GAMBELLA UNIVERSITY

College of Engineering and Technology

Department of computer Science

Object-Oriented Programming

Course code: CoSc2051

Course description

- This programming course emphasizes the methodology of programming from an object-oriented perspective and software engineering principles.
- It allows students to develop the ability to analyse programming problems and design and document suitable solutions and to implement reliable and robust software using contemporary program design methods.
- Topics to be dealt with are: ***classes***: ***data abstraction***, ***information hiding***, ***overloading***; ***inheritance***; ***polymorphism***; ***exceptions handling***.

Course objectives

Upon successful completion of the course, students will be able to:

- *Explain the basic object oriented concepts*
- *Successfully code, debug and run programs with appropriate development environment*
- *Work on more advanced programs*
- *Have clear differentiation between structural and object oriented programming paradigms*

Chapter One

Introduction to Object-Oriented Programming

1.1. Types of programming paradigms

1.2. Overview of OO principles

1.3. Editing, Compiling and Interpreting

Types of programming paradigms

- **Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or *properties*), and code, in the form of procedures (often known as *methods*).

- Programming languages are the heart of computer science. They are the tools used to communicate not only with computers but also with people.

- The challenge of designing language features that support clear expressions, the puzzle of fitting together, the different features to make a useful language, the challenge of appropriately using those features for the clear expression of algorithms all these make up part of the excitement of the study of programming languages.

Cont…

- PL is designed to communicate ideas about algorithms between people and computers.
- There are 4 computational models that describe most programming languages paradigm.

  - Imperative (Procedural languages)
  - Declarative
  - Functional

□ Object Oriented

Cont….

Procedural /imperative Languages

- A **procedural language** is a computer programming language that follows, in order, a set of commands.
- The developer specifies in the source code precisely **what the computer should do, step by step**, to achieve the result.
- The focus is on the “how” of the solution path. For example, this approach can be found in Java, Pascal, and C.
- They make use of functions, conditional statements, and variables to create programs that allow a computer to calculate and display a desired output.

- Using a procedural language to create a program can be accomplished using a programming editor or IDE, like Adobe Dreamweaver, Eclipse, or Microsoft Visual Studio.
- These editors help users develop programming code using one or more procedural languages, test the code, and fix bugs in the code.

Cont...

Functional programming

- Functional programming is a way of thinking about software construction by creating pure functions.

It avoid concepts of shared state, mutable data observed in Object Oriented Programming.(Refer:https://dev.to/christinamcmahon/what-is-functional-programming-5773

- Functional languages emphasis on expressions and declarations rather than execution of statements.

Therefore, unlike other procedures which depend on a local or global state, value output in FP depends only on the arguments passed to the function.

- Ii is a form of declarative programming that expresses a computation directly as pure functional transformation of data.

Cont..

Characteristics of Functional Programming

- Functional programming languages are designed on the concept of mathematical functions that use expressions and recursion to perform computation.
- Functional programming supports
   - **Higher-order functions** : functions that can (1) take functions as arguments and (2) return functions

□**First-class functions:** function values can be as freely used as numbers, strings, Booleans, etc.

- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.
- Example: SQL, Lisp, Python, Prolog

Cont….

Declarative Languages

- Declarative programming the “what” of the solution is described directly.
- Declarative programming paradigm, which means programming is done with expressions or declarations instead of statements.

- Rule based language execute by checking for the presence of a certain enabling condition and when present executing an appropriate action.
- The most common rule based language is Prolog, also called a logic programming language because the basic enabling conditions are certain classes of predicate logic expressions.
- Programming often consists of building a table of possible conditions and then giving appropriate actions if such a condition occurs.

Declarative vs Imperative

- Functional programming is a declarative paradigm, meaning that the program logic is expressed without explicitly describing the flow control.

- Where imperative code usually relies on statements, pieces of code that perform some action.
- Declarative code utilizes expressions, code which evaluates to some value through some combination of function calls, values, and operators.
- In other words, declarative programs abstract the flow control process and focus on the data flow: *what* to do; the *how* gets abstracted away.

Cont…

Object Oriented Programming

- The program is a collection of objects that interact with each other by passing messages that transform their state.
- Complex objects are designed as extensions of simpler objects, inheriting properties of the simpler objects.
- By building concrete data objects, an object oriented program gains the efficiency of imperative languages.
- By building classes of functions that use a restricted set of data objects, we built the flexibility and reliability of the applicative model.
- Object modeling, classification and inheritance are fundamental building blocks for object oriented programming.

procedural vs object-oriented programming

- The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into objects that expose behavior (methods) and data (members or attributes) using interfaces.
- The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an "object", which is an instance of a class, operates on its "own" data structure.
- Terminology varies between the two, although they have similar semantics

Overview of OO principles

The four basics of object-oriented programming

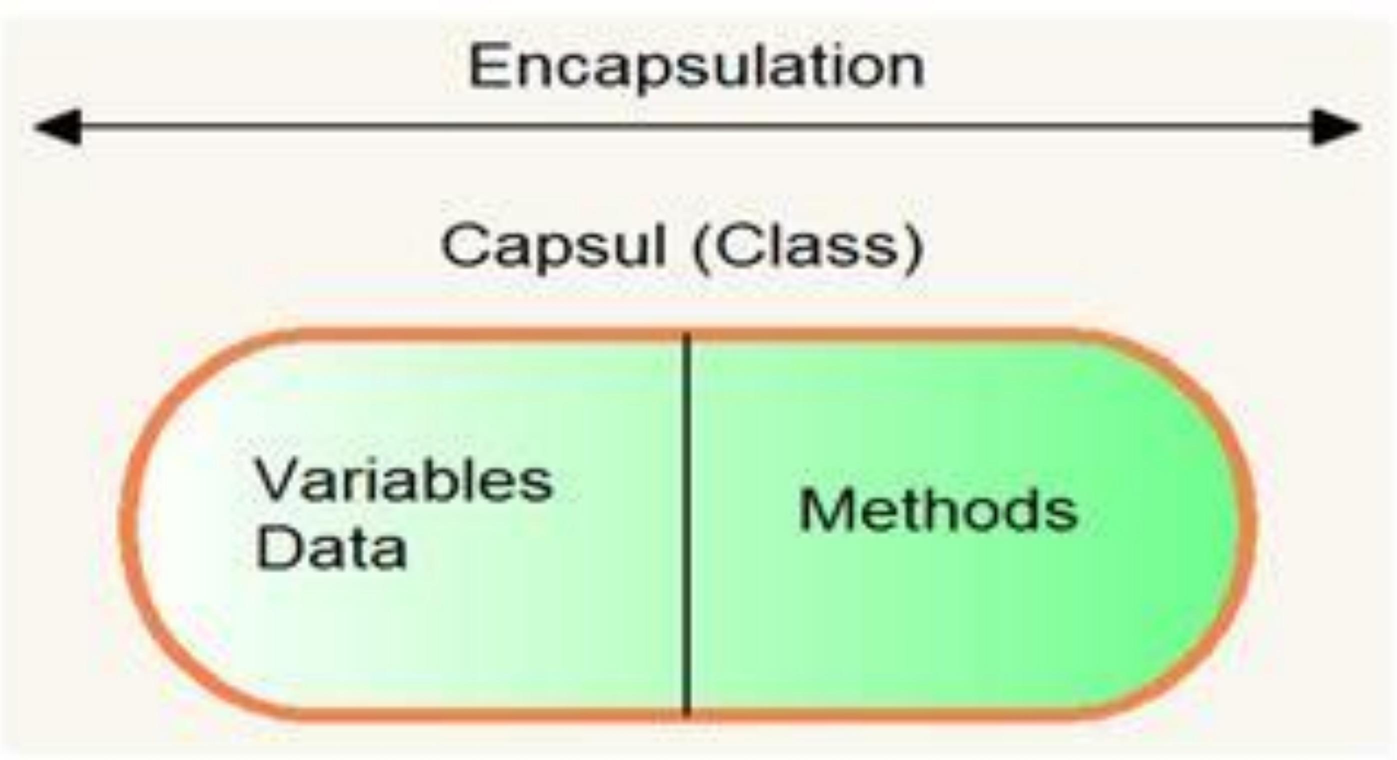
- Object-oriented programming has four basic concepts: encapsulation, abstraction, inheritance, and polymorphism.
- Even if these concepts seem incredibly complex, understanding the general framework of how they work will help you understand the basics of an OOP computer program.
- Below, we outline these four basic principles and what they entail
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism

Encapsulation

Encapsulation

- Encapsula :-oriented programn

- The process Encapsulati
- Due to enca data and codes.
- Through Pu
- Java provid om outside world. priva
- Purpose of



Encapsulation

Cont…

Example1

- ✦Bank gives controlled access to check our balance, amount deposit or amount withdraw by providing methods like balanceCheck(), amtDeposit(), amtWithdraw() etc.

What are the primary benefits of encapsulation?

- ✦It improves maintainability and flexibility and re-usability.
- ✦Encapsulation helps to make our code secure.
- ✦Control the way data is accessed or modified.
- ✦Encapsulation in Java makes unit testing easy.

- ✦ Encapsulation allows you to change one part of code without affecting other parts of code.

Abstraction

- Abstraction focus on functionalities not how does it works. But what it does.
- Abstraction is process of hiding the internal implementation and showing only necessary functionality/features to the user.
- Abstraction is implemented using interface and abstract class.

Abstraction

Example

- ATM machine: Here we can withdraw money using shown features but internal details or implementation details are hidden which are not really required to the user.
- TV remote: Person who using TV remote can able to operate. But don’t know how it works internally.

Abstraction vs Encapsulation?

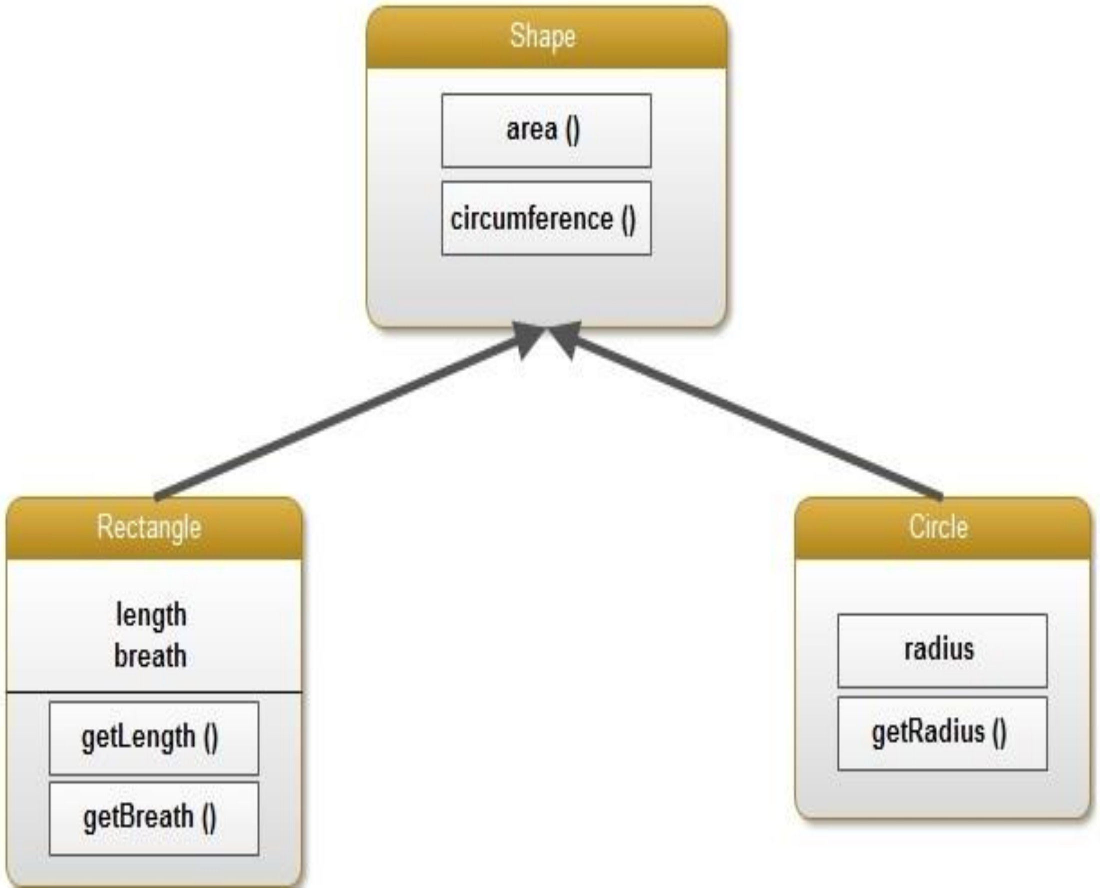
- In higher end both abstraction and Encapsulation looks similar.
- But purpose of these two concepts is totally different.
- Abstraction hides the things at design level but encapsulation hide things at implementation level.

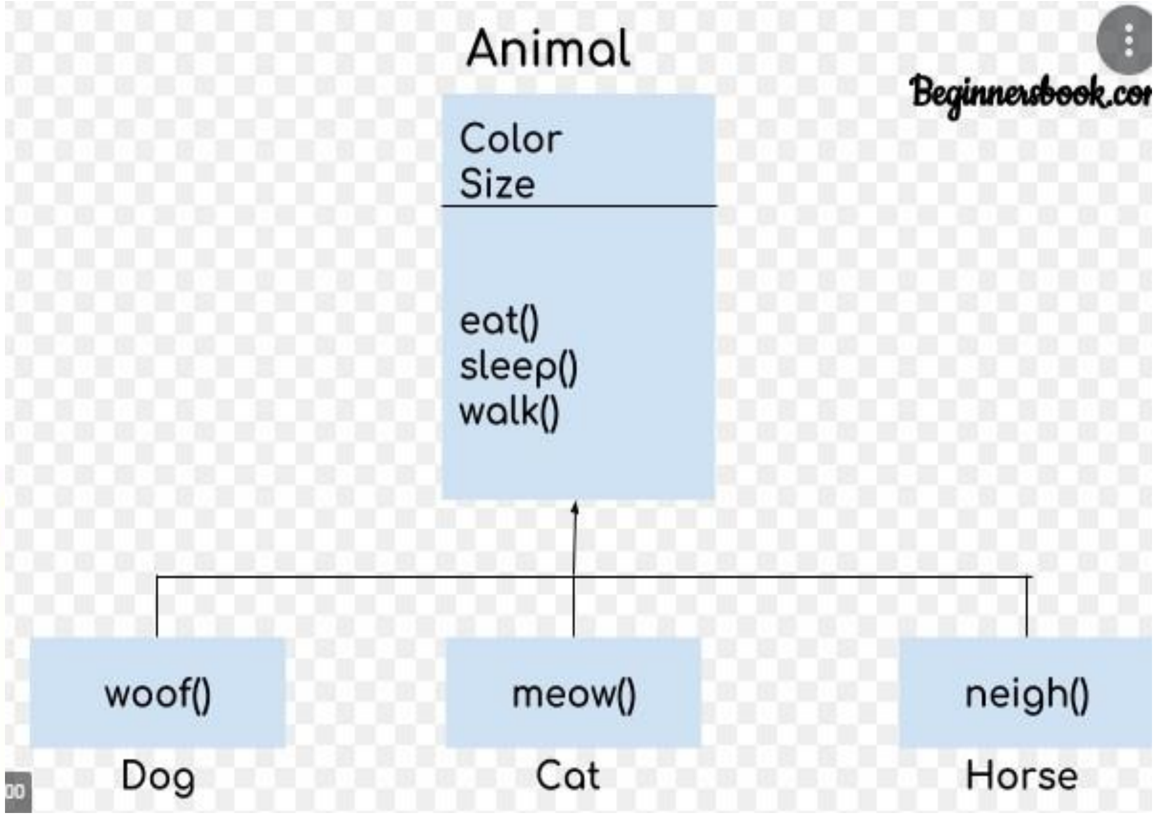
- Abstraction can be achieved using interface and abstract class while Encapsulation can be achieved by using access modifiers
  e.g. public, private and protected.
- Abstraction focus on What should be done and Encapsulation focus on How it should be done.

Inheritance

- Inheritance represents the IS-A relationship between the objects which is also known as a parent-child relationship.
- The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called inheritance.

