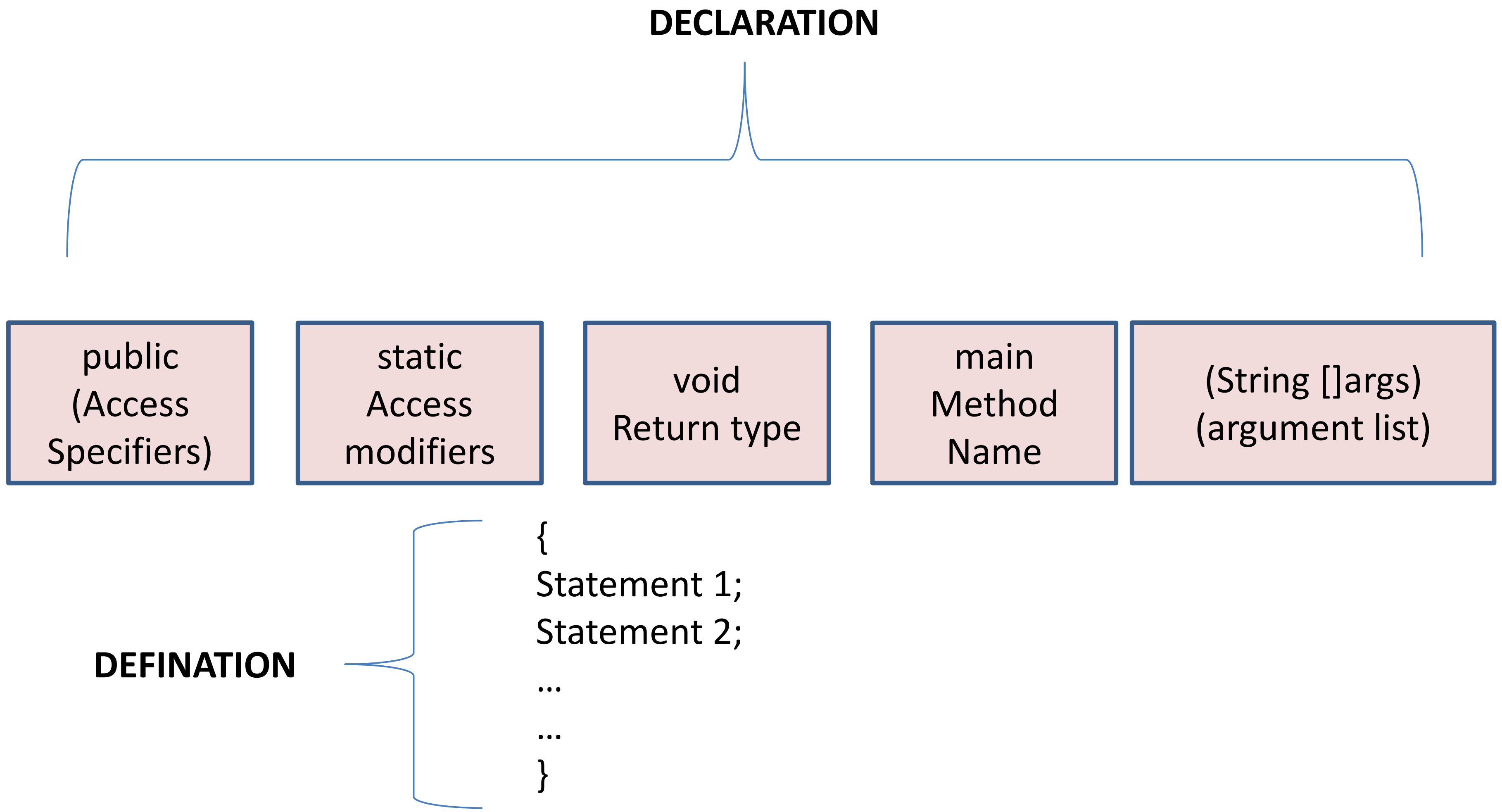


```
{
Statement 1;
Statement 2;
…
…
}
```

- <u>Access Specifies :</u> Specifies the visibility of code component
- Defines component is belonging to particular class or object
- If return type is other than void then program should have return statement at he end of he program

METHODS

- Methods are named block of codes which performs specific tasks
- Method being called is known as called method
- Method calling a method is known as calling method.
- One class can contain multiple methods.
- But method cannot contain another method inside it.
- Any pre-defined words in java starting with small letter is a method which should have () after the name of the method. If there is no () then it is a variable.

ARRAYS

- Group of homogeneous data that has some index and fixed size

DECLARATION:

1. **datatype []arrayName;**
2. **datatype[] arrayName;**
3. **datatype arrayName[];**

CREATION:

arrayName =new datatype[SIZE];

INITIALIZATION:

arrayName[index]=value;

DIFFERENT WAYS OF ARRAY DECLARATION AND INITIALIZATION

- **datatype[] arrayName = new datatype[SIZE] //all in single line**
- **Datatype[] arrayName = {value1,value2,…};**

STRINGS

- String is a reference type.As a result string variable holds the address to an object created by string class.
- Even though strings are primitive types java compiler has some features designated to let you work as non- primitive data types.
- Java automatically converts the primitive type to a string.
- String is a sequence of characters.
- Strings are immutable.Once created cannot change the value or one cannot reinitialization.
- User-defined data type.
- Operator overloading is supported

String s = “hello”;

String str = “hello”;

DECLARATION AND INITIALIZATION

String stringName;
Str = “ value “ ;

String stringName=new String(“ value “);

- The string contains two parts.

    Constant Pool →Duplication not allowed

    Non-constant pool →Duplication are allowed

String Class

- It is powerful but it is not efficient.
- Because string objects are immutable, any method of the string class that modifies the string in any way must create a new string object. To overcome this problem java offers two alteration to the string class, **StringBuffer and StringBuilder.**
- StringBuilder and StringBuffer are two major images.Both have the same methods and perform the same string manipulation.
- StringBuffer is **thread safe** where as StringBuilder is **not thread safe.**

- We cannot assign string literals(values) directly using StringBuilder and StringBuffer we have to create an object using new keyword.

CONSRUCTORS

```
public StringBuffer() {
      super(16);
}
```

Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

```
public StringBuffer(int capacity) {
      super(capacity);
}
```

Constructs a string buffer with no characters in it and the specified initial capacity.

@param capacity the initial capacity.

@exception NegativeArraySizeException if the {@code capacity} argument is less than {@code 0}.

```
StringBuffer sb2 = new StringBuffer(-1);
```

```
public StringBuilder() {
      super(16);
   }
```

```
public StringBuilder(int capacity) {
      super(capacity);
   }
```

```
public StringBuffer(String str) {
        super(str.length() + 16);
        append(str);
 }
```

Constructs a string buffer initialized to the contents of the specified string. The initial capacity of the string buffer is
{@code 16} plus the length of the string argument.
@param   str   the initial contents of the buffer.

```
StringBuffer sb = new StringBuffer("hello");
System.out.println(sb.length());//5
System.out.println(sb.capacity());//21
```

```
public StringBuilder(String str) {
        super(str.length() + 16);
        append(str);
    }
```

```
public StringBuffer(CharSequence seq) {
        this(seq.length() + 16);
        append(seq);
    }
```

```
public StringBuilder(CharSequence seq) {
        this(seq.length() + 16);
        append(seq);
    }
```

* Constructs a string buffer that contains the same characters
* as the specified {@code CharSequence}. The initial capacity of
* the string buffer is {@code 16} plus the length of the
* {@code CharSequence} argument.
* <p>
* If the length of the specified {@code CharSequence} is
* less than or equal to zero, then an empty buffer of capacity
* {@code 16} is returned.
* **@param seq the sequence to copy.**
* **@since 1.5**

String	StringBuffer	StringBuilder
We can assign the values directly to a string variable	Cannot create a string without new keyword	
There are 16 constructors	There are only 4	
Not efficient	More Efficient	
Because of immutability cannot be used in threads	Threads safe	Not thread safe
When two different strings given same values they refer to same address	Does not refer to same address	
	Synchronization is affected due to thread safety	Better than StringBuffer

STRING METHODS

String str = " jung kook";

length()	To print length of a string. Return type is int. **str.length(); //9**
toCharArray()	To convert string to char array. Return type is char array **char[] ch = str.toCharArray();** **System.out.println(ch[8]);//k**

charAt(index)	To get the character from the specified string. Return type is char.
	str.charAt(7) // 4
	str.charAt(11); //string index out of range

equals(stringName)	To compare two strings. Concerned with cases. Return type is boolean. str.equals(str2) //false (str2= "JUNG_kook")

String str = “ jung kook”; String str2 ="JUNG_kook"

replace(oldChar,newChar)	Replaces old characer with new. Return type is String. **str.replace('o','i')//Jung_Kiik**
equalsIgnoreCase(stringName)	Compares two strings. Not concerned with cases. Return type is boolean. **str.equalsIgnoreCase(str2)//true**
indexOf(singleChar)	To get index of specified character. First occcurence char index will be get . Return type is integer. **str.indexOf('o')//6**
toUpperCase()	To convert all the characters of string to upper case. Return type is string. **str.toUpperCase()//JUNG_KOOK**

toLowerCase();	To convert all the characters of string to lower case. Return type is string **str.toLowerCase()//jung_kook**

substring(index)
 	
substring(beginning , ending))
	Prints letters after the index. Return type is string.
	str.substring(3) //g_Kook
	Print letters from beginning letter to ending-1 letters. Return type is string .
	str.substring(3,7) //g_Ko

contains(string)	To check whether the following string is present or not. Return type is boolean . **str.contains("Jun")//true**

REFERENCE TYPE

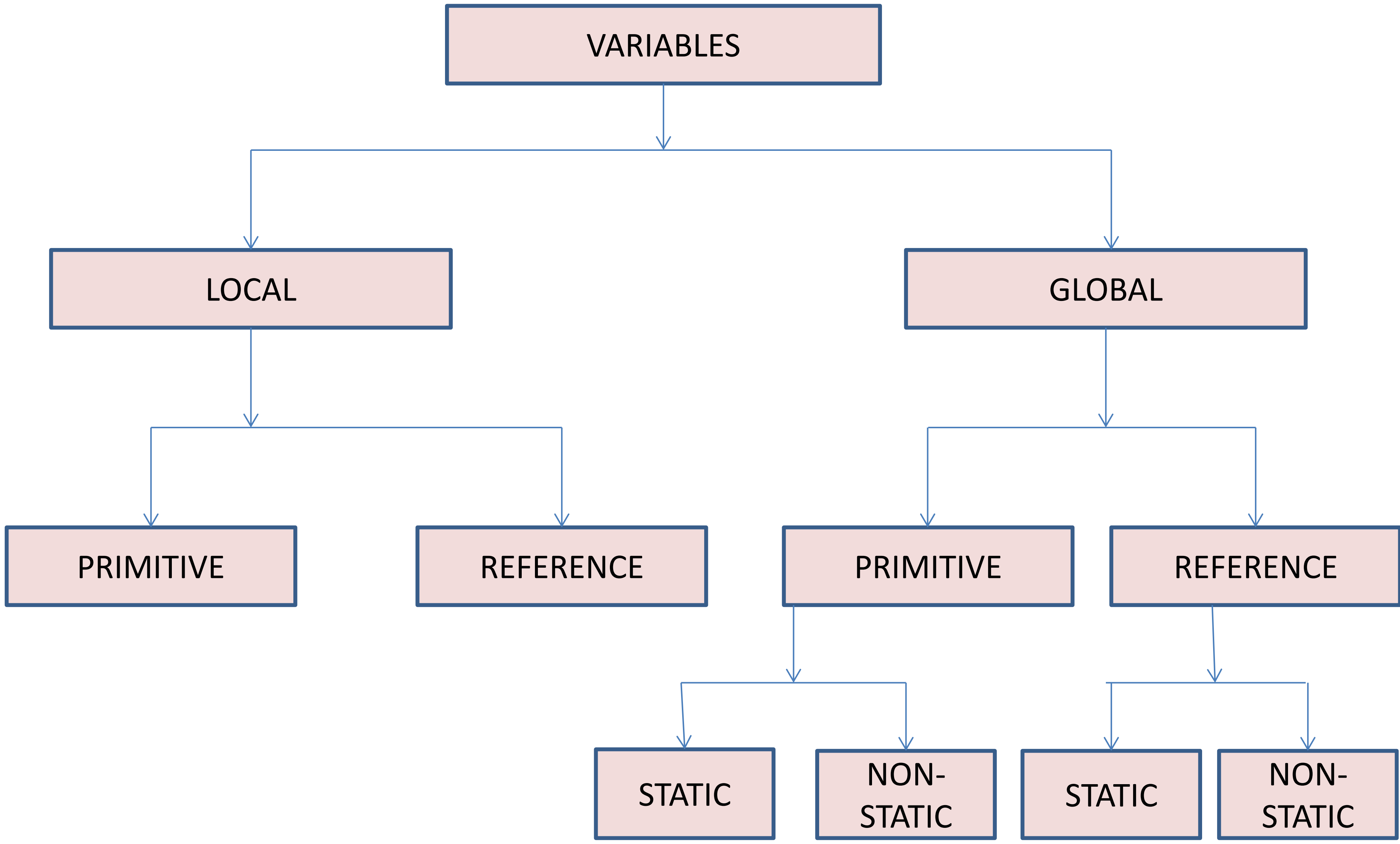
- Reference type is a type that is based on class rather than primitive data type
- It can be based on pre-defined classes in java or classes defined by programmer or developer.
- Purpose of using new is to create objects. Since new is used in creating string it will create new object of string.
- Reference type does not store content of the object but the address of the object where it is stored.

STATIC

- Any member of the class that has been declared as static is a static member.
- No need to create a object of class which has been declared as static. Static member of a class can be used in another class without creating object.
- static variable declaration inside static method is not allowed

NON-STATIC

- Member of class which has been not declared with static keyword.
- When used in another class then object creation is mandatory


VARIABLES
LOCAL
GLOBAL
PRIMITIVE
REFERENCE
PRIMITIVE
REFERENCE
STATIC
NON-STATIC
STATIC
NON-STATIC

```
public class static_non_static {

public static void main(String[] args) {

static_non_static sm = new static_non_static();

int a=calcArea(4);                    //static method
System.out.println(a+" is the area of square");

int areaR = sm.areaRec(4,2);   //non-static method
System.out.println(areaR+" is the area of rectangle");

}
public static int calcArea(int side){
int area=side*side;
return area;
}

public  int areaRec(int l,int b){
int area=l*b;
return area;
}

}
```

```
public class static_non_static {

int a=0;                         //non-static variable
static int area =0;         //static variable
public static void main(String[] args) {

static_non_static sm = new static_non_static();

sm.a=calcArea(4);        //using non-static variable
System.out.println(sm.a+" is the area of square");

int areaR = sm.areaRec(4,2);
System.out.println(areaR+" is the area of rectangle");

}
public static int calcArea(int side){
area=side*side;               //using static variable
return area;
}
public  int areaRec(int l,int b){
area=l*b;
return area;
}
}
```

```
public static int calcArea(int side){
area=side*side;
int n = sm.areaRec(2,3); //local variable
System.out.println(n+" is the area of rectangle");
return area;
}
```

CONSTRUCTORS

- It is a special type of method which has a same name as class name and does not have any return type.
- Invoked when object is created.

```
public class demo{
}
```

```
D:\UST_Global\java>javac demo.java

D:\UST_Global\java>javap demo
Compiled from "demo.java"
public class demo {
  public demo();
}
```

```
public class demo{
		public demo(int i){
		}
		public static void main(String args[]){
		}
}
```

```
D:\UST_Global\java>javac demo.java

D:\UST_Global\java>javap demo
Compiled from "demo.java"
public class demo {
  public demo(int);
  public static void main(java.lang.String[]);
}
```

```
public class demo{
		public static void main(String args[]){
		}
}
```

```
D:\UST_Global\java>javac demo.java

D:\UST_Global\java>javap demo
Compiled from "demo.java"
public class demo {
  public demo();
  public static void main(java.lang.String[]);
}
```

- No-argument constructor will be created by programmer.
- Can have statements inside body.

```
  public ConstructorExample() {
          System.out.print("helloo Vibha");
  }
  public static void main(String[] args) {
        ConstructorExample ce= new ConstructorExample();
 }
```

- Default constructor will be created by compiler.
- Cannot have any statements inside body.

- Parameter Constructor : Constructor having arguments passed

```
public ConstructorExample(String i) {
System.out.println("helloo Piaa");
}
ConstructorExample ce1= new ConstructorExample("a");
```

CONSTRUCTOR OVERLOADING

DIFFERENT TYPE AS PARAMETERS:

```
public ConstructorExample( int i ) {
System.out.println("helloo Vidya");
}

public ConstructorExample( String i ) {
System.out.println("helloo Piaa");
}


public static void main(String[] args) {

ConstructorExample ce= new ConstructorExample( 1 );
ConstructorExample ce1= new ConstructorExample( "a“ );
}
```

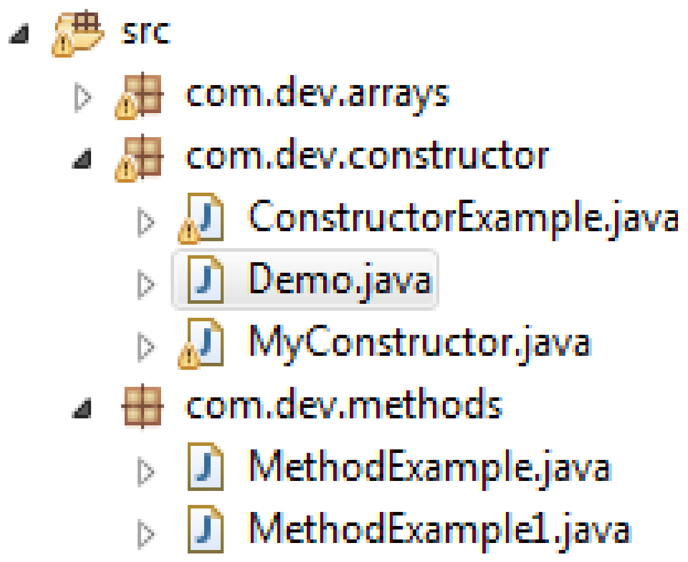
Different order of arguments

```
public ConstructorExample(String i,int j) {
System.out.println("consructor with first string and second is int");
}

public ConstructorExample(int i,String j) {
System.out.println("consructor with first int and second is String");
}


 public static void main(String[] args) {

 ConstructorExample ce2= new ConstructorExample("a",1);
 ConstructorExample ce3= new ConstructorExample(1,"a");
 }
```

PACKAGES

- To import any classes from one package to another package we use the import statement.
- Package helps in managing data in proper manner and helps in code re-useability

