



# D06

# PROGRAMMING with JAVA

## Ch9 – Designing Classes

# Chapter Goals

---

- To learn how to choose appropriate classes to implement
- 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 the difference between **instance methods** and **static methods**
- To introduce the concept of **static fields**
- To understand the **scope** rules for local variables and instance fields
- To learn about **packages**

# Choosing Classes



In object-oriented programming, the actions appear as **methods**. Each method belongs to a class. **Classes** are collections of **objects**, and objects are not actions –they are entities. So you have to start the programming activity by identifying objects and the classes to which they belong. A class should represent a single concept

- Some of the classes that you have seen represent **concepts from mathematics**: **Point**, **Rectangle**, **Ellipse**, ...
- Other classes are abstractions of **real-life entities**: **BankAccount**, **CashRegister**, ... For these classes, the properties of a typical object are easy to understand
- Another useful category of classes can be described as **actors**: **Scanner**, **Random**, ... Objects of an actor class do some kinds of work for you
- Very occasionally, a class has no objects, but it contains a collection of related static methods and constants: **Math**, ... Such a class is called a **utility class**
- Finally, you have seen **classes with only a `main()` method**. Their sole purpose is to start a program

# Self Test

---

1. What is the rule of thumb for finding classes?
2. Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `NextMove`?

# Answers

---

1. Look for nouns in the problem description
2. Yes (`ChessBoard`) and no (`NextMove`)

# Cohesion



A class should represent a single concept

- The public methods and constants that the public interface exposes should be **cohesive**. That is, all interface features should be closely related to the single concept that the class represents
- The following **CashRegister** class lacks cohesion, since it involves two concepts: a cash register that holds coins and computes their total, and the values of individual coins:

```
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;
    . . .
}
```

*Continued...*

# Cohesion

- It makes sense to have a separate **Coin** class and have coins responsible for knowing their values:

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

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

# Coupling

- Many classes need other classes in order to do their jobs. For example, the restructured `CashRegister` class depends on `Coin` to determine the value of the payment
- To visualize relationships, such as dependence between classes, programmers draw **class diagrams**
- The **Unified Modeling Language (UML)** notation distinguishes between **object diagrams** and **class diagrams**:
  - In an **object diagram** the class names are underlined;
  - In a **class diagram** the class names are not underlined, and you denote dependency by a dashed line with a shaped open arrow tip ( $\rightarrow$ ) that points to the dependent class

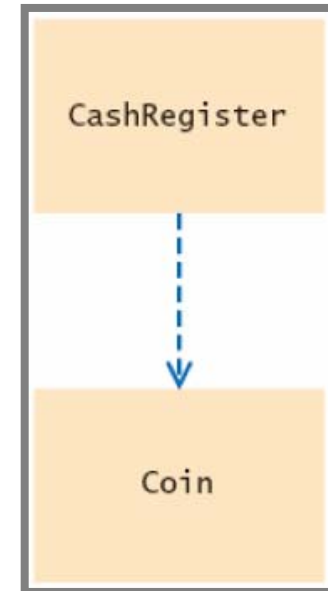


Figure 1  
Dependency Relationship Between the  
CashRegister and Coin Classes

*Continued...*

# Coupling

- If many classes of a program depend on each other, then we say that the **coupling** between classes is high



Why does coupling matter? If we would like to use a class in another program, we have to take with it all the classes on which it depends. Minimize coupling to minimize the impact of interface changes

