

Chapter 8 – Designing Classes

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Chapter Goals

- To learn how to discover appropriate classes for a given problem
- To understand the concepts of cohesion and coupling
- To minimize the use of side effects
- To document the responsibilities of methods and their callers with preconditions and postconditions
- To understand static methods and variables
- To understand the scope rules for local variables and instance variables
- To learn about packages
- T** 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: only have a `main` method
- Don't turn actions into classes

- *Paycheck is a better name than ComputePaycheck*

Self Check 8.1

What is the rule of thumb for finding classes?

Self Check 8.2

Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?

Cohesion

- 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
- This class lacks cohesion:

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

Cohesion

- `CashRegister`, as described above, involves two concepts:
cash register and *coin*
- Solution: Make two classes:

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

```
public class CashRegister
{
    public void enterPayment(int coinCount, Coin coinType)
    { ... }
    ...
}
```


Coupling

- A class *depends* on another if it uses objects of that class
- `CashRegister` depends on `Coin` to determine the value of the payment
- `Coin` does not depend on `CashRegister`
- High coupling = Many class dependencies
- Minimize coupling to minimize the impact of interface changes
- To visualize relationships draw class diagrams
- UML: Unified Modeling Language
 - *Notation for object-oriented analysis and design*

Dependency

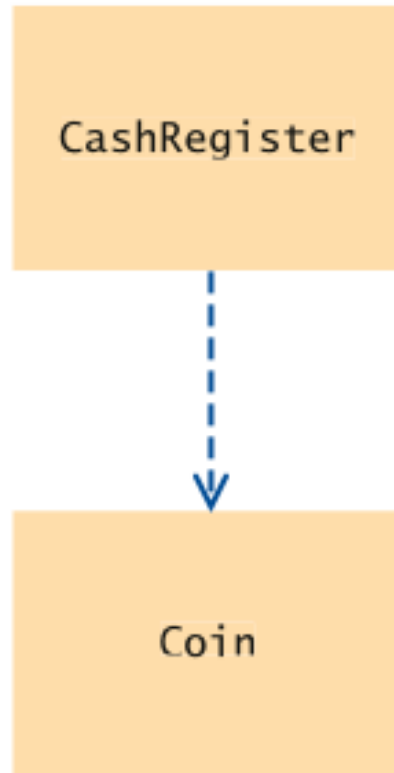


Figure 1
Dependency Relationship
Between the CashRegister
and Coin Classes

High and Low Coupling Between Classes

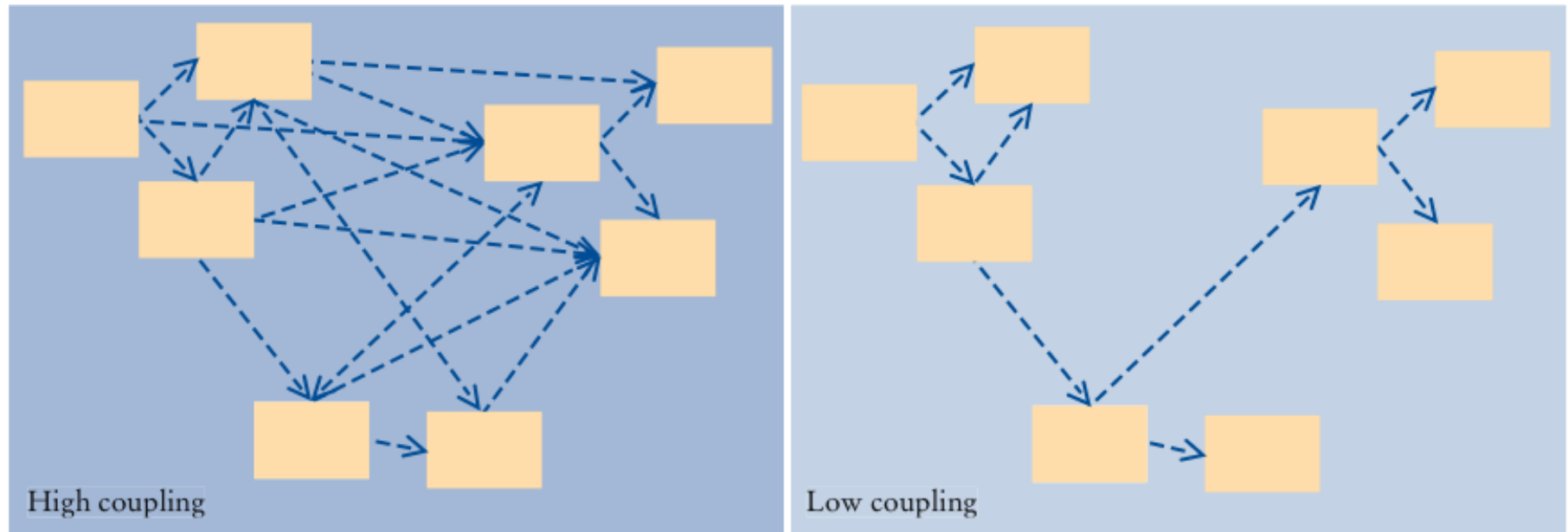


Figure 2 High and Low Coupling Between Classes

Self Check 8.3

Why is the `CashRegister` class from Chapter 4 not cohesive?

.

Self Check 8.4

Why does the `Coin` class not depend on the `CashRegister` class?

Self Check 8.5

Why should coupling be minimized between classes?

