Chapter 4

Methods

Goal

- Implement well-tested Java methods

4.1 Methods

A java class typically has two or more methods. There are two major components to a method:

1. the method **heading**
2. the block (a pair of curly braces with code to complete the method's functionality

Several modifiers may begin a method heading, such as `public` or `private`. The examples shown here will use only the modifier `public`. Whereas **`private`** methods are only accessible from the class in which they exist, **`public`** methods are visible from other classes. Here is a general form for method headings.

***General Form:* A public method heading**

`public` *return-type method-name* **(***parameter-1* **,** *parameter-2* **,** *…,* *parameter-n* **)**

The *return-type* represents the type of value returned from the method. The return type can be any primitive type, such as `int` or `double` (as in `String`'s `length` method or `BankAccount`'s `withdraw` method, for example). Additionally, the return type can be any reference type, such as `String` or `Scanner`. The return type may also be `void` to indicate that the method returns nothing, as see in `void main` methods.

The *method-name* is any valid Java identifier. Since most methods need one or more values to get the job done, method headings may also specify **parameters** between the required parentheses. Here are a few syntactically correct method headings:

Example Method Headings

```
public int charAt(int index)                            // String
public void withdraw(double withdrawalAmount)           // BankAccount
public int length()                                     // String
public String substring(int startIndex, int endIndex)   // String
```

The other part of a method is the body. A method body begins with a curly brace and ends with a curly brace. This is where the programmer places variable declarations, object constructions, assignments, and other messages that accomplish the purpose of the method. For example, here is the very simple `deposit` method from the `BankAccount` class. This method has access to the parameter `depositAmount` and to the `BankAccount` instance variable named `myBalance` (instance variables are discussed in a later chapter).

```
// The method heading . . .
public void deposit(double depositAmount) {
  // followed by the method body
  myBalance = myBalance + depositAmount;
}
```

Parameters

A **parameter** is an identifier declared between the parentheses of a method heading. Parameters specify the number and type of arguments that must be used in a message. For example, `depositAmount` in the `deposit` method heading above is a parameter of type `double`. The programmer who wrote the method specified the number and type of values the method would need to do its job.

A method may need one, two, or even more arguments to accomplish its objectives. "How much money do you want to withdraw from the `BankAccount` object?" "What is the beginning and ending index of the `substring` you want?" "How many days do you want to add". Parameters provide the mechanism to get the appropriate information to the method when it is called. For example, a `deposit` message to a `BankAccount` object requires that the amount to be deposited, (a `double`), be supplied.

```
public void deposit(double depositAmount)
                                  ↑
            anAccount.deposit(123.45);
```

When this message is sent to anAccount, the value of the argument 123.45 is passed on to the associated parameter `depositAmount`. It may help to read the arrow as an assignment statement. The argument `123.45` is assigned to `depositAmount` and used inside the `deposit` method. This example has a literal argument `(123.45)`. The argument may be any expression that evaluates to the parameter's declared type, such as `(checks + cash)`.

```
double checks = 123.45;
double cash = 100.00;
anAccount.deposit(checks + cash);
```

When there is more than one parameter, the arguments are assigned in order. The replace method of the `String` type requires two character values so the method knows which character to replace and with which character.

```
public String replace(char oldChar, char newChar)
                                  ↖      ↗
  String newString = str.replace('t', 'X');
```

Reading Method Headings

When properly documented, the first part of a method, the heading, explains what the method does and describe the number of arguments and the type All of these things allow the programmer to send messages to objects without knowing the details of the implementation of those methods. For example, to send a message to an object, the programmer must:

- know the method name
- supply the proper number and type of arguments
- use the return value of the method correctly

All of this information is specified in the method heading. For example, the `substring` method of Java's `String` class takes two `int` arguments and evaluates to a `String`.

```
// Return portion of this string indexed from beginIndex through endIndex-1
public String substring(int beginIndex, int endIndex)
```

The method heading for `substring` provides the following information:

- type of value returned by the method: `String`
- method name: `substring`
- number of arguments required: 2
- type of the arguments required: both are `int`