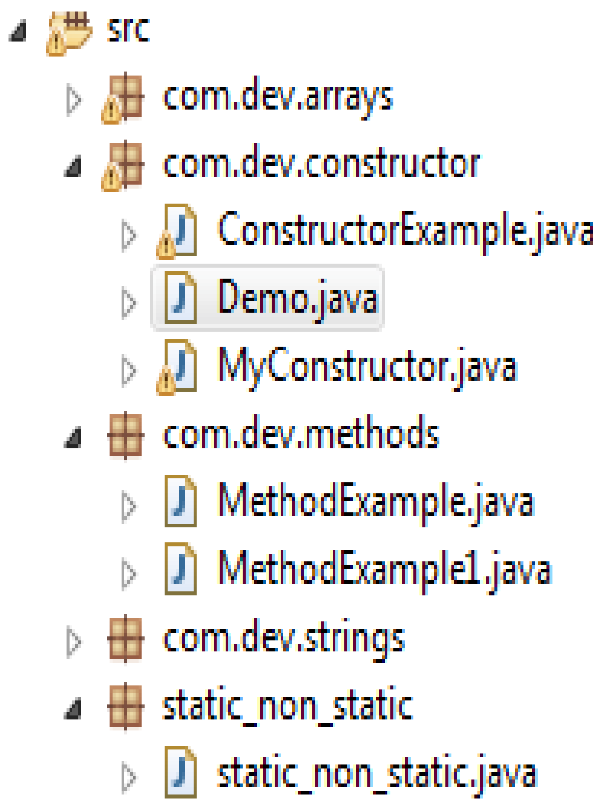
```
package com.dev.constructor;

import com.dev.methods.MethodExample;

public class Demo {

public static void main(String[] args) {
int i = MethodExample.calcArea(3);
System.out.println(i);
}

}
```



```
src
  com.dev.arrays
  com.dev.constructor
    ConstructorExample.java
    Demo.java
    MyConstructor.java
  com.dev.methods
    MethodExample.java
    MethodExample1.java
```

static_non_static.java

```
public static int area=0;

public static int calcArea(int side){
area=side*side;
int n= sm.areaRec(2,3);
return area;
}

public  int areaRec(int l,int b){
area=l*b;
return area;
}
```



```
src
  com.dev.arrays
  com.dev.constructor
    ConstructorExample.java
    Demo.java
    MyConstructor.java
  com.dev.methods
    MethodExample.java
    MethodExample1.java
  com.dev.strings
  static_non_static
    static_non_static.java
```

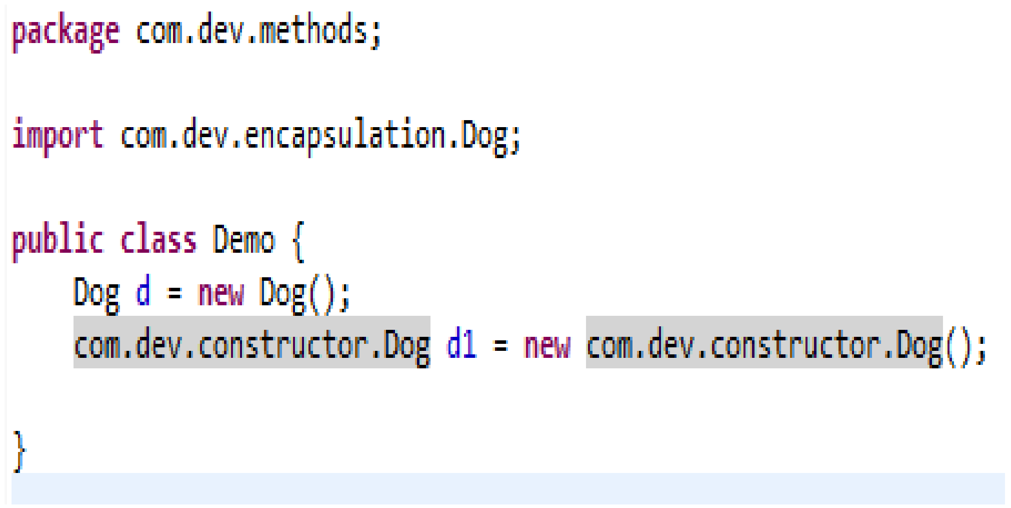
Demo.java

```
import static_non_static.static_non_static;  //importing package

int j = static_non_static.calcArea(3);          //class from static_non_static
System.out.println(static_non_static.area); //static variable from static_non_static
```

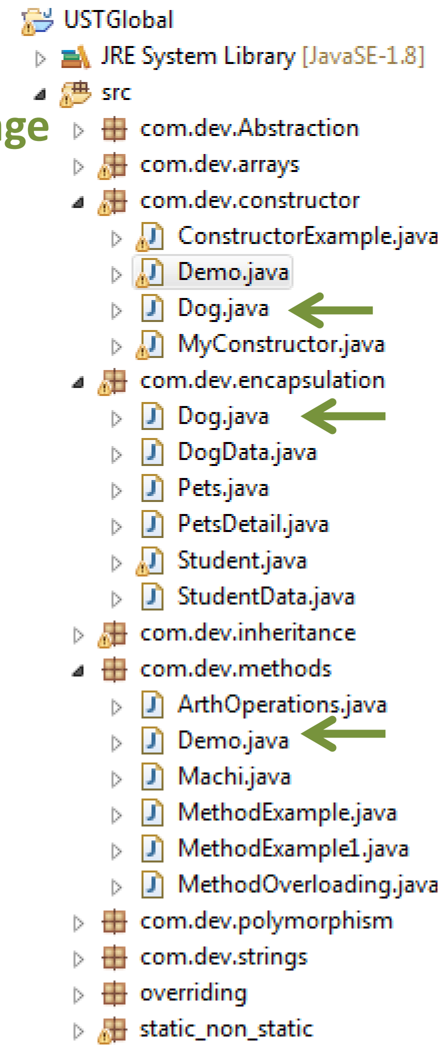
When one wants to access two classes from two different packages having same class name, one class package can be imported and another by fully qualified

import com.dev.methods.MethodExample; → Importing package

com.dev.methods.MethodExample → Fully qualified class



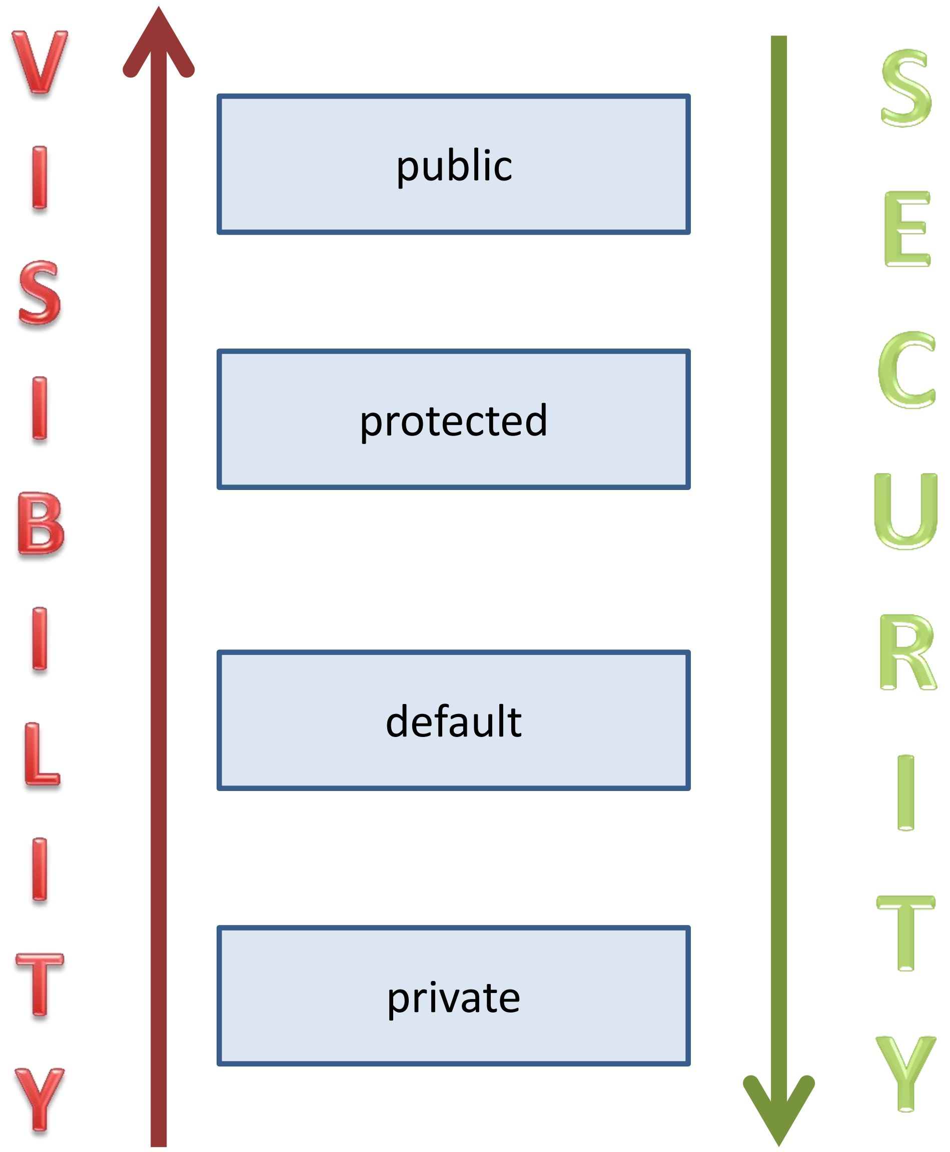
```
package com.dev.methods;

import com.dev.encapsulation.Dog;

public class Demo {
    Dog d = new Dog();
    com.dev.constructor.Dog d1 = new com.dev.constructor.Dog();

}
```



```
USTGlobal
  JRE System Library [JavaSE-1.8]
  src
    com.dev.Abstraction
    com.dev.arrays
    com.dev.constructor
      ConstructorExample.java
      Demo.java
      Dog.java
      MyConstructor.java
    com.dev.encapsulation
      Dog.java
      DogData.java
      Pets.java
      PetsDetail.java
      Student.java
      StudentData.java
    com.dev.inheritance
    com.dev.methods
      ArthOperations.java
      Demo.java
      Machi.java
      MethodExample.java
      MethodExample1.java
      MethodOverloading.java
    com.dev.polymorphism
    com.dev.strings
    overriding
    static_non_static
```

DATA-TYPES

- Entity declared as **public** can be used by any other classes. Can be accessed by any of the classes or packages
- Method declared as **protected** cannot be accessed by any other packages. Can be accessed by different classes in same package.
- Can be used under certain conditions.(inheritance)
- Package level/ **default** can be accessed by any of the classes in same package. Cannot be used in other packages.
- **Private** entities can be accessed only within that class.


VISIBILITY
public
protected
default
private
SECURITY

ASSOSIATION

Association in java is relationship between two different classes.

1. One to one
2. One to many
3. Many to many
4. Many to one

$$\begin{tikzcd}
 & \text{Aggregation :A class not dependent on another class.} \\
\text{Has A Relationship} \arrow[ur] \arrow[dr] & \\
 & \text{Composition: A class dependent on another class}
\end{tikzcd}$$

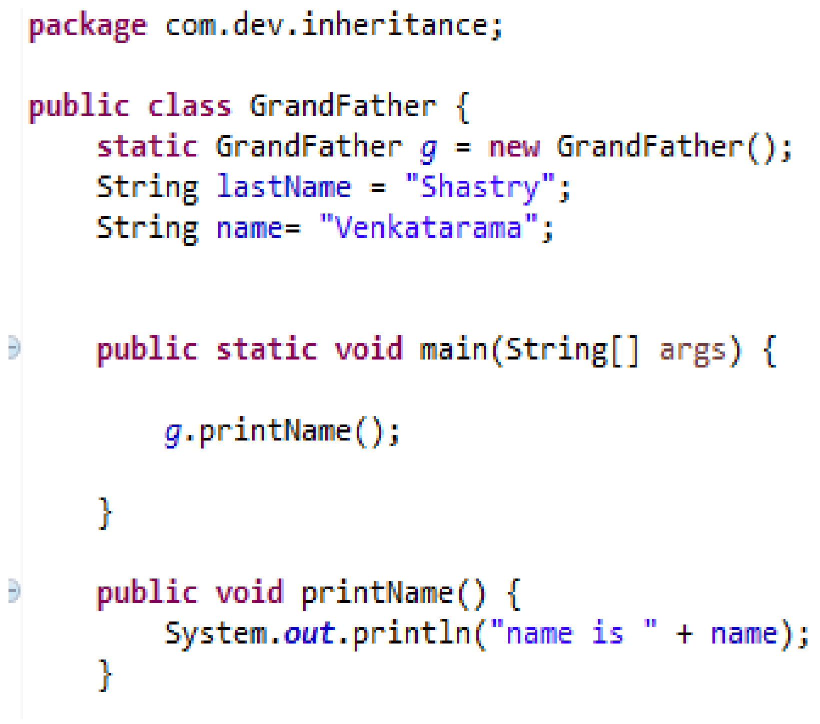
$$\begin{tikzcd}
\text{Is A Relationship} \arrow[r] & \text{Inheritance}
\end{tikzcd}$$

INHERITANCE

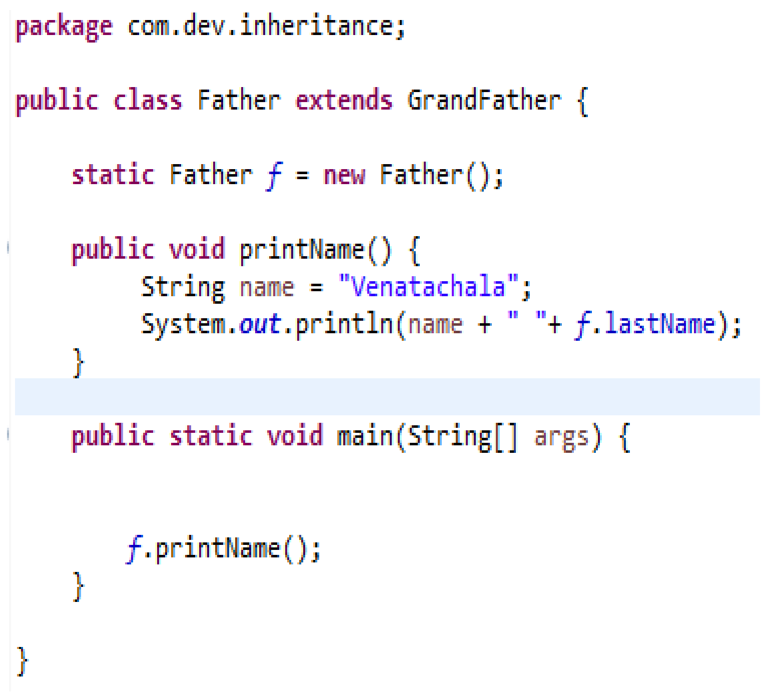
- The process by which one class acquires the properties and functionalities of another class is called as inheritance.
- The aim of inheritance is to provide code **re-useability.**
- Class whose properties or functionalities are being inherited by other class is called "**parent class , super class or base class".**
- Class that inherits properties from other class is called **"child class, sub class o derived class".**
- To make use of inheritance in java make use of keyword **extends**
- One can access properties of super and sub class using object of sub class.
- **Final classes** cannot be inherited , final members of a super class can be inherited but cannot be changed.
- Private members and constructors of super class cannot be inherited.

TYPES OF INHERITANCE

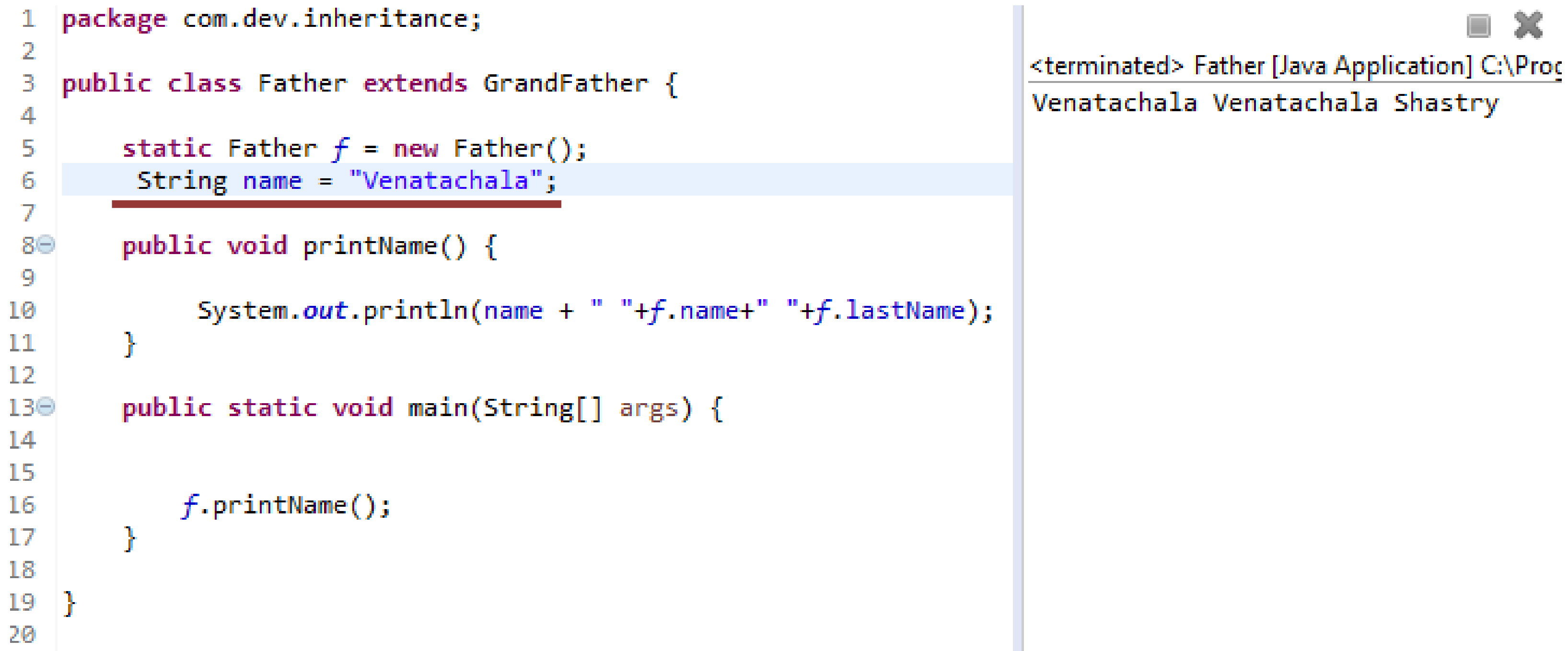
Single Inheritance : A class aquiring the properties from another class.



```java
package com.dev.inheritance;

public class GrandFather {
    static GrandFather g = new GrandFather();
    String lastName = "Shastry";
    String name= "Venkatarama";


    public static void main(String[] args) {

        g.printName();

    }

    public void printName() {
        System.out.println("name is " + name);
    }
```

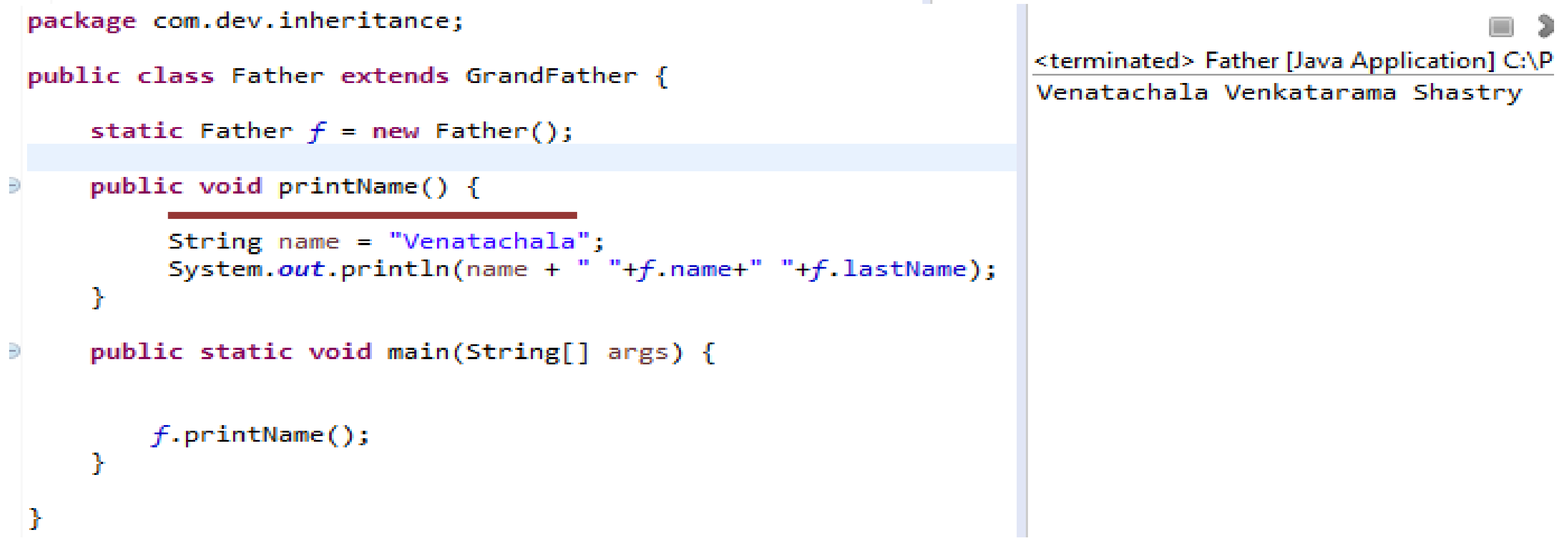


```java
package com.dev.inheritance;

public class Father extends GrandFather {

    static Father f = new Father();

    public void printName() {
        String name = "Venatachala";
        System.out.println(name + " "+ f.lastName);
    }

    public static void main(String[] args) {


        f.printName();
    }

}
```



```java
package com.dev.inheritance;

public class Father extends GrandFather {

    static Father f = new Father();
     String name = "Venatachala";

    public void printName() {

         System.out.println(name + " "+f.name+" "+f.lastName);
    }

    public static void main(String[] args) {


        f.printName();
    }

}