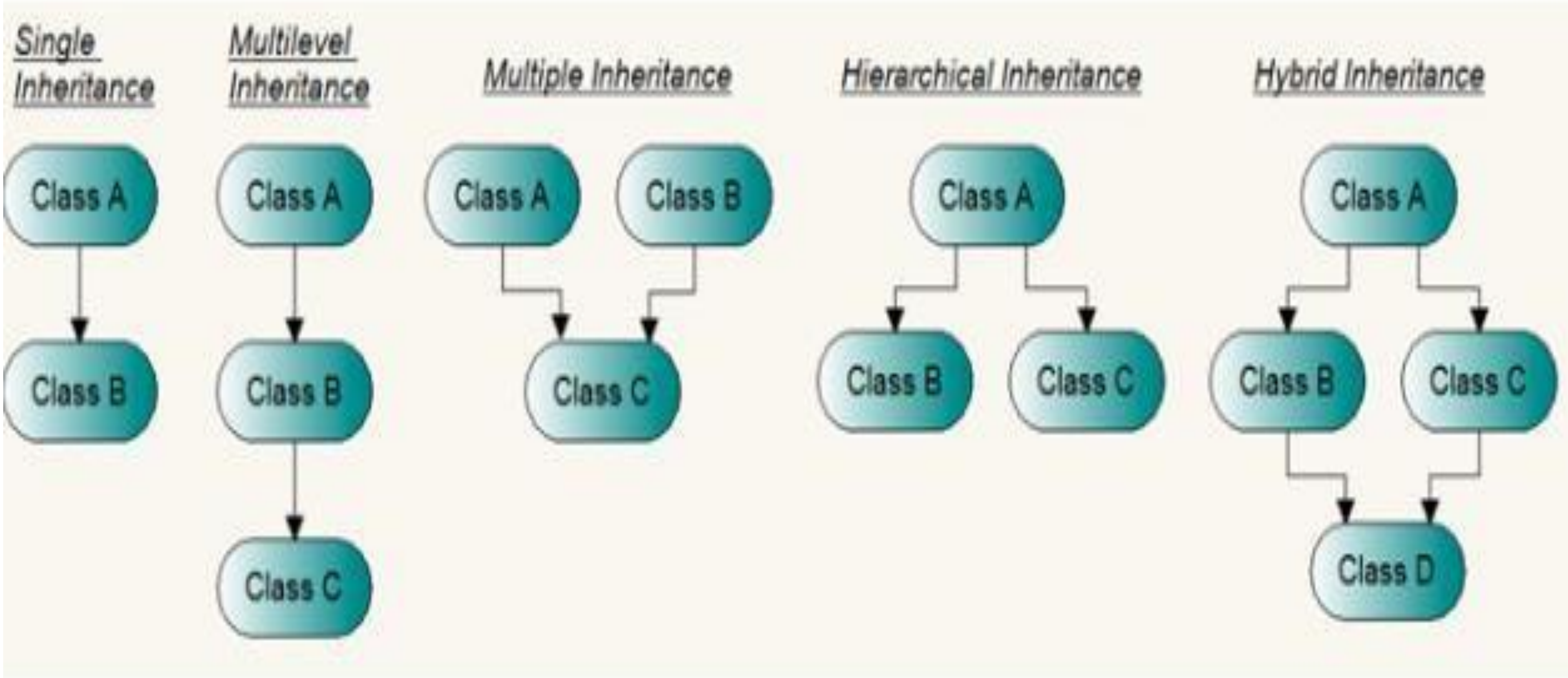
- Inheritance which allows a class to inherit behavior and data from other class. In other word, inheritance is a mechanism where in a new class is derived from an existing class.
- The code that is already present in base class need not be rewritten in the child class.
- The inheritance mechanism is very useful in code reuse.
- A class which is inherited is called a Base Class / Parent class / Super class. The classes which are inheriting Base classes are called Child class / Sub class.
- In Java, A subclass can have only one Parent class, whereas a Parent class may have one or more Child classes. Because Java doesn’t support multiple inheritance.
- The meaning of “extends” is to increase the functionality.


Shape
area ()
circumference ()
Rectangle
length
breath
getLength ()
getBreath ()
Circle
radius
getRadius ()


Animal
Beginnersbook.co
Color
Size
eat()
sleep()
walk()
woof()
Dog
meow()
Cat
neigh()
Horse

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance


Single Inheritance
Class A
Class B
Multilevel Inheritance
Class A
Class B
Class C
Multiple Inheritance
Class A
Class B
Class C
Hierarchical Inheritance
Class A
Class B
Class C
Hybrid Inheritance
Class A
Class B
Class C
Class D

Cont…

What is the primary benefit of inheritance?

- Code reusability and reduces duplicate code.
- It makes code more flexible.
- It takes less time for application development and execution.
- Memory consumption is less and performance will be improved.
- Since less number of code, storage cost will be reduced.

Polymorphism

- Polymorphism

  □makes to perform a single action in different ways.

Single task can be done in different way.

is the ability of an object to take on many forms.

- We can perform polymorphism in java by method overloading and method overriding.
- Like below image, same person (Mr.Jhon) will behave like Son, Husband, Father, Employee in difference places.

Polymorphism: example

Mr. Jhon behave like '**Son**' with his parents

Mr. Jhon behave like '**Husband**' with his wife

Mr. Jhon behave like '**Father**' with his daughter

Mr. Jhon behave like '**Employee**' in Office

Polymorphism

Important points on Polymorphism

1. Types of Polymorphism

- **Method Overloading:** (Compile time/static polymorphism or static/early binding).
- **Method Overriding:** (Runtime time/dynamic polymorphism or dynamic/late binding).

2. What is Overloading and overriding?

- **Overloading**: Whenever several methods have same names and it is determined at the compile time.
- **Overriding**: If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java. It is determined at the run time.

3. More points on static and dynamic binding

Static binding

- Static binding will be done for static, private and final methods during compile-time.
- Compiler knows that these methods cannot be overridden. So compiler doesn’t have any difficulties to identify the object of the local class. So this type of binding is called Static binding.

- The binding which can be resolved at compile time by compiler

Cont…

Dynamic binding

- Dynamic binding will be done for methods which can be overridden. Compiler has no idea as to which method has to be called.
- Since dynamic binding happens in run time so the binding would be delayed.

Important Points

- Private, final and static members use static binding while for virtual methods binding is done during run time based upon run time object.

When developing programs with command-line tools, you typically iterate the followingEditing, Compiling and Interpreting steps:

1.Edit:

- Use an *editor* to develop the code in your programming language of choice.
- **Result:** a *text file* with the source code of the program

  (e.g. Sample1.java)

2.Compile

- Use a *compiler* to translate the source code into a version understood by the machine.
- **Result:** an *executable file* containing byte code of the virtual machine, in this case theJava Virtual Machine (JVM),

  e.g. Sample1.class

3.Run

- Execute the program on the machine. **Result:** you see the result of executing your program (e.g. messages printed by the

Chapter Two

Objects and Classes

- Defining a class
- Creating an Object
- Instantiating and using objects
- Instance fields
- Constructors and Methods

- Access Modifiers
- Encapsulation

Defining a class

Class Definition in Java

- In object-oriented programming, a **class** is a basic building block.
- A class is a user defined blueprint or template or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
- Instantiating a class is creating an object (variable) of that class that can be used to access the member variables and methods of the class.

- A class can also be called a logical template to create the objects that share common properties and methods.
- For example, an Employee class may contain all the employee details in the form of variables and methods.
- If the class is instantiated i.e. if an object of the class is created (say e1), we can access all the methods or properties of the class.

Cont…

In general, class definition includes the following in the order as it appears:

- **Modifiers:** A class can be public or has default access.
- **class keyword:** The class keyword is used to create a class.
- **Class name:** The name must begin with an initial letter (capitalized by convention).

- **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

Syntax

```
<access specifier> class class_name

  {

       // member variables

       // class methods

}
```

 fields are used to store data

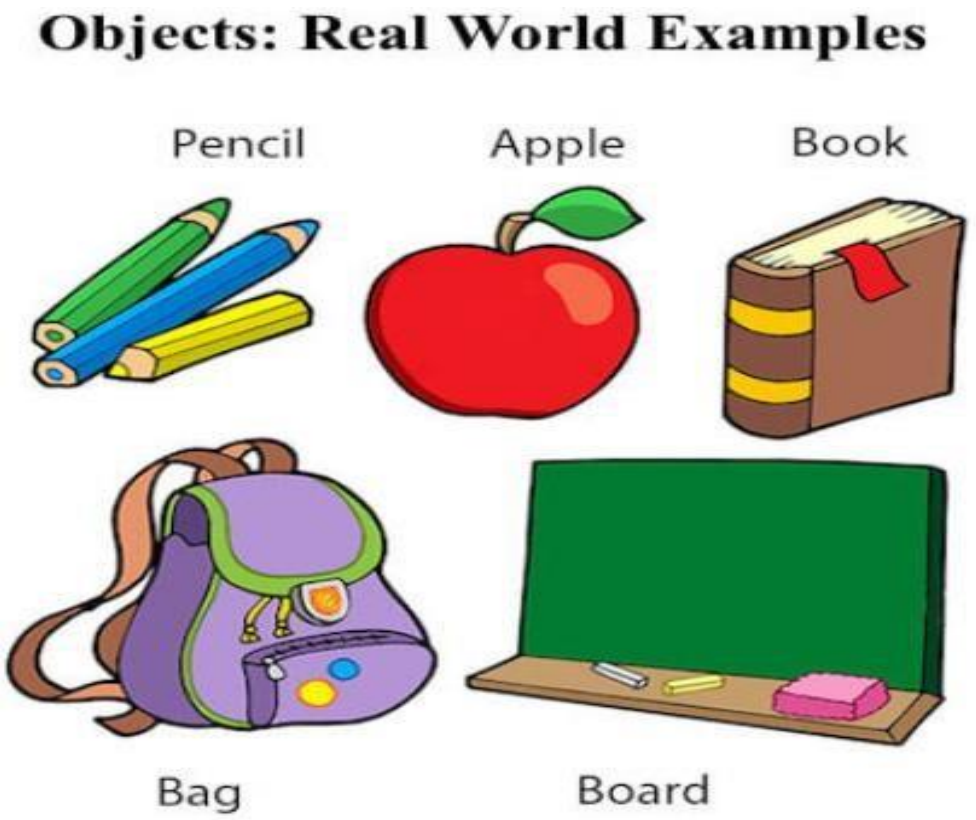
Example

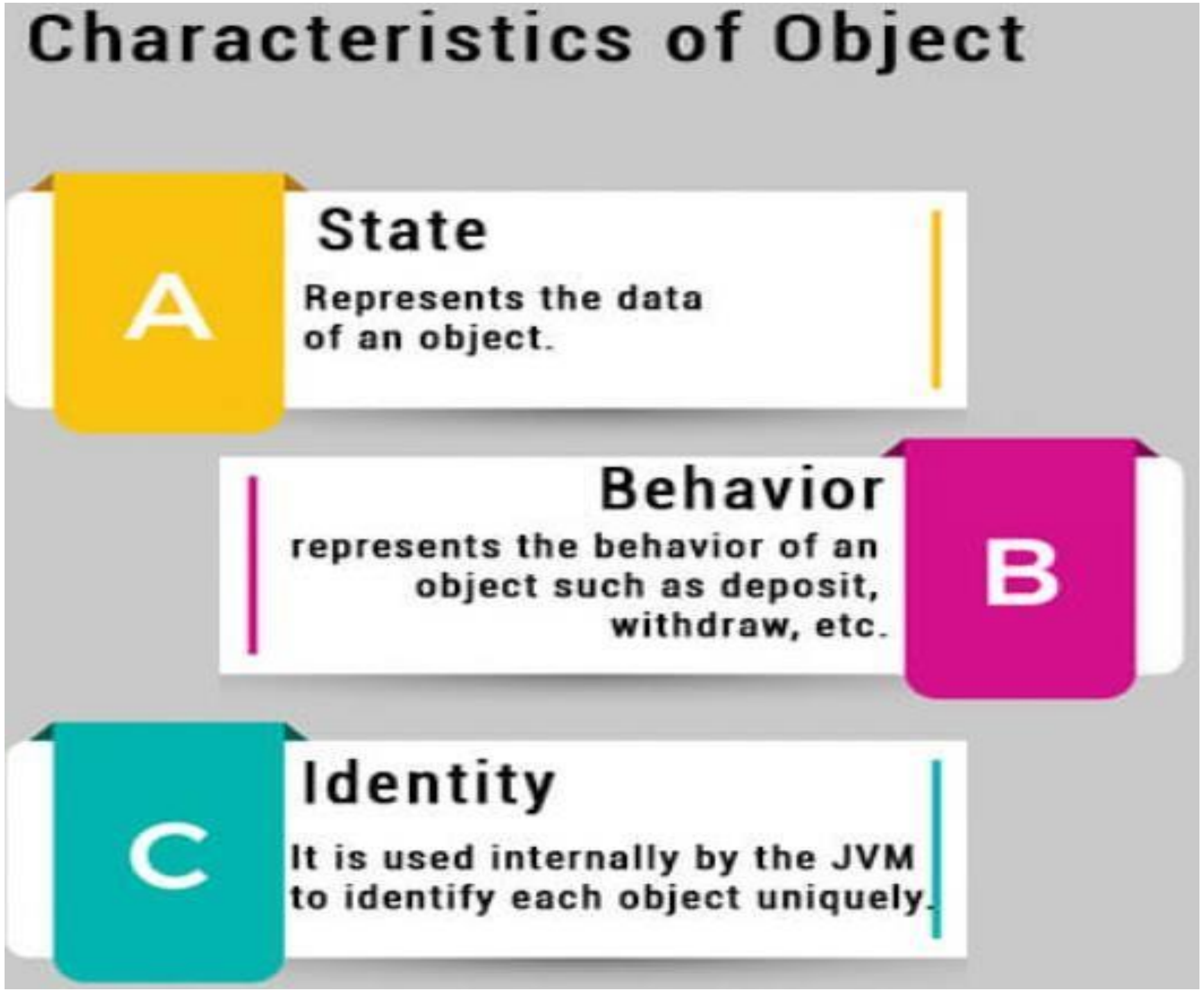
```
Public class Bicycle {
 // state or field private
int gear = 5;
// behavior or method
  public void braking() {
    System.out.println("Working of Braking");
      }
}
```

- methods are used to perform some operations

What is an object?

- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.
- It can be physical or logical (tangible and intangible).
- The example of an intangible object is the banking system.





Characteristics of Object

State

Represents the data of an object.

Behavior

represents the behavior of an object such as deposit, withdraw, etc.

Identity

It is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is *BIC*; color is Blue, known as its state. It is used to write, so writing is its behavior.

Cont…

- **An object is an instance of a class.**
  - A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
- **Object Definitions:**
  - An object is *a real-world entity*.
  - An object is *a runtime entity*.
  - The object is *an entity which has state and behavior*.

- The object is *an instance of a class*.

Object creation

Create an object called "myObj" and print the value of x

```
public class Main { int x = 5; public

 static void main(String[] args) {

   Main myObj = new Main();
   System.out.println(myObj.x);
      }
```

}

Creating an Object(cont….)

- In Java, you create an object by creating an *instance of a class* or, in other words, *instantiating a class*.
- Often, you will see a Java object created with a statement like this one: Date today = new Date(); This statement creates a new Date object (Date is a class in the java.util package). This single statement actually performs three actions: declaration, instantiation, and initialization.

- Datetoday is a variable declaration which simply declares to the compiler that the name today will be used to refer to an object whose type is Date, the new operator instantiates the Date class (thereby creating a new Date object), and Date() initializes the object.

Declaring an Object (instantiating a class)

- When an object of a class is created, the class is said to be instantiated.
- All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

- While the declaration of an object is not a necessary part of object creation, object declarations often appear on the same line as the creation of an object. Like other variable declarations, object declarations can appear alone like this: Datetoday;Either way, declaring a variable to hold an object is just like declaring a variable to hold a value of primitive type:
- Typenamewhere Typeis the data type of the object and nameis the name to be used for the object.

Cont...

- In Java, classes and interfaces are just like *a data type.*
- So Type can be the name of a class such as the Date class or the name of an interface.

- Variables and Data Types discusses variable declarations in detail.
- Declarations notify the compiler that you will be using name to refer to a variable whose type is *Type*.
- **Declarations do not create new objects.** Datetodaydoes not create a new Date object, just a variable named today to hold a Date object.
- To instantiate the Date class, or any other class, use the newoperator.

Cont…

- Rectangle rect = new Rectangle(0, 0, 100, 100); (Recall from Variables and Data Types that a class essentially defines a new reference data type. So, Rectangle can be used as a data type in your Java programs.
- The value of any variable whose data type is a reference type, such as rect, is a reference (a pointer in other terminology) to the actual value or set of values represented by the variable.
- In this tutorial, a reference may also be called an object reference or an array reference depending on the data that the reference is referring to.)

