Week 6

Designing Classes

(Chapter 8)

Chapter Goals

© Ivan Stevanovic/iStockphoto.

- To learn how to choose appropriate classes for a given problem
- To understand the concept of cohesion
- To minimize dependencies and side effects
- To learn how to find a data representation for a class
- To understand static methods and variables
- To learn about packages
- To learn about unit testing frameworks

Discovering Classes

- A class represents a single concept from the problem domain.
- Name for a class should be a *noun* that describes concept.
- Concepts from mathematics:

```
Point
Rectangle
Ellipse
```

- Concepts from real life:

```
BankAccount
CashRegister
```

Discovering Classes

- Actors (end in -er, -or) — objects do some kinds of work for you:

  ```
  Scanner
  Random // Better name: RandomNumberGenerator
  ```

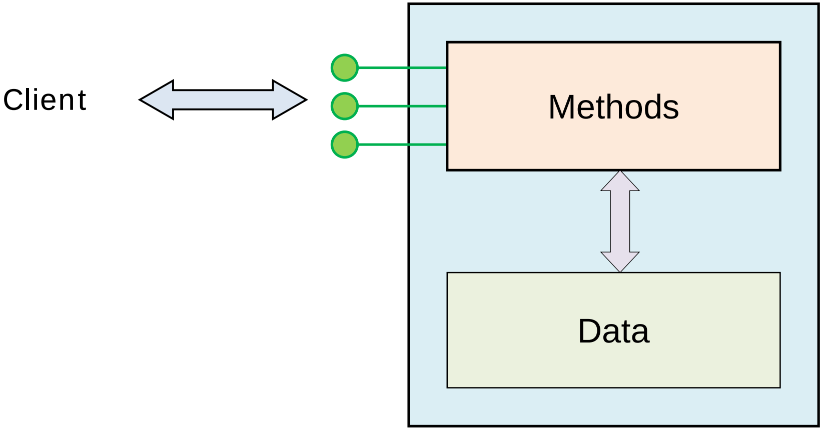
- Utility classes — no objects, only *static* methods and constants:

  ```
  Math
  ```

- Program starters: a class with only a `main` method (*also known as driver*)
- The class name should indicate what objects of the class will do: `Paycheck` is a better name than `PaycheckProgram.`
- Don't turn a single operation action into a class: `Paycheck` is a better name than `ComputePaycheck`.

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client.
- The client invokes the interface methods and they manage the instance data. This is commonly known as API (Application Program Interface)



Visibility Modifiers

Java has three visibility modifiers:

Public

- Members with *public visibility* can be referenced anywhere.

Private

- Members with private visibility can be referenced only *within* that class.

Protected

- The protected modifier involves inheritance, which we will discuss later.

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

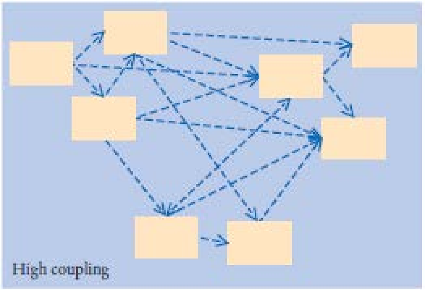
Designing Good Methods -Cohesion/Coupling

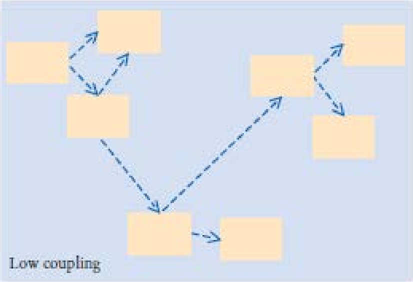
- A class should represent *a single concept*.
- The public interface of a class is cohesive if *all of its features are related to the concept that the class represents*.
- The members of a cohesive team have a common goal.

© Sergey Ivanov/iStockphoto.

Goal:
- High Cohesion
- Low-Coupling

- Coupling: dependency of one class/module on another





Designing Good Methods - Maximize Cohesion

- This class lacks cohesion.

```
public class CashRegister
{
   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
   public static final double NICKEL_VALUE = 0.05;
   . . .
   public void receivePayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   . . .

}
```

- It contains two concepts
  - A cash register that holds coins and computes their total
  - The values of individual coins.

Designing Good Methods – Maximize Cohesion

- Solution: Make two classes:

```
public class Coin
{
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
```

```
public class CashRegister
{
   . . .
   public void receivePayment(int coinCount, Coin coinType)
   {
      payment = payment + coinCount * coinType.getValue();
   }
   . . .
}
```

- Now `CashRegister` class can handle any type of coin.

