Object Oriented Programming using Java – Part2

By

Dr. Sunirmal Khatua,

Visvesvaraya Young Faculty Fellow, Govt. of India

Assistant Professor, University of Calcutta

Preface

- JRE In-Memory Structure
- Compile-time Polymorphism through Method Overloading
- Constructor and Finalize
- Memory Handling through Garbage Collector

Method Overloading

- Used to allow same name for two different methods within the same class.
- The overloaded methods must differ in signature.
- Signature is identified by:
  - *No. of arguments*
  - *Types of arguments*
  - *Order of arguments*
- It implements compile time polymorphism

Constructor

- A special method having the following properties:
  - Same name as the class
  - Doesn’t have any return type
  - Used to initialize the object
  - Every class must have a constructor. If we missed one Java environment will provide a default constructor automatically.
- Types of Constructor
  - Default Constructor
  - Parameterized Constructor
  - Copy Constructor

Constructor(cont.)

- We can call the overloaded constructor by using the keyword "*this*"

- this (if present) must be the first statement within the constructor of a class

Finalize

- A special method which is called before removing the object from memory.

- It has the signature:

```
protected void finalize(){
    // finalize code
}
```

Garbage Collector(GC)

- A demon thread running inside the JRE used to free the memory of unused objects.
- Most of the times GC sleeps and at regular interval it checks for garbage collection.
- Garbage collection can’t be forced but can be requested using *System.gc()*

Inheritance

- Allow a class to reuse(inherit) code from another class.
- The class that is reusing the code is called sub class and the class from which the code is reused is called the super class
- Only non-private members are inherited.
- Types of inheritance:
  - Single inheritance
  - Multilevel inheritance
  - Multiple inheritance

Inheritance(cont.)

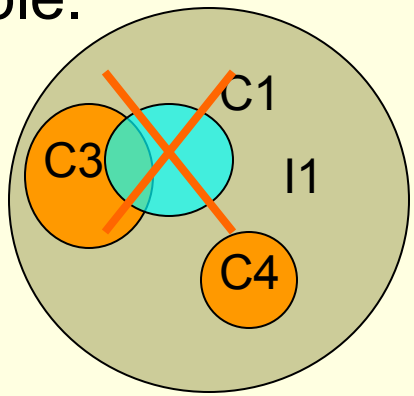
- A super class reference variable can refer to both super class object as well as sub class object.
- A sub class reference variable can't refer to a super class object.
- When a super class reference variable refer to a sub class object you need to typecast to sub class for accessing the subclass members.

Method Overriding

- Inheritance allow a sub class to write a method with the same signature of a method in the super class. The sub class will override the inherited method from the super class.
- Overridden method can't reduce visibility.
- The sub class can still access the super class method using keyword "*super*"
- A super class reference variable will call the version of the method depending on the object it currently refer. Thus it implements runtime polymorphism

Packages

- What is a Package?
  - Container for Classes & Interfaces.
  - Packages may be nested but not overlapped.
- Why Package?
  - Managing huge number of Classes & Interfaces.
  - Namespace resolution.
  - Visibility Control.
  - Managing CLASSPATH environment variable.
- Examples:
  - java.lang
  - java.util