Immutable Classes

- **Accessor:** Does not change the state of the implicit parameter:

```
double balance = account.getBalance();
```

- **Mutator:** Modifies the object on which it is invoked:

```
account.deposit(1000);
```

- **Immutable class:** Has no mutator methods (e.g., `String`):

```
String name = "John Q. Public";  
String uppercased = name.toUpperCase();  
// name is not changed
```

- It is safe to give out references to objects of immutable classes; no code can modify the object at an unexpected time

Self Check 8.6

Is the `substring` method of the `String` class an accessor or a mutator?

Self Check 8.7

Is the `Rectangle` class immutable?

Side Effects

- **Side effect of a method:** Any externally observable data modification:

```
harrysChecking.deposit(1000);
```

- Modifying explicit parameter can be surprising to programmers — avoid it if possible:

```
public void addStudents(ArrayList<String> studentNames)
{
    while (studentNames.size() > 0)
    {
        String name = studentNames.remove(0);
        // Not recommended
        . . .
    }
}
```

Side Effects

- This method has the expected side effect of modifying the implicit parameter and the explicit parameter `other`:

```
public void transfer(double amount, BankAccount other
{
    balance = balance - amount;
    other.balance = other.balance + amount;
}
```

Side Effects

- Another example of a side effect is output:

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

Bad idea: Message is in English, and relies on `System.out`

- Decouple input/output from the actual work of your classes
- Minimize side effects that go beyond modification of the implicit parameter

Self Check 8.8

If `a` refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?

Self Check 8.9

Consider the `DataSet` class of Chapter 6. Suppose we add a method

```
void read(Scanner in)
{
    while (in.hasNextDouble())
        add(in.nextDouble());
}
```

Does this method have a side effect other than mutating the data set?

Common Error: Trying to Modify Primitive Type Parameters

- ```
void transfer(double amount, double otherBalance)
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
}
```

- Won't work

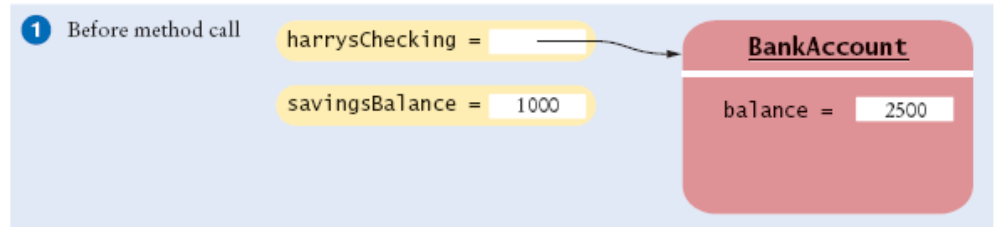
- Scenario:

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
System.out.println(savingsBalance);
```

- In Java, a method can never change parameters of primitive type

# Common Error: Trying to Modify Primitive Type Parameters

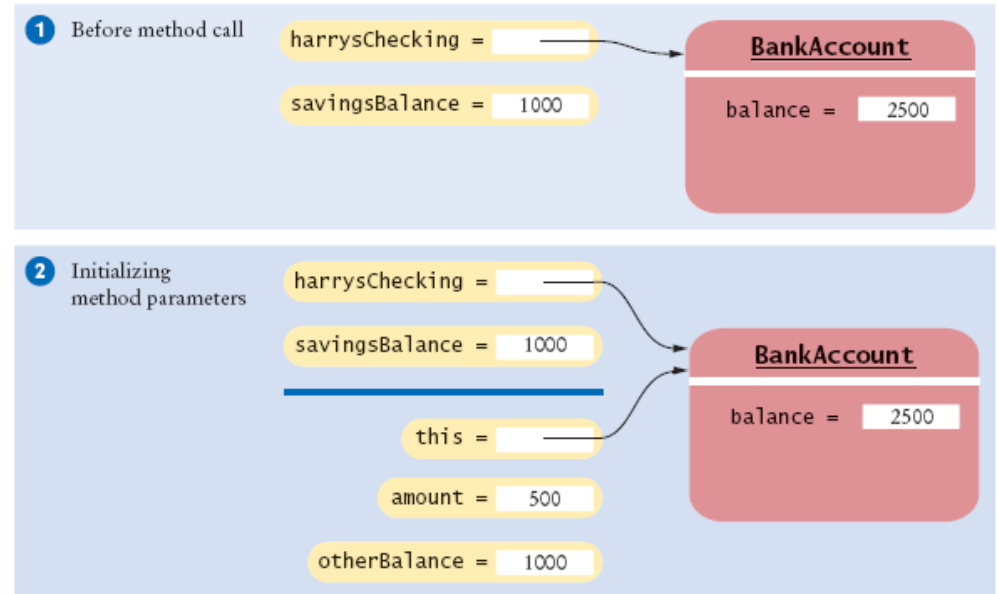
```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance)
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
}
```





# Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) ❷
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
}
```



