RAMAIAH
Institute of Technology

Welcome to course on Programming in Java

Unit -2

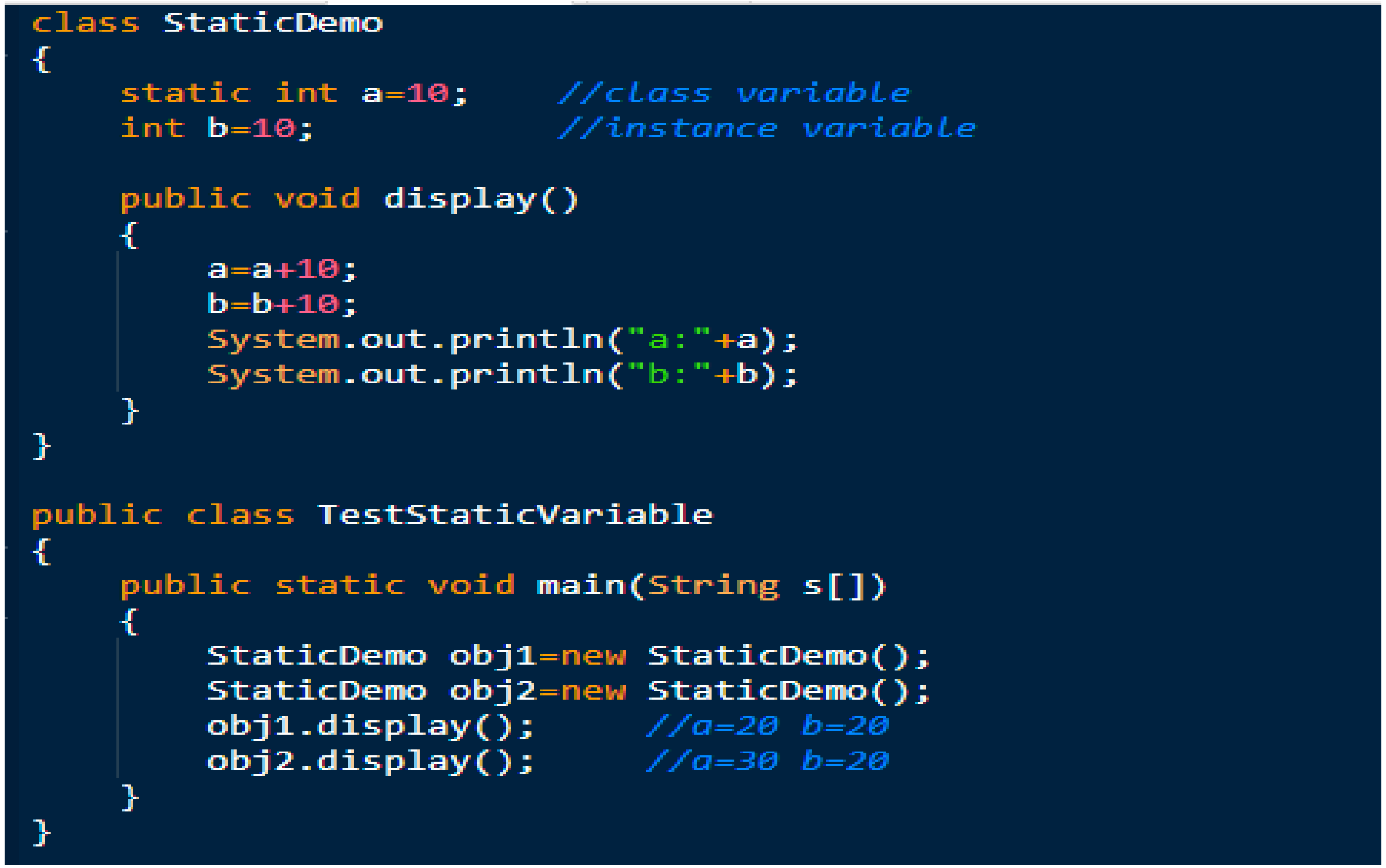
Java Programming Fundamentals: Constructors, The this Keyword, Garbage Collection, The finalize( ) Method, Overloading Methods, Using Objects as

Parameters, A Closer Look at Argument Passing, Returning Objects, Introducing Access Control, Understanding static, Introducing final, Introducing Nested and Inner Classes.

Understanding static

- We can define a class member that will be used independently of any object of that class.
- When a member (variable or method) is declared static, it can be accessed before any objects of its class are created.
- Methods declared as static have several restrictions
  - They can only directly call other static methods.
  - They can only directly access static data.
  - They cannot refer to this or super in any way.

Understanding static Variable



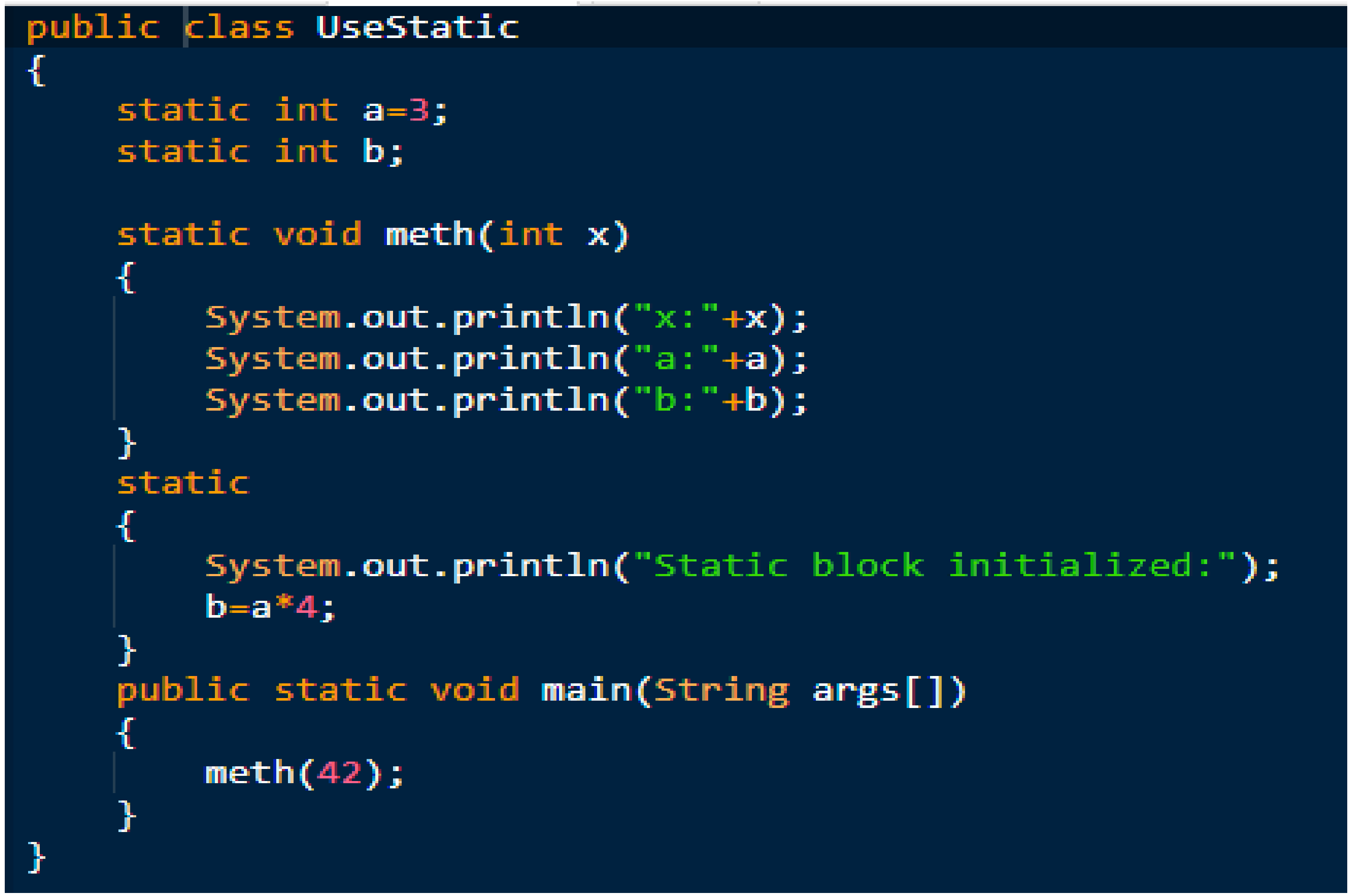
```
class StaticDemo
{
    static int a=10;    //class variable
    int b=10;           //instance variable

    public void display()
    {
        a=a+10;
        b=b+10;
        System.out.println("a:"+a);
        System.out.println("b:"+b);
    }
}

public class TestStaticVariable
{
    public static void main(String s[])
    {
        StaticDemo obj1=new StaticDemo();
        StaticDemo obj2=new StaticDemo();
        obj1.display();     //a=20 b=20
        obj2.display();     //a=30 b=20
    }
}
```

OUTPUT

```
a:20
b:20
a:30
b:20
```

TestStaticVariable.java

Understanding static block



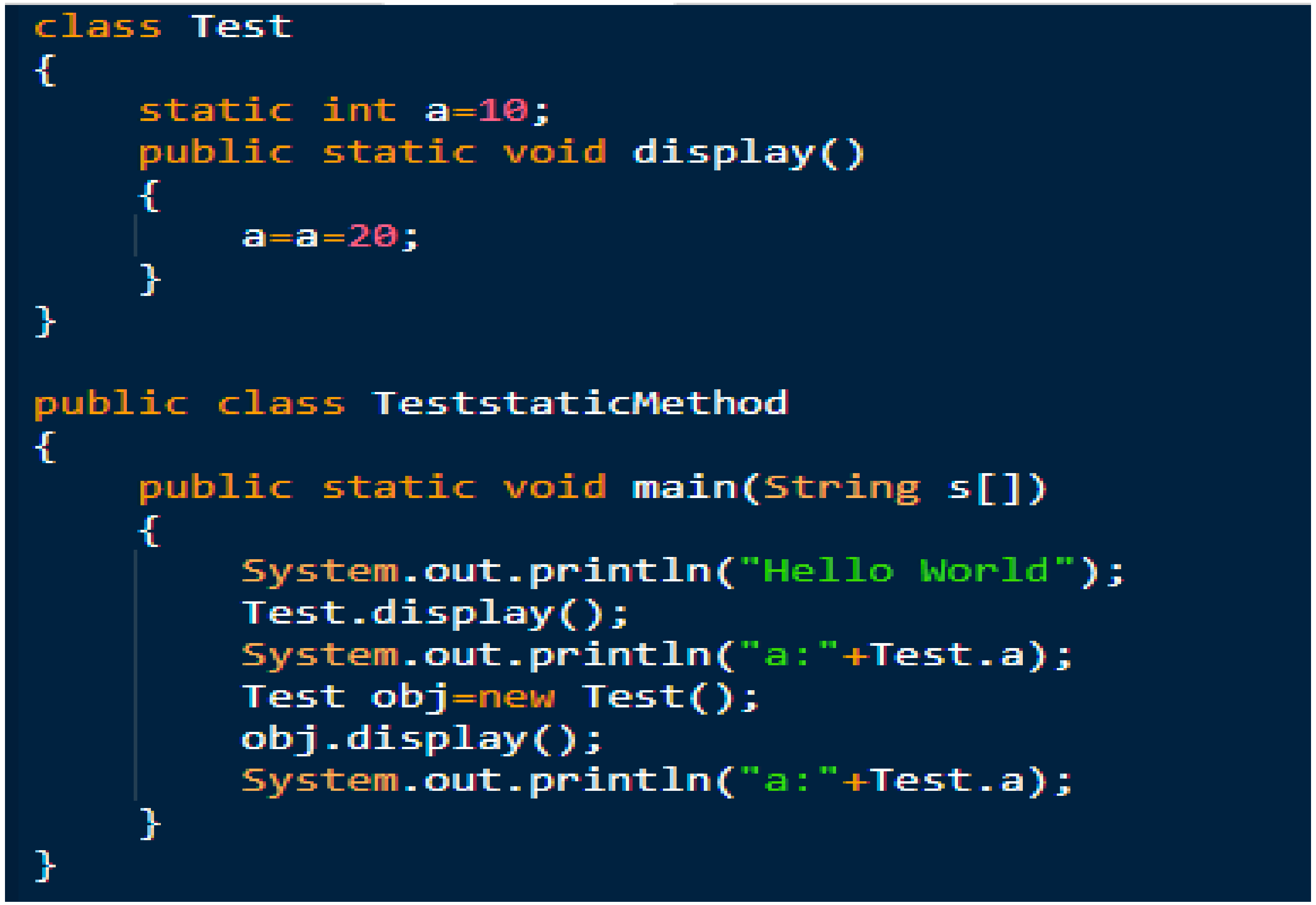
```
public class UseStatic
{
    static int a=3;
    static int b;

    static void meth(int x)
    {
        System.out.println("x:"+x);
        System.out.println("a:"+a);
        System.out.println("b:"+b);
    }
    static
    {
        System.out.println("Static block initialized:");
        b=a*4;
    }
    public static void main(String args[])
    {
        meth(42);
    }
}
```

OUTPUT

```
Static block initialized:
x:42
a:3
b:12
```

UseStaticBlockjava

Understanding static method



```
class Test
{
     static int a=10;
     public static void display()
     {
          a=a=20;
     }
}

public class TeststaticMethod
{
     public static void main(String s[])
     {
          System.out.println("Hello World");
          Test.display();
          System.out.println("a:"+Test.a);
          Test obj=new Test();
          obj.display();
          System.out.println("a:"+Test.a);
     }
}
```

OUTPUT

```
Hello World
a:20
a:20
```

TestStaticMethod.java

Constructors in Java

- In Java, a constructor is a block of codes similar to the method.
- It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.

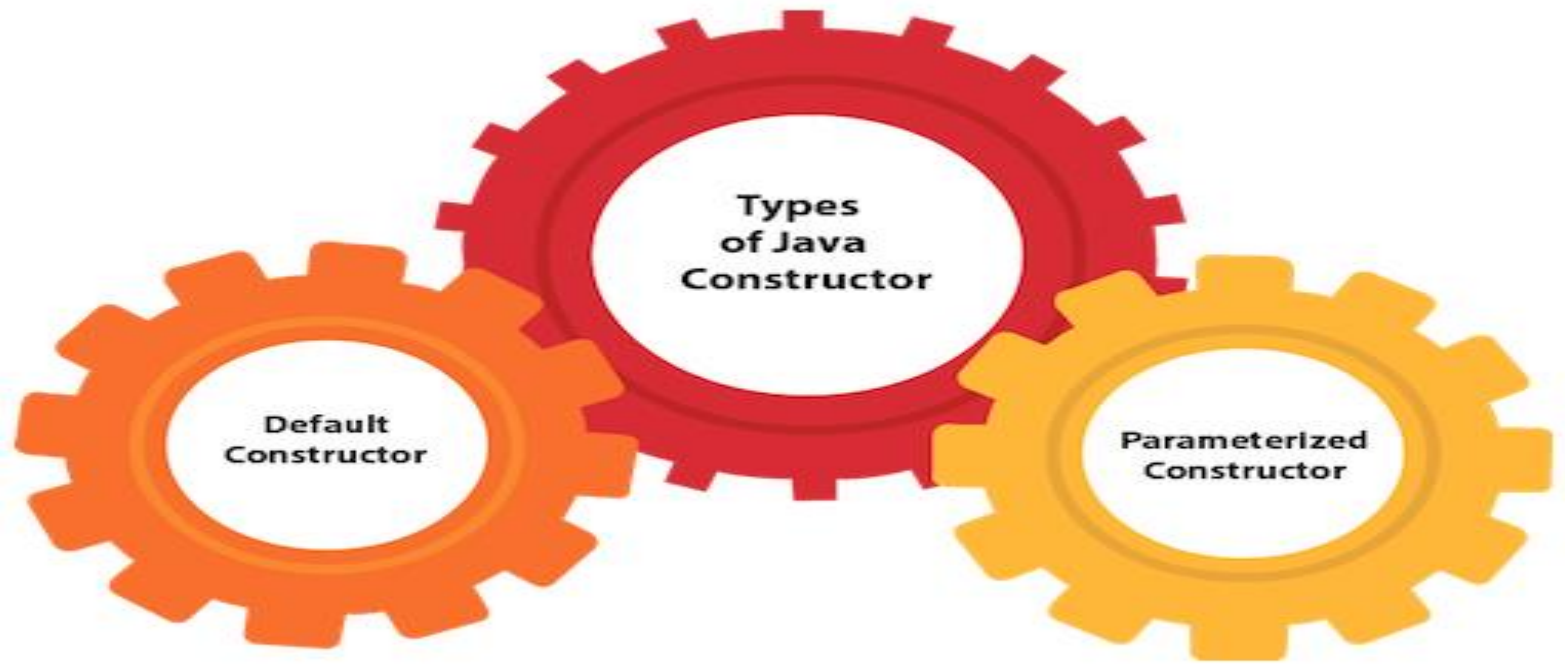
Constructors in Java

- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- There are two types of constructors in Java: **no-arg constructor, and parameterized constructor.**
- **Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for Creating Constructors in java

- There are two rules defined for the constructor.
- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized
- **Note:** We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Constructors in java



RAMAIAH
Institute of Technology

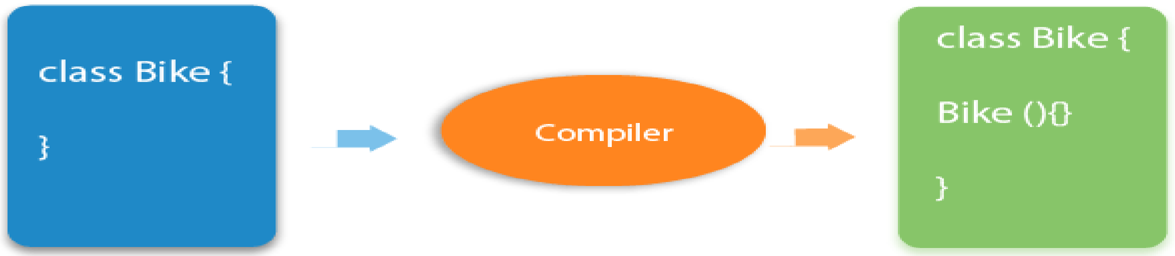
Default Constructors in java

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

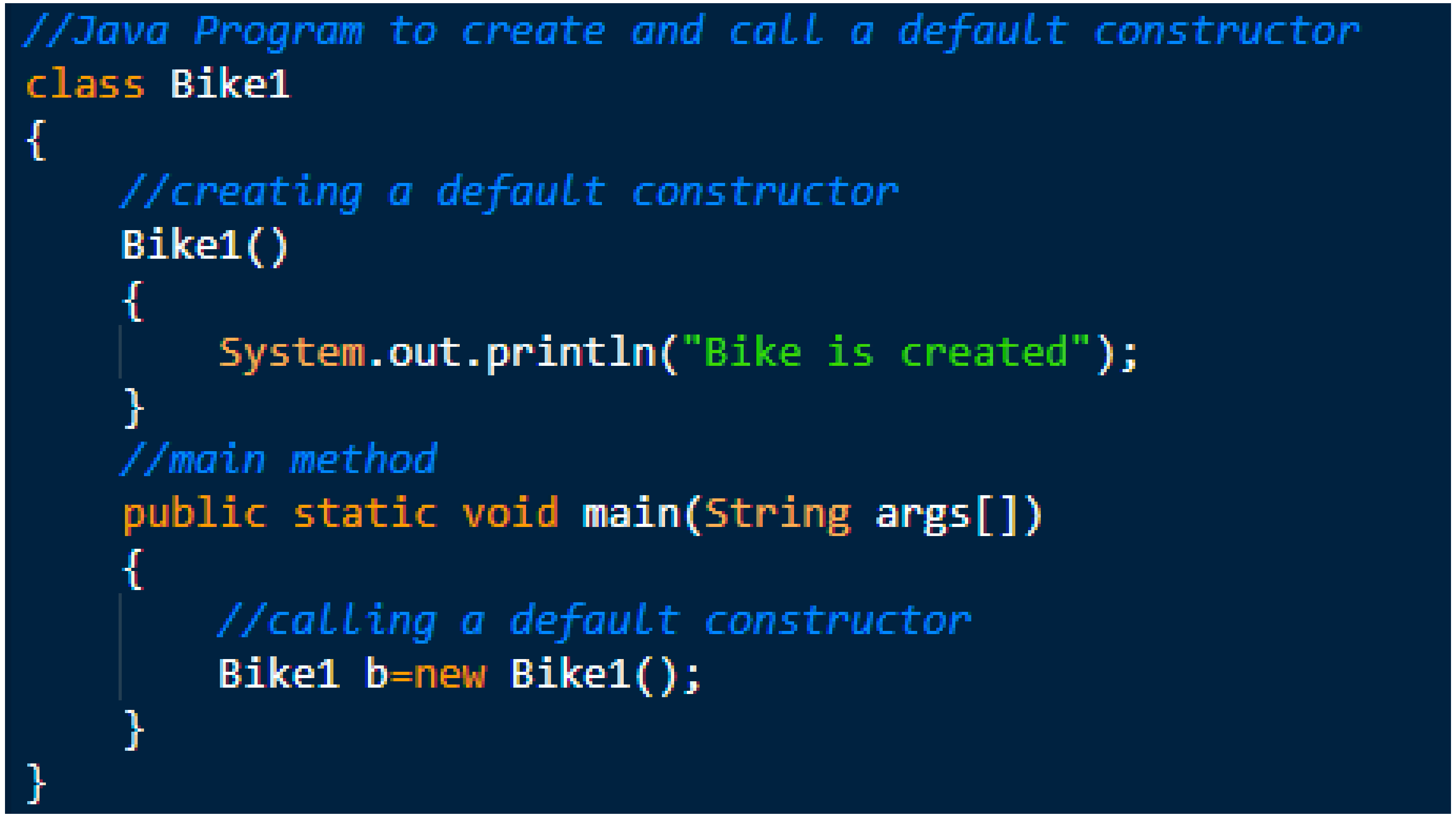
```
class_name()
{

}
```

If there is no constructor in a class, compiler automatically creates a default constructor.