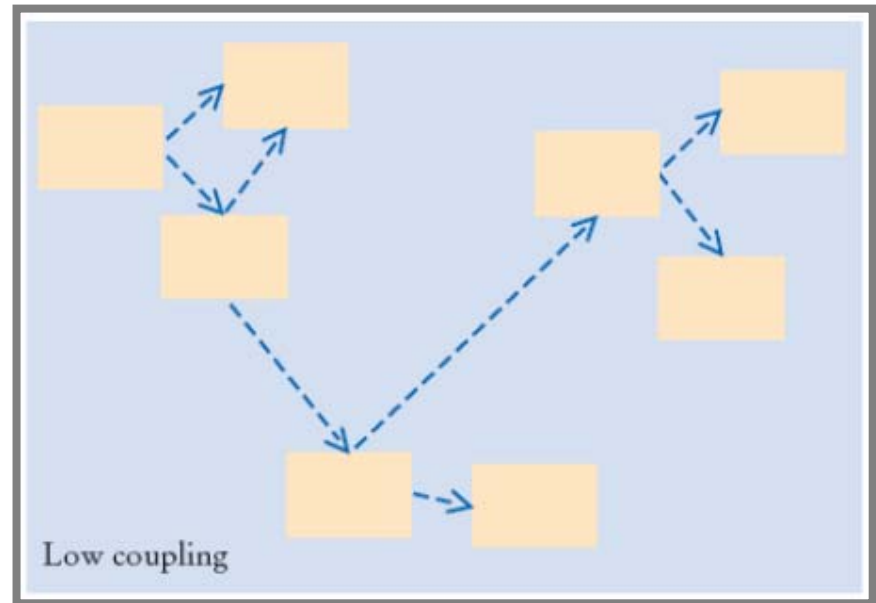
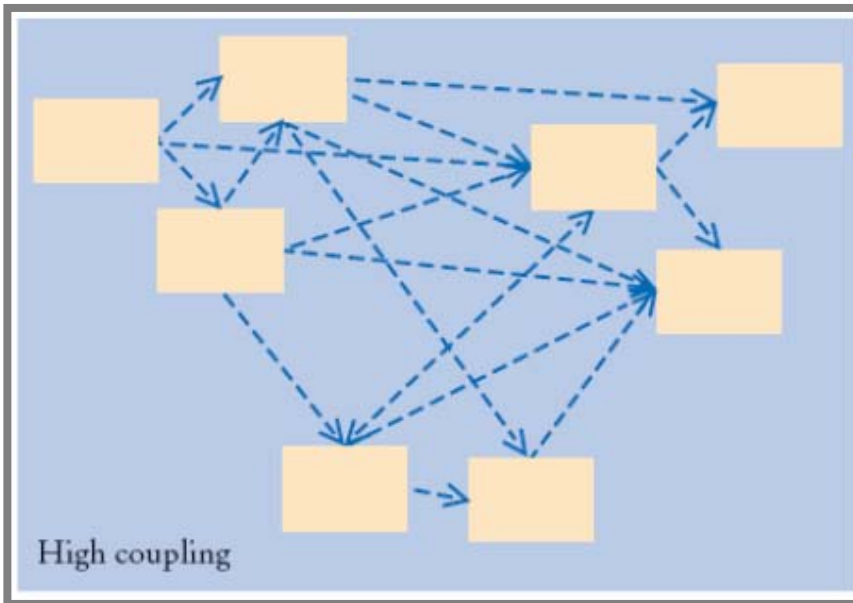


Figure 2  
High and Low Coupling Between Classes

# Consistency

---



In this section you learned of two criteria to analyze the quality of the public interface of a class: you should maximize cohesion and remove unnecessary coupling

- There is another criterion, **consistency**: when you have a set of methods, follow a consistent scheme for their names and parameters

# Self Check

---

3. Why is the `CashRegister` class from Chapter 4 not cohesive?
4. Why does the `Coin` class not depend on the `CashRegister` class?
5. Why should coupling be minimized between classes?

# Answers

---

3. Some of its features deal with payments, others with coin values
4. None of the coin operations require the `CashRegister` class
5. If a class doesn't depend on another, it is not affected by interface changes in the other class

# Accessors, Mutators, and Immutable Classes

- A **mutator** method modifies the object on which it is invoked, whereas an **accessor** method merely accesses information without making any modification

```
double balance = account.getBalance(); // accessor  
account.deposit(1000); // mutator
```

- You can call an accessor method as many times as you like –you always get the same answer, and it does not change the state of your object. Some classes have been designed to have only accessor methods and no mutator methods at all. Such classes are called **immutable**

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



An **immutable class** has a major advantage: It is safe to give out references to its objects freely (if no method can change the object's value, then no code can modify the object at an unexpected time)

# Self Check

---

6. Is the `substring` method of the `String` class an accessor or a mutator?
7. Is the `Rectangle` class immutable?

# Answers

---

6. It is an accessor—calling `substring` doesn't modify the string on which the method is invoked. In fact, all methods of the `String` class are accessors
7. `No-translate` is a mutator

# Side Effects

- A **side effect** of a method is any kind of modification of data that is observable outside the method. Mutator methods have a side effect, namely the modification of the implicit parameter. Here is an example of a method with another kind of side effect, the updating of an explicit parameter:

```
public void transfer(double amount, BankAccount other)
{
    balance = balance - amount;
    other.balance = other.balance + amount;
    // Modifies explicit parameter
}
```



Updating an explicit parameter can be surprising to programmers, and it is best to avoid it whenever possible

*Continued...*

# Side Effects

- Another example of a side effect is output. Consider how we have always printed a bank balance:

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

Why don't we simply have a `printBalance()` method?:

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

Bad idea!: message is in English, and relies on `System.out` (it won't work in an embedded system, such as the computer inside an automatic teller machine). The `printBalance()` method couples the `BankAccount` class with the `System` and `PrintStream` classes. It is best to decouple input/output from the actual work of your classes

# Self Check

---

8. If `a` refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?
9. Consider the `DataSet` class of Chapter 7. Suppose we add a method

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

Does this method have a side effect?

