Programming methodically

A method is a **named block of code** that can be executed easily by using the method's name (e.g., “A” or “B”, and so on, in the Figure “Modularization”). Identifying code that you expect to reuse is a technique to achieve **modularity** in your programs. This is one approach for **decomposing** complex programming tasks into smaller, more manageable subtasks. Modularity allows the programmer to break the functionality of a program into smaller pieces that can be coded and debugged separately (at least to some extent). In addition, modularity can reduce or...
eliminate redundancies in your code. Finally, once a method is written and debugged, it can generally be used over and over again reliably. Decomposition is an excellent practice that you should strive to master.

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.

![Flow of Execution for a Method](image)

The flowchart ("Flow of Execution for a Method") visually depicts the flow of execution (in purple) as it jumps to the invoked method and then returns when complete.

As an example, methods were used *extensively* in the PizzaTime advanced app (Chapter 8). The code needed to draw the face of the clock (i.e., the pizza and toppings) was placed in the method `drawPizzaClock()`. While that isn’t a
really surprising name, the “expectedness” of it is actually an important goal when naming your methods: It should be perfectly clear what the method will do from simply reading the signature. In any case, since the clock face has to be redrawn constantly, placing the code in an easily reusable method makes sense.

With respect to drawPizzaClock(), this method is called in the paintComponent() method, which is contained within a custom JPanel (PizzaPanel). After the code in the body of drawPizzaClock() is executed, the flow jumps back to the calling line of code and executes whatever comes next (see “as easy as 1-2-3”).

**Method syntax**

Surprise! You’ve already been writing and using methods. In fact, you have 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 two methods: main() and Hello().

They looked something like this:

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

Hello() {
    System.out.println("Hello World!");
}
```
The method name, as you can see, directly precedes the parentheses (which may contain additional information, as is the case with main(), or be empty, as is the case with Hello(). The body of the method (the code that is executed when the method is called) is enclosed in curly braces {}.

The general format for a method is:

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

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

1. The optional access modifier, which is optional in some cases, is either public, private, or protected (we’ll discuss the meaning of these later).

2. 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).

3. The required method name follows the return type and precedes the parentheses and must be a valid Java identifier (i.e., following the same naming rules as classes and variables).

4. The optional parameter list declares what data must be passed to the method. If the method does not need to be sent any parameter data, then the parentheses may be left empty (but still must be present).

Looking at the “Hello.java” methods, the first method is called the “main method” for obvious reasons and one is required in all Java apps. You might also notice that it is public (which is required) and void (also a requirement). This is the method that is run first when the app is loaded. Its job is to set the app up and may include tasks like sizing the app’s window, setting it’s title, and running a method that sets up the GUI. In the case of “Hello.java”, we are initializing the app and calling the Hello() method (all in one line of code).

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). We’ll cover constructors in more detail in Chapter 11.
We have also made use of many methods that are built-into Java (such as the String manipulations we covered in Chapter 7, including `toUpperCase()` and `indexOf()`). In addition to these built-in methods, and the methods that we are required to write (like `main()` and `actionPerformed()`), it is very helpful to be able to write your own.

### Types of methods

Broadly, there are two major considerations when designing a method: will the method accept parameters (and if so, how many and of what types) 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 (see “Four types of method signatures”).

<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>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Four types of method signatures**

Each of these types has a distinct form and use. Let’s look at a new app (DoubloonMethods.java) to find some practical examples of each of these types.

**As an Example: DoubloonMethods**

DoubloonMethods is a simple app that simulates a simple app that looks up current account balances for our pirate customers. Based on these current 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.
Accepting parameters (or not)

Methods may accept parameters (or "arguments") that pass data with the invocation to the method for use by the method (table cells 3 & 4). 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 */ }
```

Here we see that the method `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 names declared (in this case, `customerDoubloons`).

On the other hand, methods may be written which accept no parameters (table cells 1 & 2), although they still **must have the parentheses** after the method name (they will just be empty). For example:

```java
int getBalance() { /* method body */ }
```
In this method, a balance (in doubloons) for a customer is created and returned (as an int integer) to the calling expression.

**Syntax note: Bad parameters**

It is important to note, however, that 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 that says so. For example, if you try to pass an integer to the `getClosestMinutes()` method, you would see error shown.

In this case, trying to pass the double value 22.99 is causing the error: `getMembershipLevel()` expects an int integer parameter, not a double. If you want to be able to send a double parameter to this method, you have to either modify the method or provide an additional version that takes the parameter (this is called “method overloading” and is discussed below).

