

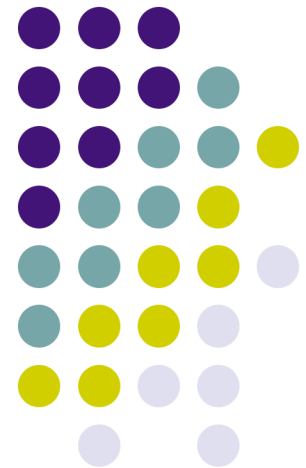


# Principles of Computer Science I

Prof. Nadeem Abdul Hamid

CSC 120 – Fall 2006

Lecture Unit 9 - Designing Classes





# Lecture Outline

- Choosing and designing classes
  - UML
- Understanding side effects
- Pre- and postconditions
- Static methods and fields
- Scope rules
- Organizing classes using packages



# Choosing Classes

- Class represents a single concept/abstraction from the problem domain
  - Name for the class should be a *noun*
- Concepts from mathematics
  - Point
  - Rectangle
  - Eclipse
- Abstractions of real-life entities
  - BankAccount
  - CashRegister



# Choosing Classes (cont.)

- *Actor* classes (names end in *-er*, *-or*)
  - Objects of these classes do some sort of work for you
  - `Scanner`
  - `Random` (better name: `RandomNumberGenerator`)
- *Utility* classes
  - No objects; just contain collection of static methods and constants
  - `Math`
- *Program starters*
  - Contain only a `main` method
- Actions are not classes: e.g. `ComputePaycheck`



# Cohesion

- Criteria for analyzing quality of a public interface: *cohesion and coupling*
- A class should represent a single concept
  - Cohesive: all its features relate to the concept that the class represents
- Non-cohesive example (split into two classes):

```
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;  
    . . .  
}
```



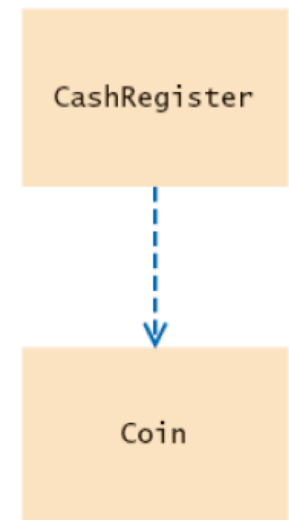
# Coupling

- A class *depends* on another if it uses objects of that class
  - `CashRegister` depends on `Coin` (not vice versa)
- *Coupling*: the amount of dependence classes have on each other
  - Many classes of a program depend on each other: high coupling
  - Few dependencies between classes: low coupling
- Which is better, high or low?
  - Hint: think about effect of interface changes



# UML Diagrams

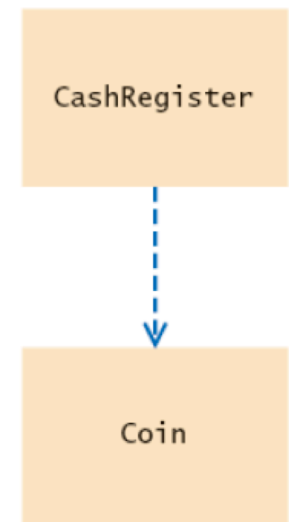
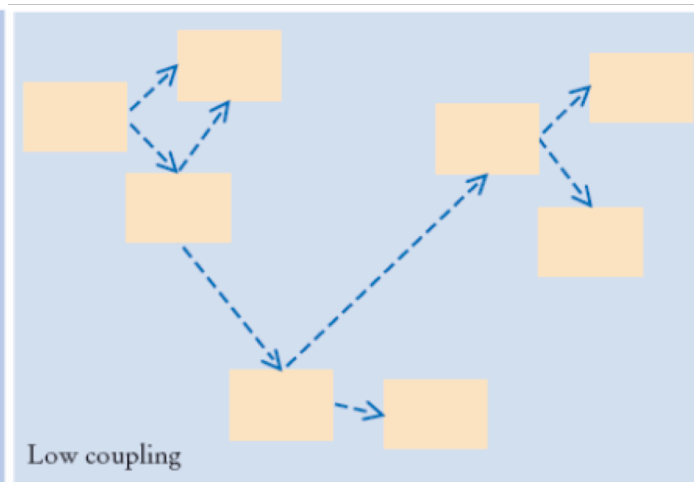
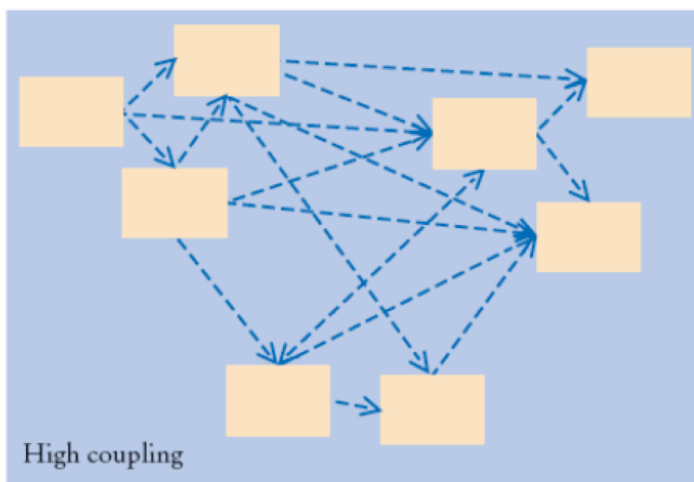
- ‘Unified Modeling Language’
  - Notation for object-oriented analysis and design
- Class diagrams denote dependencies by dashed line with arrow pointing to class that is depended on