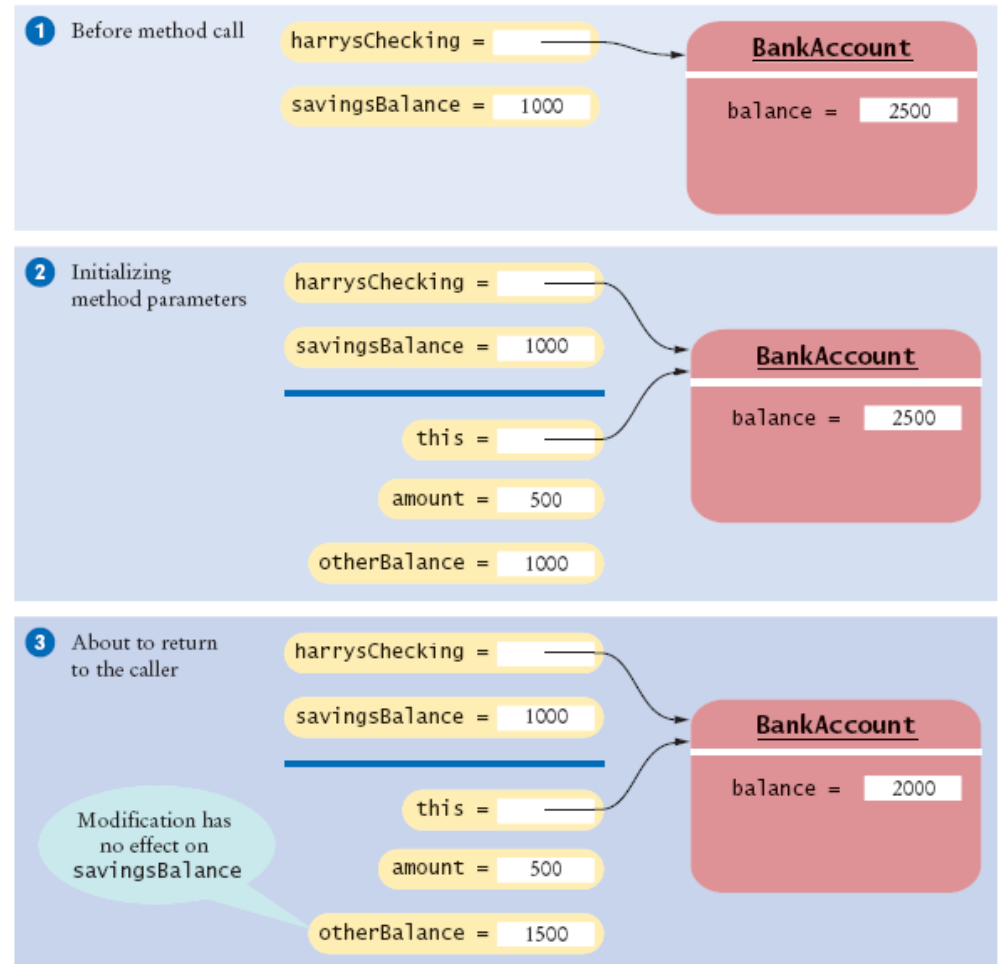
# Common Error: Trying to Modify Primitive Type Parameters

---

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) ❷
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
} ❸
```

***Continued***

# Common Error: Trying to Modify Primitive Type Parameters



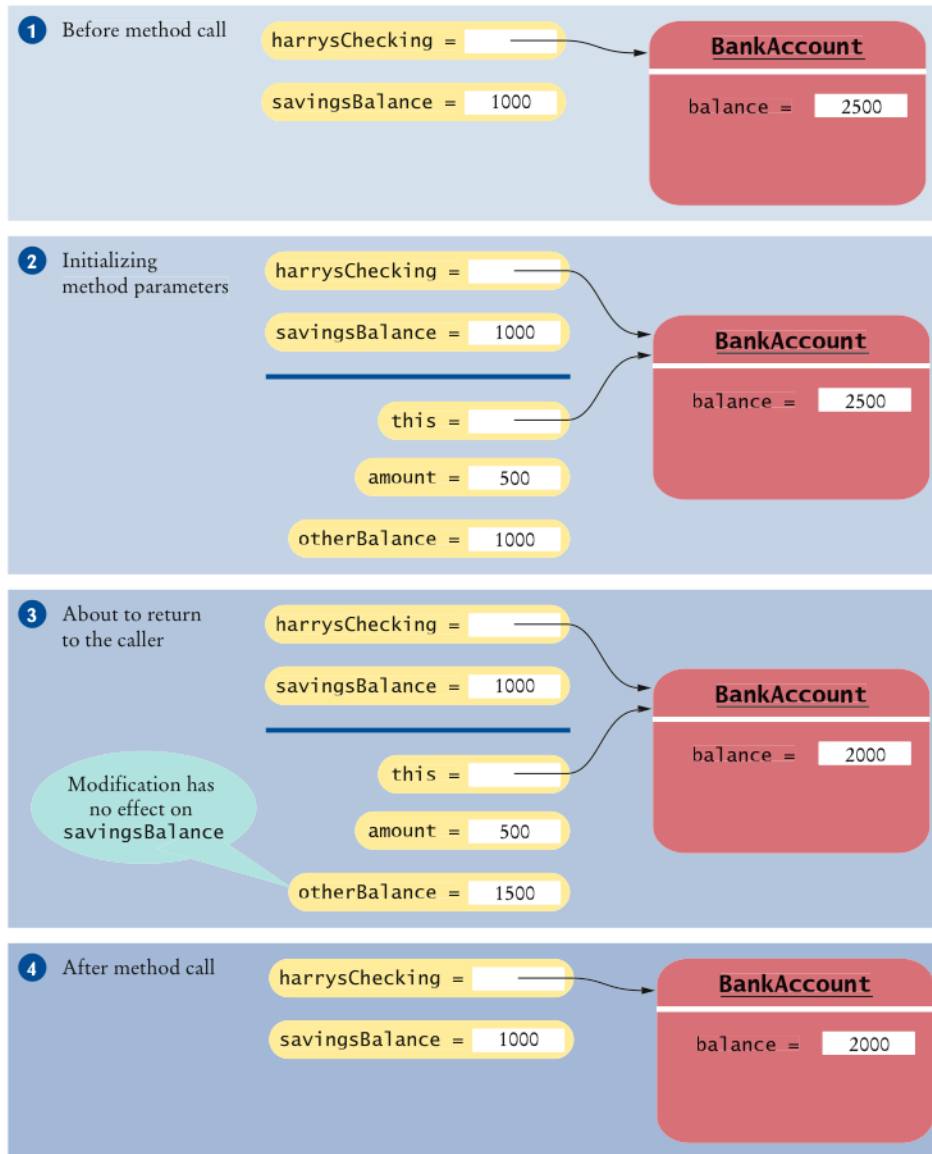
# Common Error: Trying to Modify Primitive Type Parameters

---

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance); ❷
...
void transfer(double amount, double otherBalance) ❸
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
} ❹
```