```

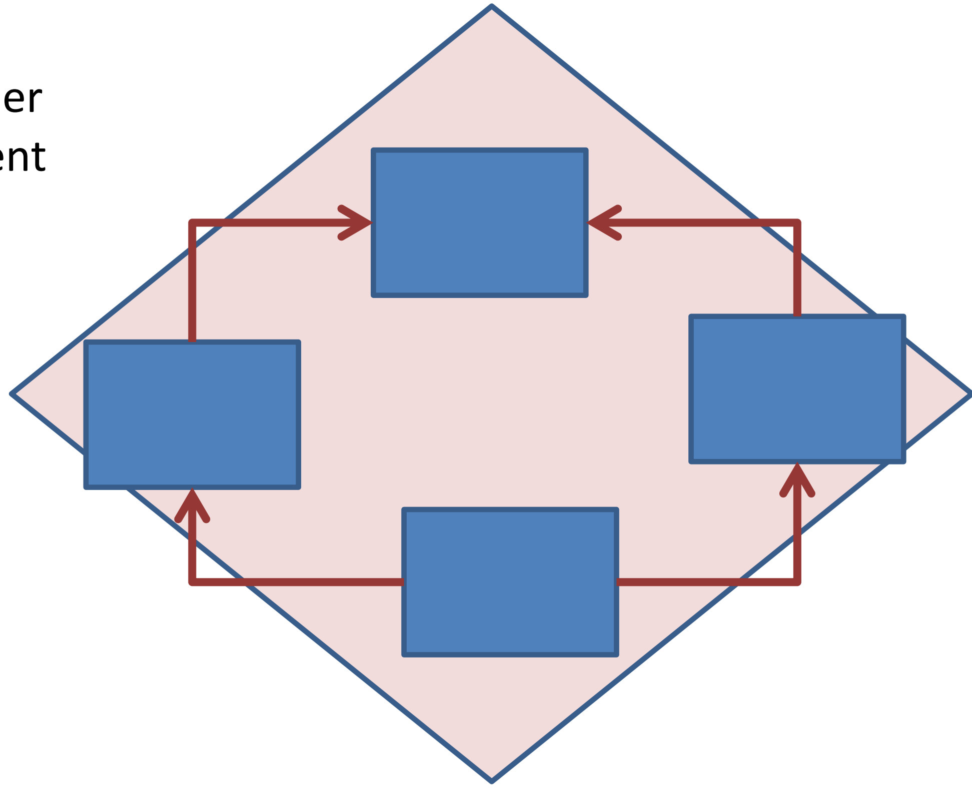
```
<terminated> Father [Java Application] C:\Proc
Venatachala Venatachala Shastry
```



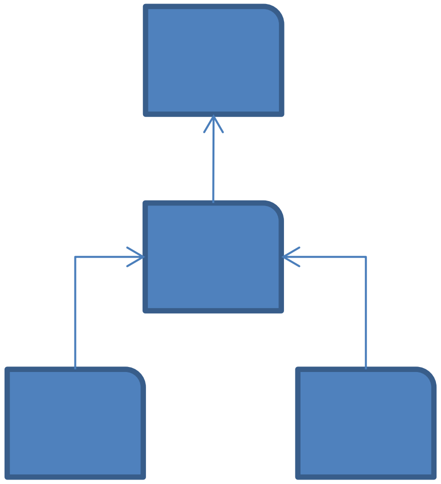
```java
package com.dev.inheritance;

public class Father extends GrandFather {

    static Father f = new Father();

    public void printName() {

         String name = "Venatachala";
         System.out.println(name + " "+f.name+" "+f.lastName);
    }

    public static void main(String[] args) {


        f.printName();
    }

}
```

```
<terminated> Father [Java Application] C:\P
Venatachala Venkatarama Shastry
```

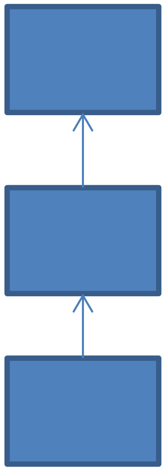
Multiple level is not supported in java. Compiler gets confused to acquire properties from different parents.
Diamond problem.



Hierarchical level

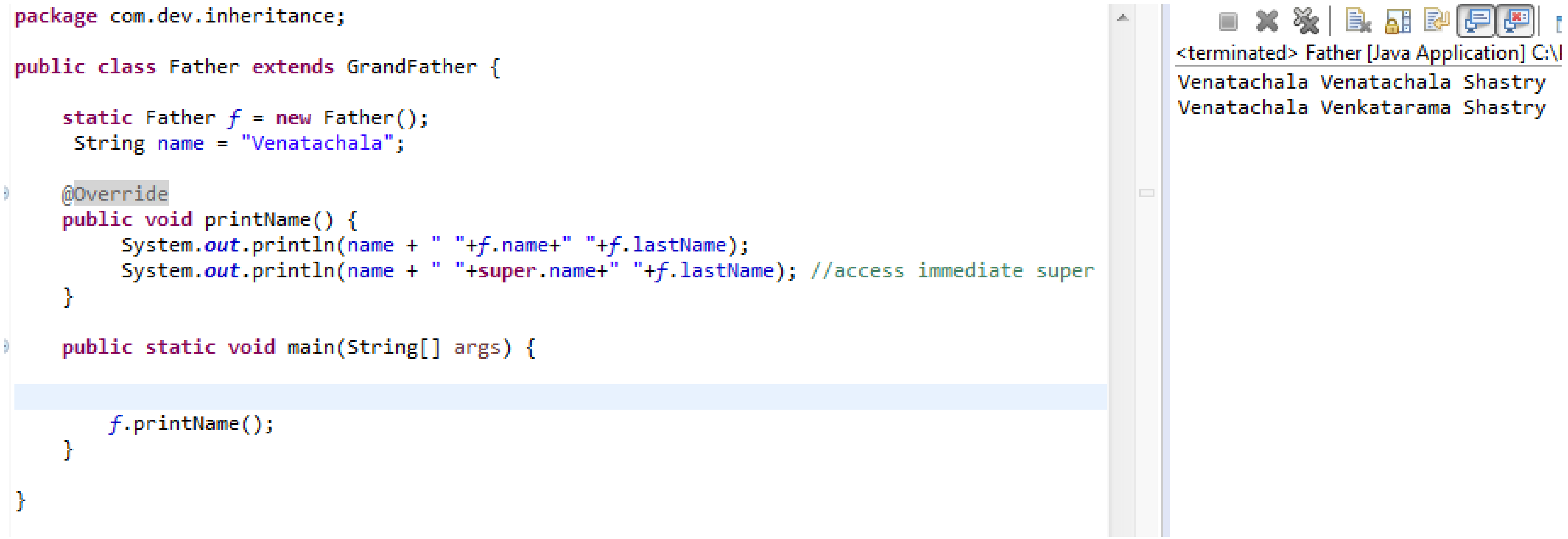


Multi level



SUPER

- Super keyword refers to an object of immediate parent o super class.
- To access the data member of a parent class when both child and parent class when they have data member with same name.

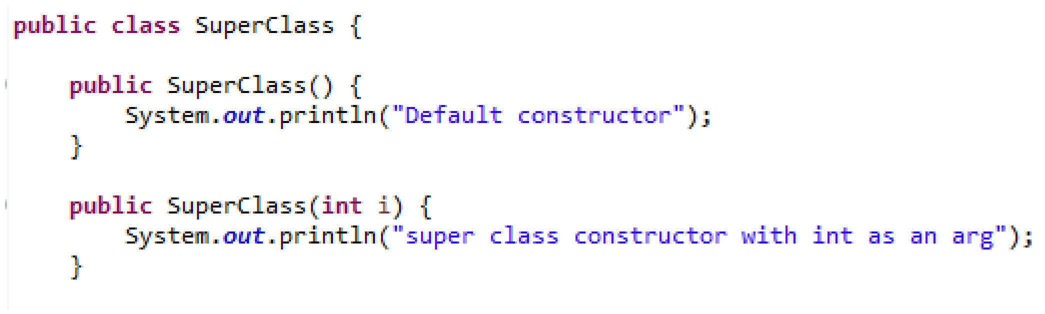


```java
package com.dev.inheritance;

public class Father extends GrandFather {

    static Father f = new Father();
     String name = "Venatachala";

    @Override
    public void printName() {
         System.out.println(name + " "+f.name+" "+f.lastName);
         System.out.println(name + " "+super.name+" "+f.lastName); //access immediate super
    }

    public static void main(String[] args) {


        f.printName();
    }

}
```

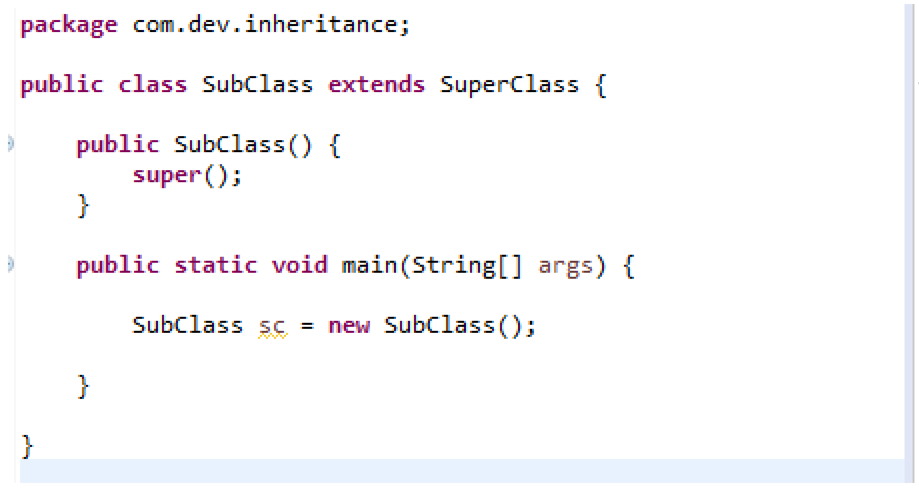
```
<terminated> Father [Java Application] C:\
Venatachala Venatachala Shastry
Venatachala Venkatarama Shastry
```

SUPER()

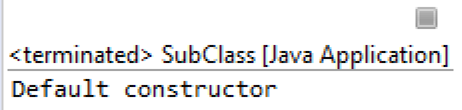
Calling super class constructor in sub class.



```java
public class SuperClass {

    public SuperClass() {
        System.out.println("Default constructor");
    }

    public SuperClass(int i) {
        System.out.println("super class constructor with int as an arg");
    }
```

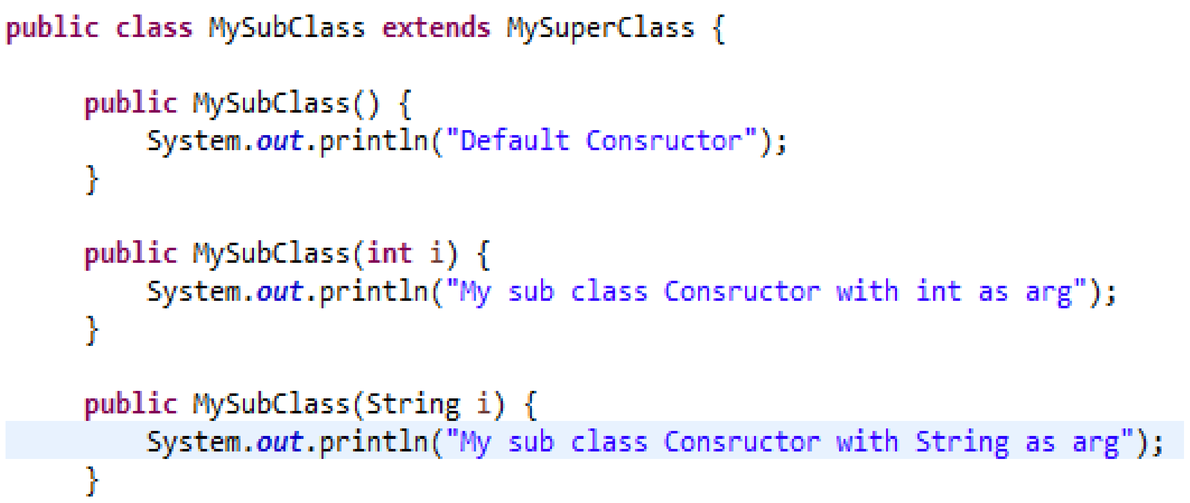


```java
package com.dev.inheritance;

public class SubClass extends SuperClass {

    public SubClass() {
        super();
    }

    public static void main(String[] args) {

        SubClass sc = new SubClass();

    }

}
```

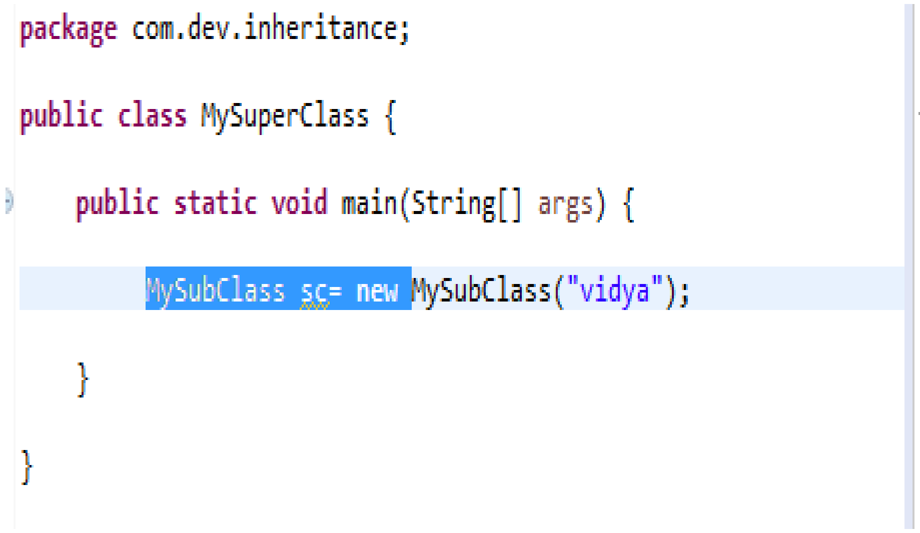


```
<terminated> SubClass [Java Application]
Default constructor
```

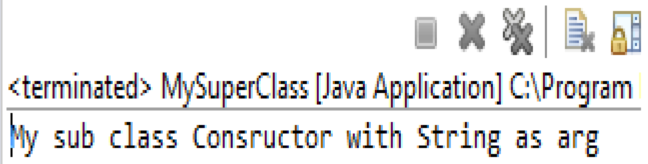
Calling sub class constructor in super class.



```
public class MySubClass extends MySuperClass {

    public MySubClass() {
        System.out.println("Default Consructor");
    }

    public MySubClass(int i) {
        System.out.println("My sub class Consructor with int as arg");
    }

    public MySubClass(String i) {
        System.out.println("My sub class Consructor with String as arg");
    }
```



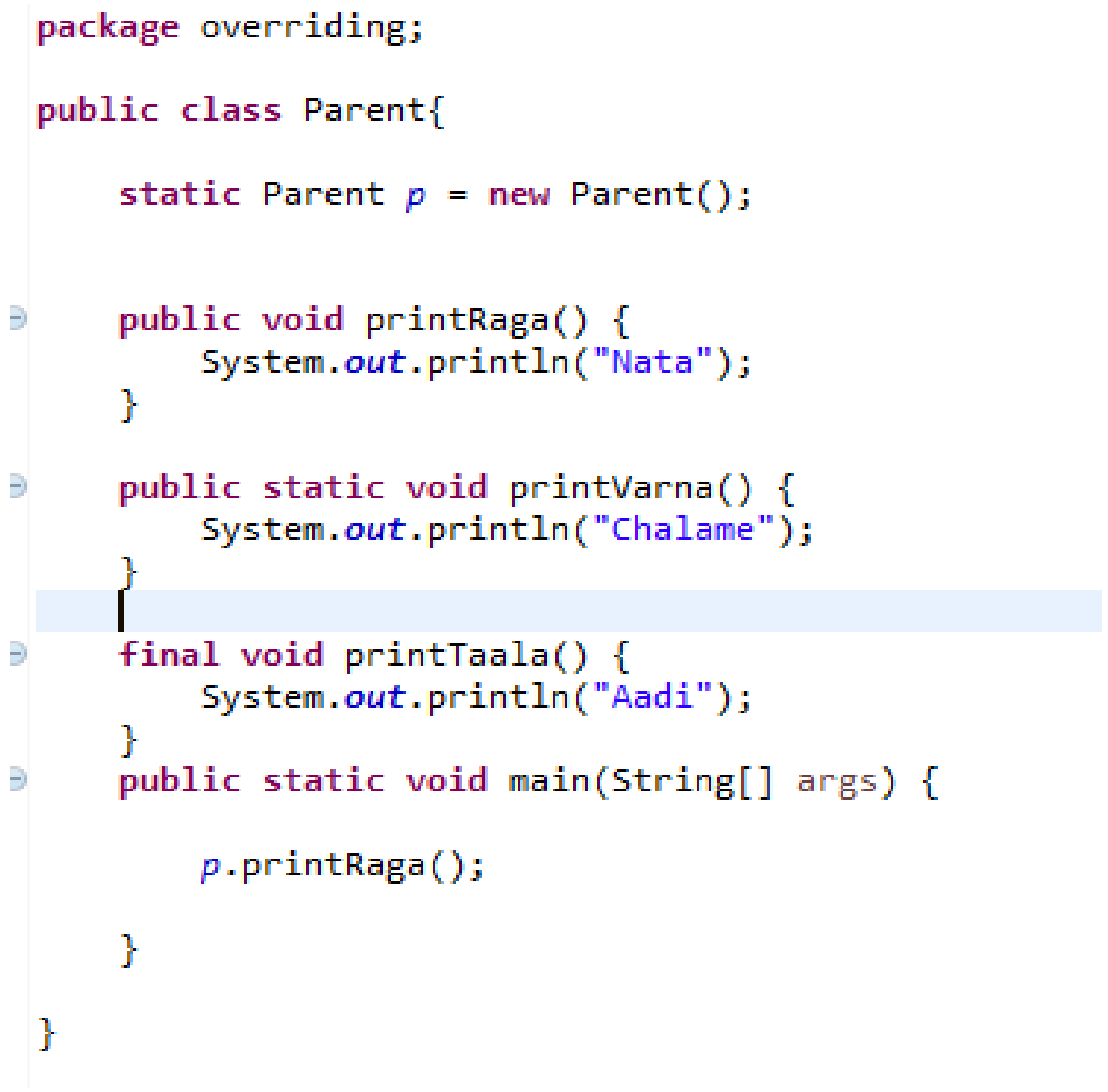
```
package com.dev.inheritance;

public class MySuperClass {

    public static void main(String[] args) {

        MySubClass sc= new MySubClass("vidya");

    }

}
```