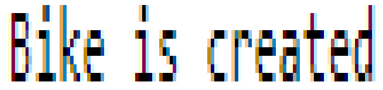
RAMAIAH
Institute of Technology

Default Constructors in java



```
//Java Program to create and call a default constructor
class Bike1
{
    //creating a default constructor
    Bike1()
    {
        System.out.println("Bike is created");
    }
    //main method
    public static void main(String args[])
    {
        //calling a default constructor
        Bike1 b=new Bike1();
    }
}
```

OUTPUT

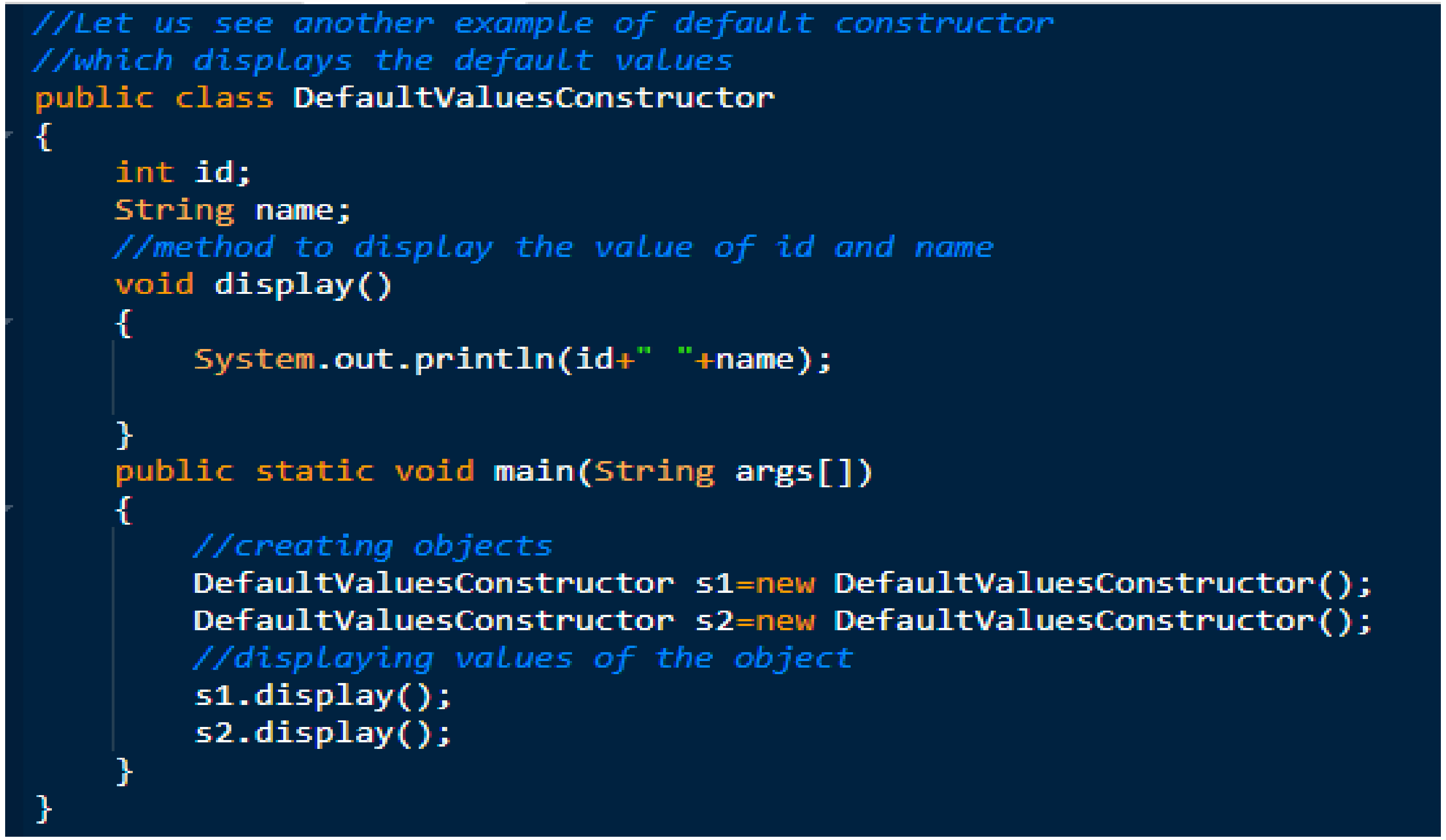


```
Bike is created
```

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Bike1.java

Default Constructors in java



```
//Let us see another example of default constructor
//which displays the default values
public class DefaultValuesConstructor
{
    int id;
    String name;
    //method to display the value of id and name
    void display()
    {
        System.out.println(id+" "+name);

    }
    public static void main(String args[])
    {
        //creating objects
        DefaultValuesConstructor s1=new DefaultValuesConstructor();
        DefaultValuesConstructor s2=new DefaultValuesConstructor();
        //displaying values of the object
        s1.display();
        s2.display();
    }
}
```

OUTPUT

```
0 null
0 null
```

In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

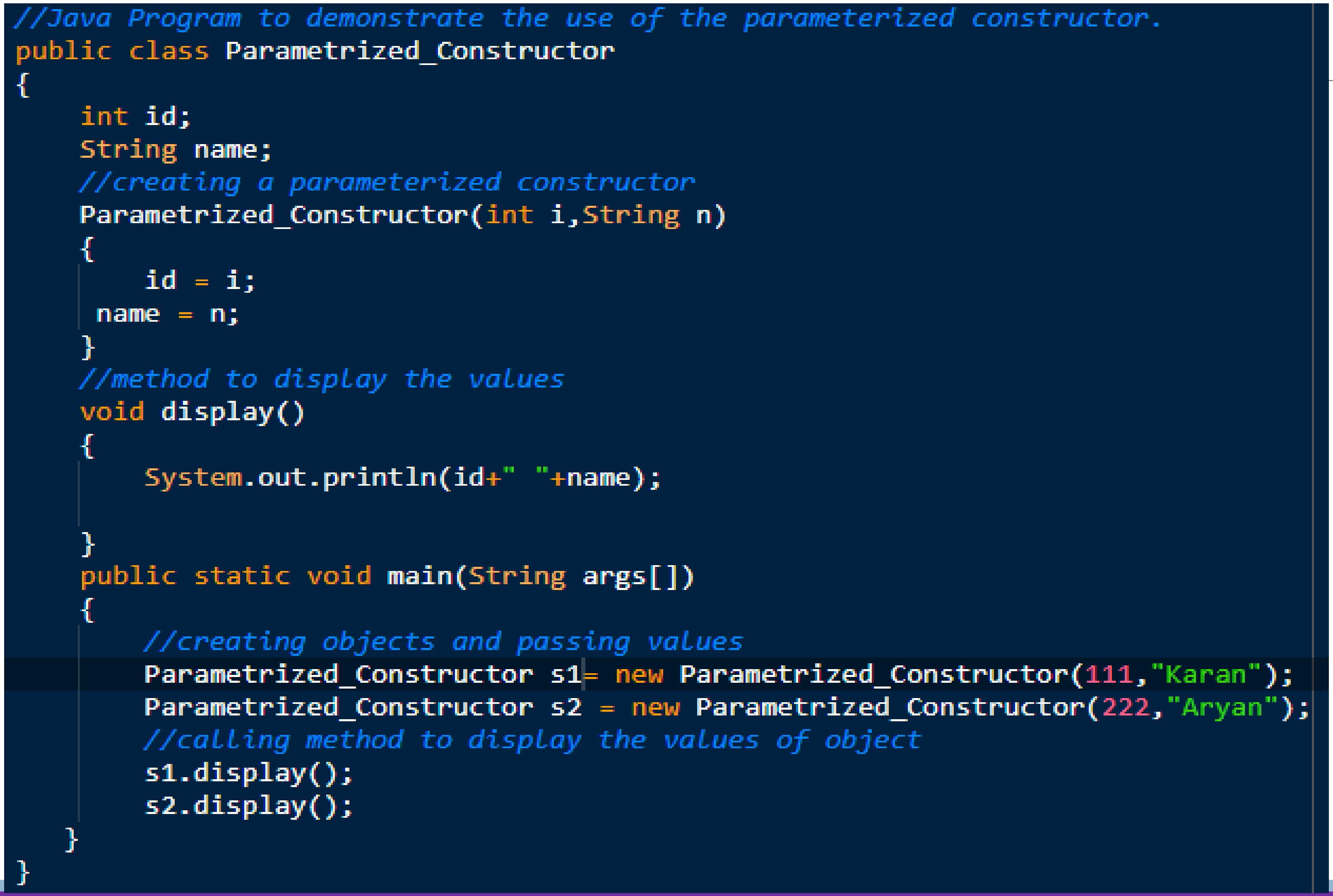
DefaultvaluesConstructor.java

Parameterized Constructors in java

- A constructor which has a specific number of parameters is called a parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

RAMAIAH
Institute of Technology

Parameterized Constructors in java



```
//Java Program to demonstrate the use of the parameterized constructor.
public class Parametrized_Constructor
{
    int id;
    String name;
    //creating a parameterized constructor
    Parametrized_Constructor(int i,String n)
    {
        id = i;
     name = n;
    }
    //method to display the values
    void display()
    {
        System.out.println(id+" "+name);

    }
    public static void main(String args[])
    {
        //creating objects and passing values
        Parametrized_Constructor s1= new Parametrized_Constructor(111,"Karan");
        Parametrized_Constructor s2 = new Parametrized_Constructor(222,"Aryan");
        //calling method to display the values of object
        s1.display();
        s2.display();
   }
}
```

OUTPUT

```
111 Karan
222 Aryan
```

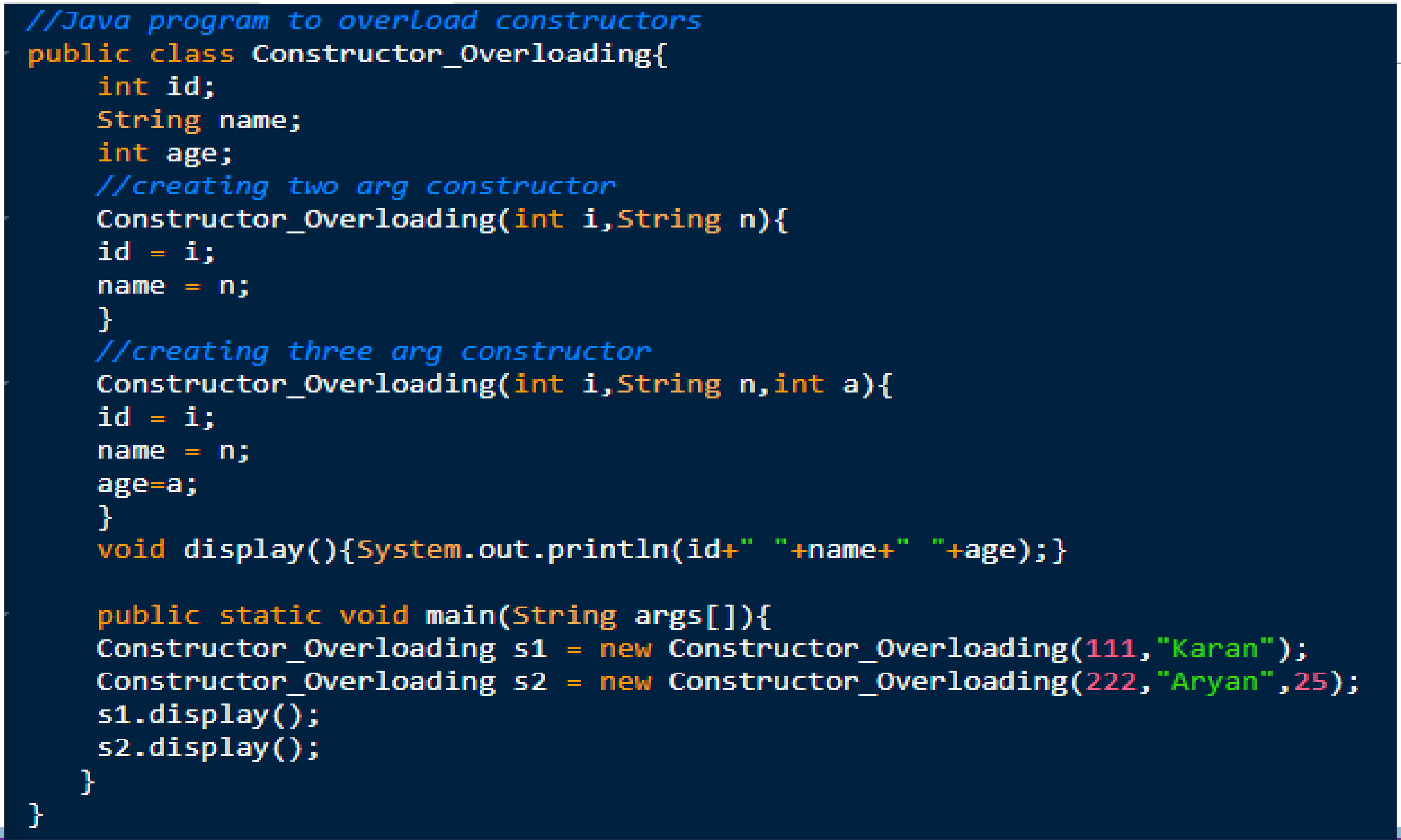
Parametrized_Constructor.java

Constructor Overloading in java

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

RAMAIAH
Institute of Technology

Constructor Overloading in java



```java
//Java program to overload constructors
public class Constructor_Overloading{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Constructor_Overloading(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Constructor_Overloading(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Constructor_Overloading s1 = new Constructor_Overloading(111,"Karan");
    Constructor_Overloading s2 = new Constructor_Overloading(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

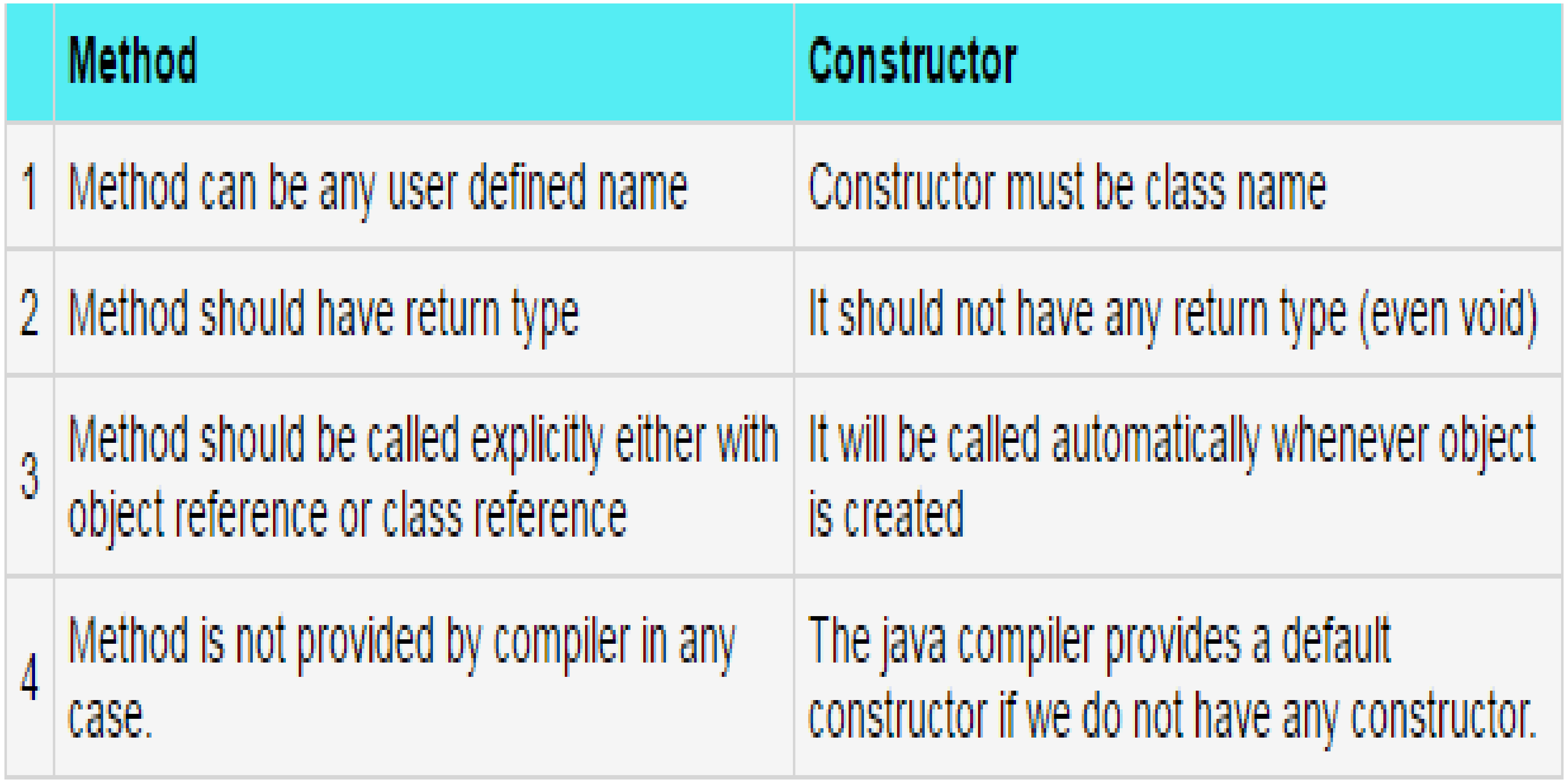
OUTPUT

```
111 Karan 0
222 Aryan 25
```

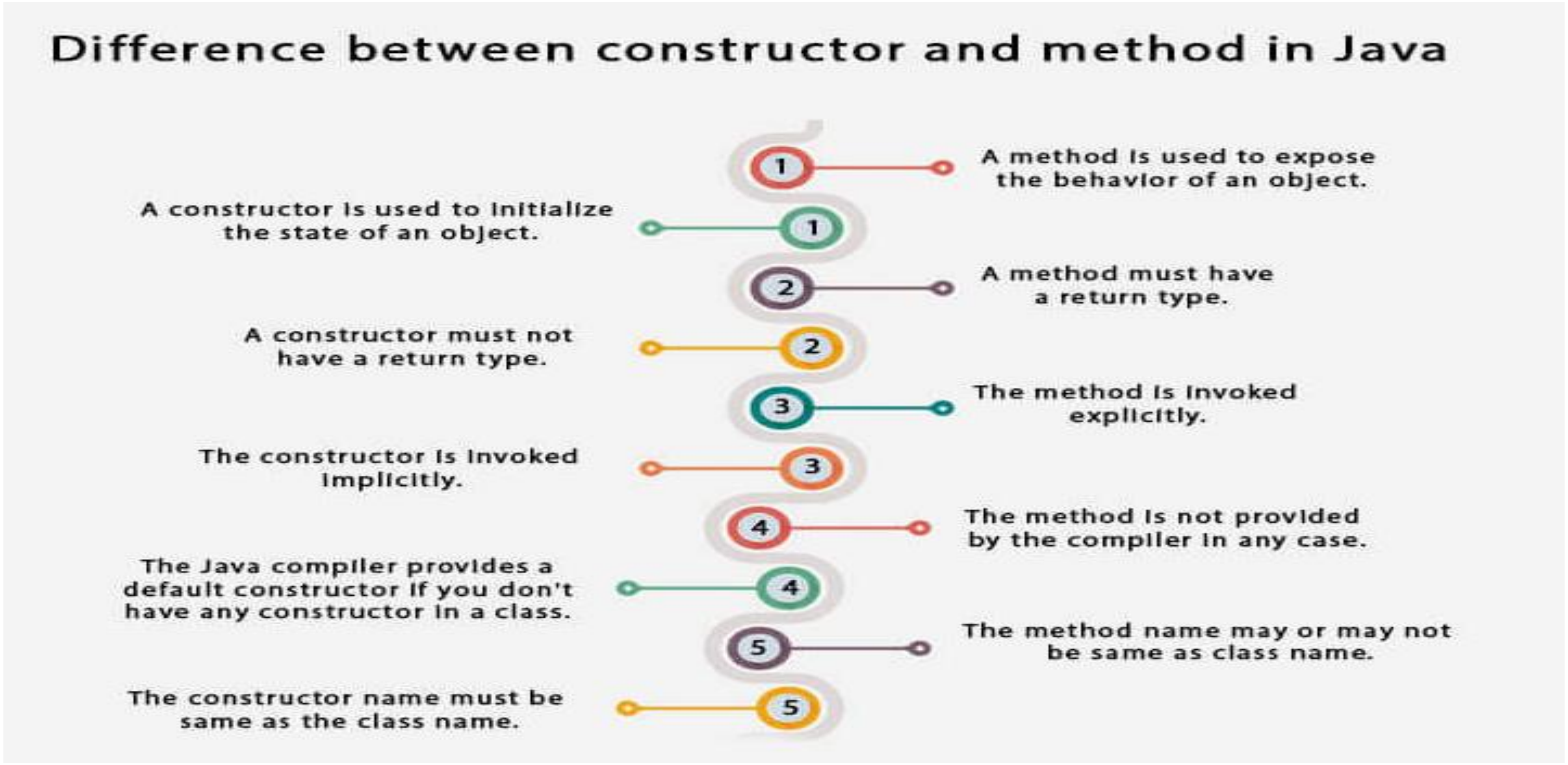
Constructor_Overloading.java

RAMAIAH
Institute of Technology

Difference between constructor and method in Java



	Method	Constructor
1	Method can be any user defined name	Constructor must be class name
2	Method should have return type	It should not have any return type (even void)
3	Method should be called explicitly either with object reference or class reference	It will be called automatically whenever object is created
4	Method is not provided by compiler in any case.	The java compiler provides a default constructor if we do not have any constructor.



Difference between constructor and method in Java

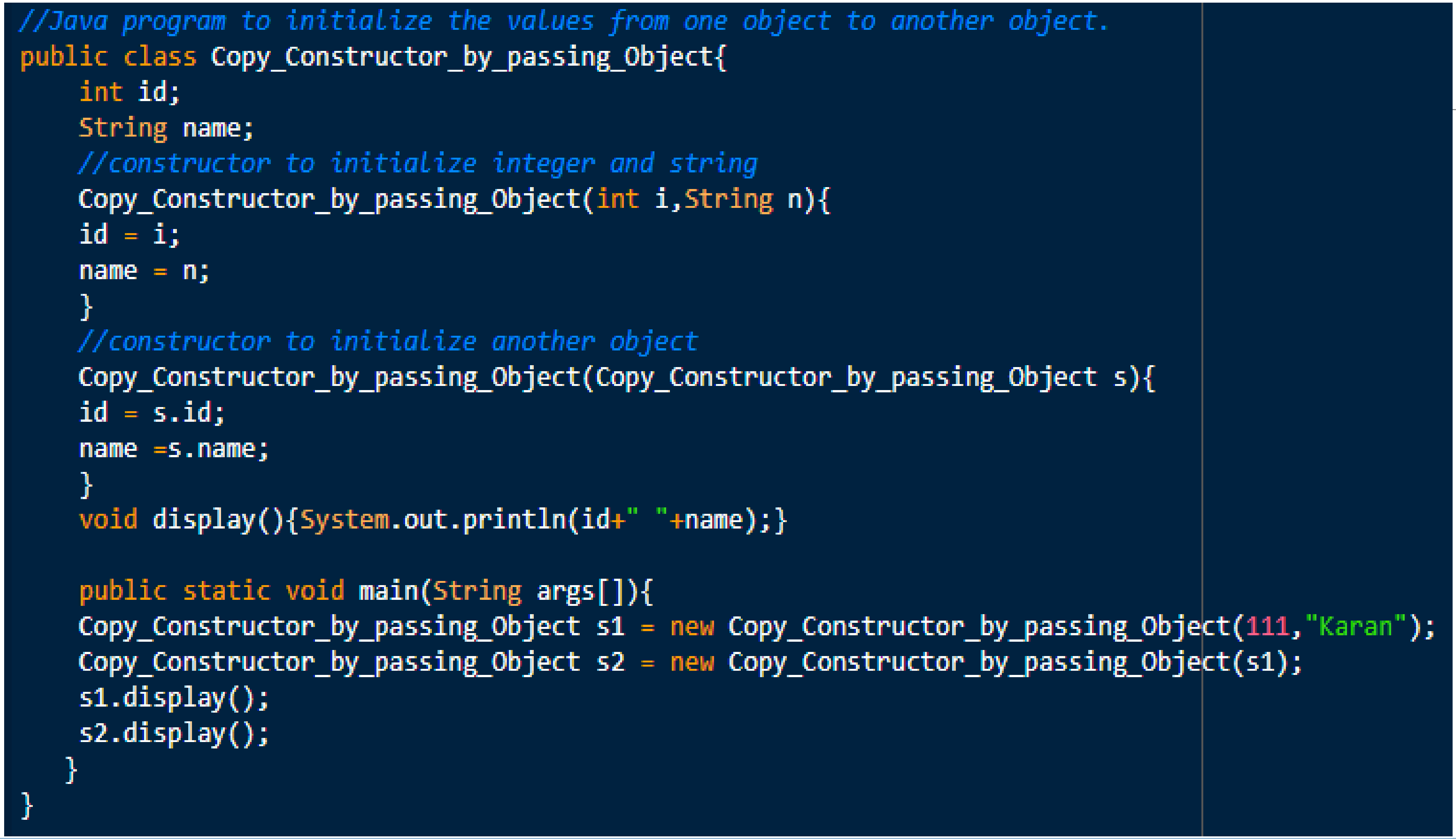
1. A method is used to expose the behavior of an object.
1. A constructor is used to initialize the state of an object.
2. A method must have a return type.
2. A constructor must not have a return type.
3. The method is invoked explicitly.
3. The constructor is invoked implicitly.
4. The method is not provided by the compiler in any case.
4. The Java compiler provides a default constructor if you don't have any constructor in a class.
5. The method name may or may not be same as class name.
5. The constructor name must be same as the class name.

RAMAIAH
Institute of Technology

Java Copy Constructor

- There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.
- There are many ways to copy the values of one object into another in Java. They are:
  - By constructor
  - By assigning the values of one object into another

Java Copy Constructor



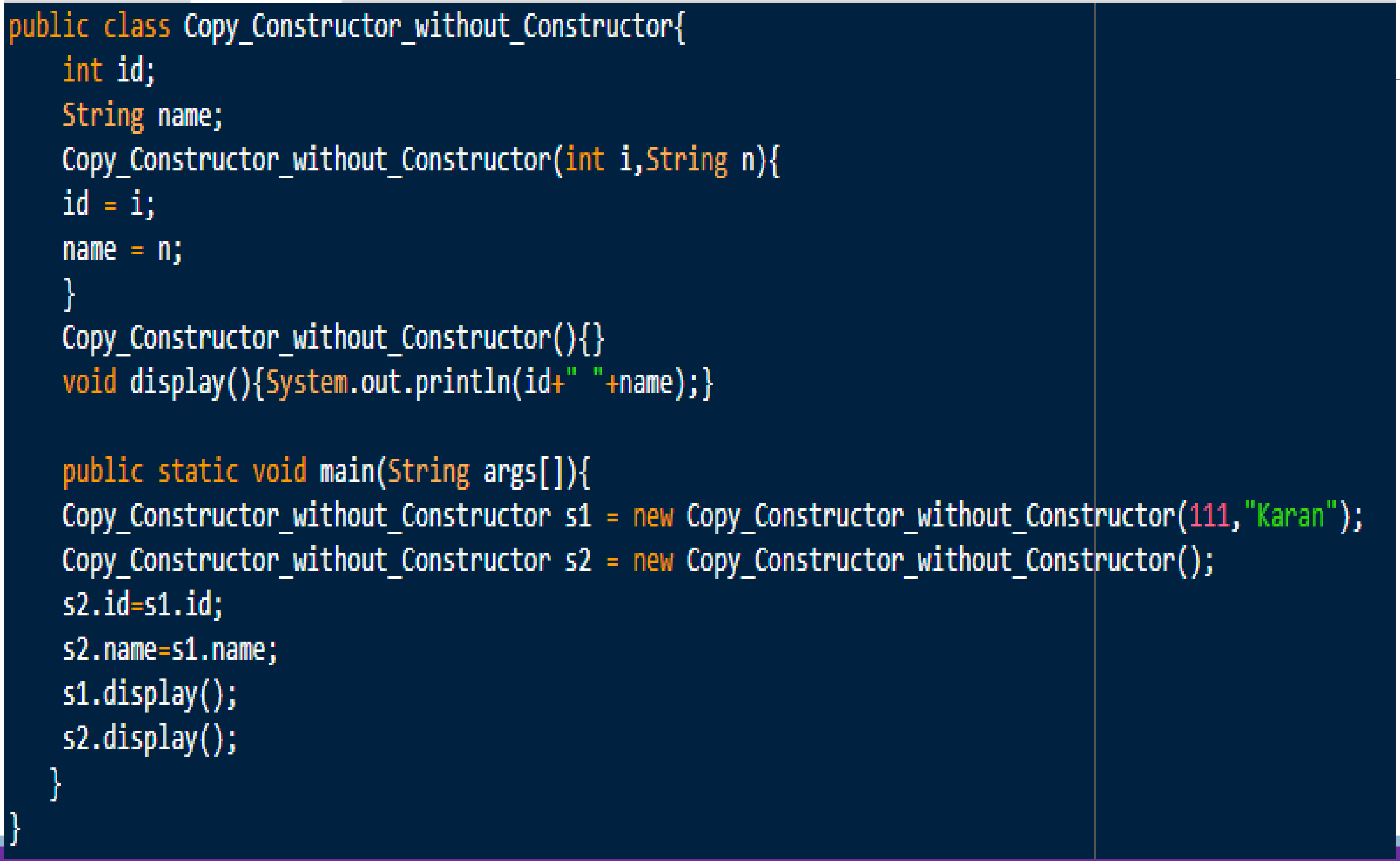
```java
//Java program to initialize the values from one object to another object.
public class Copy_Constructor_by_passing_Object{
    int id;
    String name;
    //constructor to initialize integer and string
    Copy_Constructor_by_passing_Object(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Copy_Constructor_by_passing_Object(Copy_Constructor_by_passing_Object s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Copy_Constructor_by_passing_Object s1 = new Copy_Constructor_by_passing_Object(111,"Karan");
    Copy_Constructor_by_passing_Object s2 = new Copy_Constructor_by_passing_Object(s1);
    s1.display();
    s2.display();
   }
}
```

<u>OUTPUT</u>

