Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- Increases the readability of the program.
- There are two ways to overload the method in java

  1) By changing number of arguments.
  2) By changing the data type.

1) Method Overloading: changing no. of arguments

```
class Adder
{
        static int add(int a,int b)
        {
        return a+b;
        }
        static int add(int a,int b,int c)
        {
        return a+b+c;
        }
}
class TestOverloading1
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

2) Method Overloading: changing data type of arguments

```
class Adder
{
static int add(int a, int b)
{return a+b;}
static double add(double a, double b)
{return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

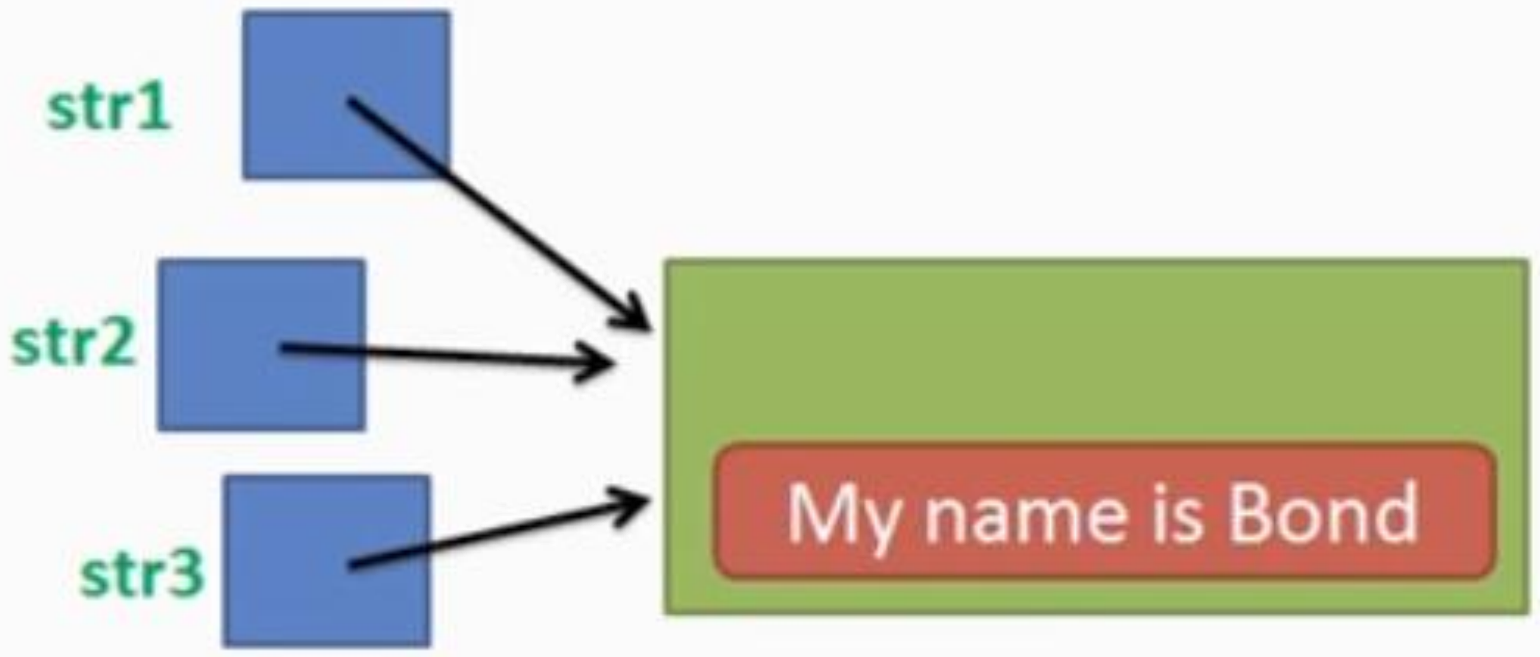
String Class

- Java.lang.string – Final class hence no other class can inherit it
- String class is immutable: Once object is created and initialized, changes in that objet can not be done.
- An attempt to make any change in such object results in a new object and reference variable will refer to the newly created object.

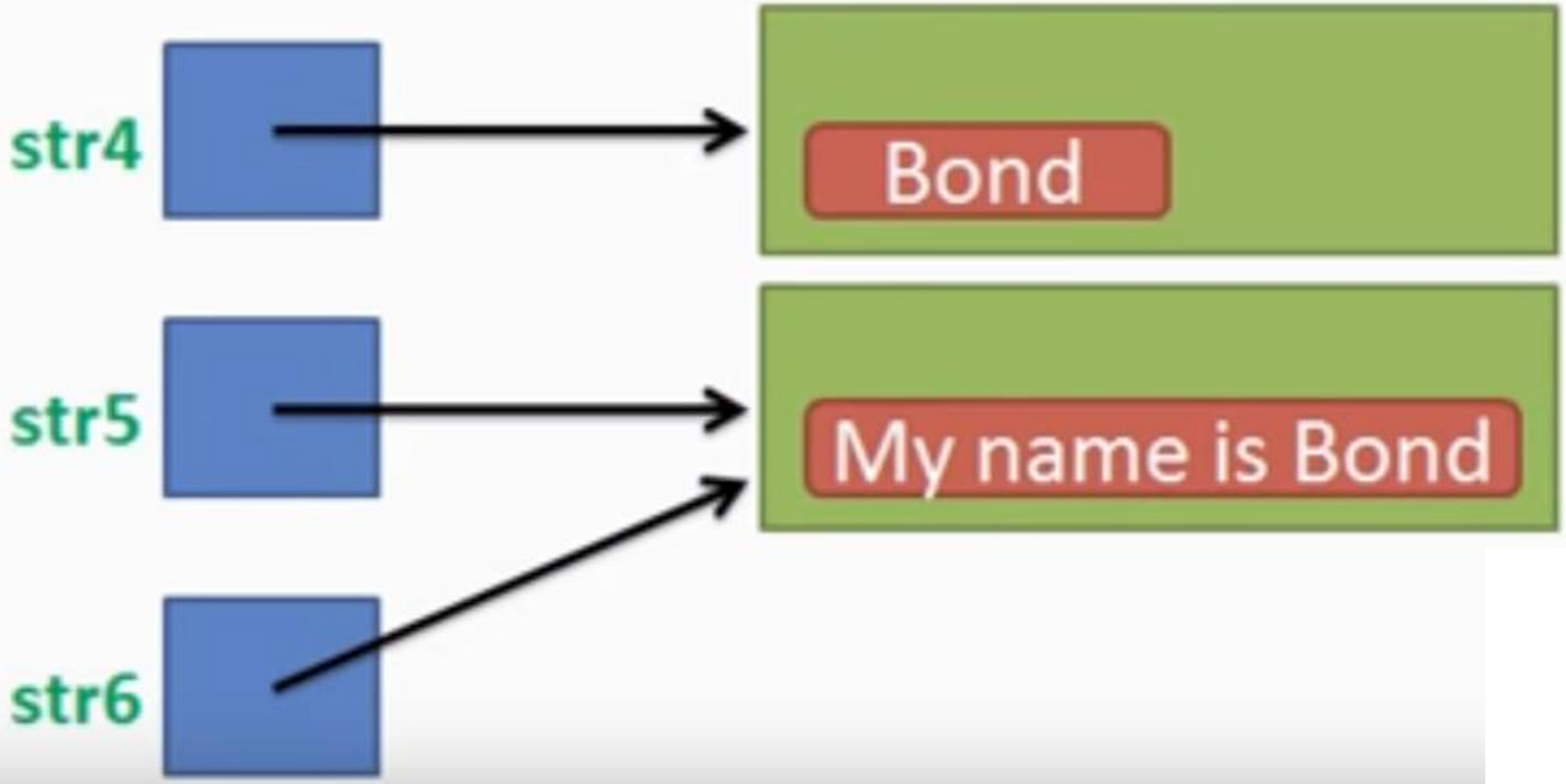
Creating object of String Class

- String **str1** = “My name is Bond” ;
- No need of ‘new’ keyword to create a string object.
  - String **str1** = “My name is Bond” ;
  - String **str2** = “My name is Bond” ;
  - String **str3** = “My name” + “is Bond” ;
- Above all the three references str1, str2, str3 denote the same string object.
- If 2 or more strings have same set of characters in the same sequence with same case then they share same reference in memory.