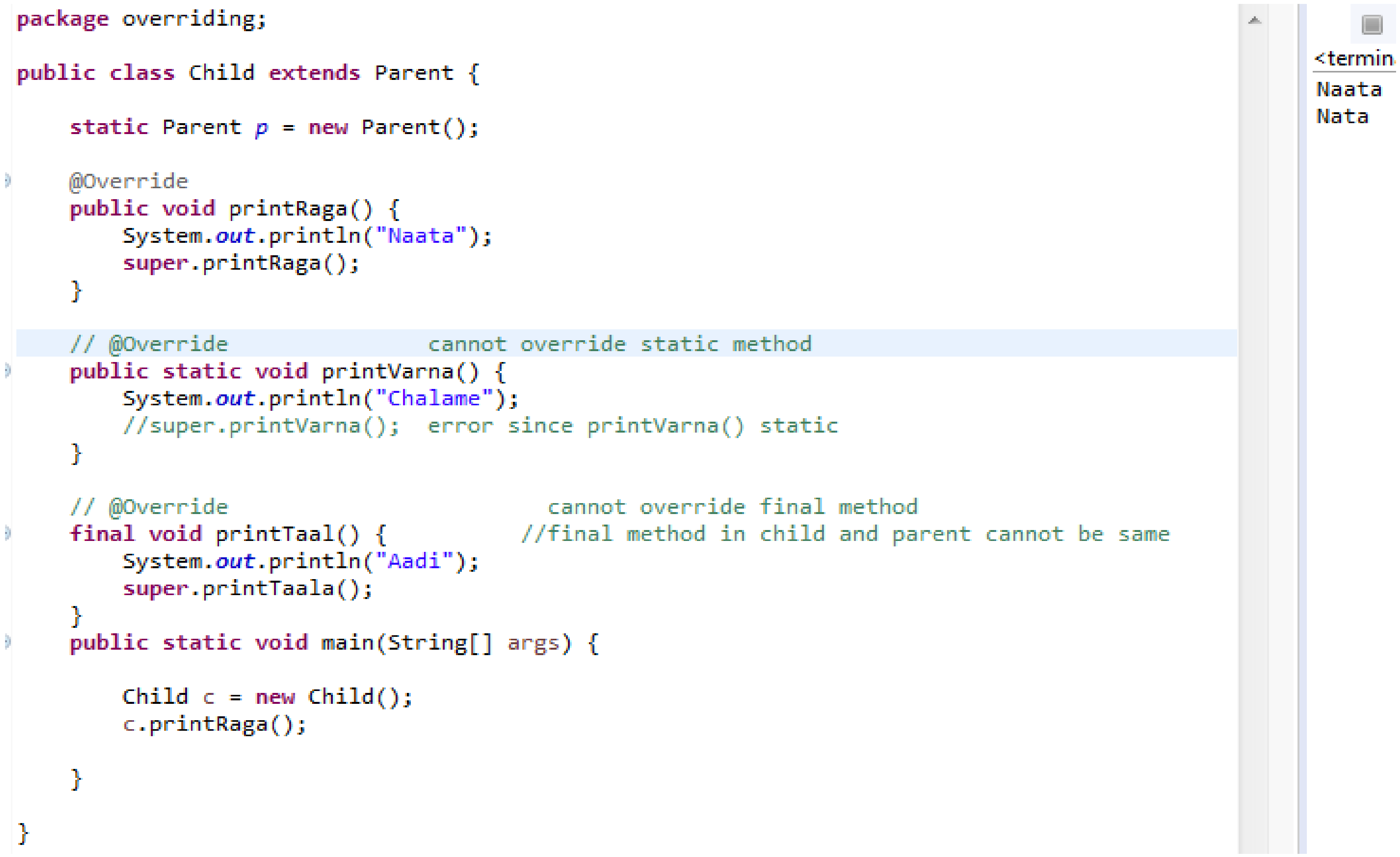
```
<terminated> MySuperClass [Java Application] C:\Program
My sub class Consructor with String as arg
```

METHOD OVERRIDING

- ❖ Declaring a method in child class which has already been declared in parent class is known as method overriding.
- ❖ Method overriding is done to providing implementation specific to a child class.
- ❖ Overidden method will be written in parent and overriding will be written in child.
- ❖ Advantage is that one can provide implementation to the child class method without changing the code present in parent class.
- ❖ Private methods cannot be overridden.
- ❖ Final methods can be inherited but cannot be changed



```java
package overriding;

public class Parent{

    static Parent p = new Parent();


    public void printRaga() {
        System.out.println("Nata");
    }

    public static void printVarna() {
        System.out.println("Chalame");
    }

    final void printTaala() {
        System.out.println("Aadi");
    }
    public static void main(String[] args) {

        p.printRaga();

    }

}
```



```java
package overriding;

public class Child extends Parent {

    static Parent p = new Parent();

    @Override
    public void printRaga() {
        System.out.println("Naata");
        super.printRaga();
    }

    // @Override               cannot override static method
    public static void printVarna() {
        System.out.println("Chalame");
        //super.printVarna();  error since printVarna() static
    }

    // @Override                        cannot override final method
    final void printTaal() {          //final method in child and parent cannot be same
        System.out.println("Aadi");
        super.printTaala();
    }
    public static void main(String[] args) {

        Child c = new Child();
        c.printRaga();

    }

}
```

```
<termin
Naata
Nata
```

METHOD OVERLOADING

It is a feature in JAVA that allows us to have same methods(same name) in a single class more than once provided the argument lists differ:

- Number of parameters
- Order of parameters
- Data-type of parameters

Return type of methods having same name does affect even when return types are different but not the argument.

Static method does not affect method overloading.

POLYMORPHISM

- One method acting differently for different classes.
- Method overriding is a type of polymorphism.
- Method overriding is an example of run-time polymorphism since called at the time of execution
- At the time of execution method declaration is connected with that particular method definition. Since this happens during run time overriding is called **"Run-time polymorphism"/"late binding".**
- In method overloading, at the time of compilation they act according to number of arguments passed and based on this specific method declaration will be bind. Since this happens during compilation its called **"Compile –time polymorphism"/"Early binding".**

ABSTRACTION

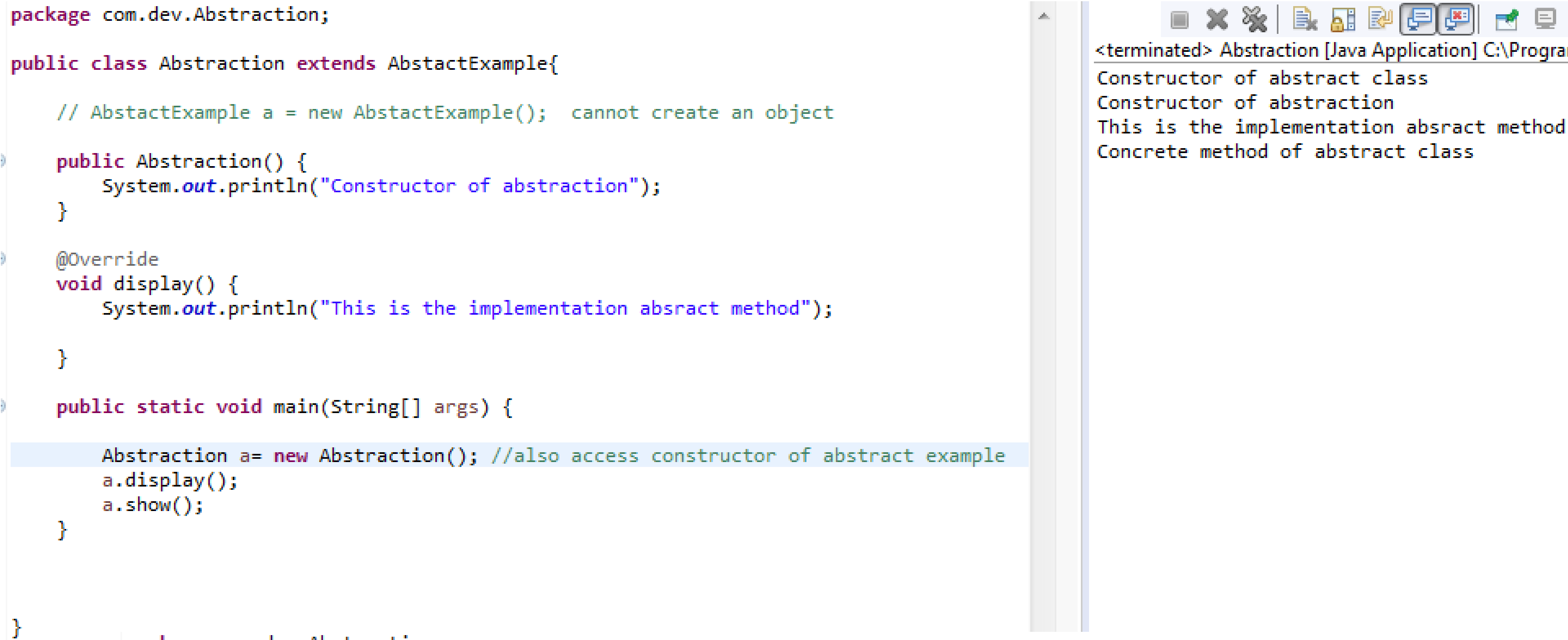
- Only concerned with invoking method or functionalities but **not with implementation.**

  Ex: println()
- Any invoking method can be called as abstraction.
- Hiding implementation and providing only functionalities .
- Advantage : Application built by one can only send .class file to client rather than giving code since he can change the code as a result there might be crash in application due to change in code.

ABSTRACT CLASS

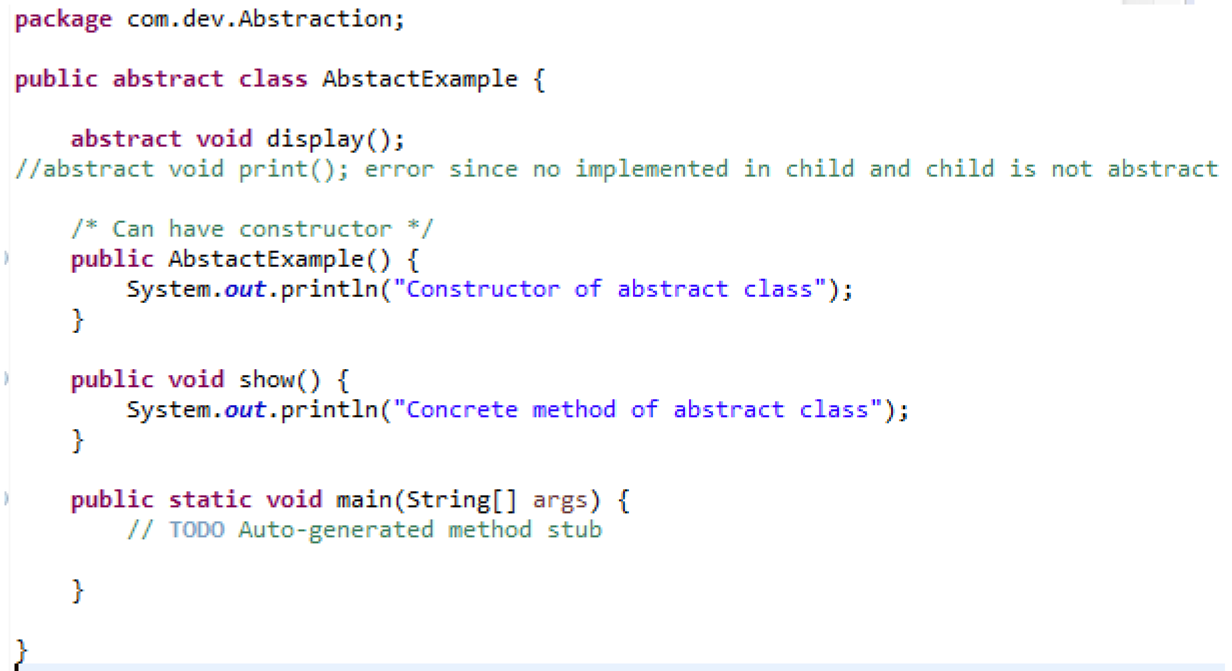
- Any method declared with keyword abstract is known as abstract class.
- It does not have a body.
- It can have both abstract methods as well as concrete methods.
- Any class having an **abstract method** should be declared as **abstract.**
- If a class is declared as abstract it is not mandatory to have abstract method.
- Objects cannot be created for abstract classes.
- They can have constructors.
- One should override each and every method in child from abstract parent class. To avoid this make child class an abstract.

- If concrete class is extending abstract class then it should override all the abstract methods declared in that abstract class.



```
package com.dev.Abstraction;

public class Abstraction extends AbstactExample{

    // AbstactExample a = new AbstactExample();  cannot create an object

    public Abstraction() {
        System.out.println("Constructor of abstraction");
    }

    @Override
    void display() {
        System.out.println("This is the implementation absract method");

    }

    public static void main(String[] args) {

        Abstraction a= new Abstraction(); //also access constructor of abstract example
        a.display();
        a.show();
    }



}
```

```
<terminated> Abstraction [Java Application] C:\Progra
Constructor of abstract class
Constructor of abstraction
This is the implementation absract method
Concrete method of abstract class
```



```
package com.dev.Abstraction;

public abstract class AbstactExample {

    abstract void display();
//abstract void print(); error since no implemented in child and child is not abstract

    /* Can have constructor */
    public AbstactExample() {
        System.out.println("Constructor of abstract class");
    }

    public void show() {
        System.out.println("Concrete method of abstract class");
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

INTERFACE

- When a single class wants to inherit the properties from two different classes its not possible in java so overcome this problem a concept called interface was introduced. Achieved using keyword implements.
- Interface is just like a class where we declare methods.
- By default , methods declared in interface will be abstract methods & public.
- Interfaces **cannot** have **constructors**.
- Concrete methods can be created in interface as **default or static.** If declared default then call by object if not then by interface name.
- Variables can be created in interface as default, static and final. They cannot be declared but if declared they should be initialized.
- One interface cannot implement another interface but a class can extend any number of interfaces at a time. But one class cannot extend two classes at a time.

TYPES OF INTERFACE

- If an interface is **Functional Interface** then that interface can have only one abstract method but can have any number of concrete methods. Functional interface is recognized by using @FuncionalInterface.
- If an interface does not contain any methods then it is called **Marker Interface.**Used to provide special behaviour to class , which can be achieved using Marker interfaces. Serializable, Remote, cloneable are some of the marker interfaces. Clone is a method in object to provide special behaviour to the class.
- If an interface has n number of methods then it is called Typical Interface .

Abstract	Interface
Keyword - abstract	Keyword – interface
Methods declared with abstract keyword	Methods declared will be by default abstract and public
Constructors can be created	Cannot be created
Abstract classes are extended	Interfaces are implemented
To achieve Multi level inheritance	To extract properties from two different classes i.e multiple inheritance
Concrete methods can be in any visibility	Concrete methods should be either static or default
A class can extend only one class.	A class can implement more than one interface

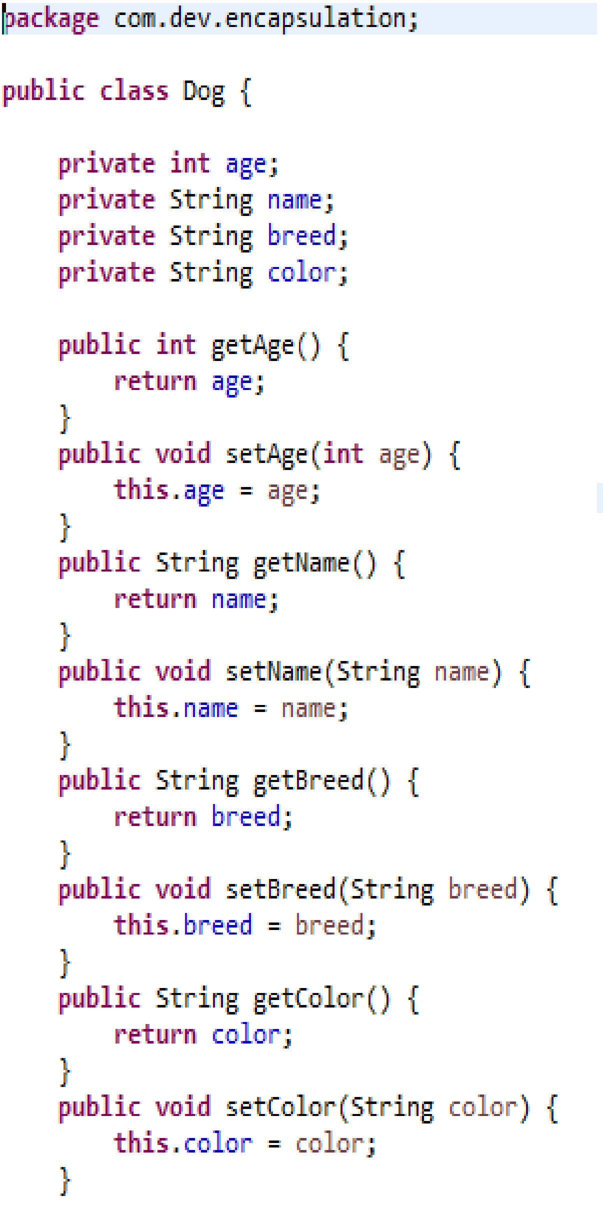
- All the classes and methods present in java.lang will be imported automatically. They need not to be imported explicitly.

Ex. Cloneable is an interface present in java.lang it need not be imported explicitly

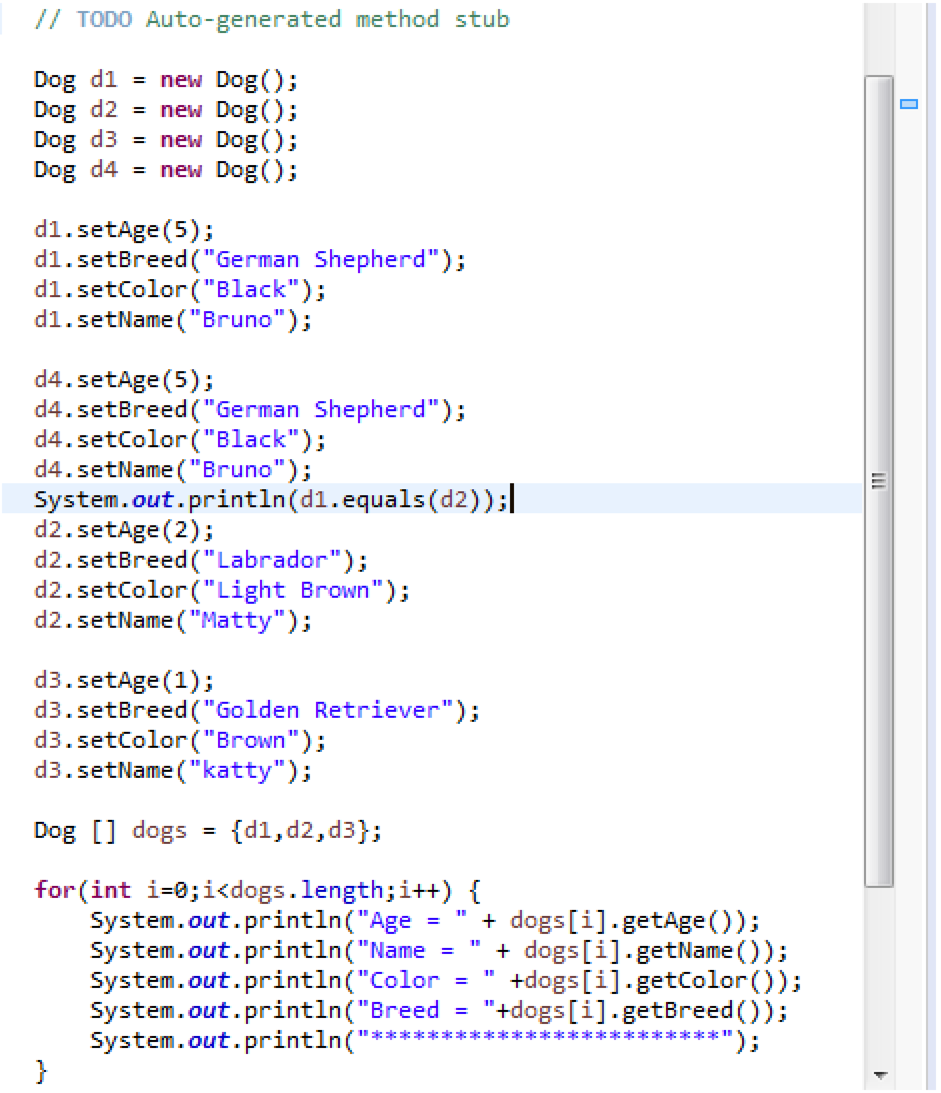
- But Serializable is an interface present in java.io. Which need to be imported explicitly

ENCAPSULATION

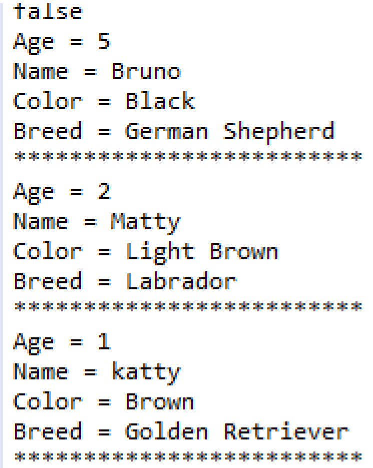
- Enacapsulation is a mechanism with which we wrap up the data members and function methods
- It helps to hide the data.
- Using this one can make data read only (only getters) or write only(only setters).
- Also called data hiding.



