Lập trình hướng đối tượng

Biên soạn: Nguyễn Thị Tuyết Hải

Email: tuyethai@ptithcm.edu.vn

Liên lạc với giảng viên

Email: tuyethai@ptithcm.edu.vn

Tiêu đề (Subject): **[Java-02]**

Email không ghi rõ tiêu đề trên sẽ được bộ lọc tự động xoá

Chapter 4: Objects and Classes

4.1 Introduction to Object-Oriented Programming

- An object-oriented program is made of **objects**. Each object has a **specific functionality**, exposed to its users, and a **hidden implementation**.
  - “off-the-shelf” from a library
  - custom designed
- Traditional structured programming: a set of procedures (or algorithms), small problems
  - algorithms first
  - data structures second
- OOP reverses the order: larger problems
  - data first
  - algorithms to operate on the data

Chapter 4: Objects and Classes

4.1 Introduction to Object-Oriented Programming

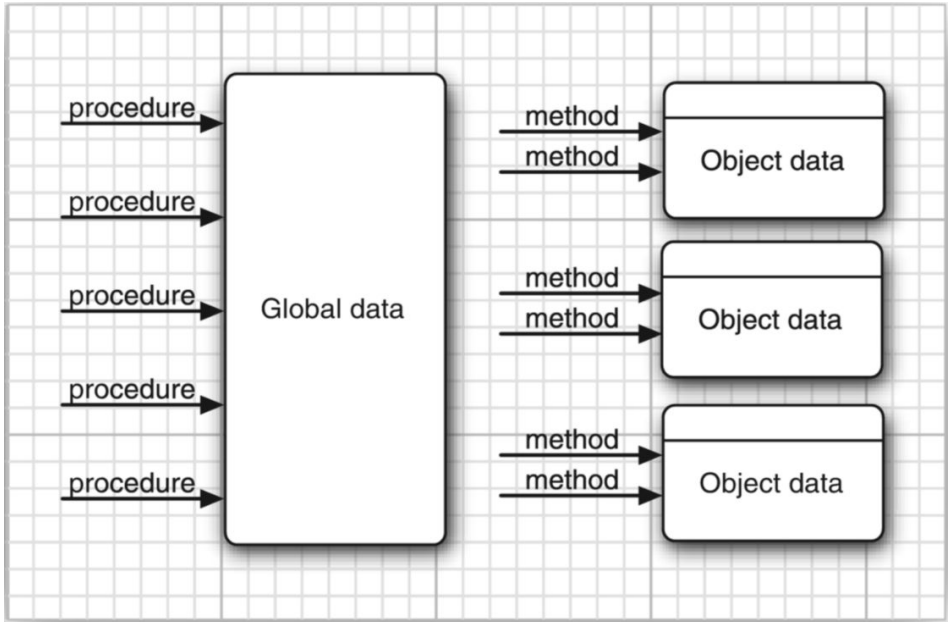


Figure 4.1 Procedural vs. OO programming

Chapter 4: Objects and Classes

4.1.1 Classes

- a class is the **template** from which objects are made.
- constructing **an object** from a class = creating **an instance of the class**.

Chapter 4: Objects and Classes

4.1.1 Classes

- Encapsulation:
  - **combining** data and behavior in one data package, **hiding** the implementation details from the users
  - data as **instance fields**; behavior as **methods**
  - a specific **object** (an **instance** of a class) have specific values of its instance fields: as current **state** of the object
  - programs should interact with object data only through the object's methods
- All Java classes extend the class named **Object**: the new class has all the properties and methods of **Object**

Chapter 4: Objects and Classes

4.1.2 Objects

Three key characteristics of objects:

- **object's behavior**—what can you do with this object?
- **object's state**—how does the object react when you invoke those methods? (information about what it currently looks like)
- object's identity—how is the object distinguished from others that may have the same behavior and state?

Chapter 4: Objects and Classes

4.1.3 Identifying Classes

- Simple rule: identifying **classes** is to look for **nouns** in the problem analysis. **Methods**, on the other hand, correspond to **verbs**
- E.g., in an order-processing system, some of the nouns are
  - Item
  - Order
  - Shipping address
  - Payment
  - Account

  => The classes Item, Order, …

  Items are added to orders.
  Orders are shipped or canceled.
  Payments are applied to orders.
  => With each verb, such as “add,” “ship,” “cancel,” or “apply,”

Chapter 4: Objects and Classes

4.1.4 Relationships between Classes

- The most common relationships between classes are
  - Dependence (“uses–a”)
  - Aggregation (“has–a”)
  - Inheritance (“is–a”)

Chapter 4: Objects and Classes

4.1.4 Relationships between Classes

- Dependence (“uses–a”) or coupling in software engineering: a class depends on another class if its methods use or manipulate objects of that class.
  For example,
  The Order class uses the Account class because Order objects need to access Account objects to check for credit status.

