Unit 1 & 2

Object-Oriented Programming (OOP)

Object-oriented programming is the successor of procedural (structural) programming. Procedural programming describes programs as groups of reusable code units (procedures) which define input and output parameters. Procedural programs consist of procedures, which invoke each other. The problem with procedural programming is that code reusability is hard and limited – only procedures can be reused and it is hard to make them generic and flexible. There is no easy way to work with abstract data structures with different implementations.

The object-oriented approach relies on the paradigm that each and every program works with data that describes entities (objects or events) from real life. For example: accounting software systems work with invoices, items, warehouses, availabilities, sale orders, etc.

This is how objects came to be. They describe characteristics (properties) and behavior (methods) of such real life entities.

The main advantages and goals of OOP are to make complex software faster to develop and easier to maintain. OOP enables the easy reuse of code by applying simple and widely accepted rules (principles). Let's check them out.

Fundamental Principles of OOP

What is Object Oriented Programming (OOP)?

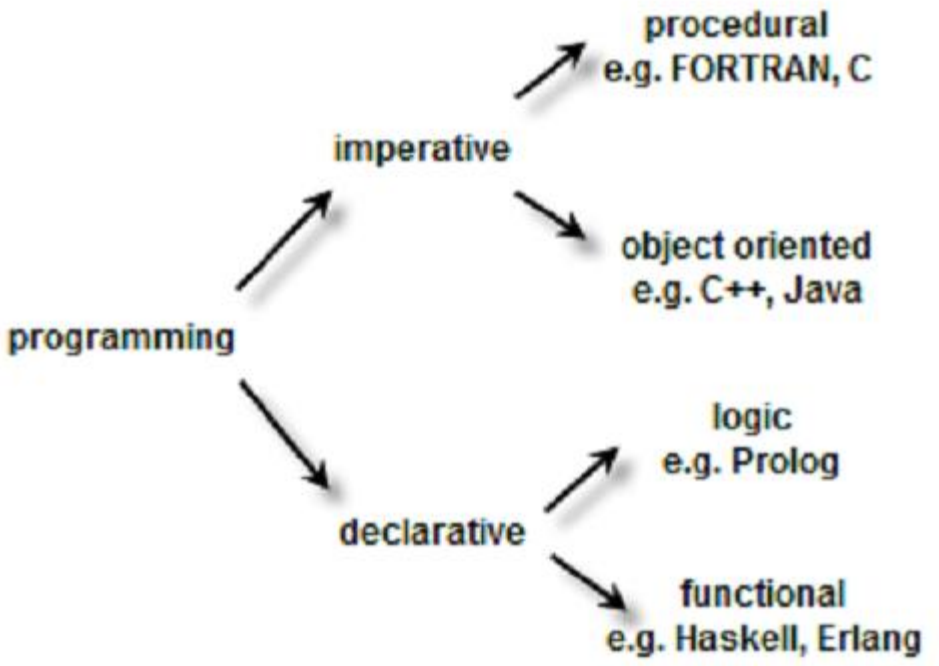
OOP is a high-level programming language where a program is divided into small chunks called objects using the object-oriented model, hence the name. This paradigm is based on objects and classes.

- **Object** – An object is basically a self-contained entity that accumulates both data and procedures to manipulate the data. Objects are merely instances of classes.
- **Class** – A class, in simple terms, is a blueprint of an object which defines all the common properties of one or more objects that are associated with it. A class can be used to define multiple objects within a program.

  In order for a programming language to be object-oriented, it has to enable working with classes and objects as well as the implementation and use of the fundamental object-oriented principles and concepts: inheritance, abstraction, encapsulation and polymorphism. Let's summarize each of these fundamental principles of OOP:

There are four main OOP concepts in Java. These are:

- **Abstraction**. Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it. In Java, abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.
- **Encapsulation.** This is the practice of keeping fields within a class private, then providing access to them via public methods. It's a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.
- **Inheritance.** This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.
- **Polymorphism**. This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is method overloading. That's when different meanings are implied by the code itself. The other form is method overriding. That's when the different meanings are implied by the values of the supplied variables. See more on this below.



What is Procedure Oriented Programming (POP)?

POP follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do. The program is divided into small parts called functions and then it follows a series of computational steps to be carried out in order.

It follows a top-down approach to actually solve a problem, hence the name. Procedures correspond to functions and each function has its own purpose. Dividing the program into functions is the key to procedural programming. So a number of different functions are written in order to accomplish the tasks.

Difference between POP and OOP

OOP	POP
OOP takes a bottom-up approach in designing a program.	POP follows a top-down approach.
Program is divided into objects depending on the problem.	Program is divided into small chunks based on the functions.
Each object controls its own data.	Each function contains different data.
Focuses on security of the data irrespective of the algorithm.	Follows a systematic approach to solve the problem.
The main priority is data rather than functions in a program.	Functions are more important than data in a program.
ons of the objects are linked via message	Different parts of a program are interconnected via parameter passing.
Data hiding is possible in OOP.	No easy way for data hiding.
Inheritance is allowed in OOP.	No such concept of inheritance in POP.
Operator overloading is allowed.	Operator overloading is not allowed.
C++, Java.	Pascal, Fortran.

