Lecture 3:
Object Oriented Programming II

Object Creation

class Body {
    private long idNum;
    private String name = “empty”;
    private Body orbit;
    private static long nextID = 0;
}

Body sun = new Body();

- An object is created by the new method
- The runtime system will allocate enough memory to store the new object
- If no enough space, the automatic garbage collector will reclaim space from other no longer used objects. If there is still no enough space, then an OutOfMemoryError exception will be thrown
- No need to delete explicitly
Constructor

- constructor is a way to initialize an object before the reference to the object is returned by `new`
- has the same name as the class
- can have any of the same access modifiers as class members
- similar to methods. A class can have multiple constructors as long as they have different parameter list. Constructors have NO return type.
- Constructors with no arguments are called no-arg constructors.
- If no constructor is provided explicitly by the programmer, then the language provides a default no-arg constructor which sets all the fields which has no initialization to be their default values. It has the same accessibility as its class.

Sample Class and Constructors

class Body {
    private long idNum;
    private String name = "empty";
    private Body orbits;
    private static long nextID = 0;

    Body() {
        idNum = nextID++;
    }

    Body(String bodyName, Body orbitsAround) {
        this();
        name = bodyName;
        orbits = orbitsAround;
    }
}

Assume no any Body object is constructed before:

Body sun = new Body();  
Body earth = new Body("Earth", sun);

Assume no any Body object is constructed before:

Body sun = new Body("Sol", null);

Diagram:

```
+-------------------+        +-------------------+
| idNum: 0          |        | idNum: 1          |
| name: empty       |        | name: Earth       |
| orbits: null      |        | orbits: sun       |
| nextID = 1        |        | nextID = 2        |
```
Usage of *this*

- inside a constructor, you can use *this* to invoke another constructor in the same class. This is called *explicit constructor invocation*. It **MUST** be the first statement in the constructor body if exists.

- *this* can also be used as a reference of the current object. It **CANNOT** be used in a static method

Example: usage of *this* as a reference of the current object

```java
class Body {
    private long idNum;
    private String name;
    private Body orbits;
    private static long nextID = 0;
    private static LinkedList bodyList = new LinkedList();
    
    Body(String name, Body orbits) {
        this.name = name;
        this.orbits = orbits;
    }
    
    private void inQueue() {
        bodyList.add(this);
    }
}
```
Other initialization methods(1)

- Initialization block
  - a block of statements to initialize the fields of the object
  - outside of any member or constructor declaration
  - they are executed BEFORE the body of the constructors!

Other initialization methods(2)

- Static initialization block
  - Resembles a non-static initialization block except that it is declared static, can only refer to static members and cannot throw any checked exceptions
  - Gets executed when the class is first loaded

Example

class Primes {
    static int[] primes = new int[4];
    static {
        primes[0] = 2;
        for(int i=1; i<primes.length; i++) {
            primes[i] = nextPrime( );
        }
    }
    //declaration of nextPrime() . . .
Packages

- Classes can be grouped in a collection called *package*
- Java’s standard library consists of hierarchical packages, such as *java.lang* and *java.util*
  
  [http://java.sun.com/j2se/1.4.2/docs/api](http://java.sun.com/j2se/1.4.2/docs/api)
- Main reason to use package is to guarantee the uniqueness of class names
  - classes with same names can be encapsulated in different packages
  - tradition of package name: reverse of the company’s Internet domain name
    
    e.g. hostname.com -> com.hostname

Class importation (1)

- **Two ways of accessing PUBLIC classes of another package**
  1) explicitly give the full package name before the class name.
     - E.g.
       ```java
       java.util.Date today = new java.util.Date();
       ```
  2) import the package by using the *import* statement at the top of your source files (but below package statements). No need to give package name any more.
     - to import a single class from the java.util package
       ```java
       import java.util.Date;
       Date today = new Date();
       ```
     - to import all the public classes from the java.util package
       ```java
       import java.util.*;
       Date today = new Date();
       ```
     - * is used to import classes at the current package level. It will **NOT** import classes in a sub-package.
**Sample class:**

```java
import javax.swing.*;

public class SampleClass {
    MenuEvent c;
}
```

```bash
%> javac SampleClass.java
SampleClass.java:4: cannot find symbol
Symbol : class MenuEvent
Location: class SampleClass
    MenuEvent c;
^ 1 error
```

MenuEvent is a class in the package `javax.swing.event`, which locates in the package `javax.swing`. You need this statement:

```java
import javax.swing.event.*;
```

---

**Class importation (2)**

- What if you have a name conflict?

  **Eg**

  ```java
  import java.util.*;
  import java.sql.*;
  Date today = new Date(); //ERROR:java.util.Date
  //or java.sql.Date?
  ```
  
  - if you only need to refer to one of them, import that class explicitly
    ```java
    import java.util.*;
    import java.sql.*;
    import java.util.Date;
    ```
  
  - if you need to refer to both of them, you have to use the full package name before the class name
    ```java
    import java.util.*;
    import java.sql.*;
    java.sql.Date today = new java.sql.Date();
    java.util.Date nextDay = new java.util.Date();
    ```
See this code:

```java
import java.lang.Math;

public class importTest {
    double x = sqrt(1.44);
}
```

Compile:
```
%> javac importTest.java
importTest.java:3: cannot find symbol
  symbol : method sqrt(double)
location: class importTest
doUBLE x = sqrt(1.44);

1 error
```

? For the static members, you need to refer them as `className.memberName`

---

**Static importation**

- In J2SE 5.0, importation can also be applied on static fields and methods, not just classes. You can directly refer to them after the static importation.
  - E.g. import all static fields and methods of the Math class
    ```java
    import static java.lang.Math.*;
    double x = PI;
    ```
  - E.g. import a specific field or method
    ```java
    import static java.lang.Math.abs;
    double x = abs(-1.0);
    ```