In memory

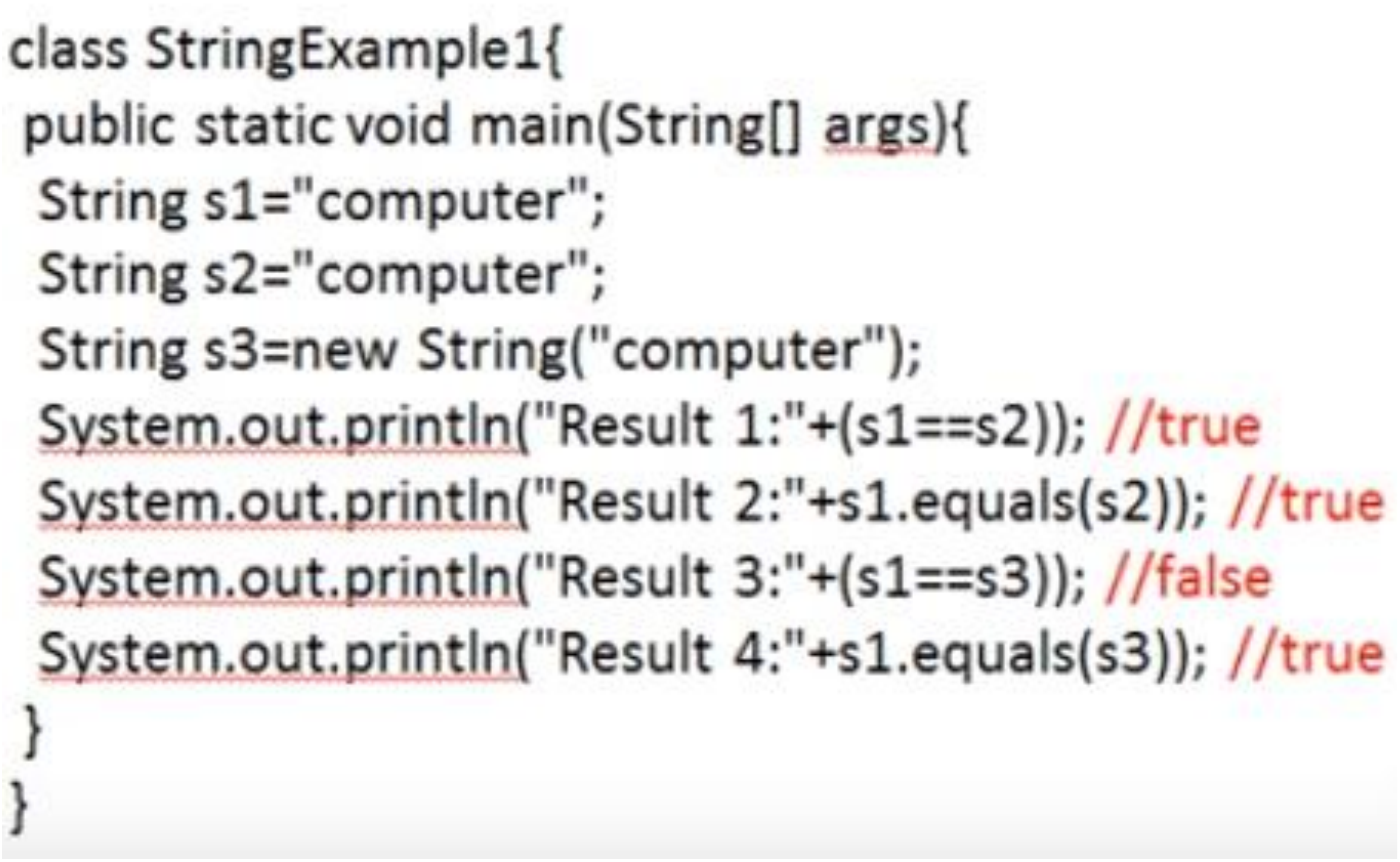


❑ String str4 = “Bond”;
String str5 = “My name is ”+ str4;
String str6 = “My name is Bond”;



Creating String with ‘new’ keyword

- String str5 = new String(“My name is Bond”);
- Every time if, object is created with new keyword, a fresh new object is created even if the character set, sequence and case is same of the objects being created.



```
class StringExample1{
 public static void main(String[] args){
  String s1="computer";
  String s2="computer";
  String s3=new String("computer");
  System.out.println("Result 1:"+(s1==s2)); //true
  System.out.println("Result 2:"+s1.equals(s2)); //true
  System.out.println("Result 3:"+(s1==s3)); //false
  System.out.println("Result 4:"+s1.equals(s3)); //true
 }
}
```

Constructors in Java

- It’s a member function of the class with same name as the name of the class.
- They do not have return types not even void.
- Gets called implicitly as soon as the object of the class gets created.
- Not mandatory to create constructor.
- When no constructor is written, compiler implicitly provides a default constructor.
- Constructors can be overloaded.
- Types:- 1) Default 2) Parameterized
- Like C++, Java also supports copy constructor. But, unlike C++, Java doesn’t create a default copy constructor if you don’t write your own.
- Copy Constructor: to prepare a duplicate of an existing object of a class, can take only one parameter, which is a reference to an object of the same class.

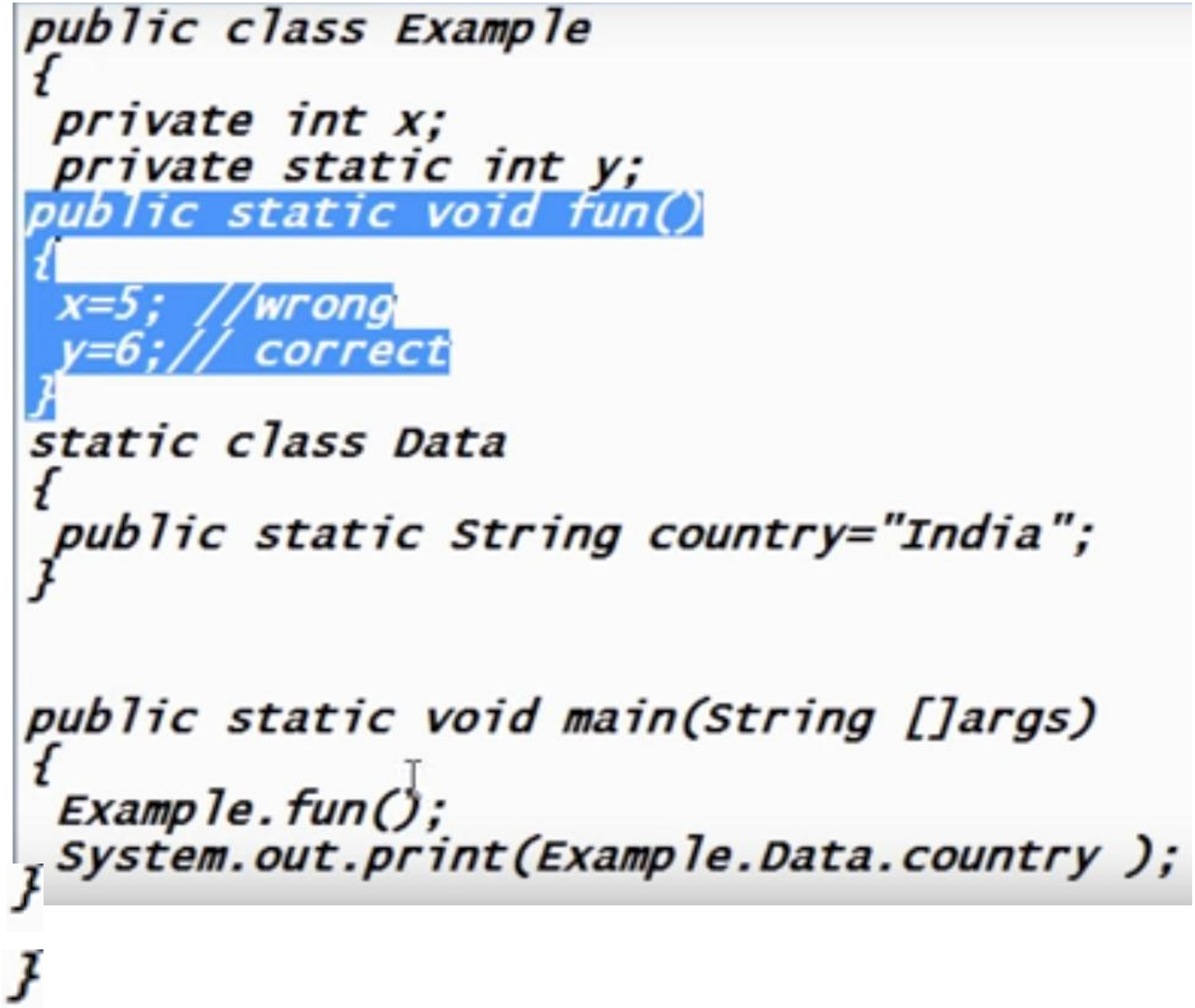
Static members in Java

(variables, methods, class)

- Variables and methods can be made static with the help of static keyword.
- Variables inside a method are called local and can not be made static.
- But we can create a class inside a class called as inner class, and make it static.
- Static variables belongs to whole class and are independent of the objects.
- Has a single copy for the whole class.
- Static functions are called with: class_name.function_name;

Static members in Java

- Static functions can assess only static members of the class.
- Static Class: Inner class (a class inside class) can be made static only.
- Outer class can never be static.



```
public class Example
{
 private int x;
 private static int y;
public static void fun()
{
 x=5; //wrong
 y=6;// correct
}
static class Data
{
 public static String country="India";
}


public static void main(String []args)
{
 Example.fun();
 System.out.print(Example.Data.country );
}

}
```