# Answers

---

8. No—a side effect of a method is any change outside the implicit parameter
9. Yes—the method affects the state of the `Scanner` parameter

# Common Error – Trying to Modify Primitive Type Parameter



Methods can't update parameters of primitive type (numbers, char, and boolean):

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

- This doesn't work. Let's consider a method call:

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

As the method starts, the parameter variable `otherBalance` is set to the same value as `savingsBalance`. Then the value of the `otherBalance` variable is modified, but that modification has no effect on `savingsBalance`, because `otherBalance` is a separate variable. When the method terminates, the `otherBalance` variable dies, and `savingsBalance` isn't increased

*Continued...*

# Common Error – Trying to Modify Primitive Type Parameter

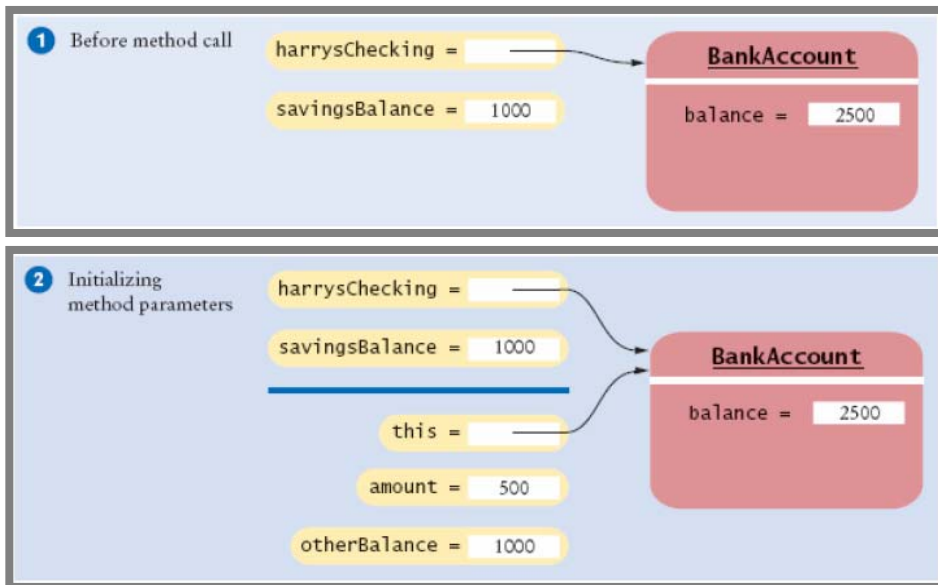
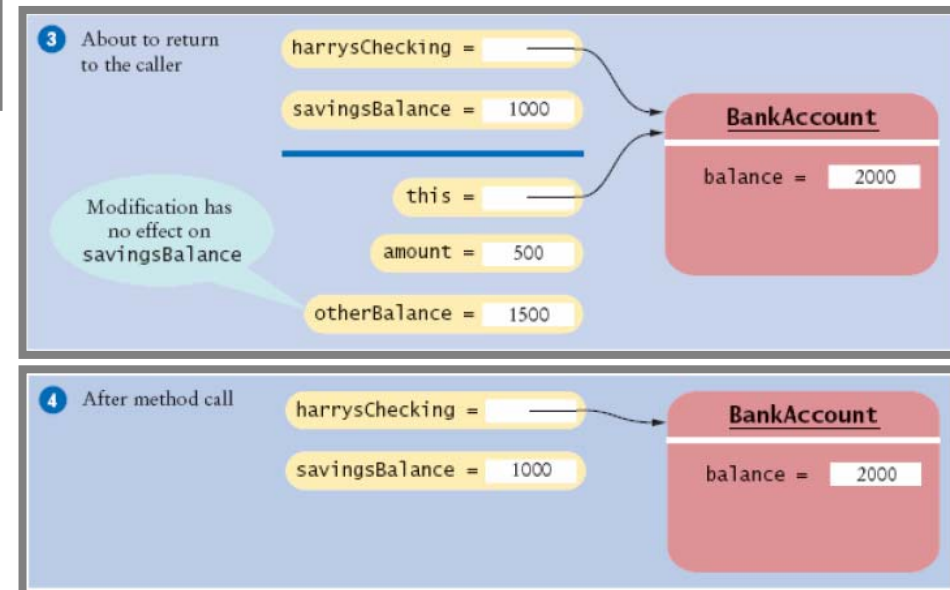


Figure 3:  
Modifying a Numeric  
Parameter Has No Effect  
on Caller



# Call By Value and Call By Reference

- In Java, method parameters are copied into the parameter variables when a method starts. Computer scientists call this call mechanism “call by value”. Other programming languages, such as C++, support an alternate mechanism, called “call by reference”



In Java, objects themselves are never passed as parameters; instead, both numbers and object references are copied by value. For that reason, in Java: (i) it is not possible to implement methods that modify the contents of number variables, and (ii) a method can change the state of an object reference parameter, but it cannot replace the object reference with another:

```
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
    }
}
```