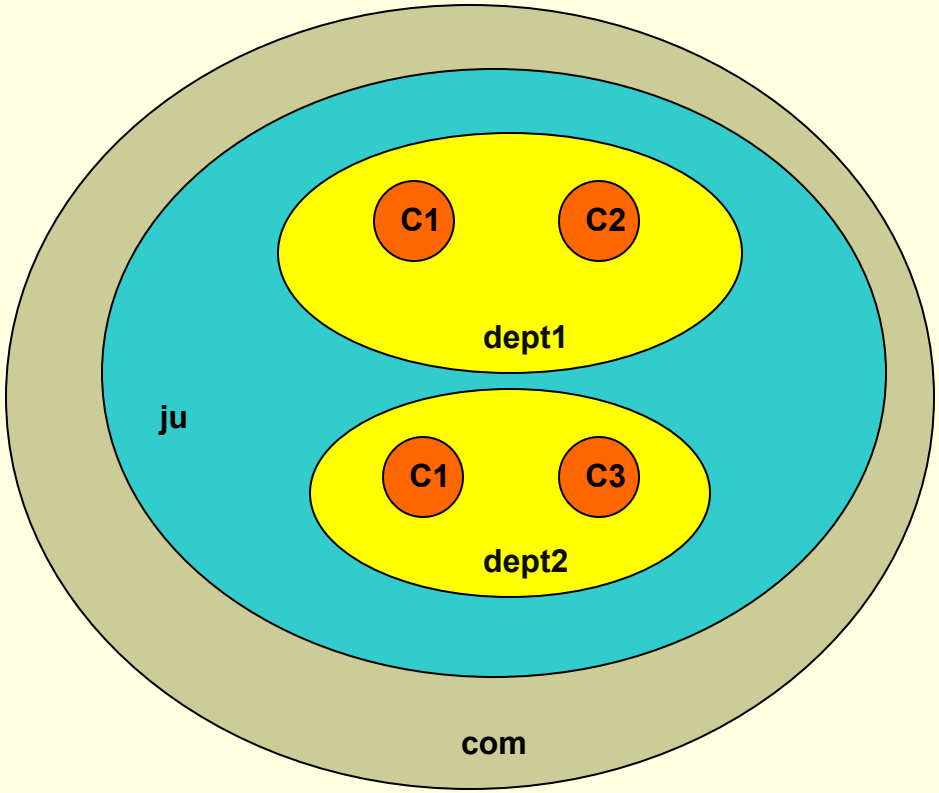


How to create Packages

- Just write the package statement at the beginning

```
// Prog1.java
package com.ju.dept1;
class C1{
    ….
 }
class C2{
    ….
 }

// Prog2.java
package com.ju.dept2;
class C1{
    ….
 }
class C3{
    ….
 }
```



Packaging of Packages

- Java Archive (JAR)
  - You can optionally specify the main class within a jar through a manifest file to create an executable jar file:

    Main-Class: <main class name>
- Syntax of creating JAR

  D:\>jar c[m]f [<manifest-file>] <jar-file> <list of starting directories of packages>
- Syntax of extracting JAR

  D:\>jar xf <jar-file>
- Directly running a JAR incase manifest file is specified

  D:\>java –jar <jar-file>

Visibility Control : Access Specifier

- Package provides visibility control through the following four Access Specifiers:
  - private
  - default (No Specifiers)
  - protected
  - public

Specifiers	Accessibility
private	Within the same class
default(no specifiers)	Also in the subclasses & non-subclasses within the same package (Everywhere within a package)
protected	Also in the subclasses of other packages
public	Everywhere

Abstract Modifier

- The *abstract* modifier can be applied to classes and methods.
- An abstract class can't be instantiated.
- It provides a way to defer implementation to subclasses.
- An abstract class may contain both concrete as well as abstract methods.
- An abstract method doesn't have any body and looks like

  abstract void m1();
- An abstract method can't be declared private.
- If a class contains any abstract method the class must be declared abstract whereas an abstract class not necessarily contain any abstract method.
- If an abstract class contains only abstract methods, then it is called pure abstract class.

Final Modifier

- A final variable can’t be changed
- A final method can’t be overridden
- A final class can’t be inherited

Transient Modifier

- Transient modifier can be applied to variables.
- When we persist or serialize an object the transient variables are not persisted or serialized.

Static Modifier

- A static member is not tied to any instance and is stored in the Class Data of the Method Area
- A static member can be accessed from the context of the class as well as objects
- A non-static method can access both static or non-static member.
- A static method can access only static members

Static Block

- A block of code inside a class and outside of any method having the signature:

```
static{
    //static initialization
      }
```

- Used to initialize during loading of a Class

Singleton Design Pattern

- Design a Class from which only a single object can be created

```
public  class  MyClass {

        private static MyClass  _instance = null;

        private MyClass(){

        }

        public static MyClass  getInstance(){
              if ( _instance == null ) {
                    _instance = new MyClass();
               }
               return _instance;
         }
 }
```

Interfaces

- An interface is a collection of public abstract methods & final static variables
- How to create interface?

```
interface I1{
        int i = 30;
        void m1();
        void m2();
}
```

- How to implement an Interface?

```
class C1 implements I1{
      void m1(){
          System.out.println("i= "+i);
       }
       void m2(){
            …
       }
}
```

- If you don't want to implement all the methods, your class must be declared *abstract*

Interfaces(cont…)

- Interface supports multiple inheritance.

```
interface I1 extends I2, I3 {
        .....
    }
```

- A class can extend another class and implement multiple interfaces simultaneously.

```
class C1 extends C2 implements I1, I2 {
        .....
    }
```

- Interface can be used to support single interface multiple implementation.

- It can also be used for developing APIs.

Java Strings

- Java uses String, StringBuffer & StringBuilder classes to encapsulate string of characters.
- The class String contains an immutable string of characters.
- It can be created in a number of ways such as:

```
String s1 = "abc";
String s2 = new String("abc");
```

- StringBuffer & StringBuilder classes contains a mutable string of characters.
- The difference between StringBuffer & StringBuilder is that the class StringBuilder is not threadsafe.

Java Strings(cont…)

- You can tokenize a string using split method of the String object

```
String[] tokens = str.split("regular expression");
```

- Looping on String collection:

```
for(String ob : tokens){
        System.out.println(ob);
    }
```

Java Arrays

- Array subscript begins with 0.
- Array declaration

  int[] v1; or int v2[][];
- Array construction

  v1 = new int[10]; or v2 = new int[3][];
- Array initialization

  int [][] v1 = {{1},{2,3,4},{5,6}};
- *System.arrayCopy(Object*System.arrayCopy(Object src , *int srcPos, Object dest, int destPos, int length)* method