Minimizing Dependencies - Low Coupling

- A class *depends* on another class if its methods use that class in any way.
  - `CashRegister` depends on `Coin`
- UML: Unified Modeling Language
  - Notation for object-oriented analysis and design

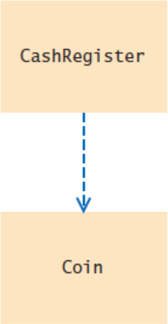
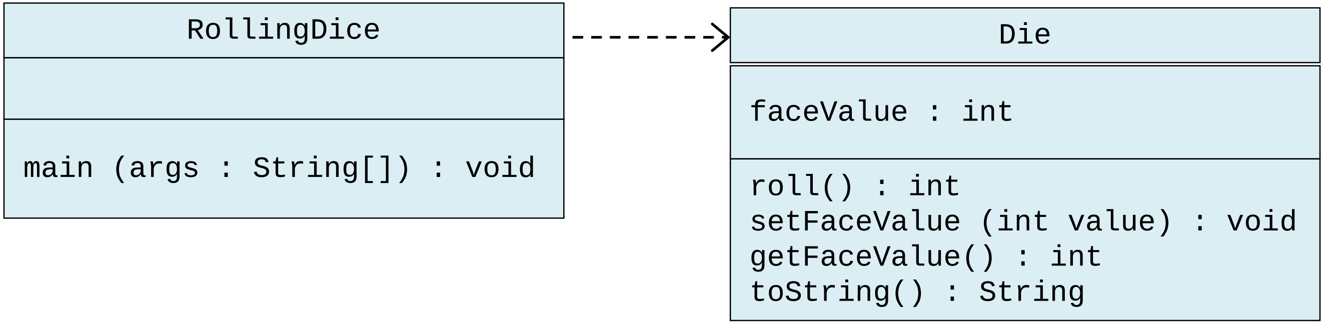


Figure 1 UML class diagram showing dependency relationship between the `CashRegister` and `Coin` Classes.

The `Coin` class does not depend on the `CashRegister` class.



RollingDice

main (args : String[]) : void

Die

faceValue : int

roll() : int
setFaceValue (int value) : void
getFaceValue() : int
toString() : String

Minimizing Dependencies - Low Coupling

- Example: printing `BankAccount` balance
- Recommended

```
System.out.println("The balance is now $" + momsSavings.getBalance());
```

- Don't add a `printBalance` method to `BankAccount`

```
public void printBalance() // Not recommended
{
  System.out.println("The balance is now $" + balance);
}
```

  - The method depends on `System.out`
  - Not every computing environment has `System.out`
  - Violates the rule of minimizing dependencies
- Best to decouple input/output from the work of your classes
  - Place the code for producing output or consuming input in a separate class.

Separating Accessors and Mutators

- A **mutator method** changes the state of an object.
- An **accessor method** asks an object to compute a result, without changing the state.
- An **immutable** class has no *mutator methods*.
- `String` is an immutable class
  - No method in the `String` class can modify the contents of a string.
- References to objects of an immutable class can be safely shared.

Separating Accessors and Mutators

- In a mutable class, separate accessors and mutators
- A method that returns a value *should not be* a mutator.
- In general, all mutators of your class should have return type `void`.
- Sometimes a mutator method can return an informational value.
  - ArrayList `remove` method returns `true` if the removal was successful.
- To check the temperature of the water in the bottle, you could take a sip, but that would be the equivalent of a mutator method.

© manley099/iStockphoto.

Minimizing Side Effects

- A side effect of a method is any externally observable data modification.
- Mutator methods have a side effect, namely the modification of the implicit parameter.
- In general, a method should not modify its parameter variables

```
/**
  Computes the total balance of the given accounts.
  @param accounts a list of bank accounts
*/
  public double getTotalBalance(ArrayList<String> accounts)
  {
    double sum = 0;
    while (studentNames.size() > 0)
    {
      BankAccount account = accounts.remove(0); // Not
      recommended  sum = sum + account.getBalance();
    }
    return sum;
  }
}
```

- Such a side effect would not be what most programmers expect.

Minimizing Side Effects

- The following method mutates the System.out object, which is not a part of the BankAccount object.

