# Chapter 9

# Classes with Instance Variables

## Goals

- Implement Java Classes as a set of methods and variables
- Experience designing and testing a class that is part of a large system

|

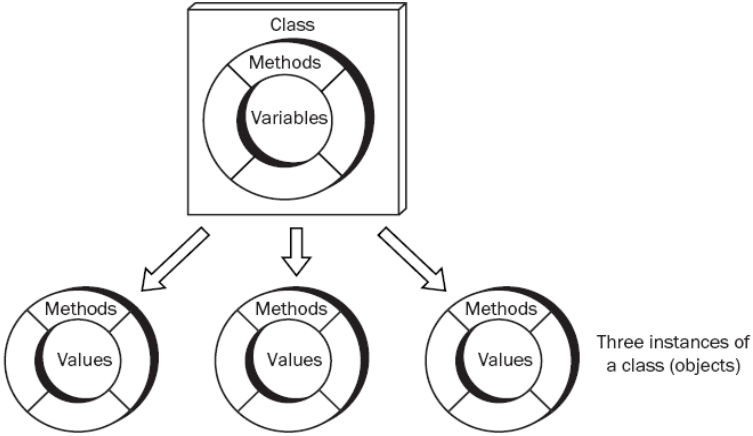
---

## 9.1 Constructing Objects from Classes

Object-oriented programs use objects constructed from many different classes. They may be established Java classes that are part of the download, classes bought from other software developers, classes downloaded for free, or classes designed by programmers to fulfill the needs of a particular application. A class provides a blueprint for

constructing objects, and defines the messages that will be available to instances of each class. The class also defines the values that are encapsulated in every object as the object's state.

**One class constructing three objects, each with its own set of values (state)**




Every Java class has methods that represent the messages each object will understand. Each object of the class has its own set of instance variables to store the values contained in each object. The collection of instance variables is also known as the state of the object.

## Methods and Data Together

All classes have these things in common:

- private instance variables that store the state of the objects
- constructors that initialize the state
- methods to modify the state of objects
- methods to provide access to the current state of objects

Java classes begin with `public class` followed by the class name. The instance variables and methods follow within a set of matching curly braces. The methods and state should have some sort of meaningful connection.

***Simplified General Form:*** **A Java class**

```
public class class-name {

   // Instance variables (every instance of this class will get its own)
   private variable declaration;
   private variable initialization;

   // Constructor(s) (methods with the same name as the class and no return type)
   public class-name (parameters) {
      // ...
   }

   // Any number of methods
   public return-type method-name-1 (parameters) {
      // ...
    }
}
```

Here is a simplified version of the `BankAccount` class. The two instance variables `ID` and `balance` are available to all methods of the class.

```
// This class models a minimal bank account.
public class BankAccount {

  // Instance variables--every BankAccount object will have its own values.
  private String ID;
  private double balance;

  // Initialize instance variables during construction.
  public BankAccount(String initialID, double initialBalance) {
    ID = initialID;
    balance = initialBalance;
  }

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

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

  public String getID() {
    return ID;
  }
  public double getBalance() {
    return balance;
  }
}
```

❶ ❷ ❸ ❹ ❺

With the class stored in a file, it can be used as a blueprint to construct many objects. Each object will have its own `ID` and balance. Each object will understand the `withdraw`, `deposit`, `getID`, and `getBalance` methods. In the

following program, the numbers (❶ for example) indicate which method will execute when the message is sent. For example, ❷ represents transfer of control from the `main` method to the `deposit` method in the `BankAccount` class.

```
@Test
public void testToDemonstrateControlFlow() {
  BankAccount acctOne = new BankAccount("01543C", 100.00);❶
  acctOne.deposit(50.0);                                   ❷
  acctOne.withdraw(25.0);                                  ❸
  assertEquals("01543C", acctOne.getID());                 ❹
  assertEquals(125.0, acctOne.getBalance(), 1e-14);        ❺
}
```

## Instance Variables

In this first example of a type implemented as a Java class, each `BankAccount` object stores data to represent a simple account at a bank. Each `BankAccount` object stores some unique identification ID and an account balance. `BankAccount` methods include making deposits, making withdrawals, and accessing the ID and the current balance.

The private instance variables represent the state. `BankAccount` has two private instance variables: `ID` (a `String`) and `balance` (a `double`). Every `BankAccount` object remembers its own ID and its own current balance.

Notice that the instance variables are not declared within a method. They are declared within the set of curly braces that bounds the class. This means that the instance variables will be accessible throughout the class, and every method will have access to them.