Since `substring` is a method of the `String` class, the message begins with a reference to a string before the dot.

```
String str = new String("small");
assertEquals("mall", str.substring(1, str.length()));

// Can send messages to String literals ...
assertEquals("for", "forever".substring(0, 3));
```

A `substring` message requires two arguments, which specify the beginning and ending index of the `String` to return. This can be observed in the method heading below, which has two parameters named `beginIndex` and `endIndex`. Both arguments in the message `fullName.substring(0,6)` are of type `int` because the parameters in the `substring` method heading are declared as type `int`.

```
public String substring(int beginIndex, int endIndex)
                                        ↑          ↑
                  fullName.substring(0,          6);
```

When this message is sent, the argument 0 is assigned to the parameter `beginIndex`, and the argument 6 is assigned to the parameter `endIndex`. Control is then transferred to the method body where this information is used to return what the method promises. In general, when a method requires more than one argument, the first argument in the message will be assigned to the first parameter, the second argument will be assigned to the second parameter, and so on. In order to get correct results, the programmer must also order the arguments correctly. Whereas not supplying the correct number and type of arguments in a message results in a compile time (syntax) error, supplying the correct number and type of arguments in the wrong order results in a logic error (i.e., the program does what you typed, not what you intended).
And finally, there are several times when the `substring` method will throw an exception because the integer arguments are not in the correct range.

```
String str = "abc";
str.substring(-1, 1)  // Runtime error because beginIndex < 0
str.substring(0, 4)   // Runtime error because endIndex of 4 is off by 1
str.substring(2, 1)   // Runtime error because beginIndex > endIndex
```

Self-Check

Use the following method heading to answer the first three questions that follow. This `concat` method is from Java's `String` class.

```
// Return the concatenation of str at the end of this String object
public String concat(String str)
```

4-1 Using the method heading above, determine the following for String's concat method:

-a return type
-b method name
-c number of arguments
-d first argument type (or class)
-e second argument type (or class)

4-2 Assuming `String s = new String("abc");`, write the return value for each valid message or explain why the message is invalid.

-a `s.concat("xyz");`
-b `s.concat();`
-c `s.concat(5);`
-d `s.concat("x", "y");`
-e `s.concat("wx" + " yz");`
-f `s.concat("d");`

4-3 What values make these assertions pass?

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringTest {

  @Test
  public void testConcat() {
     String s = "abc";
     assertEquals(____________, s.concat("!"));
     assertEquals(____________, s.concat("cba"));
     assertEquals(____________, s.concat("123"));
  }
}
```

Use the following method heading to answer the first three questions that follow. This `concat` method is from Java's `String` class.

```
  // Returns a new string resulting from replacing all
  // occurrences of oldChar in this string with newChar.
  public String replace(char oldChar, char newChar)
```

4-4 Using the method heading above, determine the following for String's replace method:

-a return type
-b method name
-c number of arguments
-d first argument type
-e second argument type

4-5 Assuming `String s = new String("abcabc");`, write the return value for each valid message or explain why the message is invalid.

-a `s.replace("a");`
-b `s.replace('c', 'Z');`
-c `s.replace('b', 'Z');`
-d `s.replace("x", "y");`
-e `s.replace('a', 'X');`
-f `s.concat('X', 'a');`

4-6 What values make the assertions pass?

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringTest {
  @Test
  public void testReplace () {
     String s = "aabbcc";
     assertEquals("__a.___", s.replace('a', 'T'));
     assertEquals("__b.___", s.replace ('b', ' '));
     assertEquals("__c.___", s.replace ('c', 'Y'));
  }
}
```

Methods that return Values

When a method is called, the values of the arguments are copied to the parameters so the values can be used by the method. The flow of control then transfers to the called method where those statements are executed. One of those statements in all non-void methods must return a value. This is done with the Java `return` statement that allows a method to return information. Here is the general form:

General Form **return statement**

`return` *expression* **;**

The following examples show the return statement in the context of complete methods. The three methods are captured in a class named `ExampleMethods`, which implies there is no relationship between the methods. It simply provides methods with different return types.

```
// This class contains several unrelated methods to provide examples.
public class ExampleMethods {

  // Return a number that is twice the value of the argument.
  public double f(double argument) {
    return 2.0 * argument;
  }

  // Return true if argument is an odd integer, false when argument is even.
  public boolean isOdd(int argument) {
    return (argument % 2 != 0);
  }

  // Return the first two and last two characters of the string.
  // Precondition: str.length() >= 4
  public String firstAndLast(String str) {
    int len = str.length();
    String firstTwo = str.substring(0, 2);
    String lastTwo = str.substring(len - 2, len);
    return firstTwo + lastTwo;
  }

} // End of class with three example methods.
```

When a `return` statement is encountered, the *expression* that follows `return` replaces the message part of the statement. This allows a method to communicate information back to the caller. Whereas a `void` method returns nothing (see any of the void `main` methods or `test` methods), any method that has a return type other than `void` *must* return a value that matches the return type. So, a method declared to return a `String` must return a reference to a `String` object. A method declared to return a `double` must return a primitive `double` value. Fortunately, the compiler will complain if you forget to return a value or you attempt to return the wrong type of value.

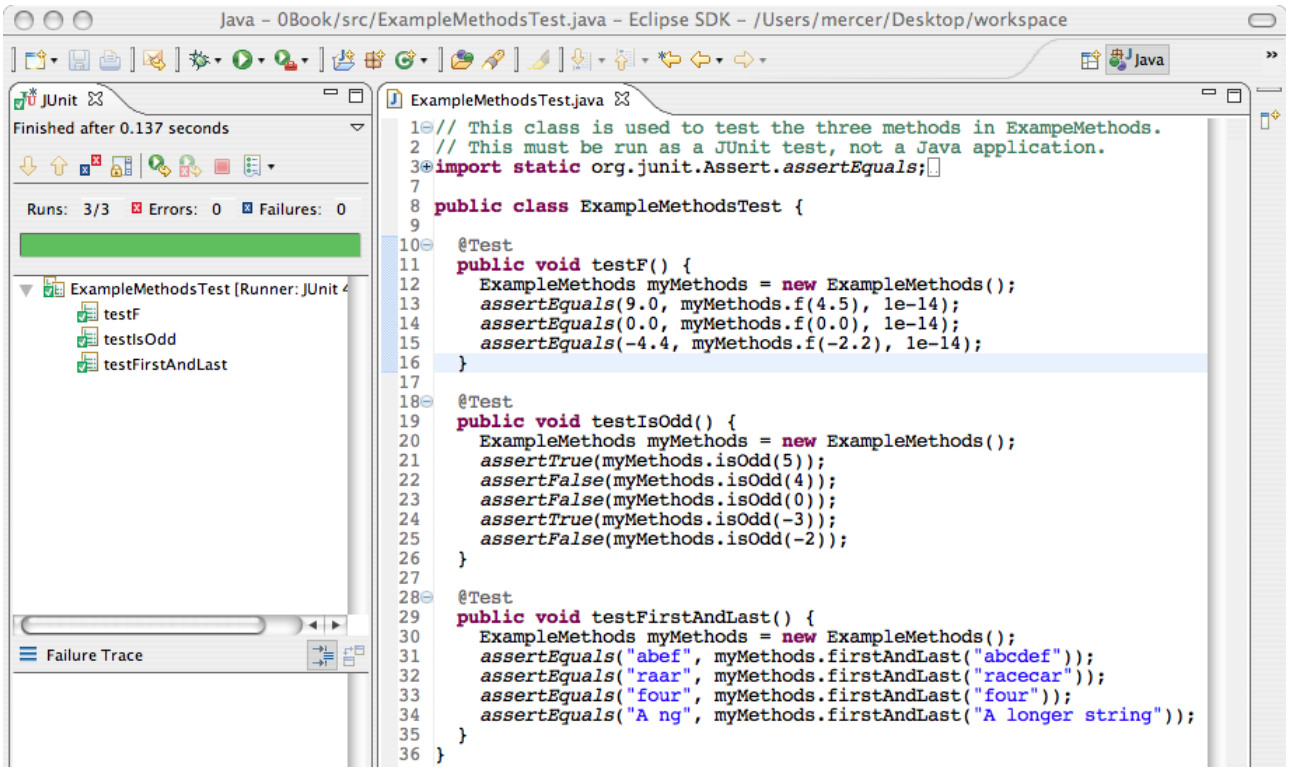
As suggested in Chapter 1, testing can occur at many times during software development. When you write a method, test it. For example, a test method for `firstAndLast` could look like this.

```
  @Test
  public void testFirstAndLast() {
    ExampleMethods myMethods = new ExampleMethods();
    assertEquals("abef", myMethods.firstAndLast("abcdef"));
    assertEquals("raar", myMethods.firstAndLast("racecar"));
    assertEquals("four", myMethods.firstAndLast("four"));
    assertEquals("A ng", myMethods.firstAndLast("A longer string"));
  }
```