```
111 Karan
111 Karan
```

Copy_Constructor_by_passing_Object.java

Java Copy Constructor



```java
public class Copy_Constructor_without_Constructor{
    int id;
    String name;
    Copy_Constructor_without_Constructor(int i,String n){
    id = i;
    name = n;
    }
    Copy_Constructor_without_Constructor(){}
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Copy_Constructor_without_Constructor s1 = new Copy_Constructor_without_Constructor(111,"Karan");
    Copy_Constructor_without_Constructor s2 = new Copy_Constructor_without_Constructor();
    s2.id=s1.id;
    s2.name=s1.name;
    s1.display();
    s2.display();
   }
}
```

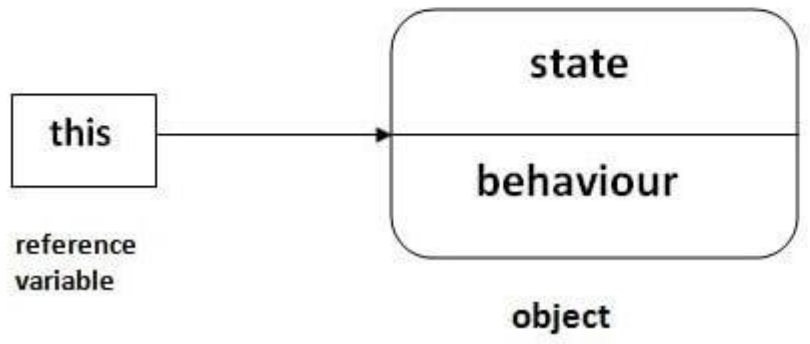
OUTPUT

```
111 Karan
111 Karan
```

Copy_Constructor_without_Constructor.java

this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.



RAMAIAH
Institute of Technology

Usage of this keyword in java

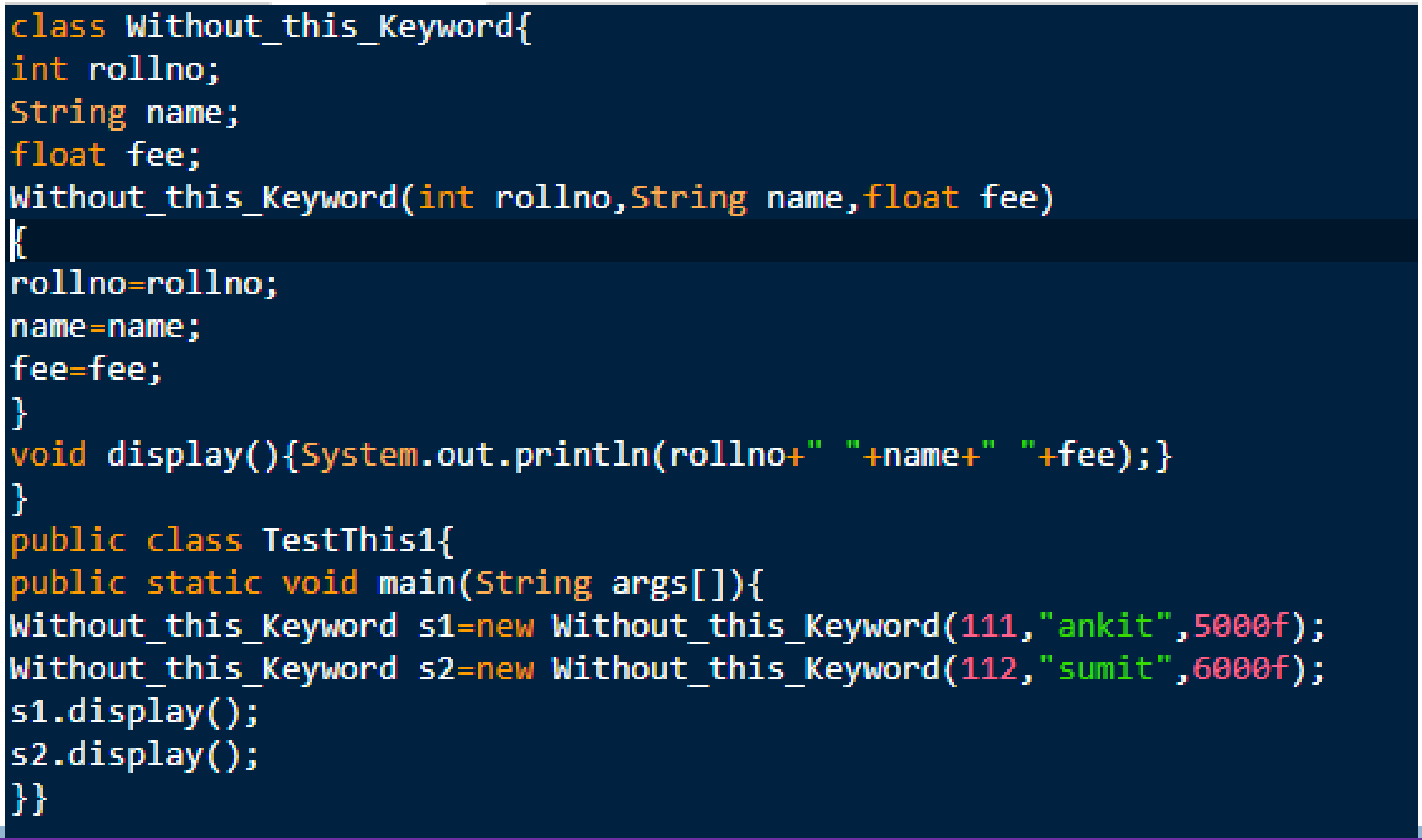
1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

1
this can be used to refer current class instance variable.
2
this can be used to invoke current class method (implicitly)
3
this() can be used to invoke current class constructor.
4
this can be passed as an argument in the method call.
5
this can be passed as argument in the constructor call.
6
this can be used to return the current class instance from the method.

1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword



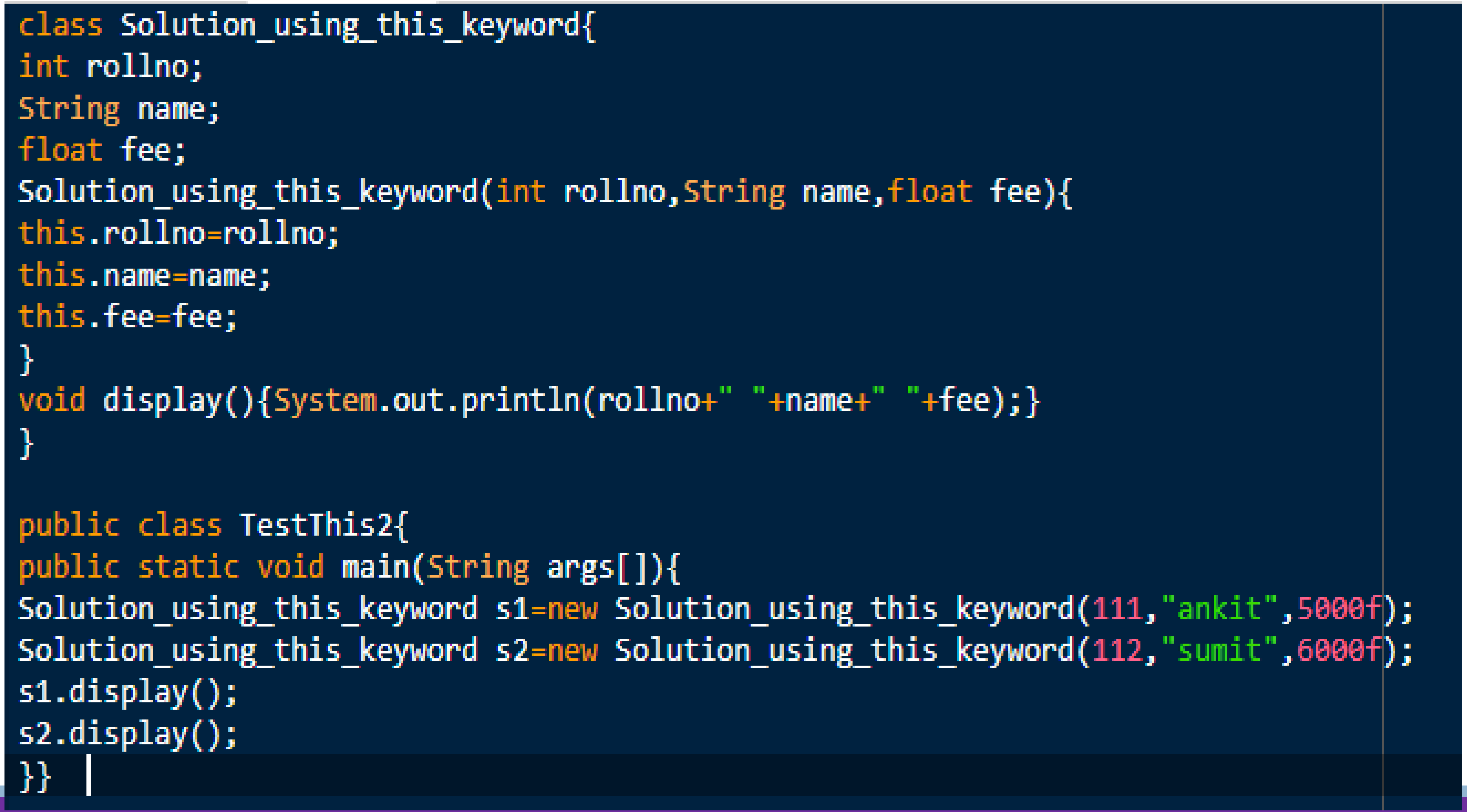
```
class Without_this_Keyword{
int rollno;
String name;
float fee;
Without_this_Keyword(int rollno,String name,float fee)
{
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
public class TestThis1{
public static void main(String args[]){
Without_this_Keyword s1=new Without_this_Keyword(111,"ankit",5000f);
Without_this_Keyword s2=new Without_this_Keyword(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

OUTPUT

```
0 null 0.0
0 null 0.0
```

TestThis1.java

Solution to the problem using this keyword



```java
class Solution_using_this_keyword{
int rollno;
String name;
float fee;
Solution_using_this_keyword(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

public class TestThis2{
public static void main(String args[]){
Solution_using_this_keyword s1=new Solution_using_this_keyword(111,"ankit",5000f);
Solution_using_this_keyword s2=new Solution_using_this_keyword(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

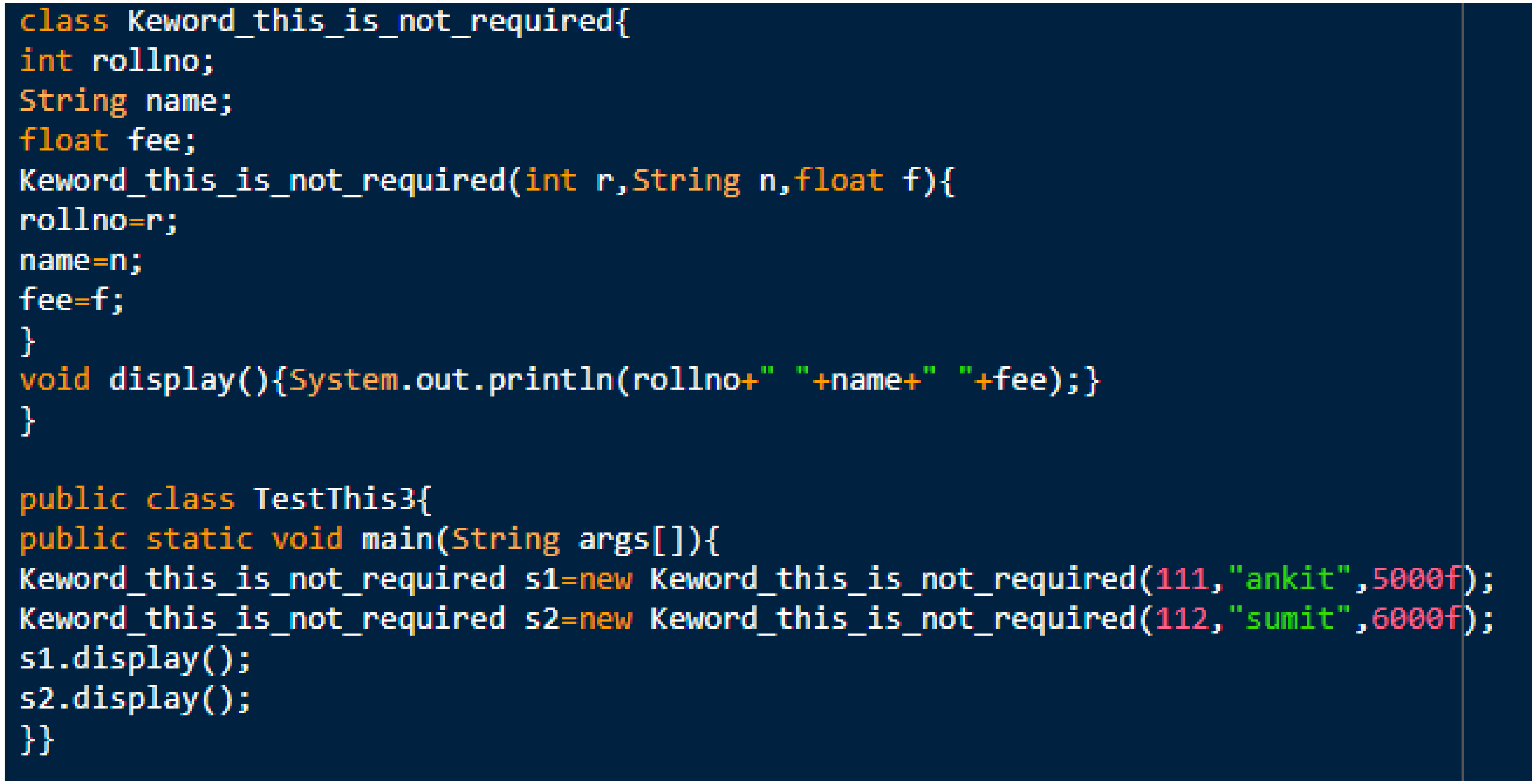
OUTPUT

```
111 ankit 5000.0
112 sumit 6000.0
```

TestThis2.java

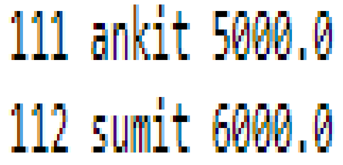
RAMAIAH
Institute of Technology

Program where this keyword is not required



```
class Keword_this_is_not_required{
int rollno;
String name;
float fee;
Keword_this_is_not_required(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

public class TestThis3{
public static void main(String args[]){
Keword_this_is_not_required s1=new Keword_this_is_not_required(111,"ankit",5000f);
Keword_this_is_not_required s2=new Keword_this_is_not_required(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

OUTPUT



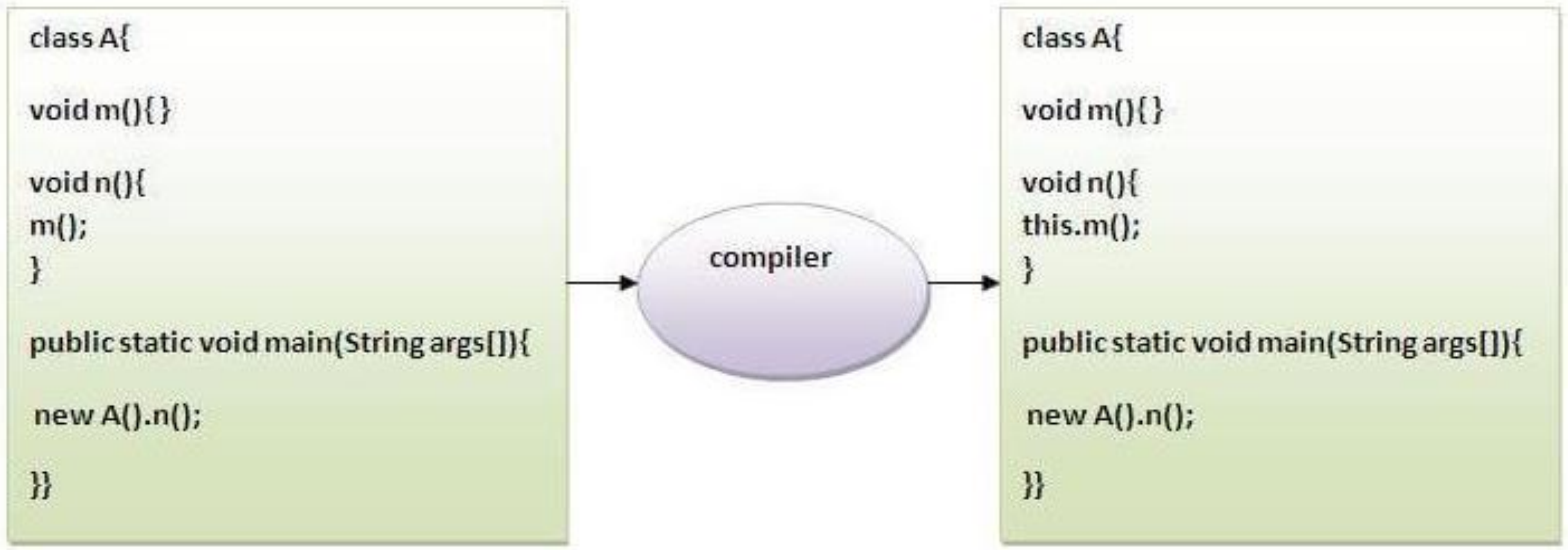
```
111 ankit 5000.0
112 sumit 6000.0
```

TestThis3.java

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword

2) this: to invoke current class method

- You may invoke the method of the current class by using the this keyword.
- If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