**Returning results (or not)**

A method can either "return" a calculated result (table cells 2 & 4) or not (1 & 3); the programmer makes this decision and specifies it in the method signature. The return type “void” means that the method does not send any sort of value back ("returned") to the section of code that called (invoked) the method.
This is the case, for example, with the `main()` method that we've used in all of our apps:

```java
public static void main(String[] args) { /* method body */ }
```

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

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

When calling a void method, the method call itself is a complete Java statement and does not need to be plugged into a line of code (i.e., to complete an expression). There are no results returned that need to be consumed by the calling line of code. So, for example, to draw a plain line in the report text area, it is sufficient to do this:

```java
drawLine(1);
```

Of course, void methods can be much more complex. In addition, void methods may perform significant calculations for internal purposes (the results of which are not returned). In addition, void methods may make changes to instance variables (which are available throughout the app).

**Non-void methods**

Otherwise, 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 a realistic integer result and then sends the value it found 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());
}
```

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., to a data type), the IDE will present you with 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: public & private**

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 between classes. Public means that the entity (class, method, or variable) is accessible to running instances of other classes (i.e., classes that did not declare the entity), even when the other classes are in a different package. Public is the least restrictive access modifier. Private entities, on the other hand, can only be accessed by an instance of the class that declares them. Private is the most restrictive access modifier.

For example, if you have a `Pizza` class that described individual pizza objects, and you had an `Order` class that described the pizzas being ordered, you might want some methods in the pizza objects to be accessible to the current order. It might make sense to have a `getDescription()` method, for example, that returned a string representation of the pizza such as:

"Large, hand-tossed, pepperoni pizza"
If you wanted to be able to pull such pizza descriptions while you were processing the order, perhaps to append to an onscreen list of some sort, then you would want the `getDescription()` method to be public. Only classes, methods, and instance entities may be declared to be public. Variables declared inside of methods are scope-restricted to the method itself (i.e., are not visible to code outside of the method body), as such, declaring them to be public wouldn’t make any sense (and can’t be done).

However, there may be other entities in pizza objects that you don’t care about outside of them (but want instance access to), such as file location of an image used to depict the object or the method used to clean and format the list of ingredients. We may very well care that each pizza object has an image and an ingredient list; however, the mechanics that the object uses to create these may not be of interest to users of pizza objects. As such, those entities may be declared to be private.

Finally, there are examples where we haven’t indicated an access modifier at all. This creates “default” entities which are similar to public, except that accessing objects must be within the same package as the default resource. This is also referred to as “package” access.

That which must be done in public

There are several program entities that must be public. Using anything else will lead to a compiler error and/or an app that won’t work correctly. Each of these are required to be public because they are directly used by the Java VM. Here’s the short list:

1. Any “top-level” class (a class declared outside of any containing class), including the one that contains the `main()` method (i.e., your app). We’ll discuss classes more in Chapter 11.
2. The `main()` method itself.
3. All event handlers (e.g., `actionPerformed()`, `adjustmentValueChanged()`, etc.).
4. The `get()` and `set()` methods in a class (also called the “getters and setters”).

Other entities can be private or “default” at this point. The access modifiers for anything not on this list won’t affect how your app runs at all, until we get to apps that have multiple classes (such as `Pizza` and `Order`), at which point we will revisit this issue.
Method overloading

Okay, this is cool. Our 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 they call a method.

For example, maybe I do want to be able to pass `drawLine()` an integer parameter as well as leaving it empty. No problem. We’ll just create two versions of the method to handle both situations. Details on writing those two methods are covered below.

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

Although it may seem confusing to have more than one method with the same name, the difference in the way the method is called makes it clear which is being executed. That is, if I pass no parameter, I am calling the original method, while if I pass any integer value, I am calling the overloaded method.

Nesting methods? Sorry, no

Note that, in Java you are not allowed to nest methods. You can certainly call (or invoke) a method from within a method you are writing, as we have frequently done, 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, it’s just that Java does not allow it. You could make a case that having a method that was local in scope to another method could sometimes be helpful; 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.

As an example, assume you had some data that you needed to work with. It is in an array called “data” and is a simple array of integers `{44, 29, 3, 22, 61, 28, 11}`.
Perhaps you’ve read this in from a file and the exact numbers in the array may change when the program runs. In addition, it is possible that you will receive additional arrays of the same type. You need to total (sum) the integers in the array, which can be done nicely with a method. However, if you are working inside of an existing method (perhaps the constructor or an event-handler), trying to write the method there will not work. See MethodNesting.java below:

```java
public class MethodNesting {
    public static void main(String[] args) {
        MethodNesting app = new MethodNesting();
    }

    int[] data = {44, 29, 3, 22, 61, 28, 11};

    MethodNesting() {
        /* doesn't work here */
        int getSum(int[] d) {
            int sum = 0;
            for(int i : d) {
                sum+=i;
            }
            return sum;
        }
        int sum = getSum(data);
        System.out.println("Sum of data = " + sum);
    }