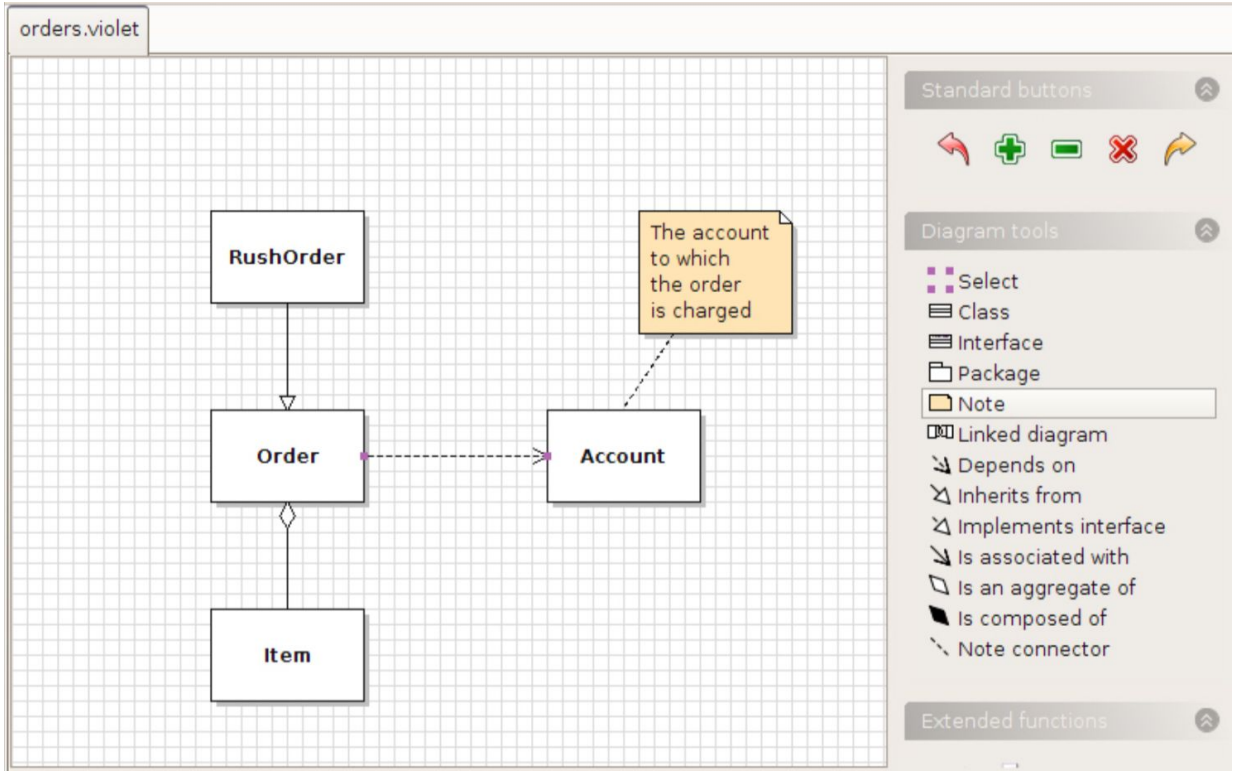
Chapter 4: Objects and Classes

4.1.4 Relationships between Classes

- Aggregation (“has–a”): objects of class A contain objects of class B.
  - For example, an Order object contains Item objects.
- Inheritance (“is–a”): a more special and a more general class.
  - If class A extends class B, class A inherits methods from class B but has more capabilities.
  - For example, a RushOrder class inherits from an Order class.
  - The specialized RushOrder class has
    - special methods for priority handling
    - a different method for computing shipping charges
    - other methods inherited from the Order class: adding items and billing

Chapter 4: Objects and Classes

4.1.4 Relationships between Classes



Chapter 4: Objects and Classes

4.2.1 Objects and Object Variables

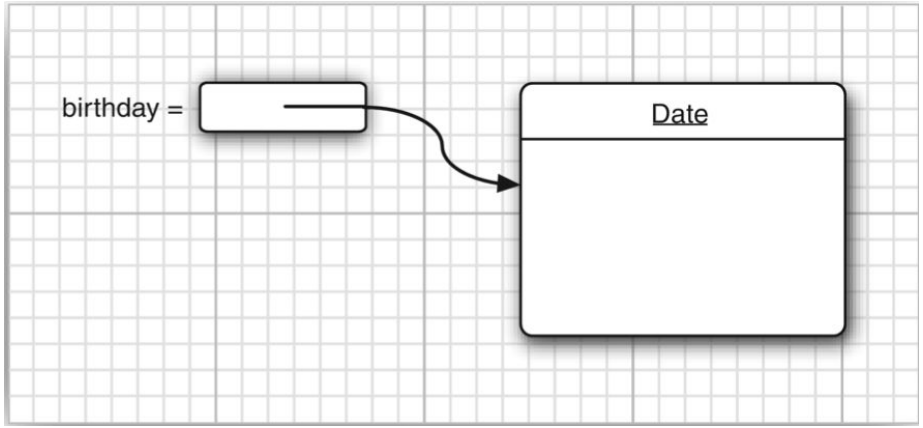
- Declare object Variables:

  Date birthday; // doesn't refer to any object

- Initialize objects using constructors

  Date birthday = new Date();