Introduction of eclipse (IDE) for developing programs in Java

Eclipse is an integrated development environment (IDE) for Java and other programming languages like C, C++, PHP, and Ruby etc. Development environment provided by Eclipse includes the Eclipse Java development tools (JDT) for Java, Eclipse CDT for C/C++, and Eclipse PDT for PHP.

The Eclipse platform which provides the foundation for the Eclipse IDE is composed of plug-ins. Developed using Java, the Eclipse platform can be used to develop rich client applications, integrated development environments and other tools. he Java Development Tools (JDT) project provides a plug-in that allows Eclipse to be used as a Java IDE.

How to download and install Eclipse IDE

To install on windows, you need a tool that can extract the contents of a zip file.

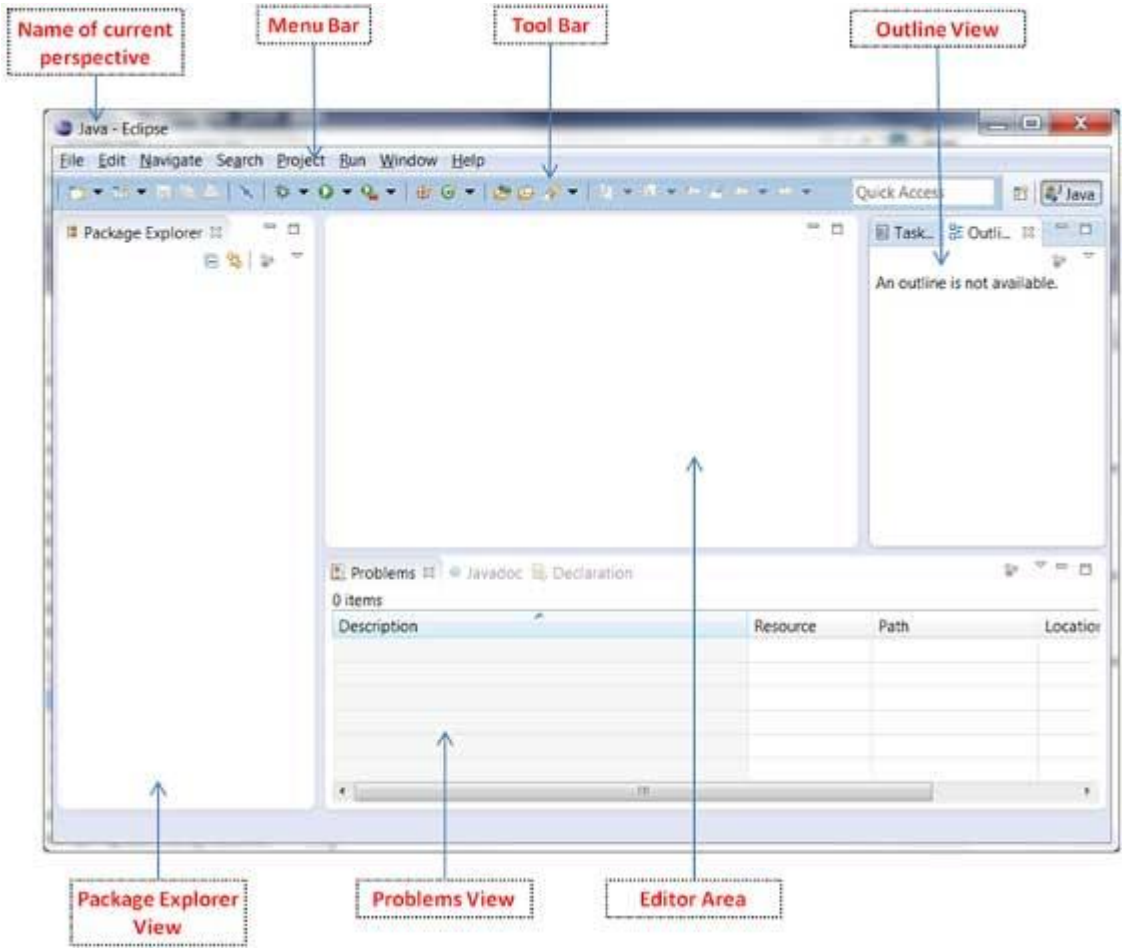
On the windows platform, if you extracted the contents of the zip file to c:\, then you can start eclipse by using c:\eclipse\eclipse.exe.

When eclipse starts up for the first time it prompts you for the location of the workspace folder. All your data will be stored in the workspace folder.

The major visible parts of an eclipse window are −

- Views
- Editors (all appear in one editor area)
- Menu Bar
- Toolbar

An eclipse perspective is the name given to an initial collection and arrangement of views and an editor area. The default perspective is called java. An eclipse window can have multiple perspectives open in it but only one perspective can be active at any point of time. A user can switch between open perspectives or open a new perspective. A perspective controls what appears in some menus and tool bars.