```
package com.dev.encapsulation;

public class Dog {

    private int age;
    private String name;
    private String breed;
    private String color;

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getBreed() {
        return breed;
    }
    public void setBreed(String breed) {
        this.breed = breed;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
```



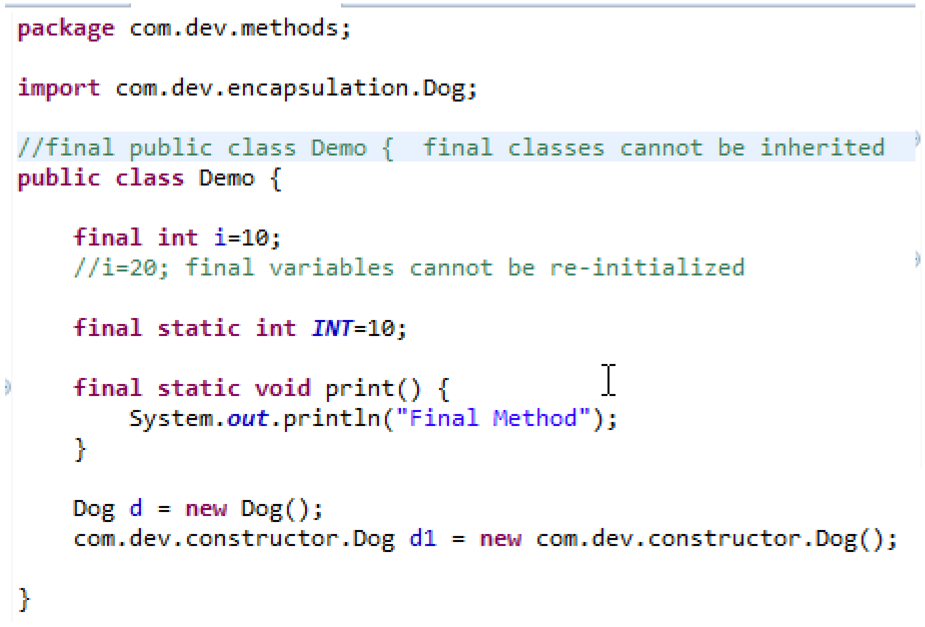
```
// TODO Auto-generated method stub

Dog d1 = new Dog();
Dog d2 = new Dog();
Dog d3 = new Dog();
Dog d4 = new Dog();

d1.setAge(5);
d1.setBreed("German Shepherd");
d1.setColor("Black");
d1.setName("Bruno");

d4.setAge(5);
d4.setBreed("German Shepherd");
d4.setColor("Black");
d4.setName("Bruno");
System.out.println(d1.equals(d2));
d2.setAge(2);
d2.setBreed("Labrador");
d2.setColor("Light Brown");
d2.setName("Matty");

d3.setAge(1);
d3.setBreed("Golden Retriever");
d3.setColor("Brown");
d3.setName("katty");

Dog [] dogs = {d1,d2,d3};

for(int i=0;i<dogs.length;i++) {
    System.out.println("Age = " + dogs[i].getAge());
    System.out.println("Name = " + dogs[i].getName());
    System.out.println("Color = " +dogs[i].getColor());
    System.out.println("Breed = "+dogs[i].getBreed());
    System.out.println("*************************");
}
```



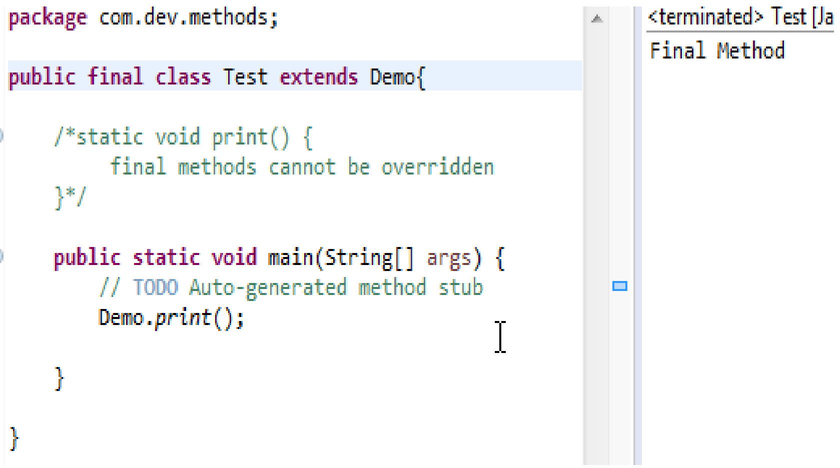
```
false
Age = 5
Name = Bruno
Color = Black
Breed = German Shepherd
*************************
Age = 2
Name = Matty
Color = Light Brown
Breed = Labrador
*************************
Age = 1
Name = katty
Color = Brown
Breed = Golden Retriever
*************************
```

FINAL

- ❖ Variable initialized as final cannot change its value inside its scope.
- ❖ Final methods cannot be overridden.
- ❖ Super class cannot be final class but sub class can be final



```java
package com.dev.methods;

import com.dev.encapsulation.Dog;

//final public class Demo {  final classes cannot be inherited
public class Demo {

    final int i=10;
    //i=20; final variables cannot be re-initialized

    final static int INT=10;

    final static void print() {
        System.out.println("Final Method");
    }

    Dog d = new Dog();
    com.dev.constructor.Dog d1 = new com.dev.constructor.Dog();

}
```



```java
package com.dev.methods;

public final class Test extends Demo{

    /*static void print() {
        final methods cannot be overridden
    }*/

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Demo.print();

    }

}
```

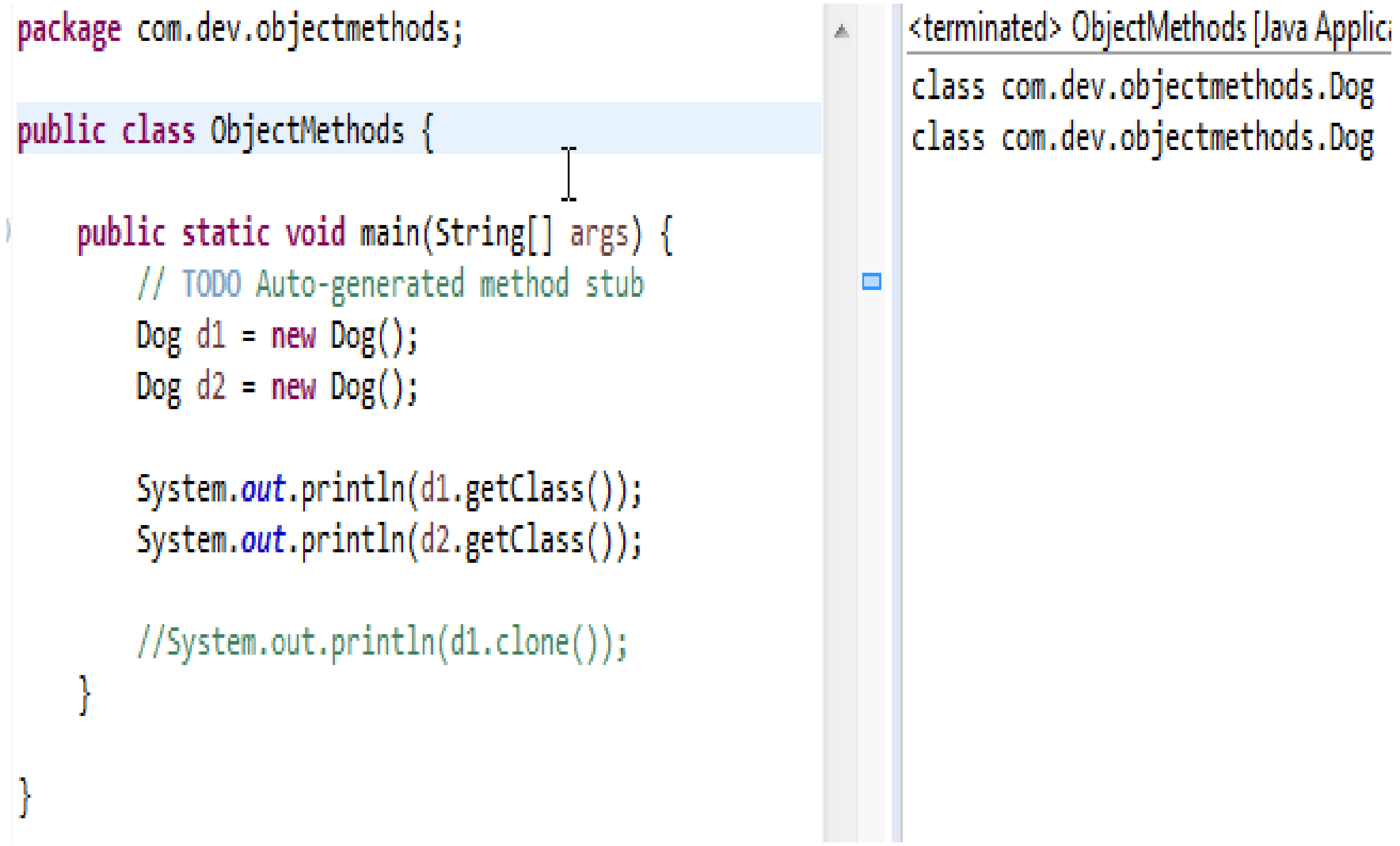
```
<terminated> Test [Ja
Final Method
```

OBJECT

- In java each and every class directly or indirectly inherits the properties of object class i,e object class is the super most class in java.
- Each and every class either a pre-defined class or a user defined class is a child class of object.
- Object is super most class among all the classes present in java programming java.

getClass() :

Returns name of the class it belongs to.



```java
package com.dev.objectmethods;

public class ObjectMethods {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Dog d1 = new Dog();
        Dog d2 = new Dog();

        System.out.println(d1.getClass());
        System.out.println(d2.getClass());

        //System.out.println(d1.clone());
    }

}
```

```
<terminated> ObjectMethods [Java Applic
class com.dev.objectmethods.Dog
class com.dev.objectmethods.Dog
```

equals():

Should be used along with hashCode() otherwise boolean value will be false even though they are equal. Need to override to use equals to compare two objects.To do this go to source and select generate equals() and hashCode()

Methods of object class:

Dog d1 = **new Dog();**

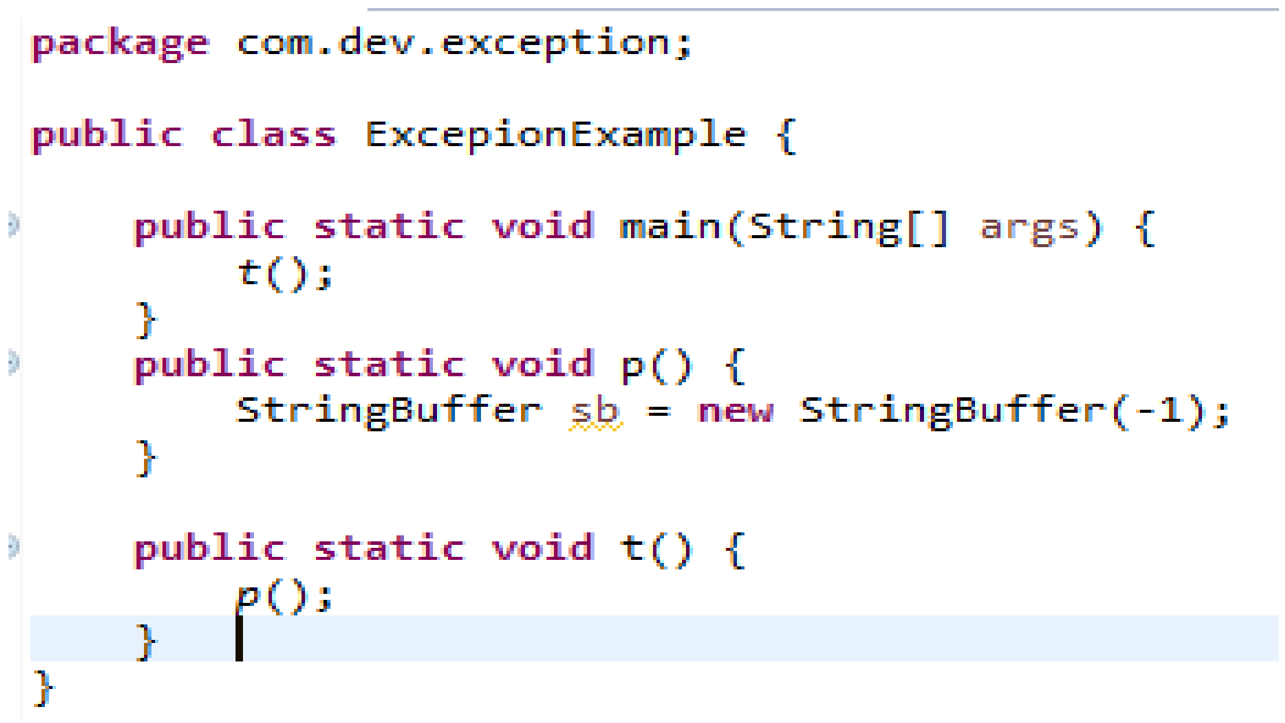
Object Class	Description
clone()	To use this the class should extend cloneable method.
equals()	Should be used along with hashCode() otherwise boolean value will be false even though they are equal. Need to override to use equals to compare two objects.To do this go o source and select generate equals() and hashCode() d1.equals(d4) // true
toString()	Override
Wait()	

EXCEPTIONS

- Exception is an error event which happens during the execution of a program. It disrupts the normal flow of program.
- Database server down, hardware failure, network connection failure

Exception Handling

- Overcoming problems during run-time or exceptions.
- Java being OOP language wherever an error occurs while executing a statement, creates an exception object and then the normal flow of program halts and JRE tries to find someone who can handle the raised exception
- Exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, type of exception etc



```java
package com.dev.exception;

public class ExcepionExample {

	public static void main(String[] args) {
		t();
	}
	public static void p() {
		StringBuffer sb = new StringBuffer(-1);
	}

	public static void t() {
		p();
	}
}
```

```
Exception in thread "main" java.lang.NegativeArraySizeException
at java.lang.AbstractStringBuilder.<init>(AbstractStringBuilder.java:68)
at java.lang.StringBuffer.<init>(StringBuffer.java:128)
at com.dev.exception.ExcepionExample.p(ExcepionExample.java:21)
at com.dev.exception.ExcepionExample.t(ExcepionExample.java:25)
at com.dev.exception.ExcepionExample.main(ExcepionExample.java:14)
```

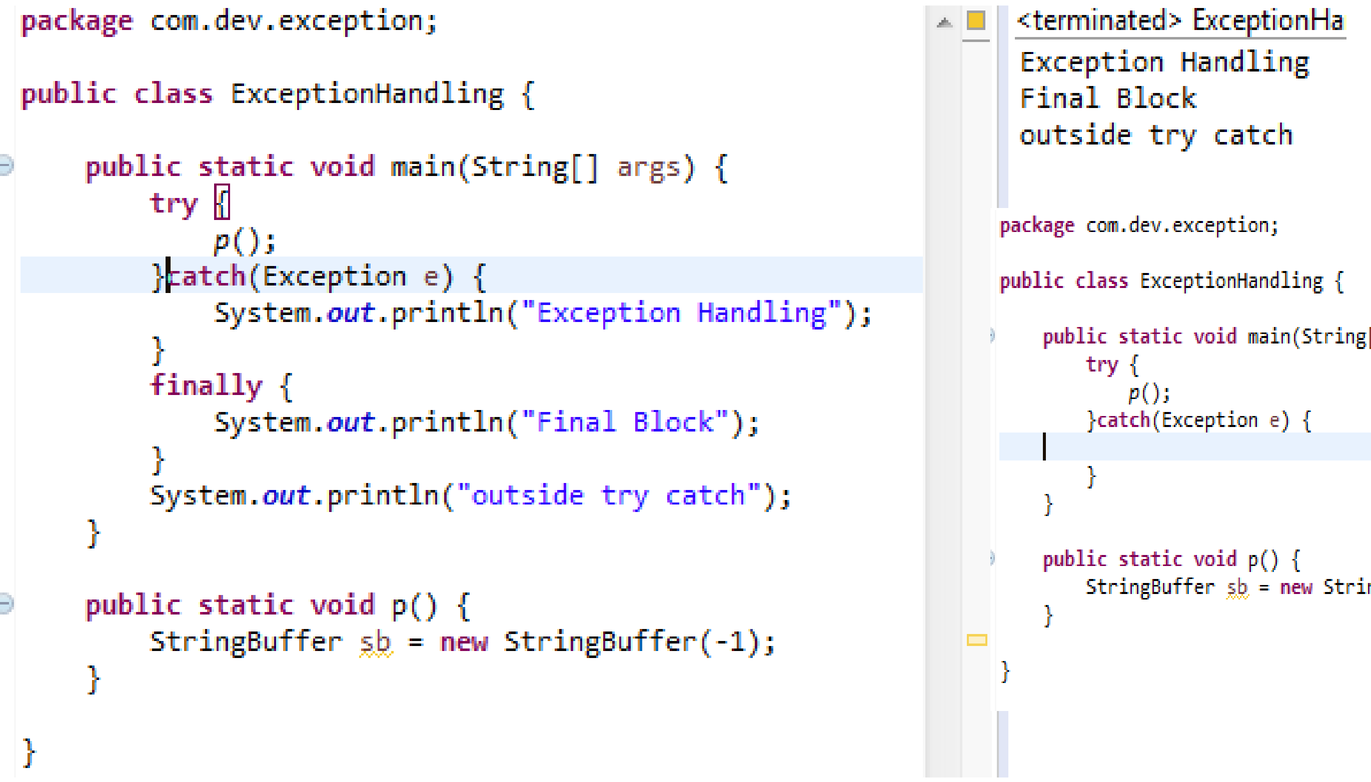
$\left.\begin{array}{l} \text{p} \\ \text{t} \\ \text{main} \end{array}\right\}$ **Method Hierarchy**

- When the exception occurs in a method, the process of creating an exception object and handling it to run-time environment is called throwing an exception.
- Once runtime receives the exception object, it tries to find the handler for the exception. Exception handler is a block of code that can process the exception object.
- Logic :Starting the search in the method where error occurred. If not found then it is passed to calling method and so on.
- If any exception is found, exception object is passed to handler to process it. The handler is said o be exception handler.
- Exception cannot solve compile time errors.
- Differentiate exceptions and errors:

  Errors are solved using programming skills, errors occur during compile time

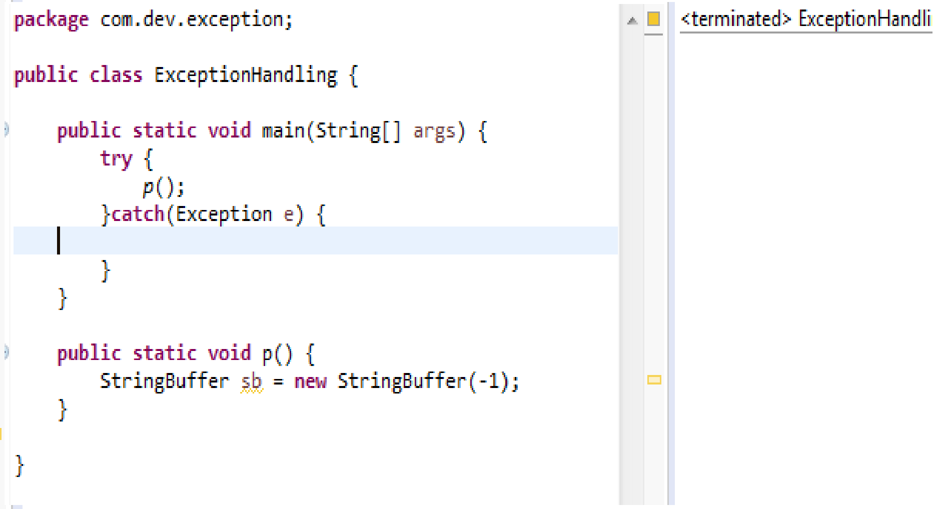
  Exceptions are solved using exception handler, occurs during run-time

try-catch – used for exception handling our code. Try can have multiple catch bloc and we can have nested try-catch. Catch block should take parameter that should be of type exception. If exception parameter need to be included in catch,then should be written in last catch block .[Try can have arguments.(resources from JDBC)]



