Chapter 3

Objects and JUnit

Goals

This chapter is mostly about using objects and getting comfortable with sending messages to objects. Several new types implemented as Java classes are introduced to show just a bit of Java's extensive library of classes. This small subset of classes will then be used in several places throughout this textbook. You will begin to see that programs have many different types of objects. After studying this chapter, you will be able to:

- Use existing types by constructing objects
- Be able to use existing methods by reading method headings and documentation
- Introduce assertions with JUnit
- Evaluate Boolean expressions that result in true or false.

3.1 Find the Objects

Java has two types of values: primitive values and reference values. Only two of Java's eight primitive types (`int` and `double`) and only one of Java's reference types (the Scanner class) have been shown so far. Whereas a primitive variable stores only one value, a reference variable stores a reference to an object that may have many values. Classes allow programmers to model real-world entities, which usually have more values and operations than primitives.

Although the Java programming language has only eight primitive types, Java also come with thousands of reference types (implemented as Java classes). Each new release of Java tends to add new reference types. For example, instances of the Java `String` class store collections of characters to represent names and addresses in alphabets from around the world. Other classes create windows, buttons, and input areas of a graphical user interface. Other classes represent time and calendar dates. Still other Java classes provide the capability of accessing databases over networks using a graphical user interface. Even then, these hundreds of classes do not supply everything that every programmer will ever need. There are many times when programmers discover they need their own classes to model things in their applications. Consider the following system from the domain of banking:

The Bank Teller Specification

> Implement a bank teller application to allow bank customers to access bank accounts through unique identification. A customer, with the help of the teller, may complete any of the following transactions: withdraw money, deposit money, query account balances, and see the most recent 10 transactions. The system must maintain the correct balances for all accounts. The system must be able to process one or more transactions for any number of customers.

You are not asked to implement this system now. However, you should be able to pick out some things (objects) that are relevant to this system. This is the first step in the analysis phase of object-oriented software development. One simple tool for finding objects that potentially model a solution is to write down the nouns and noun phrases in the problem statement. Then consider each as a candidate object that might eventually represent part of the system. The objects used to build the system come from sources such as

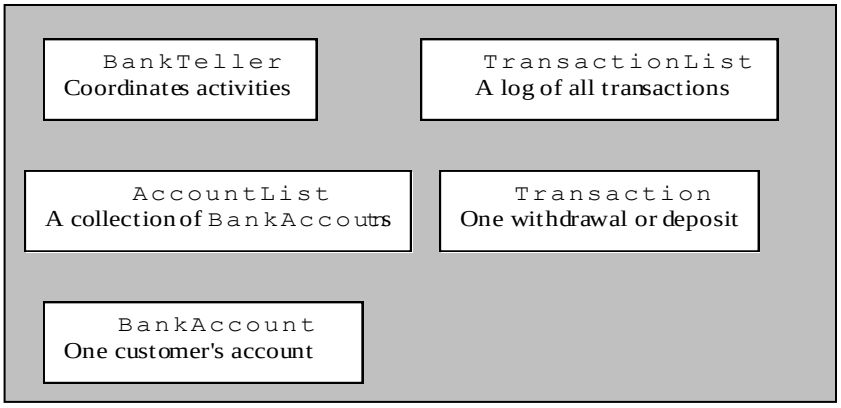
- the problem statement
- an understanding of the problem domain (knowledge of the system that the problem statement may have missed or taken for granted)
- the words spoken during analysis
- the classes that come with the programming language

The objects should model the real world if possible. Here are some candidate objects:

Candidate Objects to Model a Solution

bank teller	transaction
customers	most recent 10 transactions
bank account	window

Here is a picture to give an impression of the major objects in the bank teller system. The `BankTeller` will accomplish this by getting help from many other objects.



We now select one of these objects—`BankAccount`.