Instantiating an Object

- The new operator instantiates a class by allocating memory for a new object of that type.
- new requires a single argument: a call to a constructor method. Constructor methods are special methods provided by each Java class that are responsible for initializing new objects of that type.
- *The new operator creates the object, the constructor initializes it.*
- Here's an example of using the new operator to create a Rectangle object (Rectangle is a class in the java.awt package):

- new Rectangle(0, 0, 100, 200); In the example, Rectangle(0, 0, 100, 200) is a call to a constructor for the Rectangle class.
- The new operator returns a reference to the newly created object. This reference can be assigned to a variable of the appropriate type.

Initializing an Object

- Initializing an object means storing data into the object.
- As mentioned previously, classes provide constructor methods to initialize a new object of that type. A class may provide multiple constructors to perform different kinds of initialization on new objects.
- When looking at the implementation for a class, you can recognize the constructors because they have the same name as the class and have no return type. Recall the creation of the Dateobject used at the beginning of this section.
- The Dateconstructor used there doesn't take any *arguments:*Date().

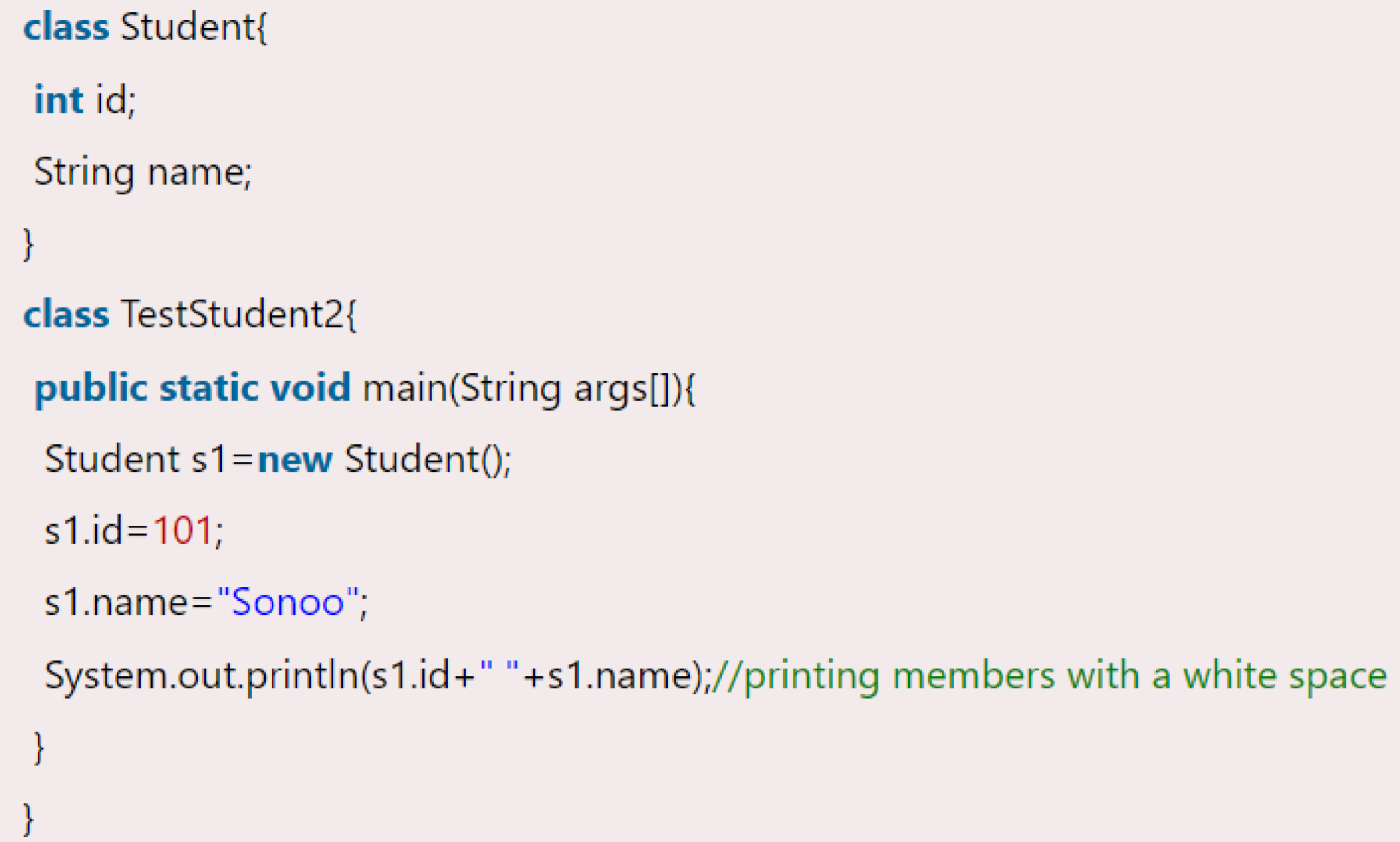
- A constructor that takes no arguments, such as the one shown, is known as the *default constructor*. Like Date, most classes have at least one constructor, the default constructor.
- If a class has multiple constructors, they all have the same name but a different number or type of arguments.

Cont…

- Each constructor initializes the new object in a different way. Besides the default constructor used to initialize a new Date object earlier, the Date class provides another constructor that initializes the new Date with a year, month, and day.
- Date MyBirthday = new Date(1996, 8, 30); The compiler can differentiate the constructors through the type and number of the arguments.

- This section talked about how to use a constructor.
- Constructors later in this lesson explains how to write constructor methods for your classes.

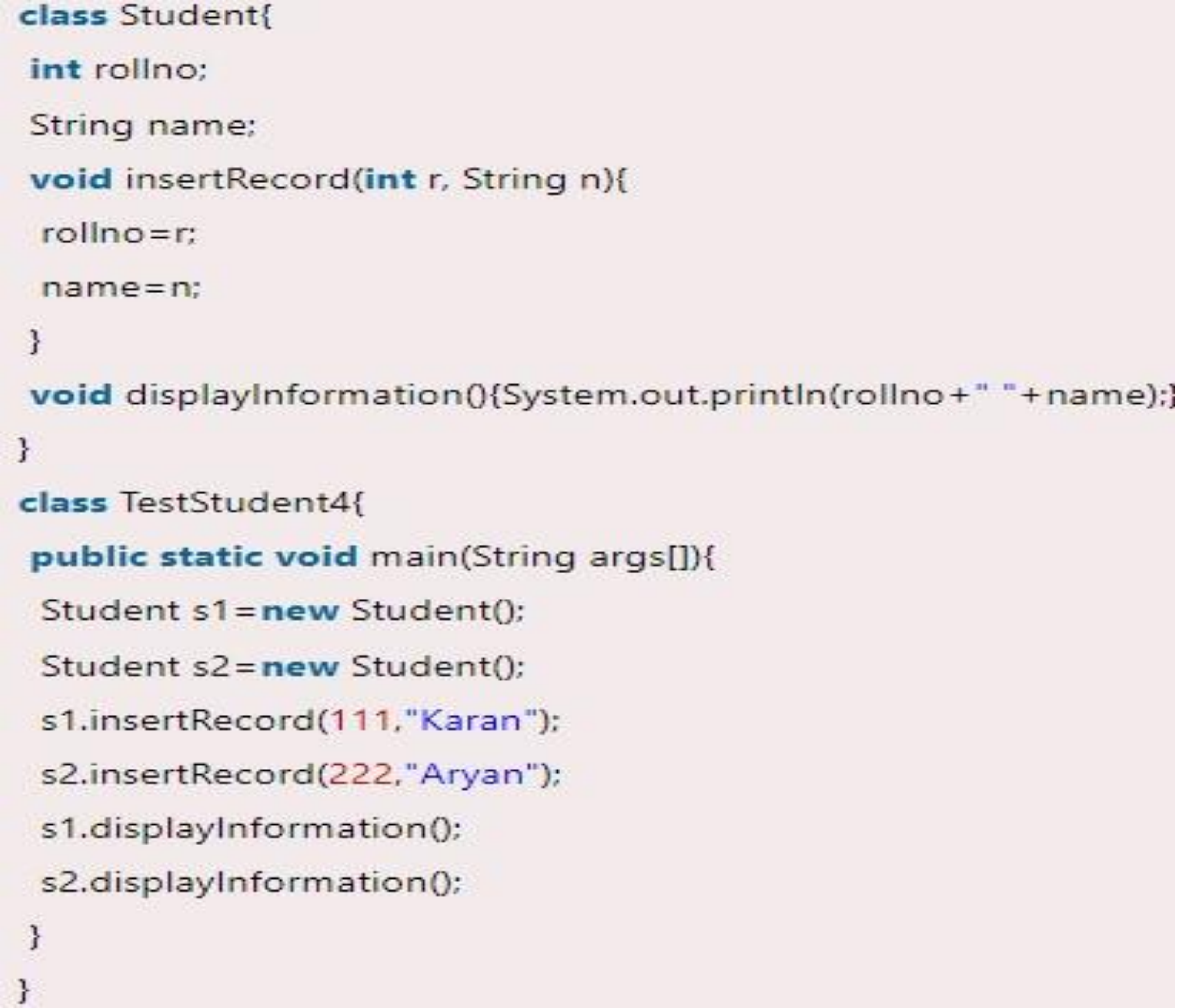
- 3 Ways to initialize object
  By reference variable
  By method
  By constructor(to be discussed in detail later)



```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

2) Object and Class Example: Initialization through method

- In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method.
- Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.



```java
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

3) Object and Class Example: Initialization through a constructor

You will read more about Object and Class Example: Initialization through a constructor

Member Variables

- A *member variable* has a *name* (or *identifier*) and a *type*; and holds a *value* of that particular type.
- *Variable Naming Convention*
- A variable name shall be a noun or a noun phrase made up of several words.
- The first word is in lowercase and the rest of the words are initial-capitalized (camel-case), e.g., fontSize, roomNumber, xMax, yMin and xTopLeft.
- The formal syntax for variable definition in Java is:
- [*AccessControlModifier*] *type variableName* [**=** *initialValue*]**;**
- [*AccessControlModifier*] *type variableName-1* [**=** *initialValue-1*] [**,** *type variableName-2* [**=** *initialValue-2*]] ... **;**

- For example
- private double radius;
- public int length = 1, width = 1;

Member Methods

- A method
  1. receives arguments from the caller,
  2. performs the operations defined in the method body, and
  3. returns a piece of result (or void) to the caller.
- The syntax for method declaration in Java is as follows:

```
[AccessControlModifier] returnType methodName ([parameterList]) {
    // method body or implementation
     ......
}
```

- For examples:

```
// Return the area of this Circle instance
 public double getArea() {
    return radius * radius * PI;
 }
```

Cont....

✝ Method Naming Convention

- A method name shall be a verb, or a verb phrase made up of several words.
- The first word is in lowercase and the rest of the words are initial-capitalized (camelcase).
- For example, getArea(), setRadius(), getParameterValues(), hasNext().

✞ Variable name vs. Method name vs. Class name

- A variable name is a noun, denoting an attribute; while a method name is a verb, denoting an action.
- They have the same naming convention (the first word in lowercase and the rest are initial-capitalized). Nevertheless, you can easily distinguish them from the context. Methods take arguments in parentheses (possibly zero arguments with empty parentheses), but variables do not.
- In thiswriting, methods are denoted with a pair of parentheses, e.g., println(), getArea() for clarity.
- On the other hand, class name is a noun beginning with uppercase.

Constructors

- A constructor initializes an object when it is created.

- It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.
- Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.
- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Cont…

- Syntax

```
Class Student {
        Student(){
          .........
         }
}
```

- Two types of constructors
- No argument Constructors
- Parameterized Constructors

Cont...

No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

```
public Class Student {              Public class TestStudent{

    Int id;                             Public static void main(string args[]){

                                                                            id=101;
        Student() {   Student       s=      new      Student();
        System.out.printl(s.id);
        }                           }

   }
```

Cont…

Parameterized Constructors

- Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

```
public Class Student{
 int id;

 Student(int i) {


   id=i;
```

```
public class TestStudent {
    public static void main(string args[]){

     Student s= new student(101);
    System.out.printl(s.id);

     }
    }                                   }
}
```

Constructor (cont…)

- A constructor is different from an ordinary method in the following aspects:
- The name of the constructor method is the same as the class name. By classname's convention, it begins with an uppercase (instead of lowercase for ordinary methods).
- Constructor has no return type. It implicitly returns void. No return statement is allowed inside the constructor's body.
- Constructor can only be *invoked via the "new" operator.* It can only be used *once* to initialize the created object. Once an instance is constructed, you cannot call the constructor anymore.

- Constructors are not inherited .
- **Default Constructor**:
  - A constructor with no parameter is called the *default constructor*.
  - It initializes the member variables to their default value.

```
Class Student {
  int rollNo;
 String name;

 Student(int r, String
   n){ rollNo=r;
   name=n;
  }

 Void
   displayStudentInfo() {
```

Example

```
Class TestStudent{ public static void
  main(String args[])
  { student s =new
   Student(101,”Abebe”);
              s. displayStudentInfo();
         }
```

```
                        }
            System.out.println(rollNo+” ”+ name);
        }
    }
```

Access Modifiers

public vs. private

- An *access control modifier* can be used to *control the visibility* of a class, or a member variable or a member method within a class.
- We begin with the following two access control modifiers:
- public: The class/variable/method is accessible and available to *all* the other objects in the system.

- private: The class/variable/method is accessible and available *within this class only*.

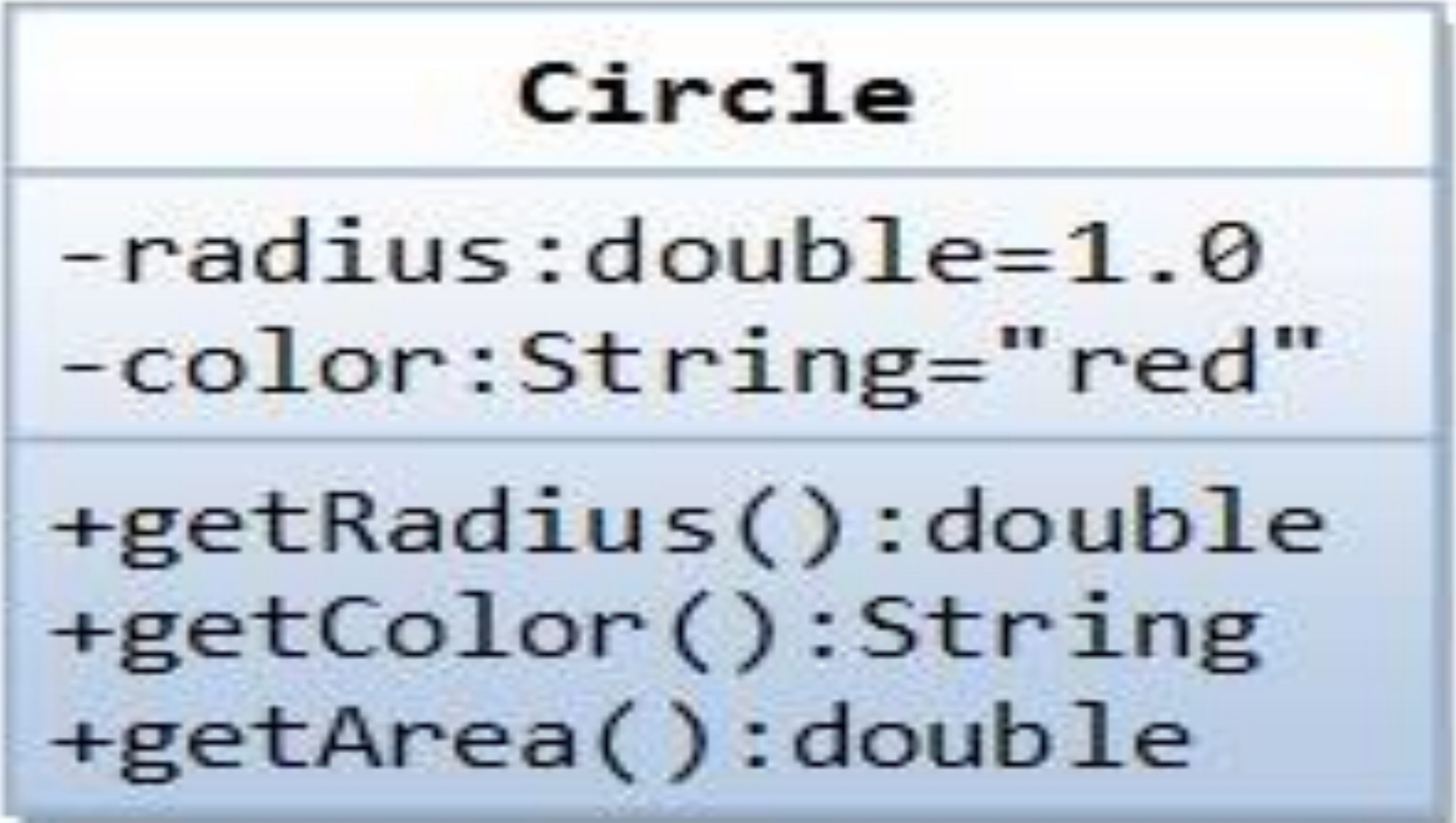
```
Class Student {
    private int rollNo;
    public String name;
      …….
     ………
      }
```

```
Class  TestStudent(){  public  static  void
   main(String args[])  {
        Student s =new Student();
             s. rollNo=101;
             s. name="Abebe";
System.out.println(rollNo+" "+name); //error
           }  }
```

Cont…

- For example, in the above *Student* definition, the member rollNo is declared private. As the result, rollNo is accessible inside the Student class, but NOT in the TestStudent class. In other words, you cannot use "s. rollNo " to refer to s's rollNo in TestStudent.