- Constructors have the same name as the class name



Chapter 4: Objects and Classes

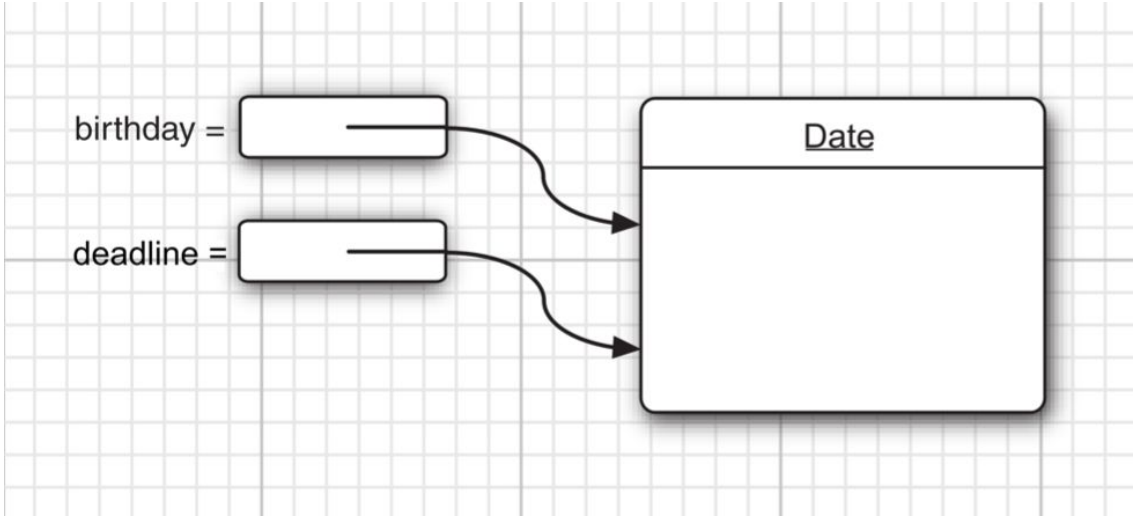
4.2.1 Objects and Object Variables

- String representation of the date:

  birthday.toString();

- Assignment

  deadline = birthday;



Chapter 4: Objects and Classes

4.2.2 The LocalDate Class of the Java Library

- Date class: represents a point in time
- LocalDate class: expresses days in the familiar calendar notation
- constructs a new object that represents the date at which the object was constructed: `LocalDate.now()`
- construct an object for a specific date: `LocalDate.of(1999, 12, 31)`

```
LocalDate newYearsEve = LocalDate.of(1999, 12, 31);
```

- methods

```
int year = newYearsEve.getYear(); // 1999
int month = newYearsEve.getMonthValue(); // 12
int day = newYearsEve.getDayOfMonth(); // 31

LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
```

Chapter 4: Objects and Classes

4.2.3 Mutator and Accessor Methods

- mutator methods: after invoking it, the state of the object has changed.
- accessor methods: only access objects without modifying them: e.g., date.getMonthValue(), date.getDayOfWeek(), …

  `Code: CalendarTest.java`

```
Mon Tue Wed Thu Fri Sat Sun
                            1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26* 27  28  29
 30
```

Chapter 4: Objects and Classes

4.3 Defining Your Own Classes
4.3.1 An Employee Class

```
class ClassName {
        field1
        field2
        ...
        constructor1
        constructor2
        ...
        method1
        method2
        ...
}
```

```
32  class Employee
33  {
34     private String name;
35     private double salary;
36     private LocalDate hireDay;
37
38     public Employee(String n, double s, int year, int month, int day)
39     {
40        name = n;
41        salary = s;
42        hireDay = LocalDate.of(year, month, day);
43     }
44
45     public String getName()
46     {
47        return name;
48     }
49
50     public double getSalary()
51     {
52        return salary;
53     }
54
55     public LocalDate getHireDay()
56     {
57        return hireDay;
58     }
59
60     public void raiseSalary(double byPercent)
61     {
62        double raise = salary * byPercent / 100;
63        salary += raise;
64     }
65  }
```

Chapter 4: Objects and Classes

```
11 public class EmployeeTest
12 {
13    public static void main(String[] args)
14    {
15       // fill the staff array with three Employee objects
16       Employee[] staff = new Employee[3];
17
18       staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
19       staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
20       staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
21
22       // raise everyone's salary by 5%
23       for (Employee e : staff)
24          e.raiseSalary(5);
25
26       // print out information about all Employee objects
27       for (Employee e : staff)
28          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
29                + e.getHireDay());
30    }
31 }
```

Chapter 4: Objects and Classes

4.3.2 Use of Multiple Source Files

Put *.class in the same folder as *.java (e.g., Employee.java, EmployeeTest.java), two ways of compile:

- javac Employee*.java
- **javac EmployeeTest.java**

Chapter 4: Objects and Classes

4.3.4 First Steps with Constructors

- A constructor has the same name as the class.
- A class can have more than one constructor.
- A constructor can take zero, one, or more parameters.
- A constructor has no return value.

```
32 class Employee
33 {
34    private String name;
35    private double salary;
36    private LocalDate hireDay;
37
38    public Employee(String n, double s, int year, int month, int day)
39    {
40       name = n;
41       salary = s;
42       hireDay = LocalDate.of(year, month, day);
43    }
44
45    public String getName()
46    {
47       return name;
48    }
49
50    public double getSalary()
51    {
52       return salary;
53    }
54
55    public LocalDate getHireDay()
56    {
57       return hireDay;
58    }
59
60    public void raiseSalary(double byPercent)
61    {
62       double raise = salary * byPercent / 100;
63       salary += raise;
64    }
65 }
```

Chapter 4: Objects and Classes

4.3.4 First Steps with Constructors

- A constructor is always called with the **new** operator for initializing fields of an instance.