`BankAccount` Objects

Implementing a **`BankAccount`** type as a Java class gives us the ability to have many (thousands of) `BankAccount` objects. Each instance of `BankAccount` represents an account at a bank. Using your knowledge of the concept of a bank account, you might recognize that each `BankAccount` object should have its own account number and its own account balance. Other values could be part of every `BankAccount` object: a transaction list, a personal identification number (PIN), and a mother's maiden name, for example. You might visualize other banking methods, such as creating a new account, making deposits, making withdrawals, and accessing the current balance. There could also be many other banking messages—`applyInterest` and `printStatement`, for example.

As a preview to a type as a collection of methods and data, here is the `BankAccount` type implemented as a Java class and used in the code that follows. The Java class with methods and variables to implement a new type will be discussed in Chapters 4 (Methods) and 10 (Classes). Consider this class to be a blueprint that can be used to construct many `BankAccount` objects. Each `BankAccount` object

will have its their own `balance` and `ID`. Each `BankAccount` will understand the same four messages: `getID`, `getBalance`, `deposit`, and `withdraw`.

```
// A type that models a very simple account at a bank.
public class BankAccount {

  // Values that each object "remembers":
  private String ID;
  private double balance;

  // The constructor:
  public BankAccount(String initID, double initBalance) {
    ID = initID;
    balance = initBalance;
  }

  // The four methods:
  public String getID() {
    return ID;
  }

  public double getBalance() {
    return balance;
  }

  public void deposit(double depositAmount) {
    balance = balance + depositAmount;
  }

  public void withdraw(double withdrawalAmount) {
    balance = balance - withdrawalAmount;
  }
}
```

This `BankAccount` type has been intentionally kept simple for ease of study. The available `BankAccount` messages include—but are not limited to—`withdraw`, `deposit`, `getID`, and `getBalance`. Each will store an account ID and a balance.

Instances of `BankAccount` are constructed with two arguments to help initialize these two values. You can supply two initial values in the following order:

1. a sequence of characters (a string) to represent the account identifier (a name, for example)
2. a number to represent the initial account balance

Here is one desired object construction that has two arguments for the purpose of initializing the two desired values:

```
BankAccount anAccount = new BankAccount("Chris", 125.50);
```

The construction of new objects (the creation of new instances) requires the keyword `new` with the class name and any required initial values between parentheses to help initialize the state of the object. The general form for creating an instance of a class:

***General Form:* Constructing objects (initial values are optional)**

class-name object-name **=** **new** *class-name***(** *initial-value(s)* **);**

Every **object** has

1. a name (actually a reference variable that stores a reference to the object)
2. state (the set of values that the object remembers)
3. messages (the things objects can do and reveal)

Every instance of a class will have a reference variable to provide access to the object. Every instance of a class will have its own unique state. In addition, every instance of a class will understand the same set of messages. For example, given this object construction,

```
BankAccount anotherAccount = new BankAccount("Justin", 60.00);
```

we can derive the following information:

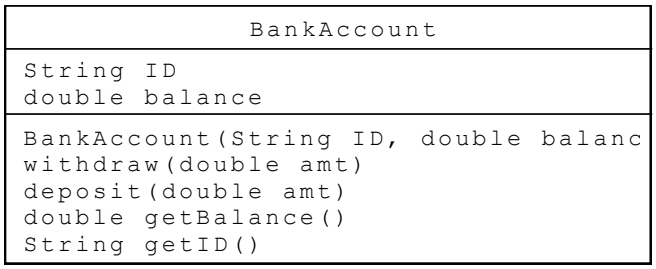
1. name: `anotherAccount`
2. state: an account ID of `"Justin"` and a balance of `60.00`
3. messages: anotherAccount understands `withdraw`, `deposit`, `getBalance`, ...

Other instances of `BankAccount` will understand the same set of messages. However, they will have their own separate state. For example, after another `BankAccount` construction,

```
BankAccount theNewAccount = new BankAccount("Kim", 1000.00);
```

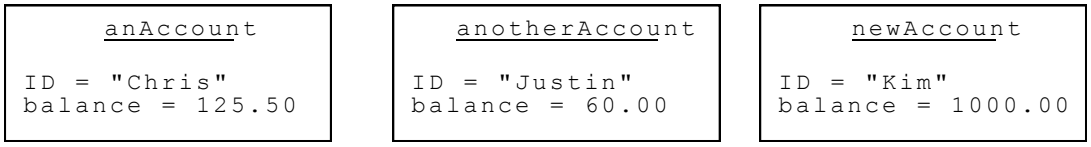
`theNewAccount` has its own ID of "Kim" and its own balance of 1000.00.