# UML Diagrams

- ‘Unified Modeling Language’
  - Notation for object-oriented analysis and design
- Class diagrams denote dependencies by dashed line with arrow pointing to class that is depended on







# Consistency

- Another useful criterion for good design
- Follow consistent scheme for class/method names and parameters
- Java standard library contains many inconsistencies
  - `JOptionPane.showInputDialog( prompt )`
  - `JOptionPane.showMessageDialog( null, message )`



# Accessor/Mutator Methods

- *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 Classes

- Contains only accessor methods, no mutator methods
- Example: `String`

```
String name = "John Q. Public";  
String uppercased = name.toUpperCase();  
    // name is not changed
```
- Advantage of immutable classes
  - Safe to give out copies of references to objects – object cannot be modified unexpectedly



# Side Effects

- *Side effect*: any externally observable modification of data
  - Mutator method modifies implicit parameter object
  - Another kind of side effect:

```
public void transfer(double amount, BankAccount other) {  
    balance = balance - amount;  
    other.balance = other.balance + amount;  
    // Modifies explicit parameter  
}
```
  - Updating explicit parameter can be surprising – best to avoid
  - Another kind of side effect: output



# Output Side Effects

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

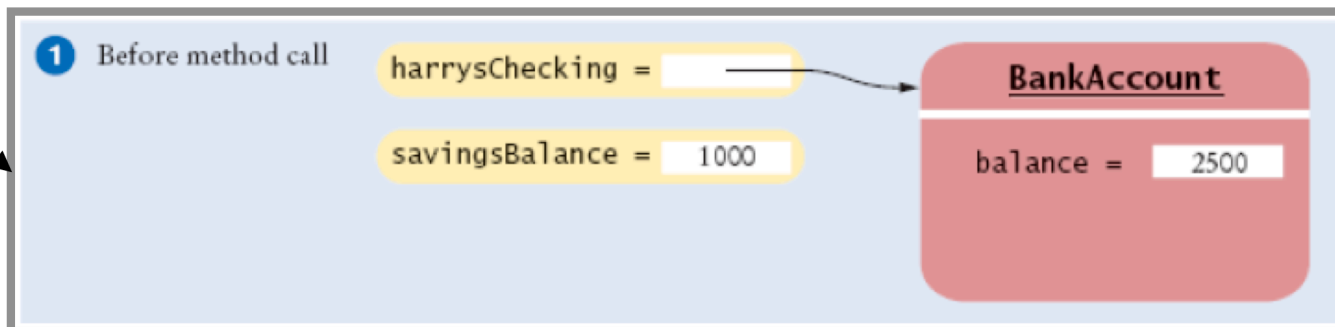
- Problems:
  - Message in English
  - Depends on `System.out`
- Best to decouple input/output from actual work of classes
- In general, try to minimize side effects beyond modification of implicit parameter