Employee staff_0 = new Employee("Carl Cracker", 75000, 1987, 12, 15);

```
32  class Employee
33  {
34     private String name;
35     private double salary;
36     private LocalDate hireDay;
37
38     public Employee(String n, double s, int year, int month, int day)
39     {
40        name = n;
41        salary = s;
42        hireDay = LocalDate.of(year, month, day);
43     }
44
45     public String getName()
46     {
47        return name;
48     }
49
50     public double getSalary()
51     {
52        return salary;
53     }
54
55     public LocalDate getHireDay()
56     {
57        return hireDay;
58     }
59
60     public void raiseSalary(double byPercent)
61     {
62        double raise = salary * byPercent / 100;
63        salary += raise;
64     }
65  }
```

Chapter 4: Objects and Classes

4.3.5 Implicit and Explicit Parameters

- Methods operate on objects and access their instance fields.

```
public void raiseSalary(double byPercent){
   double raise = salary * byPercent / 100;  //salary: implicit, byPercent: explicit
   salary += raise;
}
```

- Consider the call

```
Employee staff_0 = new Employee("Carl Cracker", 75000, 1987, 12, 15);
staff_0.raiseSalary(5); // byPercent = 5
```

- Detailed execution

```
double raise = staff_0.salary * 5 / 100;
staff_0.salary += raise;
```

- "this" keyword

```
public void raiseSalary(double byPercent){
   double raise = this.salary * byPercent / 100;  //salary: implicit, byPercent: explicit
   this.salary += raise;
}
```

Chapter 4: Objects and Classes

4.3.6 Benefits of Encapsulation

- Field accessors simply return the values of instance fields
- Why don't set all data fields as **public** (instead of **private**)?
  - name, hireDay: read-only
  - salary: only be changed by the raiseSalary method
- Accessors and mutators
  - A private data field;
  - A public field accessor method;
  - A public field mutator method.

```
32  class Employee
33  {
34     private String name;
35     private double salary;
36     private LocalDate hireDay;
37
38     public Employee(String n, double s, int year, int month, int day)
39     {
40        name = n;
41        salary = s;
42        hireDay = LocalDate.of(year, month, day);
43     }
44
45     public String getName()
46     {
47        return name;
48     }
49
50     public double getSalary()
51     {
52        return salary;
53     }
54
55     public LocalDate getHireDay()
56     {
57        return hireDay;
58     }
59
60     public void raiseSalary(double byPercent)
61     {
62        double raise = salary * byPercent / 100;
63        salary += raise;
64     }
65  }
```

Chapter 4: Objects and Classes

4.3.6 Benefits of Encapsulation

- easily change the internal implementation
- E.g.,
  - if the storage of the name is changed to
    `String firstName;`
    `String lastName;`
  - getName method can be changed to

`return firstName + " " + lastName`

- mutators perform error checking

```
32  class Employee
33  {
34     private String name;
35     private double salary;
36     private LocalDate hireDay;
37
38     public Employee(String n, double s, int year, int month, int day)
39     {
40        name = n;
41        salary = s;
42        hireDay = LocalDate.of(year, month, day);
43     }
44
45     public String getName()
46     {
47        return name;
48     }
49
50     public double getSalary()
51     {
52        return salary;
53     }
54
55     public LocalDate getHireDay()
56     {
57        return hireDay;
58     }
59
60     public void raiseSalary(double byPercent)
61     {
62        double raise = salary * byPercent / 100;
63        salary += raise;
64     }
65  }
```

Chapter 4: Objects and Classes

4.3.6 Benefits of Encapsulation

!!! Don't to write accessor methods that return references to mutable objects.

hireDay: read-only, but …

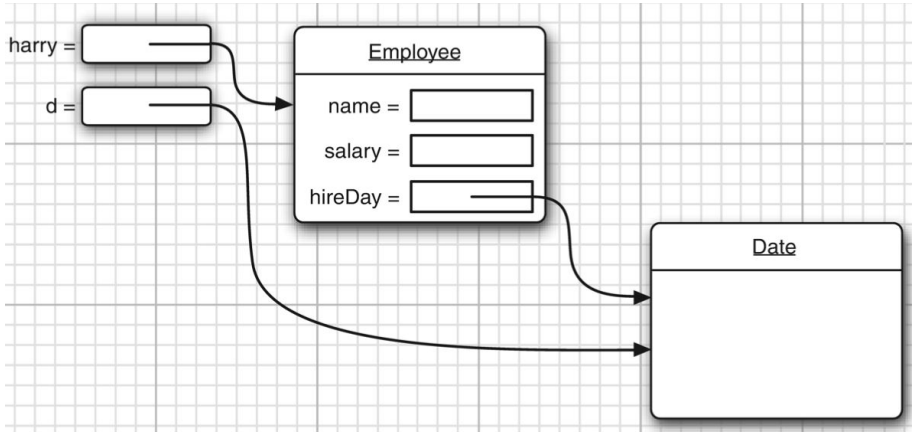
```
38 class Employee
39 {
40    private String name;
41    private double salary;
42    private Date hireDay;
43
44    public Employee(String n, double s, int year, int month, int day)
45    {
46       name = n;
47       salary = s;
48       hireDay = new Date(year, month, day);
49    }
50
51    public String getName()
52    {
53       return name;
54    }
55
56    public double getSalary()
57    {
58       return salary;
59    }
60
61    public Date getHireDay()
62    {
63       return hireDay;
64    }
65
66    public void raiseSalary(double byPercent)
67    {
68       double raise = salary * byPercent / 100;
69       salary += raise;
70    }
71 }
```

Chapter 4: Objects and Classes

4.3.6 Benefits of Encapsulation

!!! Don’t to write accessor methods that return references to mutable objects.