The three characteristics of an object can be summarized with diagrams. This class diagram represents one class.



A class diagram lists the class name in the topmost compartment. The instance variables appear in the compartment below it. The bottom compartment captures the methods.

Objects can also be summarized in instance diagrams.



These three object diagrams describe the current state of three different `BankAccount` objects. One class can be used to make have many objects, each with its own separate state (set of values).

Sending Messages to Objects

In order for objects to do something, your code must send messages to them. A **message** is a request for the object to provide one of it services through a method.

General Form: **Sending a message to an object**

object-name **.** *message-name* **(***argument1* **,** *argument2* **, ...)**

Some messages ask for the state of the object. Other messages ask an object to do something, For example, each `BankAccount` object was designed to have the related operations `withdraw`, `deposit`, `getBalance`, and `getID`. These messages ask the two different `BankAccount` objects to return information:

```
anAccount.getID();
anAccount.getBalance();
anotherAccount.getID();
anotherAccount.getBalance();
```

These messages ask two different `BankAccount` objects to do something:

```
  anAccount.withdraw(40.00);
  anAccount.deposit(100.00);
  anotherAccount.withdraw(20.00);
  anotherAccount.deposit(157.89);
```

The optional **arguments**—expressions between the parentheses—are the values required by the method to fulfill its responsibility. For example, `withdraw` needs to know how much money to withdraw. On the other hand, `getBalance` doesn't need any arguments to return the current balance of the `BankAccount` object. The output below indicates `deposit` and `withdraw` messages modify the account balances in an expected manner:

```
// Construct two objects and send messages to them.
public class ShowTwoBankAccountObjects {

  public static void main(String[] args) {

    BankAccount b1 = new BankAccount("Kim", 123.45);
    BankAccount b2 = new BankAccount("Chris", 500.00);

    System.out.println("Initial values");
    System.out.println(b1.getID() + ": " + b1.getBalance());
    System.out.println(b2.getID() + ": " + b2.getBalance());

    b1.deposit(222.22);
    b1.withdraw(20.00);
    b2.deposit(55.55);
    b2.withdraw(10.00);
    System.out.println();
    System.out.println("Value after deposit and withdraw messages");
    System.out.println(b1.getID() + ": " + b1.getBalance());
    System.out.println(b2.getID() + ": " + b2.getBalance());
  }
}
```

Output

```
Initial values
Kim: 123.45
Chris: 500.0

Value after deposit and withdraw messages
Kim: 325.67
Chris: 545.55
```

3.2 Making Assertions about Objects with JUnit

The `println` statements in the program above reveal the changing state of objects. However, in such examples, many lines can separate the output from the messages that affect the objects. This makes it a bit awkward to match up the expected result with the code that caused the changes. The current and changing state of objects can be observed and confirmed by making assertions. An assertion is a statement that can relay the current state of an object or convey the result of a message to an object. Assertions can be made with methods such `assertEquals`.

General Form: JUnit's assertEquals method for int and double values

```
assertEquals(int expected, int actual);
assertEquals(double expected, double actual, double errorTolerance);
```

Examples to assert integer expressions:

```
assertEquals(2, 5 / 2);
assertEquals(14, 39 % 25);
```

Examples to assert a floating point expression:

```
assertEquals(325.67, b1.getBalance(), 0.001);
assertEquals(545.55, b2.getBalance(), 0.001);
```