```
package com.dev.exception;

public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            p();
        }catch(Exception e) {
            System.out.println("Exception Handling");
        }
        finally {
            System.out.println("Final Block");
        }
        System.out.println("outside try catch");
    }

    public static void p() {
        StringBuffer sb = new StringBuffer(-1);
    }

}
```

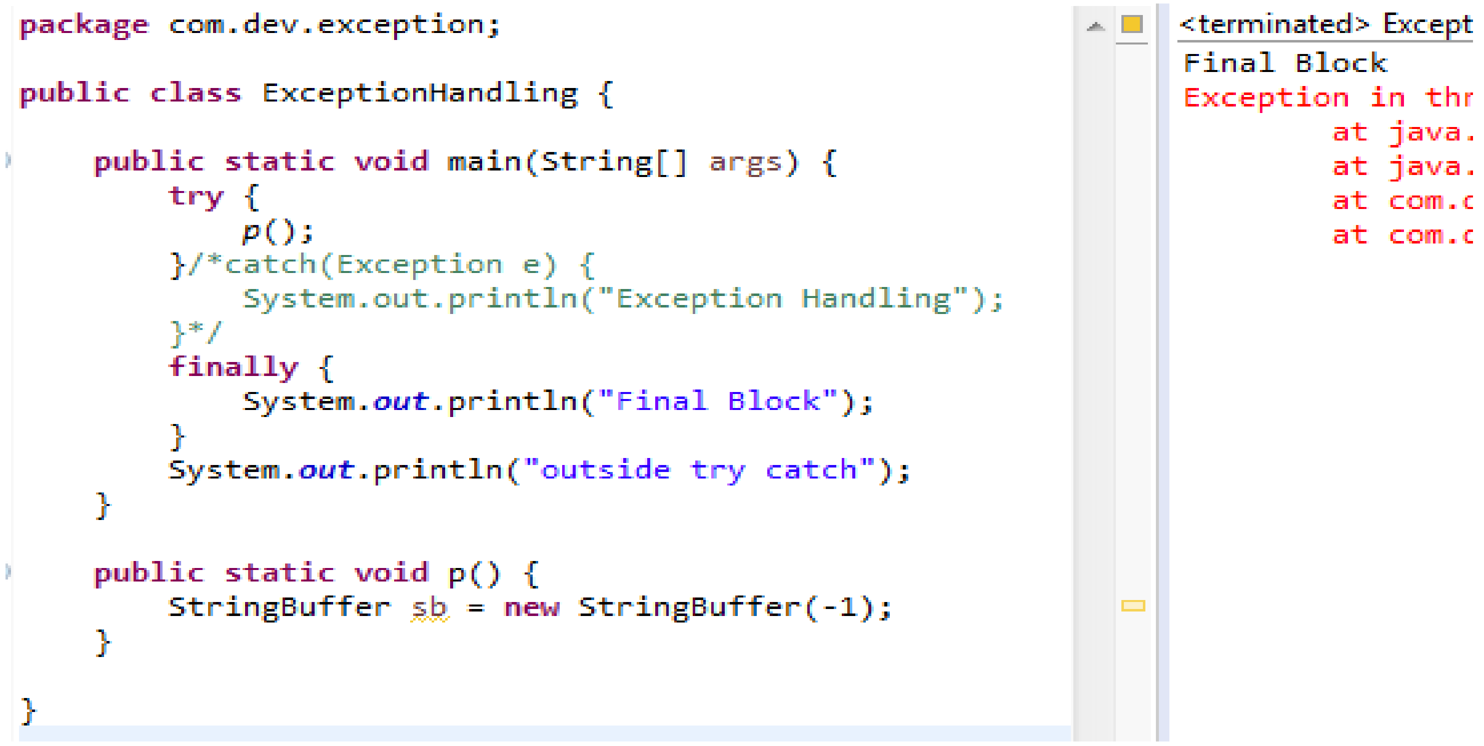
```
<terminated> ExceptionHa
Exception Handling
Final Block
outside try catch
```



```
package com.dev.exception;

public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            p();
        }catch(Exception e) {

        }
    }

    public static void p() {
        StringBuffer sb = new StringBuffer(-1);
    }

}
```

```
<terminated> ExceptionHandli
```

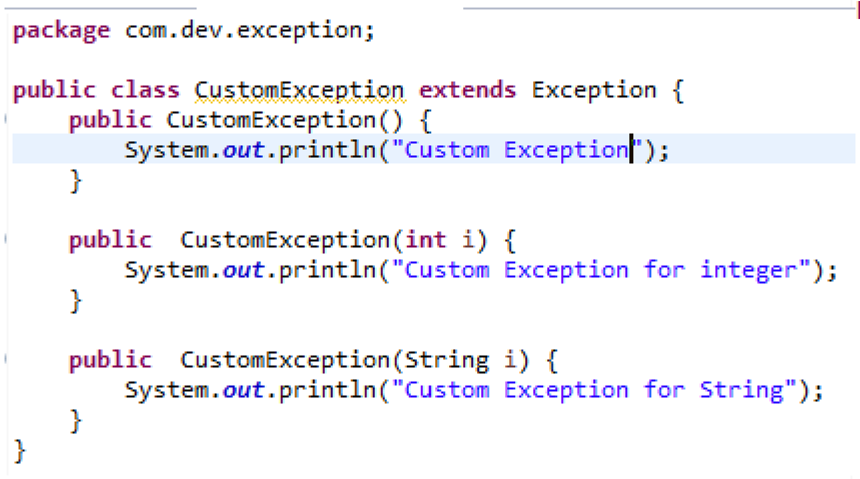
finally – code written here will be executed even if there is try-catch or any exceptions.Finally can have try catch



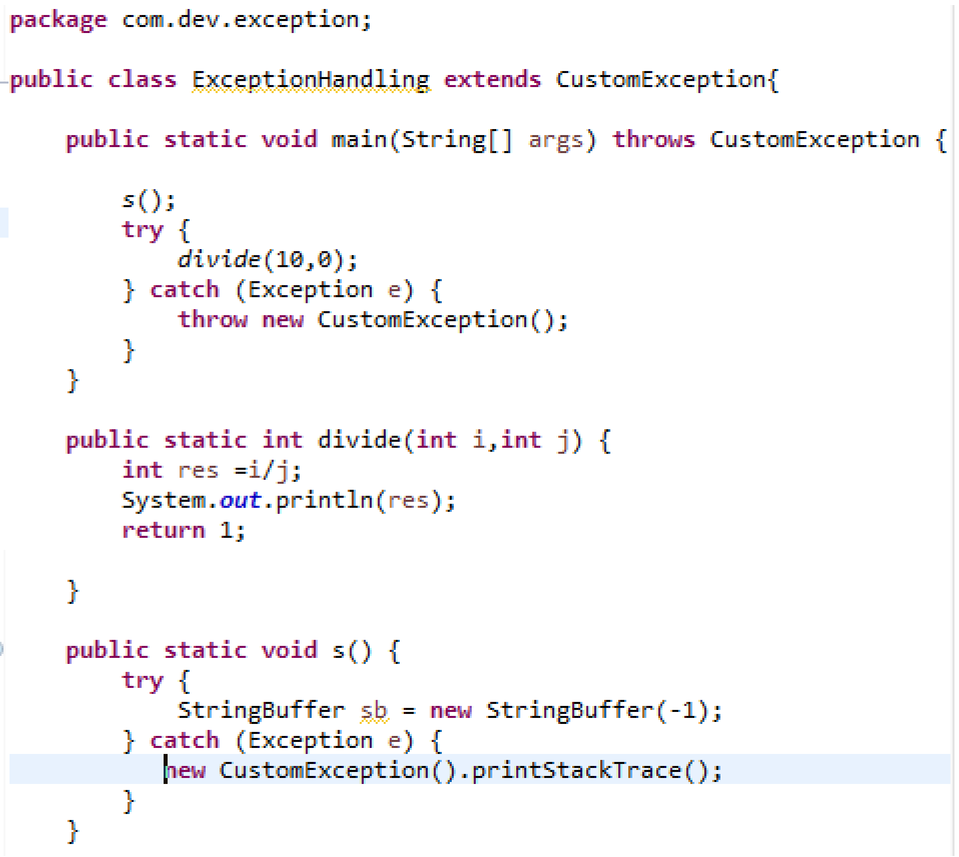
```
package com.dev.exception;

public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            p();
        }/*catch(Exception e) {
            System.out.println("Exception Handling");
        }*/
        finally {
            System.out.println("Final Block");
        }
        System.out.println("outside try catch");
    }

    public static void p() {
        StringBuffer sb = new StringBuffer(-1);
    }

}
```

```
<terminated> Except
Final Block
Exception in thr
        at java.
        at java.
        at com.c
        at com.c
```

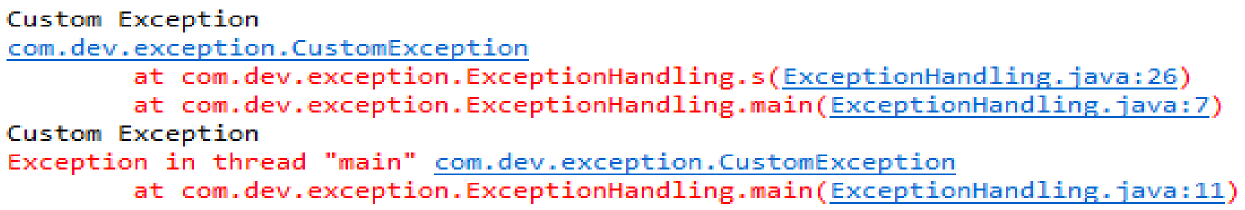
throw – exception defined or thrown by developer



```
package com.dev.exception;

public class CustomException extends Exception {
    public CustomException() {
        System.out.println("Custom Exception");
    }

    public  CustomException(int i) {
        System.out.println("Custom Exception for integer");
    }

    public  CustomException(String i) {
        System.out.println("Custom Exception for String");
    }
}
```

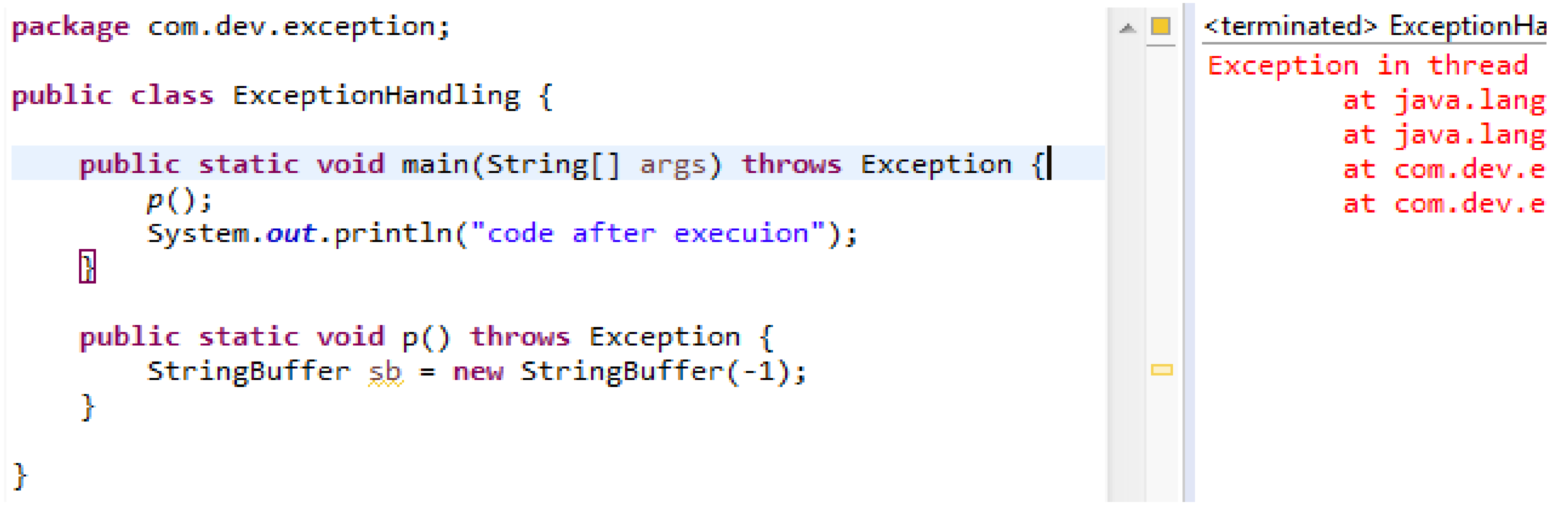


```
package com.dev.exception;

public class ExceptionHandling extends CustomException{

    public static void main(String[] args) throws CustomException {

        s();
        try {
            divide(10,0);
        } catch (Exception e) {
            throw new CustomException();
        }
    }

    public static int divide(int i,int j) {
        int res =i/j;
        System.out.println(res);
        return 1;

    }

    public static void s() {
        try {
            StringBuffer sb = new StringBuffer(-1);
        } catch (Exception e) {
            new CustomException().printStackTrace();
        }
    }
```



```
Custom Exception
com.dev.exception.CustomException
	at com.dev.exception.ExceptionHandling.s(ExceptionHandling.java:26)
	at com.dev.exception.ExceptionHandling.main(ExceptionHandling.java:7)
Custom Exception
Exception in thread "main" com.dev.exception.CustomException
	at com.dev.exception.ExceptionHandling.main(ExceptionHandling.java:11)
```

- **throws** –where exception is pre-defined. It is used when calling method is throwing an exception. It tells that a method might thrown an exception
- When a method is declared as throws then the method which calls this method should also be declared as throws.
- Can provide more than one exception at once.



```
package com.dev.exception;

public class ExceptionHandling {

    public static void main(String[] args) throws Exception {
        p();
        System.out.println("code after execuion");
    }

    public static void p() throws Exception {
        StringBuffer sb = new StringBuffer(-1);
    }