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Date d = harry.getHireDay(); // Date (1989/10/1), mutable LocalDate
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds);
// let's give Harry ten years of added seniority
```



Chapter 4: Objects and Classes

4.3.6 Benefits of Encapsulation

!!! Don’t to write accessor methods that return references to mutable objects.

If we need to return a reference to a mutable object => **clone it first**.

```
class Employee {
     ...
     public Date getHireDay() {
          return (Date) hireDay.clone(); // Ok
     }
     ...
}
```

Chapter 4: Objects and Classes

4.3.8 Private Methods

- most methods are public, private methods are useful in some cases.
  - break up the code for a computation into separate helper methods
  - such methods are best implemented as **private**

4.3.9 Final Instance Fields

- an instance field can be defined as **final** => such field must be initialized when the object is constructed.
- afterwards, the field may not be modified again.

```
class Employee {
    private final String name;
    ...
}
```

Chapter 4: Objects and Classes

4.4 Static Fields and Methods

4.4.1 Static Fields

If a field is defined as static, then there is only one such field per class.

```
class Employee {
     private static int nextId = 1;
     private int id;
     ...
     public void setId() {
          id = nextId;
          nextId++;
     }
}
```

Every employee object now has its own id field, but there is only one nextId field that is shared among all instances of the class.

Chapter 4: Objects and Classes

4.4.2 Static Constants

Static variables are quite rare but static constants are more common.

```
public class Math {
    …
    public static final double PI = 3.14159265358979323846;
    …
}
```

Access: `Math.PI`

Chapter 4: Objects and Classes

4.4.3 Static Methods

- Static methods are methods that do not operate on objects. Or they don't have any implicit parameters.
- Call static methods without having any objects

E.g. `Math.pow(x, a)`

```
class Employee {
    private static int nextId = 1;
    private int id;
    ...

    public void setId() {
        id = nextId;
        nextId++;
    }

    public static int getNextId() {
        return nextId;
        // returns static field
    }
}

int n = Employee.getNextId();
```

Chapter 4: Objects and Classes

4.4.4 Factory Methods

- **static** factory methods that construct objects

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

- vary the type of the constructed object, e.g., as a subclass object

Chapter 4: Objects and Classes

4.4.5 The main method

- Every class can have a main method which is helpful for unit testing

```
public class Application {
     public static void main(String[] args) {
     // construct objects here
     ... }
}
```

Code: v1ch04.StaticTest

Chapter 4: Objects and Classes

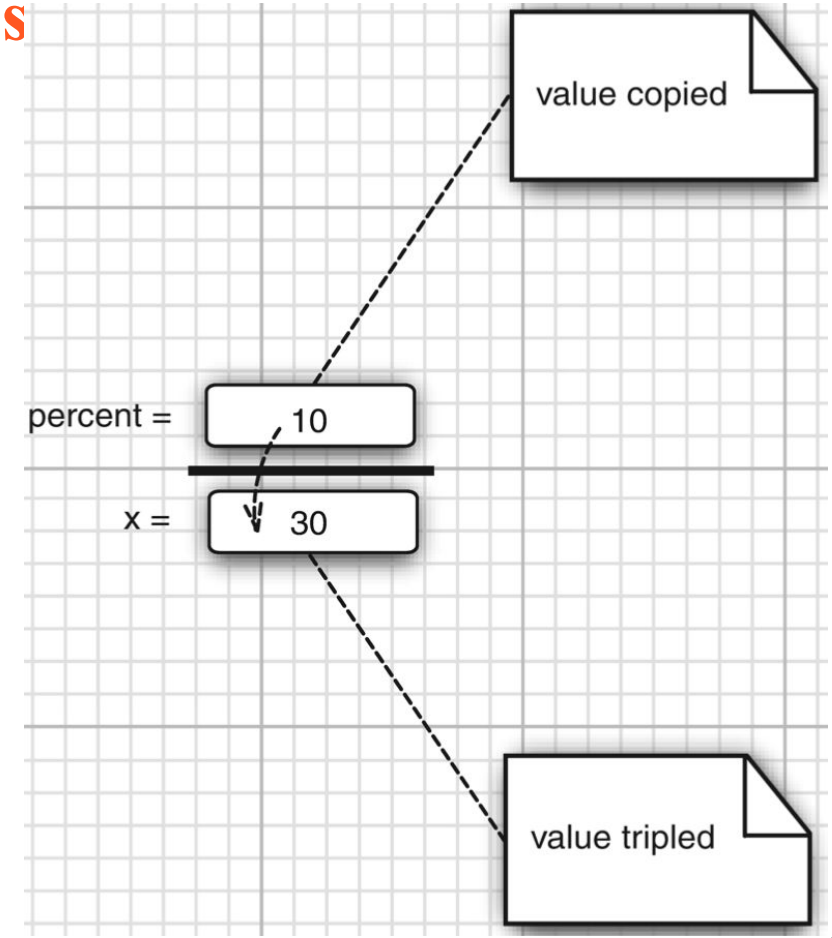
4.5 Method Parameters

Two kinds of method parameters:

- **Primitive types (numbers, boolean values) => call by value**
- Object references => call by reference