Methods may exist in any class. We could use test methods in the same class as the methods being tested because it is convenient to write methods and tests in the same file. That approach would also have the benefit not requiring an `new ExampleMethods()` object thereby requiring us to write less code. However, it is common practice to write tests in a separate test class. Conveniently, we can place test methods for each of the three `ExampleMethods` in another file keeping tests separate from the methods.

```
// This class is used to test the three methods in ExampleMethods.
import static org.junit.Assert.*;
import org.junit.Test;

public class ExampleMethodsTest {

  @Test
  public void testF() {
    ExampleMethods myMethods = new ExampleMethods();
    assertEquals(9.0, myMethods.f(4.5), 1e-14);
    assertEquals(0.0, myMethods.f(0.0), 1e-14);
    assertEquals(-4.4, myMethods.f(-2.2), 1e-14);
  }


  @Test
  public void testIsOdd() {
    ExampleMethods myMethods = new ExampleMethods();
    assertTrue(myMethods.isOdd(5));
    assertFalse(myMethods.isOdd(4));
    assertFalse(myMethods.isOdd(0));
    assertTrue(myMethods.isOdd(-3));
    assertFalse(myMethods.isOdd(-2));
  }

  @Test
  public void testFirstAndLast() {
    ExampleMethods myMethods = new ExampleMethods();
    assertEquals("abef", myMethods.firstAndLast("abcdef"));
    assertEquals("raar", myMethods.firstAndLast("racecar"));
    assertEquals("four", myMethods.firstAndLast("four"));
    assertEquals("A ng", myMethods.firstAndLast("A longer string"));
  }
}
```

This is a relatively new way to implement and test methods made possible with the JUnit testing framework. Most college textbooks use printlns and user input to show the results of running code that requires several program runs with careful input of values and careful inspection of the output each time. This textbook integrates testing with JUnit, an industry-level testing framework that makes software development more efficient and less error prone. It is easer to test and debug your code. You are more likely to find errors more quickly. When run as a JUnit test, all assertions pass in all three test-methods and the green bar appears.



With JUnit, you can set up your tests and methods and run them with no user input. The process can be easily repeated while you debug. Writing assertions also makes us think about what the method should do before writing the method. Writing assertions will help you determine how to best test code now and into the future, a worthwhile skill to develop that costs little time.

Self-Check

4-7 a) Write a complete test method named `testInRange` as if it were in class `ExampleMethodsTest` to test method `inRange` that will be placed in class `ExampleMethods`. Here is the method heading for the method that will go into class `ExampleMethods`.

```
// Return true if number is in the range of 1 through 10 inclusive.
public boolean inRange(int number)
```

b) Write the complete method named `inRange` as if it were in `ExampleMethods`.

4-8 a) Write a complete test method named `testAverageOfThree` as if it were in class `ExampleMethodsTest` to test method `averageOfThree` that will be placed in class `ExampleMethods`. Here is the method heading for the method that will go into class `ExampleMethods`.

```
// Return the average of the three arguments.
public double averageOfThree(double a, double b, double c)
```

b) Write the complete method named `averageOfThree` as if it were in `ExampleMethods`.

