Designing Classes
(POO 2007/2008)

Fernando Brito e Abreu (fba@di.fct.unl.pt)
Universidade Nova de Lisboa (http://www.unl.pt)
QUASAR Research Group (http://ctp.di.fct.unl.pt/QUASAR)

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

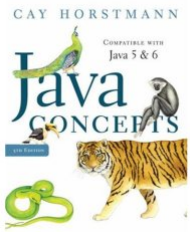
Continued…

Chapter Goals

- 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

Where can I find this chapter?

← Chapter 9



← Chapter 8

Choosing 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`

Choosing 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 better name than `ComputePaycheck`

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

Continued...

Cohesion

- 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)

Coupling

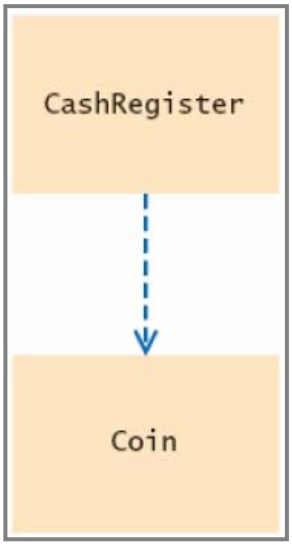


Figure 1
Dependency Relationship Between the CashRegister and Coin Classes

05-03-2008

High and Low Coupling Between Classes

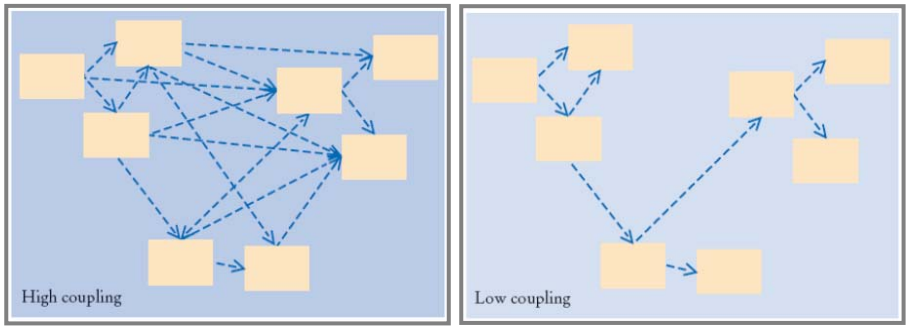


Figure 2
High and Low Coupling Between Classes

05-03-2008

Accessors, Mutators, and Immutable Classes

- Accessor (aka selector): does not change the state of the implicit parameter

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

- Mutator (aka modifier): modifies the state of the object on which it is invoked, often based on explicit parameters

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


Accessors, Mutators, and Immutable Classes

- 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

Side Effects

- Side effect of a method: any externally observable data modification

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

- Updating explicit parameter can be surprising to programmers; it is best to avoid it if possible

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`

It is best to decouple input/output from the actual work of your classes

- You should minimize side effects that go beyond modification of the implicit parameter