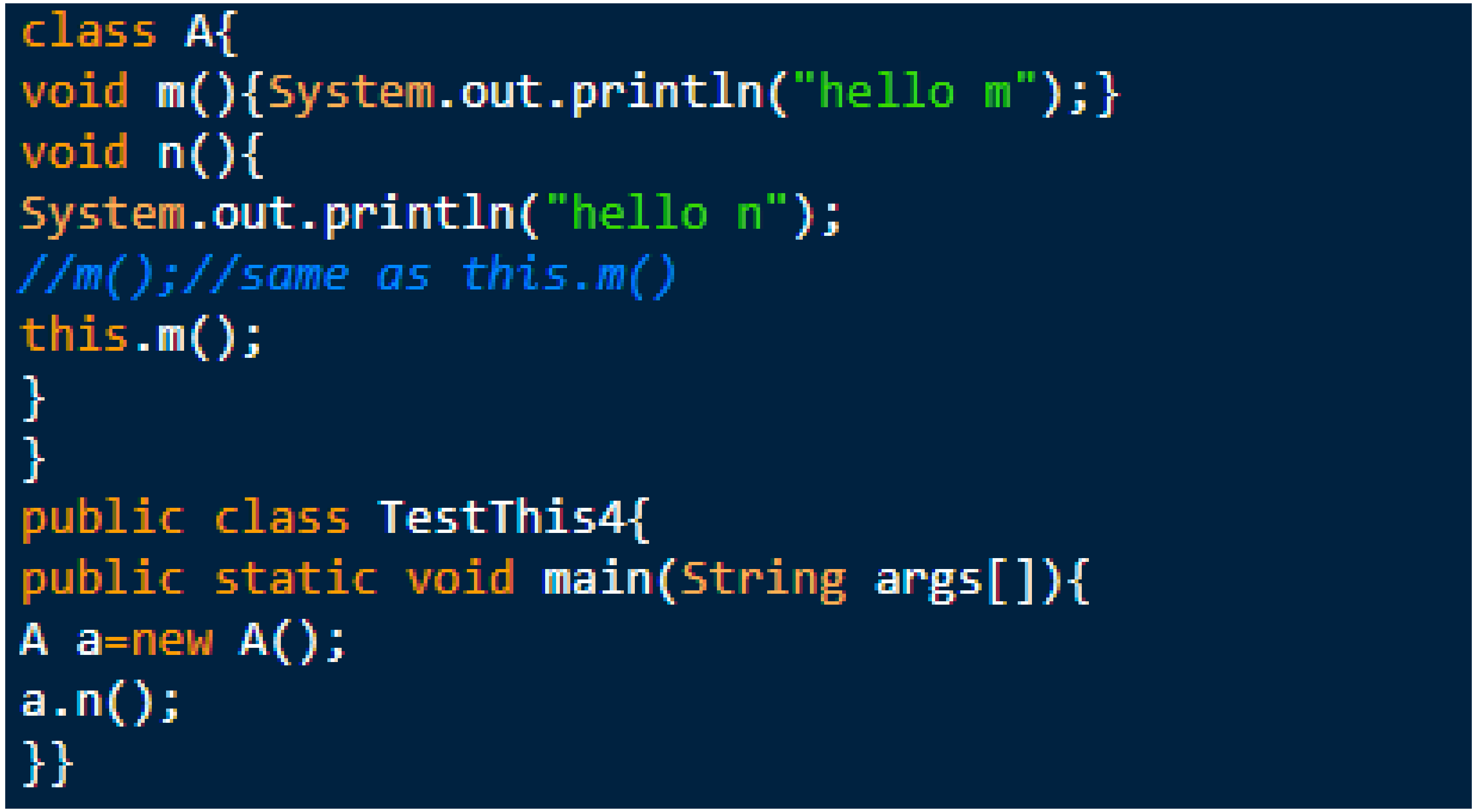


```
class A{

void m(){}

void n(){
m();
}

public static void main(String args[]){

 new A().n();

}}
```

```
class A{

void m(){}

void n(){
this.m();
}

public static void main(String args[]){

 new A().n();

}}
```

RAMAIAH
Institute of Technology

2) this: to invoke current class method



```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
public class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

OUTPUT

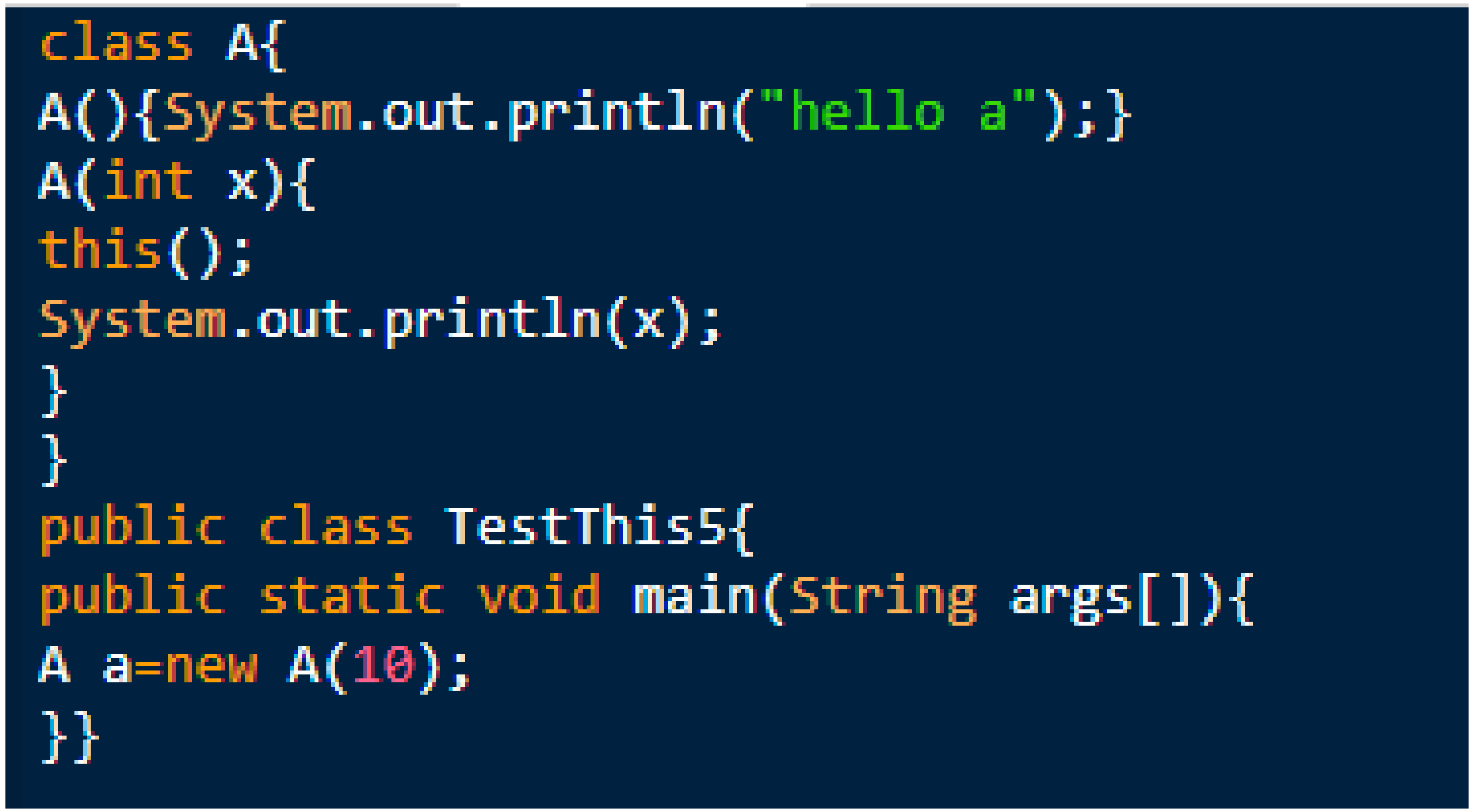
```
hello n
hello m
```

TestThis4.java

3) this() : to invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor.
- It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor



```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
public class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

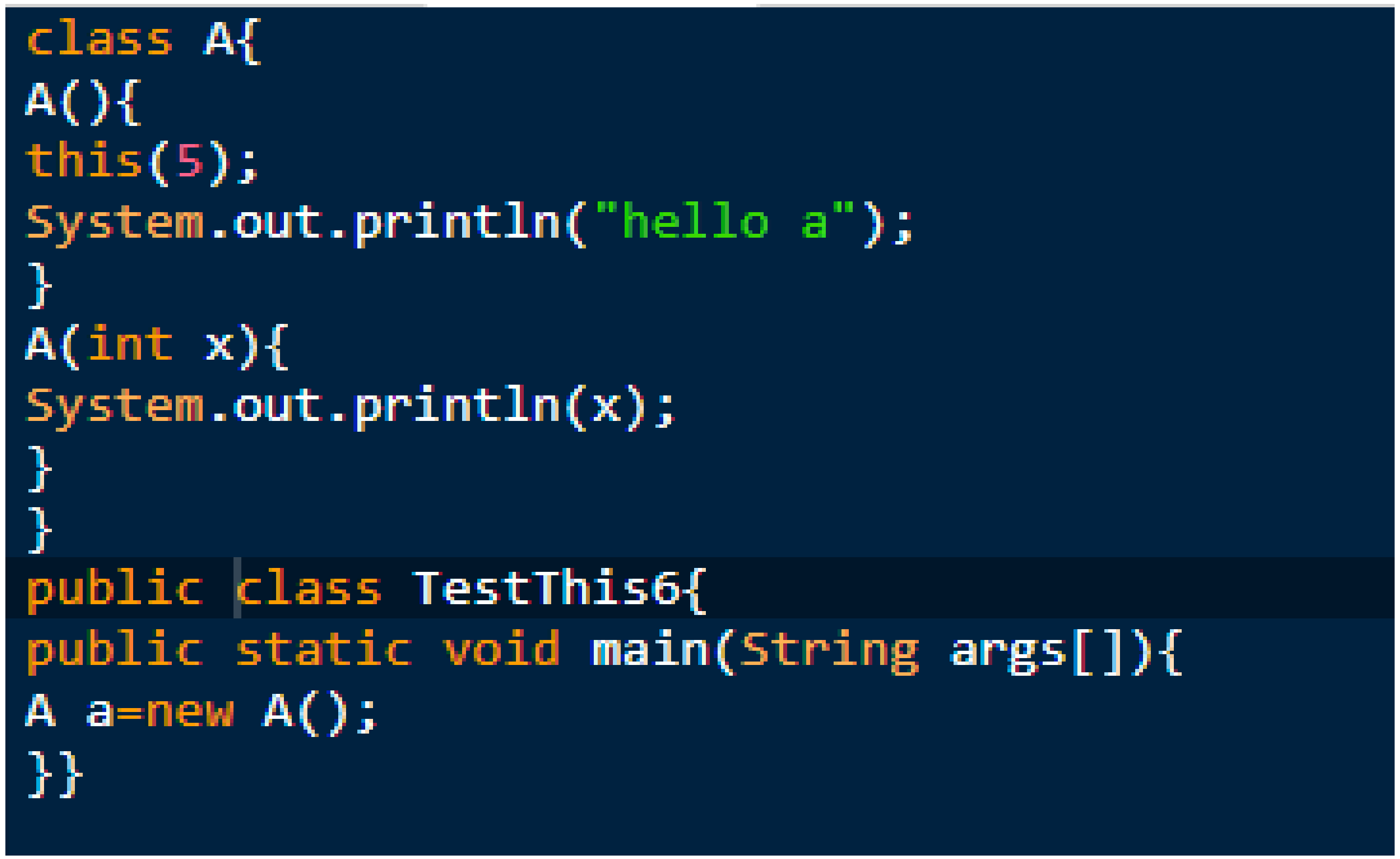
OUTPUT

```
hello a
10
```

TestThis5.java

RAMAIAH
Institute of Technology

Calling parameterized constructor from default constructor



```java
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
public class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

OUTPUT

```
5
hello a
```

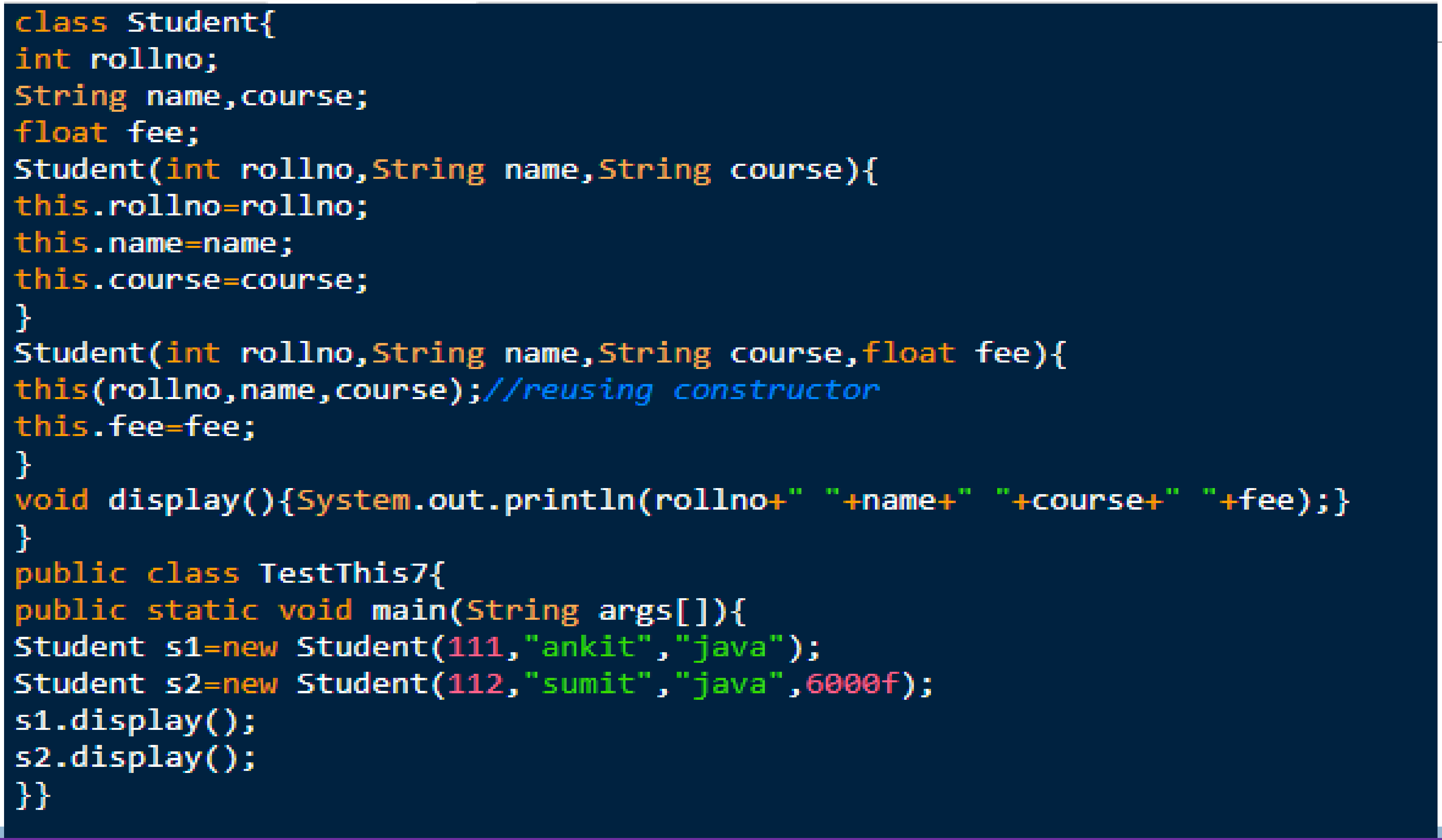
TestThis6.java

Real usage of this() constructor call

- The this() constructor call should be used to reuse the constructor from the constructor.
- It maintains the chain between the constructors i.e. it is used for constructor chaining.

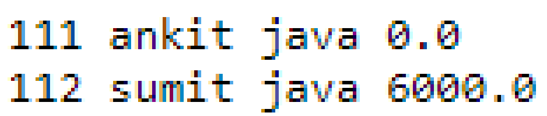
RAMAIAH
Institute of Technology

Real usage of this() constructor call



```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
public class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

OUTPUT



```
111 ankit java 0.0
112 sumit java 6000.0
```

TestThis7.java

RAMAIAH
Institute of Technology

Garbage Collection in Java

- Java garbage collection is the process by which Java programs perform automatic memory management.
- Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short.
- When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program.
- Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

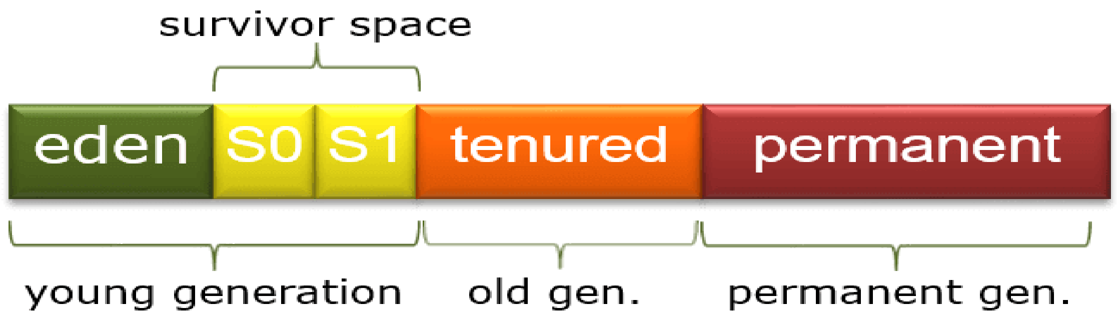
How Java Garbage Collection Works

- Java garbage collection is an automatic process.
- The programmer does not need to explicitly mark objects to be deleted.
- The garbage collection implementation lives in the JVM.
- Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification.
- Although there are many JVMs, Oracle’s HotSpot is by far the most common. It offers a robust and mature set of garbage collection options.

How Java Garbage Collection Works

When a typical Java application is running, it is creating new objects, such as **Strings** and **Files**, but after a certain time, those objects are not used anymore.

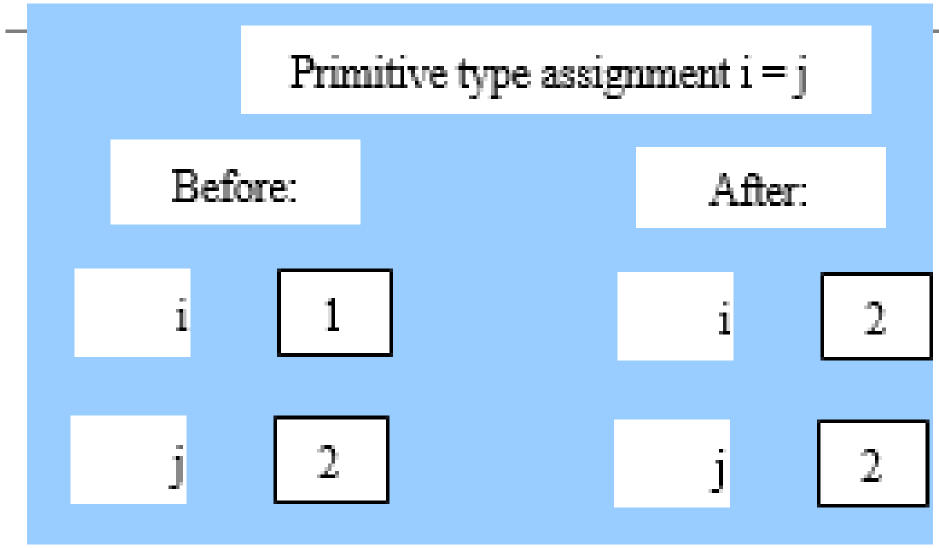
```
for (File f : files) {
String s = f.getName();
}
```

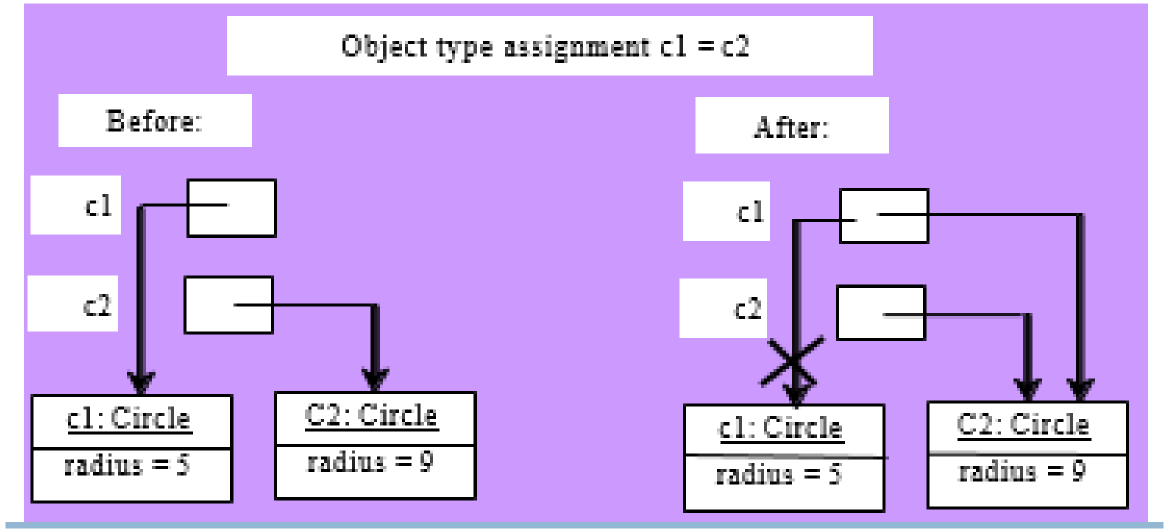


In the above code, the String s is being created on each iteration of the for loop. This means that in every iteration, a little bit of memory is being allocated to make a String object.

The garbage collector will look for objects which aren't being used anymore, and gets rid of them, freeing up the memory so other new objects can use that piece of memory.

Copying Variables of Primitive Data Types and Object Types





As shown in the figure, after the assignment statement c1 = c2, c1 points to the same object referenced by c2. The object previously referenced by c1 is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM.

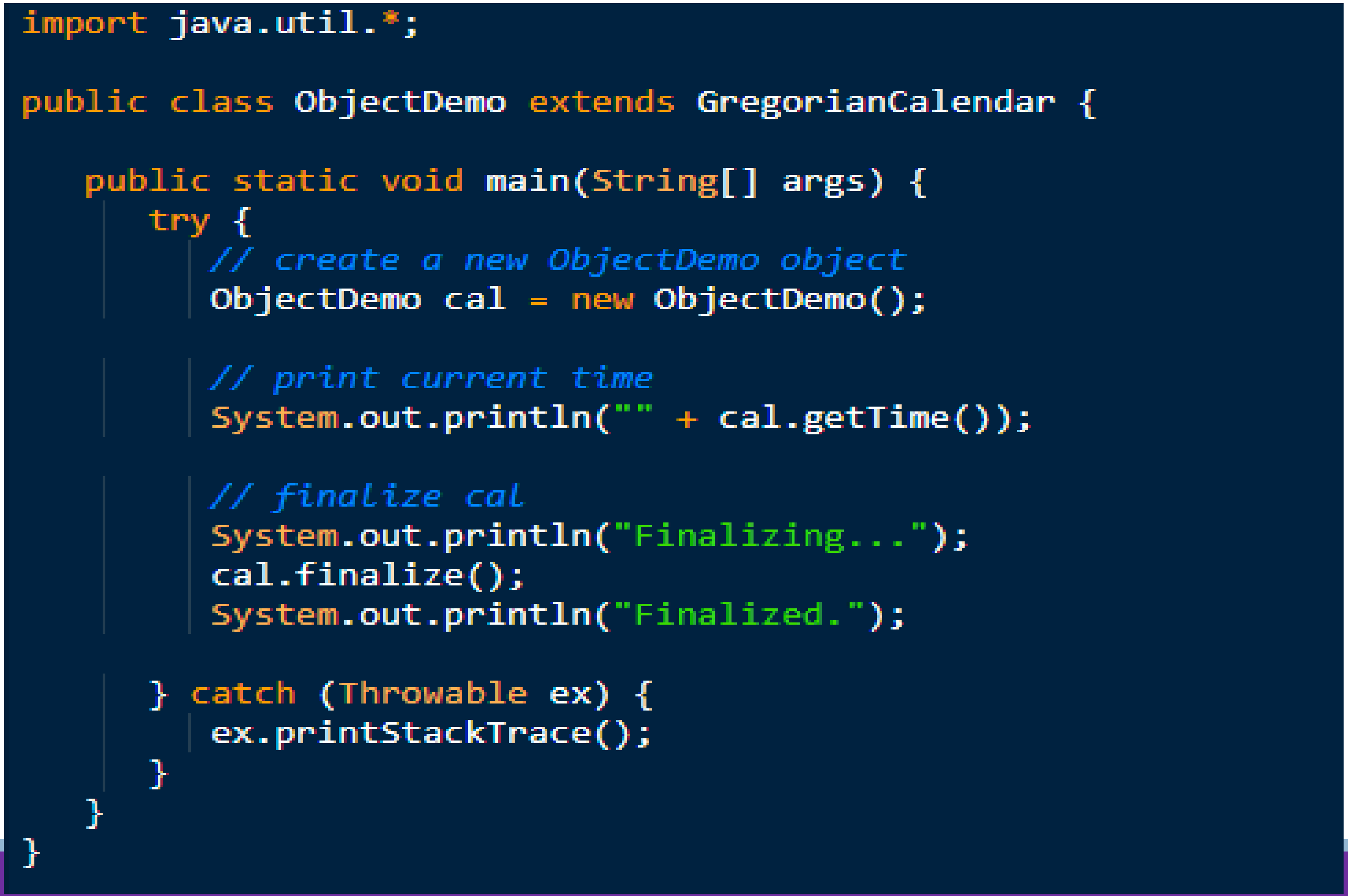
RAMAIAH
Institute of Technology

The finalize() method

The **java.lang.Object.finalize()** is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

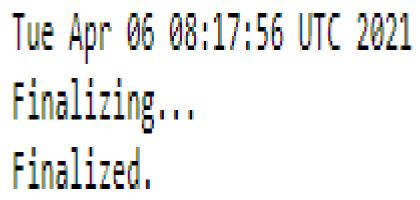
Following is the declaration for **java.lang.Object.finalize()** method
protected void finalize()

The finalize() method



```java
import java.util.*;

public class ObjectDemo extends GregorianCalendar {

   public static void main(String[] args) {
      try {
         // create a new ObjectDemo object
         ObjectDemo cal = new ObjectDemo();

         // print current time
         System.out.println("" + cal.getTime());

         // finalize cal
         System.out.println("Finalizing...");
         cal.finalize();
         System.out.println("Finalized.");

      } catch (Throwable ex) {
         ex.printStackTrace();
      }
   }
}
```

OUTPUT



```
Tue Apr 06 08:17:56 UTC 2021
Finalizing...
Finalized.
```

GregorianCalendar.java

RAMAIAH Institute of Technology

Types of polymorphism in java

There are two types of polymorphism in java

- **Runtime polymorphism(Dynamic polymorphism)**
- **Compile time polymorphism (Static polymorphism)**.