With `assertEquals`, an assertion will be true—or will "pass"—if the *expected* value equals the *actual* value. When comparing floating-point values, a third argument is needed to represent the error tolerance, which is the amount by which two real numbers may differ and still be equal. (Due to round off error, and the fact that numbers are stored in base 2 (binary) rather than in base 10 (decimal), two expressions that we consider "equal" may actually differ by a very small amount. This textbook will often use the very small error tolerance of 1e-14 or 0.00000000000001. This means that the following two numbers would be considered equal within 1e-14:

```
assertEquals(1.23456789012345, 1.23456789012346, 1e-14);
```

In contrast, these numbers are not considered equal when using an error factor of `1e-14`.

```
assertEquals(1.23456789012345, 1.23456789012347, 1e-14);
```

So using 1e14 ensures two values are equals to 13 decimal places, which is about as close as you can get. JUnit assertions allow us to place the expected value next to messages that reveal the actual state. This makes it easer to demonstrate the behavior of objects and to learn about new types. Later, you will see how assertions help in designing and testing your own Java classes, by making sure they have the correct behavior.

The `assertEquals` method is in the `Assert` class of `org.junit`. The `Assert` class needs to be imported (shown later) or `assertEquals` needs to be qualified (shown next).

```
// Construct two BankAccount objects
BankAccount anAccount = new BankAccount("Kim", 0.00);
BankAccount anotherAccount = new BankAccount("Chris", 500.00);

// These assertions pass
org.junit.Assert.assertEquals(0.00, anAccount.getBalance(), 1e-14);
org.junit.Assert.assertEquals("Kim", anAccount.getID());
org.junit.Assert.assertEquals("Chris", anotherAccount.getID());
org.junit.Assert.assertEquals(500.00, anotherAccount.getBalance(), 1e-14);

// Send messages to the BankAccount objects
anAccount.deposit(222.22);
anAccount.withdraw(20.00);
anotherAccount.deposit(55.55);
anotherAccount.withdraw(10.00);

// These assertions pass
org.junit.Assert.assertEquals(202.22, anAccount.getBalance(), 1e-14);
org.junit.Assert.assertEquals(545.55, anotherAccount.getBalance(), 1e-14);
```

To make these assertions, you must have access to the JUnit testing framework, which is available in virtually all Java development environments. Eclipse does. Then assertions like those above can be placed in methods preceded by `@Test`. These methods are known as **test methods**. They are most often used to test a new method. The test methods here demonstrate some new types. A test method begins in a very specific manner:

```
@org.junit.Test
public void testSomething()  {   // more to come
```

Much like the `main` method, test methods are called from another program (JUnit). Test methods need things from the org.junit packages. This code uses `fully qualified names`.

```
public class FirstTest {

  @org.junit.Test // Marks this as a test method.
  public void testDeposit() {
    BankAccount anAccount = new BankAccount("Kim", 0.00);
    anAccount.deposit(123.45);
    org.junit.Assert.assertEquals(123.45, anAccount.getBalance(), 0.01);
  }
}
```

Adding imports shortens code in all test methods. This feature allows programmers to write the method name without the class to which the method belongs. The modified class shows that imports reduce the amount of code by `org.junit.Assert` and `ord.junit` for every test method and assertion, which is a good thing since much other code that is required.

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

public class FirstTest {

  @Test  // Marks this as a test method.
  public void testDeposit() {
    BankAccount anAccount = new BankAccount("Kim", 0.00);
    anAccount.deposit(123.45);
    assertEquals(123.45, anAccount.getBalance());
  }

  @Test  // Marks this as a test method.
  public void testWithdraw() {
    BankAccount anotherAccount = new BankAccount("Chris", 500.00);
    anotherAccount.withdraw(160.01);
    assertEquals(339.99, anotherAccount.getBalance());
  }