```
public static void tripleValue(double x){
    x=3*x;
}

double percent = 10;
tripleValue(percent);
// percent is still 10
```



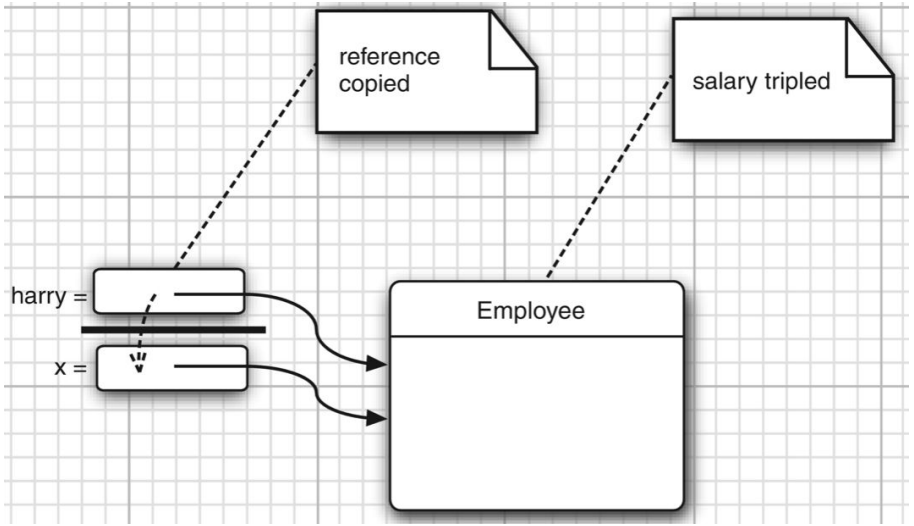
Chapter 4: Objects and Classes

4.5 Method Parameters

Two kinds of method parameters:

- Primitive types (numbers, boolean values) => call by value
- **Object references => call by reference**

```
public static void tripleSalary(Employee x){
     x.raiseSalary(200);
}
harry = new Employee(. . .);
tripleSalary(harry);
```



Code: v1ch04.ParamTest

Chapter 4: Objects and Classes

4.6 Object Construction

4.6.1 Overloading

- Some classes have more than one constructor.
- Overloading: methods have same names but different parameters.

4.6.2 Default Field Initialization

- A field is automatically set to a default value (numbers to 0, boolean values to false, and object references to null) if a field is not explicitly in a constructor,

4.6.3 The Constructor with No Arguments

- Many classes contain a constructor with no arguments that creates an object whose state is set to an appropriate default.

```
public Employee() {
     name = "";
     salary = 0;
     hireDay = LocalDate.now();
}
```

Chapter 4: Objects and Classes

4.6.4 Explicit Field Initialization

- simply assign a value to any field in the class definition.
- the assignment is carried out before the constructor executes.
- the initialization value doesn't have to be a constant value.

```
class Employee {
    private String name = "";
    ...
}
```

```
class Employee {
    private static int nextId;
    private int id = assignId();
    ...
    private static int assignId() {
        int r = nextId; nextId++;
        return r;
    }
...
}
```

Chapter 4: Objects and Classes

4.6.5 Parameter Names

- single-letter parameter names
- prefix each parameter with an “a”
- the same name, using “this” with the implicit parameter

```
public Employee(String n, double s) {
     name = n;
     salary = s;
}

public Employee(String aName, double
aSalary) {
     name = aName;
     salary = aSalary;
}

public Employee(String name, double salary)
{
     this.name = name;
     this.salary = salary;
}
```

Chapter 4: Objects and Classes

4.6.6 Calling Another Constructor

- calls another constructor of the same class

```
public Employee(String name, double salary) {
      this.name = name;
      this.salary = salary;
}

public Employee(double s) {
      // calls Employee(String, double)
      this("Employee #" + nextId, s);
      nextId++;
}
```

Chapter 4: Objects and Classes

4.6.7 Initialization Blocks

- Two ways to initialize a data field:
  - By setting a value in a constructor
  - By assigning a value in the declaration
- Class declarations can contain arbitrary blocks of code.
- These blocks are executed whenever an object of that class is constructed.
- The initialization block runs first, and then the body of the constructor is executed.

```
class Employee {
     private static int nextId;
     private int id; private String
     name; private double salary;
     // object initialization block
     {
          nextId++;
     }
     public Employee(String n, double s)
     {
          name = n;
          salary = s;
     }
     …
}
```

Chapter 4: Objects and Classes

4.6.7 Initialization Blocks

The construction process

- All data fields are initialized to their default values (0, false, or null).
- All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
- If the first line of the constructor calls a second constructor, then the body of the second constructor is executed.
- The body of the constructor is executed.

Chapter 4: Objects and Classes

4.6.7 Initialization Blocks

Static field:

- supply an initial value

```
private static int nextId = 1;
```

- use a static initialization block that occurs when the class is **first** loaded.