Inheritance in java

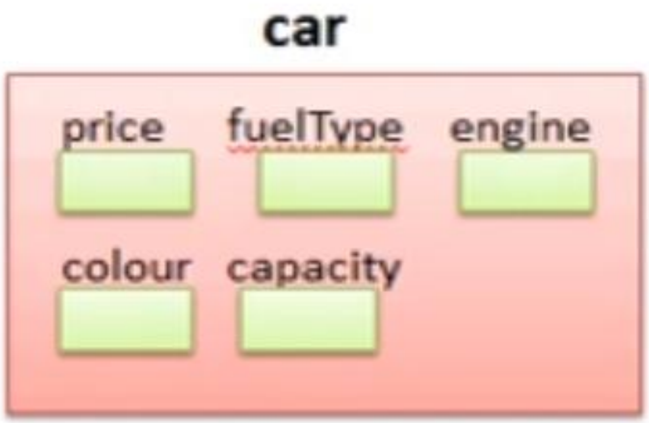
Object is real world entity

Properties

price
fuel type
engine
colour
capacity

Methods

setPrice()
setFuelType()
setEngine()
setColour()
setCapacity()
getPrice()
getFuelType()
getEngine()
getColour()
getCapacity()



Syntax

```
class SubClass extends SuperClass
{


}
```

- ❑ extends is a keyword
- ❑ Base class means Super Class
- ❑ Derived Class means Sub Class

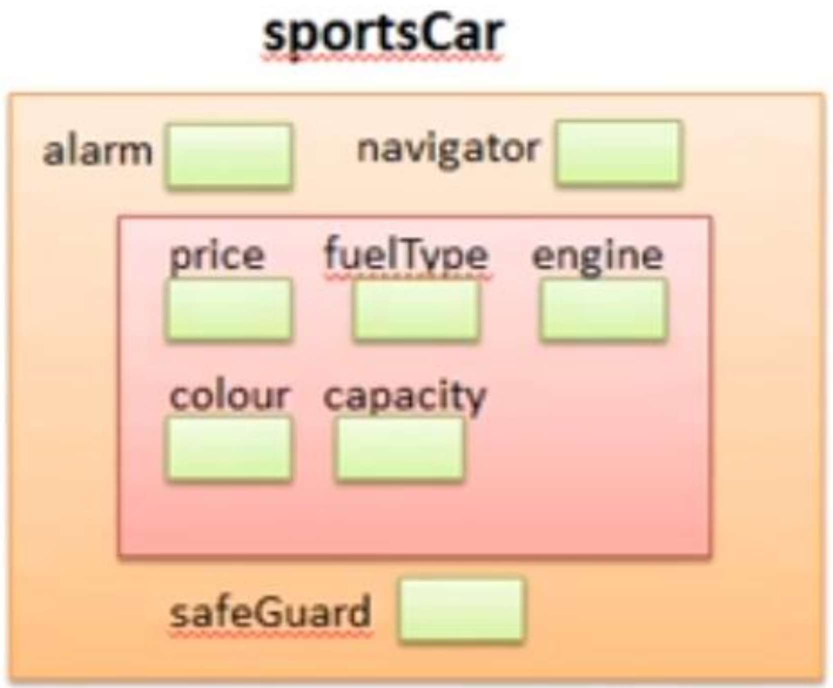
Object is real world entity

Properties

price
fuel type
engine
Colour
capacity
alarm
navigator
safeGuard

Methods

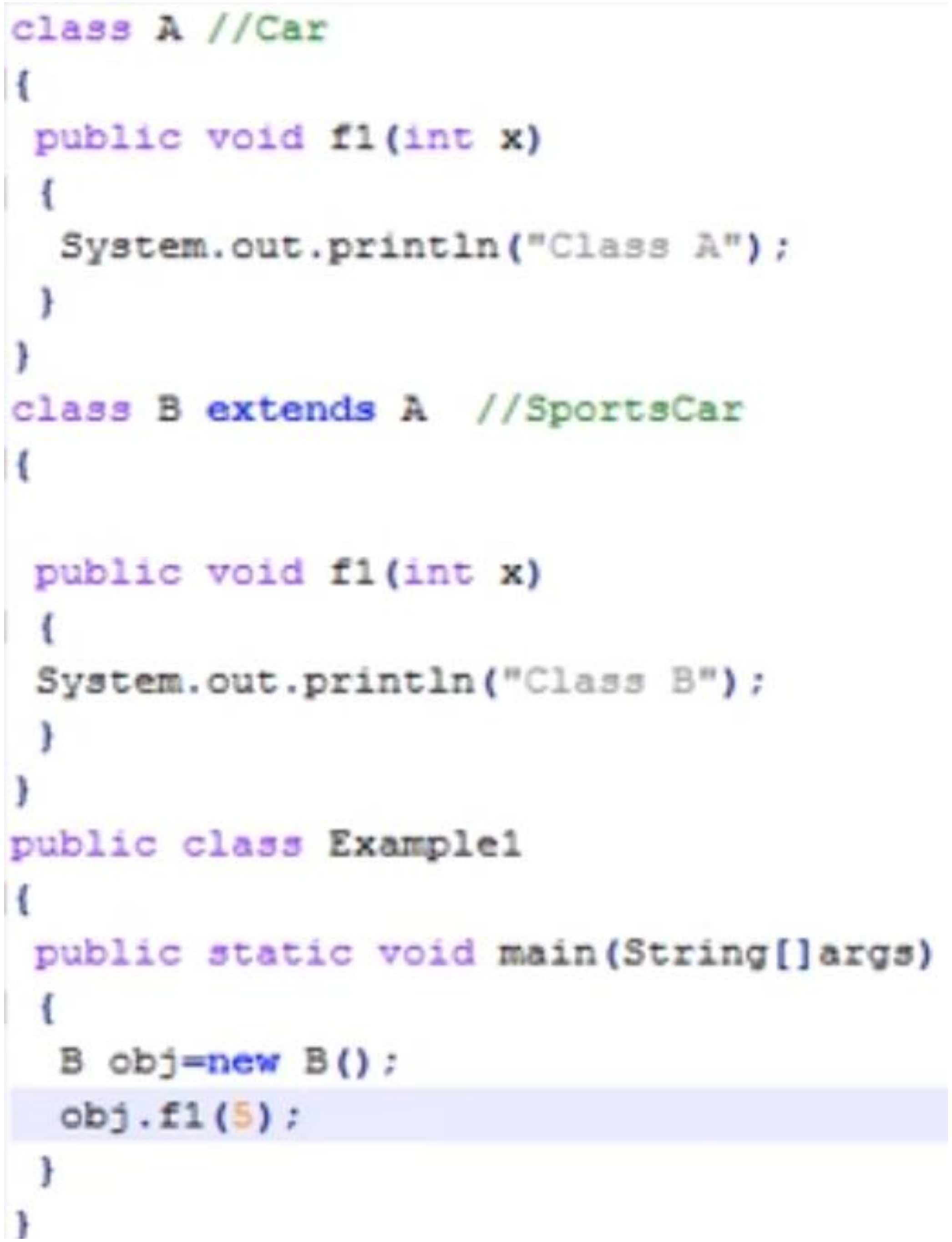
setAlarm()
setNavigator()
setSafeGuard()
getAlarm()
getNavigator()
getSafeGuard()
setPrice()
setFuelType()
setEngine()
setColour()
setCapacity()
getPrice()
getFuelType()
getEngine()
getColour()
getCapacity()



Overriding in Java

- Method overriding is defining a method in subclass with the same signature with specific implementation in respect to the subclass.

- Why Overriding?



```
class A //Car
{
 public void f1(int x)
 {
  System.out.println("Class A");
 }
}
class B extends A  //SportsCar
{

 public void f1(int x)
 {
 System.out.println("Class B");
 }
}
public class Example1
{
 public static void main(String[]args)
 {
  B obj=new B();
  obj.f1(5);
 }
}
```

Inner Classes in Java

- A java class members are: **i) Variables (instance)**

  ii) Methods

  iii) Class

- A class written within a class is called as inner class.
- The out side class is called as outer class.
- E.g.

```
class Outer
{
        int     x;                        //instance variable
        void f1()                         //method
        {    }
        class Inner                       //inner class
        {    }
}
```

- Inner classes are of 2 types:
  - **Static inner class**
  - **Non-Static inner class**

Static Inner Class –

```
class Outer
{
    static class Inner
    {
            void Function1()
            {
            System.out.print("Function Executed");
            }
    }
}
public class Example
{
    public static void main(String args[])
    {
    Outer.Inner obj1 = new Outer.Inner();
    obj1.Function1();
    }
}
```

- Static inner class object can be created outside the outer class
- syntax: **Outer.Inner OBJ=new Outer.Inner();**
- No need to create instance of Outer object.

