# Packages in Java

# Packages



- Packages provide a way to group a number of related classes and/or interfaces together into a single unit.
- That means, packages act as “containers” for classes.
- Packages avoid name space collision problem.
- There can not be two classes with same name in a same Package. But two packages can have a class with same name.

# Defining packages



- To create a package is quite easy:
- Include a **package** command as the first statement in a java source file.
- General form of the **package** statement:
  - package *pkg*
    - Where, *pkg* is name of the package.
- For Example:

```
package mypackage;
class balance
{
//code for balance class
}
```

# Example of a short package



```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

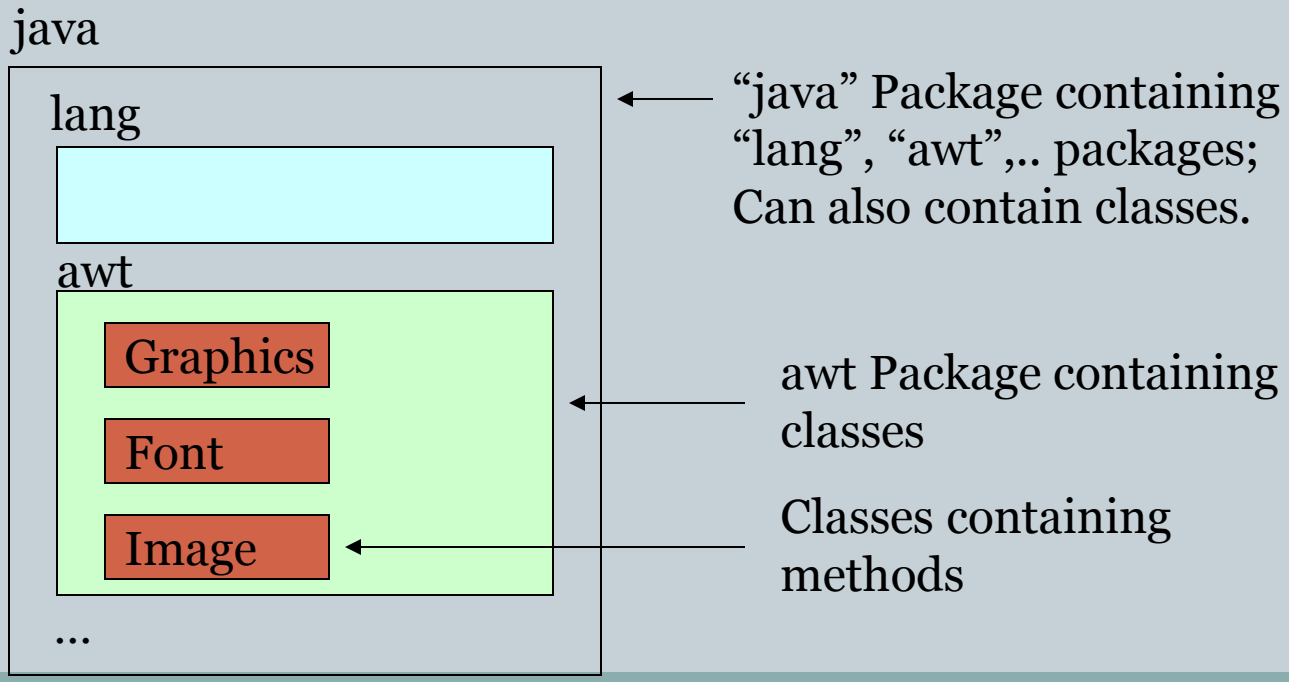
# More about Packages



- The **package** statement defines a name space in which classes are stored.
- If the package statement is omitted:
  - Then class names are put into the default package which has no name.
- Java uses file system directories to store packages.
  - i.e. All the .class files for any classes which are declared as the part of **MyPackage** must be stored in a directory called MyPackage.
- Package names have a correspondence with the directory structure.

# Using System Packages

- The packages are organised in a hierarchical structure.
- General form of a multilevel package is:
  - Package pkg1[.pkg2][.pkg3]
  - For example,
  - java.awt.images



# Finding Packages



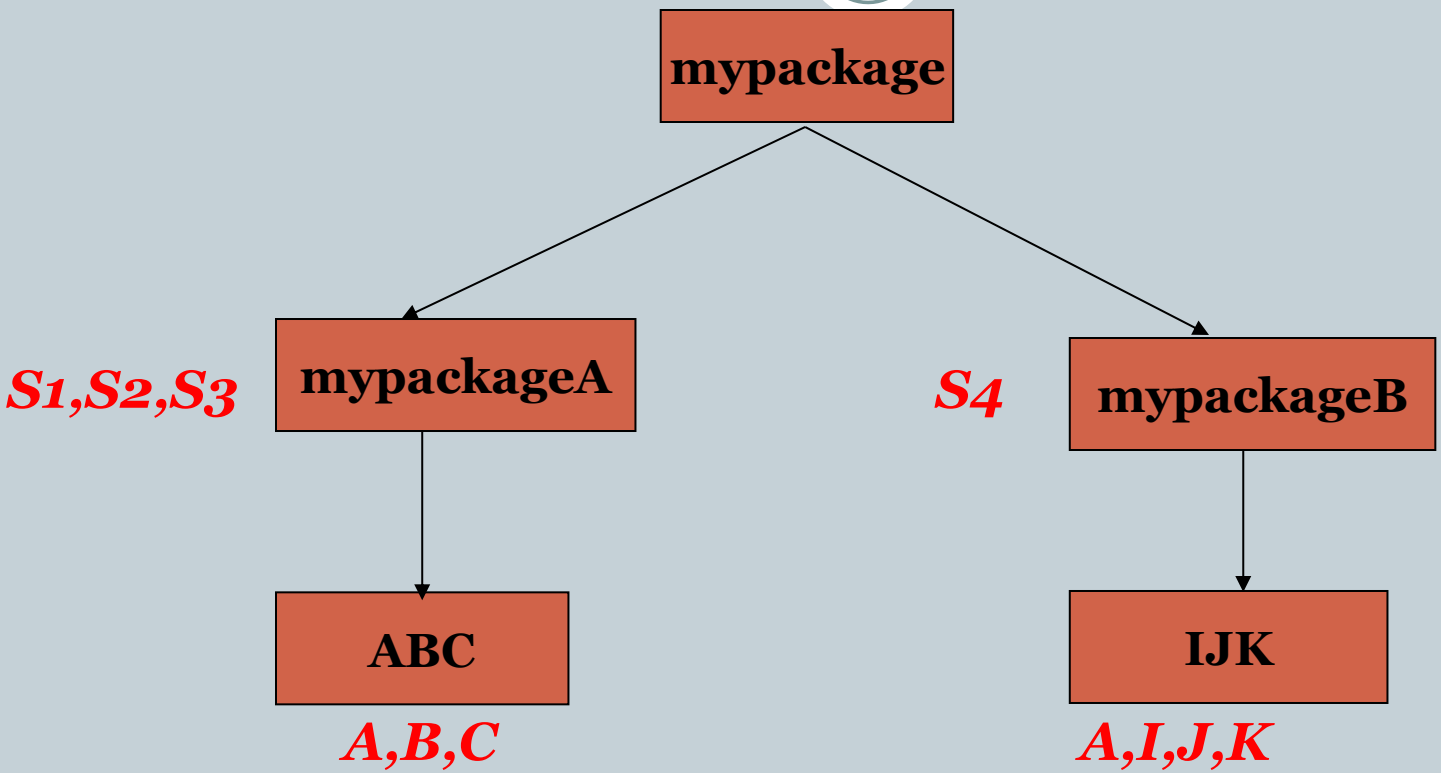
- Packages are mirrored through directory structure.
- To create a package, First we have to create a directory /directory structure that matches the package hierarchy.
- Where does the Java run-time system look for package?
  1. By default in the Current directory.
     - If your package is subdirectory of the current directory, will be found.
  2. Specify the directory path by setting the CLASSPATH environment variable.
  3. Use the ~classpath option with java command to specify the path of class.

# Importing of a package



- **import** statement allows importing of a package.
- General form of the import statement is:
  - Import *pkg1*[.*pkg2*].(*classname*|*);
    - Where, *pkg1* is top level package and *pkg2* is the name of a subordinate package and * implies entire package.
  - For example:
    - import java.io.*;
- If a class with the same name exists in two different packages that you import using the star form,
  - then compile time error occur if we try to use one of the classes.

# Creating Packages




- *Package ABC and IJK have classes with same name.*
- *A class in ABC has name mypackage.mypackageA.ABC.A*
- *A class in IJK has name mypackage.mypackageB.IJK.A*

# Example importing
# import mypackage.mypackageA.ABC;

<<packagetest.java>>