```
// static initialization block
static {
     Random generator = new Random();
     nextId = generator.nextInt(10000);
}
```

All codes: v1ch04.ConstructorTest

Chapter 4: Objects and Classes

4.6.8 Object Destruction and the finalize Method

- Java does automatic garbage collection, so Java does not support destructors
- The resource (file, system resource, ect.) need be reclaimed and recycled when it is no longer needed: **finalize** method
- The finalize method is called **before the garbage collector**
- If a resource needs to be closed when you have finished using it: do it manually
  - close
  - try - with - resource statement

Chapter 4: Objects and Classes

4.7 Packages

- Classes can be grouped in a collection called a **package.**
- Package is good
  - for organizing your work
  - for separating your work from code libraries provided by others
  - guarantee the uniqueness of class names

Chapter 4: Objects and Classes

4.7.1 Class Importation

A class can use:

- all classes from its own package
- all public classes from other packages.
  - add the full package name in front of every class name
  - import statement

```
java.time.LocalDate today =
java.time.LocalDate.now();
```

```
import java.util.*;
LocalDate today = LocalDate.now();
import java.time.LocalDate;
```

Chapter 4: Objects and Classes

4.7.2 Static Imports

- import statement permits the importing of static methods and fields
  E.g. we can use the static methods and fields of the System class without the class name prefix:

```
import static java.lang.System.*;
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit
```

Chapter 4: Objects and Classes

4.7.3 Addition of a Class into a Package

- To place classes inside a package, you must put the name of the package **at the top of your source file**

```
package com.horstmann.corejava;
public class Employee {
     ...
}
```

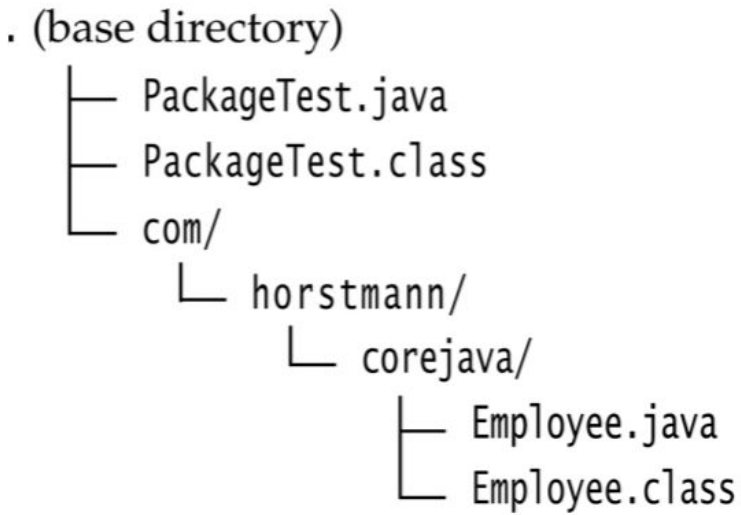
- If there is no a package statement in the source file, then its classes belong to the **default package**.

Chapter 4: Objects and Classes

4.7.3 Addition of a Class into a Package

- Place source files into a **subdirectory** that **matches the full package name**.
  E.g., all source files in the **com.horstmann.corejava** package should be in a subdirectory **com/horstmann/corejava**

- Code example: v1ch04.PackageTest



Chapter 4: Objects and Classes

4.7.4 Package Scope

- Access modifiers: public, private
  - public features (i.e., the class, method, or variable) can be used by any class
  - private features can be used only by the class that defines them
  - defaut: features accessed by all methods in the same package

  Code example: v1ch04.EmployeeTest

Chapter 4: Objects and Classes

4.8 The Class Path

- Class files and subdirectories can also be stored in a JAR (Java archive) file
- To share classes among programs, you need to do the following:
  - Place your class files inside a directory
  - Place any JAR files inside a directory
  - Set the class path: the collection of all locations that can contain class files.
    - In LINUX: **/home/user/classdir:.:/home/user/archives/archive.jar**
    - In Windows: **c:\classdir;.;c:\archives\archive.jar**

    - The base directory /home/user/classdir or c:\classdir;
    - The current directory (.);
    - The JAR file /home/user/archives/archive.jar or c:\archives\archive.jar.

Chapter 4: Objects and Classes

4.8 The Class Path

- With class path as **/home/user/classdir:.:/home/user/archives/archive.jar**

Suppose **JVM** searches for the class file of the **com.horstmann.corejava.Employee** class, JVM looks in:

1. **the system class files** that are stored in archives in the **jre/lib** and **jre/lib/ext** directories
2. **the class path**
   a. home/user/classdir/com/horstmann/corejava/Employee.class
   b. com/horstmann/corejava/Employee.class starting from the current directory • com/horstmann/corejava/Employee.class inside/home/user/archives/archive.jar

Chapter 4: Objects and Classes

4.8 The Class Path

If we refer to a class without specifying its package:

- the compiler first needs to find out the package that contains the class