***Continued***

# Common Error: Trying to Modify Primitive Type Parameters



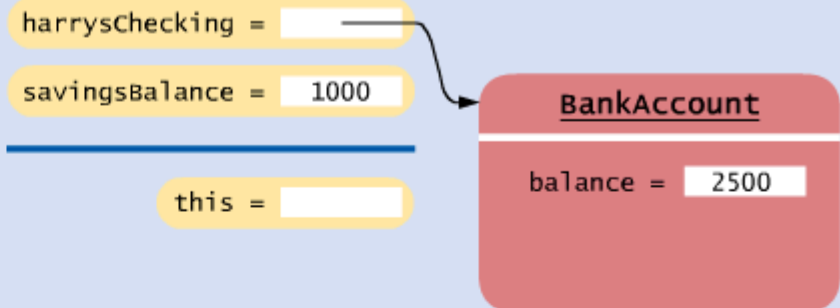
**Figure 3** Modifying a Numeric Parameter Has No Effect on Caller

# Animation 8.1 – A Method Cannot Modify a Numeric Parameter

```
public static void main(String[] args)
{
 BankAccount harrysChecking = new BankAccount(2500);
 double savingsBalance = 1000;
 harrysChecking.transfer(500, savingsBalance);
 System.out.println(savingsBalance);
}

...

public void transfer(double amount, double otherBalance)
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
 // Won't work
}
```



The implicit parameter variable `this` is created and initialized.

A Method Cannot Modify a Numeric Parameter



# Call by Value and Call by Reference

---

- **Call by value:** Method parameters are copied into the parameter variables when a method starts
- **Call by reference:** Methods can modify parameters
- Java has call by value
- A method can change state of object reference parameters, but cannot replace an object reference with another

# Call by Value and Call by Reference

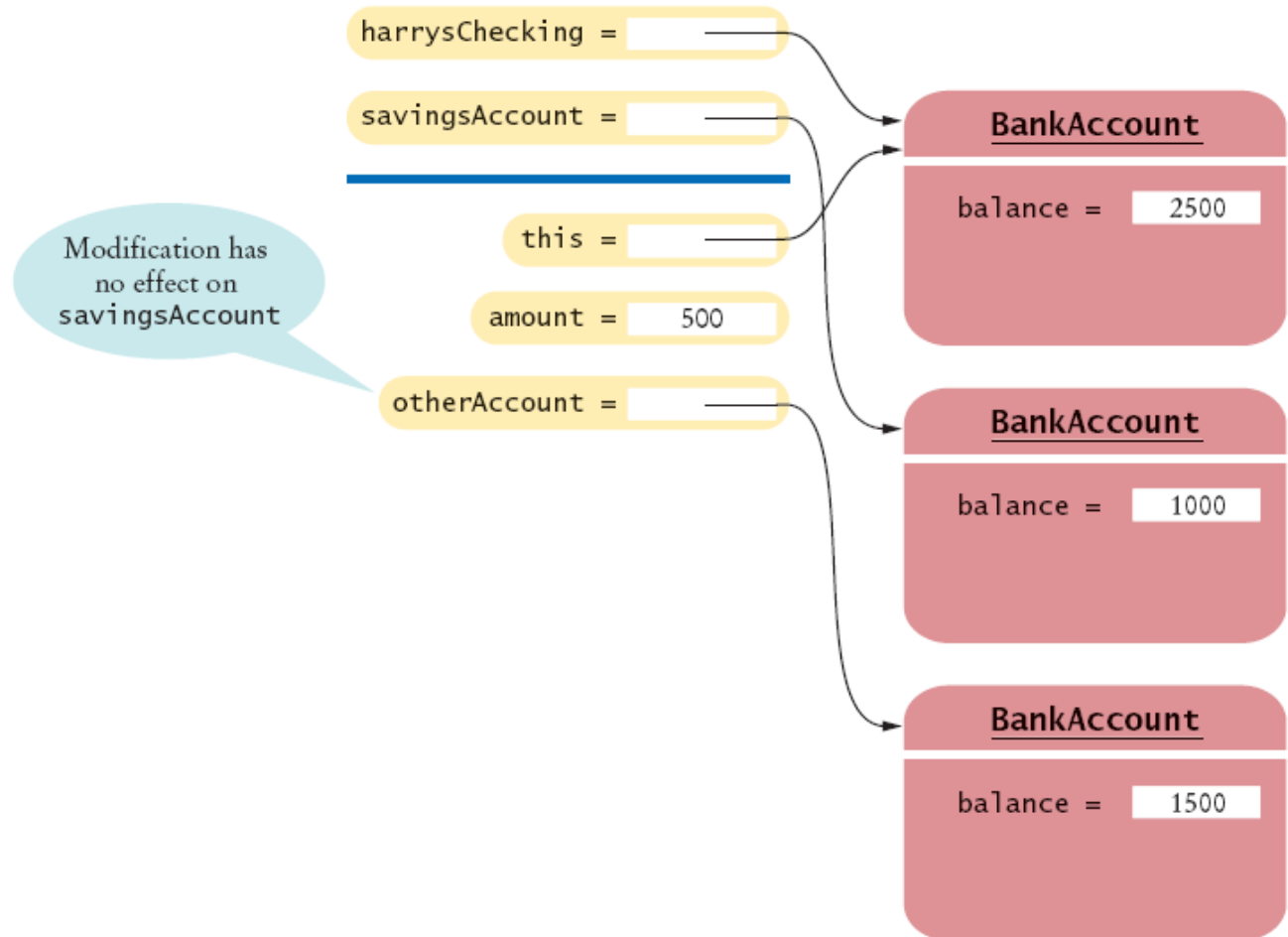
---

```
public class BankAccount
{
 public void transfer(double amount, BankAccount
 otherAccount)
 {
 balance = balance - amount;
 double newBalance = otherAccount.balance + amount;
 otherAccount = new BankAccount(newBalance);
 // Won't work
 }
}
```