```
import mypackage.mypackageA.ABC.*;
class packagetest
{
        public static void main(String args[])
        {
                B b1 = new B();
                C c1 = new C();
        }
}
```

This is Class B
This is Class C

# Contd…

```
import mypackage.mypackageA.ABC.*;
Import mypackage.mypackageB.IJK.*;
class packagetest
{
public static void main(String args[])
{
A a1 = new A();
}}
```

**mypackage.mypackageA.ABC.A a1 = new mypackage.mypackageA.ABC.A();**

**OR**

**mypackage.mypackageB.IJK.A a1 = new mypackage.mypackageB.IJK.A();**

**<< class A is present in both the imported packages ABC and IJK. So A has to be fully qualified in this case>>**

# INTRODUCING ACCESS CONTROL

# Access Control Modifiers



- The access modifiers specifies accessibility (scope) of a data member, method, constructor or class.
- There are two types of modifiers in java:
  - access modifier
  - non-access modifier.
- Access specifiers/modifiers indicate which members of a class can be used by other classes.
- There are 4 different access specifiers in JAVA:
  1. public
  2. private
  3. protected
  4. default

# public and private Access Modifiers



- public keyword
  - A public member of a class can be accessed by any other code.
  - Why main() function has always been preceded by the public specifier?
- private keyword
  - Private variables and methods are visible or accessible only to methods of the class in which they are declared
  - Declaring instance variables private is known as data hiding
  - Classes can not be private
  - Most restrictive access level

# Effects of public and private access



```
/* This program demonstrates the difference between public and private*/
class Test {
        int a; // default access
        public int b; // public access
        private int c; // private access
        // methods to access c
        void setc(int i) { // set c's value
              c = i;
        }
        int getc() { // get c's value
             return c;
        }
}
```

# Contd...



```
class AccessTest {
        public static void main(String args[]) {
                Test ob = new Test();
                 // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;

               // ob.c = 100;
               // You must access c through its methods
                  ob.setc(100); // OK
           System.out.println("a, b, and c: " + ob.a + " " +ob.b + " " + ob.getc());
}
}
```

# Example2



```
// This class defines an integer stack that can hold 10 values.
class Stack {
  /* Now, both stk and tos are private.  This means
     that they cannot be accidentally or maliciously
     altered in a way that would be harmful to the stack.
  */
  private int stk[] = new int[10];
  private int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stk[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stk[tos--];
  }
}
```

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());

    // these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stk[3] = 100;
  }
}
```

# Protected access



- Protected access applies only when inheritance is involved.
- Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected member‘s class.
- The protected access modifier cannot be applied to class and interfaces.
- Protected member is visible or accessible to
  - The current class
  - Subclasses of the current class
  - All classes that are in the same package as that of the class
- The level of protection is between public access and default access

# Example of protected access specifier



```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

**Output:**
**Hello**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

# Default access



- Default access is also known as package access.
- Package access is used when there is no access specifier.
- Package access means that all classes in the same package can access the member.
- But for all other classes the member is private.

# Example of default access modifier