A perspective has only one editor area in which multiple editors can be open. The editor area is usually surrounded by multiple views. In general, editors are used to edit the project data and views are used to view the project metadata. For example the package explorer shows the java files in the project and the java editor is used to edit a java file.

The eclipse window can contain multiple editors and views but only one of them is active at any given point of time. The title bar of the active editor or view looks different from all the others.

The UI elements on the menu bar and tool bar represent commands that can be triggered by an end user.

The typical menus available on the menu bar of an Eclipse window are −

- File menu
- Edit menu
- Navigate menu
- Search menu
- Project menu
- Run menu
- Window menu
- Help menu

Classes and Objects

Unit 3

- **Creation, accessing class members**
- **Private Vs Public Vs Protected Vs Default**
- **Constructors**
- **Object & Object Reference**

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

1.1 Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

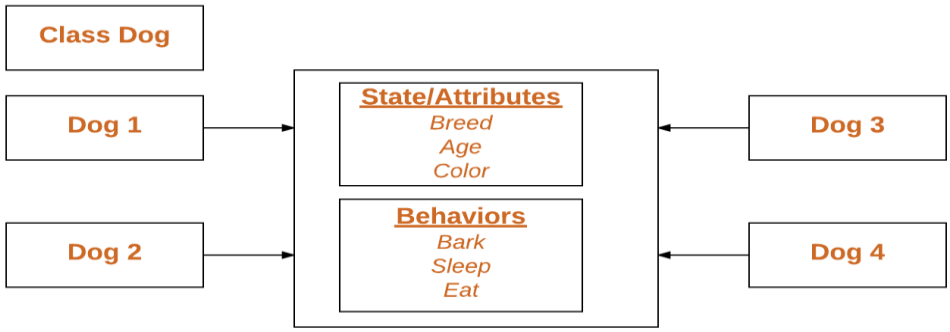
1. **Modifiers**: A class can be public or has default access (Refer this for details).
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Super class(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **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.
5. **Body:** The class body surrounded by braces, { }.

1.2 Object

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State** : It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

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.
Example :



1.3 How to define a class

Here's how a class is defined in Java:

class className{

Variables

methods

}

Example

```
class Lamp {

  private boolean isOn;

  public void turnOn()
  {    isOn = true;  }

  public void turnOff()
  {   isOn = false;  }
}
```

The class has one instance variable isOn and two methods turnOn() and turnoff(). These variables and methods defined within a class are called members of the class. Notice two keywords, private and public in the above program. These are access modifiers which will be discussed in detail in later chapters. For now, just remember:

- The private keyword makes instance variables and methods private which can be accessed only from inside the same class.
- The public keyword makes instance variables and methods public which can be accessed from outside of the class.

  In the above program isOn is a private variable and turnOn() and turnoff() are public methods.

1.4 How to create objects of class

When class is defined, only the specification for the object is defined; no memory or storage is allocated.To access members defined within the class, you need to create objects.

To create objects of a class, the new keyword is used as follows:

```
Lamp L1 = new Lamp();    // create L1 object of Lamp class

Lamp L2 = new Lamp();   // create L2 object of Lamp class
```

1.5 How to access members of a class using object

You can access members (call methods and access instance variables) by using Dot operator. For example,

L1.turnOn();

This statement calls turn On() method inside Lamp class for L1 object.

1.6 Private Vs Public Vs Protected Vs Default

Public members:

If we declare a method as public, then we can access that member from anywhere but corresponding class should be public.

Default members:

- If a member is declared as default, then we can access that member only within current package & we can't access from outside of package.
- Default access also known as package level access.

Private members:

- If a member is declared as private, then we can access that member only within the current class.
- Abstract methods should be visible to child class, private methods will not be visible. Hence private abstract combination is illegal for methods.

Protected members:

- Most understood modifier in Java.
- If a member is declared as protected, then we can access that member within current package anywhere, but outside the package only in child class.
- Within current package, we can access protected members either by parent reference or by child reference.
- Outside package, we can access protected members only by giving child reference.

Visibility	private	<default>	protected	public
Within the same class	S	S	S	S
From child class of same package	N	S	S	S
From non- child class of same package	N	S	S	S
From child class of outside package	N	N	S (we should use only child class reference)	S
From non-child class of outside package	N	N	N	S

1.7 Constructors in JAVA

What is a Constructor?

A constructor is similar to a method (but not actually a method) that is invoked automatically when an object is instantiated.

Java compiler distinguish between a method and a constructor by its name and return type. In Java, a constructor has same name as that of the class, and doesn't return any value.

```
class Test
 {
   Test()
{       // constructor body    }
}
```

Here, Test() is a constructor; it has same name as that of the class and doesn't have a return type.

```
class Test
{
   void Test()
   {       // method body    }
```

}

Here, Test() has same name as that of the class. However, it has a return type void. Hence, it's a method not a constructor.