} // End unit test for BankAccount
```

Running JUnit

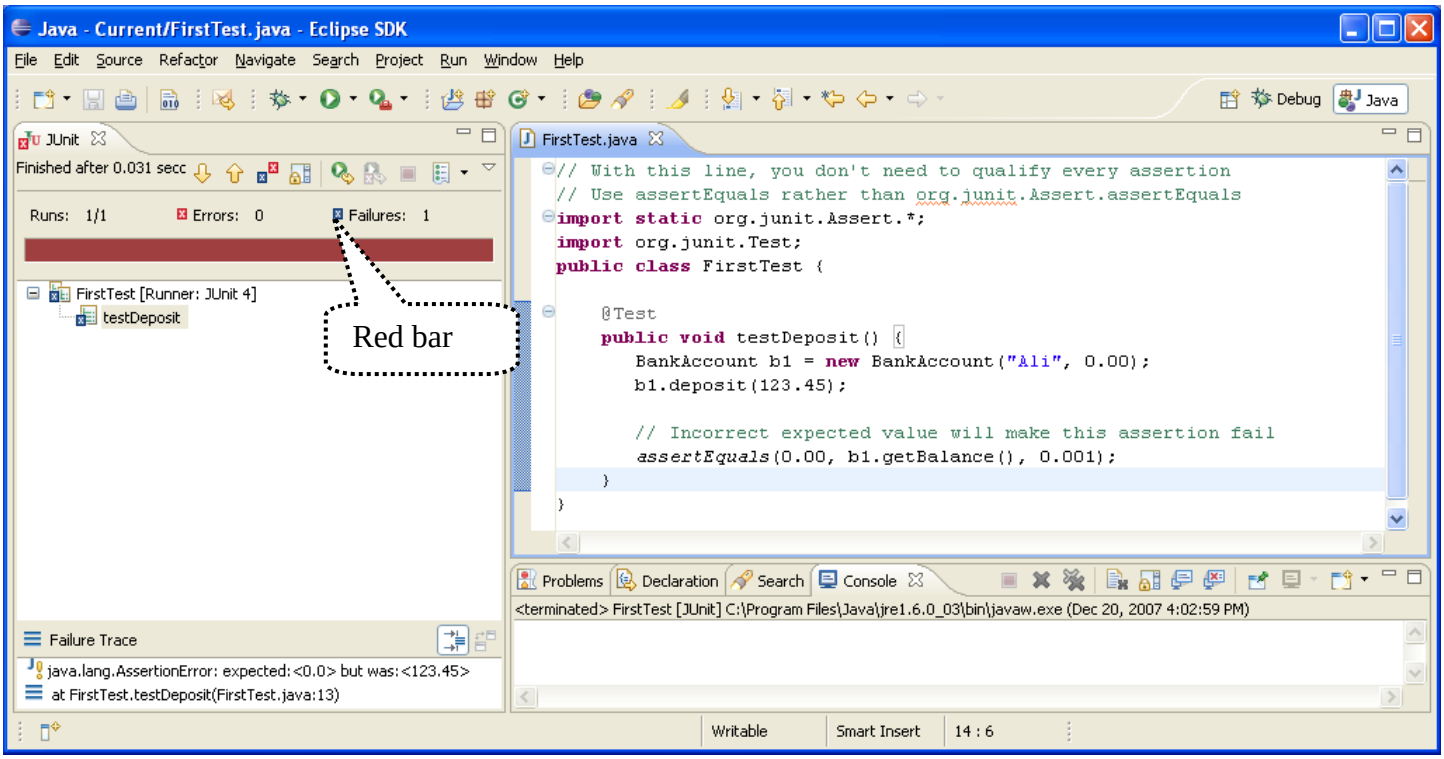
An assertion passes when the actual value equals the expected value in `assertEquals`.