Non-Static Inner Class –

- For Non-Static inner class object of outer class must be created first.
- Non Static inner class has access to all members of outer class.

```
class Outer
{
    class Inner
    {
            void Function1()
            {
            System.out.print("Function Executed");
            }
    }
}
public class Example
{
    public static void main(String args[])
    {
    Outer Out=new  Outer();
    Outer.Inner obj1 = Out.new   Inner();
    obj1.Function1();
    }
}
```

- Static inner class has access to only static members of the outer class.
- Non-Static inner class has access to all kinds of members (Static or non Static) of outer class.
- Only privately declared inner class can not be instantiated outside (i.e. Object can not be created).
- If otherwise it is public or protected, it can be instantiated.
- Inner class may be – Private, Public, Protected or Default
- But outer class may be – Public or Default.

Interfaces in Java

- It is similar to class in many aspects.
- Syntax:

```
interface   xyz
{
---------
}
class ABC implements xyz
{
---------
}
```

- We can create variables and methods in interface.
- Functions do not have body in interface.
- interface just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly pubic static final)
- We can not assign code to the methods inside interface (only declaration).
- All the members of the interface are by default public, we can not change it.

- interface fields are by default: public static final
- interface methods are by default abstract.
- E.g. interface xyz

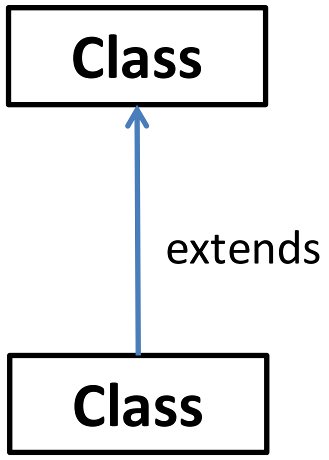
```
{
        void SomeFunction();
}
class ABC implements xyz
{
        public void SomeFunction()
        {
        Function Body code
        }
}
```

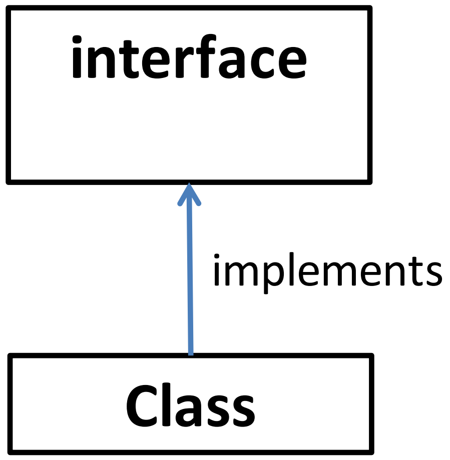
- interfaces can not be instantiated.
- interfaces do not have constructors cause all the variables are by default static (independent of various objects).
- If a class that implements an interface does not define all the methods of interface, then it must be declared abstract, and the method definitions must be provided by the subclass that extends the abstract class.
- We can not create object of interface but reference variable can be created.

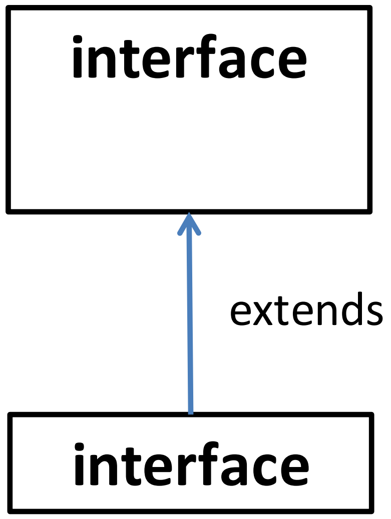
```
interface  I1
{.....}
interface  I2
{.....}
interface  I3  extends I1, I2
{.....}
interface  I4
{.....}
class  A
{.....}
class  B extends A implements I3,I4
{.....}
```

Sample Code:

```
interface I1
{   void F1();    }

interface  I2
{   void  F2();    }

class  A implements I1,I2
{
            public void  F1() { }
            public void  F2() { }
            public void  F3() { }
}

class  Example
{
            public static void main(String args[])
            {
              I1   obj  =  new  A();
              obj.F1();
              obj.F2();              //error
              obj.F3();              //error
            }
}
```


Class
extends
Class

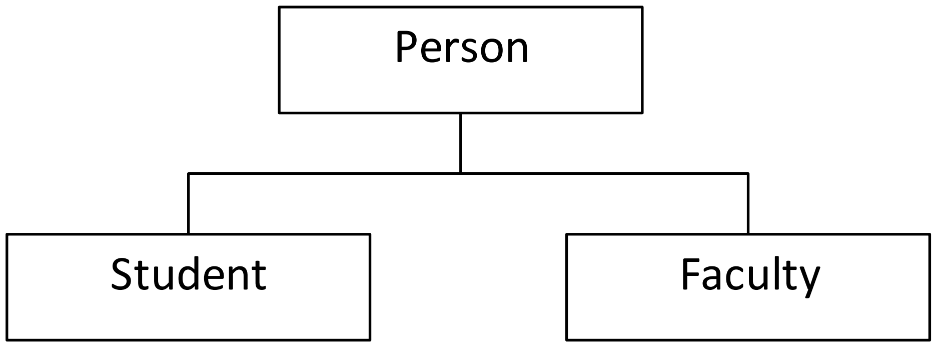

interface
implements
Class


interface
extends
interface

Abstract Class and Abstract Methods

- There is no keyword like abstract in C++.
- To create an abstract class in C++, we have to create at least one pure virtual function in that class.
- Virtual keyword in C++ is absent in Java.
- Abstract keyword in java is absent in C++.
- An abstract class can’t be instantiated, means object of it can’t be created.

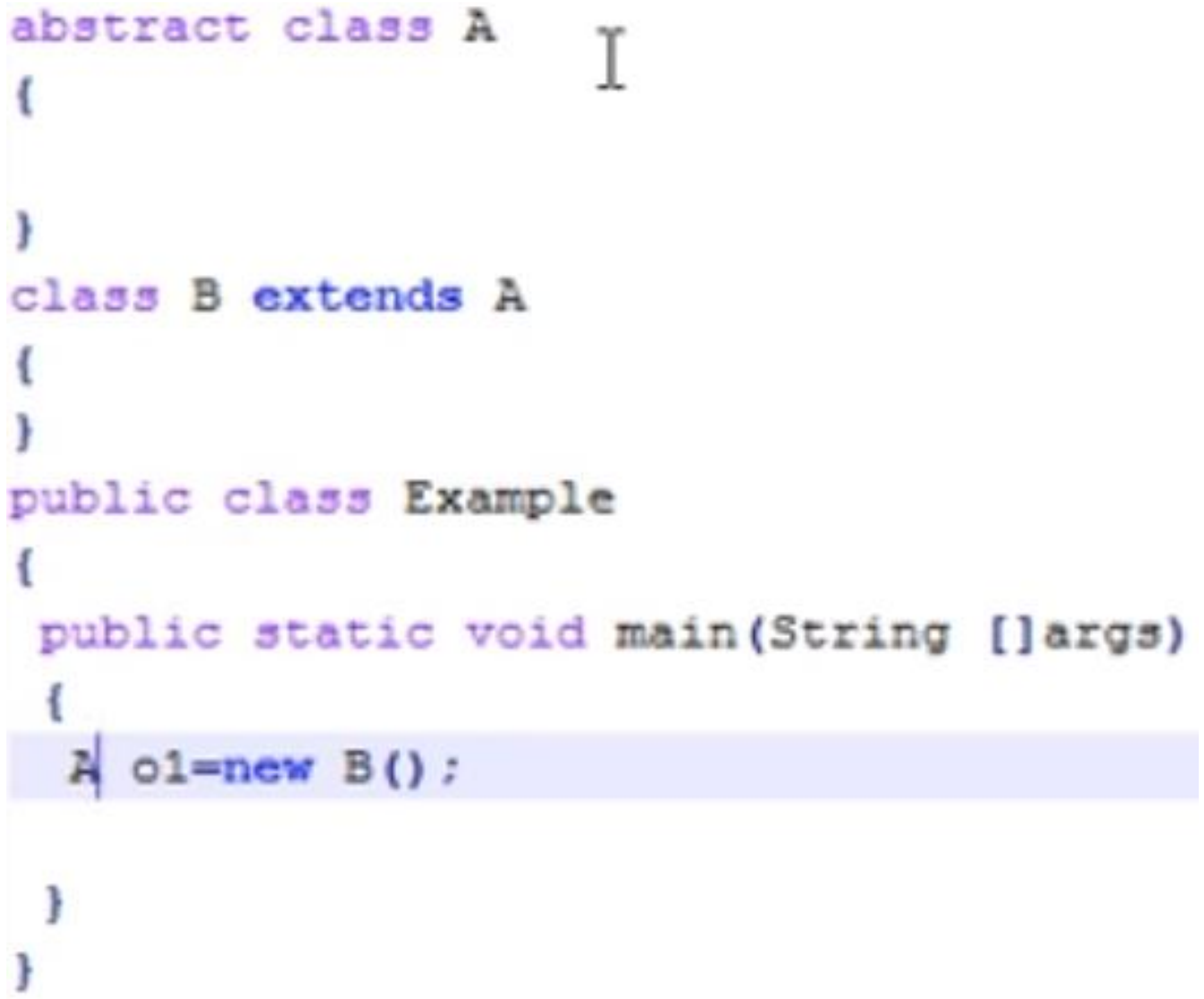
When to create abstract class



- Both Student and Faculty classes inherit some common properties of person class like name, contact no, dob, age etc.
- So we can create object of either student or faculty class to refer to those members in person class.
- But we never need to create object of person class. In such situations person class may be made abstract.
- Abstract classes are used to declare common characteristics of subclasses.