If you do not create constructors yourself, the Java compiler will automatically create a no-argument constructor during run-time. This constructor is known as default constructor. The default constructor initializes any uninitialized instance variables.

1.8 Object & Object Reference

A *reference* is an address that indicates where an object's variables and methods are stored.

You aren't actually using objects when you assign an object to a variable or pass an object to a method as an argument. You aren't even using copies of the objects. Instead, you're using references to those objects

1. We can assign value of reference variable to another reference variable.
2. Reference Variable is used to store the address of the variable.
3. Assigning Reference will not create distinct copies of Objects.
4. All reference variables are referring to same Object.

Assigning Object Reference Variables does not –

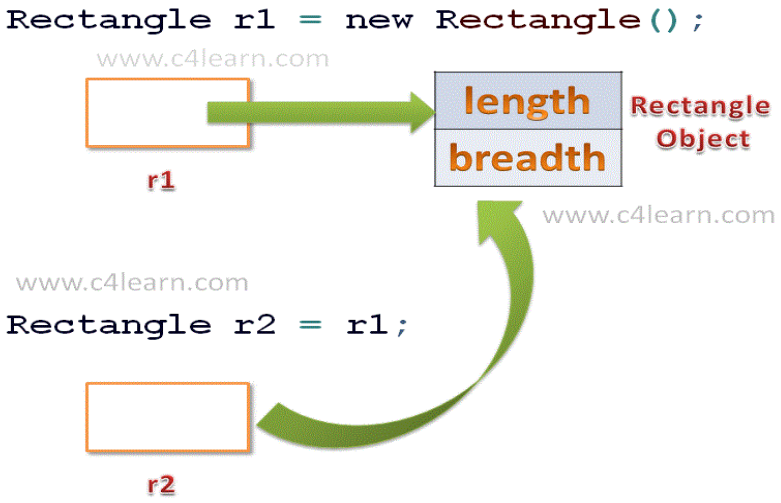
1. Create Distinct Objects.
2. Allocate Memory
3. Create duplicate Copy

Consider This Example –

```
Rectangle r1 = new Rectangle();

Rectangle r2 = r1;
```

- r1 is reference variable which contain the address of Actual Rectangle Object.
- r2 is another reference variable
- r2 is initialized with r1 means – “r1 and r2” both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.



Unit 4

Inheritance

1.1 What is inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class). It is an important part of OOPs (Object Oriented programming system).

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.

Why use inheritance in java

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

How to create a subclass from an existing class

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {

  .....

  .....

}

class Sub extends Super {
```

The super keyword

The super keyword is similar to 'this' keyword used in C++. The super keyword is used to differentiate the members of superclass from the members of subclass, if they have same names. It is also used to invoke the superclass constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class and if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super. variable

super. method();
```

1.2 Protected data, Private data, Public data

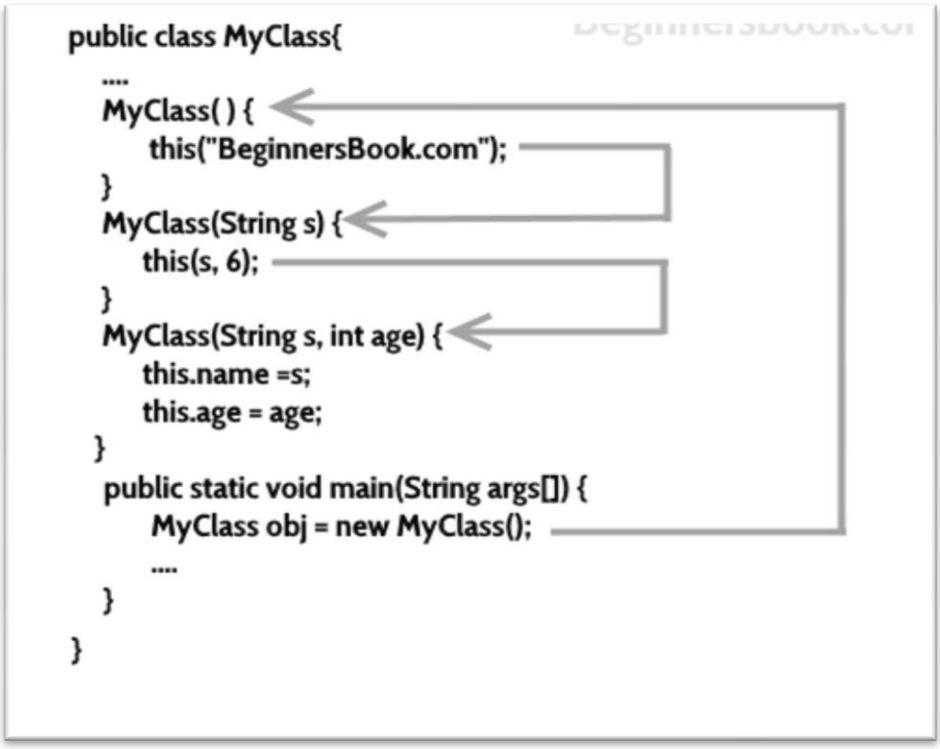
When a subclass is created from a superclass, the members of superclass becomes the members of subclass. But not all the members of superclass are inherited. There is a condition on which members can be inherited and which are not. The derived class inherits all the members and methods that are declared as public or protected. If the members or

methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected methods of super class. They cannot be inherited.