- Try inserting the statement "System.out.println(s.rollNo)" in TestStudent and observe the error message.
- Try changing rollNo to public in the Student class, and re-run the above statement.
- On the other hand, the method getRollNo() is declared public in the *Student* class. Hence, it can be invoked in the TestStudent class.
- **UML Notation:** In UML class diagram, public members are denoted with a "+"; while private members with a "-".



Circle

-radius:double=1.0
-color:String="red"

+getRadius():double
+getColor():String
+getArea():double

Information Hiding and Encapsulation

- A class encapsulates the name, attributes and behaviors into a "3compartment box".
- Member variables of a class are typically hidden from the outside word (i.e., the other classes), with private access control modifier.
- Access to the member variables are provided via *public accessor methods*, e.g., getRollNo() and getName().
- This follows the principle of *information hiding*. That is, objects communicate with each others using well-defined interfaces (public methods). Objects are not allowed to know the implementation details of others. The implementation details are hidden or encapsulated within the class. Information hiding facilitates reuse of the class.

- **Rule of Thumb:** Do not make any variable public, unless you have a good reason.

The public Getters and Setters for private Variables

- To allow other classes to *read* the value of a private variable says xxx, we provide a *get method* (or *getter* or *accessor method*) called getXxx().
- A get method needs not expose the data in raw format. It can process the data and limit the view of the data others will see. The getters shall not modify the variable.
- To allow other classes to *modify* the value of a private variable says xxx, we provide a *set method* (or *setter* or *mutator method*) called setXxx(). A set method could provide data validation (such as range checking), or transform the raw data into the internal representation.
- For example, in our Student class, the variables rollNo and *name* are declared private.

- That is to say, they are only accessible within the Student class and not visible in any other classes, such as the TestStudent class. You cannot access the private variables rollNo and name from TestStudent class directly - via says s.rollNo or s.name.

Cont…

- The Student class provides two public accessor methods, namely, getrollNo() and getName(). These methods are declared public. The class TestStudent can invoke these public accessor methods to retrieve the rollNo and name of a student object, via says s.getRollNo() and s.getName().
- There is no way you can change the rollNo or name of a Student object, after it is constructed in the TestStudent class. You cannot issue statements such as s.rollNo= 5.0 to change the rollNo of instance s, as rollNo is declared as private in the Student class and is not visible to other classes including TestStudent.

- If the designer of the Student class permits the change the rollNo and name after a Student object is constructed, he has to provide the appropriate set methods (or setters or mutator methods).

Example

```
    Class Student{ private
       int rollNo; private
       String name;
     public int getRollNo(){
       return rollNo;
      } public String
     getName(){ return name;
      }
  }
    public void setRollNo(int rollNo){
```

```
Class TestStudent{ public static void
  main(String [] args){
    Student s=new Student(){
        s.setRollNo(101);
        s.setName("Abebe");
    System.out.println(s.getRollNo());
    System.out.println(s.getName());
    this.rollNo=rollNo;}
public void
setName(String name){
this.name=name;
    }
```

Keyword "this"

- You can use keyword "this" to refer to *this* instance inside a class definition.
- One of the main usage of keyword *this* is to resolve ambiguity.

```
public class Student { int rollNo;                    // Member
   variable called “rollNo"
public Student(int rollNo) { // Method's argument also called " rollNo "
    this. rollNo = rollNo ;
```

```
            // " rollNo = rollNo " does not make sense!
            // "this. rollNo " refers to this instance's member
        variable // " rollNo " resolved to the method's argument. }
        ...
      }
```

Constants (final)

- Constants are variables defined with the modifier final. A final variable can only be assigned once and its value cannot be modified once assigned.
- For example,
  - public final double PI = 3.14;
  - private final int MAX_ID = 9999;
- MAX_ID = 10000; // error: cannot assign a value to final variable MAX_ID
- N.B // You need to initialize a final member variable during declaration private final int SIZE; //

  error: variable SIZE might not have been initialized

- **Constant Naming Convention:** A constant name is a noun, or a noun phrase made up of several words. All words are in uppercase separated by underscores '_', for examples, PI, MAX_INTEGER and MIN_VALUE.
- Advanced Notes:
- A final primitive variable cannot be re-assigned a new value.
- A final instance cannot be re-assigned a new object.
- A final class cannot be sub-classed (or extended).
- A final method cannot be overridden.

Method toString()

read by yourself

Chapter Three

Inheritance and Polymorphism

Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.

- Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Inheritance

✞Why use inheritance in java

- For Method Overriding(so runtime polymorphism can be achieved).
- For Code Reusability.

✞Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Inheritance

- The syntax of Java Inheritance **class** Subclass-name **extends** Superclass-name

  {

  //methods and fields

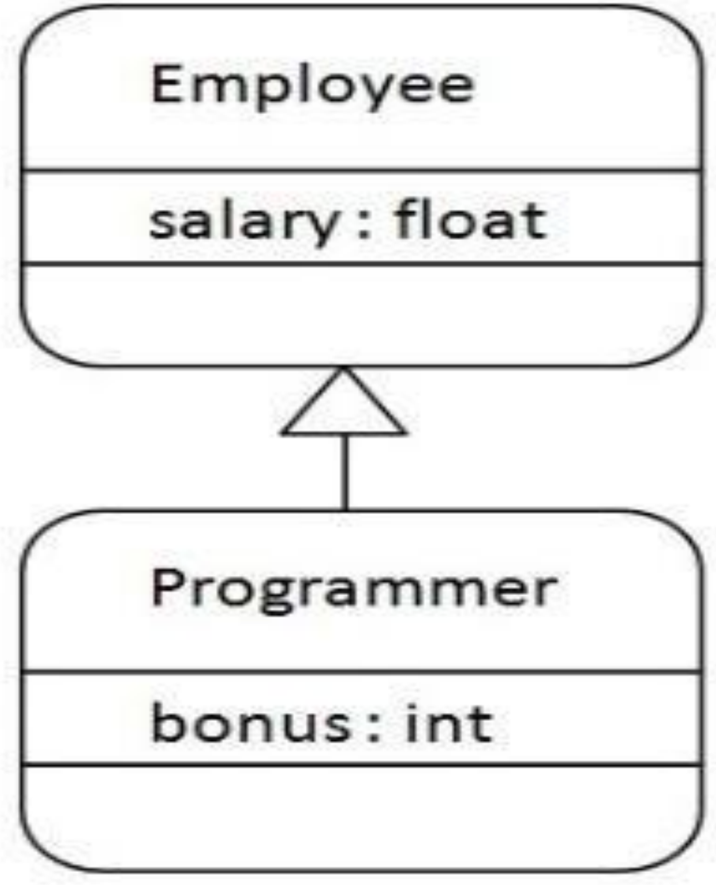
```
        }
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

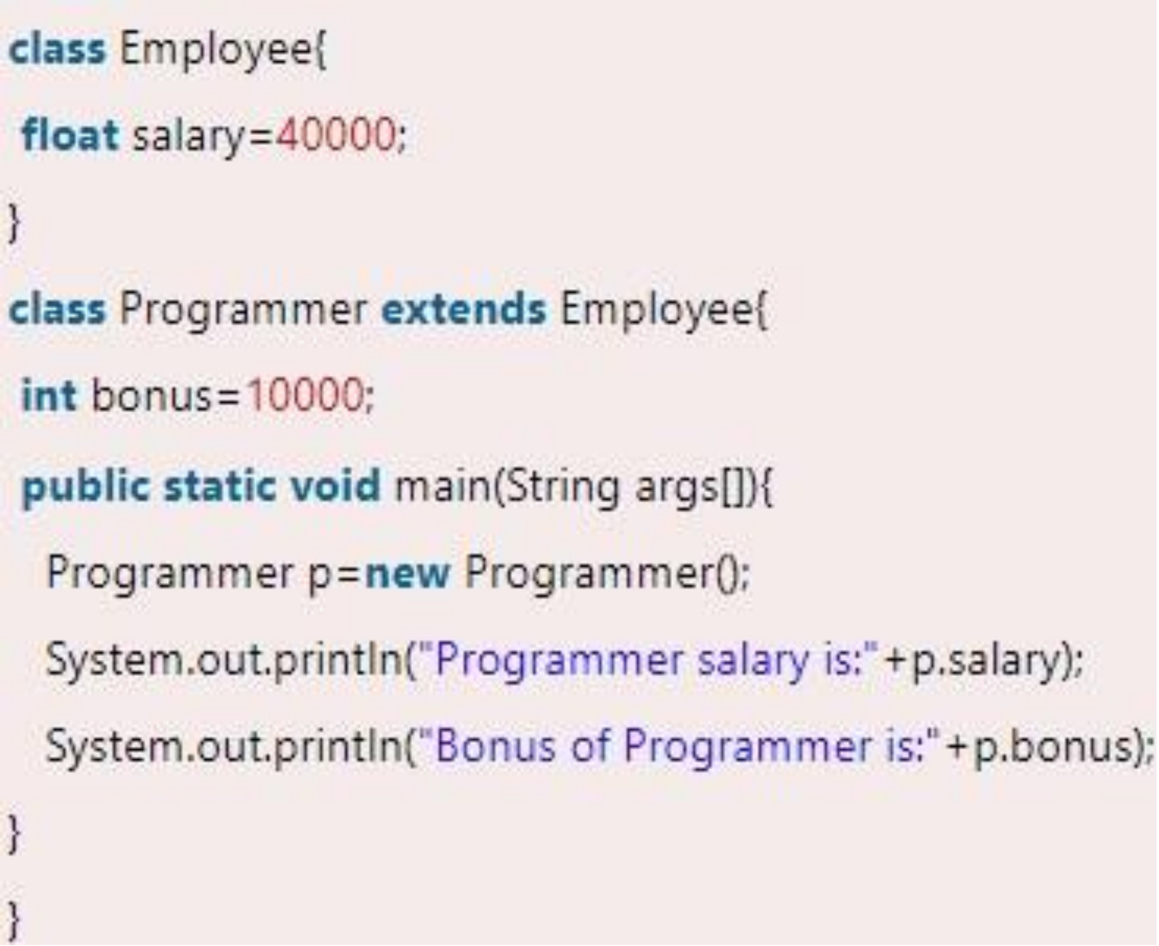
Inheritance

- As displayed in the figure, Programmer is the subclass and Employee is the superclass.
- The relationship between the two classes is **Programmer IS-A Employee**.
- It means that Programmer is a type of Employee.

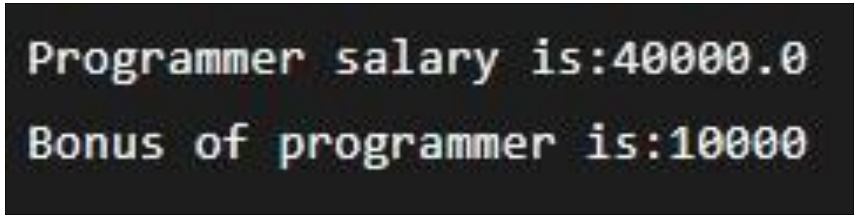
```
Programmer salary is:40000.0 Bonus of programmer is:10000
```



Inheritance



```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```



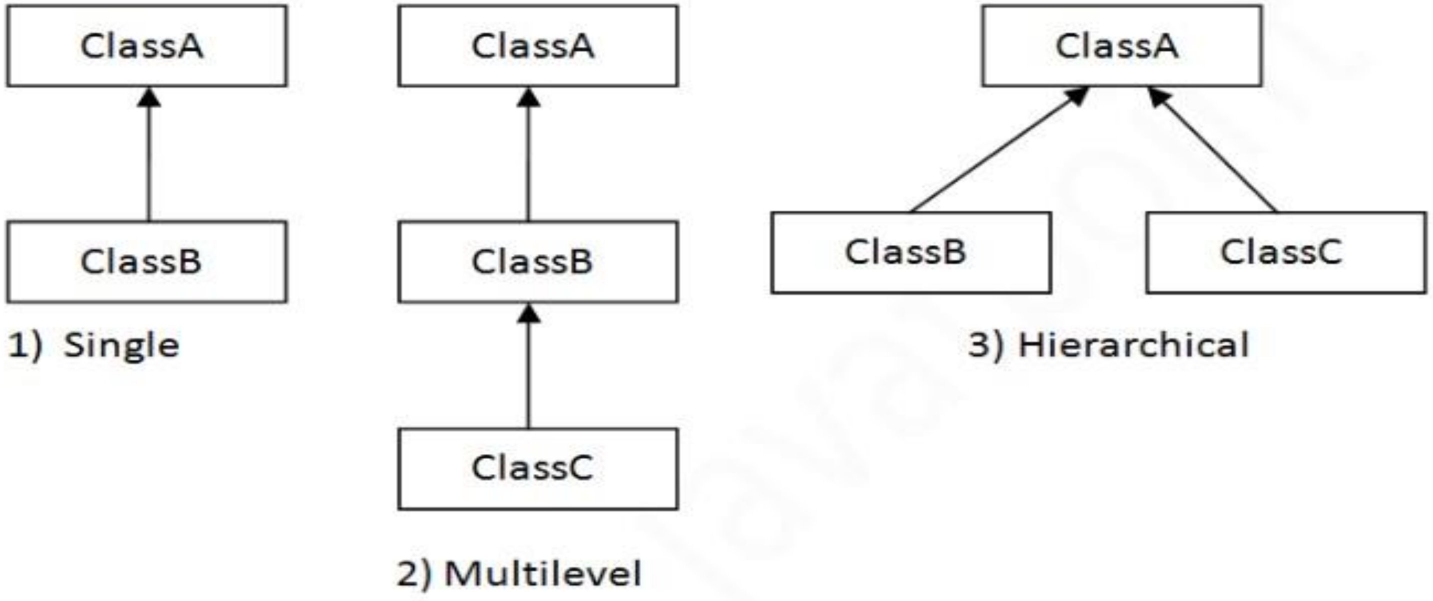
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Inheritance

Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



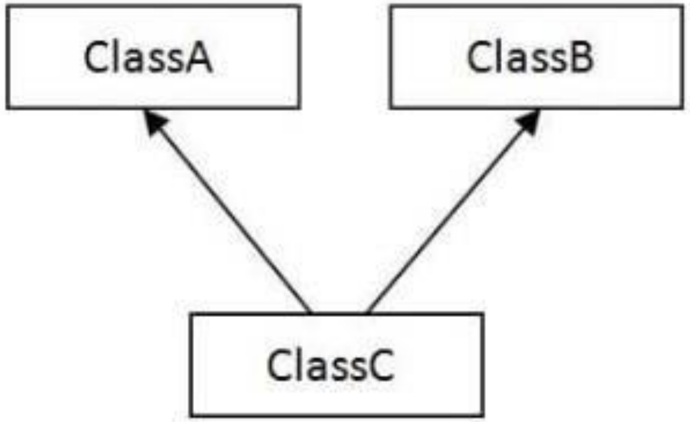
1) Single

2) Multilevel

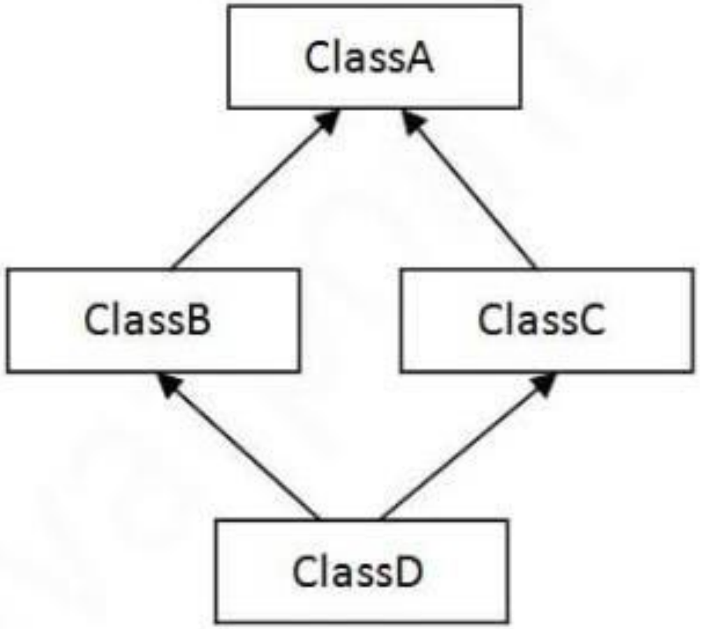
3) Hierarchical

Note: Multiple inheritance is not supported in Java through class.

