Chapter Three: Implementing Classes
Chapter Goals

• To become familiar with the process of implementing classes
• To be able to implement simple methods
• To understand the purpose and use of constructors
• To understand how to access instance fields and local variables
• To appreciate the importance of documentation comments
Black Boxes

• A black box magically does its thing
• Hides its inner workings
• Encapsulation: the hiding of unimportant details
• What is the right concept for each particular black box?
• Concepts are discovered through abstraction
• Abstraction: taking away inessential features, until only the essence of the concept remains
• In object-oriented programming the black boxes from which a program is manufactured are called objects
Levels of Abstraction: A Real Life Example

- Black boxes in a car: transmission, electronic control module, etc.
Levels of Abstraction: A Real Life Example

• Users of a car do not need to understand how black boxes work

• Interaction of a black box with outside world is well-defined
  • Drivers interact with car using pedals, buttons, etc.
  • Mechanic can test that engine control module sends the right firing signals to the spark plugs
  • For engine control module manufacturers, transistors and capacitors are black boxes magically produced by an electronics component manufacturer

• Encapsulation leads to efficiency:
  • Mechanic deals only with car components (e.g. electronic control module), not with sensors and transistors
  • Driver worries only about interaction with car (e.g. putting gas in the tank), not about motor or electronic control module
Levels of Abstraction: Software Design

**Figure 2**
Levels of Abstraction in Software Design
Levels of abstraction: Software Design

• Old times: computer programs manipulated primitive types such as numbers and characters

• Manipulating too many of these primitive quantities is too much for programmers and leads to errors

• Solution: Encapsulate routine computations to software black boxes

• Abstraction used to invent higher-level data types

• In object-oriented programming, objects are black boxes

• Encapsulation: Programmer using an object knows about its behavior, but not about its internal structure
Levels of abstraction: Software Design (cont.)

• In software design, you can design good and bad abstractions with equal facility; understanding what makes good design is an important part of the education of a software engineer

• First, define behavior of a class; then, implement it
Self Check 3.1

In Chapters 1 and 2, you used `System.out` as a black box to cause output to appear on the screen. Who designed and implemented `System.out`?
Suppose you are working in a company that produces personal finance software. You are asked to design and implement a class for representing bank accounts. Who will be the users of your class?
Specifying the Public Interface of a Class

Behavior of bank account (abstraction):
  • deposit money
  • withdraw money
  • get balance
Specifying the Public Interface of a Class: Methods

Methods of `BankAccount` class:

- deposit
- withdraw
- `getBalance`

We want to support method calls such as the following:

```java
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```
Specifying the Public Interface of a Class: Method Definition

access specifier (such as `public`)
  • return type (such as `String` or `void`)
  • method name (such as `deposit`)
  • list of parameters (`double amount` for `deposit`)
  • method body in `{ }`

Examples:
  • `public void deposit(double amount) { . . . }`
  • `public void withdraw(double amount) { . . . }`
  • `public double getBalance() { . . . }`
Syntax 3.1 Method Definition

accessSpecifier returnType methodName(parameterType parameterName, . . .)
{
    method body
}

Example:

public void deposit(double amount)
{
    . . .
}

Purpose:

To define the behavior of a method.
A constructor initializes the instance fields

Constructor name = class name

```java
public BankAccount()
{
    // body--filled in later
}
```

Constructor body is executed when new object is created

Statements in constructor body will set the internal data of the object that is being constructed

All constructors of a class have the same name

Compiler can tell constructors apart because they take different parameters
accessSpecifier ClassName(parameterType parameterName, . . .)  
{  
    constructor body  
}  

Example:  
public BankAccount(double initialBalance)  
{  
    . . .  
}  

Purpose:  
To define the behavior of a constructor.
The public constructors and methods of a class form the *public interface* of the class.

```java
public class BankAccount {
    // Constructors
    public BankAccount() {
        // body--filled in later
    }
    public BankAccount(double initialBalance) {
        // body--filled in later
    }
}
```
// Methods
public void deposit(double amount) {
    // body--filled in later
}
public void withdraw(double amount) {
    // body--filled in later
}
public double getBalance() {
    // body--filled in later
}
// private fields--filled in later
Syntax 3.3 Class Definition

```
accessSpecifier class ClassName
{
    constructors
    methods
    fields
}
```

Example:

```
public class BankAccount
{
    public BankAccount(double initialBalance) { . . . }
    public void deposit(double amount) { . . . }
    . . .
}
```

Purpose:

To define a class, its public interface, and its implementation details.
/**
 * Withdraws money from the bank account.
 * @param the amount to withdraw
 */
public void withdraw(double amount) {
    //implementation filled in later
}

/**
 * Gets the current balance of the bank account.
 * @return the current balance
 */
public double getBalance() {
    //implementation filled in later
}
Class Comment

/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class BankAccount
{
    //
}

• Provide documentation comments for
  • every class
  • every method
  • every parameter
  • every return value.
Instance Fields

• An object stores its data in instance fields
• Field: a technical term for a storage location inside a block of memory
• Instance of a class: an object of the class
• The class declaration specifies the instance fields

```java
public class BankAccount {
    . . .
    private double balance;
}
```
Instance Fields

• An instance field declaration consists of the following parts:
  • access specifier (usually private)
  • type of variable (such as double)
  • name of variable (such as balance)

• Each object of a class has its own set of instance fields

• You should declare all instance fields as private
Instance Fields

**Figure 5**  Instance Fields
**Syntax 3.4 Instance Field Declaration**

```java
accessSpecifier class ClassName
{
   ...  
   accessSpecifier fieldType fieldName;
   ...
}
```

**Example:**

```java
public class BankAccount
{
   ...  
   private double balance;
   ...
}
```

**Purpose:**

To define a field that is present in every object of a class.
Accessing Instance Fields

• The `deposit` method of the `BankAccount` class can access the private instance field:

```java
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```
Accessing Instance Fields  (cont.)