```
//save by A.java

package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
    A obj = new A();              //Compile Time Error
    obj.msg();                    //Compile Time Error
  }
}
```

In this example, the scope of class A and its method msg() is default. So, it cannot be accessed from outside the package.

# Access Control and Inheritence



- The following rules for inherited methods are enforced:

1. Methods declared public in a superclass also can be public in all subclasses.
2. Methods declared protected in a superclass can either be protected or public in subclasses; they cannot be private.
3. Methods declared without access control (no modifier was used) can be declared more private in subclasses.
4. Methods declared private are not inherited at all, so there is no rule for them.

# Class member accessibility

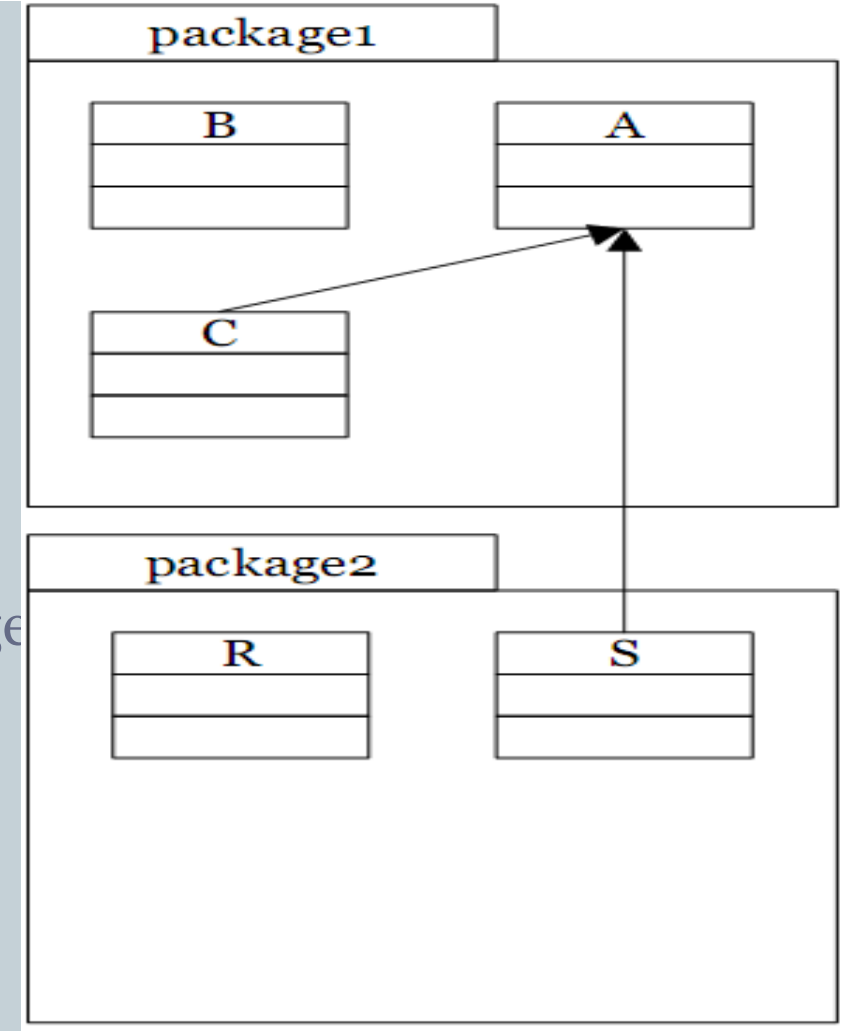


| Accessible to: | public | protected | default | private |
|---|---|---|---|---|
| Same class | Yes | Yes | Yes | Yes |
| Class in the same package | Yes | Yes | Yes | No |
| Subclass in different package | Yes | Yes | No | No |
| Non-subclass, different package | Yes | No | No | No |

# Example



- Consider the access modifier for a member x in class A. If it is:
  - private –
    it can only be used inside A.
  - default –
    it can be used anywhere in package1, that is, A,B and C.
  - protected –
    it can be used anywhere in package in subclasses of A in other packages, here S.
  - public –
    it can be used everywhere in the system



# Non Access Modifers



- The static modifier for creating class methods and variables.
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.

# Static Variables



- Static variables are class members that will be used independent of any object of the class.
- Called as **class variables.**
- Only one copy of a *static* variable is created per class.
- Static variable is just like a global variable for a class that is allocated memory once and all objects of that class share that common copy.

# Contd…



- To define a *static* variable, include the keyword *static* in its definition.
- Syntax:
  - accessSpecifier static dataType variableName;
- Example:
  - public static int countAutos = 0;

1. A static variable can be referenced either using its class name or an name object.
2. Instantiating a second object of the same type does not increase the number of static variables.

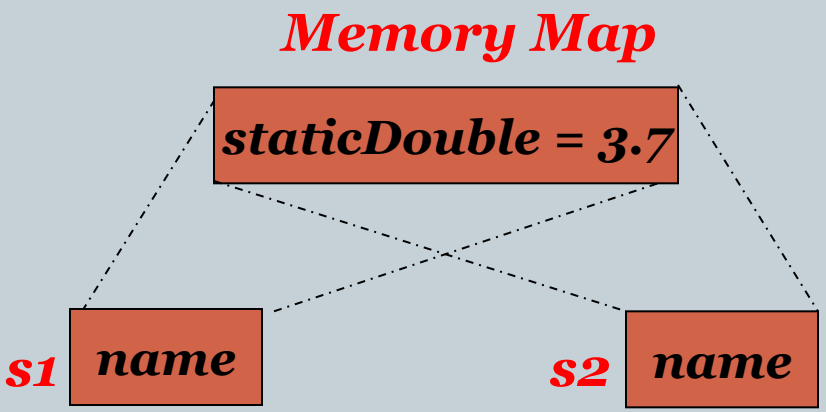
# Contd…



- **Example**

