Sri SAI RAM ENGINEERING COLLEGE
INSTITUTE OF TECHNOLOGY
West Tambaram, Chennai - 44

SAIRAM DIGITAL RESOURCES

YEAR II SEM III

CS8392

OBJECT ORIENTED PROGRAMMING
(Common to EEE, CSE, EIE, ICE, IT)

UNIT NO 2

OBJECT CLONING & INNER CLASSES

2.7 OBJECT CLONING – DEFINITION,DECLARATION
INNER CLASSES

COMPUTER SCIENCE & ENGINEERING

OBJECT CLONING – DEFINITION

What is Cloning?

- Cloning is used to create a duplicate (a copy) of an existing object
- The clone is an exact copy (content-wise) of the original object .
- Modifying the clone does not affect the original object.
- Cloning is Copying Objects - Not Copying References
- It is possible to use assignment to have two references to the same object.
- Cloning refers to constructing a second, distinct object with the same contents as the first.
- Cloning Involves the Construction of a Separate Object

OBJECT CLONING – DEFINITION

What is Cloning?

- Cloning is used to create a duplicate (a copy) of an existing object
- The clone is an exact copy (content-wise) of the original object .
- Modifying the clone does not affect the original object.
- Cloning is Copying Objects - Not Copying References
- It is possible to use assignment to have two references to the same object.
- Cloning refers to constructing a second, distinct object with the same contents as the first.
- Cloning Involves the Construction of a Separate Object

Cloning in Java

- The Object class defines a clone() method which is used to clone an object.
- Only classes that implement the Cloneable interface can be cloned.
- Cloneable interface defines no members.
- It indicates that a class allows a bitwise copy of an object (that is, a clone) to be made.
- If you try to call clone( ) on a class that does not implement Cloneable, a CloneNotSupportedException is thrown.
- When a clone is made, the constructor for the object being cloned is not called.
- Syntax:

  protected Object clone() throws CloneNotSupportedException
- A clone is simply an exact copy of the original

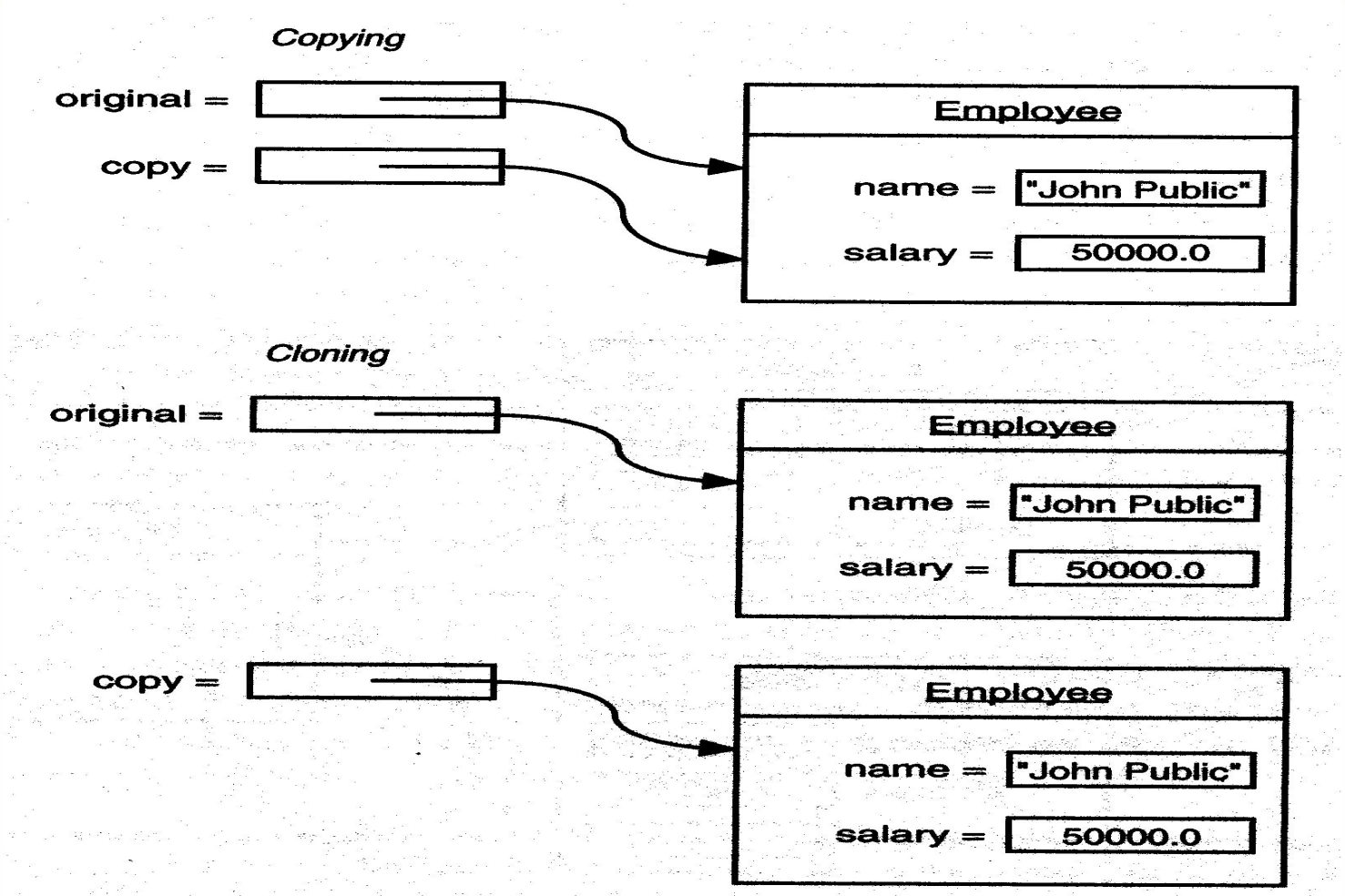
```
Employee e1= new Employee();
Employee e2=e1.clone();
```