• Other methods cannot:

```java
public class BankRobber {
    public static void main(String[] args) {
        BankAccount momsSavings = new BankAccount(1000);
        . . .
        momsSavings.balance = -1000; // ERROR
    }
}
```

• *Encapsulation* is the process of hiding object data and providing methods for data access

• To encapsulate data, declare instance fields as `private` and define public methods that access the fields
Implementing Constructors

• Constructors contain instructions to initialize the instance fields of an object

```java
public BankAccount()
{
    balance = 0;
}
public BankAccount(double initialBalance)
{
    balance = initialBalance;
}
```
Constructor Call Example

• BankAccount harrysChecking = new BankAccount(1000);
  • Create a new object of type BankAccount
  • Call the second constructor (since a construction parameter is supplied)
  • Set the parameter variable initialBalance to 1000
  • Set the balance instance field of the newly created object to initialBalance
  • Return an object reference, that is, the memory location of the object, as the value of the new expression
  • Store that object reference in the harrysChecking variable
Implementing Methods

• Some methods do not return a value

```java
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

• Some methods return an output value

```java
public double getBalance()
{
    return balance;
}
```
Method Call Example

• `harrysChecking.deposit(500);`
  • *Set the parameter variable* `amount` *to 500*
  • *Fetch the balance field of the object whose location is stored in* `harrysChecking`
  • *Add the value of amount to balance and store the result in the variable* `newBalance`
  • *Store the value of newBalance in the balance instance field, overwriting the old value*
Syntax 3.5 The return Statement

return expression;
or
return;

Example:

return balance;

Purpose:

To specify the value that a method returns, and exit the method immediately. The return value becomes the value of the method call expression.
/**
  A bank account has a balance that can be changed by deposits and withdrawals.
*/

class BankAccount
{
  /**
   Constructs a bank account with a zero balance.
  */
  public BankAccount()
  {
    balance = 0;
  }

  /**
   Constructs a bank account with a given balance.
   @param initialBalance the initial balance
  */
  public BankAccount(double initialBalance)
  {
    balance = initialBalance;
  }

  Continued
Deposits money into the bank account.
@param amount the amount to deposit
*/
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}

Withdraws money from the bank account.
@param amount the amount to withdraw
*/
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}

Gets the current balance of the bank account.
@return the current balance
*/
public double getBalance()
{
    return balance;
}

private double balance;
Unit Testing

- **Unit test**: verifies that a class works correctly in isolation, outside a complete program.

- To test a class, use an environment for interactive testing, or write a tester class.

- **Test class**: a class with a main method that contains statements to test another class.

- Typically carries out the following steps:
  1. *Construct one or more objects of the class that is being tested*
  2. *Invoke one or more methods*
  3. *Print out one or more results*
Details for building the program vary. In most environments, you need to carry out these steps:

1. Make a new subfolder for your program
2. Make two files, one for each class
3. Compile both files
4. Run the test program
A class to test the BankAccount class.

```java
public class BankAccountTester {
    public static void main(String[] args) {
        BankAccount harrysChecking = new BankAccount();
        harrysChecking.deposit(2000);
        harrysChecking.withdraw(500);
        System.out.println(harrysChecking.getBalance());
        System.out.println("Expected: 1500");
    }
}
```

Output:
1500
Expected: 1500
**Figure 6**
The Return Value of the `getBalance` Method in BlueJ
Categories of Variables

- Categories of variables
  1. Instance fields (*balance* in *BankAccount*)
  2. Local variables (*newBalance* in *deposit* method)
  3. Parameter variables (*amount* in *deposit* method)

- An instance field belongs to an object

- The fields stay alive until no method uses the object any longer

- In Java, the *garbage collector* periodically reclaims objects when they are no longer used

- Local and parameter variables belong to a method

- Instance fields are initialized to a default value, but you must initialize local variables
harrysChecking.deposit(500);
harrysChecking.deposit(500); ¹
harrysChecking.deposit(500); ①
double newBalance = balance + amount; ②
harrysChecking.deposit(500);  
double newBalance = balance + amount;  
balance = newBalance;  

Figure 7  Lifetime of Variables
Self Check 3.13

What do local variables and parameter variables have in common? In which essential aspect do they differ?

**Answer:** Variables of both categories belong to methods – they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the call values; local variables must be explicitly initialized.
Implicit and Explicit Method Parameters

• The implicit parameter of a method is the object on which the method is invoked

• The this reference denotes the implicit parameter

• Use of an instance field name in a method denotes the instance field of the implicit parameter

public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
Implicit and Explicit Method Parameters (cont.)

- `balance` is the balance of the object to the left of the dot:
  
momsSavings.withdraw(500)

  means

  double newBalance = momsSavings.balance - amount;
  >momsSavings.balance = newBalance;
Implicit Parameters and this

• Every method has one implicit parameter
• The implicit parameter is always called this
• Exception: Static methods do not have an implicit parameter (more on Chapter 8)

• double newBalance = balance + amount;
  // actually means
double newBalance = this.balance + amount;

• When you refer to an instance field in a method, the compiler automatically applies it to the this parameter

momsSavings.deposit(500);
Implicit Parameters and `this`

Figure 8  The Implicit Parameter of a Method Call