Implementing Classes

Advanced Programming

ICOM 4015

Lecture 3

Reading: Java Concepts Chapter 3
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?
Black Boxes

- 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.

Figure 1:
Levels of Abstraction in Automobile Design
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
Levels of Abstraction:
A Real-Life Example

• 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
Levels of Abstraction: Software Design

- 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

Continued…
Levels of Abstraction: Software Design

• 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

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?

2. 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?
Answers

1. The programmers who designed and implemented the Java library
2. Other programmers who work on the personal finance application
Designing the Public Interface of a Class

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

• **Methods of** BankAccount **class:**
  
  ```
  deposit
  withdraw
  getBalance
  ```

• **We want to support method calls such as the following:**

  ```
  harrysChecking.deposit(2000);
  harrysChecking.withdraw(500);
  System.out.println(harrysChecking.getBalance());
  ```
Designing 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 `{ }`

*Continued...*
Designing the Public Interface of a Class: Method Definition

Examples

```java
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
Designing the Public Interface of a Class: Constructor Definition

• A constructor initializes the instance variables

• Constructor name = class name

```java
public BankAccount()
{
    // body--filled in later
}
```
Designing the Public Interface of a Class: Constructor Definition

• 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
Syntax 3.2: Constructor Definition

accessSpecifier ClassName(parameterType parameterName, . . .)
{
    constructor body
}

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

Purpose:
To define the behavior of a constructor
BankAccount Public Interface

• 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
    }
    // body--filled in later
}
```
BankAccount Public Interface

```java
{
    // 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
Self Check

1. How can you use the methods of the public interface to empty the harrysChecking bank account?

2. Suppose you want a more powerful bank account abstraction that keeps track of an account number in addition to the balance. How would you change the public interface to accommodate this enhancement?
Answers

1. `harrysChecking.withdraw(harrysChecking.getBalance())`

2. Add an `accountNumber` parameter to the constructors, and add a `getAccountNumber` method. There is no need for a `setAccountNumber` method—the account number never changes after construction.
Commenting on the Public Interface

```java
/**
   * Withdraws money from the bank account.
   * @param amount 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
}
```
• **Provide documentation comments for**
  - every class
  - every method
  - every parameter
  - every return value.

```java
/**
   * A bank account has a balance that can be changed by deposits and withdrawals.
   */
public class BankAccount
{
  . . .
}
```
Figure 3: A Method Summary Generated by javadoc

Fall 2006  Slides adapted from Java Concepts companion slides
Javadoc Method Detail

Figure 4: Method Detail Generated by javadoc
Self Check

1. Why is the following documentation comment questionable?

```java
/**
   * Each account has an account number.
   * @return the account number of this account.
   */
int getAccountNumber()
```
1. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table

```java
/**
   Constructs a new bank account with a given initial balance.
   @param accountNumber the account number for this account
   @param initialBalance the initial balance for this account
 */
```
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 (such as `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

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

Example:
```
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;
}
```

Continued…
Accessing Instance Fields

• Other methods cannot:

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

• Encapsulation = Hiding data and providing access through methods
1. Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance fields?

2. What are the instance fields of the `Rectangle` class?
Answers

1. An instance field

```java
private int accountNumber;
```

needs to be added to the class

2.

```java
private int x;
private int y;
private int width;
private int height;
```
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

```java
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.
 */

public 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
 */
Continued...
```java
19:    public BankAccount(double initialBalance) {
20:        balance = initialBalance;
21:    }
22: 
23:    /**
24:     * Deposits money into the bank account.
25:     * @param amount the amount to deposit
26:     */
27:    public void deposit(double amount) {
28:        double newBalance = balance + amount;
29:        balance = newBalance;
30:    }
31: 
32:    /**
33:     * Withdrawing money from the bank account.
34:     * @param amount the amount to withdraw
35:     */
36:    public void withdraw(double amount) {
37:        double newBalance = balance - amount;
38:        balance = newBalance;
39:    }
```

Continued...
File BankAccount.java

```java
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;
```
Self Check

1. **How is the `getWidth` method of the Rectangle class implemented?**

2. **How is the `translate` method of the Rectangle class implemented?**
1. There is more than one correct answer. One possible implementation is as follows:

```java
public int getWidth()
{
    return width;
}
```

```java
public void translate(int dx, int dy)
{
    int newx = x + dx;
    x = newx;
    int newy = y + dy;
    y = newy;
}
```
Testing a Class with JUnit

- Import Junit TestCase and Assert classes
- Write a testing subclass of TestCase
- Create testing objects in setUp() method
- Cleanup code in tearDown() method
- Write one method for each test
- Run testing class as JUnit from Eclipse
import junit.framework.Assert;
import junit.framework.TestCase;

public class BankAccountTest extends TestCase {
    private BankAccount account1;
    private BankAccount account2;
    protected void setUp(){
        account1 = new BankAccount();
        account2 = new BankAccount(0);
    }
    protected void tearDown(){
        // No cleanup needed
    }
}
public void testConstructors() {
    Assert.assertTrue((0.0 == account1.getBalance())
    Assert.assertTrue(account1.getBalance() ==
                      account2.getBalance());
}

public void testDeposit() {
    account1.deposit(100.00);
    Assert.assertTrue(account1.getBalance() == 100.00);
}

// ... More tests here

// Continued...
Systematic Testing Principles

- Test incrementally
- Test each module independently
- Test from simple to complex modules
- Know what output to expect
- Verify boundary cases
- Verify conservation properties of methods
- Incrementally build a test suite
- Re-run test suite after every code change
Categories of Variables

- Categories of variables
  - Instance fields (`balance` in `BankAccount`)
  - Local variables (`newBalance` in `deposit` method)
  - 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
Categories of Variables

- 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.
Lifetime of Variables

- Objects live until no longer “referred to”
- Local and parameter variables die when method ends
- Instance fields live until object dies
Lifetime of Variables

Figure 7: Lifetime of Variables

Before method call

1

Method called; parameter variable initialized

Continued...
Lifetime of Variables

Figure 7: Lifetime of Variables

Local variable initialized

After method call; parameter and local variables died
Self Check

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

1. 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 target object on which the method is invoked.
- The \texttt{this} reference denotes the implicit parameter.
Implicit and Explicit Method Parameters

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

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

- **balance** is the balance of the target object to the left of the dot:

  ```java
  momsSavings.withdraw(500)
  ```

  **means**

  ```java
  double newBalance = momsSavings.balance - amount;
  momsSavings.balance = newBalance;
  ```
Implicit Parameters and \textbf{this}

- Every method has one implicit parameter
- The implicit parameter is always called \textbf{this}
- Exception: Static methods do not have an implicit parameter (more on Chapter 9)

\begin{verbatim}
double newBalance = balance + amount;
// actually means
double newBalance = \textbf{this}.balance + amount;
\end{verbatim}
• 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
Self Check

1. How many implicit and explicit parameters does the `withdraw` method of the `BankAccount` class have, and what are their names and types?

2. In the `deposit` method, what is the meaning of `this.amount`? Or, if the expression has no meaning, why not?

3. How many implicit and explicit parameters does the main method of the `BankAccountTester` class have, and what are they called?
Answers

1. One implicit parameter, called this, of type BankAccount, and one explicit parameter, called amount, of type double.

2. It is not a legal expression. this is of type BankAccount and the BankAccount class has no field named amount.

3. No implicit parameter—the method is static—and one explicit parameter, called args.
Electronic Voting Machines

Figure 9: Punch Card Ballot
Electronic Voting Machines

Figure 10: Touch Screen Voting Machine