```
  assertEquals(4, 9 / 2); // Assertion passes
```

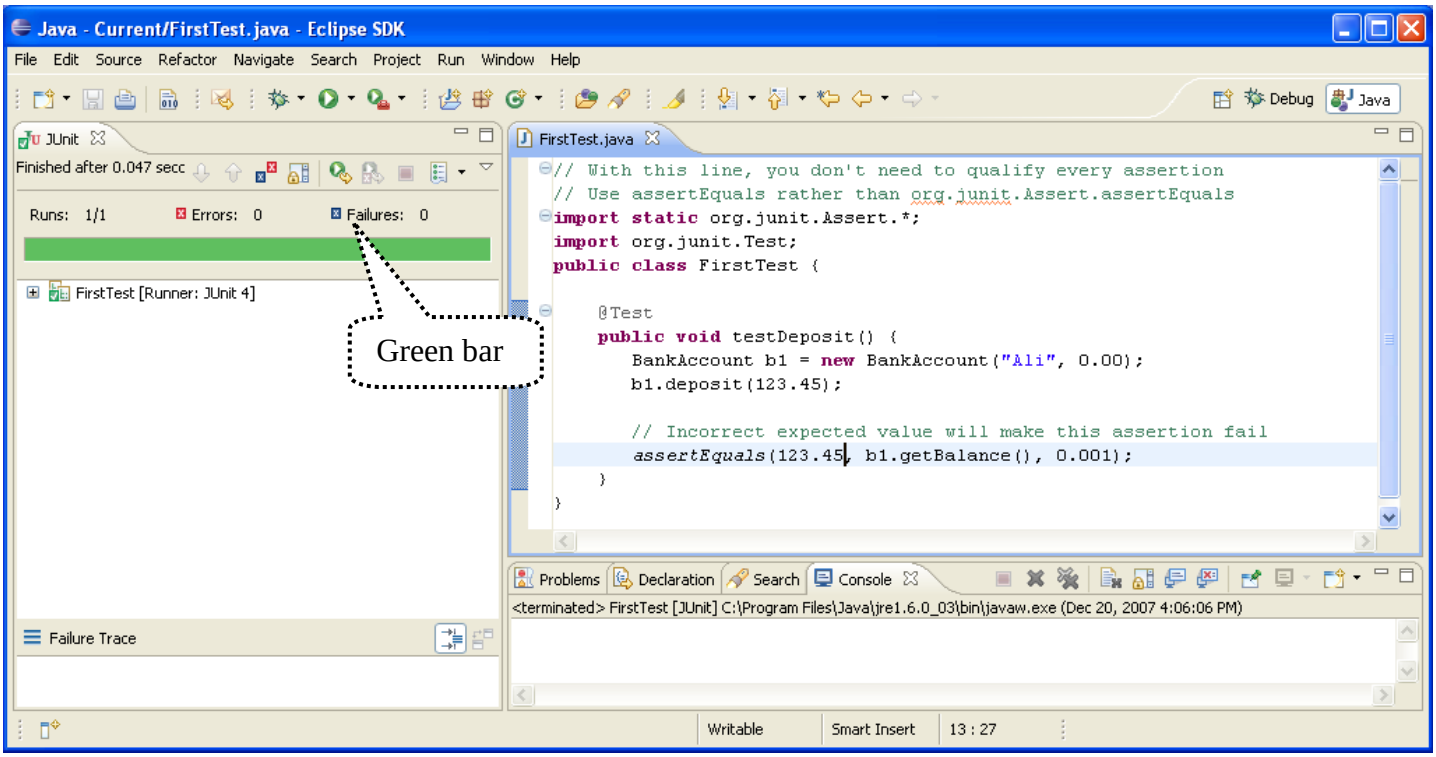
An assertion failes when the actual values does not equal the expected value.

```
  assertEquals(4.5, 9 / 2, 1e-14); // Assertion fails
```

With integrated development environments such as Eclipse, Netbeans, Dr. Java, BlueJ, when an assertion fails, you see a red bar. For example, this screenshot of Eclipse shows a red bar.



The expected and actual values are shown in the lower left corner when the code in FirstTest.java is run as a JUnit test. Changing the `testDeposit` method to have the correct expected value results in a green bar, indicating all assertions have passed successfully. Here is JUnit's window when all assertions pass:



assertTrue and assertFalse

JUnit Assert class has several other methods to demonstrate and test code. The `assertTrue` assertion passes if its Boolean expression argument evaluates to true. The `assertFalse` assertion passes if the Boolean expression evaluates to false.