1.3 Constructor chaining

Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it's not a method as it doesn't have a return type. In short, constructor and method are different. People often refer constructor as special type of method in Java. Constructor has same name as the class.

When a constructor calls another constructor of same class then this is called constructor chaining.



Constructor chaining also occurs through the use of inheritance. A subclass constructor method's first task is to call its superclass' constructor method. This ensures that the creation of the subclass object starts with the initialization of the classes above it in the inheritance chain.

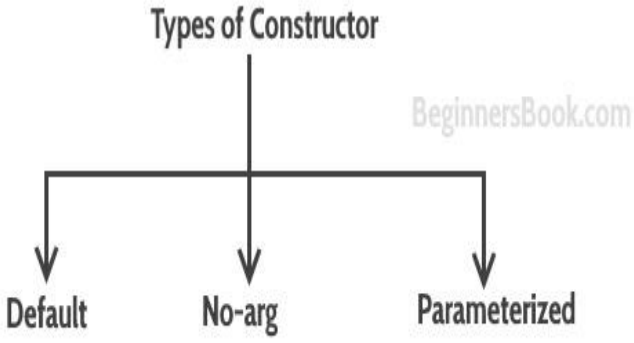
There could be any number of classes in an inheritance chain. Every constructor method will call up the chain until the class at the top has been reached and initialized. Then each subsequent

class below is initialized as the chain winds back down to the original subclass. This process is called constructor chaining.

1.3.1 Order of Invocation

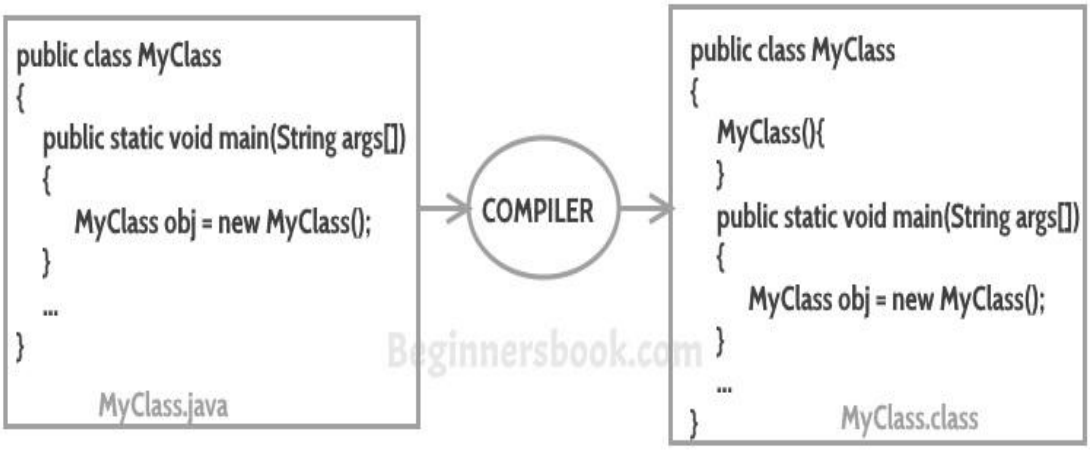
When both the child class and parent class have constructors inside them, then the order of invocation matters a lot. Also if there are more than one constructor in a class and when an object of such class is created the order of invocation of constructors matters. For understanding this we have to understand the types of constructor first.

There are three types of constructors: Default, No-arg constructor and Parameterized.



Default constructor

If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf. This constructor is known as default constructor. You would not find it in your source code (the java file) as it would be inserted into the code during compilation and exists in .class file. This process is shown in the diagram below:



If you implement any constructor then you no longer receive a default constructor from Java compiler.

no-arg constructor:

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

Parameterized constructor

Constructor with arguments (or you can say parameters) is known as Parameterized constructor. If we create parametrized constructor in class then we have to supply arguments or parameters when we create the objects of that classs. We can also create more than one constructors in a class. This is called constructor overloading. If we have two constructors, a default constructor and a parameterized constructor in the same class, and we do not pass any parameter while creating the object using new keyword then default constructor is invoked, however when you pass a parameter then parameterized constructor that matches with the passed parameters list gets invoked.When we are not using inheritance and we have more than one constructor inside a class then the constructors are invoked in the following order:

- First the default constructor is called
- After that the parameterized constructor is called.