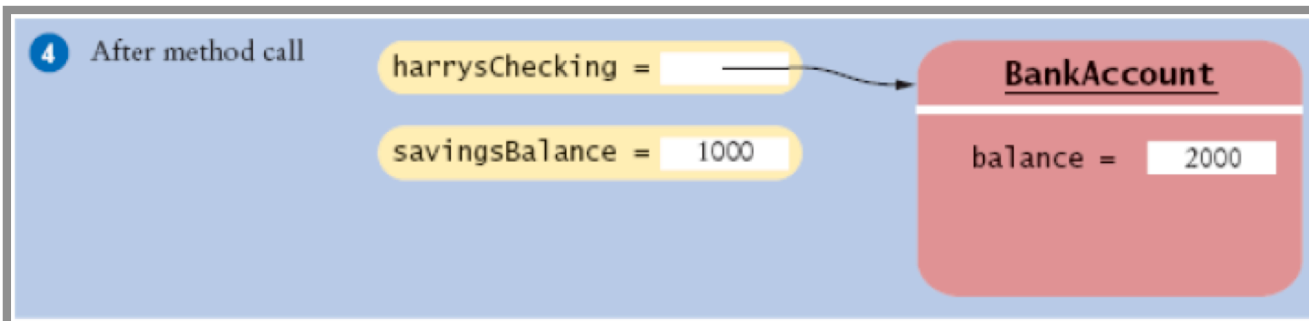
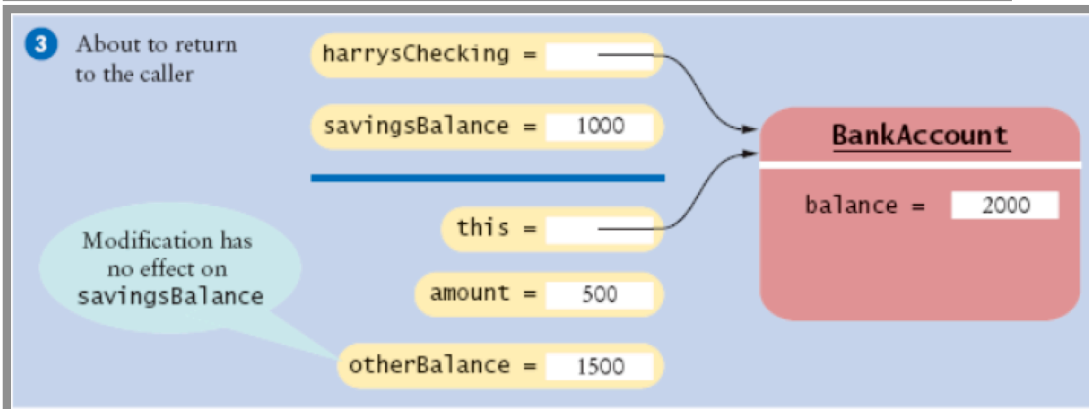
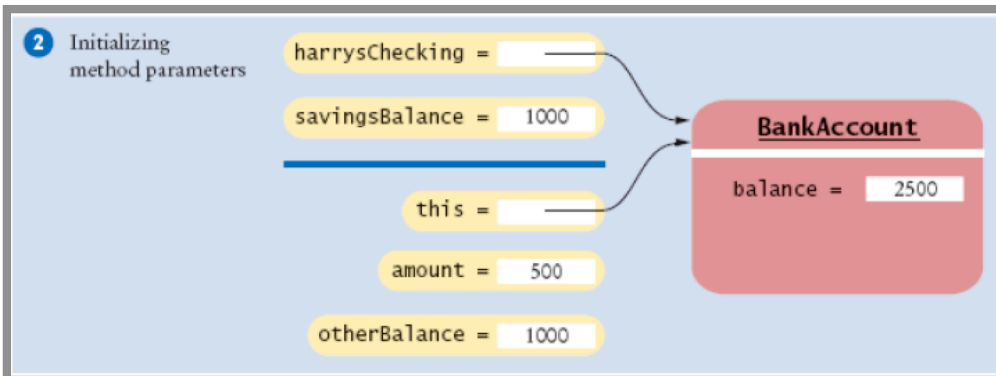
# Common Error: Trying to Modify Primitive Type Parameter