Method overriding is a perfect example of runtime polymorphism.

Method overloading is a perfect example of compile time polymorphism.

- Operator Overloading (Supported in C++, but not in Java)

Method Overloading in java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as **a(int,int)** for two parameters, and **b(int,int,int)** for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

Method Overloading in java

Advantage of method overloading

Method overloading *increases the readability of the program*.

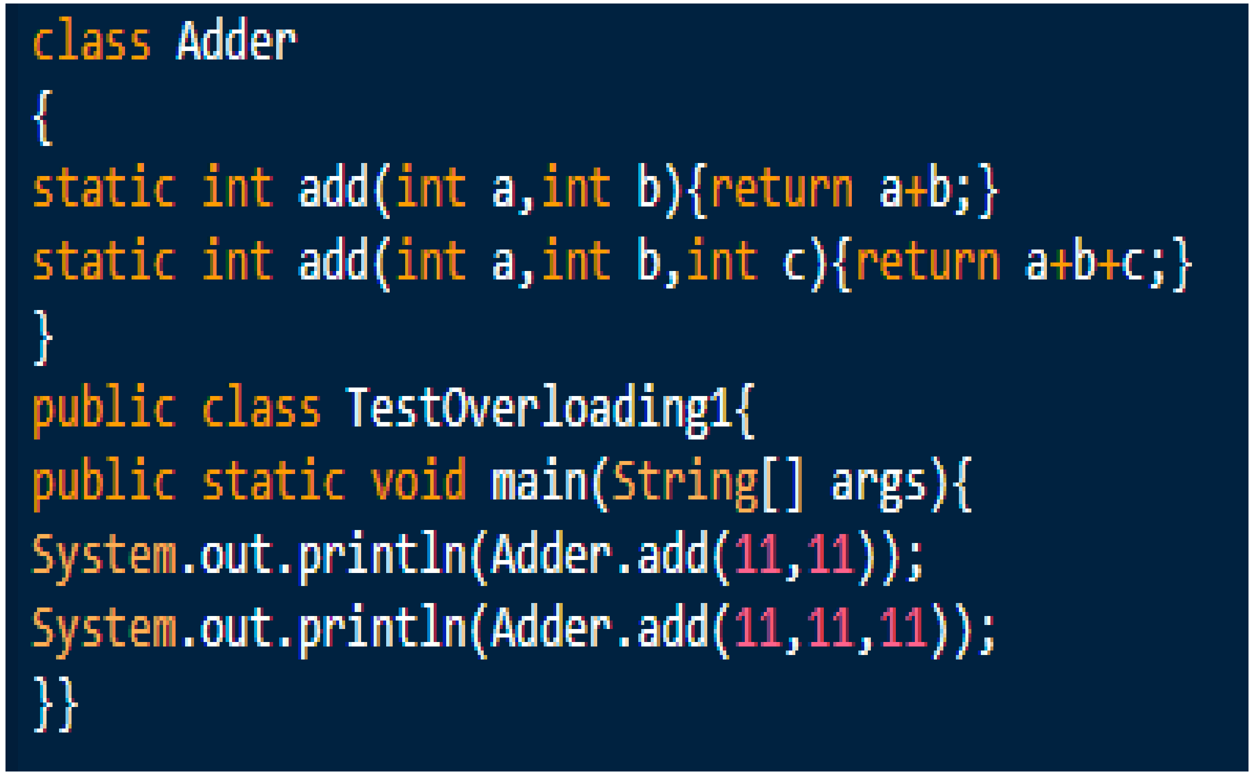
Different ways to overload the method

There are two ways to overload the method in java

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

Note: In Java, Method Overloading is not possible by changing the return type of the method only.

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

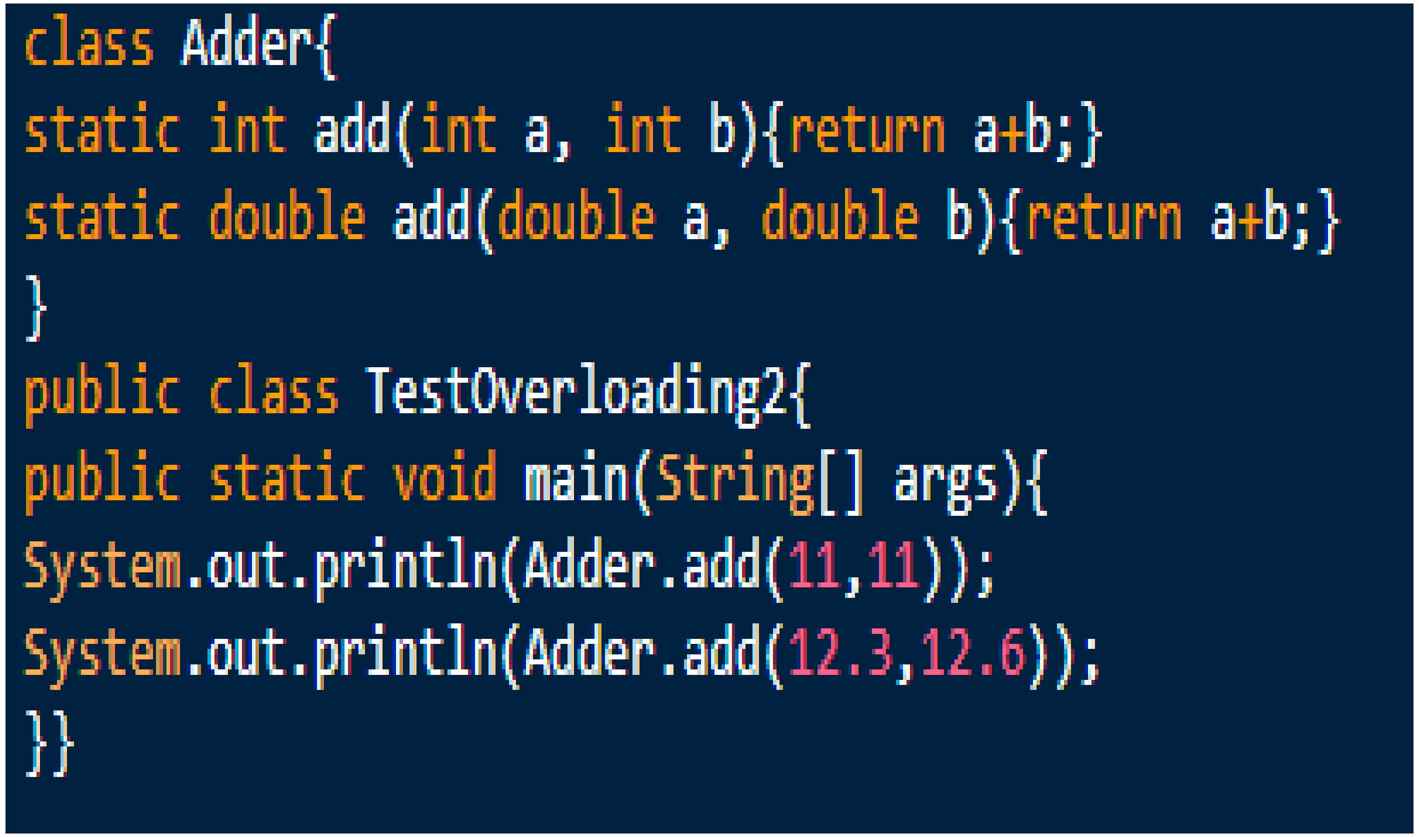
<u>OUTPUT</u>

```
22
33
```

TestOverloading1.java

RAMAIAH
Institute of Technology

Method Overloading: changing data type of arguments



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

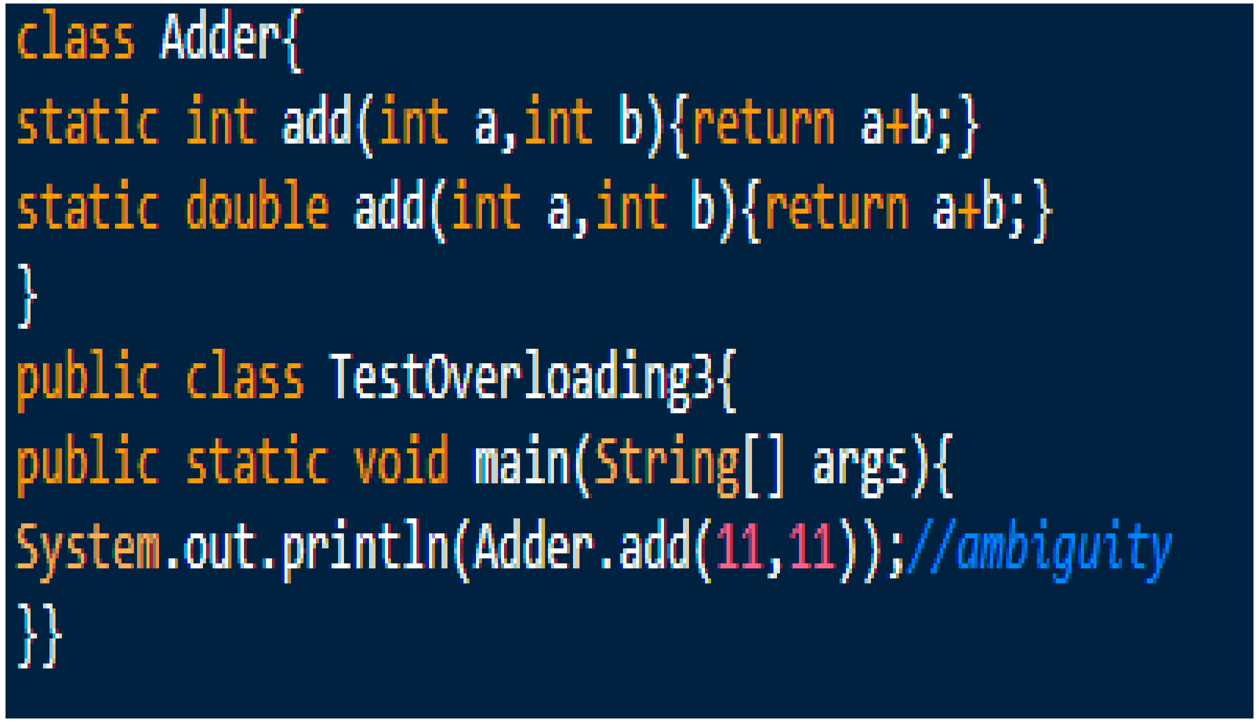
<u>OUTPUT</u>

```
22
24.9
```

TestOverloading2.java

RAMAIAH
Institute of Technology

Why Method Overloading is not possible by changing the return type of method only?



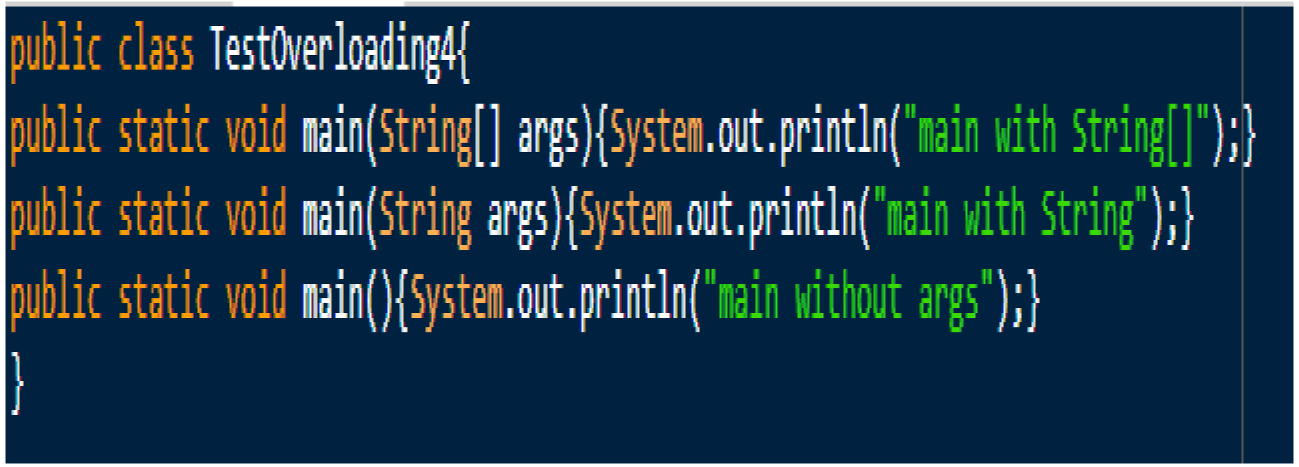
```java
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
public class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

TestOverloading3.java

OUTPUT

```
TestOverloading3.java:3: error: method add(int,int) is already defined in class Adder
static double add(int a,int b){return a+b;}
                 ^
1 error
```

Can we overload java main() method?



```
public class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

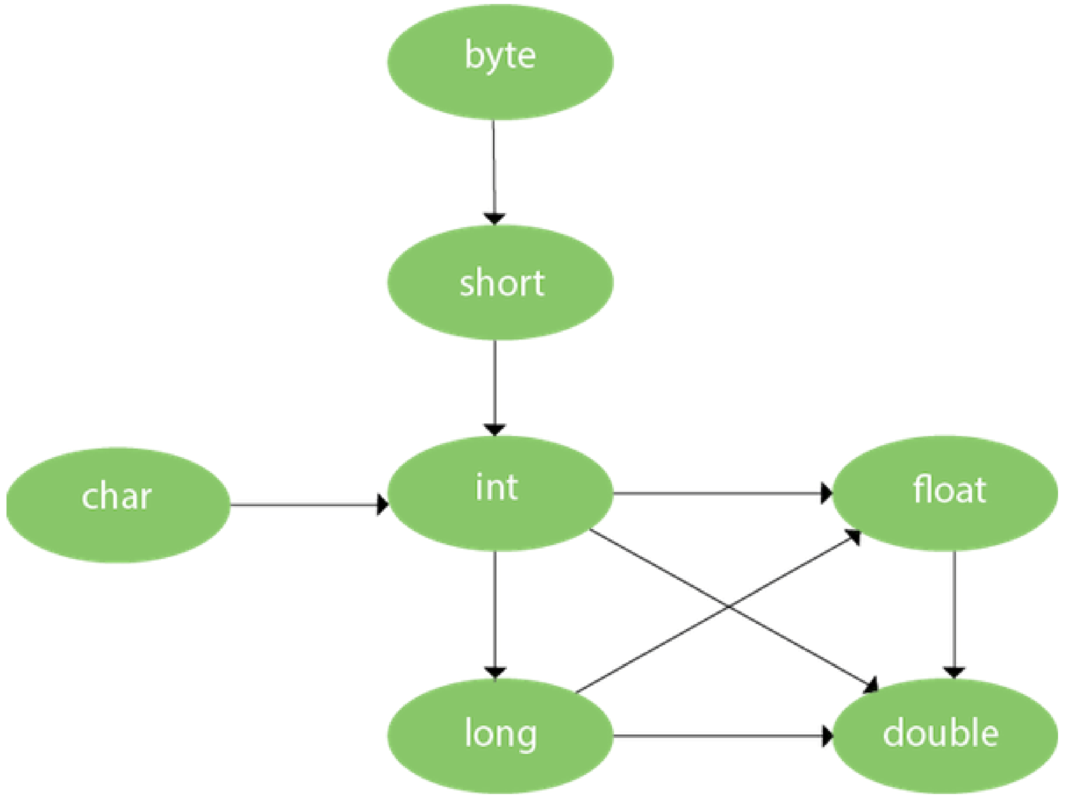
TestOverloading4.java

OUTPUT

```
main with String[]
```

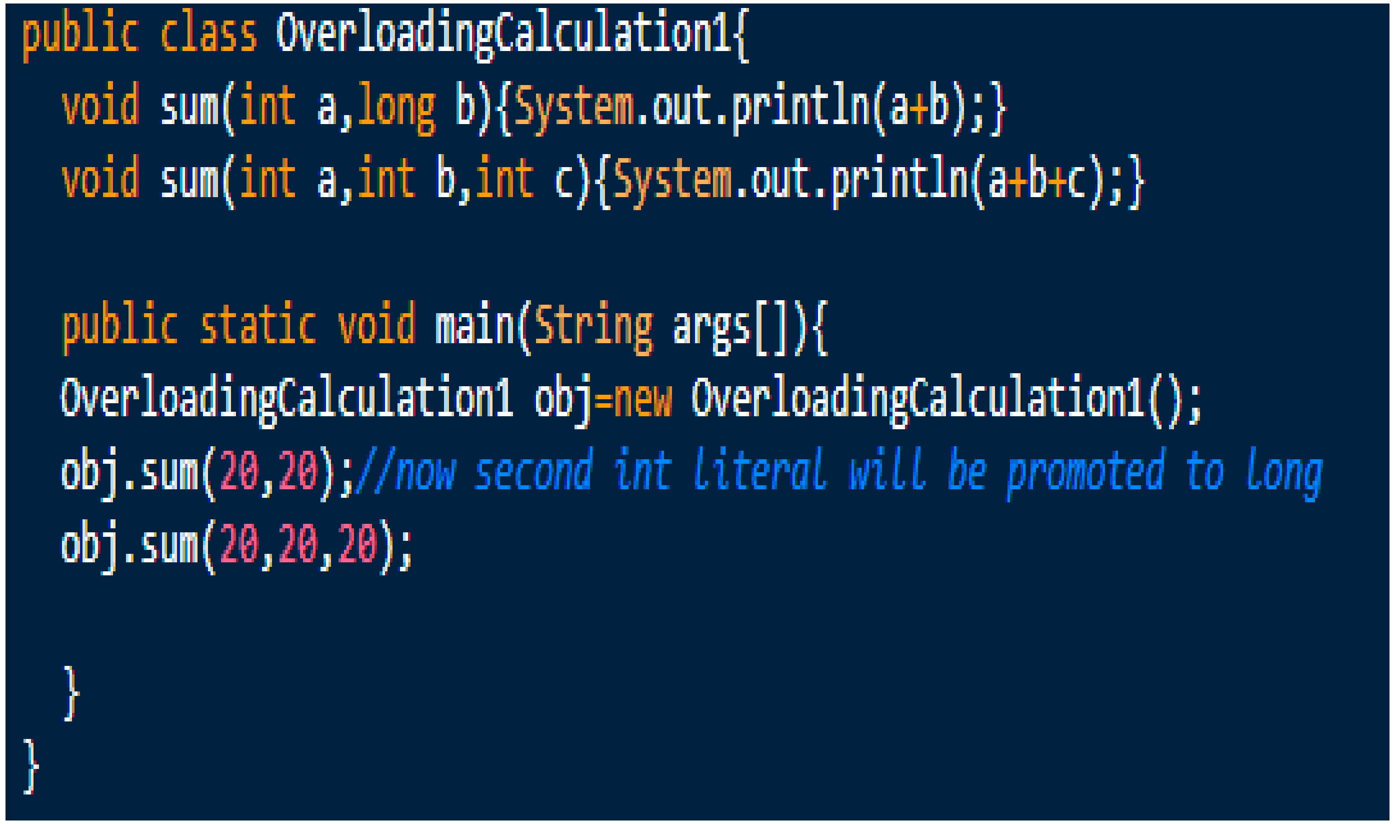
Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found.