Cloning in Java

By default, java cloning is 'field by field copy' i.e. as the Object class does not have idea about the structure of class on which clone() method will be invoked.

When JVM called for cloning,the following process occur:

- If the class has **only primitive data type members** then a completely new copy of the object will be created and the reference to the **new object copy will be returned**.
- If the class contains members of any class type then only the **object references to those members are copied** and hence the member references in both the original object as well as the cloned object refer to the same object.


Copying
original =
copy =
Employee
name = "John Public"
salary = 50000.0
Cloning
original =
Employee
name = "John Public"
salary = 50000.0
copy =
Employee
name = "John Public"
salary = 50000.0

Understanding Object Cloning

- When objects are copied using references,then the changes will be reflected in both the objects.
- Whereas if the objects are copied using clone(),both the objects are independent of each other
- Employee e1= new Employee();
  e1.x=100;
  e2 =e1;
  e2.x=500
- Now value of x will be changed for both e1 & e2 to 500
- Rather,if we clone
  e2 = e1.clone();
  e2.x=500; // changing the value of e2.x
- In e1.x we have100 and e2.xwe have 500

Facts about Cloning:

MyClass mc = new MyClass();

- Says……

mc.clone() != mc -true always

mc.getClass() == mc.clone().getClass() -true always

mc.clone().equals(mc) -true just after clone() gets called, after that,it may be false

- The purpose of the Cloneable interface is to tell the JVM that this class is eligible for cloning, but clone() will achieve the actual cloning.
- The **clone() method** saves the extra processing task for creating the exact copy of an object.
- If new keyword is used to create the object again, it will take a lot of processing time,instead we can use cloning.

Example:

```
class Student implements Cloneable
{
int rollno;
String name;
String location;

Student(int rollno,String name,String location)
{
this.rollno=rollno;
this.name=name;
this.location=location;
}

public Object clone()throws CloneNotSupportedException
{
return super.clone();
}
```

```
public static void main(String args[])
{
try{
Student s1=new Student(101,"amit","India");
Student s2=(Student)s1.clone();

System.out.println(s1.rollno+" "+s1.name+" "+s1.location);
System.out.println(s2.rollno+" "+s2.name+" "+s2.location+"\n");

s2.location="USA";
System.out.println(s1.rollno+" "+s1.name+" "+s1.location);
System.out.println(s2.rollno+" "+s2.name+" "+s2.location+"\n");

Student s3=s1;
s3.location="France";
System.out.println(s1.rollno+" "+s1.name+" "+s1.location);
System.out.println(s3.rollno+" "+s3.name+" "+s3.location+"\n");

}catch(CloneNotSupportedException c){}

}
}
```

OUTPUT:

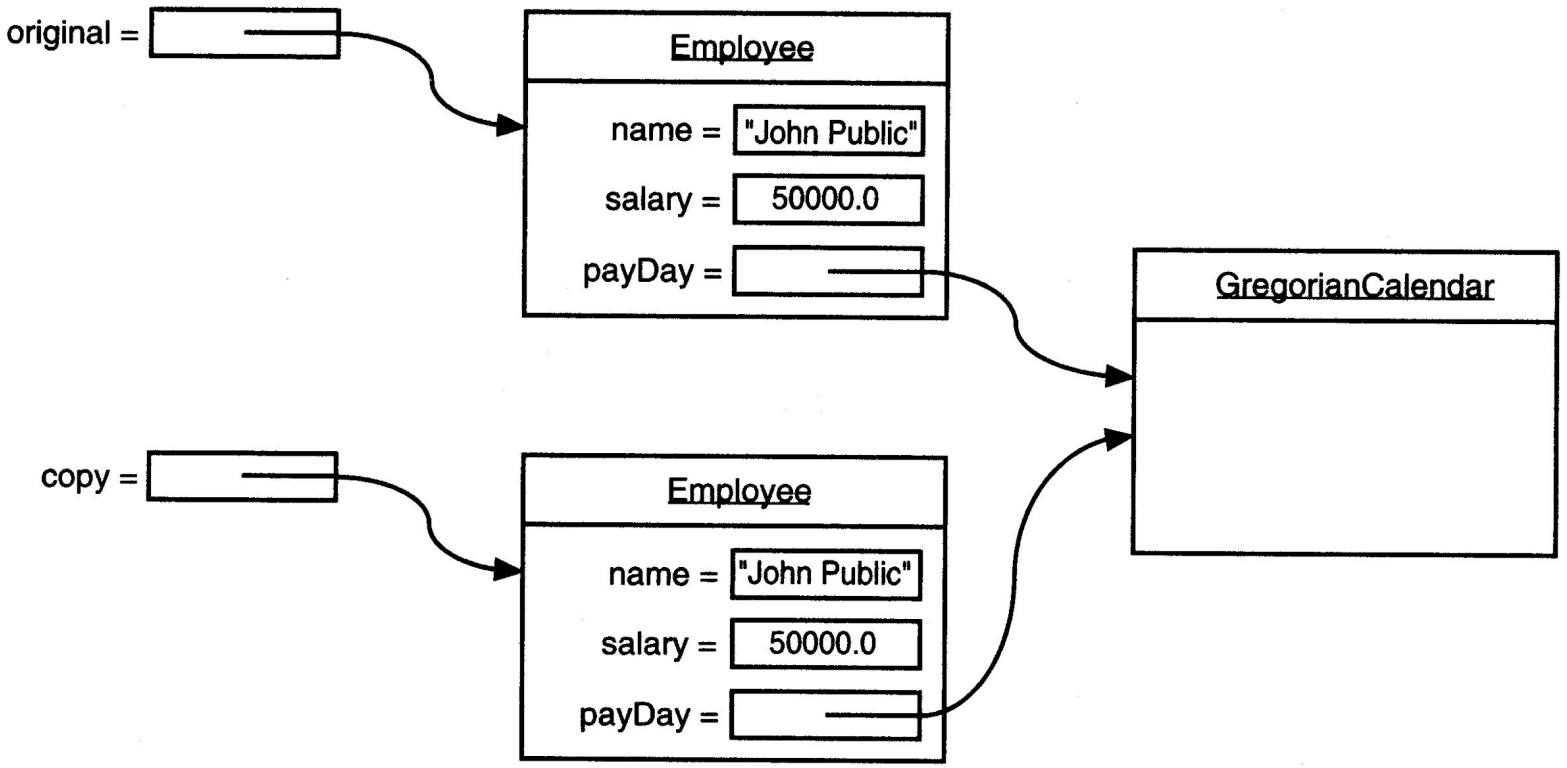
Compile by: javac Student.java

Run by: java Student

```
101 amit India
101 amit India

101 amit India
101 amit USA

101 amit France
101 amit France
```

- Cloning is of two types:
  - Shallow Copy
    - Default implementation of Java Object clone() method is shallow copy
    - shallow copy of the original object will be created with exact fields.
    - If original object has references to some other objects as fields, (like *date d =new Date())* then only the references of that object will be cloned instead of new object creation.
    - In other words,if the value of the cloned objects(which has references to other objects) changes then it will be reflected in the original as well.
    - Thus, shallow cloning is dependent on the original object.