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $" + balance);
}
```

- That is a side effect.
- Keep most of your classes free from input and output operations
- This taxi has an undesirable side effect, spraying bystanders with muddy water.

AP Photo/Frank Franklin II.

- When designing methods, minimize side effects.

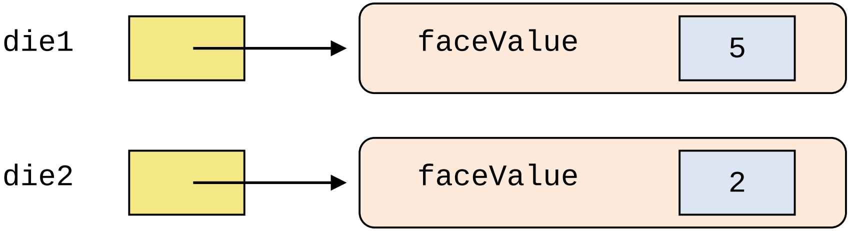
Consistency

- Use a consistent scheme for method names and parameter variables.
- While it is possible to eat with mismatched silverware, consistency is more pleasant.

Frank Rosenstein/Digital Vision/Getty Images, Inc.

Data Scope and Instant data

- The *scope* of data is the area in a program in which that data can be referenced (used).
- Data declared within a method: can be used only in that method (called *local data*)
- A variable declared at the class level is called *instance data.* Each instance (object) has its *own* instance variables and can be referenced by all methods in that class
- A class declares the type of the data, but it does not reserve memory space for it.
- Each time a Die object is created, a new faceValue variable is created as well



Each object maintains its own `faceValue` variable, and thus its own state.

Static Variables and Methods - Variables

- A static variable belongs to the class, not to any object of the class.
- To assign bank account numbers sequentially

  Have a single value of `lastAssignedNumber` that is a property of the class, not any object of the class.
- Declare it using the `static` reserved word

```
public class BankAccount
{
   private double balance;
   private int accountNumber;
   private static int lastAssignedNumber = 1000;

   public BankAccount()
   {
      lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
   . . .
}
```

Static Variables and Methods

- Every BankAccount object has its own balance and accountNumber instance variables
- All objects share a single copy of the lastAssignedNumber variable
  - That variable is stored in a separate location, outside any BankAccount objects
- Static variables should always be declared as private,
  - This ensures that methods of other classes do not change their values
- static constants may be either private or public

```
public class BankAccount
{
   public static final double OVERDRAFT_FEE = 29.95;
   . . .
}
```

- Methods from any class can refer to the constant as

BankAccount.OVERDRAFT_FEE.

Static Variables and Methods

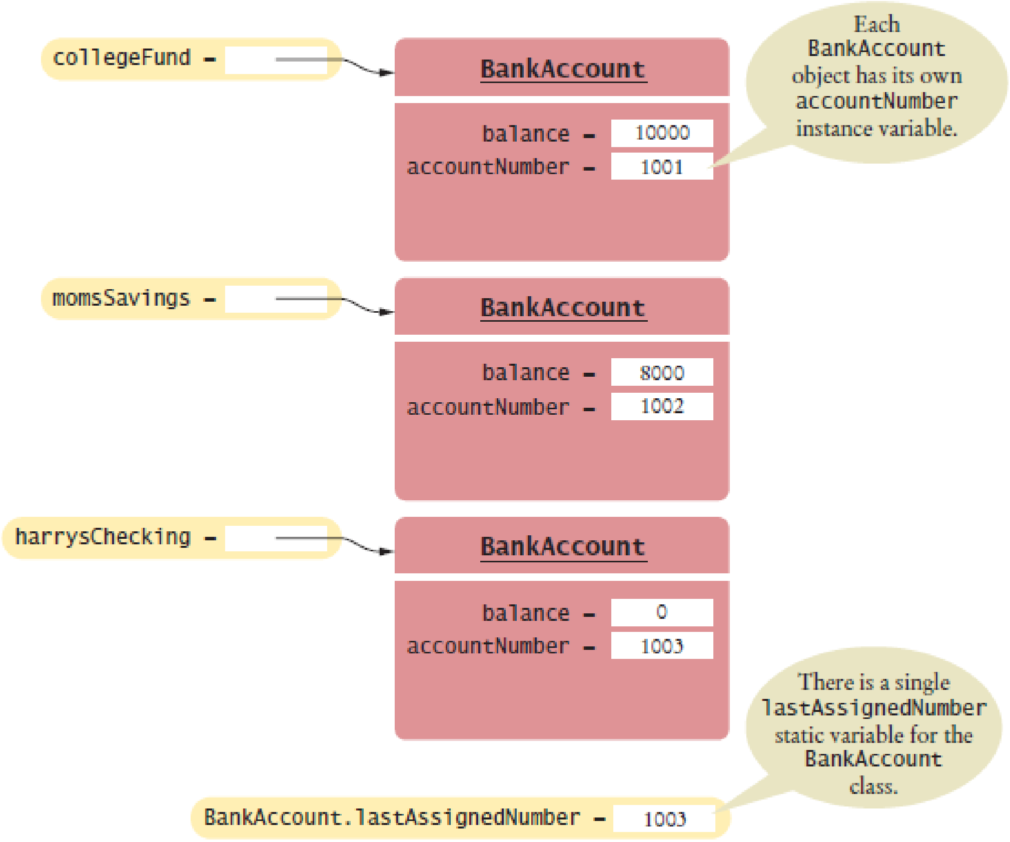

collegeFund
BankAccount
balance 10000
accountNumber 1001
Each BankAccount object has its own accountNumber instance variable.
momsSavings
BankAccount
balance 8000
accountNumber 1002
harrysChecking
BankAccount
balance 0
accountNumber 1003
There is a single lastAssignedNumber static variable for the BankAccount class.
BankAccount.lastAssignedNumber 1003

Figure 5 A Static Variable and Instance Variables

Static Variables and Methods - Methods

- Sometimes a class defines methods that are not invoked on an object

  Called a **static method**

- Example: `sqrt` method of `Math` class

  if `x` is a number, then the call `x.sqrt()` is not legal

  `Math` class provides a static method: invoked as `Math.sqrt(x)`

  No object of the `Math` class is constructed.

  The `Math` qualifier simply tells the compiler where to find the `sqrt` method.

Static Variables and Methods

- You can define your own static methods:

```
public class Financial
{
  /**
    Computes a percentage of an amount.
    @param percentage the percentage to apply
    @param amount the amount to which the percentage is applied
    @return the requested percentage of the amount
  */
  public static double percentOf(double percentage, double amount)
  {
    return (percentage / 100) * amount;
  }
}
```

- When calling such a method, supply the name of the class containing it:

```
double tax = Financial.percentOf(taxRate, total);
```

- The `main` method is always `static`.
  - When the program starts, there aren't any objects.
  - Therefore, the first method of a program must be a static method.
- Programming Tip: Minimize the Use of Static Methods

Packages

- **Package:** Set of related classes

  Important packages in the Java library:

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database Access	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`omg.w3c.dom`	Document Object Model for XML documents	`Document`

Organizing Related Classes into Packages

© Don Wilkie/iStockphoto.

In Java, related classes are grouped into packages.

Organizing Related Classes into Packages

- To put classes in a package, you must place a line

```
package packageName;
```

as the first instruction in the source file containing the classes.

- Package name consists of one or more identifiers separated by periods.
- To put the `Financial` class into a package named `com.horstmann.bigjava`, the `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
   . . .
}
```

- A special package: default package

  Has no name

  No `package` statement

- If you did not include any package statement at the top of your source file

  its classes are placed in the default package.

Importing Packages

- Can use a class *without* importing: refer to it by its full name (package name plus class name):

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

inconvenient

- `import` directive lets you refer to a class of a package by its class name, without the package prefix:

```
import java.util.Scanner;
```

- Now you can refer to the class as `Scanner` without the package prefix.
- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import `java.lang`.
- You don't need to import other classes in the same package.

Package Names

- Use packages to avoid name clashes:

```
java.util.Timer
```

vs.

```
javax.swing.Timer
```

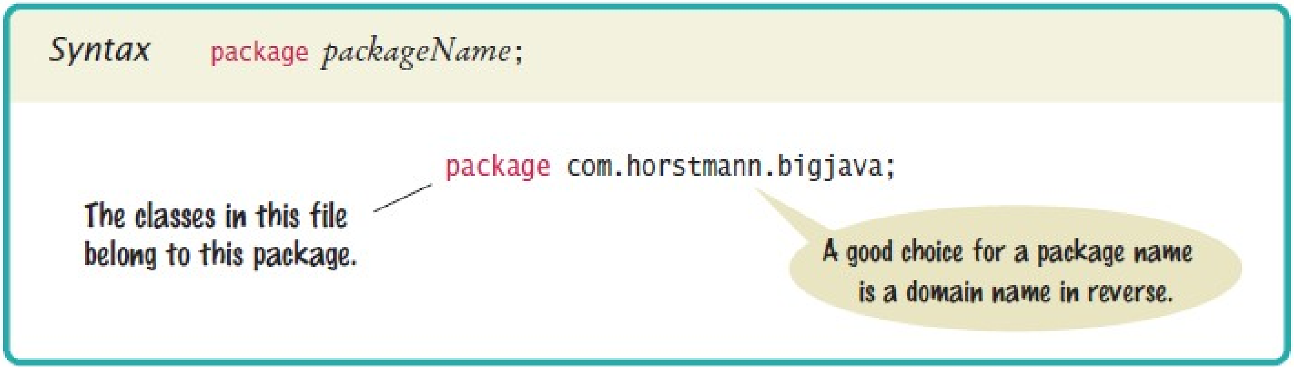
- Package names should be unique.
- To get a package name: turn the domain name around:

```
com.horstmann.bigjava
```