While using inheritance if we have a constructor in the base class then it is compulsory to create a constructor in the subclass. The child class constructor automatically invokes the base class

constructor**.** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass. There could be any number of classes in an inheritance chain. Every constructor method will call up the chain until the class at the top has been reached and initialized. Then each subsequent class below is initialized as the chain winds back down to the original subclass. This process is called constructor chaining.

Super()

Whenever a child class constructor gets invoked it implicitly invokes the constructor of parent class. You can also say that the compiler inserts a super(); statement at the beginning of child class constructor.

```
class MyParentClass {
  MyParentClass(){
      System.out.println("MyParentClass Constructor");
  }
}
class MyChildClass extends MyParentClass{
  MyChildClass() {
      System.out.println("MyChildClass Constructor");
  }
  public static void main(String args[]) {
      new MyChildClass();
  }
}
```

Output:

MyParentClass Constructor

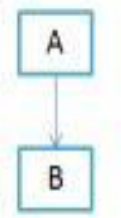
MyChildClass Constructor

1.4 Types of Inheritance

In java there are several types of inheritance available. these are explained below:

1.4.1 Single Inheritance

Single inheritance is very easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



Single Inheritance example program in Java

```
Class A
{
  public void methodA()
  {
   System.out.println("Base class method");
  }
}


Class B extends A
{
  public void methodB()
```

1.4.2 Multilevel Inheritance

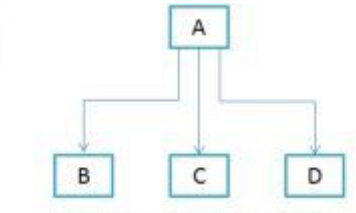
Multilevel inheritance refers to a mechanism in Object Oriented technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.

Multilevel Inhe

```
Class X
{
  public void methodX()
  {
   System.out.println("Class X method");
  }
}
Class Y extends X
{
   public void methodY()
   {
    System.out.println("class Y method");
   }
}
Class Z extends Y
```

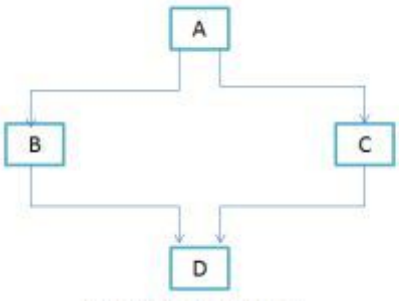
1.4.3 Hierarchical Inheritance

In such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent** class (or base class) of B, C & D.



1.4.4 Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below.



Multiple Inheritance

Multiple inheritance is not supported in java. The reason is that To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class. Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Unit 5

Polymorphism

1 Polymorphism

Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. For example, let's say we have a class Animal that has a method sound(). Since this is a generic class so we can't give it a implementation like: Roar, Meow, Oink etc. We had to give a generic message.

```
Public class Animal
{
  …
  public void sound()
{
    System.out.println ("Animal is making a sound");
  }
}
```

Now let's say we two subclasses of Animal class: Horse and Cat that extends (see Inheritance) Animal class. We can provide the implementation to the same method like this:

```
public class Horse extends Animal
{
...
   @Override
   public void sound(){
      System.out.println("Neigh");
   }
}
```

and

```
public class Cat extends Animal
{
...
   @Override
   public void sound(){
      System.out.println("Meow");
   }
}
```

As you can see that although we had the common action for all subclasses sound() but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways). It would not make any sense to just call the generic sound() method as each Animal has a different sound. Thus we can say that the action this method performs is based on the type of object. Polymorphism is the capability of a

method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the above example that we have defined the method sound() and have the multiple implementations of it in the different-2 sub classes.

There are two types of polymorphism in java:

1) **Static Polymorphism** also known as compile time polymorphism
2) **Dynamic Polymorphism** also known as runtime polymorphism

Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that's why it is called runtime polymorphism.

2 What is Overloading?

Overloading means different things having same name. Overloading allows different methods to have same name, but different working. They can differ by number of input parameters or type of input parameters or both. Overloading is related to compile time (or static) polymorphism. the benefit of overloading is that We don't have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc. whereas with overloading we can create different methods with same name SUM with different implementations

2.1 Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. By argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters. In order to overload a method, the argument lists of the methods must differ in either of these:

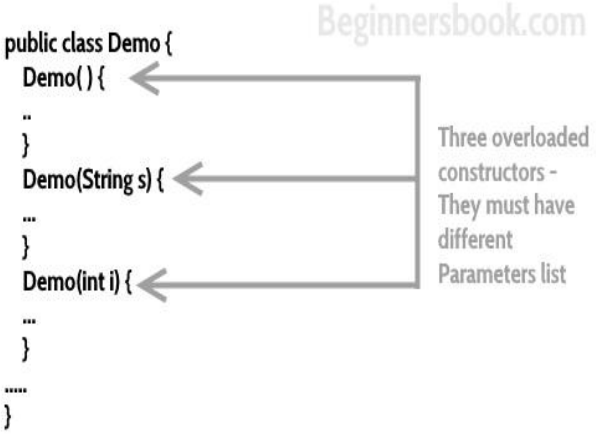
1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

However if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error. **Method overloading** is an example of Static Polymorphism. Static Polymorphism is also known as compile time binding or early binding.