# Call by Value Example

```
harrysChecking.transfer(500, savingsAccount);
```



Modifying an Object Reference Parameter Has No Effect on the Caller

# Preconditions

---

- **Precondition:** Requirement that the caller of a method must meet
- Publish preconditions so the caller won't call methods with bad parameters:
- ```
/**  
    Deposits money into this account.  
    @param amount the amount of money to deposit  
    (Precondition: amount >= 0)  
*/
```
- Typical use:
 1. *To restrict the parameters of a method*
 2. *To require that a method is only called when the object is in an appropriate state*

Preconditions

- If precondition is violated, method is not responsible for computing the correct result. It is free to do *anything*
- Method may throw exception if precondition violated — more in Chapter 11:

```
if (amount < 0) throw new IllegalArgumentException();  
balance = balance + amount;
```

- Method doesn't have to test for precondition. (Test may be costly):

```
// if this makes the balance negative, it's the  
// caller's fault  
balance = balance + amount;
```

Preconditions

- Method can do an assertion check:

```
assert amount >= 0;  
balance = balance + amount;
```

To enable assertion checking:

```
java -enableassertions MainClass
```

You can turn assertions off after you have tested your program, so that it runs at maximum speed

- Many beginning programmers silently return to the caller

```
if (amount < 0)  
    return; // Not recommended; hard to debug  
balance = balance + amount;
```

Syntax 8.1 Assertion

Syntax `assert condition;`

Example

`assert amount >= 0;`

If the condition is false
and assertion checking is enabled,
an exception occurs.

Condition that is claimed to be true.

Postconditions

- **Postcondition:** requirement that is true after a method has completed
- If method call is in accordance with preconditions, it must ensure that postconditions are valid
- There are two kinds of postconditions:
 - *The return value is computed correctly*
 - *The object is in a certain state after the method call is completed*

- `/**`

```
    Deposits money into this account.
```

```
    (Postcondition: getBalance() >= 0)
```

```
    @param amount the amount of money to deposit
```

```
    (Precondition: amount >= 0)
```

- `*/`

Postconditions

- Don't document trivial postconditions that repeat the `@return` clause
- Formulate pre- and postconditions only in terms of the interface of the class:

```
amount <= getBalance() // this is the way to state a
    postcondition
amount <= balance // wrong postcondition formulation
```

- Contract: If caller fulfills preconditions, method must fulfill postconditions

Self Check 8.10

Why might you want to add a precondition to a method that you provide for other programmers?

Self Check 8.1 1

When you implement a method with a precondition and you notice that the caller did not fulfill the precondition, do you have to notify the caller?

Static Methods

- Every method must be in a class
- A static method is not invoked on an object
- Why write a method that does not operate on an object
- Common reason: encapsulate some computation that involves only numbers.
 - *Numbers aren't objects, you can't invoke methods on them. E.g. `x.sqrt()` can never be legal in Java*

Static Methods

- Example:

```
public class Financial
{
    public static double percentOf(double p, double a)
    {
        return (p / 100) * a;
    }
    // More financial methods can be added here.
}
```

- Call with class name instead of object:

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

Static Methods

- If a method manipulates a class that you do not own, you cannot add it to that class
- A static method solves this problem:

```
public class Geometry
{
    public static double area(Rectangle rect)
    {
        return rect.getWidth() * rect.getHeight();
    }
    // More geometry methods can be added here.
}
```

- `main` is static — there aren't any objects yet

Self Check 8.12

Suppose Java had no static methods. How would you use the `Math.sqrt` method for computing the square root of a number x ?

Self Check 8.13

The following method computes the average of an array list of numbers:

```
public static double average(ArrayList<Double> values)
```

Why must it be a static method?

Static Variables

- A static variable belongs to the class, not to any object of the class:

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

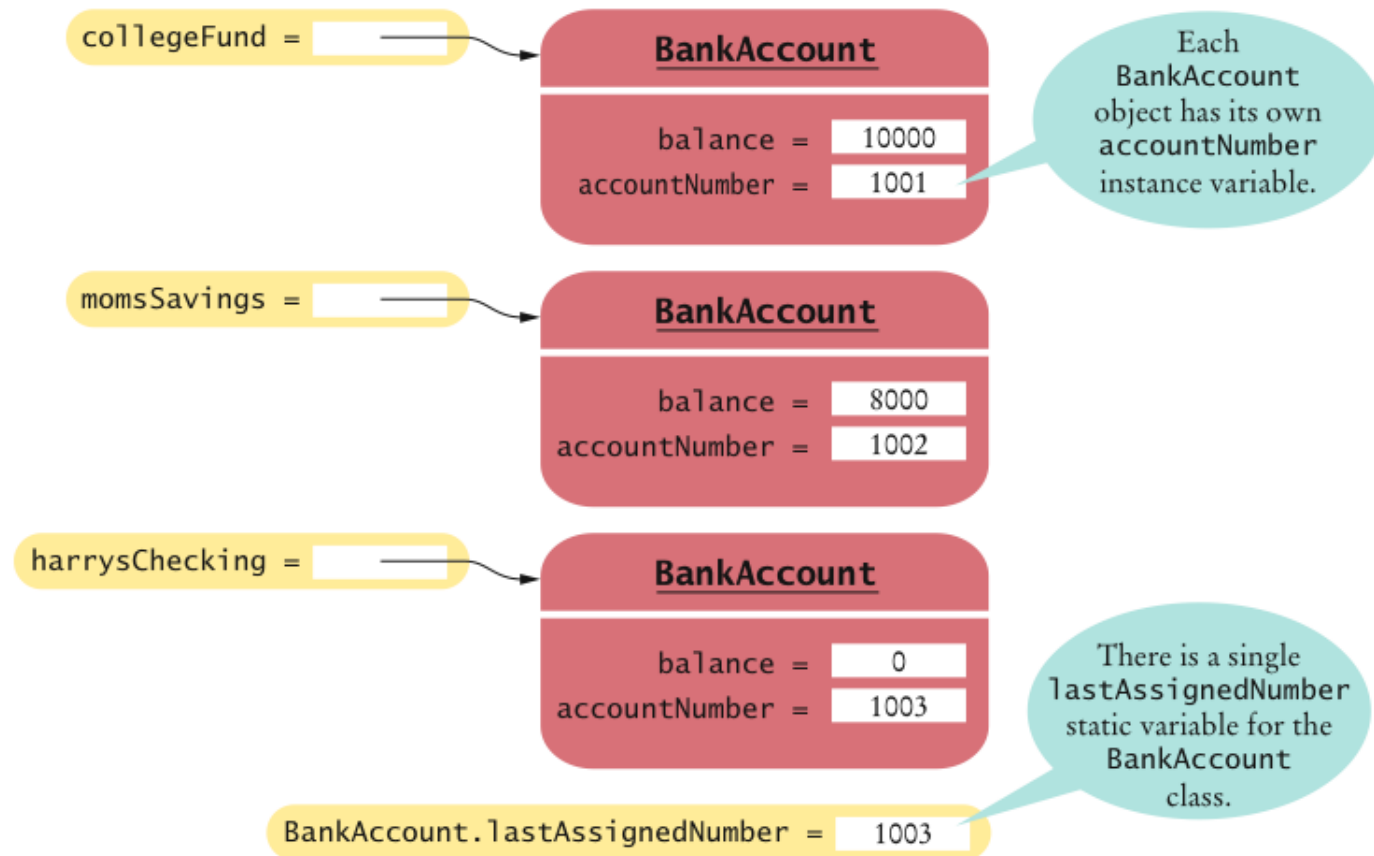
- **If** `lastAssignedNumber` **was not** `static`, each instance of `BankAccount` **would have its own value of** `lastAssignedNumber`

Static Variables

- ```
public BankAccount()
{
 // Generates next account number to be assigned
 lastAssignedNumber++; // Updates the static variable
 accountNumber = lastAssignedNumber;
 // Sets the instance variable
}
```



# A Static Variable and Instance Variables



**Figure 4**  
A Static Variable  
and Instance  
Variables

# Static Variables

---

- Three ways to initialize:
  1. *Do nothing. variable is initialized with 0 (for numbers), false (for boolean values), or null (for objects)*
  2. *Use an explicit initializer, such as*

```
public class BankAccount
{
 ...
 private static int lastAssignedNumber = 1000;
 // Executed once,
}
```
  3. *Use a static initialization block*
- Static variables should always be declared as `private`

# Static Variables

---

- Exception: Static constants, which may be either private or public:

```
public class BankAccount
{
 ...
 public static final double OVERDRAFT_FEE = 5;
 // Refer to it as BankAccount.OVERDRAFT_FEE
}
```

- Minimize the use of static variables (static final variables are ok)

## Self Check 8.14

---

Name two static variables of the `System` class.

## Self Check 8.15

---

Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and variables `static`. Then `main` can call the other static methods, and all of them can access the static variables. Will Harry's plan work? Is it a good idea?

# Scope of Local Variables

---

- **Scope of variable:** Region of program in which the variable can be accessed
- Scope of a local variable extends from its declaration to end of the block that encloses it

# Scope of Local Variables

---

- Sometimes the same variable name is used in two methods:

```
public class RectangleTester
{
 public static double area(Rectangle rect)
 {
 double r = rect.getWidth() * rect.getHeight();
 return r;
 }
 public static void main(String[] args)
 {
 Rectangle r = new Rectangle(5, 10, 20, 30);
 double a = area(r);
 System.out.println(r);
 }
}
```

- These variables are independent from each other; their scopes are disjoint

# Scope of Local Variables

---

- Scope of a local variable cannot contain the definition of another variable with the same name:

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
 double r = Math.sqrt(x);
 // Error - can't declare another variable
 // called r here
 ...
}
```



# Scope of Local Variables

---

- However, can have local variables with identical names if scopes do not overlap:

```
if (x >= 0)
{
 double r = Math.sqrt(x);
 ...
} // Scope of r ends here
else
{
 Rectangle r = new Rectangle(5, 10, 20, 30);
 // OK - it is legal to declare another r here
 ...
}
```

# Overlapping Scope

---

- A local variable can *shadow* a variable with the same name
- Local scope wins over class scope:

```
public class Coin
{
 ...
 public double getExchangeValue(double exchangeRate)
 {
 double value; // Local variable
 ...
 return value;
 }
 private String name;
 private double value; // variable with the same name
}
```

# Overlapping Scope

---

- Access shadowed variables by qualifying them with the `this` reference:

```
value = this.value * exchangeRate;
```

# Overlapping Scope

---

- Generally, shadowing an instance variable is poor code — error-prone, hard to read
- Exception: when implementing constructors or setter methods, it can be awkward to come up with different names for instance variables and parameters
- OK:

```
public Coin(double value, String name)
{
 this.value = value;
 this.name = name;
}
```

## Self Check 8.16

---

Consider the following program that uses two variables named `r`. Is this legal?