```
StaticStuff s1, s2;
 s1 = new StaticStuff();
 s2 = new StaticStuff();
 s1.staticDouble = 3.7;
 System.out.println( s1.staticDouble );
 System.out.println( s2.staticDouble );
```

```
public class StaticStuff {
   public static staticDouble ;
   public string name;
   . . .
}
```



# Static Methods



- Static methods are also called as **class methods.**
- Often defined to access and change *static* variables.
- *Static* methods cannot access instance variables:
  - *static* methods are associated with the class, not with any object.
  - *static* methods can be called before any object is instantiated, so it is possible that there will be no instance variables to access that method. Ex: main() method

*Static methods can not call non static methods.*

# Contd…



- **The declaration**
  - Static methods are declared by inserting the keyword “static” immediately after the scope specifier (*public*, *private* or *protected*).

```
public class ArrayStuff {

   public static double mean(int[] arr)   {
      double total = 0.0;
      for (int k=0; k!=arr.length; k++)   {
         total = total + arr[k];
      }
      return  total / arr.length;
   }
}
```

- **Calling**
  - Static methods are called using the name of their class in place of an object reference.

```
double myArray = {1.1, 2.2, 3.3};
...
double average = ArrayStuff.mean(myArray);
```

# Static methods: Restrictions



- The body of a static method cannot reference any non-static (instance) variable.
- The body of a static method cannot call any non-static method unless it is applied to some other instantiated object.
- **However, ...**
- The body of a static method can instantiate objects.
- **Example** (the run.java file)

```
public class run {
   public static void main(String[] args)   {
      Driver driver = new Driver();
   }
}
```

# Static Method Example



```
class num
{
        int a,b,c;
        static int d = 10;
        static double e = 20.56;
        num(int a,int b,int c)
        { this.a = a; this.b =b; this.c =c; }
        static void sum(int a1 , int b1)
        {
            //System.out.println("a="+a+"b="+b+"c="+c);
            System.out.println("d="+d+"e="+e);

             }
```

***non static instance fields***

***static instance fields***

***static method***

***a,b,c are non static fields and can not be accessed from a static method***

# Contd…



```
static double sum(double a , double b)
{
        System.out.println("d="+d+"e="+e);
        return 40.56;
}
static void pr()
{
        System.out.println("This is method pr");
}
void print()
{
        System.out.println("This is method print");
        pr();    // → call to static method from a non static method ----→ Valid
        System.out.println("a="+a+"b="+b+"c="+c);
        System.out.println("d="+d+"e="+e);
}
}
```

# Rules for static and Non static Methods



| | *static* Method | Non-*static* Method |
|---|---|---|
| Access instance variables? | no | yes |
| Access *static* class variables? | yes | yes |
| Call *static* class methods? | yes | yes |
| Call non-*static* instance methods? | no | yes |
| Use the object reference *this*? | no | yes |

# Static class



- Java allows us to define a class within another class. Such a class is called a nested class.
- The class which enclose the nested class is known as Outer class or Top level class.
- There can be two types of nested classes:
  - *Static*
  - And *Non-static.*
- Top level class can never be static.
- ***Only nested classes can be declared as static***.
- Non static nested classes are called as inner classes.

# Differences b/w static and non-static nested class



- Inner class can access both static and non-static members of Outer class. Whereas, a static class cannot access non-static members of the Outer class directly. It can access only static members of Outer class.
- Nested static class doesn't need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference.
- An instance of Inner class cannot be created without an instance of outer class and an Inner class can reference data and methods defined in Outer class in which it nests.

# Example2



```
olass Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner olass
  olass Inner {
    int y = 10; // y is looal to Inner
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // error, y not known here!
  }
}

olass InnerClassDemo {
  publio statio void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

# Static class



- Can we define static members inside an inner class?
- Ans: NO, since an inner class is associated with an instance. So members of inner class must be referenced with the help of  an instance.

# Static block



- Static block (also called static clause) can be used for static initializations of a class.
- The code inside static block is executed only once:
  - The first time an object of that class is created or the first time static member of that class is accessed.
- Also, static blocks are executed before constructors.