Inheritance



4) Multiple

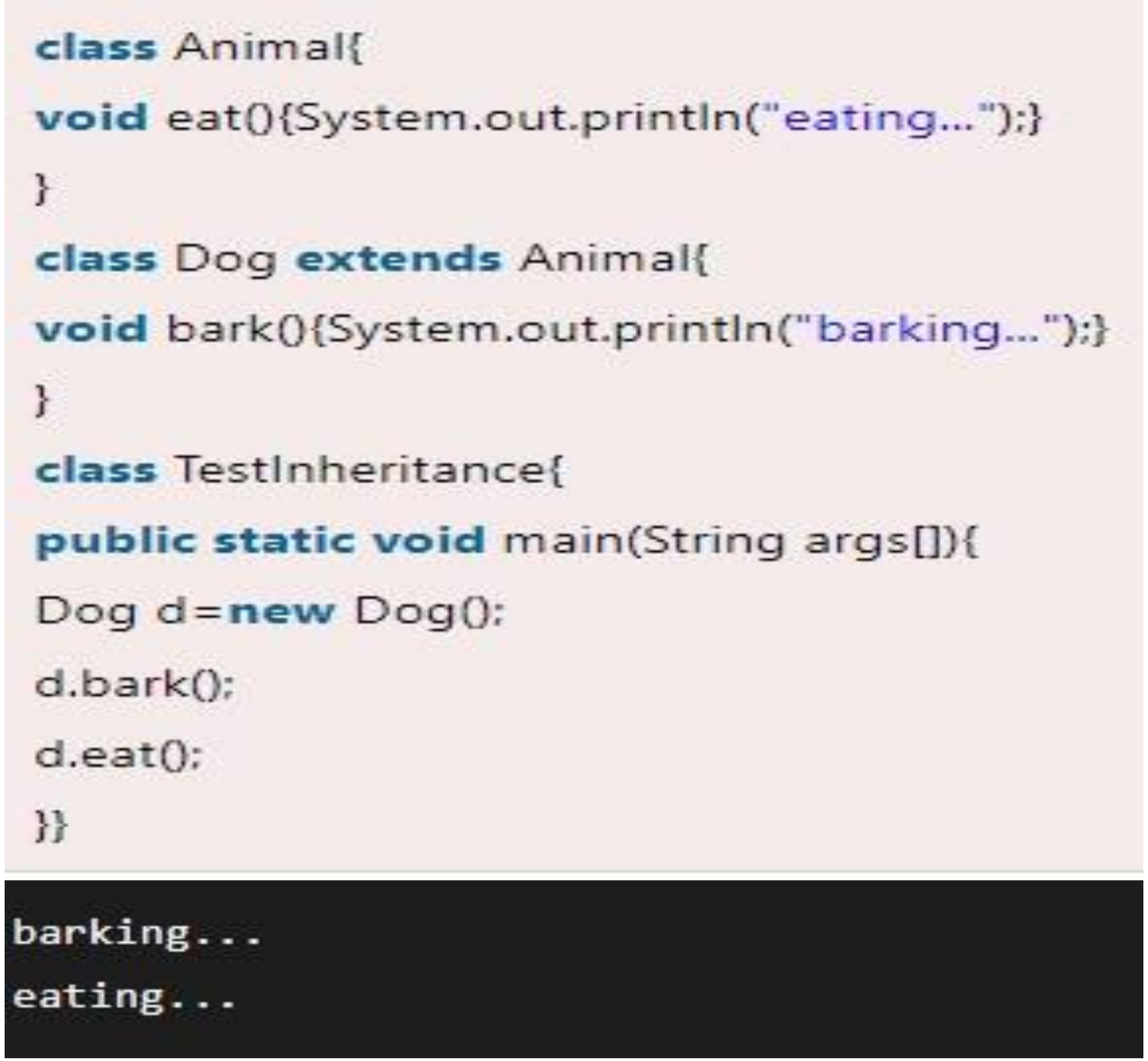


5) Hybrid

When one class inherits multiple classes, it is known as multiple inheritance.

Inheritance

Single Inheritance Example



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

```
barking...
eating...
```

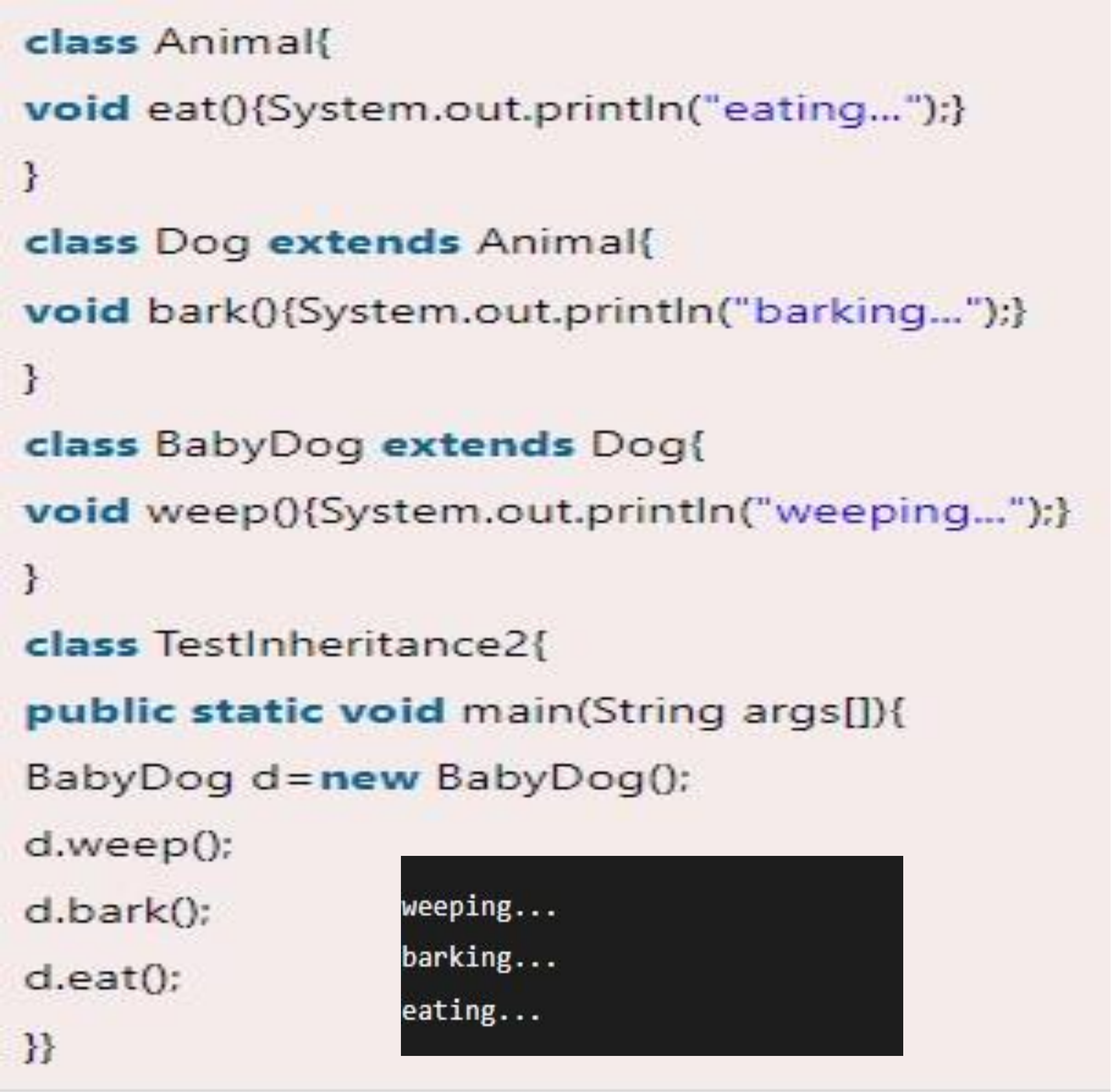
Inheritance

- When a class inherits another class, it is known as a *single inheritance.*
- In the example given, Dog class inherits the Animal class, so there is the single inheritance.

Inheritance

Multilevel Inheritance Example

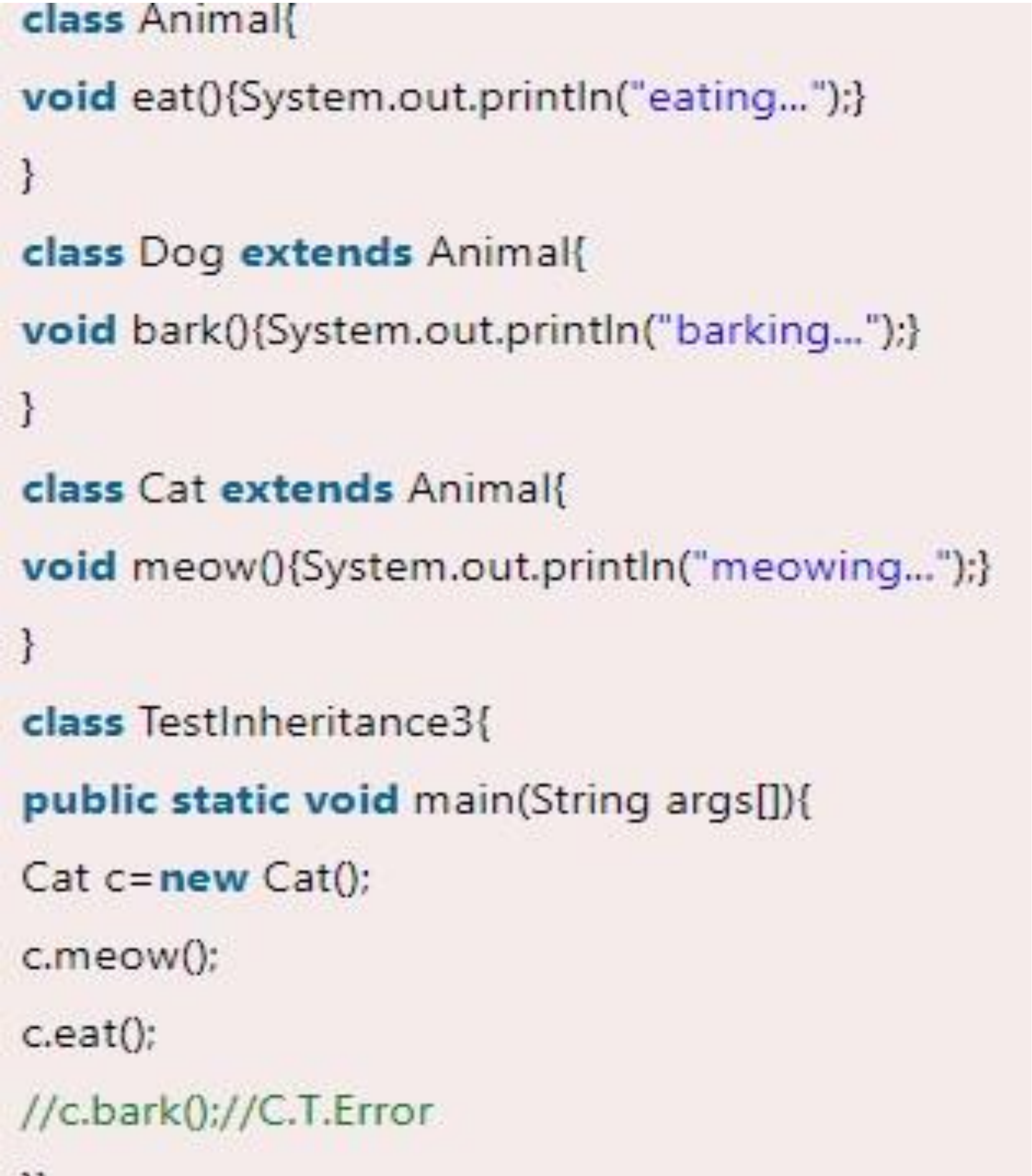
- When there is a chain of inheritance, it is known as *multilevel inheritance*.
- As you can see in the example given , BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

```
weeping...
barking...
eating...
```

Inherita



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
```

- Hierarchical Inheritance
- When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

Why multiple inheritance is not supported in java?

Read and give example with code

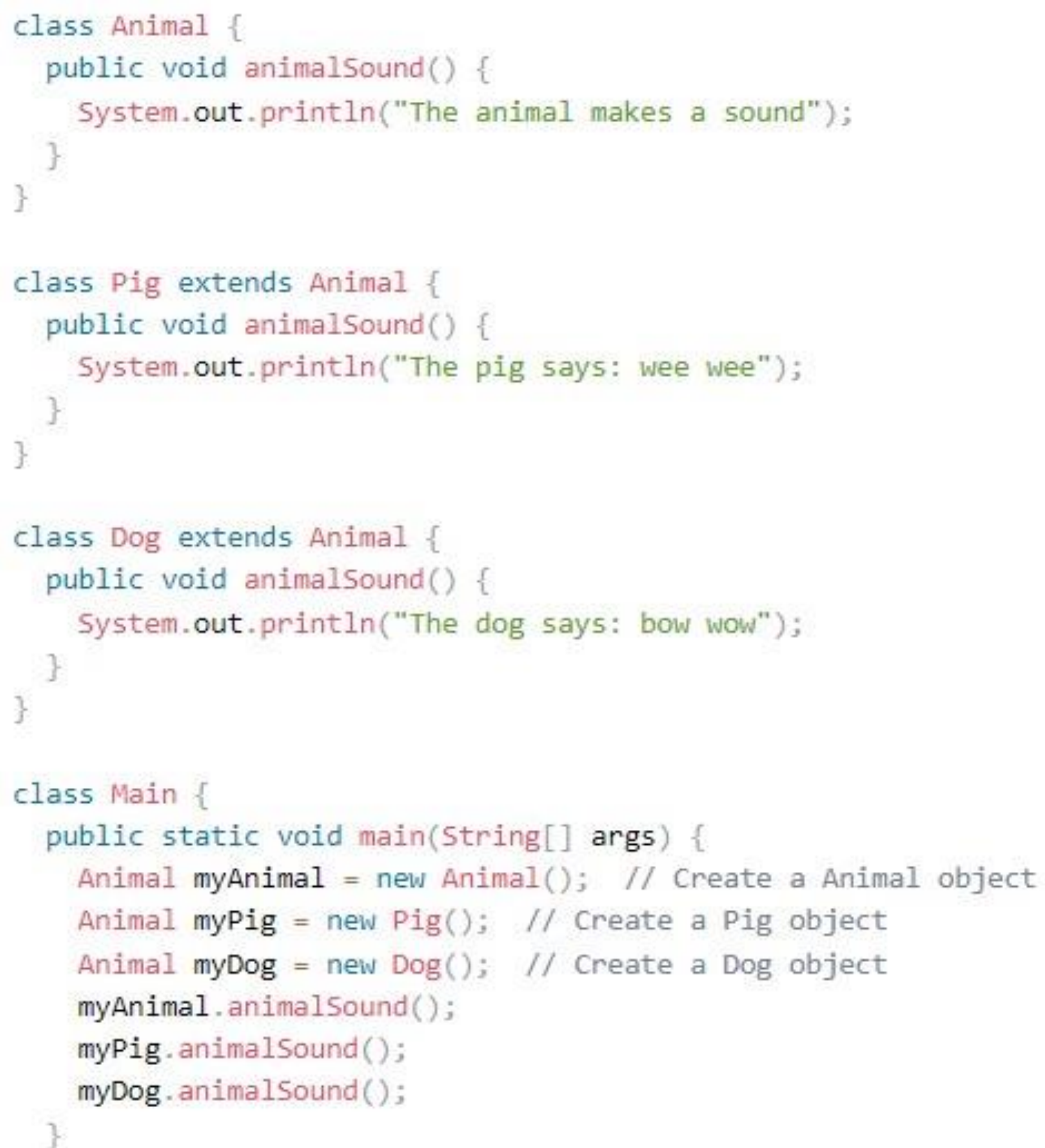
polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

- Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- For example, think of a superclass called `Animal` that has a method called `animalSound()`.
- Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

nd When To Use "Inheritance" and