Common Error – Trying to Modify Primitive Type Parameter

```
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

Modifying a Numeric Parameter Has No Effect on Caller

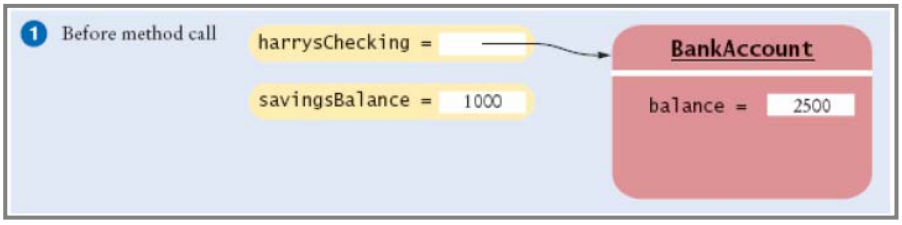


Figure 3(1):
Modifying a Numeric Parameter Has No Effect on Caller

Continued...

Modifying a Numeric Parameter Has No Effect on Caller

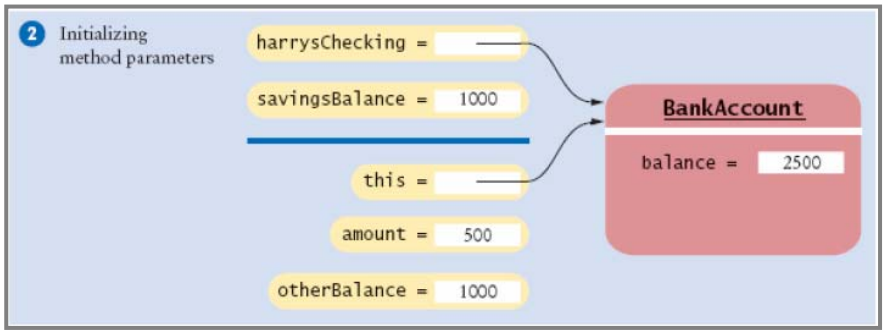


Figure 3(2):
Modifying a Numeric Parameter Has No Effect on Caller

Continued…

Modifying a Numeric Parameter Has No Effect on Caller

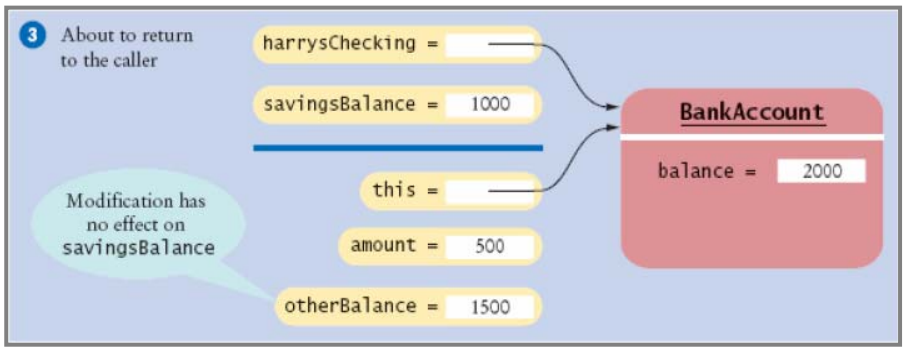


Figure 3(3):
Modifying a Numeric Parameter Has No Effect on Caller

Continued…

Modifying a Numeric Parameter Has No Effect on Caller

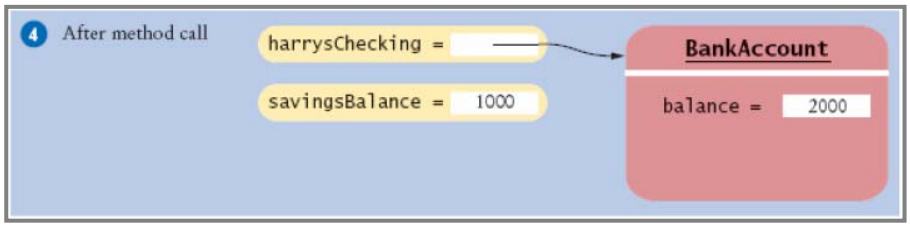


Figure 3(4):
Modifying a Numeric Parameter Has No Effect on Caller

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
  - Pascal, C or C++ also have call by reference

Continued…

Call By Value and Call By Reference

- A method can change state of object reference parameters, but cannot replace an 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
   }
}
```


Call By Value Example

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

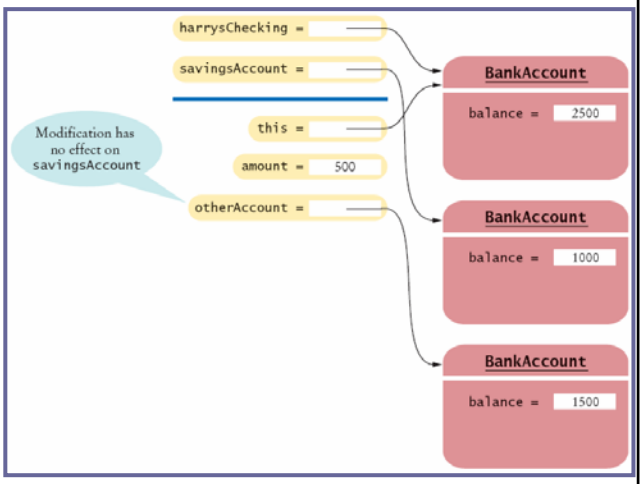


Figure 4:
Modifying an Object Reference Parameter Has No Effect on the Caller

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)
*/
```

Continued...

Preconditions

- Typical use:
  - To restrict the parameters of a method
  - To require that a method is only called when the object is in an appropriate state
- If precondition is violated, method is not responsible for computing the correct result. It is free to do *anything*.

Preconditions

- Method may throw exception if precondition violated (more on a following chapter …)

```
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

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

To enable assertion checking:

```
java -enableassertions MyProg
java -ea MyProg
```

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

Continued…

Preconditions

- Many beginning programmers silently return to the caller

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


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

- Condition 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:
  1. The return value is computed correctly
  2. The object is in a certain state after the method call is completed

Continued...