}
```

```
<terminated> ExceptionHa
Exception in thread
        at java.lang
        at java.lang
        at com.dev.e
        at com.dev.e
```

- Java exceptions are hierarchical and inheritance is used to categorize different types of exception.
- **throwable** is a parent class of java exceptions hierarchy and it has two child objects – Error and Exceptions.
- Errors : Errors are exceptional scenarios that are out of scope of application and its not possible to anticipate and recover from them.
- Ex: Hardware failure,JVM crash or out of memory error.

- Exception scenarios which can be anticipated are **checked** Exception.
- Provide errors at compilation time /warnings and errors during run-time.
- Should be caught and provide useful information to user(will be written in catch).
- Exceptions caught during compile-time are check exceptions.
- Try and catch for catch exceptions/try and finally in **unchecked.**

getMessage():

Method of throwable.

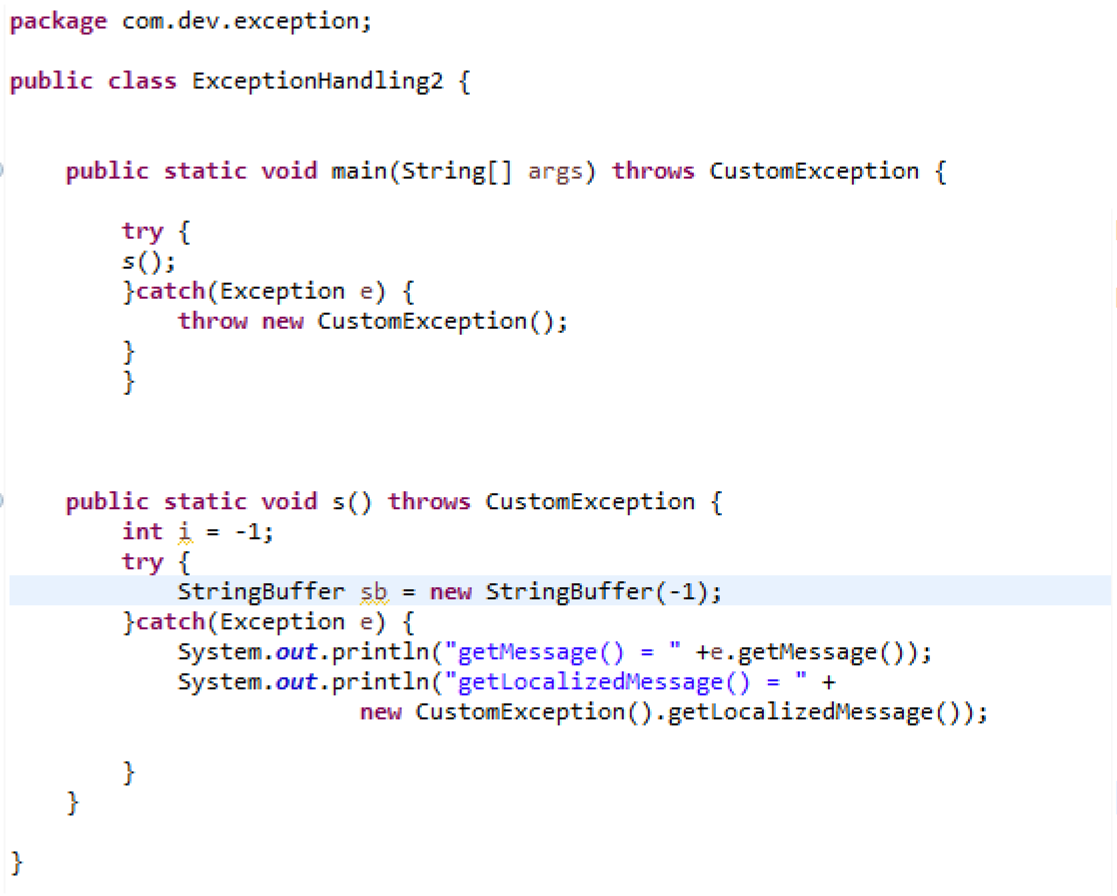
Return type is string.

If not overridden then output will be null. But should not be overridden to get custom message.But should not be used to print custom messages.???

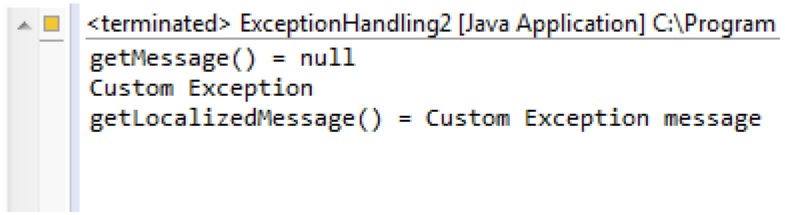
getLocalizedMessage():

Method of thowable. Retuns getMessage();

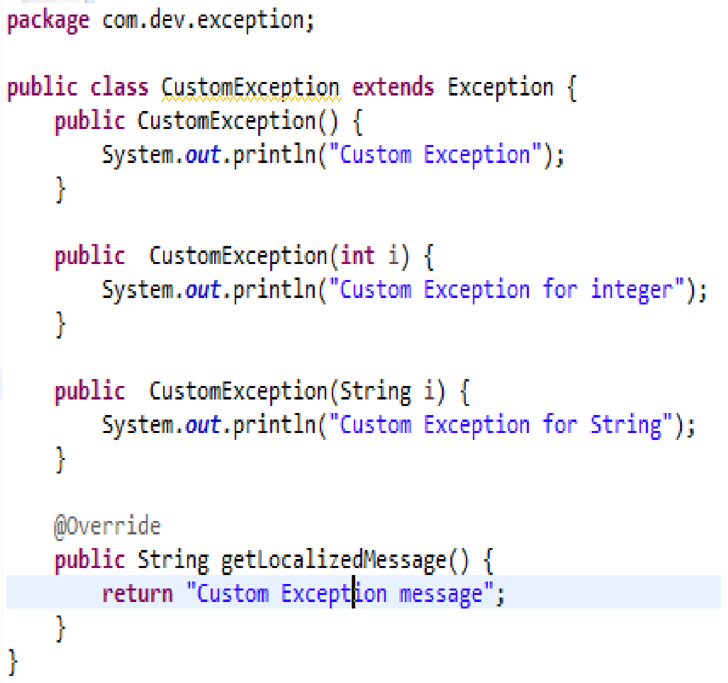
Return type is string. Should be overridden to get custom exception



```java
package com.dev.exception;

public class ExceptionHandling2 {


    public static void main(String[] args) throws CustomException {

        try {
        s();
        }catch(Exception e) {
            throw new CustomException();
        }
        }



    public static void s() throws CustomException {
        int i = -1;
        try {
            StringBuffer sb = new StringBuffer(-1);
        }catch(Exception e) {
            System.out.println("getMessage() = " +e.getMessage());
            System.out.println("getLocalizedMessage() = " +
                         new CustomException().getLocalizedMessage());

        }
    }

}
```



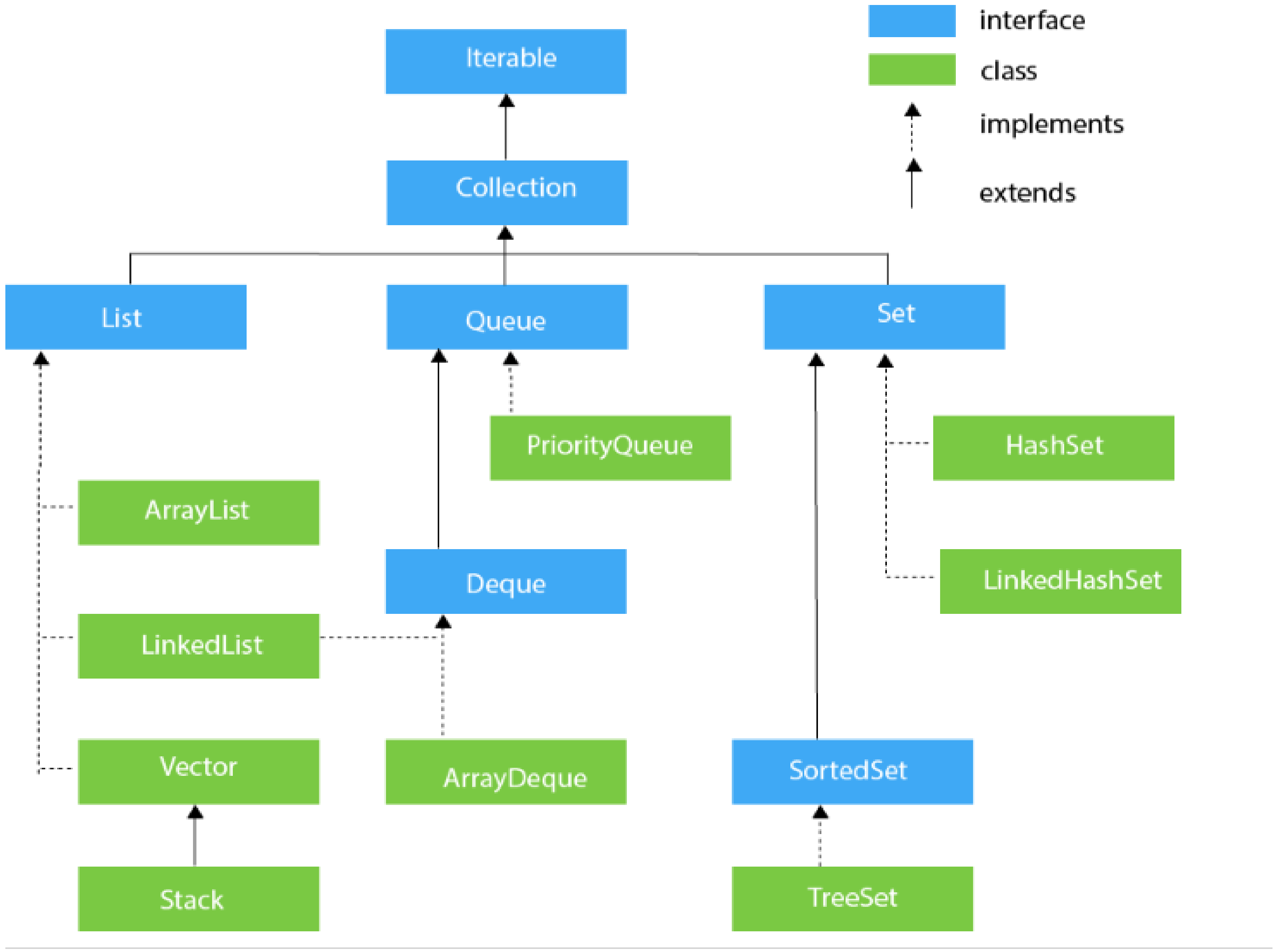
```
<terminated> ExceptionHandling2 [Java Application] C:\Program
getMessage() = null
Custom Exception
getLocalizedMessage() = Custom Exception message
```



```java
package com.dev.exception;

public class CustomException extends Exception {
    public CustomException() {
        System.out.println("Custom Exception");
    }

    public  CustomException(int i) {
        System.out.println("Custom Exception for integer");
    }

    public  CustomException(String i) {
        System.out.println("Custom Exception for String");
    }

    @Override
    public String getLocalizedMessage() {
        return "Custom Exception message";
    }
}
```

COLLECTIONS

- Containers that group multiple items in a single unit. They are used almost in every programming language and when java arrived, it also came up with collection classes. Introduced in java 1.2.
- To iterate and manipulate in a standard way → Collections Framework
- **Interfaces** →Provides abstract data type to represent collections.
- **Root interface** of collection framework → java.util.collection.
- **Methods**→ size(),iterator(),clear(),add(),remove()→these should be implemented.
- **Java interfaces** → list , set , Queue ,Map (interface of collection framework but not from collection) (abstract data type)
- All collection frameworks are present in java.util.
- Unless there is implementation interfaces are not useful.
- Create different types of collections in java program using collection classes.
- **Collection Classes** → arrayList,linkedList,treeMap,hashMap.treeMap



interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Benefits :

- ❑ Reduced Development Effort
- ❑ Increased quality
- ❑ Reusability and interoperability.

- ❖ Java are the foundation of the java collections framework. All the core collection interfaces are **generic**(generic → added for compile time checking if we want a collection to store a string object we can write as <string>.).
- ❖ Syntax: public interface Collections<E>
- ❖ It specifies type of object declared. It helps in reducing run-time errors by type-checking the objects during compile time.
- ❖ To **overcome ClassCastException** generic was introduced.
- ❖ If an unsupported operation is invoked, a collection implementation throws **UnsupportedOperationException**

COLLECTION INTEFACE

- Has methods to tell how many elements are in collections.
- Size→return type→int
- isEmpty→return type →boolean→Return true if collection contains specific element.
- add(E e) →boolean →true when perticular element is added →E is generic type
- Remove(Object o) → boolean
- Iterator() → iterator type object. Introduced in java1.2.

Bulk operation methods which can be applied to whole collection.

- containsAll(Collection<?> e) → boolean
- addAll,removeAll,seAll.clearAll;

SET INTERFACE

- Set is a collection that cannot contain duplicate elements.Used to represent sets,deck of card
- Set implimenation → HashSet,treeSet and LinkedHashSet
- At most one null element

LIST INTERFACE

- An ordered collection
- Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements <tt>e1</tt> and <tt>e2</tt> such that <tt>e1.equals(e2)</tt>, and they typically allow multiple null elements if they allow null elements at all.
- Index helps to access any elements

QUEUE

- A collection designed for holding elements prior to processing. Besides basic {@link java.util.Collection Collection} operations, queues provide additional insertion, extraction, and inspection operations.

Queue Methods

boolean add(E e); → Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

Throws exception like ClassCastException,NullPoinerException.

IllegalStateException

boolean offer(E e); →Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to {@link #add}, which can fail to insert an element only by throwing an exception.

@return {@code true} if the element was added to this queue, else {@code false}

E remove(); → Retrieves and removes the head of this queue. This method differs from {@link #poll poll} only in that it throws an exception if this queue is empty.

E poll(); → Retrieves and removes the head of this queue , or returns {@code null} if this queue is empty.

E element();

Retrieves, but does not remove, the head of this queue. This method differs from {@link #peek peek} only in that it throws an exception if this queue is empty.

E peek();

Retrieves, but does not remove, the head of this queue, or returns {@code null} if this queue is empty.

MAP INTERFACE

- An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
- Map implimentations are HashMap, TreeMap and LinkedHashMap.
- Map methods like put(),remove(),get(),containsKey(),containsValue(),size()

HashSet

HashSet Class→ Basic implementation of the set interface that is backed up by HashMap.It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the <tt>null</tt>element.

Basic operations → add(),remove(),contains(),size()

One can set initial capacity and load factor for this collection.

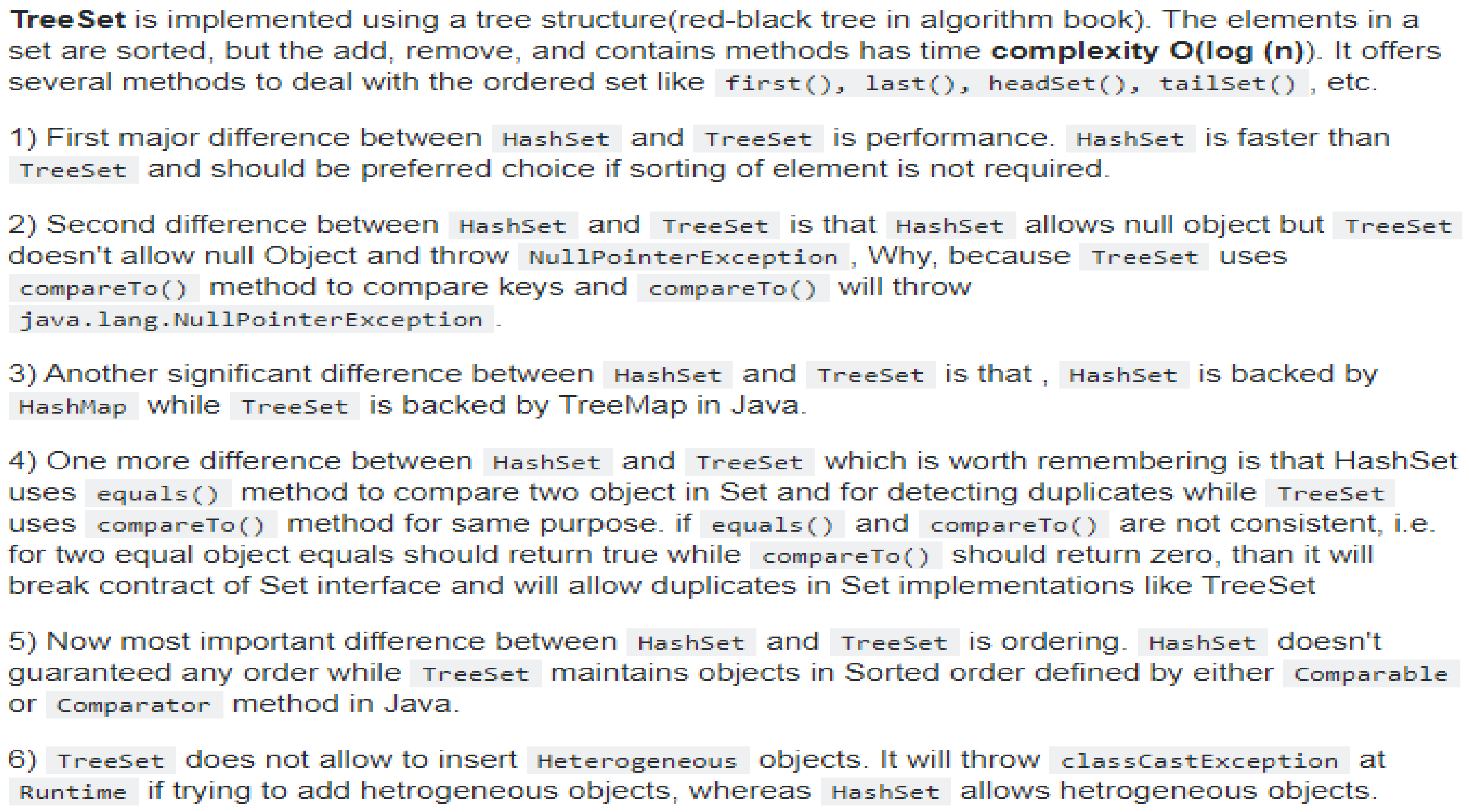
Load factor → Specifies at which level/point set should increase its capacity.

```
public HashSet() {
        map = new HashMap<>();
   }
```
→Constructs a new, empty set; the backing <tt>HashMap</tt> instance has default initial capacity (16) and load factor (0.75).

```
public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
   }
```

TreeSet

HashSet doesn't guaranteed any order while **TreeSet** maintains objects in Sorted order defined by either Comparable or Comparator method in Java.



TreeSet is implemented using a tree structure(red-black tree in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods has time **complexity O(log (n))**. It offers several methods to deal with the ordered set like `first(), last(), headSet(), tailSet()`, etc.

1) First major difference between `HashSet` and `TreeSet` is performance. `HashSet` is faster than `TreeSet` and should be preferred choice if sorting of element is not required.

2) Second difference between `HashSet` and `TreeSet` is that `HashSet` allows null object but `TreeSet` doesn't allow null Object and throw `NullPointerException`, Why, because `TreeSet` uses `compareTo()` method to compare keys and `compareTo()` will throw `java.lang.NullPointerException`.

3) Another significant difference between `HashSet` and `TreeSet` is that , `HashSet` is backed by `HashMap` while `TreeSet` is backed by TreeMap in Java.

4) One more difference between `HashSet` and `TreeSet` which is worth remembering is that HashSet uses `equals()` method to compare two object in Set and for detecting duplicates while `TreeSet` uses `compareTo()` method for same purpose. if `equals()` and `compareTo()` are not consistent, i.e. for two equal object equals should return true while `compareTo()` should return zero, than it will break contract of Set interface and will allow duplicates in Set implementations like TreeSet

5) Now most important difference between `HashSet` and `TreeSet` is ordering. `HashSet` doesn't guaranteed any order while `TreeSet` maintains objects in Sorted order defined by either `Comparable` or `Comparator` method in Java.

6) `TreeSet` does not allow to insert `Heterogeneous` objects. It will throw `classCastException` at `Runtime` if trying to add hetrogeneous objects, whereas `HashSet` allows hetrogeneous objects.

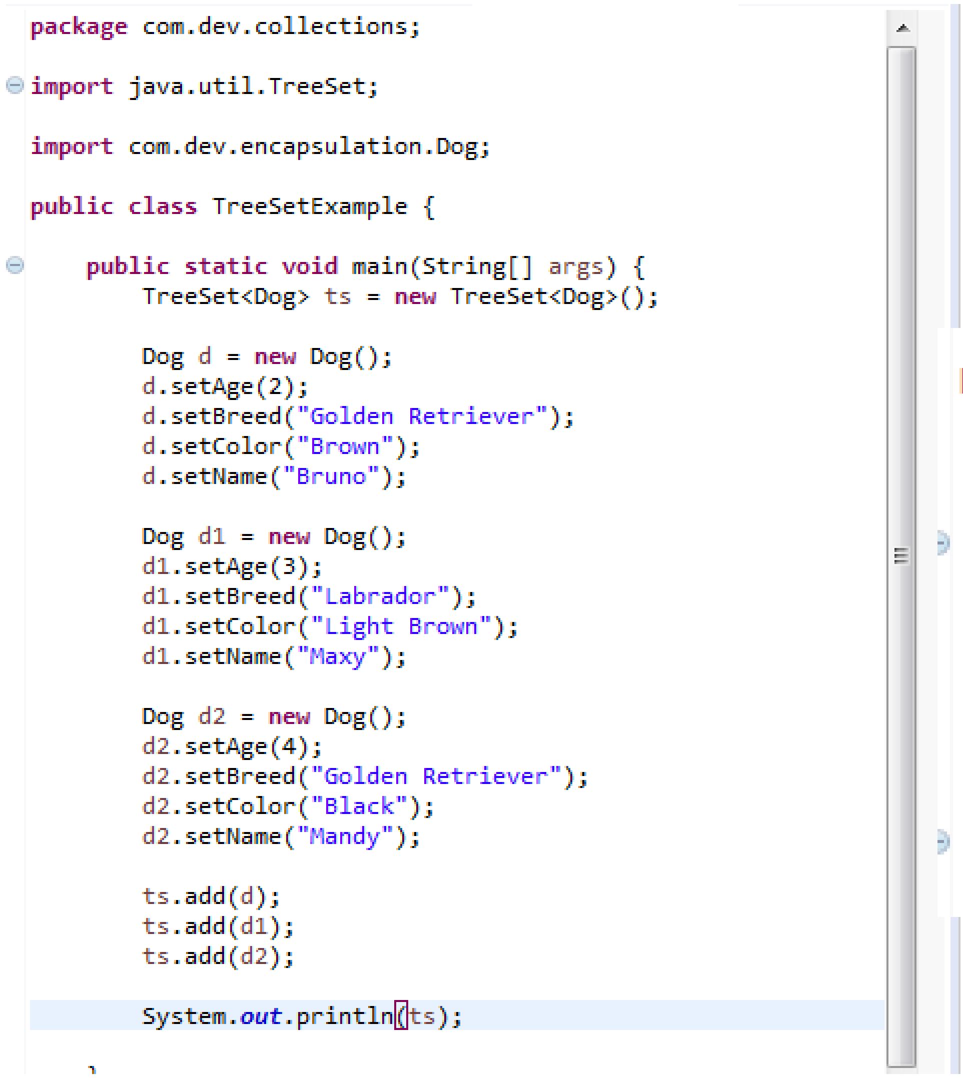
```
 public TreeSet() {
        this(new TreeMap<E,Object>());
```
}It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.