If you look at the `BankAccount` class again, you will notice that every method references at least one of the instance variables. Also, each instance variable is accessed by at least two methods (both the constructor `BankAccount` and `getID` need `ID`).

Because the instance variables are declared `private`, programs using instances of the class cannot access the

instance variables directly. This is good. The class safely encapsulated the state, which was initialized by the constructor (described below). The only way to then change or access the state of an object is through public methods.

## Constructors

The `BankAccount` class shows that all `BankAccount` method headings are public. They also have return types (including **`void`** to mean return nothing). Some have parameters. However, do you notice something different about the method named `BankAccount`?

The `BankAccount` method has no return type. It also has the same name as the class! This special method is known as a **constructor,** because it is the method called when objects are constructed. When a constructor is called, memory is allocated for the object. Then, the instance variables are initialized, often with the arguments to the constructor. Here are some object constructions that result in executing the class's constructor while passing values:

```
new String("An initial part of this object's state");
new BankAccount("Charlie", 10.00);
```

Constructor parameters often initialize the private instance variables. The constructor returns a reference to the new object. This reference value can then be assigned to an object reference of the same type. That is why you often see the class name on both sides of the assignment operator =. For example, the following code constructs a `BankAccount` object with an initial ID of `"Phoenix"` and an initial balance of `507.34`. After the constructor has been called, the reference to this new `BankAccount` object is assigned to the reference variable named `one`.

```
BankAccount one = new BankAccount("Phoenix", 507.34);
```

The following code implements `BankAccount`'s two-parameter constructor:

```
// This constructor initializes the values of the instance variables
// using the arguments use when objects are constructed.
public BankAccount(String accountID, double initialBalance) {
    ID = accountID;
    balance = initialBalance;
}
```

This method executes whenever a `BankAccount` gets constructed with two arguments (a `String` followed by a `double`). For example, in the following code, the ID `"Jessie"` is passed to the parameter `ID`, which in turn is assigned to the private instance variable `ID`. The starting balance of `500.00` is also passed to the parameter named `initialBalance`, which in turn is assigned to the private instance variable `balance`.

```
BankAccount anAccount = new BankAccount("Jessie", 500.00);
```

Some methods provide access to private instance variables. They are sometimes called “getters”, because the method "gets" the value of an instance variable (and they usually begin with `get`). These methods often simply return the value of an instance variable with the `return` statement. Getter methods are necessary because the instance variables are not directly accessible when they are declared `private`.

```
public String getID() {
  return ID;
}

public double getBalance() {
  return balance;
}
```

To get the ID and balance, send the object separate `getID` and `getBalance` messages.

```
@Test
public void showMessagesWayAhead() {
  BankAccount anAccount = new BankAccount("Jessie", 500.00);
  assertEquals("Jessie", anAccount.getID());
  assertEquals(500.00, anAccount.getBalance(), 1e-14);
}
```

The state of an object can change. Some methods are designed to modify the values of the instance variables. Both `deposit` and `withdraw` change the state.

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

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

These two simple test methods assert the changing state of an object.

```
@Test
public void testDepositWithPositiveAmount() {
  BankAccount anAccount = new BankAccount("Jessie", 500.00);
  anAccount.deposit(123.45);
  assertEquals(623.45, anAccount.getBalance(), 1e-14);
}
@Test
public void testWithdrawWithPositiveAmount() {
  BankAccount anAccount = new BankAccount("Jessie", 500.00);
  anAccount.withdraw(123.45);
  assertEquals(376.55, anAccount.getBalance(), 1e-14);
}
```

## *Self-Check*

Use the following `SampleClass` to answer the Self-Check question that follows.

```
// A class that has no meaning other than to show the syntax of a class.
public class SampleClass {

  // Instance variables
  private int first;
  private int second;

  public SampleClass(int initalFirst, int initialSecond) {
    first = initalFirst;
    second = initialSecond;
  }

  public int getFirst() {
    return first;
  }

  public int getSecond() {
    return second;
  }

  public void change(int amount) {
    first = first + amount;
    second = second - amount;
  }
} // End SampleClass
```