Postconditions

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

Don't document trivial postconditions that repeat the `@return` clause

Continued...

Postconditions

- 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 precondition, method must fulfill postcondition

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

```
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);
```

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

```
public static void main(String[] args)
{ … }
```


Static Fields

- A static field belongs to the class, not to any object of the class. Also called *class field*.

```
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

```
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
}
```

- Minimize the use of static fields. (Static final fields are ok.)

Static Fields

- Three ways to initialize:
  1. Do nothing. Field is 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,
      // when class is loaded
}
```

  3. Use a static initialization block

Continued...

Static Fields

- Static fields should always be declared as `private`
- 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
}
```


A Static Field and Instance Fields

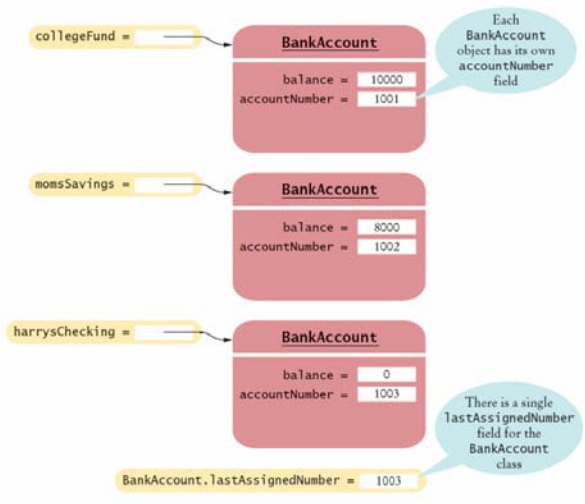


Figure 5:
A Static Field and Instance Fields

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

Continued...

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
   . . .
}
```


Continued...

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
   . . .
}
```


Scope of Class Members

- Private members have class scope: You can access all members in any method of the class
- Must qualify public members outside scope

```
Math.sqrt
harrysChecking.getBalance
```


Continued...

Scope of Class Members

- Inside a method, no need to qualify fields or methods that belong to the same class
- An unqualified instance field or method name refers to the `this` parameter

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


Overlapping Scope

- A local variable can *shadow* a field 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; // Field with the same name
}
```

Continued...

Overlapping Scope

- Access shadowed fields by qualifying them with the `this` reference

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


Organizing Related Classes Into Packages

- Package: Set of related classes
- 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

Continued…

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

Organizing Related Classes Into Packages

Package	Purpose	Sample Class
java.lang	Language Support	Math
java.util	Utilities	Random
java.io	Input and Output	PrintScreen
java.awt	Abstract Windowing Toolkit	Color
java.applet	Applets	Applet
java.net	Networking	Socket
java.sql	Database Access	ResultSet
java.swing	Swing user interface	JButton
org.omg.CORBA	Common Object Request Broker Architecture	IntHolder

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

- 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)
```

Importing Packages

- 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 and Locating Classes

- 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 Bertha Walters' classes (walters@cs.sjsu.edu)

Continued...

Package Names and Locating Classes

- Path name should match package name

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

- Path name starts with class path

```
export CLASSPATH=/home/walters/lib:.
set CLASSPATH=c:\home\walters\lib;.
```

- Class path contains the base directories that may contain package directories

Base Directories and Subdirectories for Packages

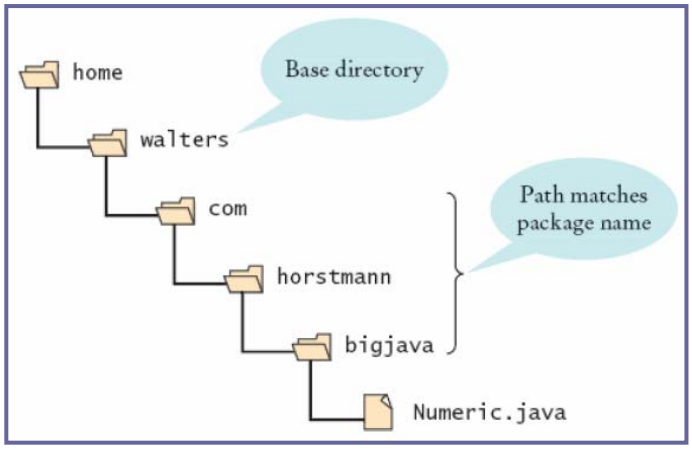


Figure 6:
Base Directories and Subdirectories for Packages