    // instead, put it outside of the method:
    int getSum(int[] d) {
        int sum = 0;
        for(int i : d) {
            sum+=i;
        }
        return sum;
    }
}
```
Instead, you must create the method outside of any other method for it to compile. It does not matter where this method is placed in relationship to the other methods (i.e., it can be below a method it is called in as it the case in this example, or moved above it).

Refactoring existing code into 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 app’s features. Both of the IDE’s we are using have several refactoring options to perform such tasks for you.

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…” which gives a similar (but somewhat simpler) dialog to Eclipse. 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 (UML)

UML ("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 nice 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 diagrams which are UML standards: The class diagram (our focus here), the use-case diagram, the state diagram, and the sequence diagram.

Simple class diagrams using UML

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. For each attribute, the diagram will display the accessibility (as a symbol), the attribute name, and the data type. For methods, the UML displays the accessibility, the name, the parameter data types, and the return type (i.e., void or a valid data type).

Additional symbols are used to describe important attributes. Accessibility symbols preceding each element in the UML class table corresponds to the Java access modifier used:

+ for public
- for private
# for protected (we’ll cover this modifier in Chapter 12)
~ for package (or “default”)

In addition, arrows are used to indicate the relationship between classes. Arrows can be solid or dashed, and the arrow heads 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: the solid line and solid/closed arrowhead (meaning an inheritance) and a dashed line with closed/empty arrowhead.
(meaning 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.

DoubloonMethods UML
As an example, what do you think the class diagram for the DoubloonMethods app (DoubloonMethods.java) would look like? If you look through the 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 a total of 11 methods which perform a variety of tasks. Finally, from the class declaration (line 10) you can see that this class is a type of JFrame (i.e., it extends JFrame to inherit its features) and uses (implements) the interface class ActionListener to give the app the ability to respond to the button press.

Using UML to describe this class looks something like the diagram above (DoubloonMethods class diagram). The diagram has 3 boxes, one for the app class itself and two for the superclass JFrame and the implemented interface class ActionListener. The 2 arrows point up from the app class to
the class resources it is based on. The arrows take the form they do to visually communicate the difference between an inheritance relationship and the implemented interface relationship.

**DoubloonMethods attributes**

Except for the `serialVersionUID`, the attributes in `DoubloonMethods` have *package access* (which means that in the code there is no access modifier used to declare them), which is okay for the apps we’ve been writing so far. However, once we start writing multi-class apps, we’ll want to think carefully about what to make *private*. We’ll use *private* often as we will provide *public* “getters and setters” (individual `get()` and `set()` methods) for any attributes we wish to provide access to. Likewise, the `serialVersionUID` can be *private* (as Eclipse makes it) or not. In general, you want to **limit the scope of a variable or object as much as you can** and you don’t want to provide external access to class entities unless you are certain you must.

**DoubloonMethods methods**

Looking more closely at the methods, you can find 1 constructor method and 10 regular methods. The constructor, which is optional, typically doesn’t have a return type and may be declared public or private (making it irregular). We can place the regular methods into the 2x2 grid used earlier:

<table>
<thead>
<tr>
<th>Does not return result (void)</th>
<th>Returns result (non-void)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Does not accept parameters</strong></td>
<td></td>
</tr>
<tr>
<td><code>void drawLine( )</code></td>
<td><code>String eol( )</code></td>
</tr>
<tr>
<td></td>
<td><code>int getBalance( )</code></td>
</tr>
<tr>
<td><strong>Accepts parameters</strong></td>
<td></td>
</tr>
<tr>
<td><code>void drawLine( int )</code></td>
<td><code>double getAverageBalance( int[] )</code></td>
</tr>
<tr>
<td><code>void main( String[] )</code></td>
<td><code>String getMembershipLevel( int )</code></td>
</tr>
<tr>
<td><code>void addSummaryData( int[] )</code></td>
<td><code>double badMoneyRound( double )</code></td>
</tr>
</tbody>
</table>

**DoubloonMethods methods**

It is often somewhat difficult to decide whether a method should accept parameters and whether it should be *void* or not.
However, there are two simple rules that may help you:

1. There’s never any need to pass an instance variable as a parameter to a method, since all of the instance variables are already available throughout your code.

2. Likewise, if the purpose of a method is to change the value of an instance entity, there’s often no need to return a result since, again, the variable that holds the result is available throughout the code.

The details

So what do these methods do? Well, you can just about figure it out by simply reading the description in the UML diagram and using what you already know about Java, but here’s a bit more detail:

1. The `drawLine()` method is `void` and simply appends a “line” made from hyphens to the report.

2. The `eol()` method simply returns a line break;

3. The `drawLine(int)` method allows the programmer to select a line style (hyphens, equal signs, or tildes).

4. The `main(String[])` method is used to configure the window app is in and call a method to set up the UI.

5. The `addSummaryData(int[])` method appends the average balance to the report. It is passed the array of customer doubloons to give you an example of passing a parameter on to another method (`getAverageBalance(int[])`) although this isn’t strictly necessary since this array is an instance variable. One of the end-of-chapter exercises will ask you to remove this parameter from this method.

6. The `getBalance()` method is used to simulate reading the customer’s current account balance (in doubloons). In reality, it just generates a random integer and returns it.

7. The `getAverageBalance(int[])` method takes an array of integers as input, calculates the average as a `double`, and calls `badMoneyRound(double)` to round the average to be returned.

8. The `badMoneyRound(double)` method uses `Math.round()` to give us a rounded result that often looks like money. This is only a first attempt at rounding money properly and is more **an example of how not to do it.**
9. The `getMembershipLevel(int)` method returns a string result that is appropriate to the number of doubloons your pirate customer has.

10. The `actionPerformed()` event-handler is `void` and handles the “generate report” button. When called, this code creates the text report appended to the text area.

**Method overloading in action**

You might have noticed that the code has two methods with the same name. You may recall the discussion of this about 6 pages prior. This is allowed since they accept different parameters.

The version without any parameters is quite simple:

```java
void drawLine() {
    ta.append("--------------/* etc */--------------");
}
```

As is the version which takes an integer as its parameter:

```java
void drawLine(int style) {
    if (style == 1) {
        drawLine();
    } else if (style == 2) {
        ta.append("=============/* etc */==============" + eol());
    } else if (style == 3) {
        ta.append("~~~~~~~~~~~/* etc. */~~~~~~~~~~~
    }
    }
}
```

We can have both methods present in the same class because their parameter list is distinct. Why not just write one method that does exactly what I want? Well, that wouldn’t give us an example of overloading, for one thing. Secondly, it does leave me with the flexibility to call the method any way I might wish (even if I can’t think of a great reason at the time I am coding it why such flexibility might be needed).
Summary

Methods are often used to make complex calculations easier to perform as needed in your code. Methods can be designed to accept parameters to use inside the body of the method and may also return a calculated 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 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.

Key skills

- Decomposing a programming task into smaller modules (facilitating teamwork and reuse).
- Controlling the flow of execution using methods.
- Writing methods that return various data types or none at all (i.e., void methods).
- Writing methods that are passed zero or more listed parameters.
- The importance of appropriately using `public` and `private` access modifiers.
- Basic UML to describe a class based on its methods and instance variables.
- The use of method overloading to make classes easier to use.
- Using the refactoring tools in your IDE to “extract” a method from highlighted code.
- Using methods to decompose the requirements of the DoubloonMethods app (to generate an old school text report).
Downloads

Here are links to the apps discussed in this chapter:

1. PizzaTime.java
2. DoubloonMethods.java
3. MethodNesting.java

End-of-chapter exercises

1. Create a method that displays your name in the console. This method is void and takes no parameters. Make an app that runs the method in response to a button press.

2. Create a version of the method in #1 that takes the text (String) to be displayed as a parameter. Allow the user to enter the text in a dialog box or text field and display that text in the console.

3. Draw the UML class diagram for #2. You may use any tool you’d like to (Visio on Windows and OmniGraffle on the Mac are both great). There are also several free (e.g., http://www.genmymodel.com/) and “freemium” online tools (e.g., Gliffy and Lucidchart).

4. Create a method to calculate the price of a pizza and return it as a double. The method should take at least 3 parameters: Size, toppings, and crust style. Create an app which makes 3 pizzas and describes them at the console.

5. Create a method to validate a “coupon code” passed to it. The code should be a String and the method should return a double representing the value of the coupon. Pick a few “valid” codes and assign them each a discount value. Return 0.0 if the code is not valid. Create an app which makes 2 pizzas, one with a valid code and one without, and describes them at the console.

6. Create an app that uses the methods you created in #4 and #5. Allow the user to enter values in four text fields for each of the parameters you need (size, toppings, crust style, and coupon code). Display the resulting price.
7. Create a method to search through the pizza ingredient list and return the ordinal location (an integer) of any searched for ingredient (e.g., “gluten is the 11th ingredient”, “molasses was not found”, etc.). Create an app that allows the user to type an ingredient in, then display the result in a means of your choosing.

8. Display a random price inside a JLabel. Create an app with two methods: One to add 10% to the price and the other to subtract 10% from the price. Use your mouse click handling skills to increase the price for a standard (left) click and reduce the price for a right-click.

9. Optional: Using the data generated by the code in the box above, create and execute a method to the following specifications:
   a. double getMax(double[] d): Return the maximum value in the array of doubles passed.
   b. double getMin(double[] d): Return the minimum value in the array of doubles passed.
   c. double getTotal(double[] d): Return the total (sum) of the values in the array of doubles passed.
   d. double getAvg(double[] d): Return the average of the values in the array of doubles passed.