*Continued...*

# Call By Value Example

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

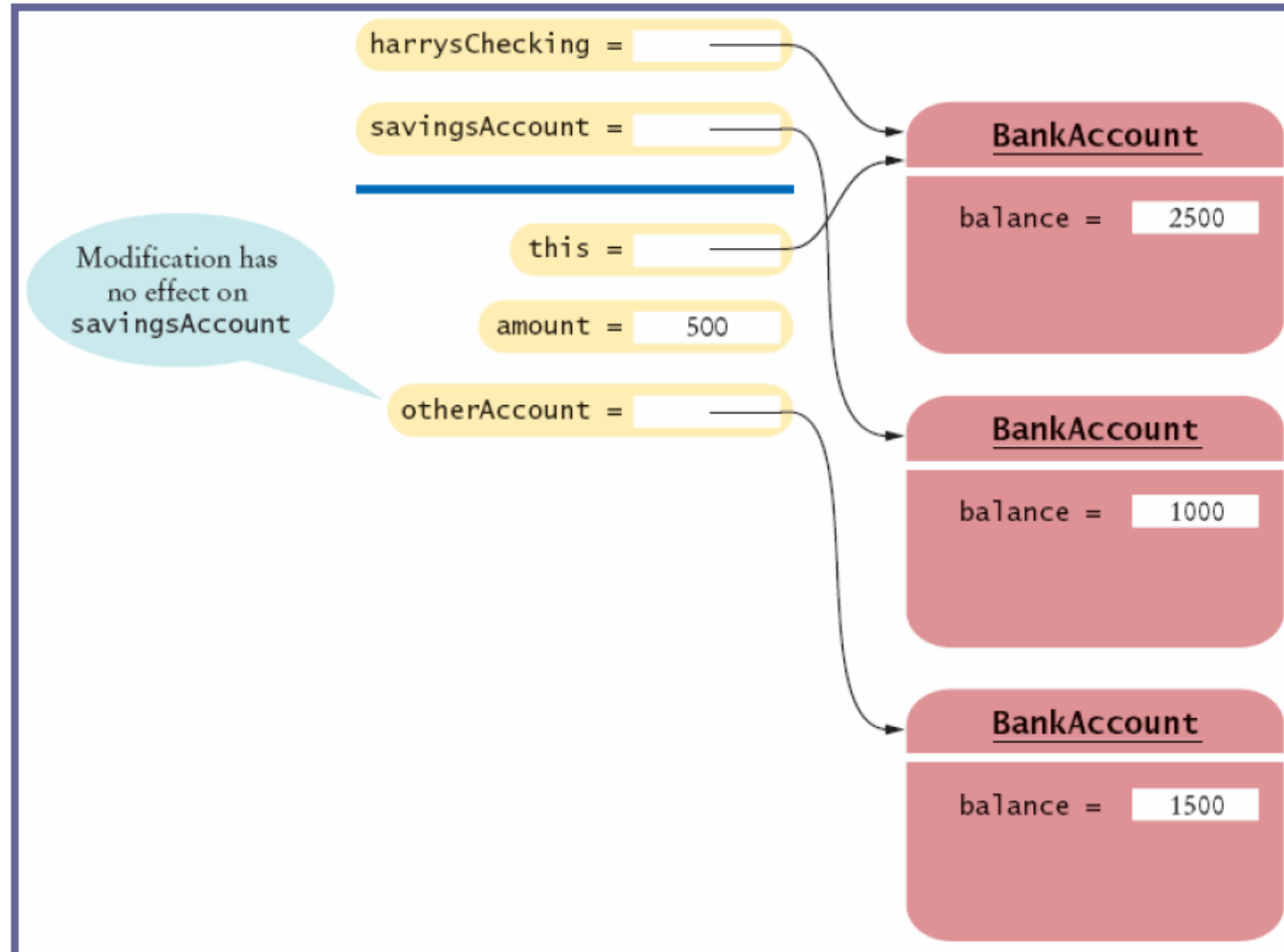


Figure 4:  
Modifying an Object  
Reference Parameter  
Has No Effect on the  
Caller

# Preconditions

- A **precondition** is a requirement that the caller of a method must obey (for example, the `deposit()` method of the `BankAccount` class has a precondition that the amount to be deposited should not be negative)



It is the responsibility of the caller never to call a method if one of its preconditions is violated. Therefore, a **precondition** is an important part of the method, and you must document it:

```
/**
 * Deposits money into this account.
 * @param amount the amount of money to deposit
 * (Precondition: amount >= 0)
 */
```

- Preconditions are typically provided for one of two reasons:
  1. To restrict the parameters of a method
  2. To require that a method is only called when it is in the appropriate state (for example, once a `Scanner` has run out of input, it is not longer legal to call the `next()` method)

*Continued...*

# Preconditions

- A method is responsible for operating correctly only when its caller has fulfilled all preconditions. The method is free to do anything if a precondition is not fulfilled
- What should a method actually do when it is called with inappropriate inputs? There are two choices:
  1. A method can check for the violation and **throw an exception**. Then the method does not return to its caller; instead, control is transferred to an exception handler. If no handler is present, then the program terminates