4-9 a) Write a complete test method named `testRemoveMiddleTwo` as if it were in class `ExampleMethodsTest` to test method `removeMiddleTwo` that will be placed in class `ExampleMethods`. `removeMiddleTwo` should return a string that has all characters except the two in the middle. Assume the String argument has two or more characters. Here is the method heading for the method that will go into class `ExampleMethods`.

```
// Return the String argument with the middle two character missing.
// removeMiddleTwo("abcd") should return "ad"
// removeMiddleTwo("abcde") should return "abd"
// Precondition: sr.length() >= 2
public String removeMiddleTwo(String str)
```

b) Write the complete method named `removeMiddleTwo` as if it were in the `ExampleMethods` class.

How do we know what to test?

Methods are deigned to have parameters to allow different arguments. This makes them generally useful in future applications. But how do we know these methods work? Is it important that they are correct? Software quality is important. It is impossible to write perfect code.

One effective technique to ensure a method does what it is supposed to do is to write assertions to fully test the method. Asserting a method returns the correct value for one value is usually not enough. How many assertions should we make? What arguments should we use? The answers are not preordained. However, by pushing the limits of all the possible assertions and values we can think of, and doing this repeatedly, we get better at testing. Examples help. Consider this maxOfThree method.

```
// Return the maximum value of the integer arguments.
public int maxOfThree(int a, int b, int c)
```

As recommended in Chapter 1, it helps to have sample input with the expected result. Some test cases to consider include all three numbers the same, all 0, and certainly all different. Testing experts will tell you that test cases include all permutations of the different integers. So the test cases should include the max of (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1). Whenever negative numbers are allowed, write assertions with negative numbers.

This large number of test cases probably seems excessive, but it doesn't take much time. There are a large number of algorithms that will make maxOfThree work. I have personally seen many of these that work in most cases, but not all cases. Especially interesting are the test cases when two are equal (students often write > rather than >=). So other test cases should include the max of (1, 2, 2), (2, 1, 2), and (2, 2, 1).

Since we can setup these test cases with the expected value and actual value next to each other and then run the tests once (or more than once if you detect a bug or use incorrect expected values). This test method contains more assertions than you would typically need due to the nature of the problem where the largest could be any of the three arguments and any one could equal another two.

```
@Test
public void testMaxOfThree() {
  ExampleMethods myMethods = new ExampleMethods();

  // All equal
  assertEquals(5, myMethods.maxOfThree(5, 5, 5));
  assertEquals(-5, myMethods.maxOfThree(-5, -5, -5));
  assertEquals(0, myMethods.maxOfThree(0, 0, 0));
```

```
    // All permutations of 3 different arguments
    assertEquals(3, myMethods.maxOfThree(1, 2, 3));
    assertEquals(3, myMethods.maxOfThree(1, 3, 2));
    assertEquals(3, myMethods.maxOfThree(2, 1, 3));
    assertEquals(3, myMethods.maxOfThree(2, 3, 1));
    assertEquals(3, myMethods.maxOfThree(3, 1, 2));
    assertEquals(3, myMethods.maxOfThree(3, 2, 1));

    // All permutations of two integers that are the largest
    assertEquals(2, myMethods.maxOfThree(1, 2, 2));
    assertEquals(2, myMethods.maxOfThree(2, 1, 2));
    assertEquals(2, myMethods.maxOfThree(2, 2, 1));

    // All permutations of two integers that are the smallest
    assertEquals(2, myMethods.maxOfThree(1, 1, 2));
    assertEquals(2, myMethods.maxOfThree(2, 1, 1));
    assertEquals(2, myMethods.maxOfThree(1, 2, 1));
  }
```

Self-Check

4-10 Consider a method that takes three integer arguments representing the three sides of a triangle. The method must reports whether the triangle is scalene (three sides different), isosceles (two sides equal), equilateral, or not a triangle (cannot be). What tests should be written and for each, what should the result be?

4-11 Boggle tests your ability to find words in a random array of dice with letters. Words must be in the range of 3..16 characters inclusive. Method `inRange(String str)` must return true if the length of `str` is in the range of 3 through 16 characters inclusive. What tests should be written and for each, what should the result be?