SHALLOW CLONING



```
class Department
 {
    String empId;
    String grade;
    String designation;

   public Department(String empId, String grade, String designation)
  {
      this.empId = empId;
       this.grade = grade;
       this.designation = designation;
   }
}
class Employee implements Cloneable {
   int id;
    String name;
    Department dept;
    public Employee(int id, String name, Department dept) {
       this.id = id;
        this.name = name;
        this.dept = dept;     }
```

```
// Default version of clone() method. It creates shallow copy of an object.
    protected Object clone() throws CloneNotSupportedException
  {
       return super.clone();
   }
}

public class ShallowCopyInJava
 {
  public static void main(String[] args)
 {
        Department dept1 = new Department ("1", "A", "AVP");
        Employee emp1 = new Employee (111, "John", dept1);
        Employee emp2 = null;
        try {
           // Creating a clone of emp1 and assigning it to emp2
            emp2 = (Employee) emp1.clone();
       } catch (CloneNotSupportedException e) {
           e.printStackTrace();        }
```

```
// Printing the name and designation of 'emp1'
        System.out.println(emp1.name); //Output: John
       System.out.println(emp1.dept.designation+"\n"); // Output : AVP
 // Changing the name  designation of 'emp2'

      emp2.name="Raj";
      emp2.dept.designation = "Director";
       // change of designation will be reflected in original Employee 'emp1'
       System.out.println(emp1.name);
       System.out.println(emp1.dept.designation+"\n"); // Output : Director
       System.out.println(emp2.name);
       System.out.println(emp2.dept.designation); // Output : Director

   }
}
```

Compile by: javac ShallowCopyInJava.java

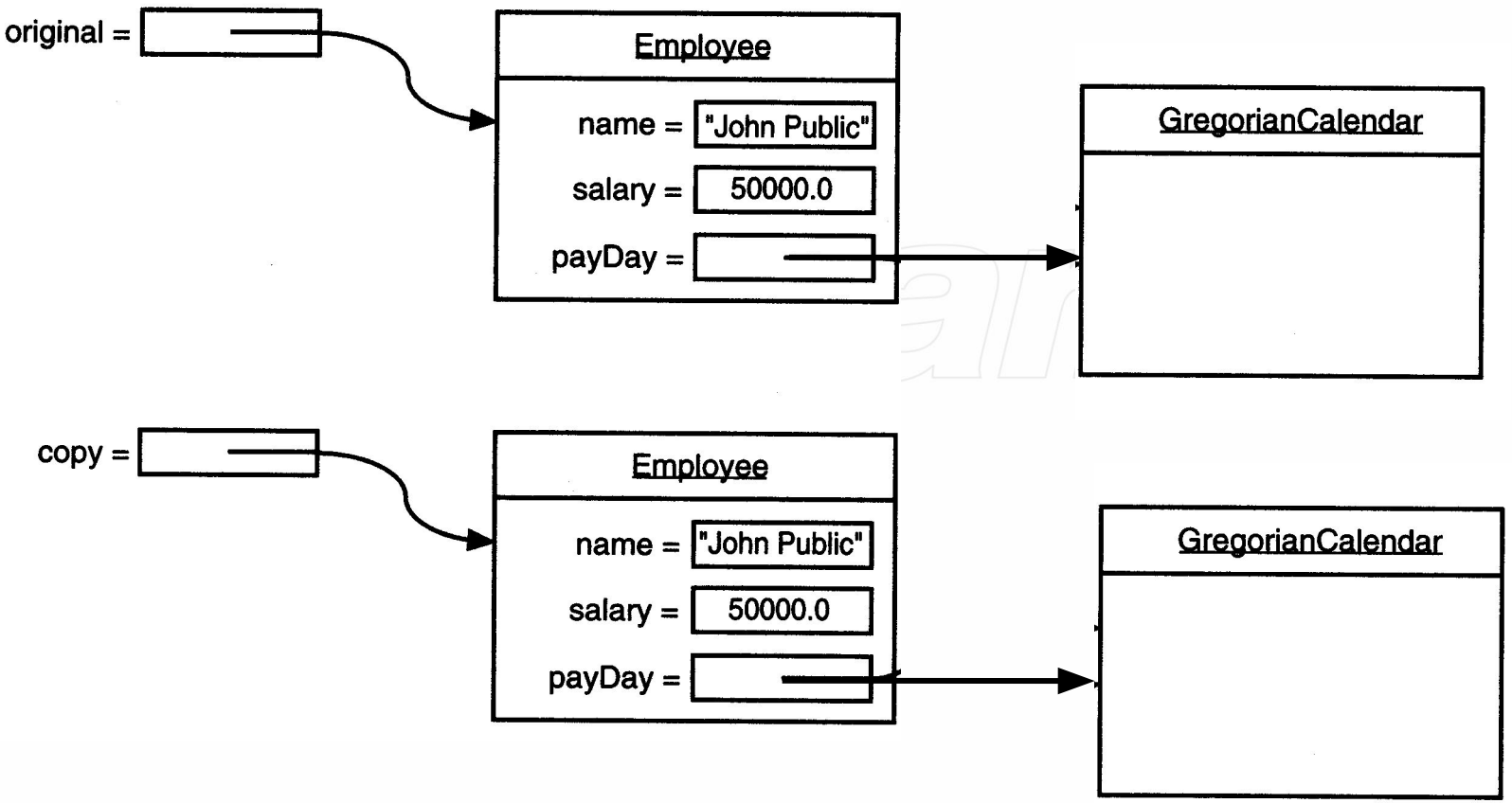
Run by: java ShallowCopyInJava

```
John
AVP

John
Director

Raj
Director
```