- Constructors of abstract class gets executed when the object of child class is created.
- It can only be used as a superclass for other classes that extend the abstract class.
- It contains variables and functions like any other class, also constructor functions.
- You can not create object of abstract class but we can create a reference variable.

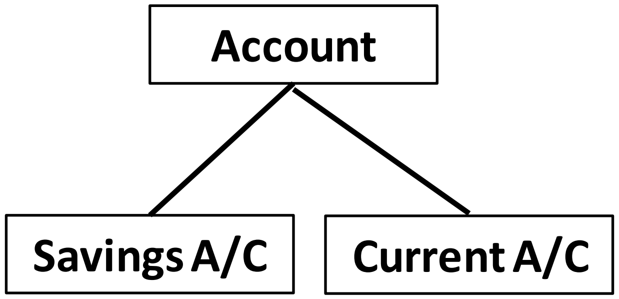
  e.g. person obj = new student();



```
abstract class A
{


}
class B extends A
{
}
public class Example
{
 public static void main(String []args)
 {
  A o1=new B();


 }
}
```

Abstract Methods

- No implementation, only declaration, hence have no body and function declaration ends with a semicolon.
- If a class contains abstract methods then its compulsory for the class to be abstract.
- If a class is declared as abstract then its not mandatory to contain abstract method(s).
- Why to create abstract class?



- In such situations, we might create object of savings a/c or current a/c class but never create object of Account class.
- Such class may be made abstract. Its functions may be overridden by its child classes.

What is wrong with the code?

```
class Person{
 ...
 abstract void show();
}
class Student extends Person{
 ...
}
 class AbstractExample3{
  public static void main(String[] args){
  Student s=new Student();
  }
}
```

```
abstract class Person{
 ...
 abstract void show();
}
abstract class Student extends Person{
 ...
}
 class AbstractExample3{
  public static void main(String[] args){
  Student s=new Student();
  }
}
```

```
abstract class Person{
       abstract void show();
}
class Student extends Person{
 void show()
 { //some code }
}
 class AbstractExample3{
  public static void main(String[] args){
  Student s=new Student();
  s.show();
  }
}
```

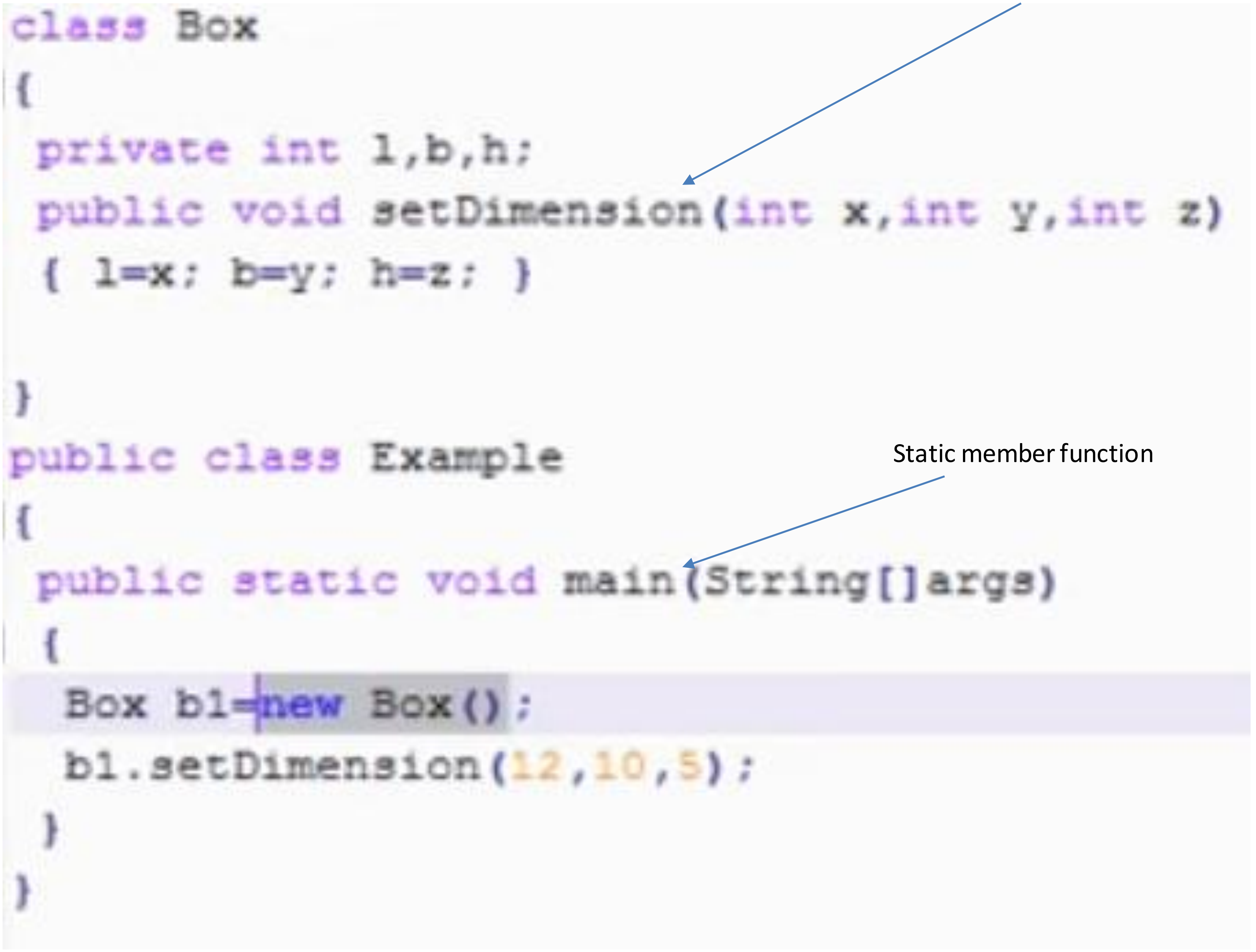
this keyword

- ❑ The this object reference is a local variable in instance member methods referring the caller object

- ❑ this keyword is used as a reference to the current object which is an instance of the current class

- ❑ The this reference to the current object is useful in situations where a local variable hides, or shadows, a field with the same name.

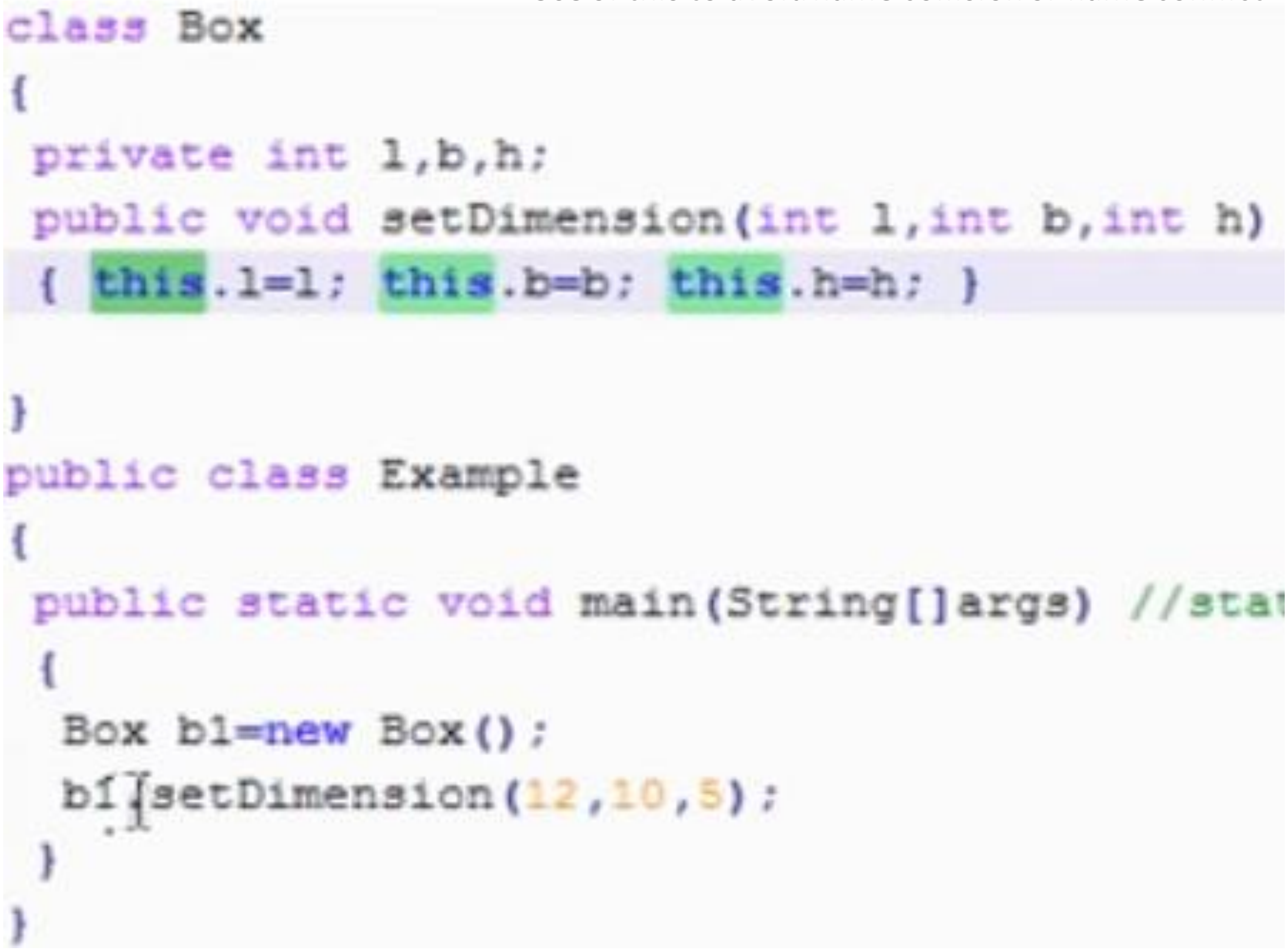
Instance member function



```
class Box
{
 private int l,b,h;
 public void setDimension(int x,int y,int z)
 { l=x; b=y; h=z; }


}
public class Example
{
 public static void main(String[]args)
 {
  Box b1=new Box();
  b1.setDimension(12,10,5);
 }
}
```