RAMAIAH
Institute of Technology

Method Overloading and Type Promotion



```java
public class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}
```

OUTPUT

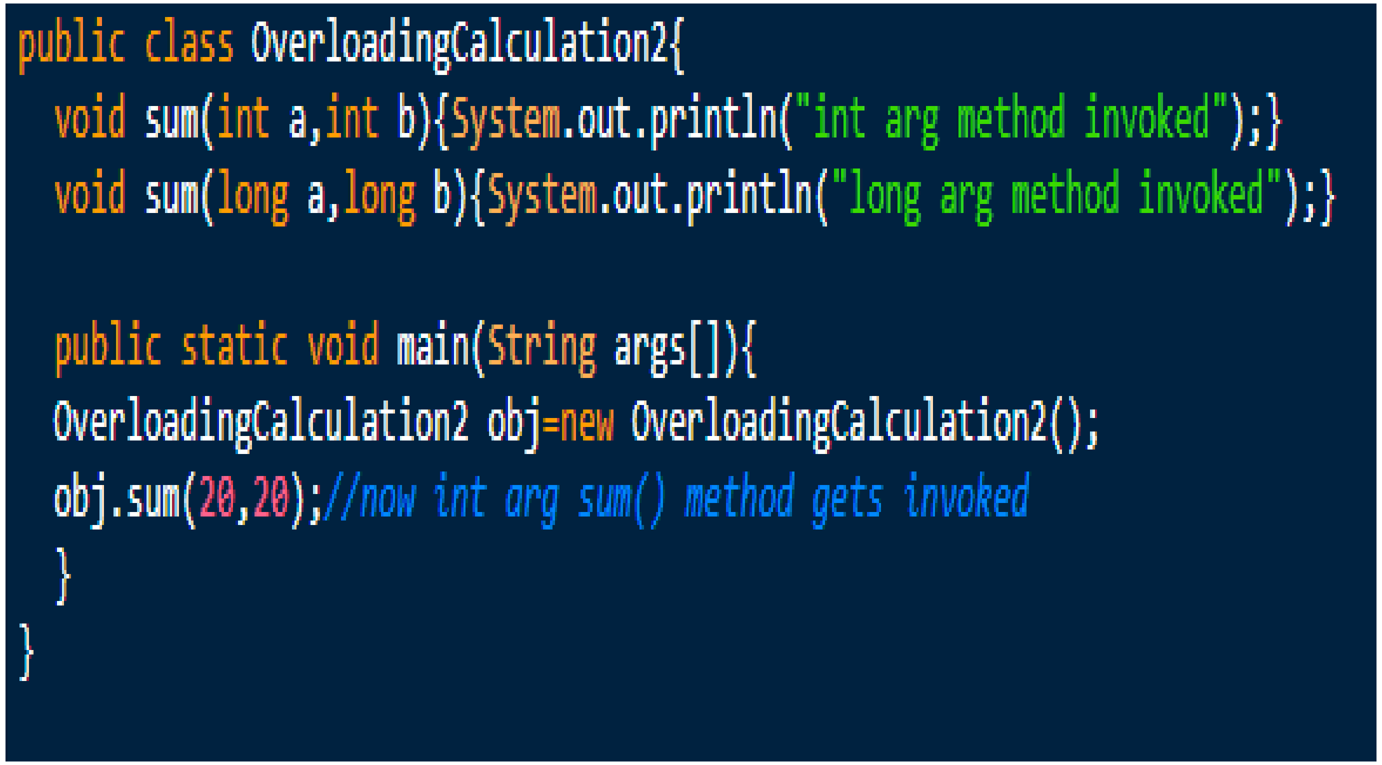
```
40
60
```

Method Overloading with TypePromotion

OverloadingCalculation1.java

RAMAIAH
Institute of Technology

Method Overloading and Type Promotion



```
public class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

OUTPUT

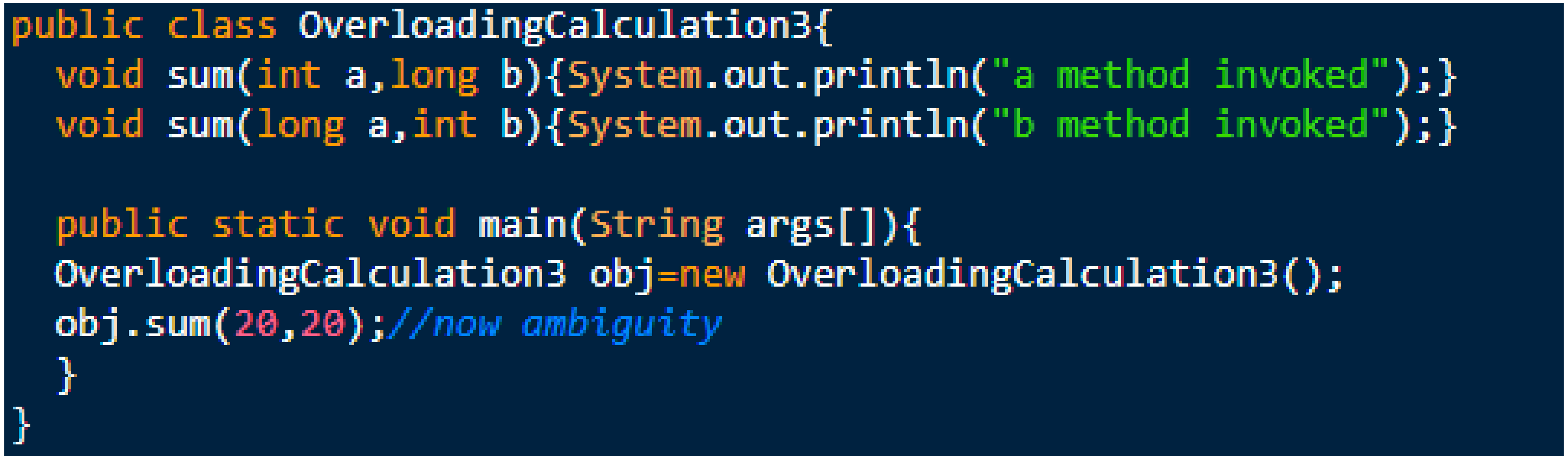
```
int arg method invoked
```

Method Overloading with Type Promotion if matching found

OverloadingCalculation2.java

RAMAIAH
Institute of Technology

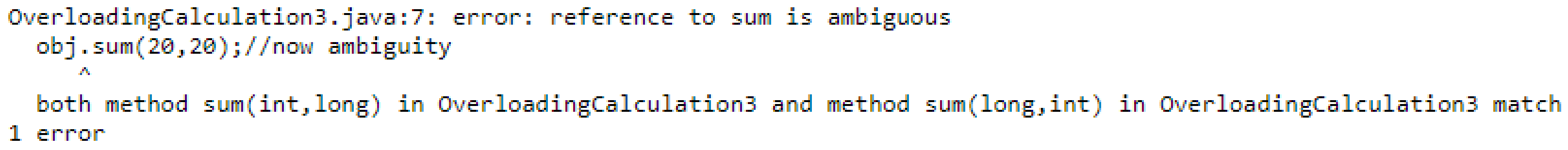
Method Overloading and Type Promotion



```
public class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}
```

OverloadingCalculation3.java

OUTPUT



```
OverloadingCalculation3.java:7: error: reference to sum is ambiguous
  obj.sum(20,20);//now ambiguity
     ^
  both method sum(int,long) in OverloadingCalculation3 and method sum(long,int) in OverloadingCalculation3 match
1 error
```

One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

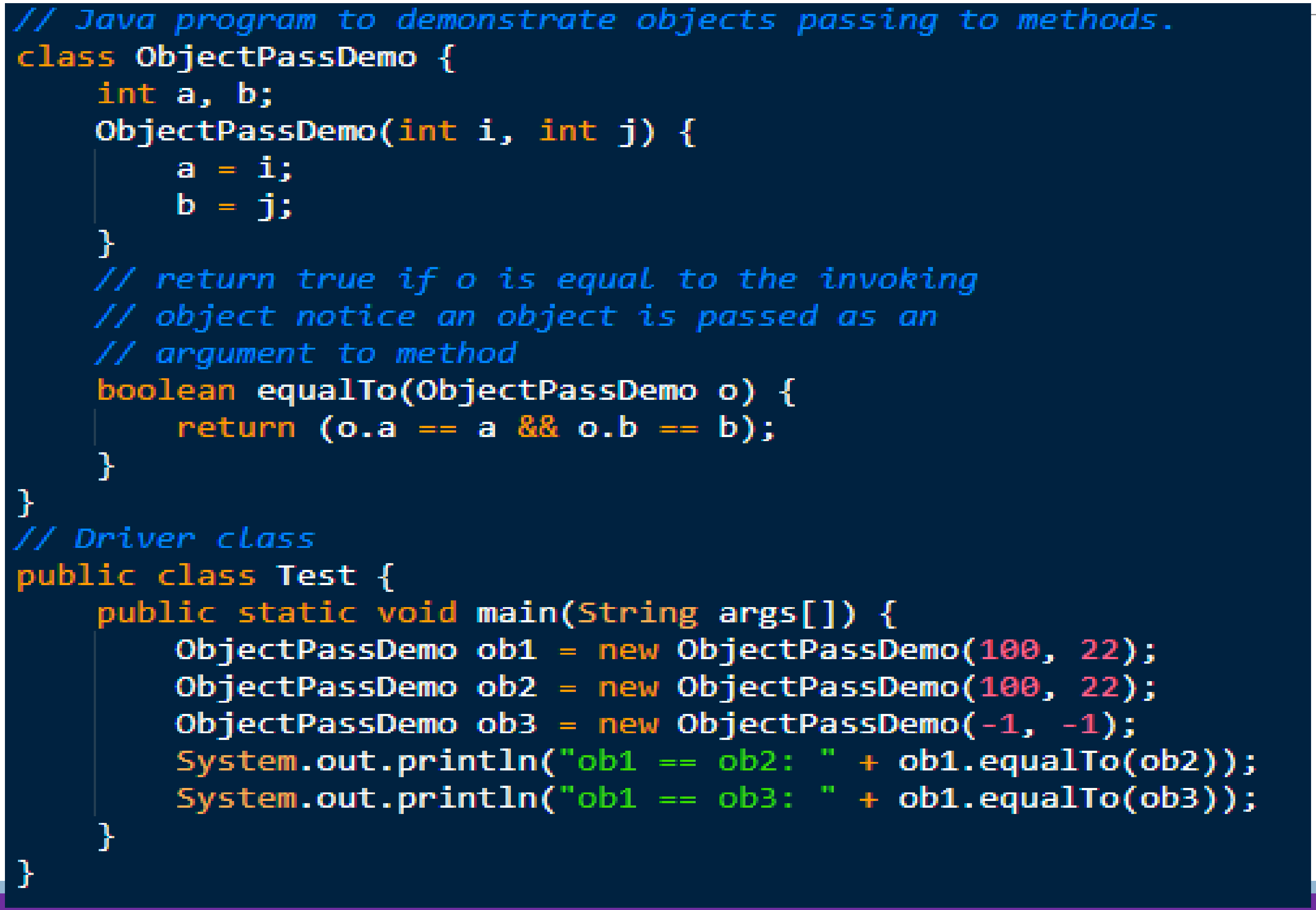
RAMAIAH
Institute of Technology

Passing and Returning Objects in Java

- Java is strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed.
- When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.

RAMAIAH
Institute of Technology

Objects passing to methods



```java
// Java program to demonstrate objects passing to methods.
class ObjectPassDemo {
    int a, b;
    ObjectPassDemo(int i, int j) {
        a = i;
        b = j;
    }
    // return true if o is equal to the invoking
    // object notice an object is passed as an
    // argument to method
    boolean equalTo(ObjectPassDemo o) {
        return (o.a == a && o.b == b);
    }
}
// Driver class
public class Test {
    public static void main(String args[]) {
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

OUTPUT

```
ob1 == ob2: true
ob1 == ob3: false
```

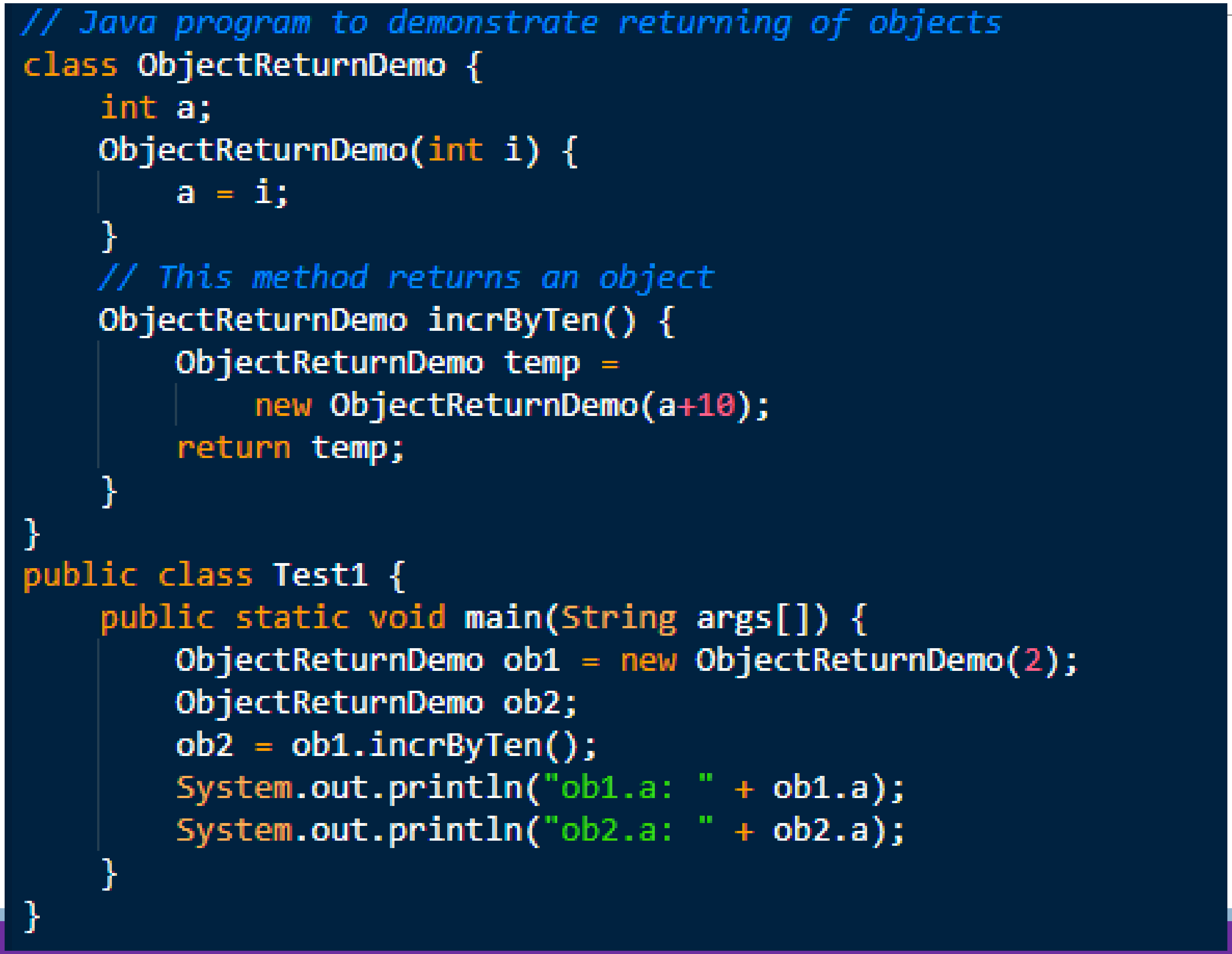
Test.java

Returning Objects

In java, a method can return any type of data, including objects. For example, in the following program, the **incrByTen( )** method returns an object in which the value of a (an integer variable) is ten greater than it is in the invoking object.

RAMAIAH
Institute of Technology

Returning Objects



```java
// Java program to demonstrate returning of objects
class ObjectReturnDemo {
    int a;
    ObjectReturnDemo(int i) {
        a = i;
    }
    // This method returns an object
    ObjectReturnDemo incrByTen() {
        ObjectReturnDemo temp =
            new ObjectReturnDemo(a+10);
        return temp;
    }
}
public class Test1 {
    public static void main(String args[]) {
        ObjectReturnDemo ob1 = new ObjectReturnDemo(2);
        ObjectReturnDemo ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
    }
}
```

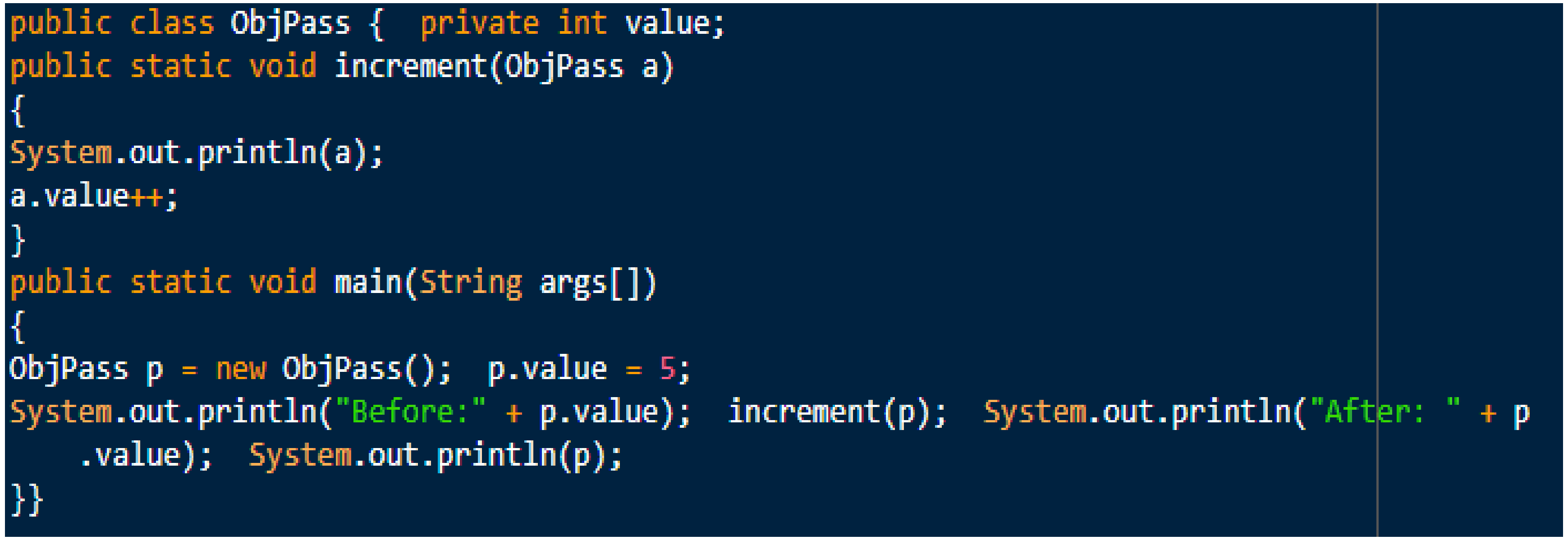
OUTPUT

```
ob1 == ob2: true
ob1 == ob3: false
```

Test1.java

RAMAIAH
Institute of Technology

Example Passing Object



```
public class ObjPass {  private int value;
public static void increment(ObjPass a)
{
System.out.println(a);
a.value++;
}
public static void main(String args[])
{
ObjPass p = new ObjPass();  p.value = 5;
System.out.println("Before:" + p.value);  increment(p);  System.out.println("After: " + p
    .value);  System.out.println(p);
}}
```

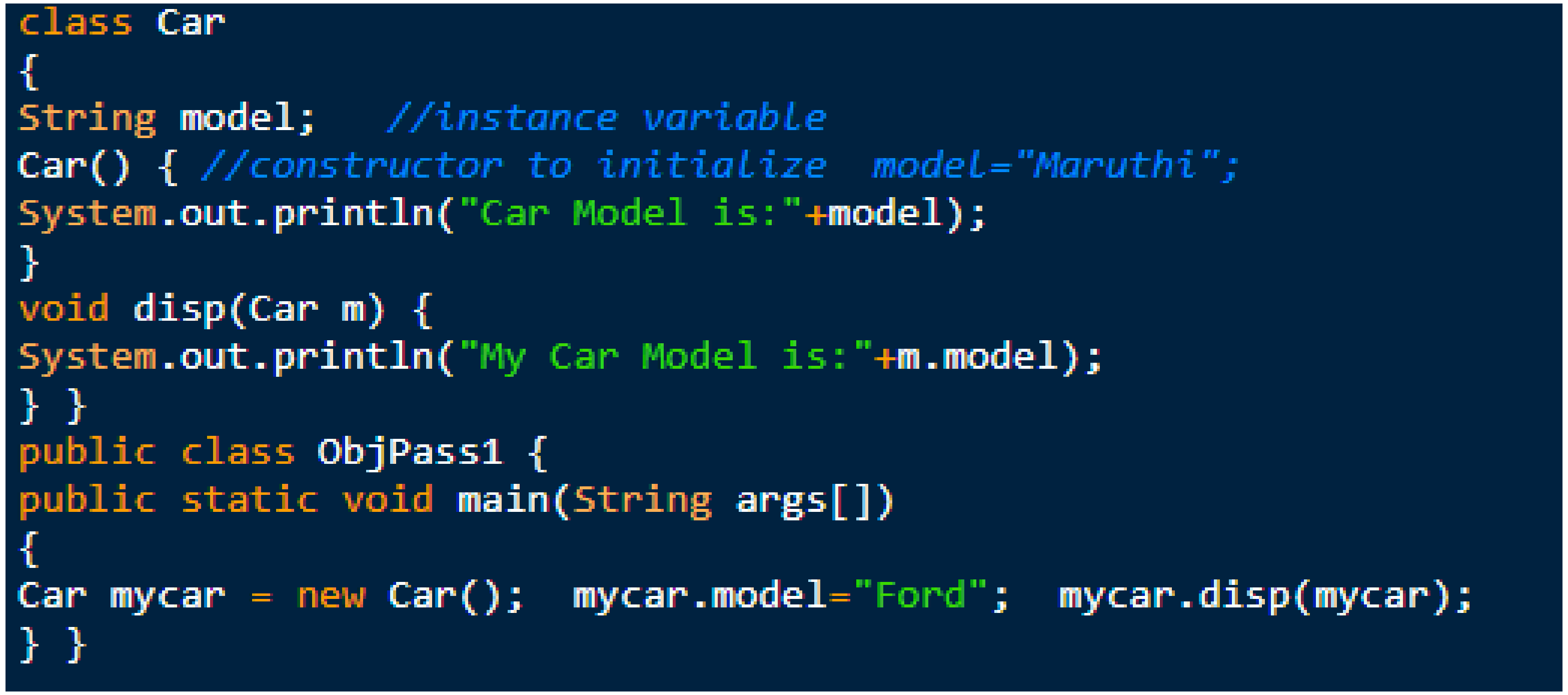
OUTPUT

```
Before:5
ObjPass@6d06d69c
After: 6
ObjPass@6d06d69c
```

Here we pass exactly is a handle of an object, and in the called method a new handle created and pointed to the same object. From the example above you can see that both **p** and **a** refer to the same object.

ObjPassjava

Example Passing Object



```
class Car
{
String model;   //instance variable
Car() { //constructor to initialize  model="Maruthi";
System.out.println("Car Model is:"+model);
}
void disp(Car m) {
System.out.println("My Car Model is:"+m.model);
} }
public class ObjPass1 {
public static void main(String args[])
{
Car mycar = new Car();  mycar.model="Ford";  mycar.disp(mycar);
} }
```

OUTPUT

```
Car Model is:null
My Car Model is:Ford
```

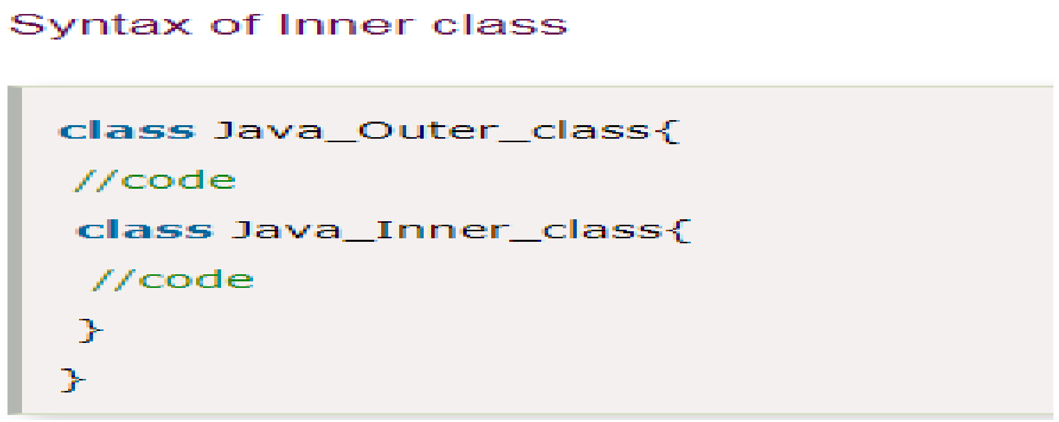
ObjPass1.java

We can access the instance variables of the object passed inside the called method. It is good practice to initialize instance variables of an object before passing object as parameter to method otherwise it will take default initial values.

RAMAIAH
Institute of Technology

Java Inner Classes

- **Java inner class** or nested class is a class which is declared inside the class or interface.
- We use inner classes 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 of Inner class

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

Advantages of Inner Classes

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.

Difference between nested class and inner class in Java

- Inner class is a part of nested class.
- Non-static nested classes are known as inner classes.