Static binding happens at compile time. So method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

2.2 Constructor Overloading

Like methods, constructors can also be overloaded. Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.



2. What is Overriding?

Overloading means methods having same name but different parameter list. But if methods have same name and same parameters then it is called overriding of methods

.

2.1 Method Overriding

Declaring a method in **subclass** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.
Let's take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat (). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.
The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

Rules for Java Method Overriding

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class.

There must be an IS-A relationship (inheritance).

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.

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch.

2.3 Constructor Overriding

By rule in Java, a **constructor cannot be overridden** but a method can be overridden. The members of a class are instance variables and methods but not constructors because

1. a constructor cannot be overridden.
2. a constructor cannot be called with an object but method can be called. To call a constructor, an object must be created.
3. in case of methods, we say, the "subclass method can call super class method". But it cannot be said with constructors and inturn we should say "subclass constructor can **access** super class constructor"; here, correct word is "access" and not "call". It is for the reason, a constructor cannot be overridden.
4. we cannot call a super class constructor with super keyword. For constructors, specially one more term is created by Designers – **super()**.
5. in method overriding, the super class method and subclass method should be of the same – same name, same return type and same parameter list. In constructors, same name constructor cannot be written in another class; it is **compilation error**.

Overloading vs. Overriding in Java

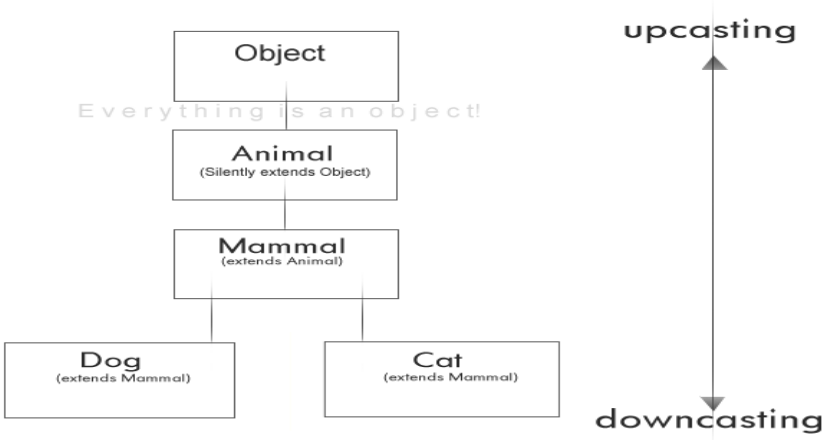
1. Overloading happens at compile-time while Overriding happens at runtime: The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. Static binding is being used for overloaded methods and dynamic binding is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. Private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.

7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type (refer this).
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

3 Up-casting & Down-casting

Typecasting is converting one data type to another.

Up-casting: Converting a subclass type to a superclass type is known as up casting.
Down-casting: Converting a superclass type to a subclass type is known as downcasting



For example in the following program class Animal contains only one method but Dog class contains two methods, then how we cast the Dog variable to the Animal Variable. If casting is done then how can we call the Dog's another method with Animal's variable.

```
class Animal
{
    public void callme()
    {
     System.out.println("In callme of Animal");
    }
}
class Dog extends Animal
{
    public void callme()
   {
```

```
      System.out.println("In callme of Dog");
    }

    public void callme2()
   {
     System.out.println("In callme2 of Dog");
   }
}

public class UseAnimlas
{
    public static void main (String [] args)
    {
    Dog d = new Dog();
    Animal a = (Animal)d;          // Up-Casting
    d.callme();
    a.callme();
    ((Dog) a).callme2();
    }
}
```

a cast from Dog to an Animal is an upcast, because a Dog is-a Animal. In general, you can upcast whenever there is an is-a relationship between two classes.

Downcasting would be something like this:

```
Animal animal = new Dog();
Dog  castedDog = (Dog) animal;
```

Unit 6

Abstract class & Interface

Key points of Abstract class & interface

difference between an abstract class & interface

implementation of multiple inheritance through interface.

In Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it. Java, abstraction is achieved using Abstract classes and interfaces.

1. Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class. An abstract class has following features:

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration. For example:

```
public abstract class Employee
{
  private String name;
  private String address;
```

```
private int number;

public Employee(String name, String address, int number) {
  System.out.println("Constructing an Employee");
  this.name = name;
  this.address = address;
  this.number = number;
}

public double computePay() {
 System.out.println("Inside Employee computePay");
 return 0.0;
}

public void mailCheck() {
  System.out.println("Mailing a check to " + this.name + " " + this.address);
}

public String toString() {
  return name + " " + address + " " + number;
}

public String getName() {
  return name;
}
```

```
    public String get Address() {
      return address;
    }

    public void set Address(String new Address) {
      address = new Address;
    }

    public int get Number() {
      return number;
    }
  }
```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor. Now if you try to instantiate the Employee class, it will generate a compilation error. We cannot create objects of an abstract class. We can inherit the properties of Employee class just like simple class. Let us say that we create a new class Salary that inherits the Employee class. you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class.