Static member function

Use of this to avoid name collision or name conflict



```
class Box
{
 private int l,b,h;
 public void setDimension(int l,int b,int h)
 { this.l=l; this.b=b; this.h=h; }

}
public class Example
{
 public static void main(String[]args) //sta
 {
  Box b1=new Box();
  b1.setDimension(12,10,5);
 }
}
```

Packages in Java

- Reusability
- We need to use classes of other programs in to program under construction
- Packages : similar to class libraries
- Packages: way of grouping variety of classes and/or interfaces together according to functionality.
- Packages act as containers for classes

Packages in Java

- Two classes in tow different packages can have same name.
- Referred by their package name . class name
- Packages provides a way of hiding classes
- Java packages are classified into two types:
  - **Java API packages**
  - **User defined packages**

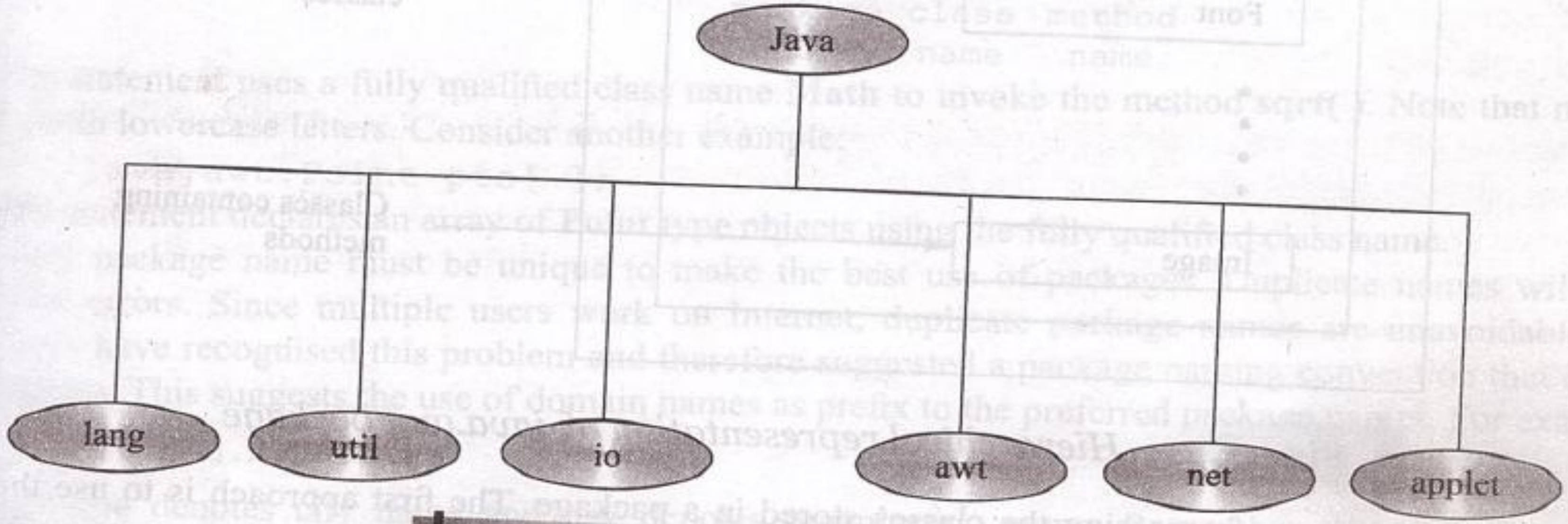


Fig. 11.1 *Frequently used API packages*

- Java.lang – languages support classes, include classes for primitive types, strings, math functions, threads and exceptions etc.
- Java.util – utility classes such as vectors, hash tables, random numbers etc.
- Java.io – input/output support classes
- Java.awt – classes for implementing graphical user interface, classes for windows, buttons, lists, menus etc.
- Java.net – classes for networking
- Java.applet – classes for creating and implementing applets.

Java Packages

- Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to

- Files in one directory (or package) would have different functionality from those of another directory.

Import Statement

import package_name.class_name;

Or

import package_name.*;

$$\text{e.g.: double y=java.}\underbrace{\text{lang}}_{\text{Package name}}\text{.}\underbrace{\text{Math}}_{\text{Class name}}\text{.}\underbrace{\text{sqrt}}_{\text{Method name}}\text{(x);}$$

Creating Package

```
package first_package;                //package declaration
Public class First_class              //class definition
    {
        ……..
        …….
        …….
     }
```

Creating Package

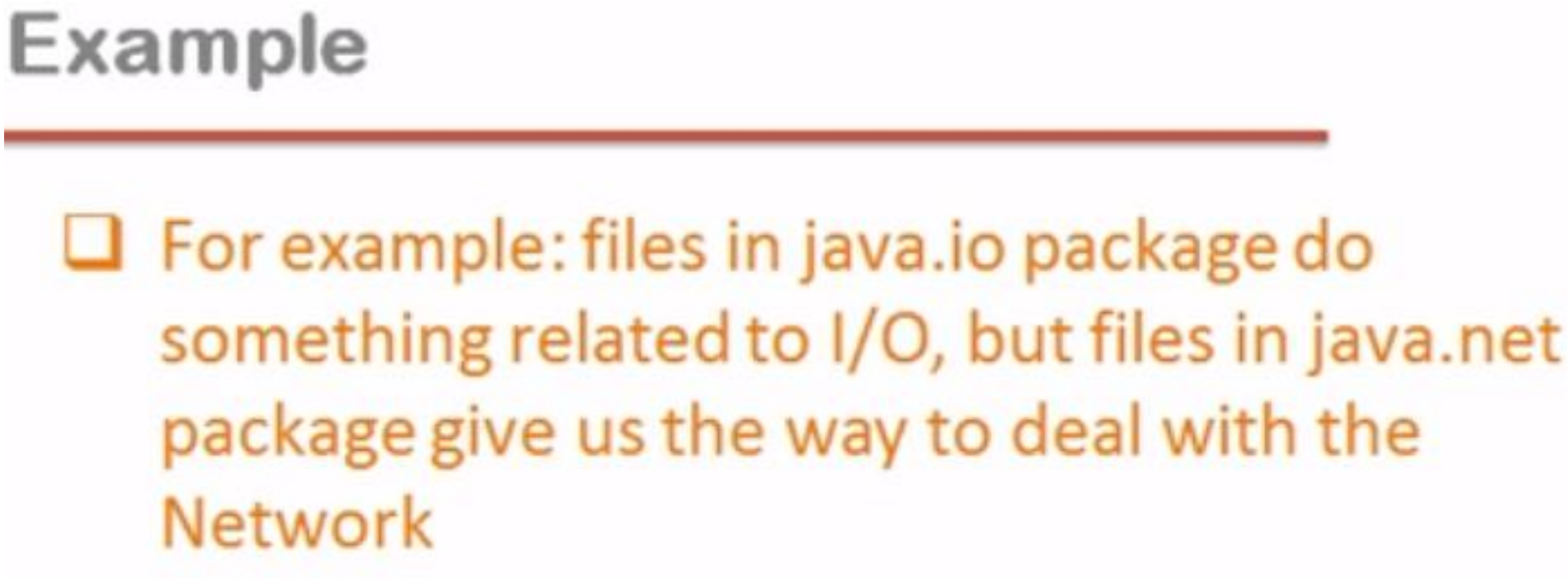
- Declare a package at the beginning of the file using the form:

  package package_name;

- Define the class that is to be put in the package and declare it to be public.
- Create a directory under the subdirectory where the main source files are located.
- Store the listing as the class_name. java file in the subdirectory created.
- Compile the file. This creates .class file in the subdirectory.
- Remember, case is significance.
- Java also supports the concept of package hierarchy.

  e.g. : package first_package.second_package

- A Single java package file may have more than one class definition. In such case only one of the classes may be declared public and that class name with .java extension is the source file.
- When a source file with more than one class definitions is compiled, java creates independent .class files for those classes.