RAMAIAH
Institute of Technology

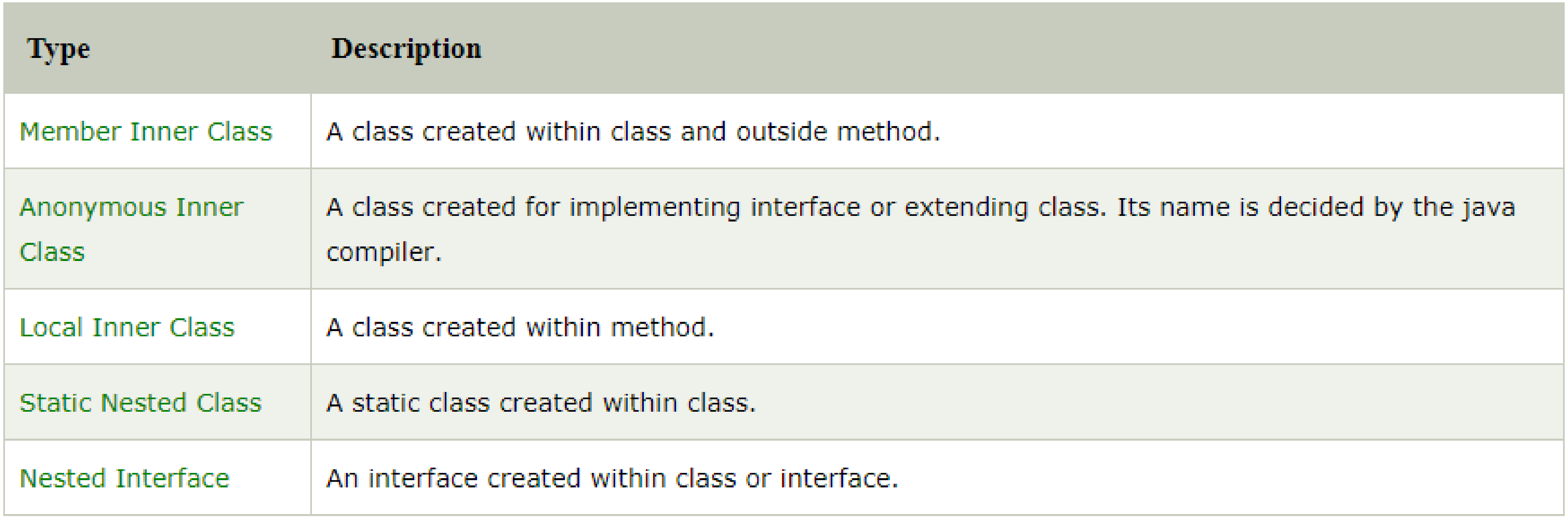
Types of Nested Classes

There are two types of nested classes non-static and static nested classes.
The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

RAMAIAH Institute of Technology

Types of Nested Classes



Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Member Inner class

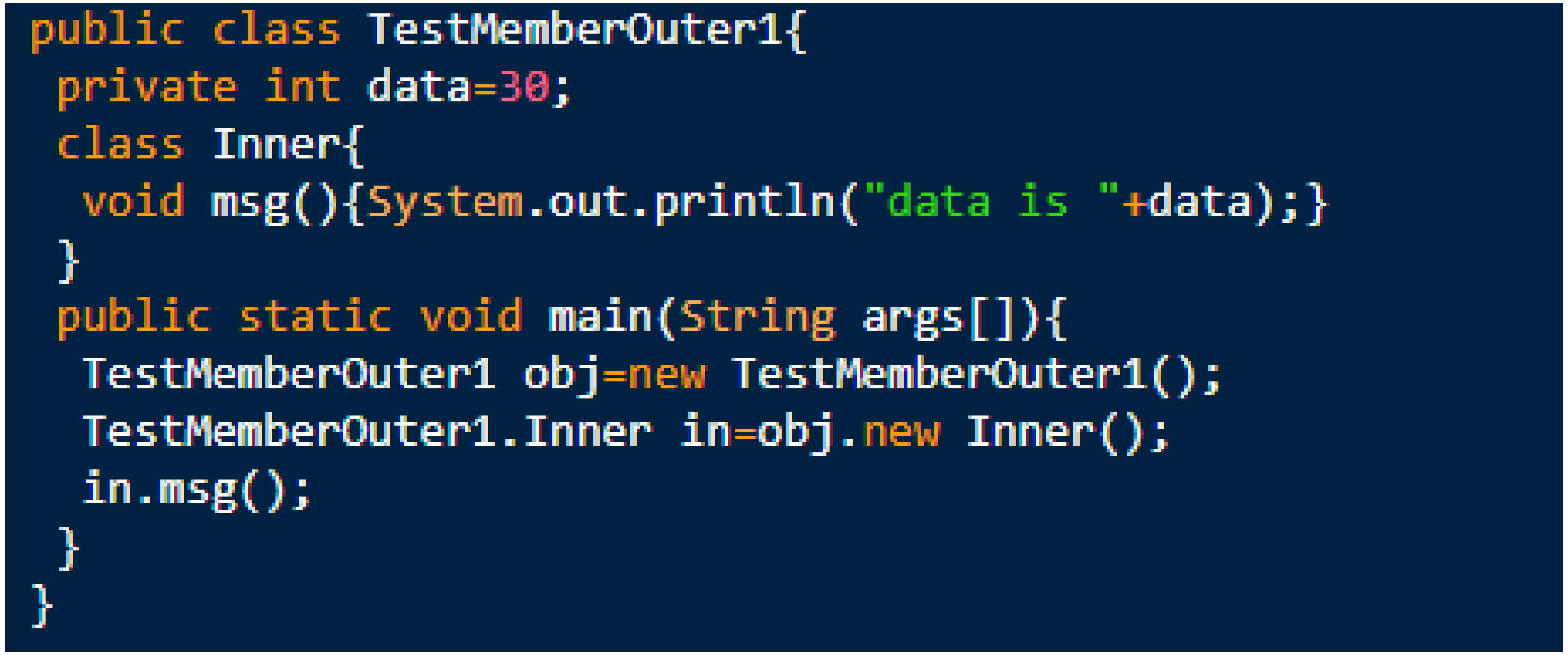
A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer
{
        //code
        class Inner
        {
                //code
        }
}
```

RAMAIAH
Institute of Technology

Member Inner class



```
public class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

OUTPUT

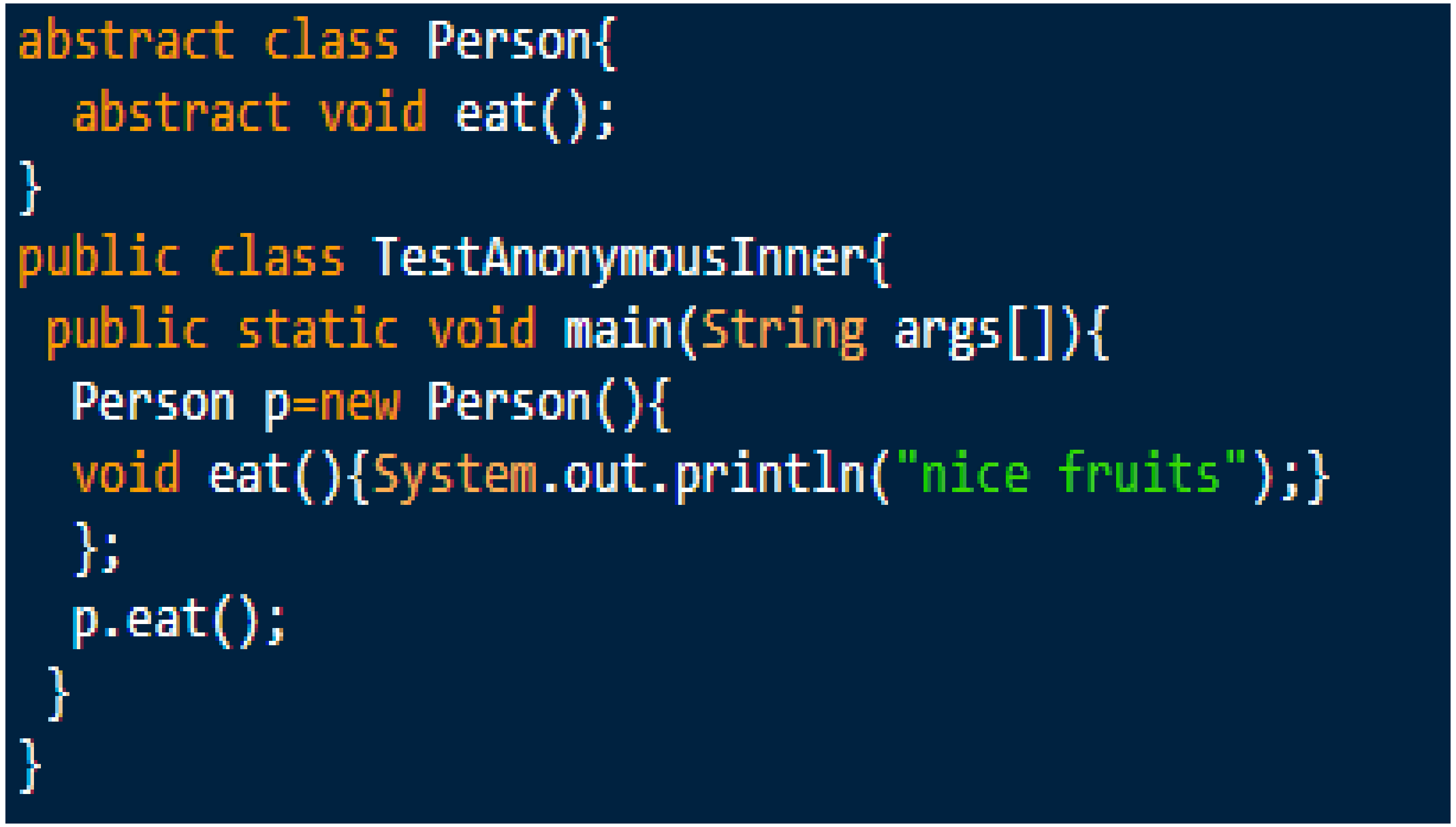
```
data is 30
```

TestMemberOuter1.java

Java Anonymous inner class

- A class that have no name is known as anonymous inner class in java.
- It should be used if you have to override method of class or interface.
- Java Anonymous inner class can be created by two ways:
    1. Class (may be abstract or concrete).
    2. Interface

Java Anonymous inner class



```
abstract class Person{
  abstract void eat();
}
public class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

OUTPUT

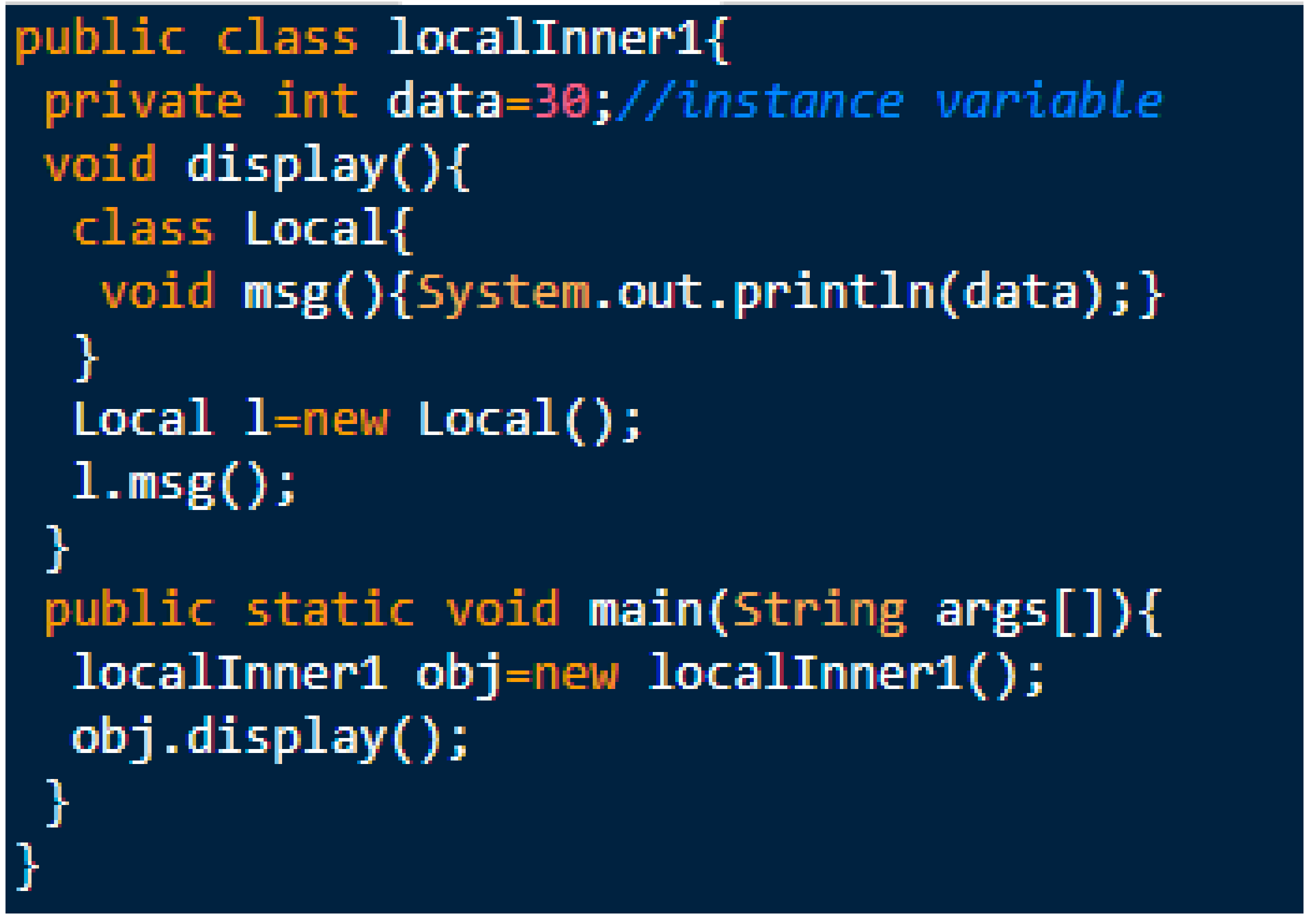
```
nice fruits
```

TestAnonymousInner.java

Java Local inner class

- A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class



```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

OUTPUT

```
30
```

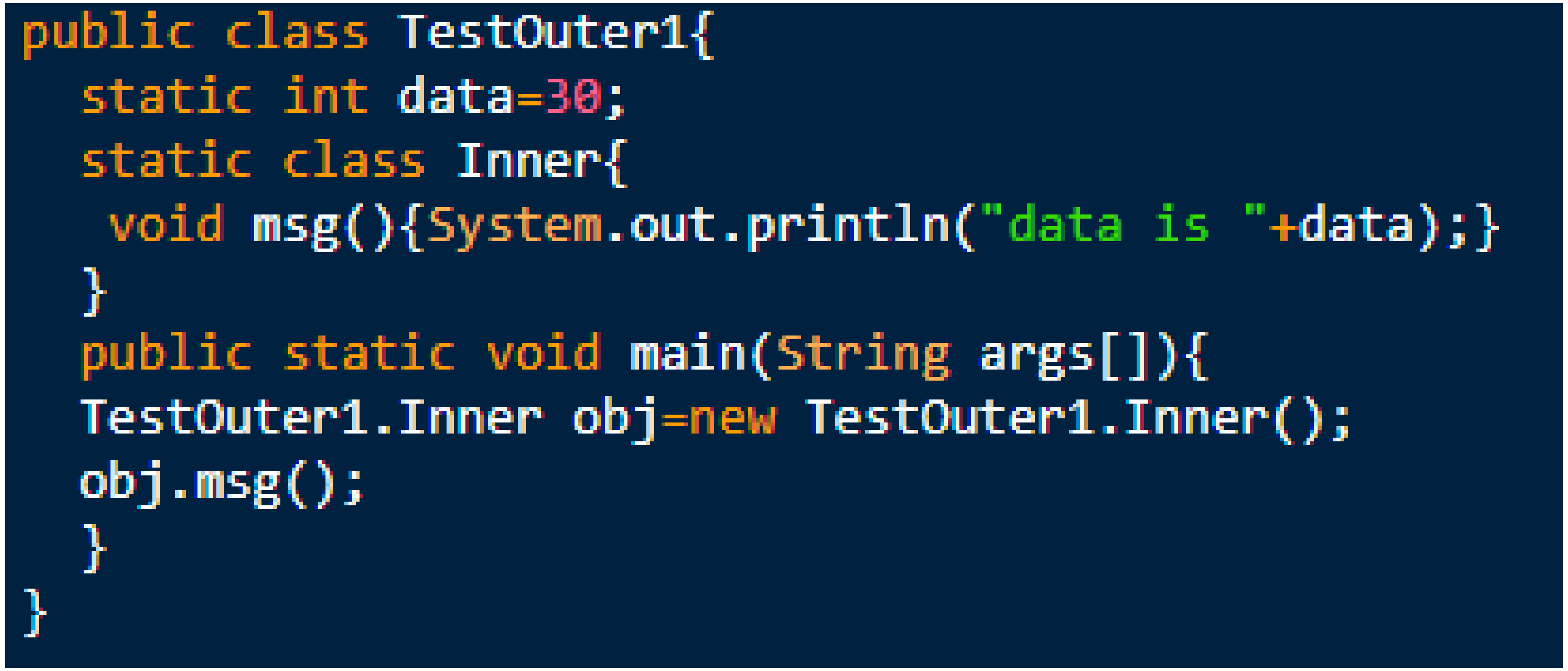
localInner1.java

RAMAIAH
Institute of Technology

Java Static inner class

- A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.
- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java Static inner class



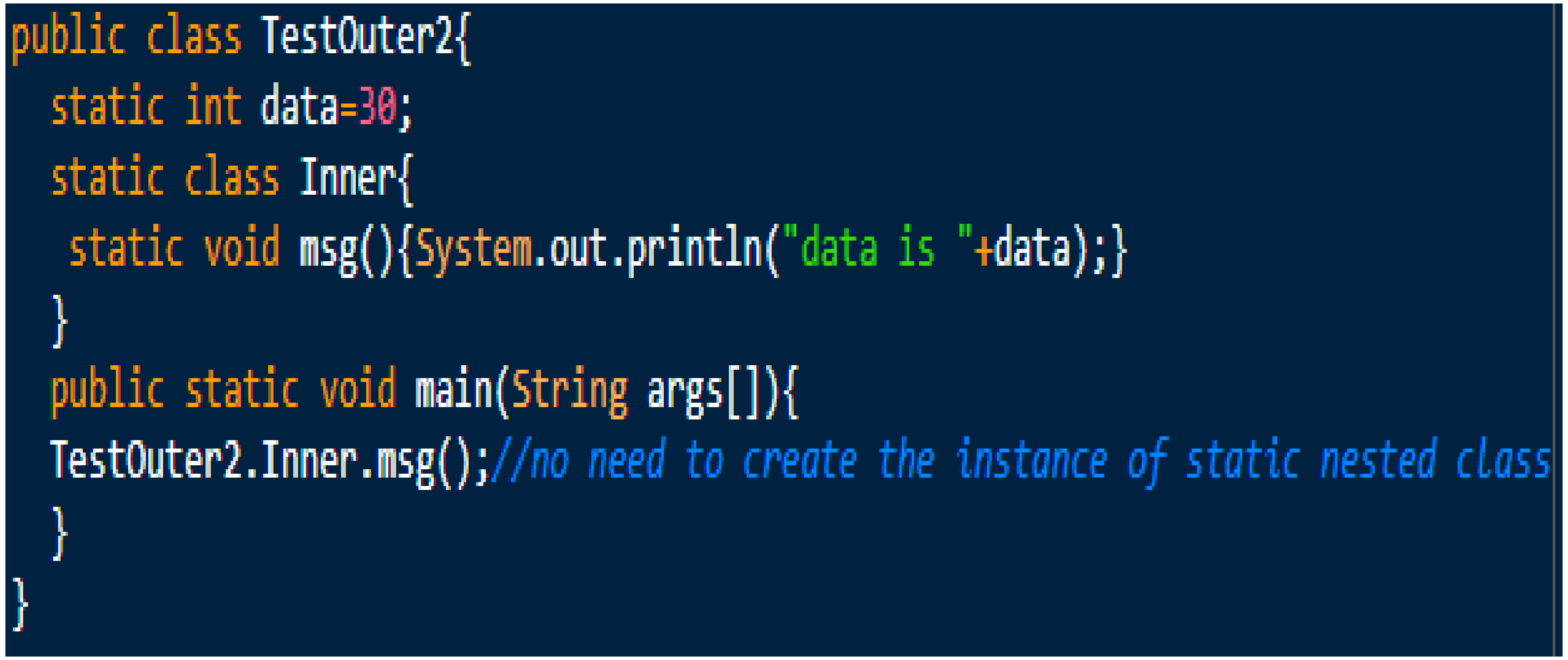
```java
public class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

OUTPUT

```
data is 30
```

TestOuter1.java

Java Static inner class



```
public class TestOuter2{
  static int data=30;
  static class Inner{
   static void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter2.Inner.msg();//no need to create the instance of static nested class
  }
}
```

OUTPUT