- Scenario (doesn't work):

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



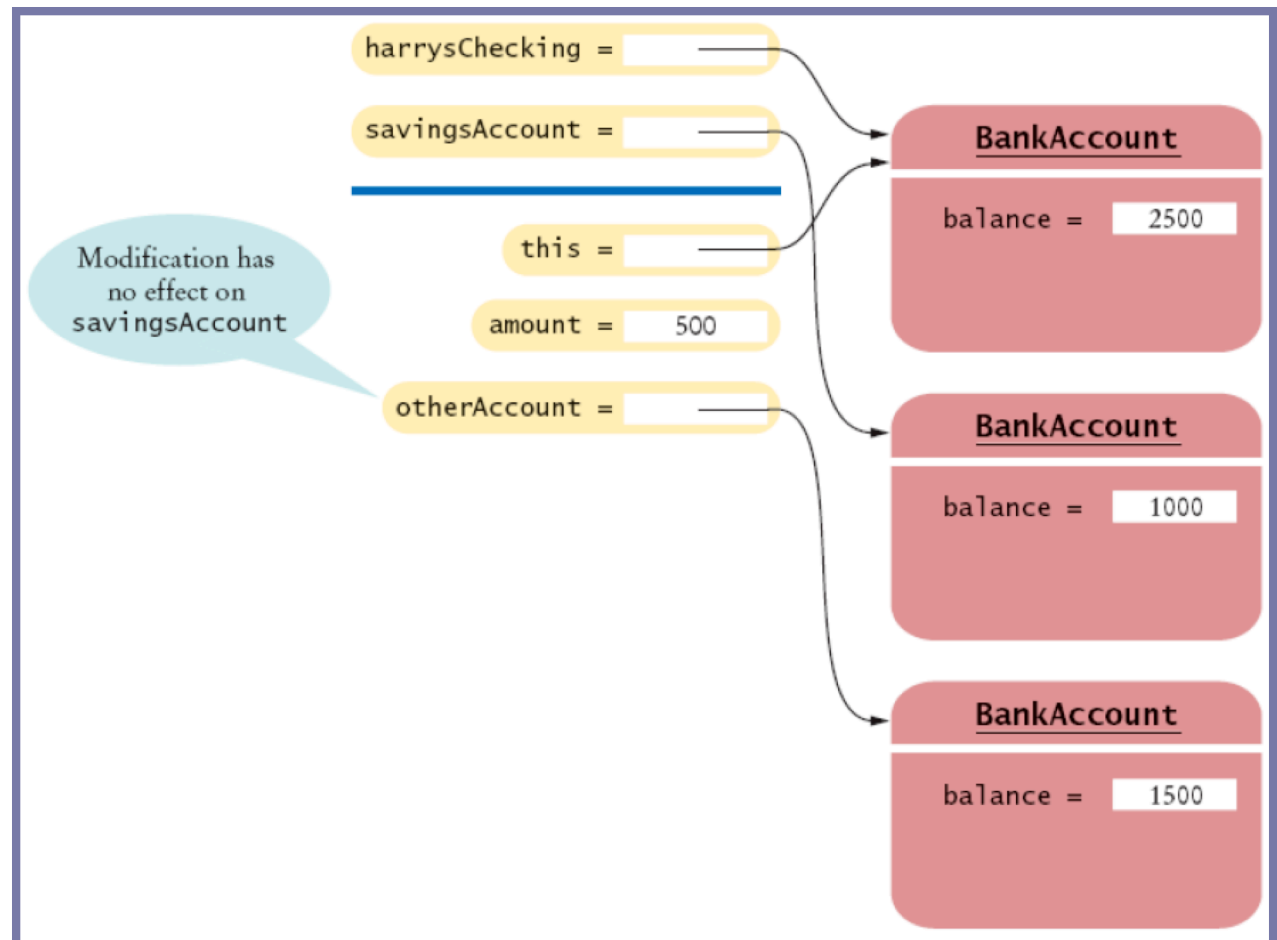
# Modifying Primitive Type Parameter has no Effect on Caller