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

2. A method can **skip the check** and work under the assumption that the preconditions are fulfilled. If they aren't, then any data corruption (such as a negative balance) or other failures are the caller's fault

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

- The first approach can be inefficient. The second approach can be dangerous. The **assertion mechanism** was invented to give you the best of both approaches

*Continued...*

# Preconditions

- An **assertion** is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true:

```
public double deposit (double amount)
{
    assert amount >= 0;
    balance = balance + amount;
}
```



When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, and assertion checking is enabled, then the program terminates with an **AssertionError**. If assertion checking is disabled, then the assertion is never checked, and the program runs at full speed

- By default, assertion checking is disabled when you execute a program. You definitely want to turn assertion checking on during program development and testing. You can turn assertions off after you have tested your program, so that it runs at maximum speed

*Continued...*

# Syntax 9.1: Assertion

---

**assert *condition*;**

**Example:**

**assert amount >= 0;**

**Purpose:**

**To assert that a condition is fulfilled. If assertion checking is enabled and the condition is false, an assertion error is thrown.**

# Postconditions

- When a method is called in accordance with its preconditions, then the method promises to do its job correctly. A different kind of promise that the method makes is called a **postcondition**: a condition that is true after a method has completed
- There are two kinds of postconditions:
  1. The return value is computed correctly
  2. The object is in a certain state after the method call is completed
- Here is a **postcondition** that makes a statement about the object state after the `deposit()` method is called (as long as the precondition is fulfilled, this method guarantees that the balance after the deposit is not negative):

```
/**  
    Deposits money into this account.  
    (Postcondition: getBalance() >= 0)  
    @param amount the amount of money to deposit  
    (Precondition: amount >= 0)  
*/
```

*Continued...*

# Self Check

---

10. Why might you want to add a precondition to a method that you provide for other programmers?
11. 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?

# Answers

---

10. Then you don't have to worry about checking for invalid values—it becomes the caller's responsibility
11. No—you can take any action that is convenient for you

# Static Methods

---

- Sometimes you need a method that is not invoked on an object. Such a method is called a **static method** or a **class method** (e.g., the `sqrt()` method in the `Math` class). In contrast, the methods that you wrote up to now are often called **instance methods** because they operate on a particular instance of an object
- Why would you want to write a method that does not operate on an object? The most common reason is that you want to encapsulate some computation that involves only numbers. Because numbers aren't objects, you can't invoke methods on them (e.g., `x.sqrt()` can never be legal in Java)

# Static Methods

- A typical example of a **static method** that carries out some simple algebra:

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

- When calling a static method, you supply the name of the class containing the method so that the compiler can find it:

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

- **main()** is static –there aren't any objects yet

# Self Check

---

12. Suppose Java had no static methods. Then all methods of the `Math` class would be instance methods. How would you compute the square root of  $x$ ?
13. Harry turns in his homework assignment, a program that plays tic-tac-toe. His solution consists of a single class with many static methods. Why is this not an object-oriented solution?

# Answers

---

12. 

```
Math m = new Math();  
y = m.sqrt(x);
```
13. In an object-oriented solution, the main method would construct objects of classes Game, Player, and the like. Most methods would be instance methods that depend on the state of these objects.

# Static Fields

- Sometimes, you need to store values outside any particular object. For this purpose, you use **static fields**, also called **class fields** (a static field belongs to the class, not to any object)
- Example: we will use a version of our `BankAccount` class in which each bank account object has both a balance and an account number. We want the bank account constructor to construct the first account with number 1001, the next with number 1002, and so on → we need to have a single value of `lastAssignedNumber` that is the same for the entire class:

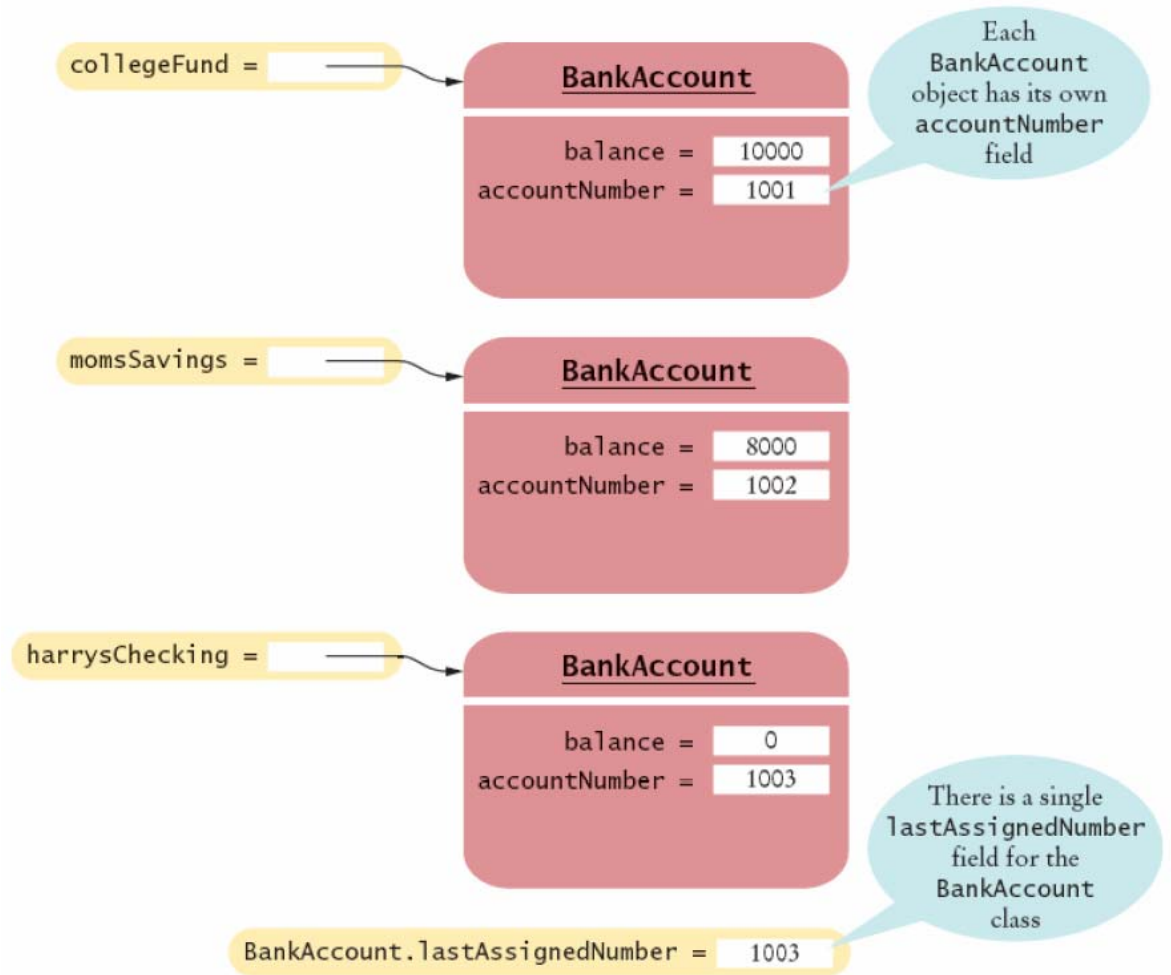
```
public class BankAccount
{
    . . .
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber = 1000;
}
```

(if `lastAssignedNumber` was not static, each instance of `BankAccount` would have its own value of `lastAssignedNumber`)

*Continued...*

# Static Fields

Figure 5:  
A Static Field and  
Instance Fields



*Continued...*

# Static Fields

- Every method of a class can access its **static fields**. Here is the constructor of the `BankAccount` class:

```
public BankAccount()  
{  
    // Generates next account number to be assigned  
    lastAssignedNumber++; // Updates the static field  
  
    // Assigns field to account number of this bank  
    account accountNumber = lastAssignedNumber; // Sets the instance field  
}
```

 There are three ways to initialize a static field:

1. Do nothing. The static field is then initialized with `0` (for numbers), `false` (for boolean values), or `null` (for objects)
2. Use an explicit initializer, such as in the former example (the initialization is executed once when the class is loaded)
3. Use a static initialization block (advanced topic)

*Continued...*

# Static Fields



Like instance fields, static fields should always be declared as **private** to ensure that methods of other classes do not change their values. The exception to this rule are **static constants**, which may be either private or public (it makes sense to declare constants as static –you wouldn't want every object of the `BankAccount` class to have its own set of variables with these constant values):

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

# Self Check

---

14. Name two static fields of the `System` class.
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 fields `static`. Then `main` can call the other static methods, and all of them can access the static fields. Will Harry's plan work? Is it a good idea?

# Answers

---

14. `System.in` and `System.out`

15. Yes, it works. Static methods can access static fields of the same class. But it is a terrible idea. As your programming tasks get more complex, you will want to use objects and classes to organize your programs.

# Scope of Local Variables

- The **scope** of a variable is the region of a program in which the variable can be accessed
- The **scope of a local variable** extends from the point of its declaration to the end of the block that encloses it
- Sometimes, the same variable name is used in two methods (these variables are independent of each other):

```
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);
    }
}
```

*Continued...*

# Scope of Local Variables

- In Java, the **scope of a local variable** can never contain the definition of another local 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
    . . .
}
```

- However, you can have local variables with identical names if their 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
    . . .
}
```

# Scope of Class Members

---

- By **class members** we mean fields and methods of a class
- **Private members** have class scope: You can access all members in any of the methods of the class
- If you want to use a **public member** (field or method) outside its class, you must **qualify the name**:
  - You qualify a static field or method by specifying the class name, such as `Math.sqrt()`
  - You qualify an instance field or method by specifying the object to which the field or method should be applied, such as `harrysChecking.getBalance()`

