want to do this? Well, it gives you flexibility when calling a method to pass a variety of parameters. With just one method, you are limited to just one set of required parameters. Overloading allows you to use a method in more than one way.
EXAMPLE

In DoubloonMethods, consider the `drawLine()` method. If I want to be able to draw different styles of lines, I could create different methods. Or, I could use overloading to allow a non-default style to be specified when the method is called.

No problem. You just create two versions of the method to handle both situations.

The methods are distinguished by the parameter data passed. If you pass no data, the first (default) method is called. If you pass an integer parameter, then the second method is called:

```
// method overloading:
drawLine( ); // draws a default line
drawLine(2); // draws a line in style #2
```
TWO METHODS

The default version (**no parameter**) draws a line using hyphens:

```java
// method type 1 (does not return a value and does not take parameters)
void drawLine() {
    ta.append("-----------------------------------------------
    ");
}
```

The styled version requires an **integer parameter** to select the style (note that style 1 is a default line and calls the default method to do the work):

```java
// method type 3 (does not return a value but does take parameters)
void drawLine(int style) {
    if (style == 1) {
        drawLine();
    } else if (style == 2) {
        ta.append("===============================================
        ");
    } else if (style == 3) {
        ta.append("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ");
    }
}
```
NESTING METHODS?

Note that, in Java you are not allowed to nest methods. You can certainly call (or invoke) a method from within a method, but you may not declare a new method inside of an existing one.

That's not to say it wouldn't make at least some sense to be able to do this. You could make a case that having a method that was local in scope to another method could sometimes be helpful (and is supported in other languages).

However, in Java all methods must be declared at the same level (i.e., inside of a class). You can, however, declare a class within a class, as well as a class within a method, which confounds the issue of nesting somewhat.
REFACTORING EXISTING CODE INTO A METHOD

Sometimes, perhaps even more often, you will write a bit of code and then realize that it would make more sense to have it as a method. Refactoring is a term used to describe the process of making structural changes to your code to make it clearer and easier to work on without changing the features (en.wikipedia.org/wiki/Code_refactoring).

Both of the IDE’s we are using have several refactoring options to perform such tasks for you.
HOW TO REFACTOR A METHOD

In Eclipse, just highlight the code you want to use, right-click and select “Refactor… Extract Method”. You will then be presented with a dialog box to configure some settings for the new method (including what you want to name it).

In Netbeans, you also select “Refactor”, then choose “Introduce > Method”.

In both IDEs, the selected code is replaced by a call to the (new) method, so the app will compile and execute as well as it did before the refactoring.
Unified Modeling Language or “UML” (en.wikipedia.org/wiki/Unified_Modeling_Language) is a standard way to visually depict the design and structure of computer programs, including classes and their constituent elements.

UML diagrams provide a way to organize and communicate the design of an app, which is often helpful when working with other developers, as well as designing and describing the system to users.

There are four different types of UML diagrams: class diagrams (our focus here), use-case diagrams, state diagrams, and sequence diagrams.
**UML CLASS DIAGRAM**

The class diagram should describe the classes used within your app and how they are related from an object oriented standpoint.

**Classes are described in boxes**, while relationships are described with arrows.

Within each class box, the **attributes and methods** are detailed.

The basic style can look something like this (there is a lot of variation):

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of class attributes (Instance variables)</td>
</tr>
<tr>
<td>List of class operations (Methods)</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
UML CLASS DETAILS

For each attribute, the diagram will display the accessibility (as a symbol), the attribute name, and the data type.

For each method, the UML displays the accessibility, the name, the parameter data types, and the return type (i.e., void or a valid data type).

Accessibility symbols preceding each element in the UML class box correspond to the Java access modifier used:

+ for public
- for private
# for protected (Chapter 12)
~ for package (or “default”)

A hidden attribute is a happy attribute
RELATIONSHIPS

Arrows are used to indicate the relationship between classes. Arrows can be solid or dashed, and the arrowheads can take a variety of forms, each describing a different type of relationship.

We will focus on two types that are covered in this book:

1. A **solid line** and solid/closed arrowhead (inheritance).
2. A **dashed line** with closed/empty arrowhead (an implements relationship).

The **DoubloonMethods** app has a number of attributes and methods to keep track of, meaning that a UML diagram could be useful...
If you look through the `DoubloonMethods` code, you’ll see 6 instance variables and objects, including the UI widgets and two arrays to hold the pirate customer data. In addition, you should also see 11 methods which perform a variety of tasks.

Finally, from the class signature you can see this class is a type of JFrame (i.e., it extends `JFrame`) and uses (implements) the interface class `ActionListener`. 
Looking more closely at the methods (i.e., the topic of this chapter), you’ll see examples of each of the four types:

<table>
<thead>
<tr>
<th>Does not accept parameters</th>
<th>Does not return result (void)</th>
<th>Returns result (non-void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void drawLine( )</td>
<td>string eol( )</td>
<td>int getBalance( )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accepts parameters</th>
<th>Does not return result (void)</th>
<th>Returns result (non-void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void drawLine( int )</td>
<td></td>
<td>string getMembershipLevel( int )</td>
</tr>
<tr>
<td>void main( String[] )</td>
<td></td>
<td>double badMoneyRound( double )</td>
</tr>
<tr>
<td>void addSummaryData( int[] )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void actionPerformed(ActionEvent)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
It is sometimes difficult to decide whether a method you are writing should accept parameters and whether it should be void or not.

There are two simple rules that may help:

1. There’s never any need to pass an in-scope instance variable as a parameter to a method, since these are already available throughout your class.

2. If the purpose of a method is to change the value of an in-scope instance variable or object, there’s generally no need to return a result since, again, this can simply be updated directly.
FLOW OF EXECUTION

It is a bit like how a waitress in our terrible pizza restaurant moves back and forth from a table to the kitchen and the bar. Kind of like Diner Dash. In fact, if we say her name is “Flo”, then she really can be thought of as the “Flo of execution”! It’s a decent analogy. Try to stay with me:

Flo takes instructions from her tables and the kitchen. Flo goes to a table to figure out what they want to order. Tables and stations can be thought of as methods. Flo takes menus to the table and returns to the kitchen with their order. The order is the “return” from the table. Anything Flo takes from a table can be thought of as a return. Flo picks up the food at the kitchen and delivers it to a table. The food is like a parameter Flo takes from the kitchen and leaves at the table. The activity at the table (talking, eating, etc.) can be thought of as the method executing. When Flo takes something with her (menus, food, drinks, etc.), these can be thought of as parameters being passed. Sometimes Flo doesn’t take anything from the table or bring anything to the table, other than her cranky pirate disposition.
Methods are often used to make complex work easier to perform as needed, avoiding redundant and difficult to update code. Methods can be designed to accept parameters to use inside the body of the method and may also return a result when the method completes.

Methods can be given the same access modifiers that we give to classes and other class members. Certain methods, such as `main()` and the event-handlers, must be public. Other methods may be given `private` or `default` (package) access as appropriate.

Methods can be visually described as part of a UML class diagram: Methods are listed as the “operations” in the 3rd cell of the table.