9-1 Fill in the blanks that would make the assertions pass.

```
// A unit test to test class SampleClass
import static org.junit.Assert.*;
import org.junit.Test;

public class SampleClassTest {

  @Test
  public void testGetters() {
    SampleClass sc1 = new SampleClass(1, 4);
    SampleClass sc2 = new SampleClass(3, 5);
    assertEquals(_____, sc1.getFirst());
    assertEquals(_____, sc1.getSecond());
    assertEquals(_____, sc2.getFirst());
    assertEquals(_____, sc2.getSecond());
  }

  @Test
  public void testChange() {
    SampleClass sc1 = new SampleClass(1, 4);
    SampleClass sc2 = new SampleClass(3, 5);
    sc1.change(7);
    sc2.change(-3);
    assertEquals(_____, sc1.getFirst());
    assertEquals(_____, sc1.getSecond());
    assertEquals(_____, sc2.getFirst());
    assertEquals(_____,sc2.getSecond());
  }
}
```

Use this Java class to answer the questions that follow.

```
// A class to model a simple library book.
public class LibraryBook {

  // Instance variables
  private String author;
  private String title;
  private String borrower;

  // Construct a LibraryBook object and initialize instance variables
  public LibraryBook(String initTitle, String initAuthor) {
    title = initTitle;
    author = initAuthor;
    borrower = null; // When borrower == null, no one has the book
  }

  // Return the author.
  public String getAuthor() {
    return author;
  }

  // Return the borrower's name if the book has been checked out or null if not
  public String getBorrower() {
    return borrower;
  }

  // Records the borrower's name
  public void borrowBook(String borrowersName) {
    borrower = borrowersName;
  }

  // The book becomes available. When null, no one is borrowing it.
  public void returnBook() {
    borrower = null;
  }
}
```

9-2 What is the name of the type above?

9-3 What is the name of the constructor?

9-4 Except for the constructor, name all of the methods.

9-5 `getBorrower` returns a reference to what type?

9-6 `borrowBook` returns a reference to what type?

9-7 What type argument must be part of all `borrowBook` messages?

9-8 How many arguments are required to construct one `LibraryBook` object?

9-9 Write the code to construct one `LibraryBook` object using your favorite book and author.

9-10 Send the message that borrows your favorite book. Use your own name as the borrower.

9-11 Write the message that reveals the name of the person who borrowed your favorite book (or `null` if no one has borrowed it).

9-12 Which of the following two assertions will pass, a, b, or both?

```
@Test
public void testGetters() {
  LibraryBook book1 = new LibraryBook("C++", "Michael Berman");
  assertEquals(null, book1.getBorrower()); // a.
  book1.borrowBook("Sam Mac");
  assertEquals("Sam Mac", book1.getBorrower()); // b.
}
```

9-13 Write method `getTitle` that returns the title of any `LibraryBook` object.

9-14 Fill in the blanks so the assertions pass.

```
@Test
public void testGetters() {
  LibraryBook book1 = new LibraryBook("C++", "Michael Berman");
  assertEquals(                  , book1.getAuthor());
  assertEquals(                  , book1.getBorrower());
}
```

9-15 Write method `isAvailable` as if it were inside the `LibraryBook` class to return `false` if a `LibraryBook` is not borrowed or true if the borrower is `null`. Use == to compare `null` to an object reference.

9-16 Fill in the blanks in this test method to verify `getTitle` works so all assertions pass.

```
@Test
public void isAvailable() {
    LibraryBook book1 = new LibraryBook("C++", "Berman");
    LibraryBook book2 = new LibraryBook("C#", "Stepp");
    assert[        ](book1.isAvailable());
    assert[        ](book2.isAvailable());

    book1.borrowBook("Sam ");
    book2.borrowBook("Li");
    assert[        ](book1.isAvailable());
    assert[        ](book2.isAvailable());
}
```

## Overriding toString

Each class should have its own `toString` method so the state of the object can be visually inspected. Java is designed such that all classes extend a class named `Object` (or each class extends a class that extends the `Object` class). This means all Java classes inherit the eleven methods of `Object`, one of which is `toString`. Doing nothing to a new class allows `toString` messages to invoke the `toString` method of class `Object`. The return string is the name of the class followed by `@` followed by a code written in hexadecimal (base 16 where 10 is A and 15 is F).

```
LibraryBook book1 = new LibraryBook("C++", "Berman");
System.out.println(book1.toString());
```

**Output**

```
LibraryBook@e4457d
```

To get a more meaningful `toString` that shows the current state of any object, you can override the `toString` method of class `Object` with the same method signature.

```
public String toString() {
  return title + ", borrower: " + borrower;
}
```

With the `toString` method of `Object` overridden to reflect the new type, the output better represents the state of the object.

```
@Test
public void testToString() {
  LibraryBook book1 = new LibraryBook("C++", "Michael A. Berman");
  LibraryBook book2 = new LibraryBook("Java", "Rick Mercer");
  book2.borrowBook("Sam Mac");
  assertEquals("C++, borrower: null", book1.toString());
  assertEquals("Java, borrower: Sam Mac", book2.toString());
}
```