Example

- For example: files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network

Name Collision

- Packaging also help us to avoid class name collision when we use the same class name as that of others

- The benefits of using package reflect the ease of maintenance, organization, and increase collaboration among developers

How to create package?

- Suppose we have a file called HelloWorld.java, and we want to put this file in a package **world**

How to create package?

```
package world;
public class HelloWorld {
 public static void main(String[] args) {
     System.out.println("Hello World");
  }
}
```

- ❑ Now compile this file as

```
path> javac –d . HelloWorld.java
```

Example

```
package package1;
public class ClassA
{
        public void displayA()
        {
        System.out.println("Class A");
        }
}
```

Source File : ClassA.java stored in subdirectory package1
Compile: ClassA.class will be created at same location

```
import package1.ClassA;
class PackageTest1
{
        public static void main(String[] args)
        {
        ClassA objectA = new ClassA();
        objectA.displayA();
        }
}
```

Example

```
package package2;
public class ClassB
{
protected int m=10;
public void displayB()
{
System.out.println("Class B");
System.out.println("m :="+m);
}
}
```

```
//Importing classes from other packages
import package1.ClassA;
import package2.ClassB;
class PackageTest2
{
public static void main(String args[])
        {
        ClassA objectA = new ClassA();
        ClassB objectB = new ClassB();
        objectA.displayA();
        objectB.displayB();
        }
}
//Save as PackageTest2.java
OutPut: Class A
        Class B
        m=10
```

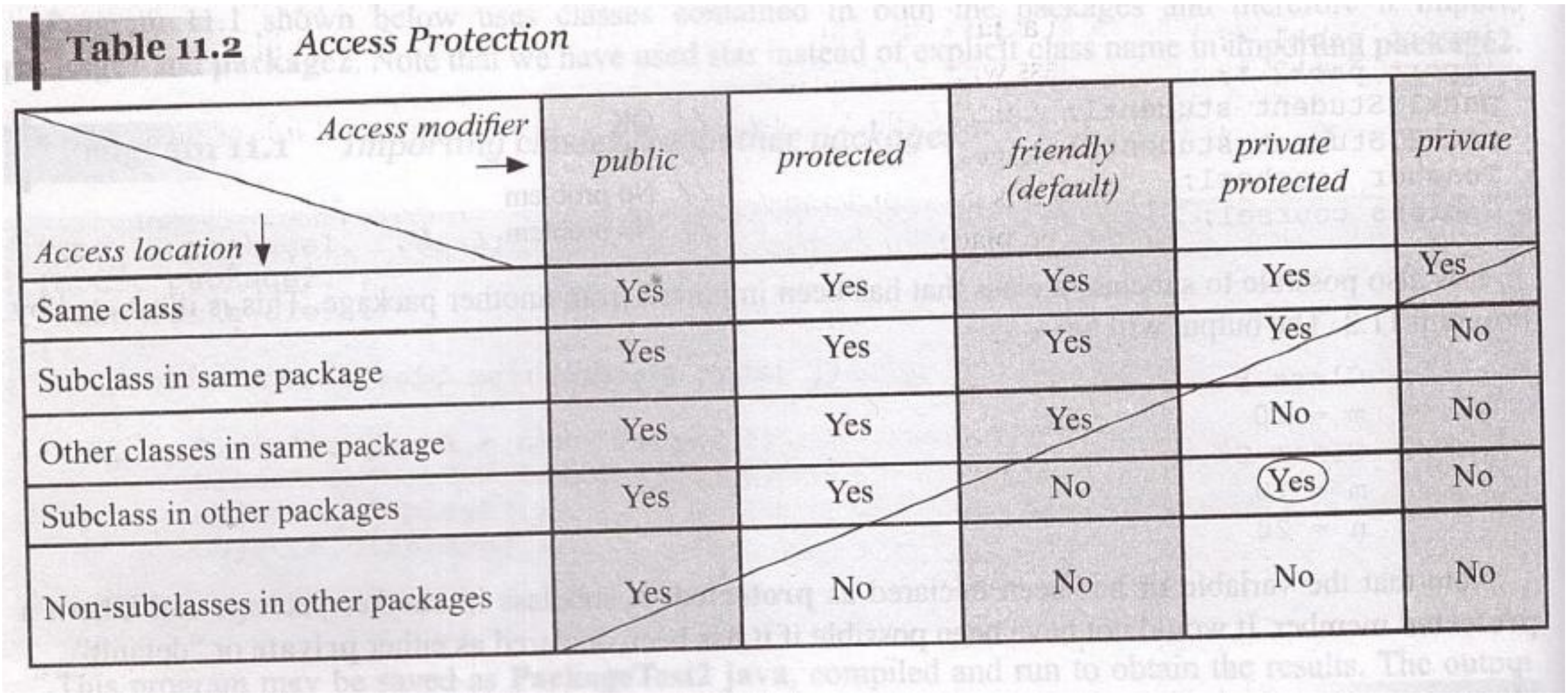


Table 11.2 *Access Protection*

Access modifier → / *Access location* ↓	*public*	*protected*	*friendly (default)*	*private protected*	*private*
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

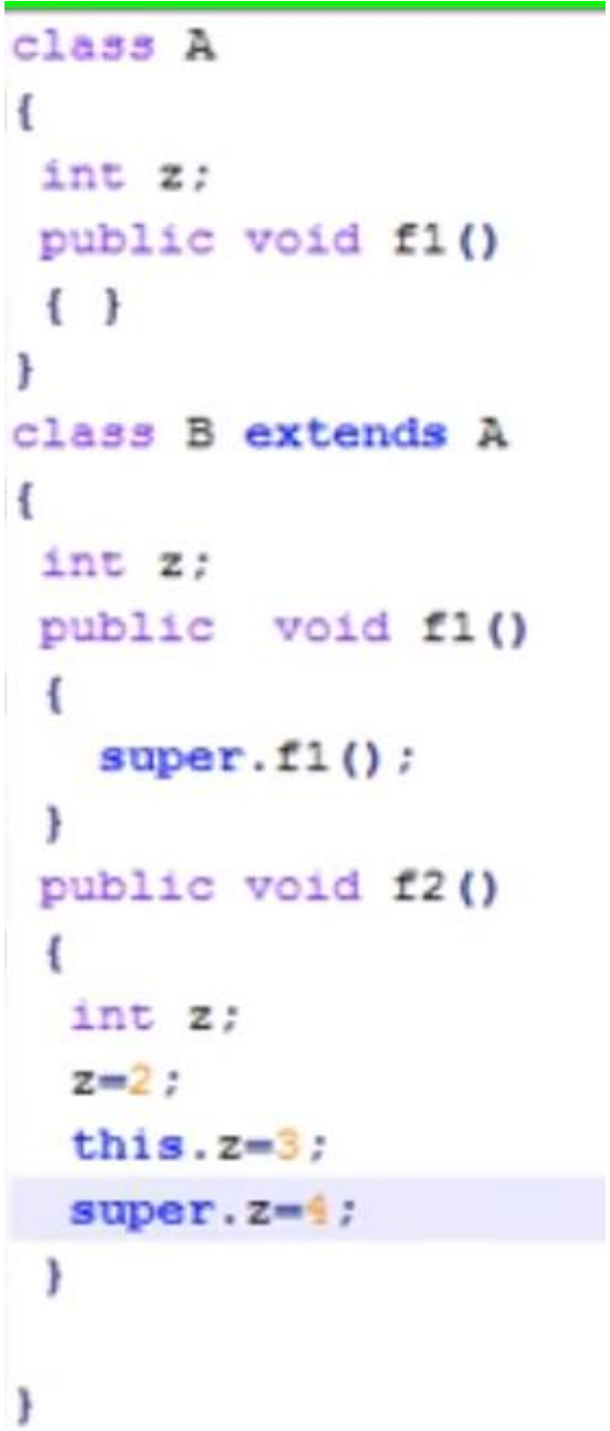
super keyword

- In inheritance, subclass object when call an instance member function of subclass only, function contains implicit reference variables this and super both referring to the current object (Object of subclass).
- The only difference in this and super is
  - this reference variable is of subclass type
  - super reference variable is of superclass type

Use of super keyword

- If your method overrides one of its superclass's methods, you can invoke the superclass version of the method through the use of the keyword super.