```
import java.util.*;
import com.horstmann.corejava.*;
public class EmployeeTest{
      public static void main(String[] args){
            staff_0 = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      }
}
```

- it then tries to find **java.lang.Employee** (java.lang package is imported by default), **java.util.Employee**, **com.horstmann.corejava.Employee**, and **Employee** in the current package.
- it searches for all of the locations of the class path.
- it even looks at the source files to check whether the source is newer than the class file

Chapter 4: Objects and Classes

4.8.1 Setting the Class Path

- classpath option

Linux: java -classpath /home/user/classdir:.:/home/user/archives/archive.jar MyProg

Windows: java -classpath c:\classdir;.;c:\archives\archive.jar MyProg

- CLASSPATH environment variable

Linux: export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar

Windows: set CLASSPATH=c:\classdir;.;c:\archives\archive.jar

!!! The class path is set until the shell exits.

Chapter 4: Objects and Classes

4.8.1 Setting the Class Path

- set the CLASSPATH environment variable permanently: do it **carefully** since many programmers forget the global setting, and are surprised when their classes are not loaded properly.

Chapter 4: Objects and Classes

4.9 Documentation Comments: Javadoc generates HTML documentation from your source files

4.9.1 Comment Insertion

The javadoc utility extracts information for:

- Packages
- Public classes and interfaces
- Public and protected fields
- Public and protected constructors and methods

Each comment is placed immediately **above** the feature it describes.

A comment starts with a /** and ends with a */.

Chapter 4: Objects and Classes

4.9 Documentation Comments: Javadoc generates HTML documentation from your source files

4.9.1 Comment Insertion

A comment starts with a /** and ends with a */.

Each comment contains free-form text

- The first sentence as summary statement
- …
- **tags** that starts with an @, such as @author or @param

4.9.2 Class Comments

Chapter 4: Objects and Classes

4.9.2 Class Comments

```
/**
* A {@code Card} object represents a playing card, such
* as "Queen of Hearts". A card has a suit (Diamond, Heart,
* Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack, * 12 = Queen, 13 = King)
*/
public class Card {
...
}
```

OR

```
/**
A {@code Card} object represents a playing card, such
as "Queen of Hearts". A card has a suit (Diamond, Heart,
Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack, * 12 = Queen, 13 = King)
*/
```

Chapter 4: Objects and Classes

4.9.3 Method Comments

Each method comment must immediately **precede the method** that it describes.

Some tags:

- @param variable description
- @return description
- @throws class description

```
/**
* Raises the salary of an employee.
* @param byPercent the percentage by which to raise the salary (e.g. 10 means 10%)
* @return the amount of the raise
*/
public double raiseSalary(double byPercent) {
     double raise = salary * byPercent / 100;
     salary += raise;
     return raise;
}
```

Chapter 4: Objects and Classes

4.9.4 Field Comments

Only need to document public fields - generally that means static constants.

```
/**
* The "Hearts" card suit
*/
public static final int HEARTS = 1;
```

Chapter 4: Objects and Classes

4.9.5 General Comments

The tags can be used in class documentation comments:

- `@author` name: we can have multiple @author tags, one for each author.
- `@version` text: the text can be any description of the current version.

Chapter 4: Objects and Classes

4.9.5 General Comments

The following tags can be used in all documentation comments:

- `@since` text: any description of the version that introduced this feature.
  E.g., `@since version 1.7.1.`
- `@deprecated` text
  This tag adds a comment that the class, method, or variable should no longer be used.
  The text should suggest a replacement.
  E.g., `@deprecated Use <code>setVisible(true)</code> instead`
- `@see` reference: adds a hyperlink in the "see also" section.
  E.g., `@see com.horstmann.corejava.Employee#raiseSalary(double)`
  `@see <a href="www.horstmann.com/corejava.html">The Core Java home page</a>`

Chapter 4: Objects and Classes

4.9.6 Package and Overview Comments

- to generate package comments, we need to add a separate file in each package directory.
  - package.html: all text between the tags <body>...</body> is extracted.
  - package-info.java: contains an initial Javadoc comment, delimited with /** and */, followed by a package statement. It should contain no further code or comments.

Chapter 4: Objects and Classes

4.9.7 Comment Extraction

- docDirectory: the directory where you want the HTML files to go
  - Change to the directory that contains the source files.
  - Run the command:

```
javadoc -d docDirectory nameOfPackage1 nameOfPackage2 ...
javadoc -d docDirectory *.java
```

Chapter 4: Objects and Classes

4.10 Class Design Hints

- Always keep data private: use an accessor or mutator method
- Always initialize data: initialize all variables explicitly, either by supplying a default or by setting defaults in all constructors.
- Don’t use too many basic types in a class: the idea is to replace multiple related uses of basic types with other classes

```
private String street;
private String city;
private String state;
private int zip;
=> Address
```

Chapter 4: Objects and Classes

4.10 Class Design Hints

- Not all fields need individual field accessors and mutators.
- Break up classes that have too many responsibilities.
- Make the names of your classes and methods reflect their responsibilities.
  - A class name should be a noun (Order), or a noun preceded by an adjective (RushOrder) or a gerund (an “-ing” word, like BillingAddress)
  - Accessor methods begin with a lowercase get (getSalary) and mutator methods use a lowercase set (setSalary).
- Prefer immutable classes

Chapter 5: Inheritance

Chapter 6: Interfaces, Lambda Expressions, and Inner Classes

Chapter 7: Exceptions, Assertions, and Logging

Chapter 8: Generic Programming

Chapter 9: Collections