*Continued...*

# Scope of Class Members

- Inside a method, you don't need to qualify fields or methods that belong to the same class (instance fields automatically refer to the implicit parameter of the method or **this**, that is, the object on which the method is invoked):

```
public class BankAccount
{
    public void transfer(double amount, BankAccount other)
    {
        withdraw(amount); // i.e., this.withdraw(amount);
        other.deposit(amount);
    }
    ...
}
```

*Continued...*

# Overlapping Scope

- Problems arise if you have two identical variable names with overlapping scope. This can never occur with local variables, but the scopes of identically named local variables and instance fields can overlap:

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



A local variable can **shadow** a field with the same name. You can access the shadowed field name by qualifying it with the **this** reference

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

# Self Check

---

16. Consider the `deposit` method of the `BankAccount` class. What is the scope of the variables `amount` and `newBalance`?
17. What is the scope of the `balance` field of the `BankAccount` class?

# Answers

---

16. The scope of `amount` is the entire `deposit` method. The scope of `newBalance` starts at the point at which the variable is defined and extends to the end of the method.
17. It starts at the beginning of the class and ends at the end of the class.

# Organizing Related Classes Into Packages

---

- A Java program consists of a collection of classes. A Java **package** is a set of related classes (the Java library consists of dozens of 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

- A **package name** consists of one or more identifiers separated by periods

*Continued...*

# Organizing Related Classes Into Packages

<b>Package</b>	<b>Purpose</b>	<b>Sample Class</b>
<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>PrintScreen</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 through SQL	<code>ResultSet</code>
<code>java.swing</code>	Swing user interface	<code>JButton</code>
<code>omg.org.CORBA</code>	Common Object Request Broker Architecture for distributed objects	<code>IntHolder</code>

# Organizing Related Classes Into Packages

- Example: to put the `Financial` class introduced into a package named `com.ajuanp.java`, the `Financial.java` file must start as follows:

```
package com.ajuanp.java;  
  
public class Financial  
{  
    . . .  
}
```



In addition to the named packages (such as `java.util`), there is a special package, called the **default package**, which has no name. If you did not include any package statement at the top of your source file, its classes are placed in the default package

# Syntax 9.2: Package Specification

---

```
package packageName;
```

**Example:**

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

**Purpose:**

To declare that all classes in this file belong to a particular package

# Importing Packages

- If you want to use a class from a package, you can refer to it by its full name:

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

- You can instead import a name with an **import** statement:

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

- You can import all classes of a package with an import statement that ends in **.\*** :

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

- You never need to import the classes in the **java.lang** package explicitly. That is the package containing the most basic Java classes, such as **Math** and **Object**. These classes are always available for you
- Finally, you don't need to import other classes in the same package you have already imported (the compiler will find them)

# Package Names and Locating Classes

- Placing related classes into a package is clearly a convenient mechanism to organize classes. However, there is a more important reason for packages: to avoid name clashes

```
java.util.Timer vs. javax.swing.Timer
```

- Package names should be unambiguous
- Recommendation: start with reversed domain name:

```
com.ajuanp  
edu.uoc.ajuanp
```

# Package Names and Locating Classes

- A **package** is located in a subdirectory that matches the package name. The parts of the name between periods represent successively nested directories (for example, the package `com.ajuanp.java` would be placed in a subdirectory `com/ajuanp/java`)

```
com/ajuanp/java/Financial.java
```

- You need to add the directories that might contain packages to the **class path**:

```
Linux → export CLASSPATH = /home/user:.  
Windows → set CLASSPATH = c:\home\user;.
```

- The class path contains the **base directories** that may contain package directories