DEEP CLONING:

- Deep cloning means copying everything from one object to another object
- Can be achieved by implementing a Cloneable interface and overriding the clone() method in every reference type
- deep copy of an object will have an exact copy of all the fields of source object like a shallow copy
- But unlike shallow copy if the source object has any reference to object as fields, then a replica of the object is created by calling clone method.
- Means that both source and destination objects are independent of each other.
- Any change made in the cloned object will not impact the source object.

DEEP COPY


original =
Employee
name =
"John Public"
salary =
50000.0
payDay =
GregorianCalendar
copy =
Employee
name =
"John Public"
salary =
50000.0
payDay =
GregorianCalendar

Example:

```
class Department implements Cloneable
{
    String empId;
    String grade;
    String designation;
   public Department(String empId, String grade, String designation)
   {
        this.empId = empId;
        this.grade = grade;
        this.designation = designation;
    }
    //Default version of clone() method.
    protected Object clone() throws CloneNotSupportedException
   {
        return super.clone();
    }
}
```

```
class Employee implements Cloneable {
    int id;
    String name;
    Department dept;
   public Employee(int id, String name, Department dept)
   {
        this.id = id;
        this.name = name;
        this.dept = dept;
    }
   // Overriding clone() method to create a deep copy of an object.
     protected Object clone() throws CloneNotSupportedException {
        Employee emp = (Employee) super.clone();
         emp.dept = (Department) dept.clone();
         return emp;
    }
}
```

```
public class DeepCopyInJava {
    public static void main(String[] args) {
        Department dept1 = new Department("1", "A", "AVP");
         Employee emp1 = new Employee(111, "John", dept1);
         Employee emp2 = null;
         try {
  // Creating a clone of emp1 and assigning it to emp2
              emp2 = (Employee) emp1.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();         }

 // Printing the designation of 'emp1'
         System.out.println("Emp1 desig  "+emp1.dept.designation); // Output : AVP
// Changing the designation of 'emp2'
         emp2.dept.designation = "Director";
// This change will be reflected in original Employee 'emp1'
         System.out.println("Emp1 desig  "+emp1.dept.designation); // Output : AVP
        System.out.println("Emp2 desig  "+emp2.dept.designation); // Output : Director
    }
}
```

Compile by: javac DeepCopyInJava.java

Run by: java DeepCopyInJava

```
Emp1 desig  AVP
Emp1 desig  AVP
Emp2 desig  Director
```

ADVANTAGES:

- Lengthy and repetitive codes can be minimised.
- Use an abstract class with a 4- or 5-line long clone() method.
- Easiest and most efficient way for copying objects, especially if it is applied to an already developed or an old project.
- Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

DISADVANTAGES:

- Should change a lot of syntax, like implementing a Cloneable interface,defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- Should implement cloneable interface which has no methods defined in it.
- Object.clone() is protected, should provide our own clone() and indirectly call Object.clone()
- Object.clone() supports only shallow copying,need to override to do deep cloning.
- Object.clone() doesn't invoke any constructor so no control over object construction.
- If child class needs a clone method then all of its superclasses should define the clone() method

INNER CLASSES

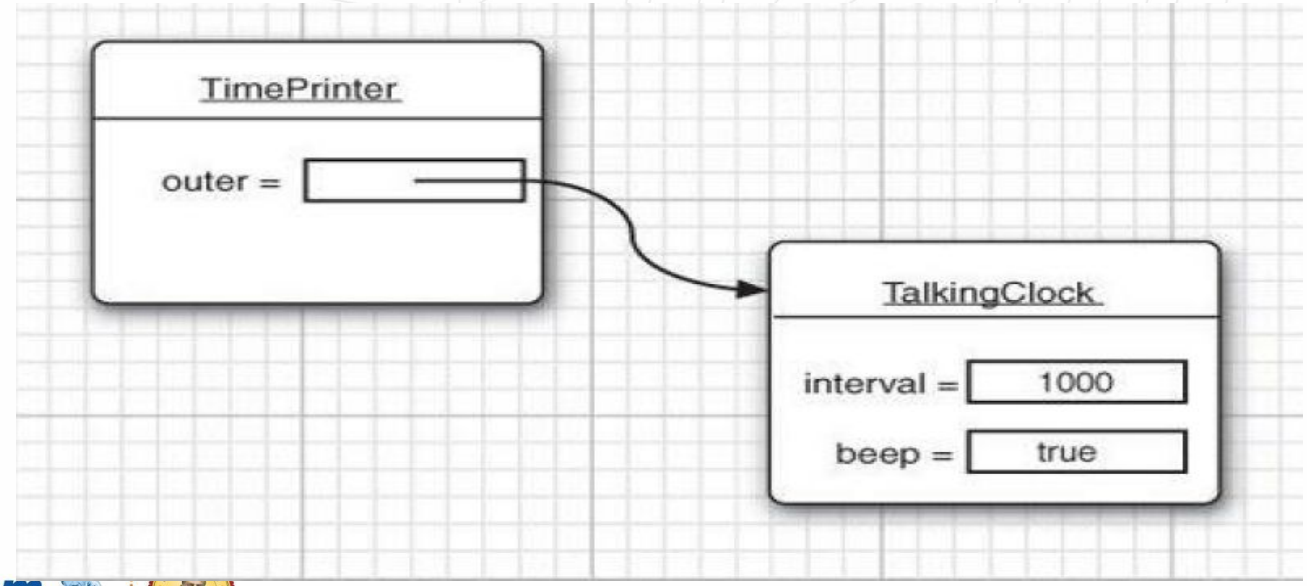
- **Java inner class** or nested class is a class which is declared inside the class or interface.
- Used to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.

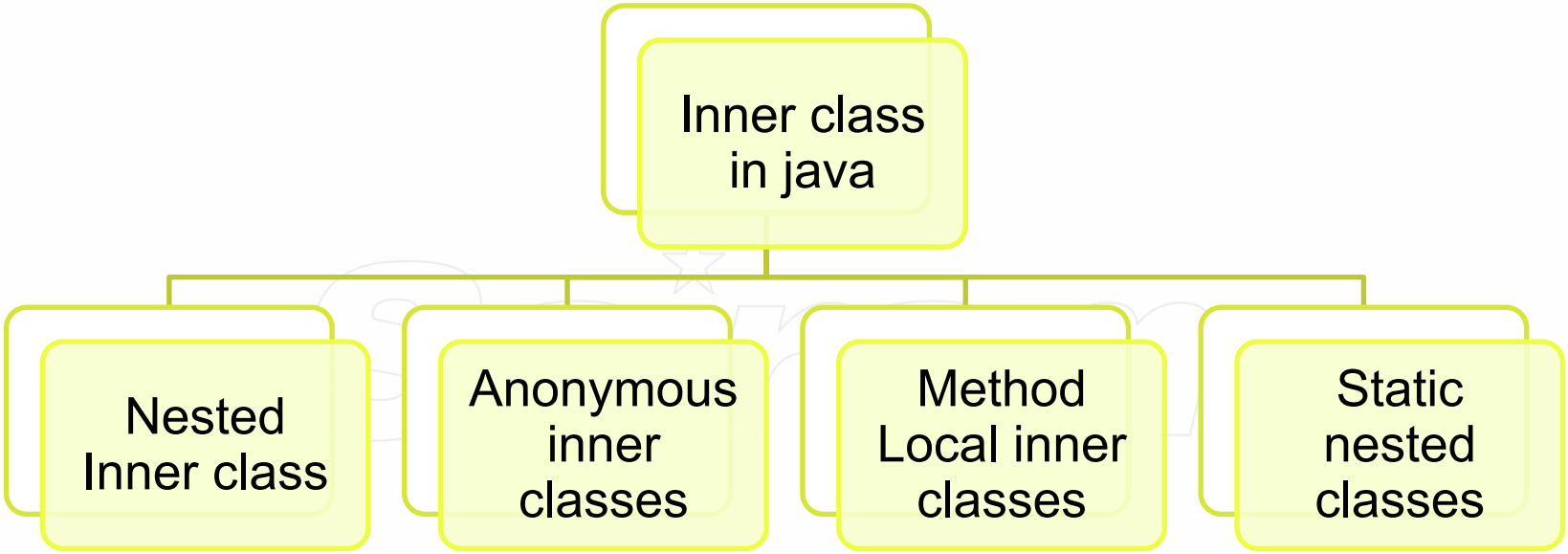
SYNTAX

```
class Java_Outer_class
{
 //code
 class Java_Inner_class
        {
            //code
        }
}
```