```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
```

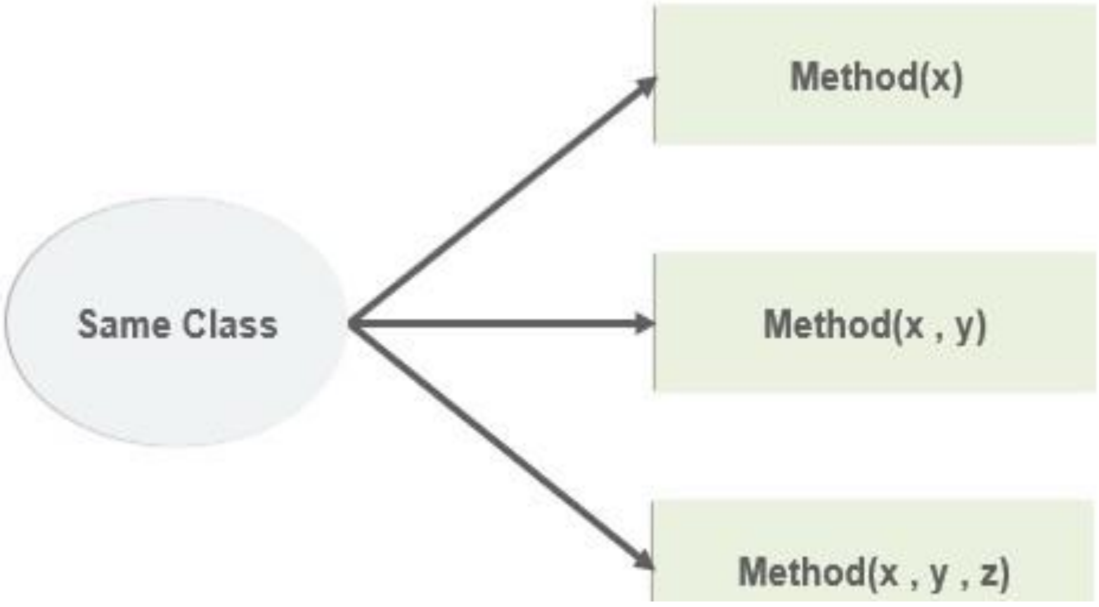
"Polymorphism"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Method Overriding and Overloading

Method Overloading

- Method overloading allows the method to have the same name which differs on the basis of arguments or the argument types.
- It can be related to compile-time polymorphism. Following are a few pointers that we have to keep in mind while overloading methods in Java.



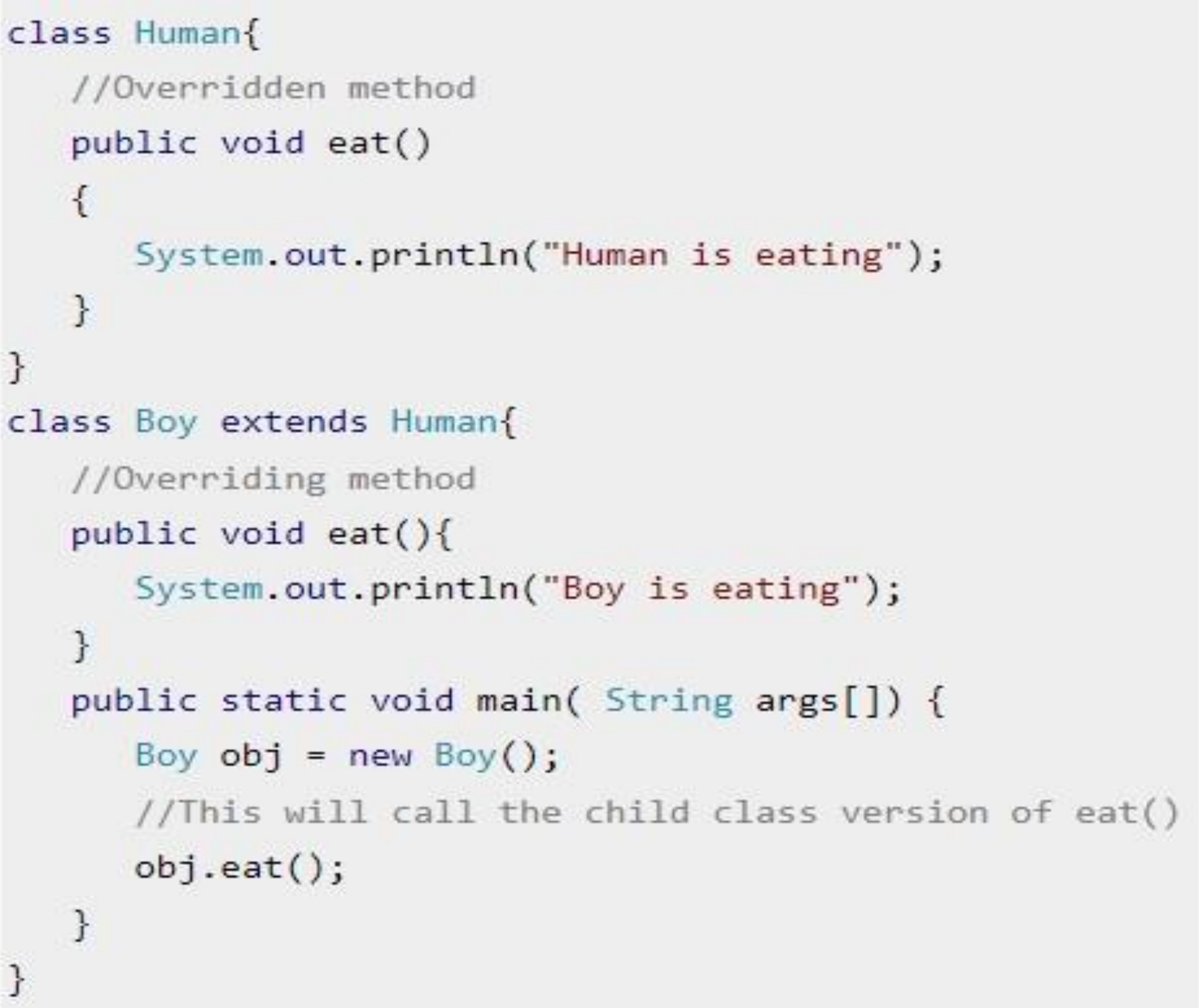
```
public class Div { public int div(int a ,
int b) { return (a / b); }

 public int div(int a , int b ,
int c){ return ((a + b ) / c);
                        }

              public static void main(String args[]){
Div ob = new Div(); ob.div(10 , 2);
ob.div(10, 2 , 3);
                                        }  }
```

Method Overriding

- Method **Overriding** occurs when two methods have the same method name and parameters. One of the methods is in the parent class, and the other is in the child class.

- Overriding allows a child class to provide the specific implementation of a method that is *already* present in its parent class.

Method Overriding



```
class Human{
   //Overridden method
   public void eat()
   {
      System.out.println("Human is eating");
   }
}
class Boy extends Human{
   //Overriding method
   public void eat(){
      System.out.println("Boy is eating");
   }
   public static void main( String args[]) {
      Boy obj = new Boy();
      //This will call the child class version of eat()
      obj.eat();
   }
}
```

Output

```
Boy is eating
```

Method Overriding

Advantage of method overriding

- The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.
- This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Super keyword in java

- The `super` keyword refers to superclass (parent) objects.
- It is used to call superclass methods, and to access the superclass constructor.
- The most common use of the `super` keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.
- To understand the `super` keyword, you should have try to understanding of Inheritance and Polymorphism.

Example

```
class Animal {
// Superclass (parent)
public void animalSound() {
  System.out.println("The animal makes a sound");
    }
  } class Dog extends
 Animal {
  // Subclass (child) public void animalSound() {
 super.animalSound(); // Call the superclass method
 System.out.println("The dog says: bow wow"); } }
   public class Main {
 public static void main(String args[]) {
   Animal  myDog  =  new  Dog();  //  Create  a  Dog  object
myDog.animalSound(); // Call the method on the Dog object } }
```

Abstract Class in Java

- A class that is declared using "**abstract**" keyword is known as abstract class.

- It can have abstract methods(methods without body) as well as concrete methods (regular methods with body).
- A normal class(non-abstract class) cannot have abstract methods.
- In this guide we will learn what is a abstract class, why we use it and what are the rules that we must remember while working with it in Java.
- An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it. Why?

Why we need an abstract class?

- Lets say we have a class Animal that has a method sound() and the subclasses(see **inheritance**) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.
- So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.
- Since the Animal class has an abstract method, you must need to declare this class abstract.

- Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Abstract class Example

```
  //abstract parent class abstract class Animal{
public abstract void sound(); //abstract method
}
    //Dog class extends Animal class
  public class Dog extends Animal{
  public void sound(){
    System.out.println("Woof"); } public
 static void main(String args[]){
 Animal obj = new Dog(); obj.sound(); }
 }
```

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

```
Output Woof
```

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
   //Declaration using abstract keyword
 Abstract class A{ abstract void myMethod(); //This
   is abstract method
     //This is concrete method with body
    void anotherMethod(){
      //Does something
          }
      }
```

Cont…

- **Note 1:** As we seen in the previous example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.
- A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.
- **Note 2:** Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call nonabstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

- **Note 3:** If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Cont..

Why can't we create the object of an abstract class?

- Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the method to invoke.
- Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.
- **Example to demonstrate that object creation of abstract class is not allowed**

- As discussed previously, we cannot instantiate an abstract class. This program throws a compilation error.

Example:Abstarct class cannot been instantiated

```
abstract class AbstractDemo{
  public void myMethod(){
    System.out.println("Hello"); }
  abstract public void anotherMethod();
  } public class Demo extends
AbstractDemo{ public void
anotherMethod() {
     System.out.print("Abstract method");
       } public static void main(String
    args[]) {
      //error: You can't create object of
it AbstractDemo obj=new AbstractDemo();
obj.anotherMethod(); } }
```

```
Unresolved compilation problem: Cannot instantiate the type AbstractDemo
```

Cont…

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

Abstract class vs Concrete class

- A class which is not abstract is referred as **Concrete class**.
- In the above example that we have seen in the beginning of this guide, `Animal` is a abstract class and `Cat`, `Dog` & `Lion` are concrete classes.

Key Points:

- An abstract class has no use unless it is extended by some other class.
- If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class.
- It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.

- It can have non-abstract method (concrete) as well.

Cont…

For now lets just see some basics and example of abstract method.

1) Abstract method has no body.

2) Always end the declaration with a **semicolon**(;).

3) It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.

4) A class has to be declared abstract to have abstract methods.

- **Note:** The class which is extending abstract class must override all the abstract methods.

Example (Abstract class and method)

```
abstract class MyClass{
   public void disp(){
    System.out.println("Concrete method of parent class");  }
    abstract public void disp2();
 } class Demo extends MyClass{ /* Must Override this
method while extending * MyClas */ public void disp2()
{
   System.out.println("overriding abstract method");
  }
  public static void main(String args[]){
    Demo obj = new Demo();
         obj.disp2(); }
      }
```

```
Output: overriding abstract method
```

Interface in java

- In the last tutorial we discussed **abstract class** which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction.
- Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user(See: **Abstraction**). In this topic, we will cover **what is an interface in java**, why we use it and what are rules that we must follow while using interfaces in **Java Programming**.

What is an interface in Java?

- **Interface** looks like a class but it is not a class.
- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method

signature, no body, see: **Java abstract method**). Also, the variables declared in an interface are public, static & final by default.

What is the use of interface in Java?

- As mentioned above they are used for full abstraction.
- Since methods in interfaces do not have body, they have to be implemented by the class before you can access them.
- The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Syntax:

Interfaces are declared by specifying a keyword “interface”.

Example

```
interface MyInterface { /* All the
methods are public abstract by default
* As you see they have no body */
public void method1(); public void
method2();
}
```

Example (Interface in Java)

- This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.
- Class implements interface but an interface extends another interface

```
interface MyInterface {
   /* compiler will treat them as: * public abstract void method1();
    * public abstract void method2();
 */ public void method1(); public void
 method2(); }
class Demo implements MyInterface {
/* This class must   have to implement both the abstract methods * else
you will get compilation error */
public void method1() { System.out.println("implementation of method1"); }
public void method2() { System.out.println("implementation of method2"); }
public static void main(String arg[]) { MyInterface obj = new Demo();
obj.method1(); } }
```

Interface and Inheritance

- As discussed previously, an interface can not implement another interface. It has to extend the other interface.
- See the below example where we have two interfaces Inf1 and Inf2.
- Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

```
interface Inf1{ public
  void method1();
  } interface Inf2 extends
 Inf1 { public void
 method2();
  } public class Demo implements
Inf2{
/* Even though this class is only implementing the *
interface Inf2, it has to implement all the methods * of
Inf1 as well because the interface Inf2 extends Inf1 */
public void method1(){
 System.out.println("method1"); }
public void method2(){
 System.out.println("method2"); } public
static void main(String args[]){
 Inf2 obj = new Demo();
obj.method2(); } }
```

Cont…

In this program, the class `Demo` only implements interface `Inf2`, however it has to provide the implementation of all the methods of interface `Inf1` as well, because interface Inf2 extends Inf1.

Advantages of interface in java:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

Chapter Four
Exception Handling in Java

- **What is Exception in Java?**

  Exception is an abnormal condition

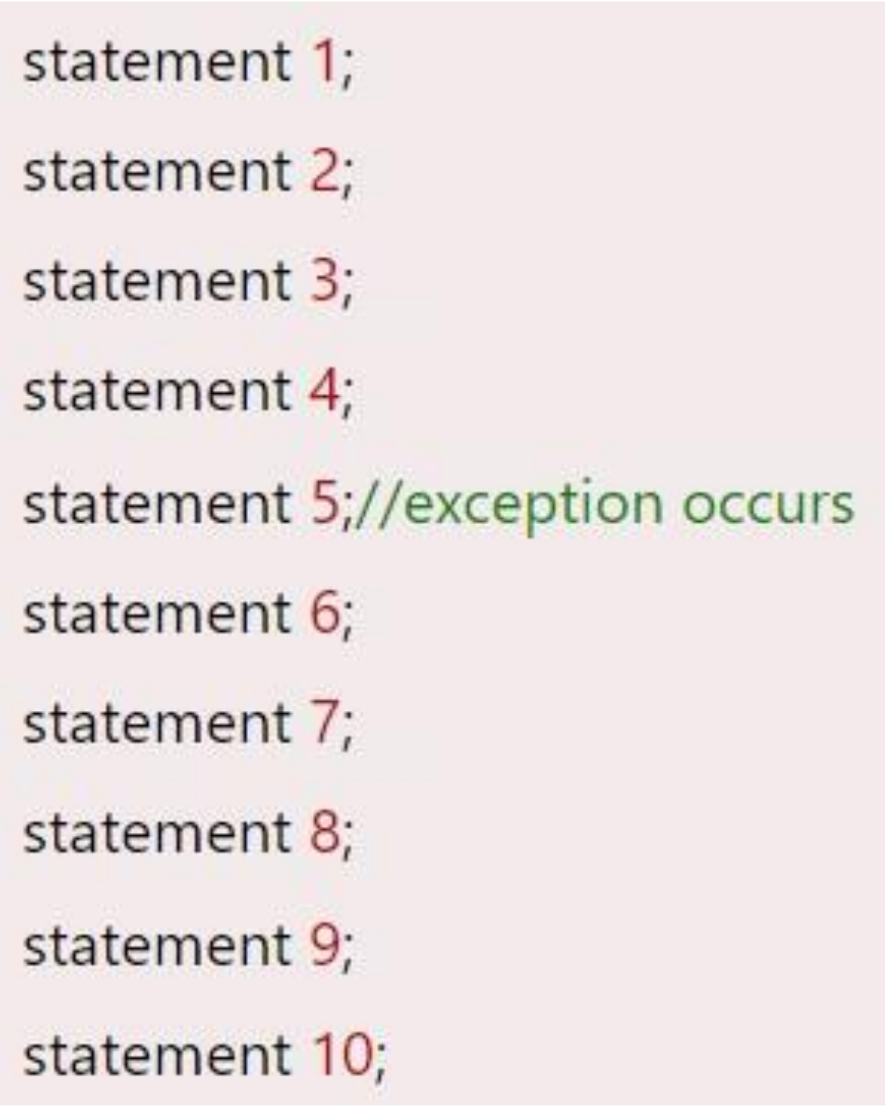
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Now, Exception Handling is a mechanism to handle runtime errors such as ArithemticException, ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application**.