- It avoids name conflict between member variables of superclass and subclass



```
class A
{
 int z;
 public void f1()
 { }
}
class B extends A
{
 int z;
 public  void f1()
 {
   super.f1();
 }
 public void f2()
 {
  int z;
  z=2;
  this.z=3;
  super.z=4;
 }


}
```

```
class Example
{
 public static void main(String[]args)
 {
  B obj=new B();
  obj.f1();
  obj.f2();


 }
}
```

Initialization block in JAVA

- There are two types of initialization blocks
  - Instance Initialization Block
  - Static Initialization Block

Instance Initialization Block

```
public class Test {
 private int x;
 {
  System.out.println("Initialization Block: x="+x);
  x=5;
 }
public Test()  {
  System.out.println("Constructor: x="+x);
 }
public static void main(String []args)  {
   Test t1=new Test();
   Test t2=new Test();
 }
}
```

Instance Initialization Block

- ❑ An *instance initializer or Initialization block* declared in a class is executed when an instance of the class is created
- ❑ return keyword cannot be used in Initialization block
- ❑ Instance initializers are permitted to refer to the current object via the keyword this and to use the keyword super

Static initialization block

```
public class Test
{
  private static int k;
  static
  {
    System.out.println("Static Initialization Block: k="+k);
    k=10;
  }
  public static void main(String []args)
  {
    new Test();
  }
}
```

Static initialization block

- ❑ A *static initializer* declared in a class is executed when the class is initialized
- ❑ Static initializers may be used to initialize the class variables of the class
- ❑ return keyword cannot be used in static Initialization block
- ❑ this or super can not be used in static block

The final keyword

- ❑ **final** instance variable
- ❑ **final** static variable
- ❑ **final** local variable
- ❑ **final** class
- ❑ **final** methods

final instance variable

- ❑ A java variable can be declared using the keyword final. Then the final variable can be assigned only once.

- ❑ A variable that is declared as final and not initialized is called a blank final variable. A blank final variable forces either the constructors to initialize it or initialization block to do this job.

final static variable

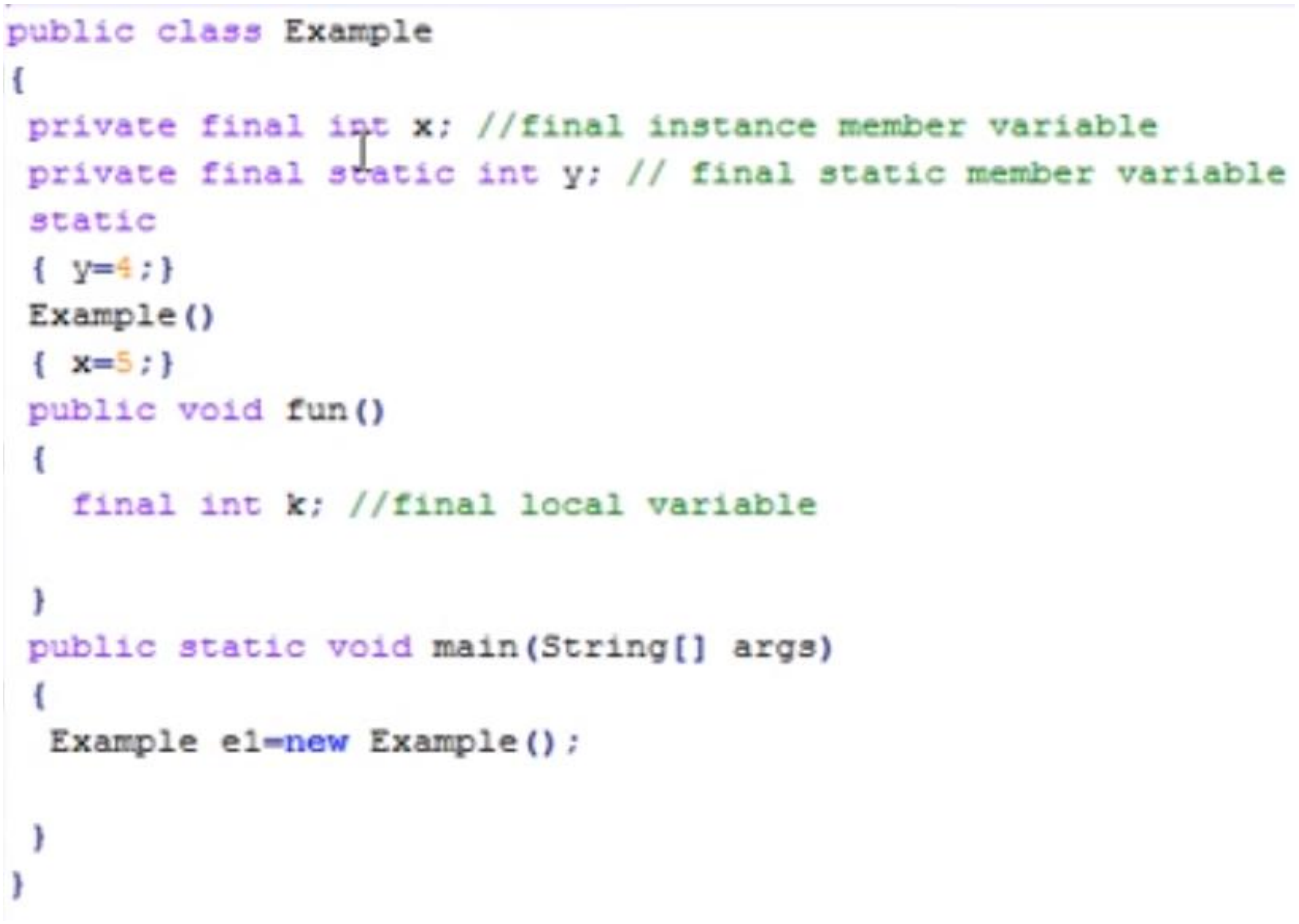
- ❑ Static member variable when qualified with final keyword, it becomes blank until initialized.

- ❑ Final static variable can be initialized during declaration or within the static block

final local variable

- Local variables that are final must be initialized before it’s use, but you should remember this rule is applicable to non final local variables too.

- Once they are initialized, can not be altered.



```
public class Example
{
 private final int x; //final instance member variable
 private final static int y; // final static member variable
 static
 { y=4;}
 Example()
 { x=5;}
 public void fun()
 {
   final int k; //final local variable


 }
 public static void main(String[] args)
 {
  Example e1=new Example();


 }
}
```

final class

- Java classes declared as final cannot be extended. Restricting inheritance!

final methods

- Methods declared as final cannot be overridden

Memory management in C++

- Static memory allocation: members (variable or object) declared and initialized

  Scope: block of code in which created
- Dynamic memory allocation: members (variable or object) created with ‘new’ operator

  Scope: till the end of program
- Free and Consumed memory area
- Memory Leak
- ‘Delete‘ in c++

In Java

- In Java destruction of object from memory is done automatically by the JVM.
- No delete keyword in Java
- When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released
- This technique is called **Garbage Collection**
- This is accomplished by the JVM.

JVM threads

- Whenever you run a java program, JVM creates three threads.
  - main thread
  - Thread Scheduler
  - Garbage Collector Thread.
- In these three threads, main thread is a user thread and remaining two are daemon threads which run in background.

Three threads in Java

- The task of main thread is to execute the main() method.
- The task of thread scheduler is to schedule the threads.
- The task of garbage collector thread is to sweep out abandoned objects from the heap memory.

Garbage collector thread

- Abandoned objects or dead objects are those objects which does not have live references.

Advantage of garbage collection

- Increases memory efficiency and decreases the chances for memory leak.

finalize method

- Garbage collector thread before sweeping out an abandoned object, it calls finalize() method of that object.

- After finalize() method is executed, object is destroyed from the memory.

Cannot force for garbage collection

- We can call garbage collector explicitly using **System.gc()** or **RunTime.getRunTime().gc()**.
- But, it is just a request to garbage collector not a command.
- It is up to garbage collector to honour this request.

Object's memory

- ❑ An object is created in the memory using **new** operator. Constructor is used to initialize the properties of that object.

- ❑ When an object is no more required, it must be removed from the memory so that that memory can be reused for other objects.

Finalize method

- finalize() method is a protected and non-static method of **java.lang.Object** class.

```
protected void finalize() throws Throwable
{
        //code here
}
```

Garbage Collector Thread

- Garbage collector thread before sweeping out an abandoned object, it calls finalize() method of that object.

finalize method

- That means clean up operations which you have kept in the finalize() method are executed before an object is destroyed from the memory.

- Garbage collector thread calls finalize() method only once for one object.

No exception in finalize

- Exceptions occurred in finalize() method are not propagated. They are ignored by the garbage collector

- finalize() methods are not chained like constructors i.e there is no calling statement to super class finalize() method inside the finalize() method of sub class. You need to explicitly call super class finalize() method.

Force execution of finalize

- ❑ You can make finalize() method to be executed forcefully using
  - eitherRuntime.getRuntime().runFinalization()
  - or Runtime.runFinalizersOnExit(true).
- ❑ But, both the methods have disadvantages. Runtime.getRuntime().runFinalization() makes the just best effort to execute finalize() method. It is not gauranteed that it will execute finalize() method.
- ❑ Runtime.runFinalizersOnExit(true) is deprecated in JDK because some times it run finalize() method on live objects also.