- Any version before J2SE 5.0 does NOT have this feature!
Encapsulation of classes into a package

- Add a class into a package — two steps:
  1. put the name of the package at the top of your source file

```java
package com.hostname.corejava;
public class Employee {
    ...
}
```

  2. put the files in a package into a subdirectory which matches the full package name

```java
stored in the file “Employee.java” which is stored under “somePath/com/hostname/corejava/”
```

To emphasize on data encapsulation (1)

Let’s see a sample class first

```java
public class Body {
    public long idNum;
    public String name = “<unnamed>”;
    public Body orbits = null;
    public static long nextID = 0;
    Body( ) {
        idNum = nextID++;
    }
    Body(String bodyName, Body orbitsAround) {
        this( );
        name = bodyName;
        orbits = orbitsAround;
    }
}
```

**Problem:** all the fields are exposed to change by everybody
To emphasize on data encapsulation (2)

*improvement on the previous sample class with data encapsulation*

```java
public class Body {
    private long idNum;
    private String name = "<unnamed>";
    private Body orbits = null;
    private static long nextID = 0;
    Body() {
        idNum = nextID++;
    }
    Body(String bodyName, Body orbitsAround) {
        this();
        name = bodyName;
        orbits = orbitsAround;
    }
}

Problem: but how can you access the fields?
```

To emphasize on data encapsulation (3)

*improvement on the previous sample class with accessor methods*

```java
public class Body {
    private long idNum;
    private String name = "<unnamed>";
    private Body orbits = null;
    private static long nextID = 0;
    Body() {
        idNum = nextID++;
    }
    Body(String bodyName, Body orbitsAround) {
        this();
        name = bodyName;
        orbits = orbitsAround;
    }
    public long getID() { return idNum; }
    public String getName() { return name; }
    public Body getOrbits() { return orbits; }
}

Note: now the fields idNum, name and orbits are read-only outside the class. Methods that access internal data are called accessor methods sometime
To emphasize on data encapsulation (4)

*modification on the previous sample class with methods setting fields*

```java
class Body {
    private long idNum;
    private String name = "<unnamed>";
    private Body orbits = null;
    private static long nextID = 0;
    // constructors omitted for space problem . . .
    public long getID() { return idNum; }
    public String getName() { return name; }
    public Body getOrbits() { return orbits; }
    public setName(String newName) { name = newName; }
    public setOrbits(Body orbitsAround) { orbits = orbitsAround; }
}
```

**Note:** now users can set the `name` and `orbits` fields. But `idNum` is still read-only

- Making fields private and adding methods to access and set them enables the users adding actions in the future
- Don’t forget the `private` modifier on a data field when necessary! The default access modifier for fields is `package`.

How the virtual machine located classes?

- How to tell the java virtual machine where to find the .class files?
  Answer: set the `classpath`.
- Class path is the collection of all directories and archive files that are starting points for locating classes.

*E.g.*
- first suppose the following is the current classpath:
  `/home/user/classdir:/home/user/archives/archive.jar`
- then suppose the interpreter is searching for the class file of the `com.horstmann.corejava.Employee` class. It will first search class in the system class files that are stored in archives in the `jre/lib` and `jre/lib/ext` directories. It can’t find the class there, so it will turn to search whether the following files exist in the following order:
  1. `/home/user/classdir/com/horstmann.corejava/Employee.class`
  2. `/com/horstmann.corejava/Employee.class`
  3. `com/horstmann/corejava/Employee.class` inside `/home/user/archives/archive.jar`
- if any of them is been found, then the interpreter stops searching process
Setting the class path

- **Tedious way:** set the class path with the `-classpath` option for the `javac` program
  
  ```bash
  javac -classpath /home/user/classdir:/:home/user/archives/archive.jar MyProg.java
  ```
  
  *(in Windows, use semicolon to separate the items of the class path)*

- **Set the CLASSPATH environment variable in a permanent way**
  
  **UNIX/Linux**
  
  - If you use the C shell, add a line such as the following to the `.cshrc` file in your home directory
    ```bash
    setenv CLASSPATH /home/user/classdir:..
    ```
  - If you use `bash`, add a line such as the following to the `.bashrc` or `.bash_profile` file in your home directory
    ```bash
    CLASSPATH=$CLASSPATH:~/home/user/classdir
    export CLASSPATH
    ```
  - After you save the modified files, run the command
    ```bash
    source .bashrc (or .cshrc or .bash_profile)
    ```

  **Windows NT/2000/XP**

  - Open the control panel, then open the **System** icon and select the **Environment** tab. Add a new environment variable named `CLASSPATH` and specify its value, or edit the variable if it exists already.

Naming conventions

- **Package names:** start with lowercase letter
  
  - E.g. `java.util`, `java.net`, `java.io` ...

- **Class names:** start with uppercase letter
  
  - E.g. `File`, `Math` ...
  
  - avoid name conflicts with packages
  
  - avoid name conflicts with standard keywords in java system

- **Variable, field and method names:** start with lowercase letter
  
  - E.g. `x`, `out`, `abs` ...

- **Constant names:** all uppercase letters
  
  - E.g. `PI` ...

- **Multi-word names:** capitalize the first letter of each word after the first one
  
  - E.g. `HelloWorldApp`, `getName` ...

- **Exception class names:** (1) start with uppercase letter (2) end with “Exception” with normal exception and “Error” with fatal exception
  
  - E.g. `OutOfMemoryError`, `FileNotFoundException`
Supplemental reading

Object-Oriented Programming Concepts
http://java.sun.com/docs/books/tutorial/java/concepts/index.html

Object and Classes in Java
http://java.sun.com/docs/books/tutorial/java/javaOO/index.html