```
// Use two other Assert methods: assertTrue and assertFalse
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class SecondTest {

  @Test
  public void showAssertTrue() {
    int quiz = 98;
    assertTrue(quiz >= 60);
  }

  @Test
  public void showAssertFalse() {
    int quiz = 55;
    assertFalse(quiz >= 60);
  }
}
```

The three Assert methods—`assertEquals`, `assertTrue`, and `assertFalse`—cover most of what we'll need.

3.3 String Objects

Java provides a `String` type to store a sequence of characters, which can represent an address or a name, for example. Sometimes a programmer is interested in the current length of a `String` (the number of characters). It might also be necessary to discover if a certain `substring` exists in a string. For example, is the substring `", "` included in the string `"Last, First"`. and if so, where does `substring` `"the"` begin? Java's `String` type, implemented as a Java class, provides a large number of methods to help with such problems required knowledge of the string value. You will use `String` objects in many programs.

Each `String` object stores a collection of zero or more characters. `String` objects can be constructed in two ways.

General Form: **Constructing `String` objects in two different ways**

`String` *identifier* **`=`** **`new`** **`String`**`(`*string-literal*`);`
`String` *identifier* **`=`** *string-literal*`;`

Examples

```
String stringReference = new String("A String Object");
String anotherStringReference = "Another";
```

String length

For more specific examples, consider two `length` messages sent to two different `String` objects. Both messages evaluate to the number of characters in the `String`.

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


public class StringTest {

  @Test
  public void showLength() {
    String stringReference = new String("A String Object");
    String anotherStringReference = "Another";
    // These assertions pass
    assertEquals(15, stringReference.length());
    assertEquals(7, anotherStringReference.length());
  }

  // . . . more test methods will appear below
}
```

String charAt

A `charAt` message returns the character located at the index passed as an `int` argument. Notice that `String` objects have zero-based indexing. The first character is located at index 0, and the second character is located at index 1, or `charAt(1)`.

```
  @Test
  public void showcharAt() {
    String stringReference = new String("A String");

    assertEquals('A', stringReference.charAt(0)); // Evaluates to 'A'
    assertEquals('r', stringReference.charAt(4)); // Evaluates to 'r'

    int len = stringReference.length() - 1;
    assertEquals('g', stringReference.charAt(len)); // The last char
  }
```

String indexOf

An `indexOf` message sent to a `String` object returns the index of the first character where the `String` argument is found. For example, `"no-yes".indexOf("yes")` returns 3. If the `String` argument does not exist, `indexOf` returns -1.

```
  @Test
  public void showIndexOf() {
    String stringReference = new String("A String Object");
    assertEquals(3, stringReference.indexOf("tri"));
    assertEquals(-1, stringReference.indexOf("not here"));
  }
```

Concatenation with the + operator

Programmers often make one `String` object from two separate strings with the `+` operator, that concatenates (connects) two or more strings into one string.

```
  @Test
  public void showConcatenate() {
    String firstName = "Kim";
    String lastName = "Madison";
    String fullName = lastName + ", " + firstName;
    assertEquals("Madison, Kim", fullName);
  }
```

String substring

A substring message returns the part of a string indexed by the beginning index through the ending index - 1.

```
@Test
public void showSubString() {
  String str = "Smiles a Lot";
  assertEquals("mile", str.substring(1, 5));
}
```

String toUpperCase and toLowerCase

A `toUpperCase` message sent to a `String` object returns a new string that is the uppercase equivalent of the receiver of the message. A `toLowerCase` message returns a new string with all uppercase letters in lowercase.

```
@Test
public void testToUpperCase() {
  String str = new String("MiXeD cAsE!");
  assertEquals("MIXED CASE!", str.toUpperCase());
  assertEquals("mixed case!", str.toLowerCase());
  assertEquals("MiXeD cAsE!", str);  // str did not change!
}
```

Although it may sound like `toUpperCase` and `toLowerCase` modify String objects, they do not. Once constructed, String objects can not be changed. String objects are immutable. Simply put, there are no `String` messages that can modify the state of a `String` object. The final assertion above shows that `str.equals("MiXeD cAsE!")` still, even after the other two messages were sent. Strings are immutable to save memory. Java also supplies `StringBuilder`, a string type that has methods that do modify the objects.