# Call-by-Value Example

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





# Call-by-Value / 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 for both primitive types and object references
  - 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  
    }  
}
```



# Preconditions

- *Precondition*: Requirement that the caller of a method must meet
  - If precondition is violated, method is not responsible for computing correct result
- Good idea to document preconditions so that callers don't pass bad parameters
- Typical uses of preconditions
  - To restrict/constrain parameters of a method
  - To require that method is only called when object is in an appropriate state

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



# Checking Preconditions

- Method may skip check of the precondition (puts full trust/responsibility on caller)
  - Efficient, but dangerous if there is a violation
- May throw an exception (Ch. 15)
  - Inefficient - has to check every time
- May use an assertion check
  - Causes error if the assertion fails
  - After testing, can disable all assertion checks to allow program to run at full speed



# Assertions

- Syntax: `assert condition;`
- Logical condition in a program that you believe should be true

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

- By default, assertion checking is disabled when running Java programs
- To enable assertion checking:
  - `java -enableassertions MyProgramName`
  - Can use `-ea` as shortcut instead of `-enableassertions`



# Bad Way to Handle Violations

```
public void deposit( double amount ) {  
    if ( amount < 0 ) return;  
    balance = balance + amount;  
}
```

- Doesn't abort the program if precondition is violated
- But hard to debug if something is going wrong – nothing to tell you cause of the problem



# Postconditions

- Condition that is true after a method is completed
  - If method is called according to its preconditions, then it must ensure that its postconditions hold
- Two kinds of postconditions
  - Return value is computed correctly
  - Object is in a certain state after method call is completed

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



# Pre- and Postconditions

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

```
/** . . .  
    @return the account balance  
    (Postcondition: the return value equals the account balance  
    . . .
```

- State conditions in terms of public interface, not private fields

```
amount <= getBalance()  
// not account <= balance
```

- Pre- and postconditions spell out a contract between pieces of code



# Class Invariants

- Statement about an object that is true after every constructor and is preserved by every mutator (provide preconditions are met)

```
/**  
    A bank account has a balance that can be  
    changed by deposits and withdrawals  
    (Invariant: getBalance() >= 0)  
*/  
public class BankAccount { . . .
```

- Once you formulate a class invariant, check that the methods preserve it





# Static Methods

- In Java, every method must be defined within a class
  - Most methods operate on a particular instance of an object of that class (the 'implicit parameter')
  - Some methods are not invoked (called) on an object
- Example: `Math.sqrt( x )`
- Why?
  - Method does some computation that only needs numbers – numbers aren't objects, so can't call methods on them: `x.sqrt()` is not legal in Java ( `x` is a `double` )



# Static Methods (cont.)

```
public class Financial {  
    /** Computes a percentage of an amount.  
     * @param p the percentage to apply  
     * @param a the amount to which the percentage is applied  
     * @return p percent of a  
     */  
    public static double percentOf(double p, double a) {  
        return (p / 100) * a;  
    }  
    // More financial methods can be added here . . .  
}
```

- To call a static method, use class name, not an object  
`double tax = Financial.percentOf( taxRate, total );`
- `main` method is static because there are no objects when program first starts
  - Note: origin of term '*static*' is historical; better name would be *class methods*



# Static Fields

- A *static field* belongs to the class, not to any single object of the class
  - Also called a ‘class field’
  - Static field is shared by all instances of the same class

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



# Initializing Static Fields

- Three ways
  - Do nothing – will be initialized to default values (0 for numbers, false for boolean, null for objects)
  - Use an explicit initializer  
`private static int lastAssignedNumber = 1000;`
  - Use a static initialization block
    - Less common - Advanced topic 9.3



# Static Field Access

- Static fields should always be **private**
- Exception: Static constants, may be **public** to allow other classes to access them