- An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

- Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed.
- However, when we perform exception handling, the rest of the statements will be executed.
- That is why we use exception handling in Java.



```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Do you know?

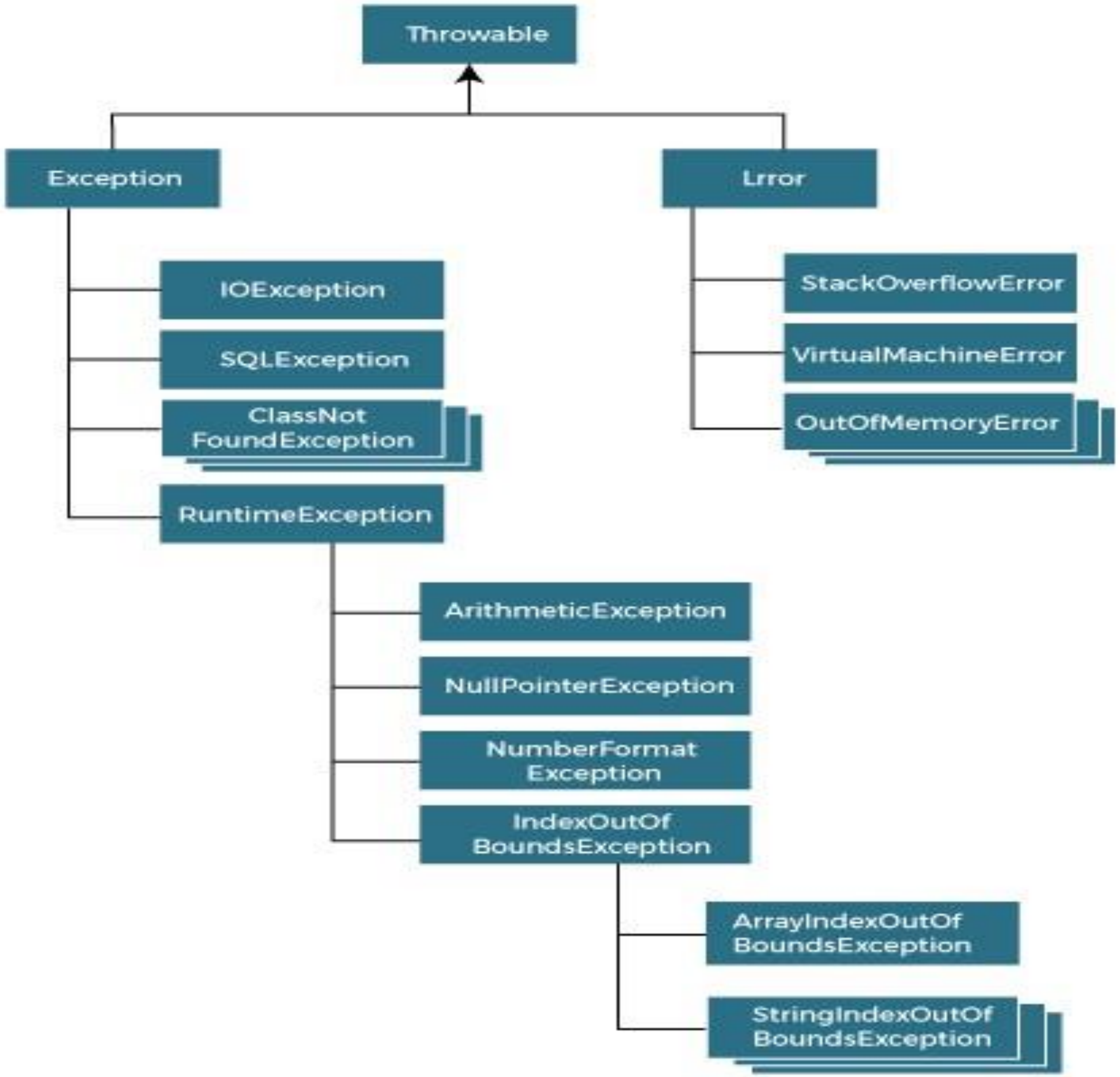
- What is the difference between checked and unchecked exceptions?

- What happens behind the code int data=50/0;?
- Why use multiple catch block?
- Is there any possibility when the finally block is not executed?
- What is exception propagation?
- What is the difference between the throw and throws keyword?
- What are the 4 rules for using exception handling with method overriding?

Hierarchy of Java Exception classes

- The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error.
- The hierarchy of Java Exception classes is given below:

Types of Java Exceptions

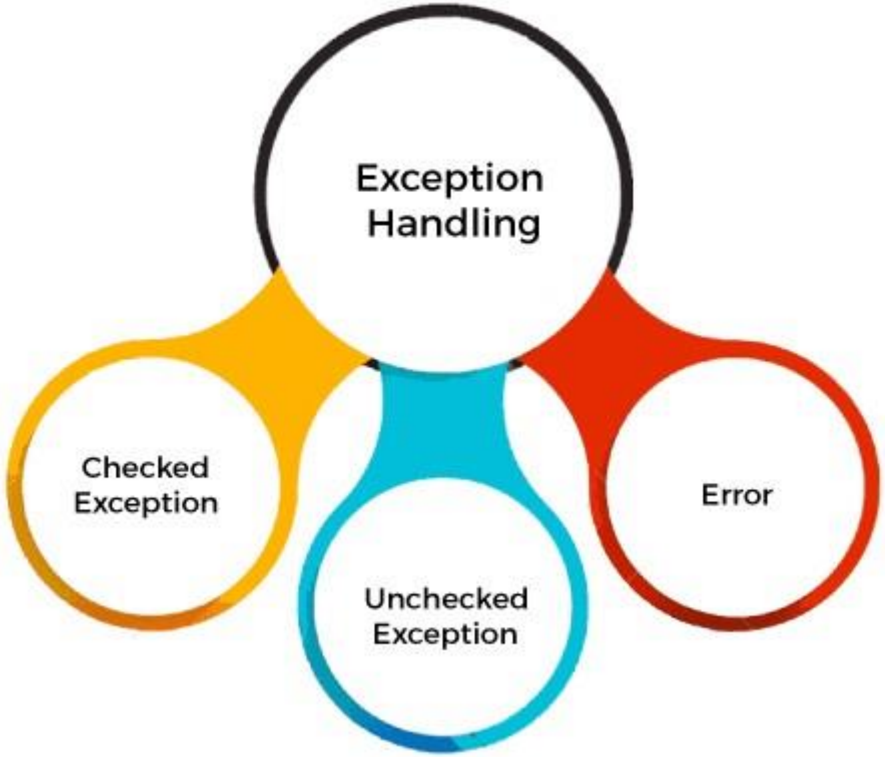


- There aremainlytwo types of exceptions: checked and unchecked.
- An error is considered as the unchecked exception.
- However, according to Oracle, there are three types of exceptions namely:

Checked Exception

Unchecked Exception

Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.
- For example, IOException, SQLException, etc.
- Checked exceptions are checked at compile-time.

2) Unchecked Exception

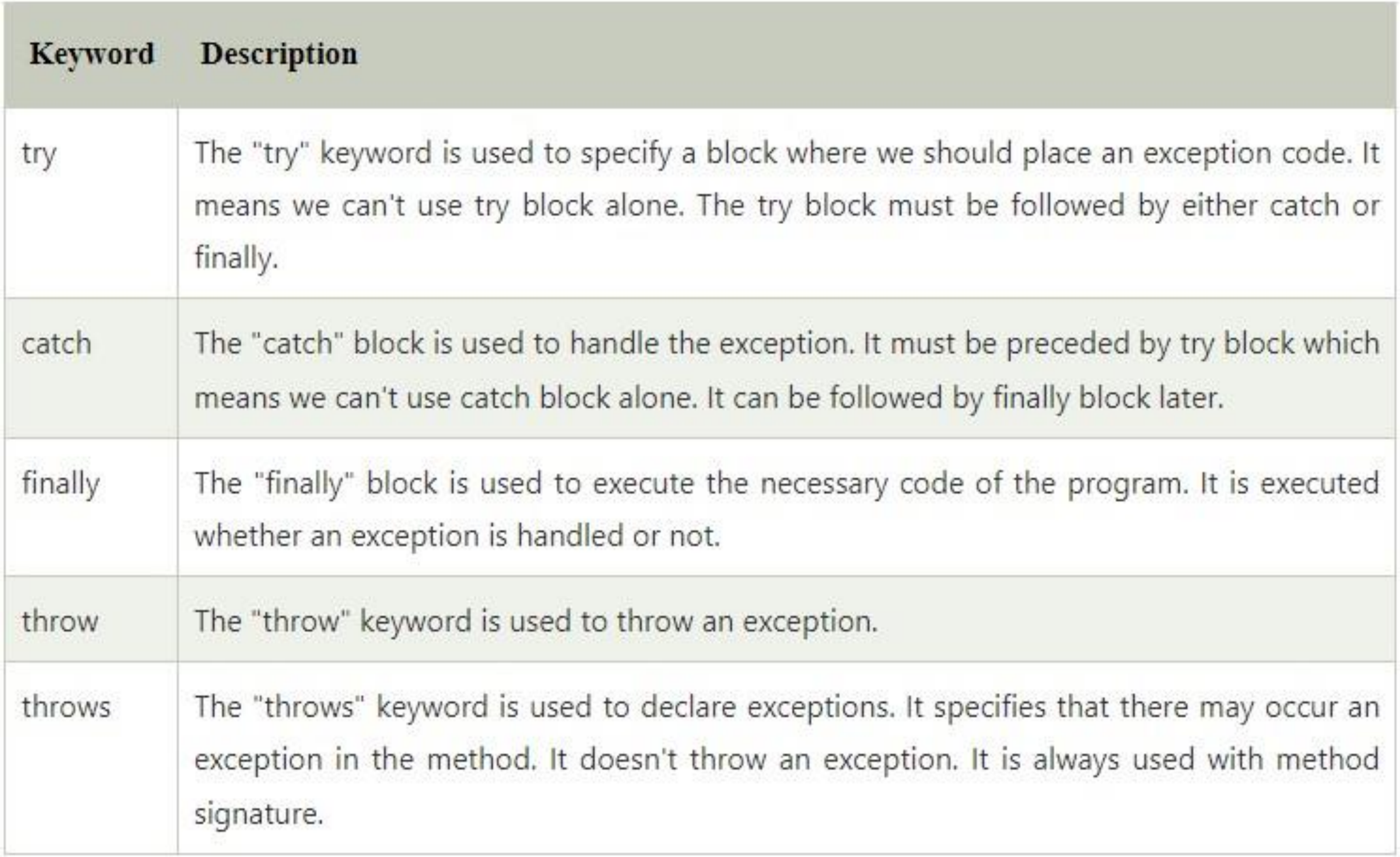
- The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error: Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

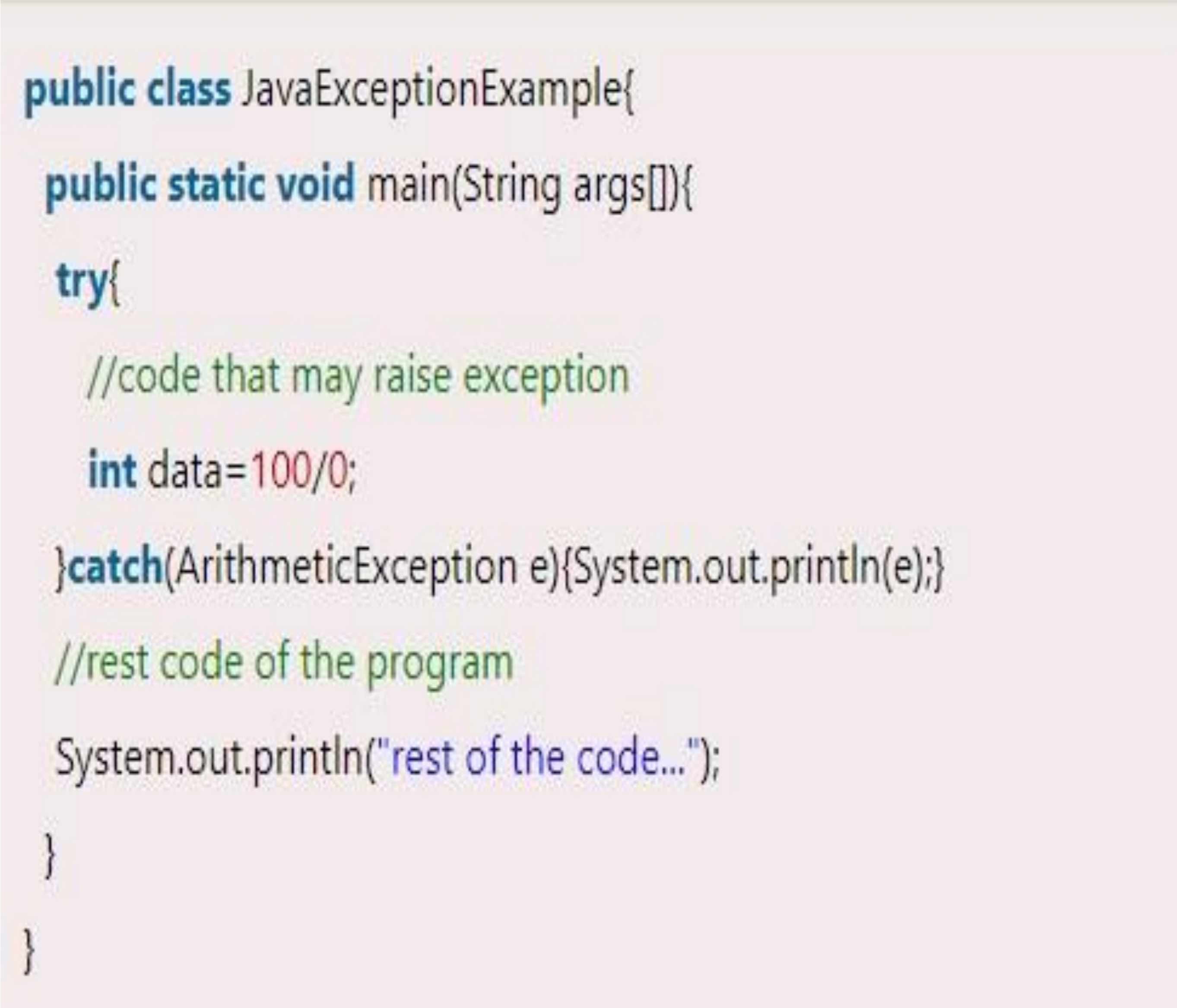
- Java provides five keywords that are used to handle the exception.
- The following table describes each.

Java Exception Handling Example



Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

- Let's see an example of Java Exception Handling in which we are using a trycatch statement to handle the exception.
- As you see in the example, 100/0 raises an ArithmeticException which is handled by a try-catch block.
- There are given some scenarios where unchecked exceptions may occur. They are as follows:



```
public class JavaExceptionExample{
 public static void main(String args[]){
  try{
     //code that may raise exception
     int data=100/0;
  }catch(ArithmeticException e){System.out.println(e);}
  //rest code of the program
  System.out.println("rest of the code...");
 }
}
```

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable,performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

- If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string

- variable that has characters; converting this variable into digit will cause NumberFormatException. String s="abc";

```
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

- When an array exceeds to it's size, the arrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException.
- Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

- Java **try** block is used to enclose the code that might throw an exception.
- It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