Answers to Self-Check Questions

4-1
- -a String
- -b concat
- -c 1
- -d String
- -e There is no second parameter

4-2
- -a "abcxyz
- -b needs argument
- -c 5 wrong type;
- -d One too many arguments
- -e "abcwx yz"
- -f "abcd"

4-3
```
assertEquals("abc!", s.concat("!"));
assertEquals("abccba" , s.concat("cba"));
assertEquals("abc123", s.concat("123"));
```

4-4
- -a String
- -b replace
- -c 2
- -d char
- -e char

4-5
- -a need 2 char arguments
- -b "abZabZ"
- -c "aZcaZb
- -d wrong type arguments. Need char, not String
- -e "XbcXbc"
- -f Wrong type and number of arguments for concat

4-6
```
assertEquals("TTbbcc", s.replace('a', 'T'));
assertEquals("aa  cc", s.replace('b', ' '));
assertEquals("aabbYY", s.replace('c', 'Y'));
```

4-7 a)
```
@Test
public void testInRange() {
  assertFalse(inRange(0));
  assertTrue(inRange(1));
  assertTrue(inRange(5));
  assertTrue(inRange(10));
  assertFalse(inRange(11));
}
```

b)
```
public boolean inRange(int number) {
  return (number >= 1) && (number <= 10);
}
```

4-8 a)
```
@Test
public void testAverageThree() {
  ExampleMethods myMethods = new ExampleMethods();
  assertEquals(0.0, myMethods.averageOfThree(0.0, 0.0, 0.0), 0.1);
  assertEquals(90.0, myMethods.averageOfThree(90.0, 90.0, 90.0), 0.1);
  assertEquals(82.5, myMethods.averageOfThree(90.0, 80.5, 77.0), 0.1);
  assertEquals(-2.0, myMethods.averageOfThree(-1, -2, -3), 0.1);
}
```

b)
```
public double averageOfThree(double a, double b, double c) {
  return (a + b + c) / 3.0;
}
```

4-9 a)
```
@Test
public void testRemoveMiddleTwo() {
  ExampleMethods myMethods = new ExampleMethods();
  assertEquals("", myMethods.removeMiddleTwo("12"));
  assertEquals("ad", myMethods.removeMiddleTwo("abcd"));
  assertEquals("ade", myMethods.removeMiddleTwo("abcde"));
  assertEquals("abef", myMethods.removeMiddleTwo("abcdef"));
}
```

b)
```
public String removeMiddleTwo(String str) {
  int mid = str.length() / 2;
  return str.substring(0, mid-1) + str.substring(mid + 1, str.length());
}
```

4-10 Equilateral: (5, 5, 5)
Isosceles with permutations: (3, 3, 2) (2, 3, 3) (3, 2, 3)
Scalene with permutations: (2, 3, 4) (2 ,4, 3) (3, 2, 4) (3, 4, 2) (4 , 2, 3) (4, 3, 2)
Not a triangle and permutations: (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 2, 1) (3, 1, 2)
Not a triangle and permutations: (1, 2, 4) (1, 4, 2) (2, 1, 4) (2, 4, 1) (4, 2, 1) (4, 1, 2)
Not a triangle and permutations: (0, 2, 3) (1, 0, 2) (2, 1, 0)
Not a triangle with negative lengths and permutations: (-1, 2, 3) (1, -2, 3) (1, 2, -3)
Not a triangle, all negative, but would be if equilateral if positive: (-5, -5, -5)

4-11
```
@Test
public void testInRangeString() {
  ExampleMethods myMethods = new ExampleMethods();
  assertFalse(myMethods.inRange("")); // Empty string
  assertFalse(myMethods.inRange("ab")); // On the border -1
  assertTrue(myMethods.inRange("abc")); // On the border
  assertTrue(myMethods.inRange("abcd")); // On the border + 1
  assertTrue(myMethods.inRange("abcdef")); // In the middle
  assertTrue(myMethods.inRange("1234567890")); // In the middle
  assertTrue(myMethods.inRange("123456789012345")); // On the border - 1
  assertTrue(myMethods.inRange("1234567890123456")); // On the border
  assertFalse(myMethods.inRange("12345678901234567")); // On the border + 1
}
```