```
public class RectangleTester
{
 public static double area(Rectangle rect)
 {
 double r = rect.getWidth() * rect.getHeight();
 return r;
 }
 public static void main(String[] args)
 {
 Rectangle r = new Rectangle(5, 10, 20, 30);
 double a = area(r);
 System.out.println(r);
 }
}
```

## Self Check 8.17

---

What is the scope of the `balance` variable of the `BankAccount` class?

# Packages

---

- **Package:** Set of related classes
- Important packages in the Java library:

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

# 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



# Organizing Related Classes into Packages

---

- For example, to put the `Financial` class introduced into a package named `com.horstmann.bigjava`, the `Financial.java` file must start as follows:

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

```
public class Financial
{
 ...
}
```

- Default package has no name, no `package` statement

## Syntax 8.2 Package Specification

*Syntax*    `package` *packageName*;

*Example*

The classes in this file  
belong to this package.

`package` com.horstmann.bigjava;

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

# Importing Packages

---

- Can always use class without importing:

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

- Tedious to use fully qualified name
- Import lets you use shorter class name:

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

- 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 unambiguous
- Recommendation: start with reversed domain name:

`com.horstmann.bigjava`

- `edu.sjsu.cs.walters`: for Britney Walters' classes  
(`walters@cs.sjsu.edu`)

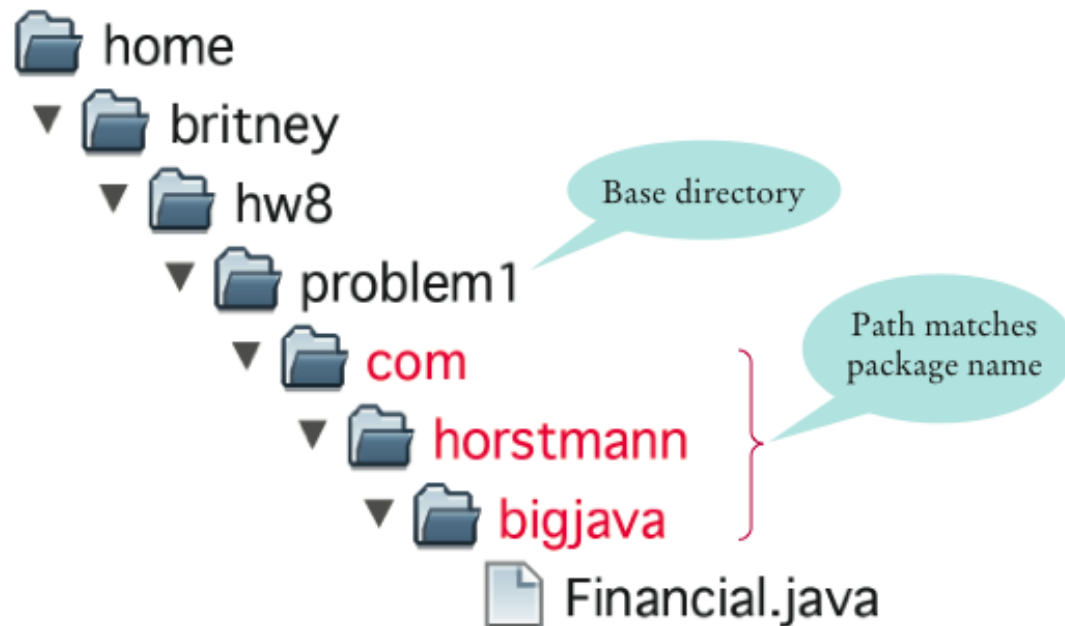
- Path name should match package name:

`com/horstmann/bigjava/Financial.java`

# Package and Source Files

- **Base directory:** holds your program's Files
- Path name, relative to base directory, must match package name:

```
com/horstmann/bigjava/Financial.java
```



**Figure 5**  
Base Directories  
and Subdirectories  
for Packages

## Self Check 8.18

---

Which of the following are packages?

- a. `java`
- b. `java.lang`
- c. `java.util`
- d. `java.lang.Math`

## Self Check 8.19

---

Is a Java program without `import` statements limited to using the default and `java.lang` packages?

## Self Check 8.20

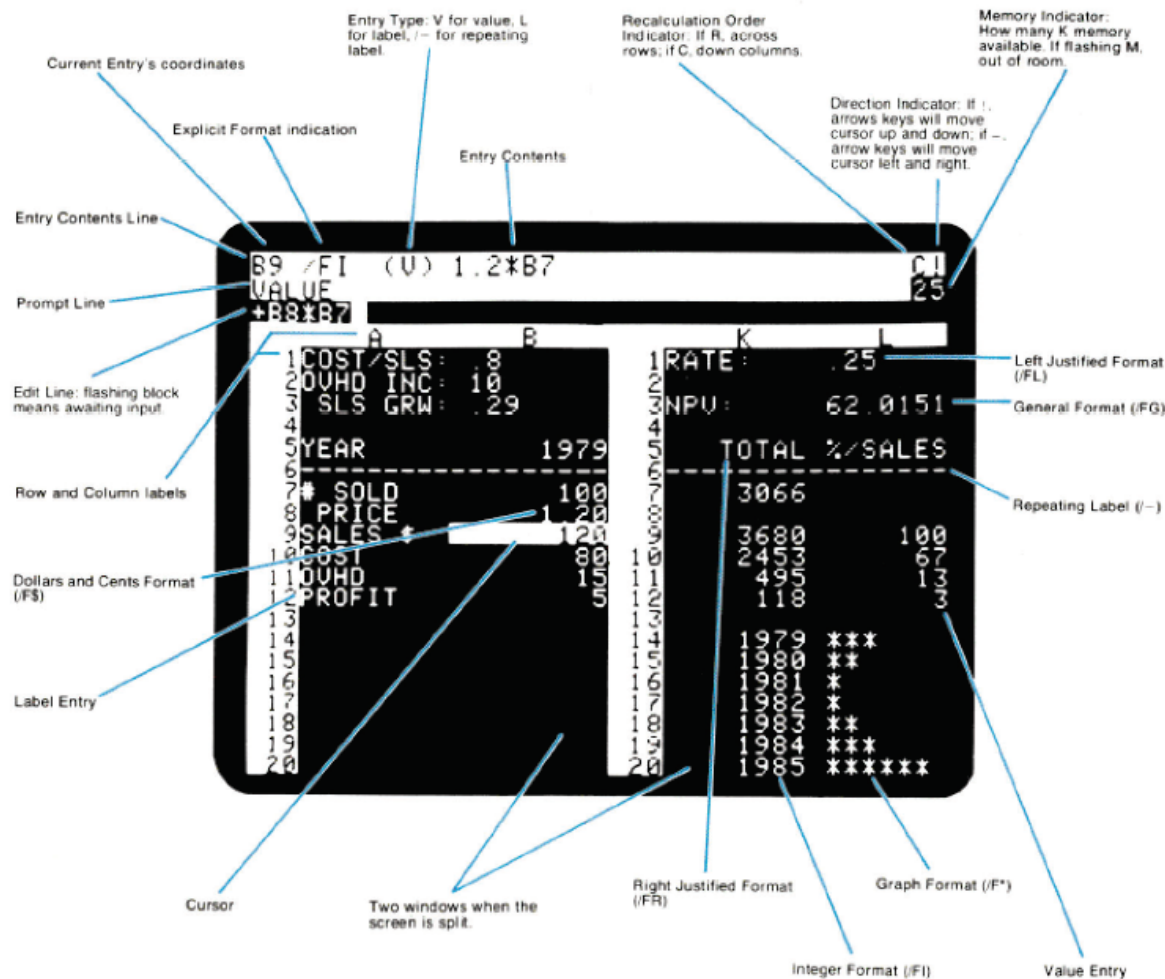
---

Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?



# The Explosive Growth of Personal Computers

## A VISICALC™ Screen:



The VisiCalc Spreadsheet Running on an Apple II

# Unit Testing 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 Testing Frameworks

---

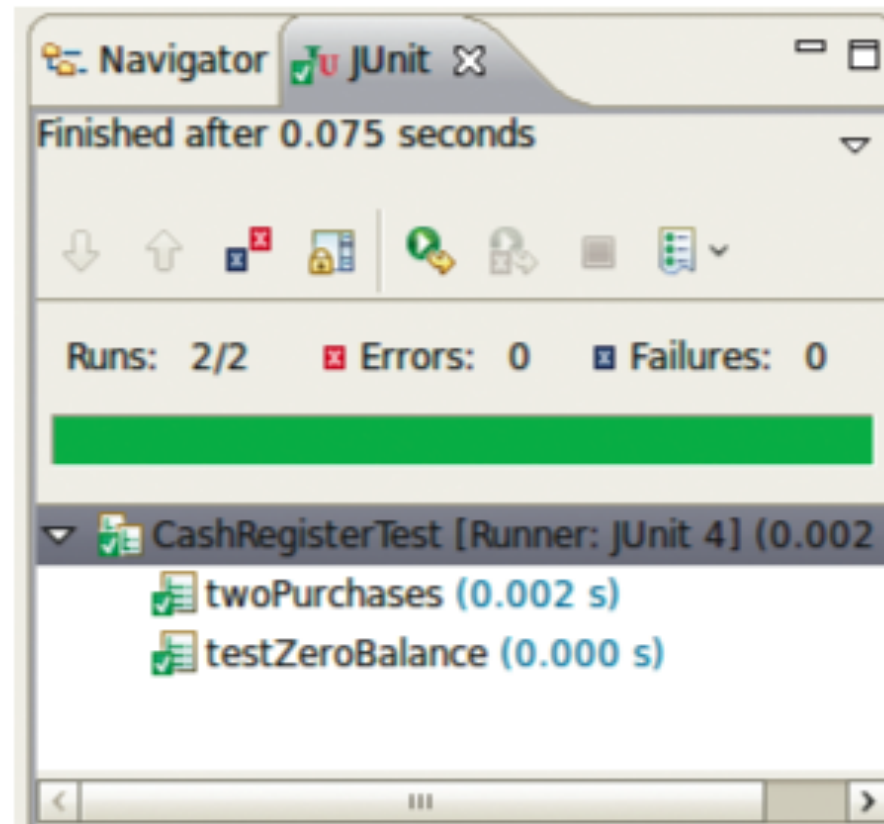
- Customary that name of the test class ends in `Test`:

```
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
 . . .
}
```

# Unit Testing Frameworks

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

**Figure 6**  
Unit Testing with JUnit



## Self Check 8.21

---

Provide a JUnit test class with one test case for the `Earthquake` class in Chapter 5.

## Self Check 8.22

---

What is the significance of the `EPSILON` parameter in the `assertEquals` method?