Use an assignment if you want to change the String reference to refer to a different String.

```
@Test
public void showHowToUpperCaseWithAssignment() {
  String str = new String("MiXeD cAsE!");
  str = str.toUpperCase();
  assertEquals("MIXED CASE!", str); // str references a new string
}
```

Comparing Strings with `equals`

JUnit's `assertEquals` method uses Java's `equals` method to compare the strings. This is the way to see if two String objects have the same sequence of characters. It is case sensitive.

```
@Test
public void showStringEquals() {
  String s1 = new String("Casey");
  String s2 = new String("Casey");
  String s3 = new String("CaSEy");
  assertTrue(s1.equals(s2));
  assertFalse(s1.equals(s3));
}
```

Avoid using == to compare strings. The results can be surprising.

```
@Test
public void showCompareStringsWithEqualEqual() {
  String s1 = new String("Casey");
  assertTrue(s1 == "Casey");   // This assertion fails.
}
```

The == with objects compares references, not the values of the objects. The above code generates two

different String objects that just happen to have the same state. Use the `equals` method of `String`. The equals method was designed to compare the actual values of the string—the characters, not the reference values.

```
@Test
public void showCompareStringWithEquals() {
  String s1 = "Casey";
  assertTrue(s1.equals("Casey"));    // This assertion passes.
}
```

Self-Check

3-1 Each of the lettered lines has an error. Explain why.

```
BankAccount b1 = new BankAccount("B.  ");          // a
BankAccount b2("The ID", 500.00);                  // b
BankAccount b3 = new Account("N.  Li", 200.00);    // c
b1.deposit();                                      // d
b1.deposit("100.00");                              // e
b1.Deposit(100.00);                                // f
withdraw(100);                                     // g
System.out.println(b4.getID());                    // h
System.out.println(b1.getBalance);                 // I
```

3-2 What values makes these assertions pass (fill in the blanks)?

```
@Test public void testAcct() {
   BankAccount b1 = new BankAccount("Kim", 0.00);
   BankAccount b2 = new BankAccount("Chris", 500.00);
   assertEquals(          , b1.getID());
   assertEquals(          , b2.getID());
   b1.deposit(222.22);
   b1.withdraw(20.00);
   assertEquals(          , b1.getBalance(), 0.001);
   b2.deposit(55.55);
   b2.withdraw(10.00);
   assertEquals(          , b2.getBalance(), 0.001);
  }
}
```

3-3 What value makes this assertion pass?

```
String s1 = new String("abcdefghi");
assertEquals(        , s1.indexOf("g"));
```

3-4 What value makes this assertion pass?

```
String s2 = "abcdefghi";
assertEquals(          , s2.substring(4, 6));
```

3-5 Write an expression to store the middle character of a String into a `char` variable named `mid`. If there is an even number of characters, store the `char` to the right of the middle. For example, the middle character of "abcde" is `'c'` and of "Jude" is `'d'`.

3-6 For each of the following messages, if there is something wrong, write "error"; otherwise, write the value of the expression.

```
String s = new String("Any String");

a. length(s)          d. s.indexOf(" ")
b. s.length           e. s.substring(2, 5)
c. s(length)          f. s.substring("tri")
```

Answers to Self-Checks

3-1 -a Missing the second argument in the object construction. Add the starting balance—a number.
-b Missing = `new BankAccount`.
-c Change `Account` to `BankAccount`.
-d Missing a numeric argument between `(` and `)`.
-e Argument type wrong. pass a number, not a `String`.
-f `Deposit` is not a method of `BankAccount`. Change D to d.
-g Need an object and a dot before `withdraw`.
-h `b4` is not a `BankAccount` object. It was never declared to be anything.
-i Missing `()`.

3-2 a? "Kim"
b? "Chris"
c? 202.22
d? 545.55

3-3 6

3-4 `"ef"`

3-5
```
String aString = "abcde";
int midCharIndex = aString.length( ) / 2;
char mid = aString.charAt( midCharIndex );
```

3-6 -a error -d 3
-b error -e `y S`
-c error -f error (wrong type of argument)