# Base Directories and Subdirectories for Packages

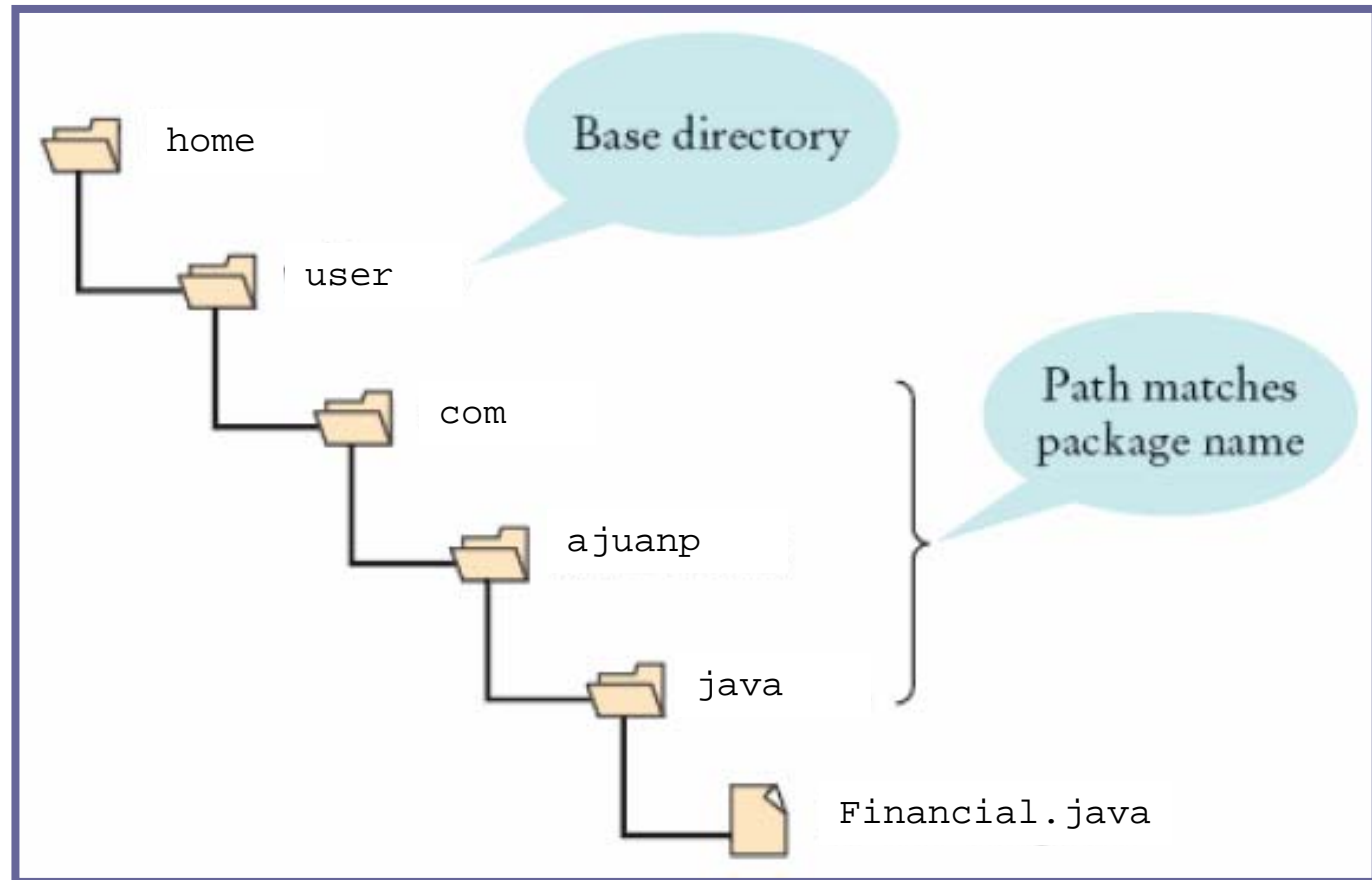


Figure 6:  
Base Directories and Subdirectories for Packages

# Self Check

---

18. Which of the following are packages?
  - a. `java`
  - b. `java.lang`
  - c. `java.util`
  - d. `java.lang.Math`
  
19. Can you write a Java program without ever using import statements?
  
20. Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\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`

# Answers

---

- 18.
- a. No
  - b. Yes
  - c. Yes
  - d. No
19. Yes –if you use fully qualified names for all classes, such as `java.util.Random` and `java.awt.Rectangle`
20. `/home/me/cs101/hw1/problem1` or, on Windows, `c:\me\cs101\hw1\problem1`

# Chapter Summary

---

- A **class** should represent a single concept from the problem domain, such as business, science, or mathematics
- The public interface of a class is **cohesive** if all of its features are related to the concept that the class represents
- A class **depends on** another class if it uses objects of that class
- It is a good practice to minimize the **coupling** (i.e., dependency) between classes
- An **immutable class** has no mutator methods
- A **side effect** of a method is any externally observable data modification
- You should minimize side effects that go beyond modification of the implicit parameter
- In Java, a method can never change parameters of primitive type
- In Java, a method can change the state of an object reference parameter, but it cannot replace the object reference with another

*Continued...*

# Chapter Summary

---

- A **precondition** is a requirement that the caller of a method must meet. If a method is called in violation of a precondition, the method is not responsible for computing the correct result
- An **assertion** is a logical condition in a program that you believe to be true
- If a method has been called in accordance with its preconditions, then it must ensure that its **postconditions** are valid
- A **static method** is not invoked on an object
- A **static field** belongs to the class, not to any object of the class
- The **scope** of a variable is the region of a program in which the variable can be accessed
- The **scope of a local variable** cannot contain the definition of another variable with the same name
- A **qualified name** is prefixed by its class name or by an object reference, such as `Math.sqrt()` or `other.balance`

*Continued...*

# Chapter Summary

---

- An unqualified instance field or method name refers to the **this** parameter
- A local variable can **shadow** a field with the same name. You can access the shadowed field name by qualifying it with the **this** reference
- A **package** is a set of related classes
- The **import** directive lets you refer to a class of a package by its class name, without the package prefix
- Use a domain name in reverse to construct unambiguous package names
- The path of a class file must match its package name