- Or write your email address backwards:

```
edu.sjsu.cs.walters
```

Syntax 8.1 Package Specification



Syntax `package packageName;`

```
package com.horstmann.bigjava;
```

The classes in this file belong to this package.

A good choice for a package name is a domain name in reverse.

Packages and Source Files

- The path of a class file must match its package name.
- The parts of the name between periods represent successively nested directories.
- **Base directory:** holds your program's files
- Place the subdirectory inside the base directory.
- If your homework assignment is in a directory

`/home/britney/hw8/problem1`

  - Place the class files for the `com.horstmann.bigjava` package into the directory:

o `/home/britney/hw8/problem1/com/horstmann/bigjava` (UNIX)

Or

o `c:\Users\Britney\hw8\problem1\com\horstmann\bigjava` (Windows)

Packages and Source Files

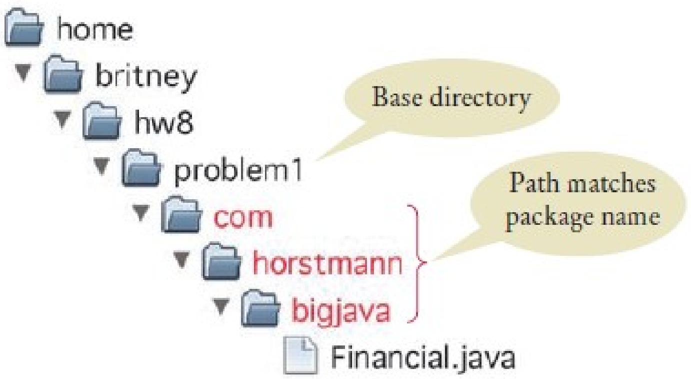


Figure 6 Base Directories and Subdirectories for Packages

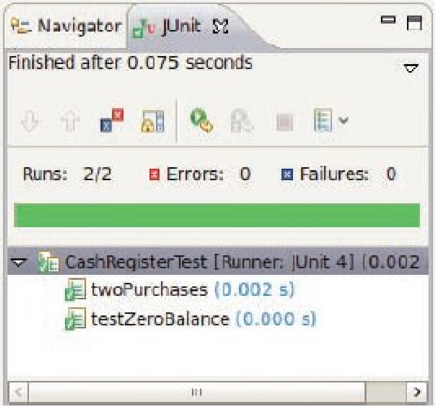
Unit Test Frameworks

- Unit test frameworks simplify the task of writing classes that contain many test cases.
- JUnit: http://junit.org
  - Built into some IDEs like BlueJ and Eclipse
- Philosophy: whenever you implement a class, also make a companion test class. Run all tests whenever you change your code.

Unit Test Frameworks

- Customary that name of the test class ends in `Test`:

```
import org.junit.Test;
import org.junit.Assert;
public class CashRegisterTest
{
  @Test public void twoPurchases()
  {
    CashRegister register = new CashRegister();
    register.recordPurchase(0.75);
    register.recordPurchase(1.50);
    register.enterPayment(2, 0, 5, 0, 0);
    double expected = 0.25;
    Assert.assertEquals(expected, register.giveChange(), EPSILON);
  }
  // More test cases
  . . .
}
```



```
Finished after 0.075 seconds
Runs:  2/2      Errors:  0      Failures:  0
CashRegisterTest [Runner: JUnit 4] (0.002
  twoPurchases (0.002 s)
  testZeroBalance (0.000 s)
```

Figure 7 Unit Testing with JUnit

- If all test cases pass, the JUnit tool shows a green bar:

View to watch https://www.youtube.com/watch?v=I8XXfgF9GSc