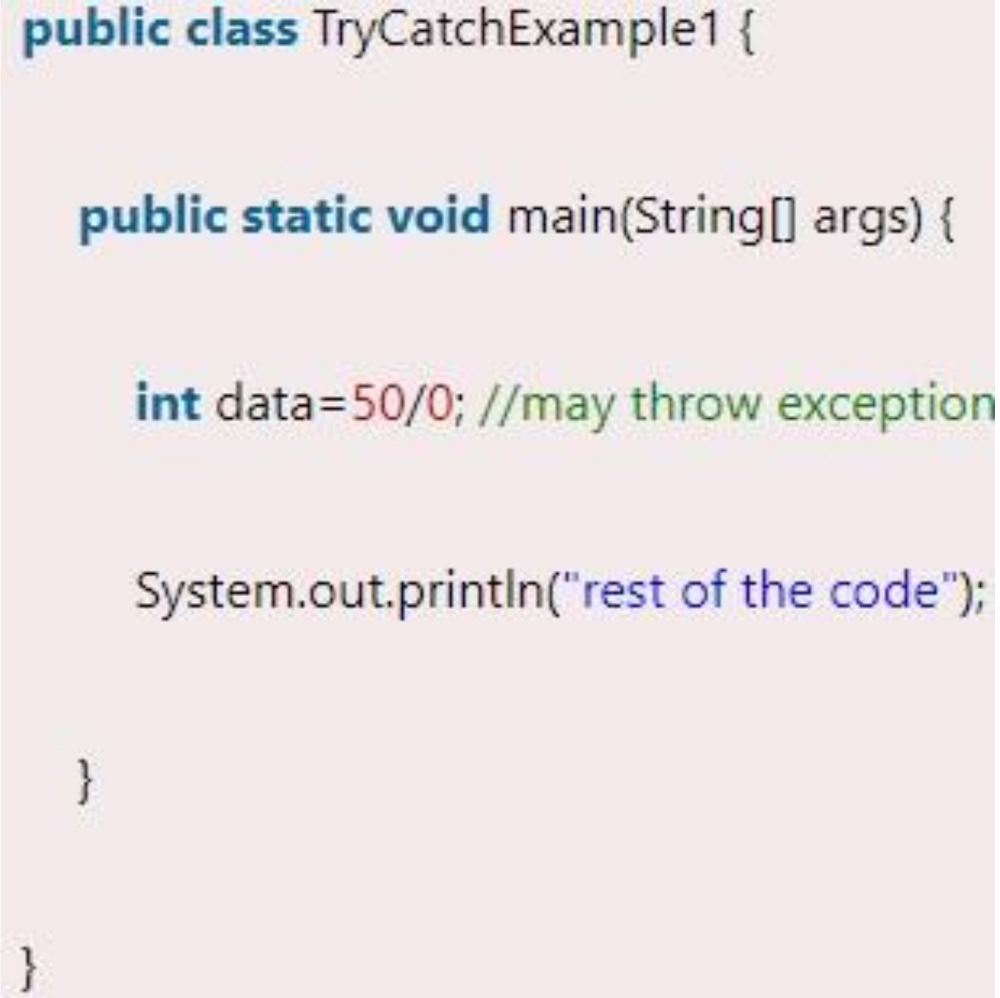
```
try{

//code that may throw an exception
}catch(Exception_class_Name
ref){}
```

Syntax of try-finally block

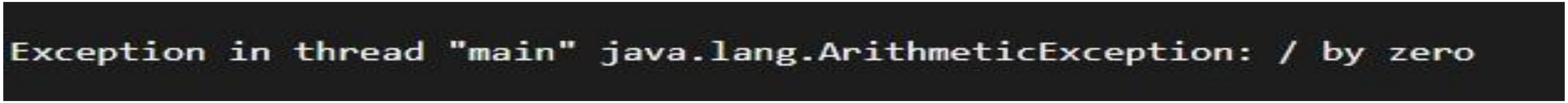
```
try{
    //code that may throw an exception

}finally{}
```

Example:TryCatchExample1

```
public class TryCatchExample1 {
  public static void main(String[] args) { int
     data=50/0; //may throw exception
     System.out.println("rest of the code");
    }
 }
```

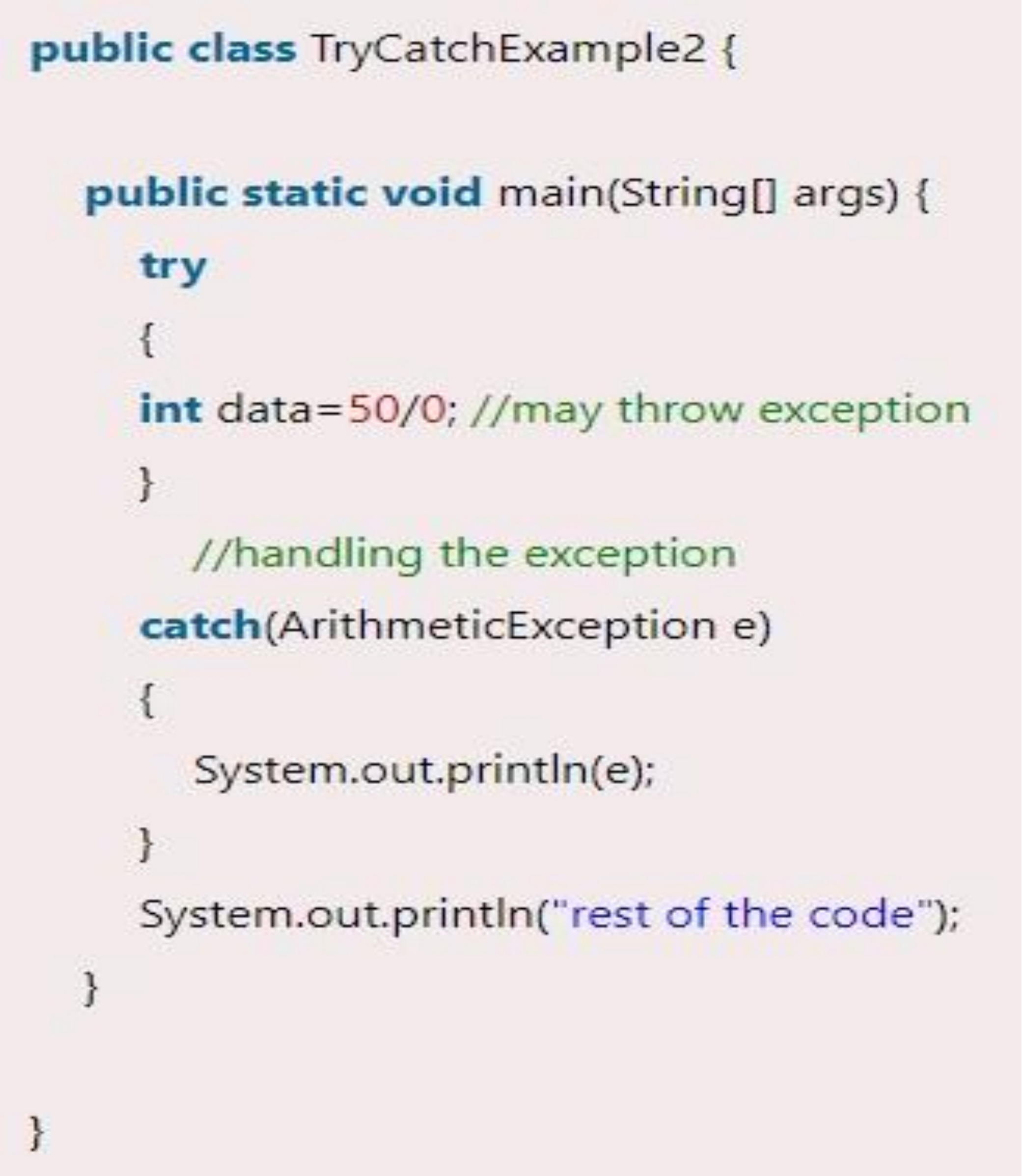


```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```



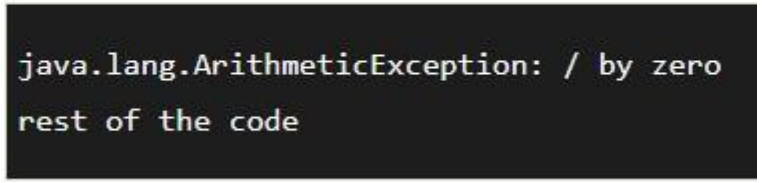
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Solution by exception handling



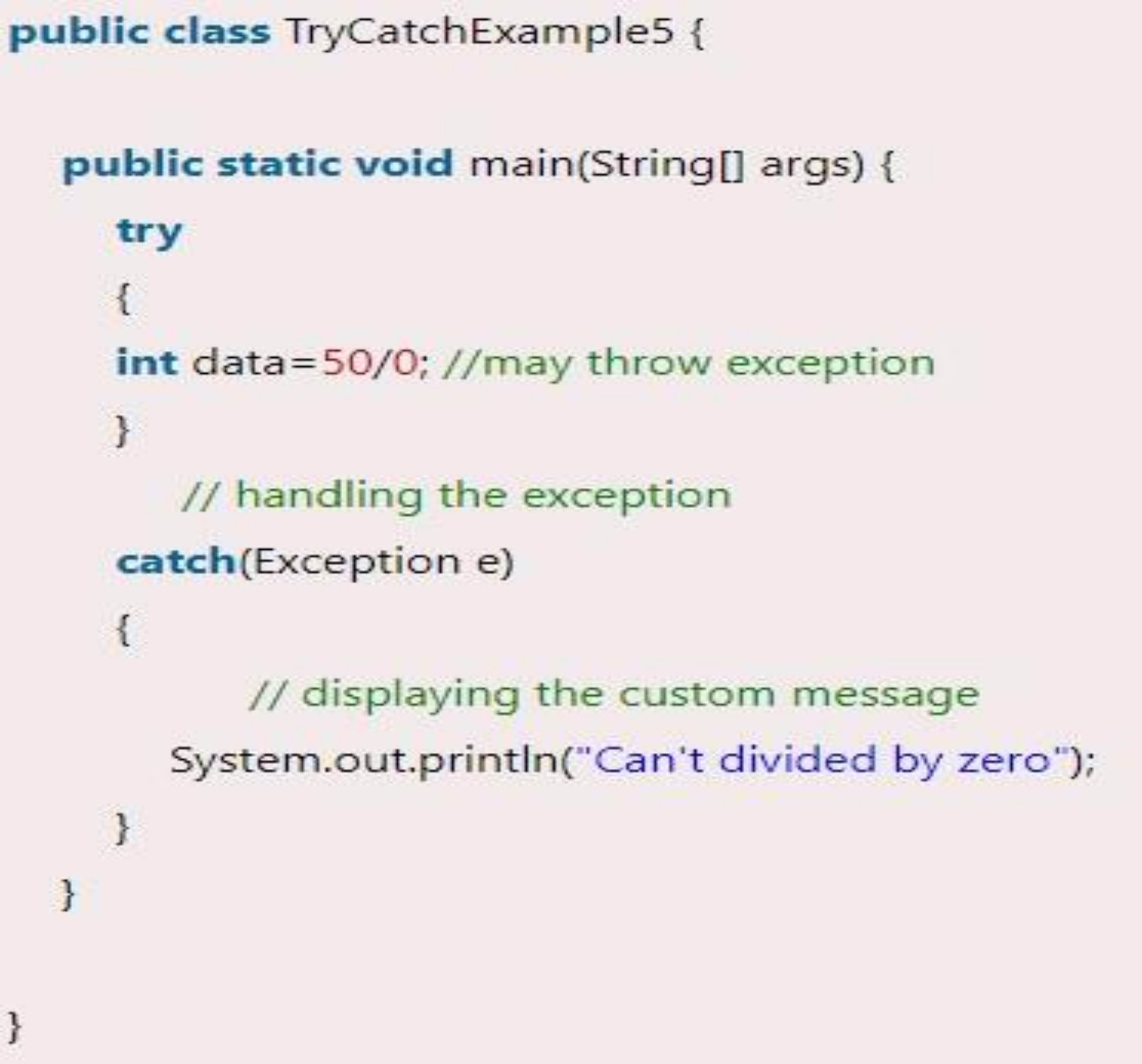
```
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:



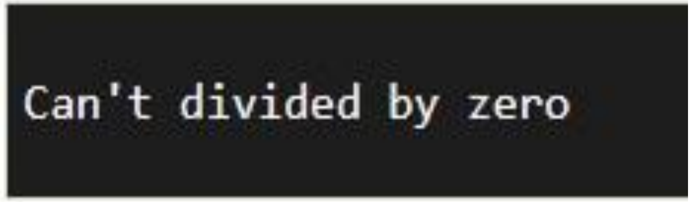
```
java.lang.ArithmeticException: / by zero
rest of the code
```

- Let's see the solution of the previous problem by a java trycatch block.
- **TryCatchExample2**

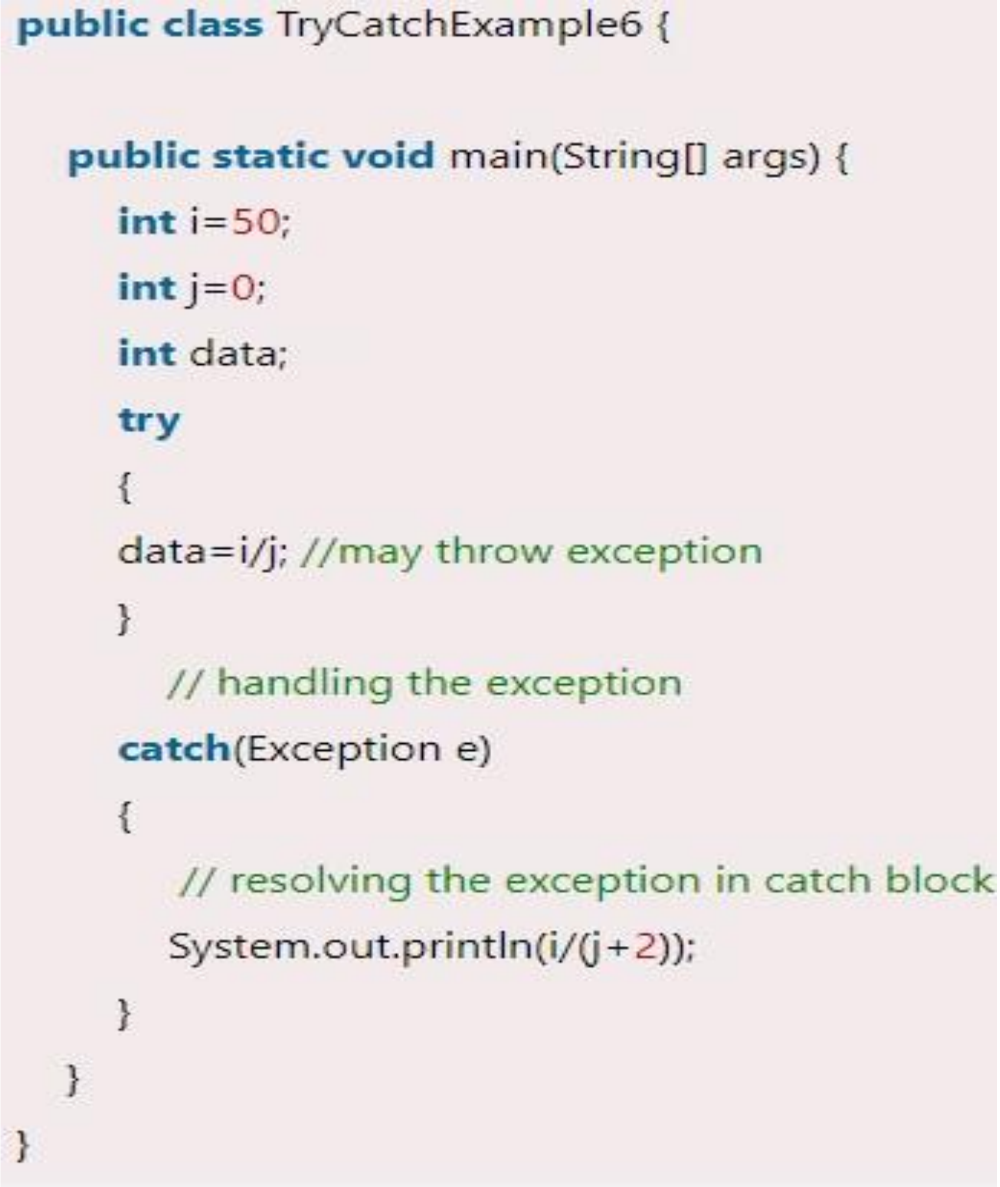


```
public class TryCatchExample5 {


  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
    }
        // handling the exception
    catch(Exception e)
    {
             // displaying the custom message
      System.out.println("Can't divided by zero");
    }
  }


}
```

Output:



```
Can't divided by zero
```



```java
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output: 25

Read or refer This link to know more about Exception handling in java

- https://www.javatpoint.com/try-catch-block