```
data is 30
```

TestOuter2.java

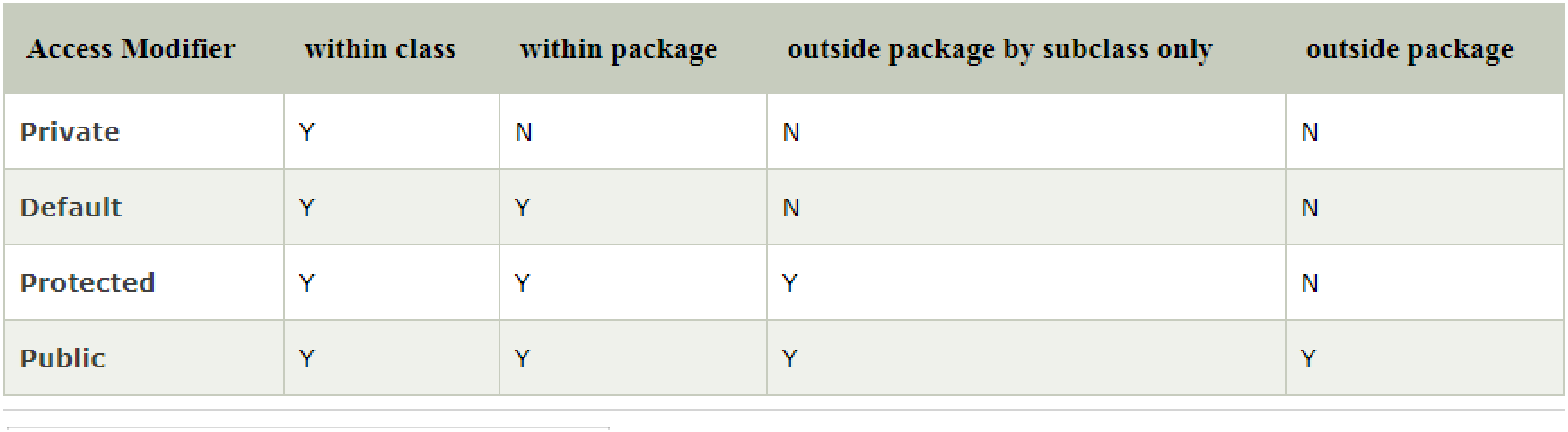
If you have the static member inside static nested class, you don't need to create instance of static nested class.

Access modifiers in Java

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

RAMAIAH
Institute of Technology

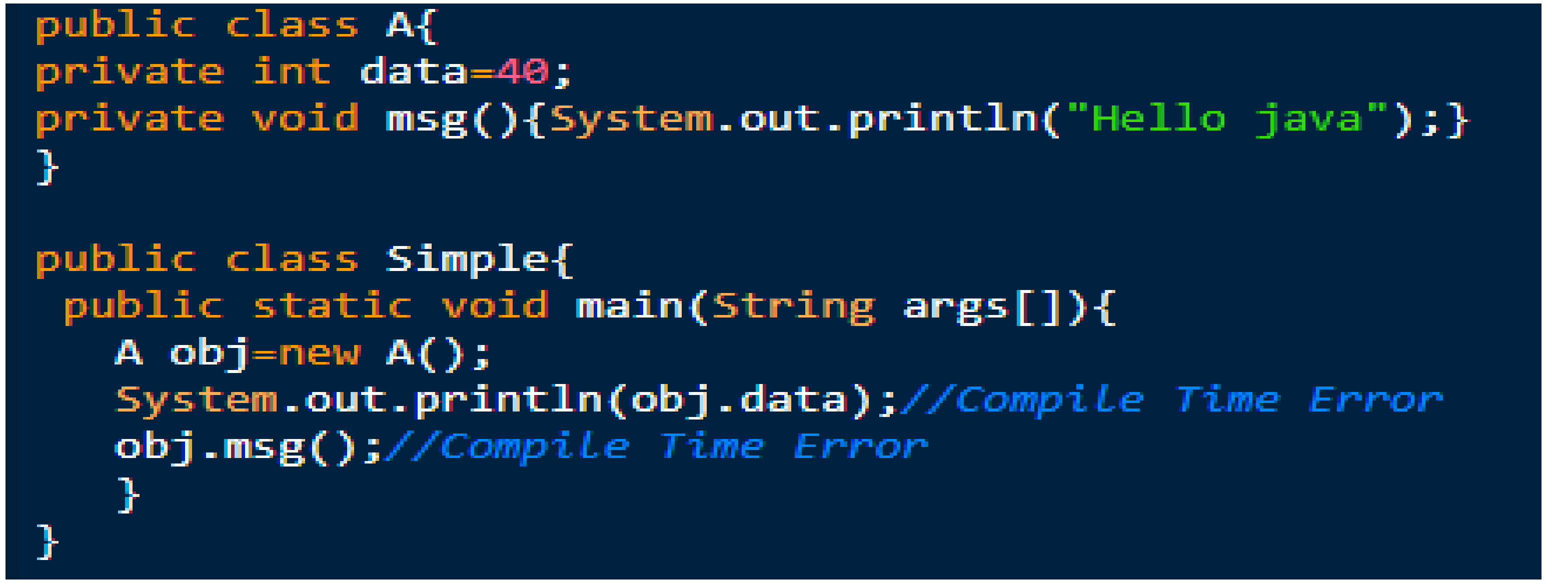
Access modifiers in Java



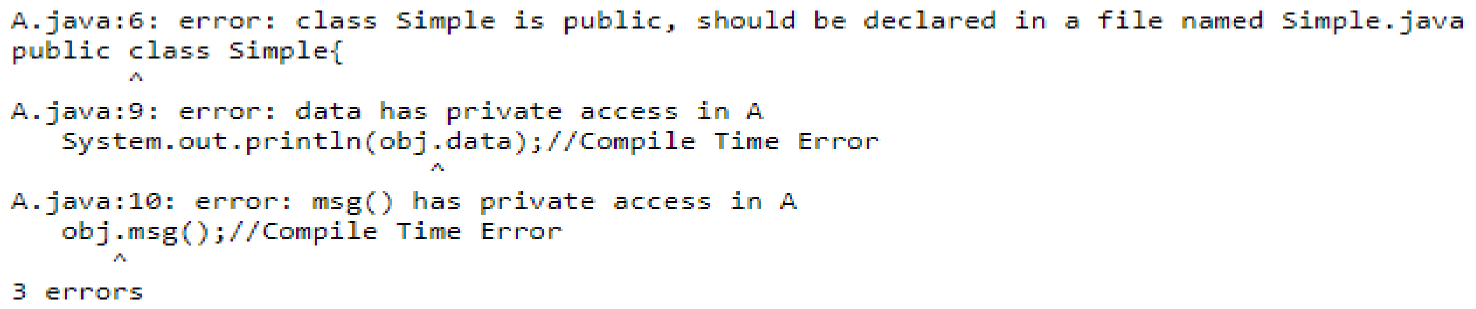
Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

RAMAIAH
Institute of Technology

private



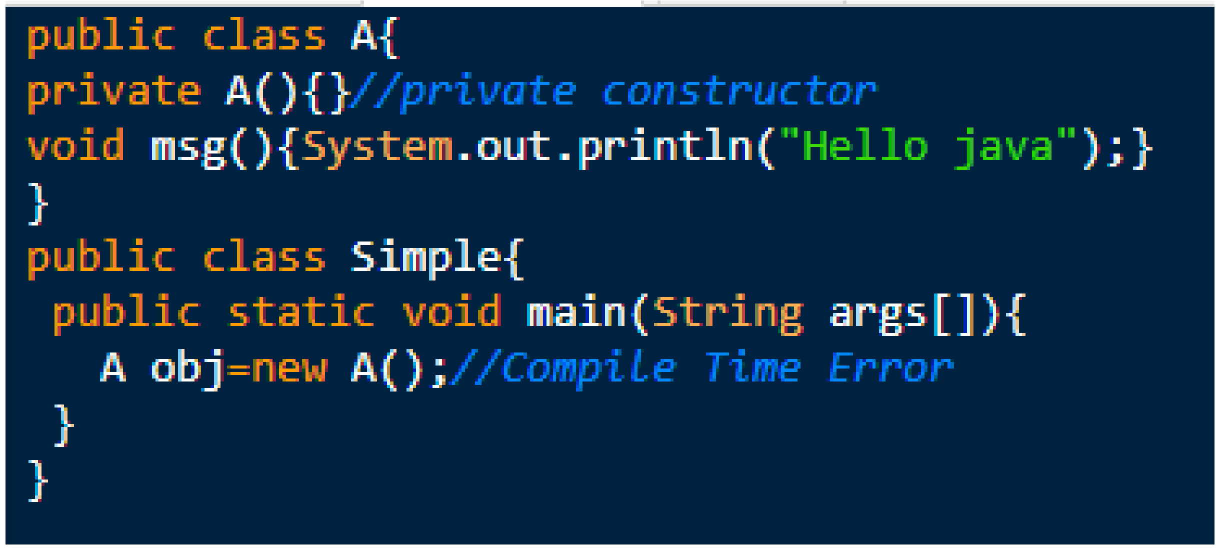
```
public class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

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



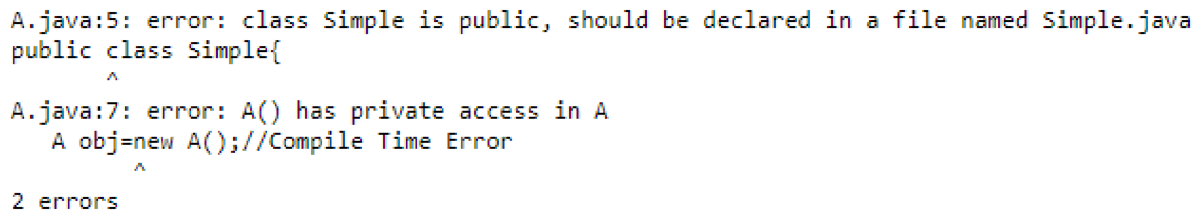
```
A.java:6: error: class Simple is public, should be declared in a file named Simple.java
public class Simple{
       ^
A.java:9: error: data has private access in A
   System.out.println(obj.data);//Compile Time Error
                         ^
A.java:10: error: msg() has private access in A
   obj.msg();//Compile Time Error
      ^
3 errors
```

RAMAIAH
Institute of Technology

private



```
public class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

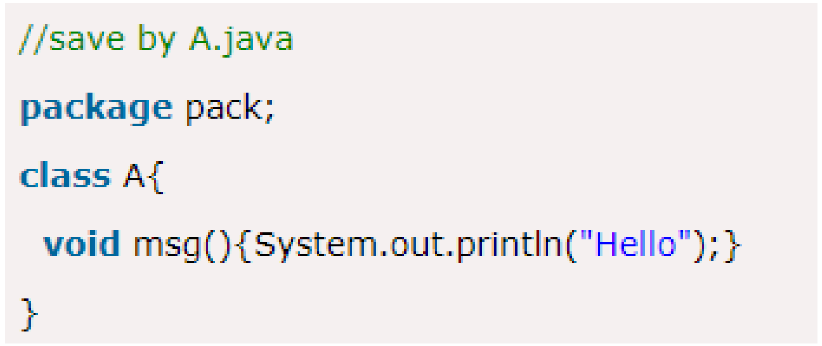


```
A.java:5: error: class Simple is public, should be declared in a file named Simple.java
public class Simple{
       ^
A.java:7: error: A() has private access in A
   A obj=new A();//Compile Time Error
         ^
2 errors
```

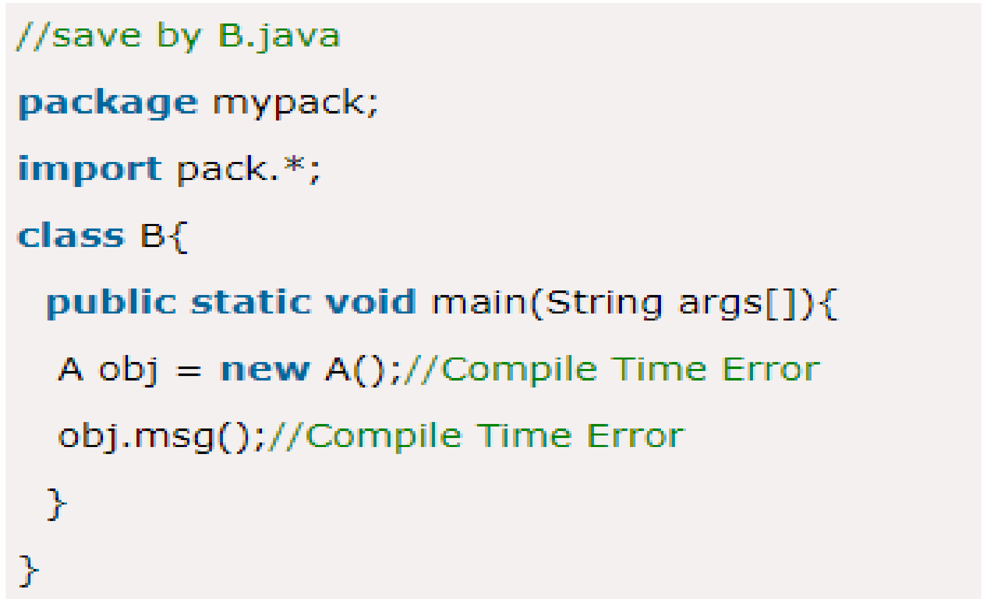
If you make any class constructor private, you cannot create the instance of that class from outside the class.

default

If you don't use any modifier, it is treated as **default** by deaccessible only within package. It cannot be accessed fault. The default modifier is from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.



```
//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
  }
}
```

the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

protected

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- It provides more accessibility than the default modifer.

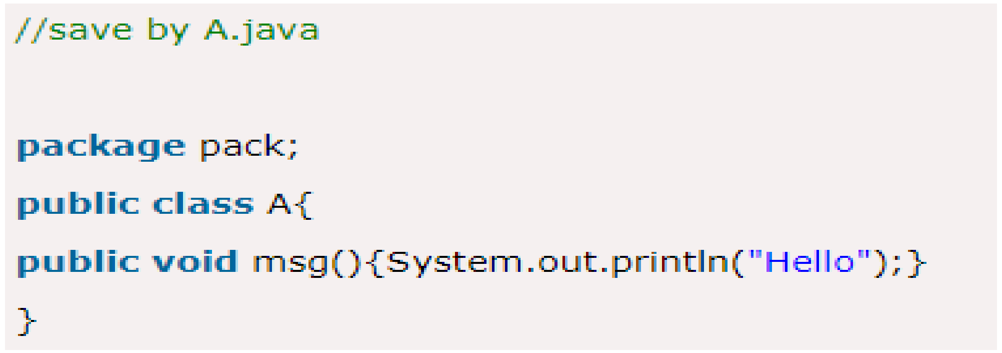
```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

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

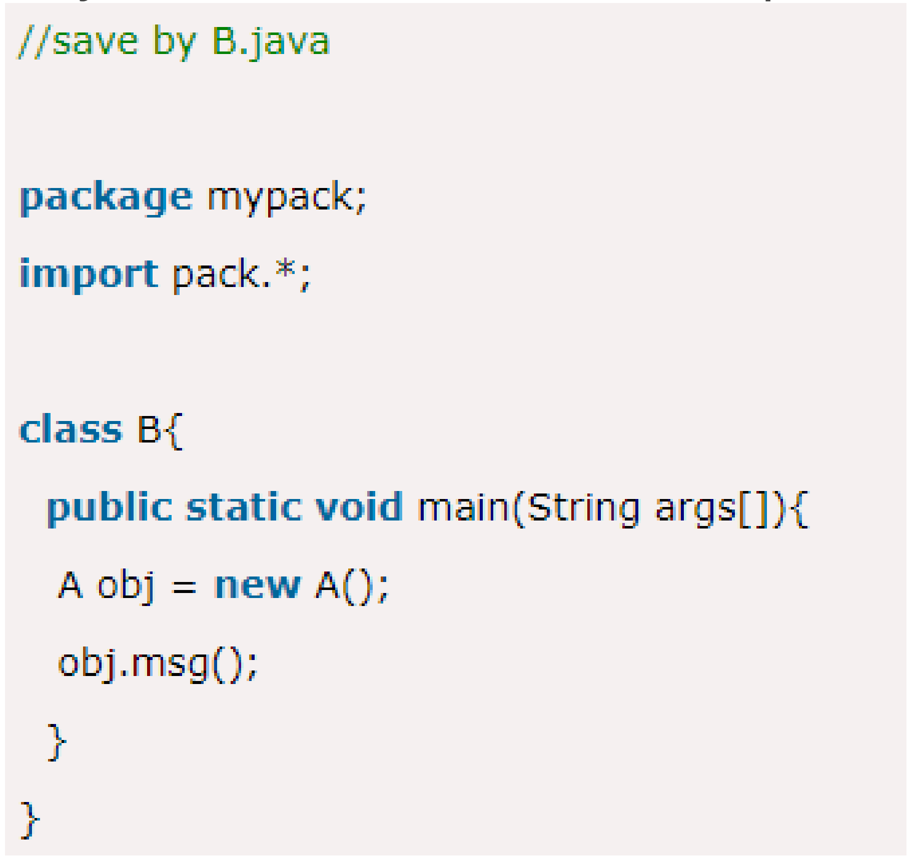
```
Output:Hello
```

public

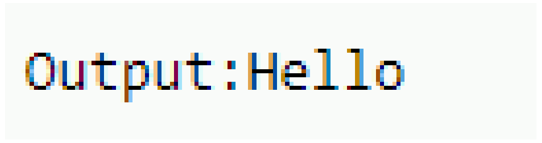
The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.



```
//save by A.java

package pack;
public class A{
public 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();
   obj.msg();
  }
}
```



```
Output:Hello
```

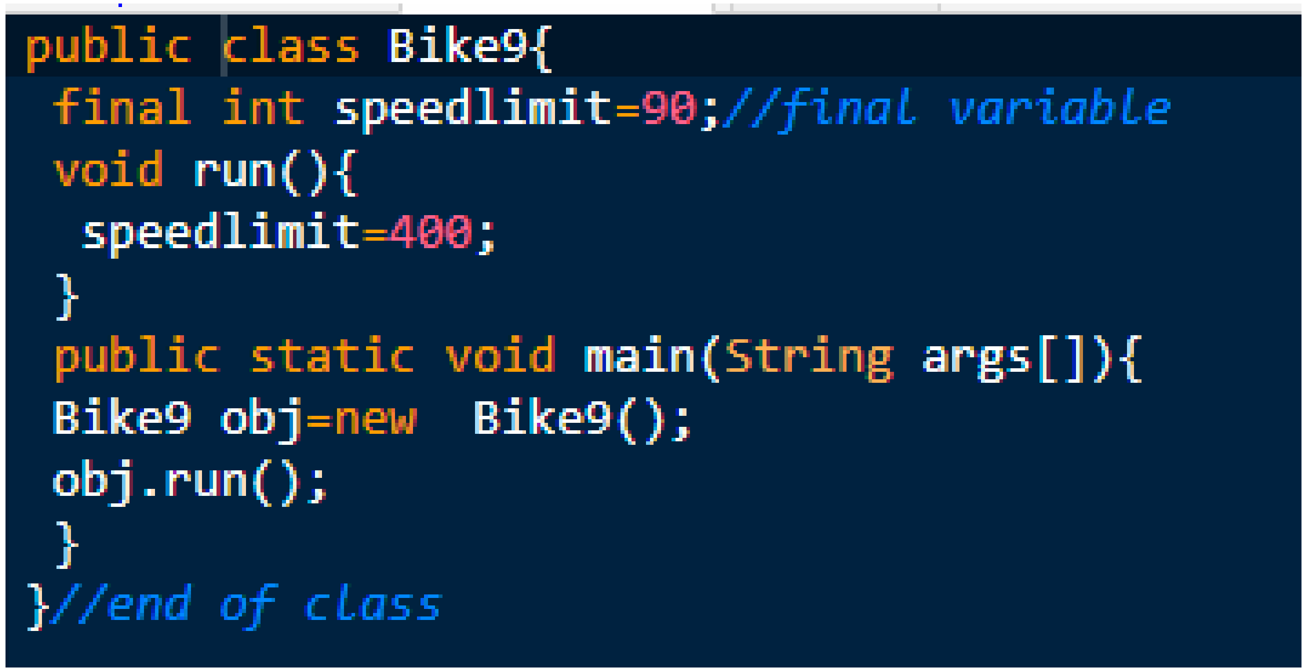
RAMAIAH
Institute of Technology

final keyword in java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

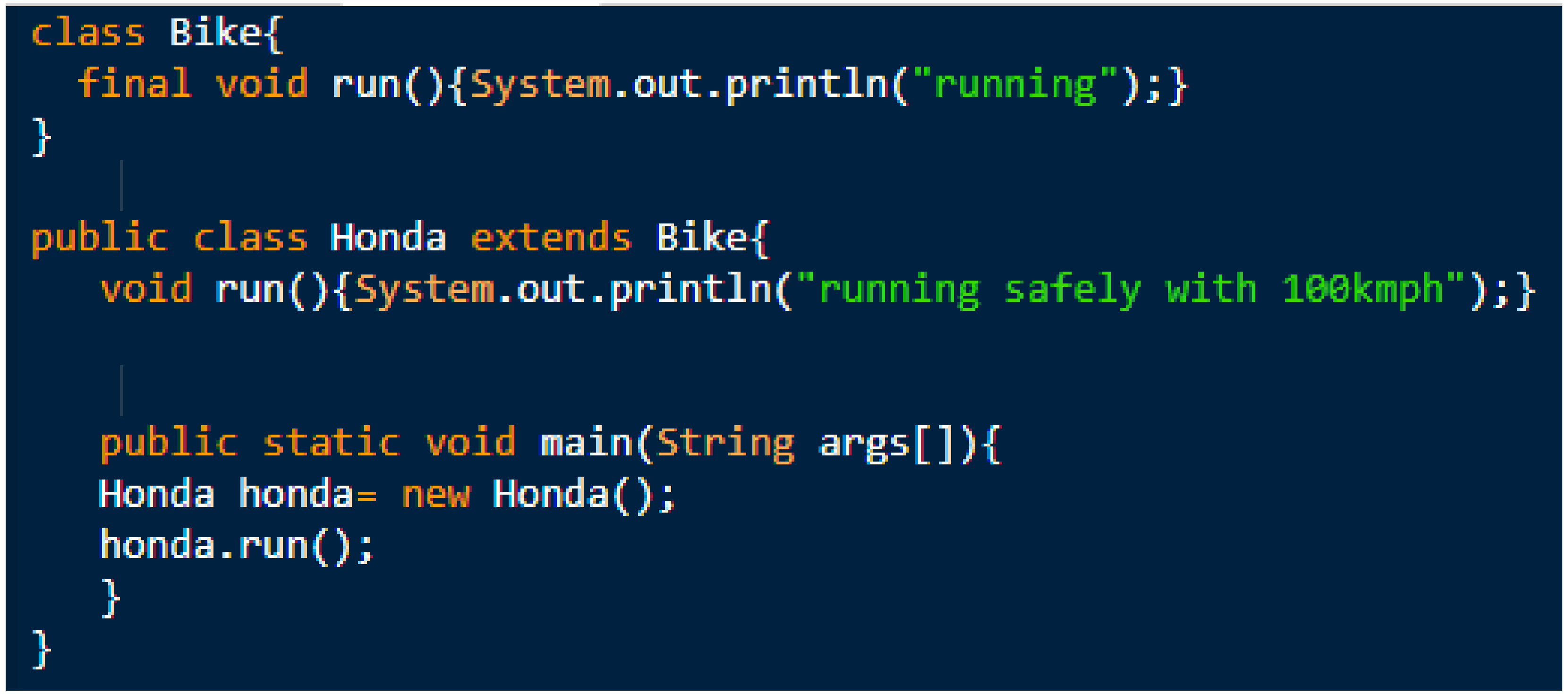
1. variable
2. method
3. class

final variable


```
public class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

```
Bike9.java:4: error: cannot assign a value to final variable speedlimit
  speedlimit=400;
  ^
1 error
```

final method

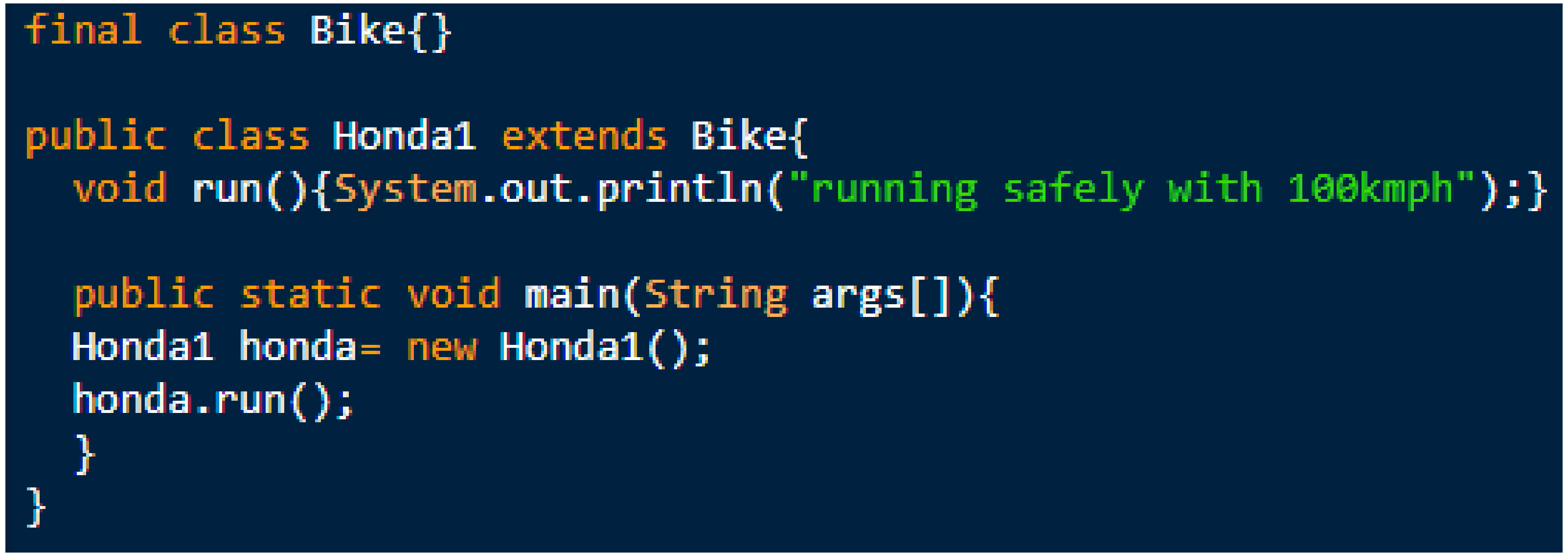


```
class Bike{
  final void run(){System.out.println("running");}
}

public class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}


   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Honda.java:6: error: run() in Honda cannot override run() in Bike
   void run(){System.out.println("running safely with 100kmph");}
        ^
  overridden method is final
1 error
```

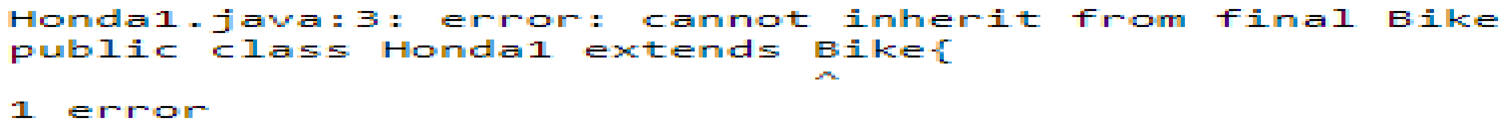
If you make any method as final, you cannot override it.

final class



```
final class Bike{}

public class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

If you make any class as final, you cannot extend it.



```
Honda1.java:3: error: cannot inherit from final Bike
public class Honda1 extends Bike{
                            ^
1 error
```

RAMAIAH
Institute of Technology

Thank you