```
public class BankAccount {  
    . . .  
    public static final double OVERDRAFT_FEE = 5.0;  
}
```

- Minimize the use of static fields (except static final fields - constants)



# Scope of Variables

- The *scope* of a variable: region of a program in which the variable can be accessed
- Local variable scope extends from point of declaration to end of enclosing block
- Scope of a local variable cannot contain definition of another variable with the same name
- Can have local variables with identical names if scopes do not overlap
  - Example: same variable name can be used in different methods - refers to different variables



# Scope of Class Members

- Members: fields and methods collectively
- Private members have *class scope*: can be accessed anywhere within the class
- Public members accessible by any code
  - From outside the class, must use *qualified name*
    - `Math.sqrt` or `other.balance`
  - Within the class, do not need to qualify field and method names
    - Refer automatically to `this` – the implicit parameter



# Overlapping Scope

```
public class Coin {  
    . . .  
    public Coin( double value, String name ) {  
        this.value = value;  
        this.name = name;  
    }  
  
    public double getExchangeValue( double exchRate ) {  
        double value; // local variable  
        . . .  
        return value;  
    }  
    . . .  
    private String name;  
    private double value;  
}
```

- Local variables *shadow* fields (instance variables) with the same name
- Shadowed fields can still be accessed by qualifying them explicitly with the `this` reference





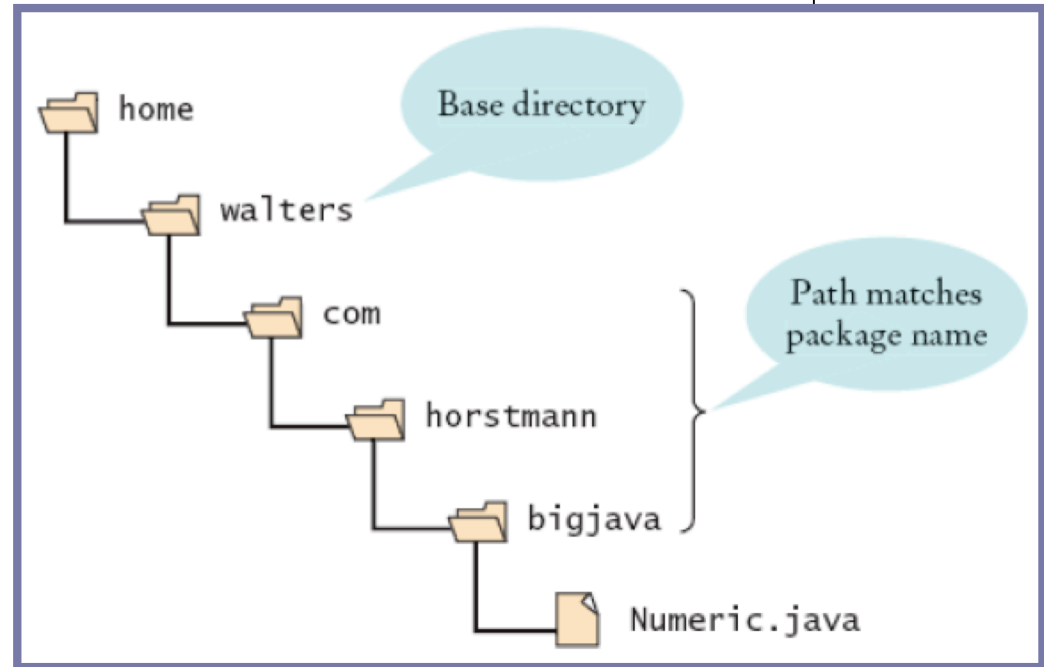
# Organizing Classes

- Large Java programs consist of many classes (10+, 100+, ...)
  - Problem: having all those files just in one directory
- *Package*: set of related Java classes
  - Structuring mechanism to organize files/classes
- All classes in a given package have line at the top of the file:  
`package packageName;`
  - Package names consist of one or more identifiers separated by periods
    - `com.horstmann.bigjava`



# Package Structure

- Java *package* and *class* names correspond to *directory* (folder) and *file* names



# Some Standard Library 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
omg.org.CORBA	Common Object Request Broker Architecture	IntHolder



# Importing Packages

- Can always use classes without ‘importing’ their package – just use fully qualified name

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

- But, gets tedious using qualified names
- Importing the package allows you to just use the class name

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

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

- Shortcut to import all classes in a package:

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

- Don't need to import `java.lang` or other classes in the same package