---

## *Self Check*

9-17 Add a `toString` method for the `BankAccount` class to show the ID followed by a blank space and the current balance. You will need the instance variables in `BankAccount`.

```
public class BankAccount {
  private String ID;
  private double balance;

  public BankAccount(String initID, double initBalance) {
    ID = initID;
    balance = initBalance;
  }
  // Add toString as if it were here
}
```

## Naming Conventions

A method that modifies the state of an object is typically given a name that indicates its behavior. This is easily accomplished if the designer of the class provides a descriptive name for the method. The method name should describe—as best as possible—what the method actually does. It should also help to distinguish modifying methods from accessing methods. Use verbs to name modifying methods: `withdraw`, `deposit`, `borrowBook`, and `returnBook`, for example. Give accessing methods names to indicate that the messages will return some useful information about the objects: `getBorrower` and `getBalance`, for example. Above all, always use intention-revealing identifiers to accurately describe what the method does. For example, don't use `foo` as the name of a method that withdraws money.

## **`public`** or **`private`**?

One of the considerations in the design of any class is declaring methods and instance variables with the most appropriate access mode, either `public` or `private`. Whereas programs outside the class can access the public methods of a class, the **`private`** instance variables are only known in the class methods. For example, the `BankAccount` instance variable named `balance` is known only to the methods of the class. On the other hand, any method declared `public` is known wherever the object was declared.

| **Access Mode** | **Where the Identifier Can Be Accessed (where the identifier is visible)** |
|---|---|
| `public` | In all parts of the class and anywhere an instance of the class is declared |
| `private` | Only in the same class |

Although instance variables representing state could be declared as `public`, it is highly recommended that all instance variables be declared as `private`. There are several reasons for this. The consistency helps simplify some design decisions. More importantly, when instance variables are made `private`, the state can be modified only through a method. This prevents other code from indiscriminately changing the state of objects. For example, it is impossible to accidentally make a credit to `acctOne` like this:

```
BankAccount acctOne = new BankAccount("Mine", 100.00);
// A compiletime error occurs: attempting to modify private data
acctOne.balance = acctOne.balance + 100000.00;  // <- ERROR
```

or a debit like this:

```
// A compiletime error occurs at this attempt to modify private data
acctOne.balance = acctOne.balance - 100.00;  // <- ERROR
```

This represents a widely held principle of software development—data should be hidden. Making instance variables `private` is one characteristic of a well-designed class.

---

# Answers to Self-Check

9-1
```
SampleClass sc2 = new SampleClass(3, 5);
    assertEquals(__1__, sc1.getFirst());
    assertEquals(__4__, sc1.getSecond());
    assertEquals(__3__, sc2.getFirst());
    assertEquals(__5__, sc2.getSecond());

    sc2.change(-3);
    assertEquals(__8__, sc1.getFirst());
    assertEquals(_-3__, sc1.getSecond());
    assertEquals(__0__, sc2.getFirst());
    assertEquals(__8__, sc2.getSecond());
```

9-2 type: `LibraryBook`

9-3 constructor: `LibraryBook`

9-4 `LibraryBook` (constructor) `getAuthor getBorrower borrowBook returnBook`

9-5 `String`

9-6 nothing, it is a void return type.

9-7 `String`

9-8 two (both `String`)

9-9 `LibraryBook aBook = new LibraryBook("Computing Fundamentals", "Rick Mercer");`

9-10 `aBook.borrowBook("Kim");`

9-11 `aBook.getBorrower();`

9-12 both a and b pass

9-13
```
public String getTitle() {
  return title;
}
```

9-14
```
@Test
public void testGetters() {
  LibraryBook book1 = new LibraryBook("C++", "Michael Berman");
  assertEquals("Michael Berman", book1.getAuthor());
  assertEquals(null, book1.getBorrower());
}
```

9-15
```
public boolean isAvailable() {
  return borrower == null;
}
```

9-16 Fill in the blanks in this test method to verify `getTitle` works so all assertions pass.

```
assertTrue(book1.isAvailable());
assertTrue (book2.isAvailable());
book1.borrowBook("Sam Mac");
book2.borrowBook("Sam Mac");
assertFalse(book1.isAvailable());
assertFalse(book2.isAvailable());
assertEquals("_Sam _", book1.getBorrower());
assertEquals("_Li__", book2.getBorrower());
```

9-17

```
public String toString() {
  return "" + ID + " " + balance;
}
```