# Example1



```
class Test {
    static int i;
    int j;

    // start of static block
    static {
        i = 10;
        System.out.println("static block called ");
    }
    // end of static block
}
 class Main {
    public static void main(String args[]) {
         // Although we don't have an object of Test, static block is
        // called because i is being accessed in following statement.
        System.out.println(Test.i);
    }
}
```

**OUTPUT:**
**Static block called**
**10**

# Example2



```
class Test {
    static int i;
    int j;
    static {
        i = 10;
        System.out.println("static block called ");
    }
    Test(){
        System.out.println("Constructor called");
    }
}
class Main {
    public static void main(String args[]) {

      Test t1 = new Test();
       Test t2 = new Test();
    }
}
```

**OUTPUT:**
**Static block called**
**Constructor called**
**Constructor called**

Although we have two objects, but static block is executed only once.

# Comparison of static keyword in C++ and Java



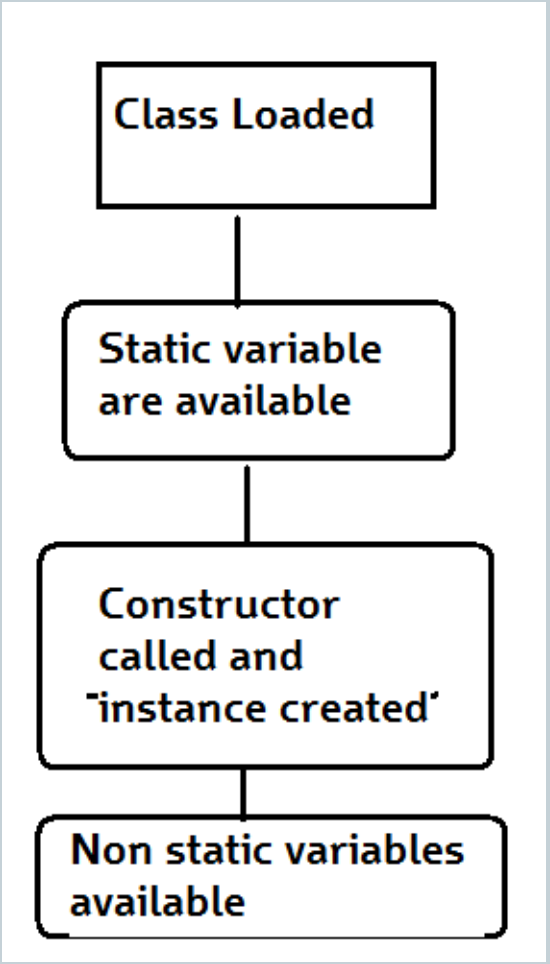
- Static Data Members:
  - Like C++, static data members in Java are class members and shared among all objects.
  - Unlike C++, Java doesn't support static local variables.
- Static Block:
  - Unlike C++, Java supports a special block, called static block which can be used for static initialization of a class. The code inside static block is executed only once.
- Like C++, static data members and static methods can be accessed without creating an object. They can be accessed using class name.

# How it works



Basic Steps of how objects are created

1. Class is loaded by JVM
2. Static variable and methods are loaded and initialized and available for use
3. Constructor is called to instantiate the non static variables
4. Non static variables and methods are now available



# Singleton class



- Purpose of the Singleton class is to control the object creation.
- Limiting the number of objects to one only.

# Singleton.java



```
public class Singleton
{
        private static Singleton singleton = new Singleton( );
        /* A private Constructor prevents any other * class from instantiating. */
        private Singleton(){ }
        /* Static 'instance' method */
        public static Singleton getInstance( )
        {
                return singleton;
         }
         /* Other methods protected by singleton-ness */
        protected static void demoMethod( )
        {
                 System.out.println("demoMethod for singleton");
         }
}
```

# singletonDemo.java



```
public class SingletonDemo
{
        public static void main(String[] args)
        {
                Singleton tmp = Singleton.getInstance( );
                tmp.demoMethod( );
        }
}
```

**OUTPUT:**

```
demoMethod for Singleton
```

# Example 2



```
public class ClassicSingleton
{
        private static counter=0;
        private ClassicSingleton instance =null;
         private ClassicSingleton()
        { // Exists only to defeat instantiation.
         }
         public static ClassicSingleton getInstance()
        {
                if(counter==0)
                 {
                        instance = new ClassicSingleton();
                        counter++;
                 }
                 return instance;
        }
 }
```

This example employs a technique called “lazy instantiation” to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. Which ensures that singleton instances are created only when needed.

# Singleton classes



- Ques: how to create exactly n number of objects using singleton class concept?