Why we need inner class:

- Inner class methods can access the data from the scope in which they are defined—including the data that would otherwise be private.
- Inner classes can be hidden from other classes in the same package.
- *Anonymous* inner classes are handy when you want to define callbacks without writing a lot of code




Inner class in java
Nested Inner class
Anonymous inner classes
Method Local inner classes
Static nested classes

NESTED INNER CLASS

- **Nested Inner class** can access any private instance variable of outer class.
- Like any other instance variable, we can have access modifier private, protected, public and default modifier.
- Like class, interface can also be nested and can have access specifiers.

PROGRAM

```
class Outer
{
class Inner    // Simple nested inner class
 {
      public void show()
      {
      System.out.println("In a nested class method");
      }
 }
}
class Main
{
   public static void main(String[] args) {
    Outer.Inner in = new Outer().new Inner();
    in.show();
   }
}
```

Compile by: javac Main.java

Run by: java Main

```
In a nested class method
```

Anonymous inner classes

- Anonymous inner classes are declared without any name at all.
- In the case of anonymous inner classes, we instantiate and declare it at the same time.
- Whenever we want to override the method of class or an interface, we use this class.

- They are created in two ways.

a) As subclass of specified type

b) As implementer of the specified interface

PROGRAM:

```
class Demo {
void show()
{
      System.out.println("i am in show method of super class");
}
}
class Flavor1Demo {
// An anonymous class with Demo as base class
static Demo d = new Demo() {
void show()
{
      super.show();
      System.out.println("i am in Flavor1Demo class");
}
};
public static void main(String[] args)
{
      d.show();
}
}
```

Output

```
i am in show method of super class
i am in Flavor1Demo class
```

METHOD LOCAL INNER CLASSES

- Inner class can be declared within a method of an outer class.
- In the following example, Inner is an inner class in outerMethod().

```
class Outer
{
   void outerMethod()
    {
     System.out.println("inside outerMethod");
       // Inner class is local to outerMethod()
       class Inner
          {
           void innerMethod()
           {
               System.out.println("inside innerMethod");
           }
       }
       Inner y = new Inner();
       y.innerMethod();
   }
}
class MethodDemo
{
   public static void main(String[] args) {
       Outer x = new Outer();
       x.outerMethod();
   }
}
```

Compile by: javac MethodDemo.java

Run by: java MethodDemo

```
inside outerMethod
inside innerMethod
```

STATIC NESTED CLASSES

- A static inner class is a nested class which is a static member of the outer class.
- It can be accessed without instantiating the outer class, using other static members.
- Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

PROGRAM

```
public class Outer
{
static class Nested_Demo
 {
public void my_method()
 {
 System.out.println("This is my nested class");
}
 }
public static void main(String args[])
{
 Outer.Nested_Demo nested = new
Outer.Nested_Demo(); nested.my_method();
 }
}
```

Output

```
This is my nested class
```

ADVANTAGES:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.