```
public TreeSet(Comparator<? super E> comparator) {
        this(new TreeMap<>(comparator));
    }
```
It is used to construct an empty tree set that will be sorted according to given comparator

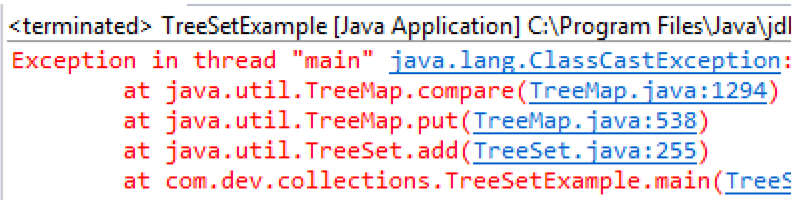
```
public TreeSet(Collection<? extends E> c) {
        this();
        addAll(c);
    }
```
It is used to build a new tree set that contains the elements of the collection c.

```
 public TreeSet(SortedSet<E> s) {
        this(s.comparator());
        addAll(s);
    }
```
It is used to build a TreeSet that contains the elements of the given SortedSet.

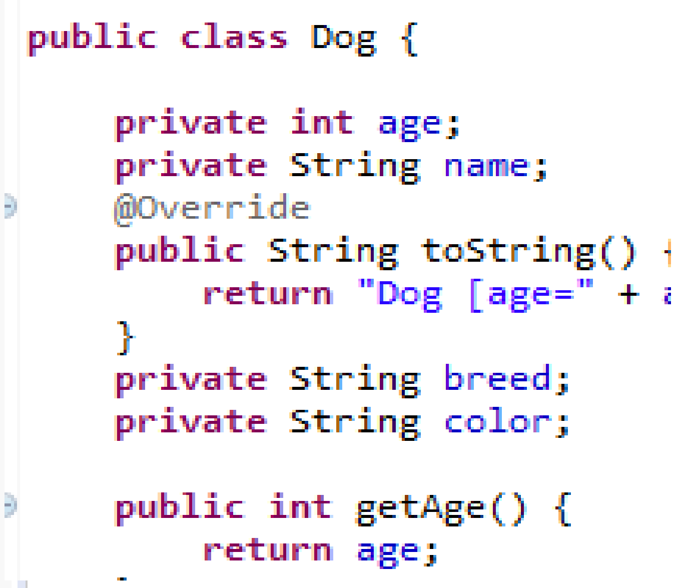
Before implementing Comparable<> in Dog



```
package com.dev.collections;

import java.util.TreeSet;

import com.dev.encapsulation.Dog;

public class TreeSetExample {

    public static void main(String[] args) {
        TreeSet<Dog> ts = new TreeSet<Dog>();

        Dog d = new Dog();
        d.setAge(2);
        d.setBreed("Golden Retriever");
        d.setColor("Brown");
        d.setName("Bruno");

        Dog d1 = new Dog();
        d1.setAge(3);
        d1.setBreed("Labrador");
        d1.setColor("Light Brown");
        d1.setName("Maxy");

        Dog d2 = new Dog();
        d2.setAge(4);
        d2.setBreed("Golden Retriever");
        d2.setColor("Black");
        d2.setName("Mandy");

        ts.add(d);
        ts.add(d1);
        ts.add(d2);

        System.out.println(ts);
```

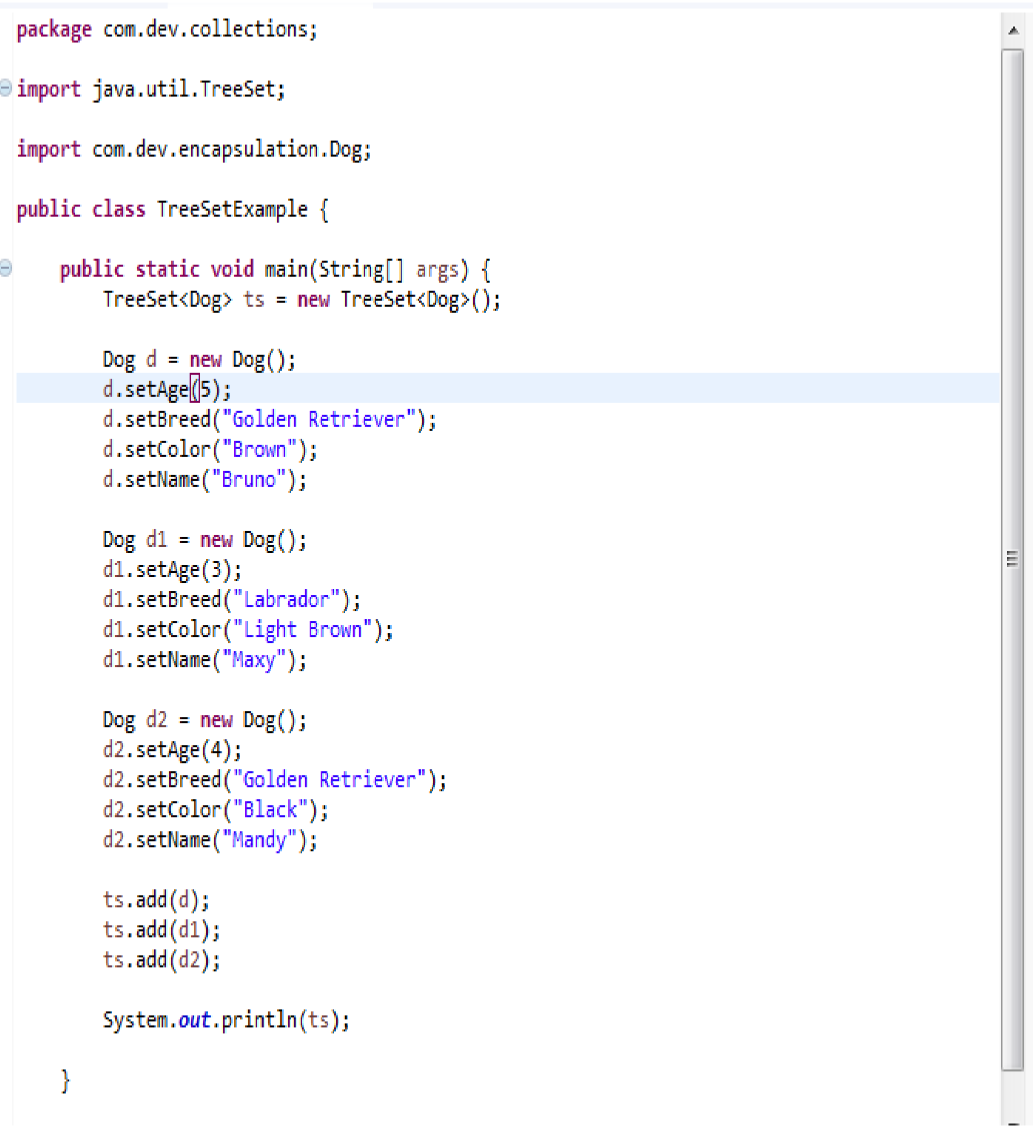


```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jdl
Exception in thread "main" java.lang.ClassCastException:
        at java.util.TreeMap.compare(TreeMap.java:1294)
        at java.util.TreeMap.put(TreeMap.java:538)
        at java.util.TreeSet.add(TreeSet.java:255)
        at com.dev.collections.TreeSetExample.main(TreeS
```

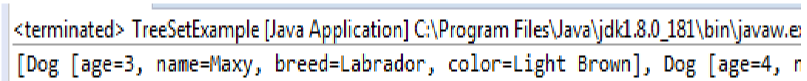


```
public class Dog {

    private int age;
    private String name;
    @Override
    public String toString() {
        return "Dog [age=" + a
    }
    private String breed;
    private String color;

    public int getAge() {
        return age;
```

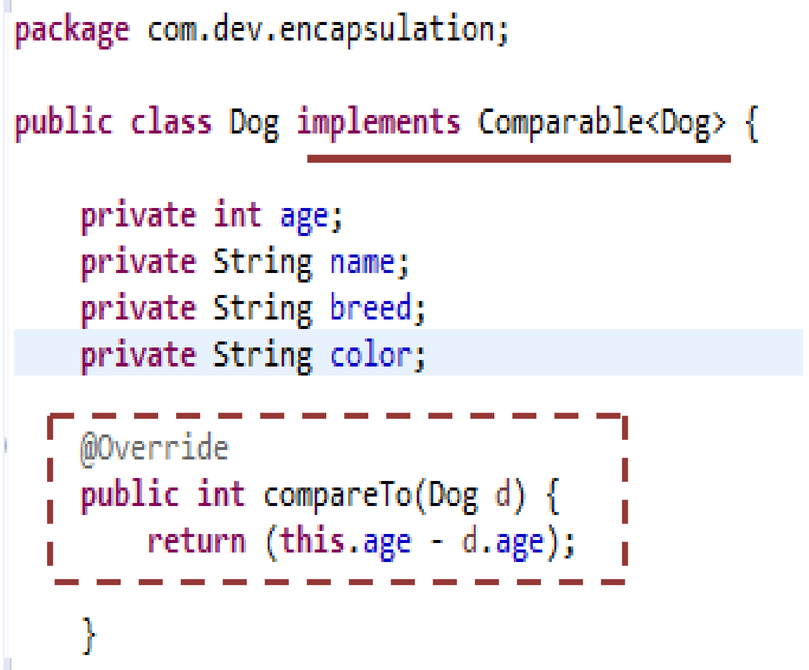
- <u>After implementing Comparable<> in Dog</u>
- Output executed in order to id. Since id is compared in overridden compareTo()



```
package com.dev.collections;

import java.util.TreeSet;

import com.dev.encapsulation.Dog;

public class TreeSetExample {

    public static void main(String[] args) {
        TreeSet<Dog> ts = new TreeSet<Dog>();

        Dog d = new Dog();
        d.setAge(5);
        d.setBreed("Golden Retriever");
        d.setColor("Brown");
        d.setName("Bruno");

        Dog d1 = new Dog();
        d1.setAge(3);
        d1.setBreed("Labrador");
        d1.setColor("Light Brown");
        d1.setName("Maxy");

        Dog d2 = new Dog();
        d2.setAge(4);
        d2.setBreed("Golden Retriever");
        d2.setColor("Black");
        d2.setName("Mandy");

        ts.add(d);
        ts.add(d1);
        ts.add(d2);

        System.out.println(ts);

    }
```



```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jdk1.8.0_181\bin\javaw.ex
[Dog [age=3, name=Maxy, breed=Labrador, color=Light Brown], Dog [age=4, n
```



```
package com.dev.encapsulation;

public class Dog implements Comparable<Dog> {

    private int age;
    private String name;
    private String breed;
    private String color;

    @Override
    public int compareTo(Dog d) {
        return (this.age - d.age);

    }
```

ArrayList

public void trimToSize() {
}
Trims the capacity of this <tt>ArrayList</tt> instance to be the list's current size. An application can use this operation to minimize the storage of an <tt>ArrayList</tt> instance.

public void ensureCapacity(int minCapacity) {
}
Increases the capacity of this <tt>ArrayList</tt> instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. **@param minCapacity the desired minimum capacity**

```
public ArrayList(int initialCapacity) {
}
```

```
public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
```

Constructs an empty list with an initial capacity of ten.

```
public ArrayList(Collection<? extends E> c) {
}
```

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.**@param c the collection whose elements are to be placed into this list @throws NullPointerException if the specified collection is null**.

Collection	Overriding	Key value	Duplicate Elements	Null element	Thread Safe	Load factor	Ensure capacity
ArrayList	Yes	no	Yes	yes	No	yes	Yes
LinkedList	Yes	No	Yes	Yes	No		
HashSet	No	No	No	Yes	No	Yes	Yes
TreeSet	Yes	No	No	No	No		
HashMap	No	Yes	No	Yes	No	Yes	Yes
TreeMap	Yes	Yes	No	No	No		
Hashtable	No	Yes	No	No	Yes	Yes	Yes
Vector	yes	No	yes	Yes	Yes	yes	

JAVA LAMBDA EXPRESSIONS

- New features of java 8.
- It provides clear and concise way to represent functional interface using an expression.
- It helps to iterate, filter and extract data from collection.
- The lambda expression is used to provide the implementation of a functional interface.
- In case of lambda expression we don’t need to define the method again for providing the implementation.
- Providing less coding
- Syntax:
- Argument-list → {body}

THREADS

- A thread is a single sequence of executable code with a larger program.
- The main thread that starts automatically when you run a program .
- Java lets to create programs that start additional threads to perform specific tasks.
- Multiple programs are run parrellaly.

MULTIHREADING

- Multithreading is a java feature that allows concurrent execution of two or more parts of a program for max utilization.

Threads Creation

Threads can be created in two ways:

- ❑ By extending thread class
- ❑ By implementing runnable interface
- ❑ To start a thread start() method is invoked which internally invokes run().???

By extending thread class

Create a class that extends the java.lang.Thread.

This class overrides the run() method available in the Thread class

By implementing runnable interface

Create a new class which implements Runnable interface and override run() method.

Thread Properties:

Thread Name →Thread name can be created by programmer in order to identify tthreads

Thread id→. It is a unique number which is created and assigned by the thread scheduler to every single thread in order to identify them uniquely.

Thread Priority →It is used by the thread scheduler to decide the order of execution of the given threads.

It is an integer value ranging between 1-10 which can be set by using setPriority();

To make data in consistence then we have to use join, sleep or make method synchronized

- Thread method creates the system resources, necessary to run the thread, schedules the thread to run, and calls the thread's run method.

States:

- New →Thread that has no started yet
- Runnable → When start() method is called on thread it enters runnable state.

Running →Thread scheduler selects thread to go from runnable to running state. In running state Thread starts executing by entering run() method. **>**Thread scheduler selects thread from the runnable pool on basis of priority, if priority of two threads is same, threads are scheduled in unpredictable manner. Thread scheduler behaviour is completely unpredictable.

>When threads are in running state, **yield()** method can make thread to go in Runnable state.

Blocked → In this state a thread is not eligible to run.

>Thread is still alive, but currently it's not eligible to run. In other words.

> How can Thread return from waiting to runnable state ?

Once **notify() or notifyAll()** method is called object monitor/lock becomes available and thread can again return to runnable state.

Waiting → A thread waiting for another thread to perform.

> How can Thread go from running to waiting state ?

By calling **wait()** method thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock. wait(),notify(),notifyAll()

- ❑ Deadlock →Threads blocked forever. Overcome ITC(Inter Thread Communication).wait(),notify(), notifyAll().
- ❑ Terminated → A thread is considered dead when its run() method completes. destroy() method puts thread directly into dead state.

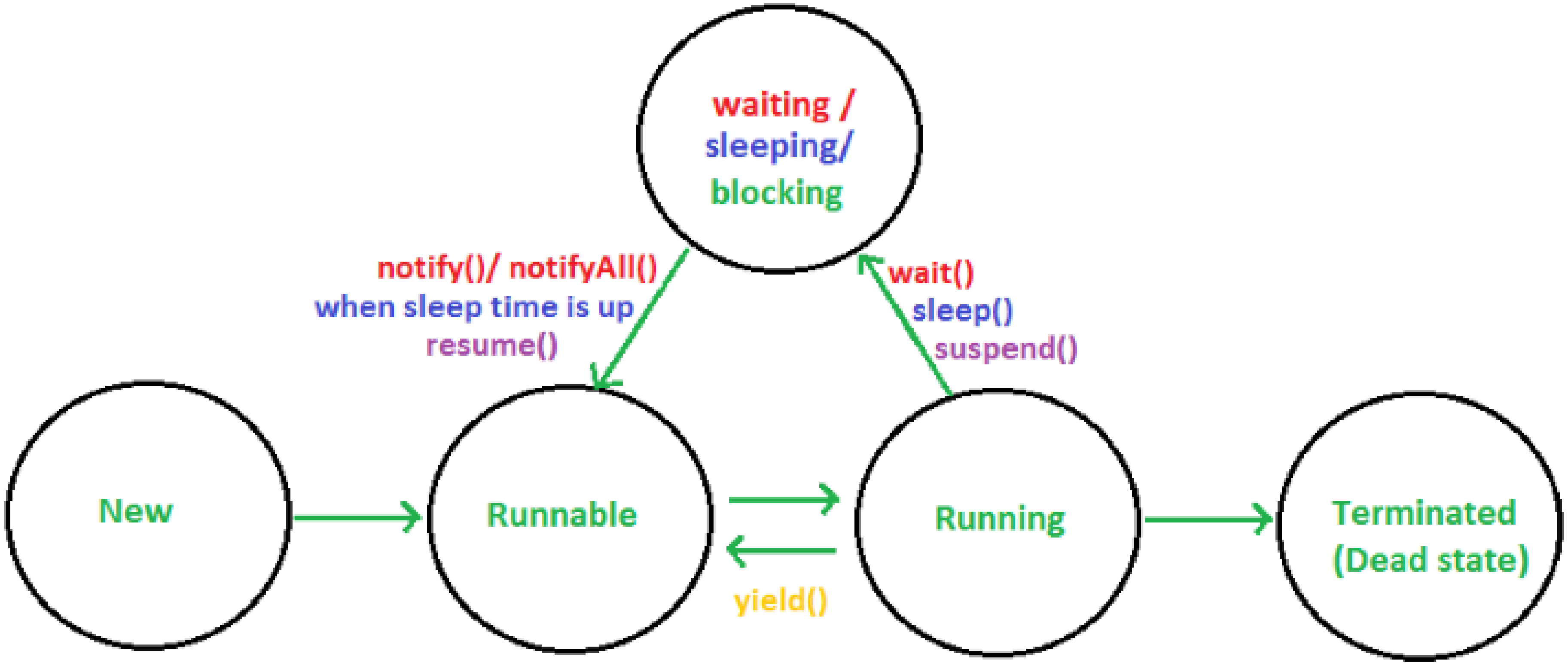


Fig. THREAD STATES

SYNCHRONIZATION

- In many cases concurrently running threads share data and two threads try to do operations on the same variables at the same time. This often results in corrupt data as two threads try to operate on the same data.
- A popular solution is to provide some kind of lock primitive

run()	start()
New thread is not created when invoked but executes	New thread will be created when invoked and executes run()
run() can be called multiple times	start() can be called only once. Throws illegalStartException
Defined in java.lang.runnable	Defined in java.lang.thread

sleep()	wait()
Pauses the thread for few seconds	Thread waits until nottify() or notifyAll () is called.
sleep() is used to pause the execution	wait() is used for inter-thread communication
sleep() doesn’t releases the lock or monitor while waiting.	wait() releases the lock or monitor while waiting.
for multi-thread-synchronization	for time-synchronization

RACE CONDITION

REGULAR EXPRESSION

Defines a pattern of a string
Used to edit,manipulate and search text.
Present in java.util.regex
Pattern → Compiled version of regular exception.Doesnt have any public constructorr and we use its public static method compile() to create the pattern object by passing regular expression argument.
PatternSyntaxException →
Matcher→