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.

- Instead of curly braces, an abstract method will have a semi colon (;) at the end.

Interfaces

In java multiple inheritance is not allowed. That is, a class cannot inherit features from more than one class. Interface is an alternate for multiple inheritance. For inheritance extends keyword is used whereas for interface **implements** keyword is used.

It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types.

An interface is **similar to a class** in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is **different from a class** in several ways, including

- You cannot instantiate an interface that is you cannot create objects of interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; **it is implemented by a class.**
- An interface can extend multiple interfaces. That is an interface can inherit features from multiple interfaces. Multiple inheritance is not allowed for classes but allowed for interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

```
interface Name Of Interface
```

```
{
    // Any number of final, static fields
    // Any number of abstract method declarations\
}

interface Animal {
  public void eat();
  public void travel();
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
public class MammalInt implements Animal
{
    public void eat() {
      System.out.println("Mammal eats");
    }

    public void travel() {
```

```
        System.out.println("Mammal travels");
      }

      public int noOfLegs() {
        return 0;
    }

    public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
    }
  }
```

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Difference between Interface and Abstract Class

1. Main difference is that methods of a Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implement a default behavior.
2. Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
3. Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected, etc..

4. Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
5. An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
6. A Java class can implement multiple interfaces but it can extend only one abstract class.
7. Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main () exists.
8. In comparison with java abstract classes, java interfaces are slow as it requires extra indirection.

Unit 7

Exception Handling

Exception Handling

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

1.A user has entered an invalid data.

2.A file that needs to be opened cannot be found.

3. A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions.

Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception
FileNotFoundException; must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

```
public class Unchecked_Demo {

   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
       at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

Errors − these are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

Syntax

```
try {
   // Protected code
} catch (Exception Name e1) {
   // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name: ExcepTest.java
Import java.io.*;

public class ExcepTest {

   public static void main(String args[]) {
      try {
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
   file = new FileInputStream(fileName);
   x = (byte) file.read();
} catch (IOException i) {
   i.printStackTrace();
   return -1;
} catch (FileNotFoundException f) // Not valid! {
   f.printStackTrace();
   return -1;
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it −

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
   throw ex;
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

Example

```
import java.io.*;
public class className {

   public void deposit(double amount) throws RemoteException {
      // Method implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException −

Example

```
import java.io.*;
public class className {

   public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
      // Method implementation
   }
   // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

Syntax

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

Example

```
public class ExcepTest {

   public static void main(String args[]) {
      int a[] = new int[2];
      try {
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }finally {
         a[0] = 6;
         System.out.println("First element value: " + a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
    The finally statement is executed
```

Note the following −

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.

- Any code cannot be present in between the try, catch, finally blocks.

Exception Handling API - For an advanced and clean exception handling, we propose the following API that provides the following features: -

1. Reduce the development cost of exception handling (just one single line in a single catch clause).
2. Avoid the misunderstanding of try/catch and throws by eliminating the need for throws.
3. Make the actual exception handling centralized and implemented by the appropriate people.
4. Give developers the ability to plug their exception handling if required.
5. Make exception handling consistent across the whole application (e.g. a database down exception should be handled in the same way across all the system views).
6. The exception handling shouldn't affect back-end and API's when using different front-end technologies (e.g. web or native).

A dvantages of Exception Handling in Java

Java provides a sophisticated exception handling mechanism that enables you to detect exceptional conditions in your programs and fix the exceptions as and when they occur. Using exception handling features offers several advantages. Let's examine these advantages in detail.

Provision to Complete Program Execution:

One of the important purposes of exception handling in Java is to continue program execution after an exception is caught and handled.

The execution of a Java program does not terminate when an exception occurs. Once the exception is resolved, program execution continues till completion.

By using well-structured try, catch, and finally blocks, you can create programs that fix exceptions and continue execution as if there were no errors. If there is a possibility of more than one exception, you can use multiple catch blocks to handle the different exceptions.

The following program generates two random integers in each iteration of the for loop and performs a division operation. If a division by zero error occurs, the exception is handled in the catch block. Thus, an arithmetic exception does not terminate the program and the for loop continues execution after the catch block is executed.

Advantages of Exception Handling in Java

Java provides a sophisticated exception handling mechanism that enables you to detect exceptional conditions in your programs and fix the exceptions as and when they occur. Using exception handling features offers several advantages. Let's examine these advantages in detail.

Provision to Complete Program Execution:

One of the important purposes of exception handling in Java is to continue program execution after an exception is caught and handled.

The execution of a Java program does not terminate when an exception occurs. Once the exception is resolved, program execution continues till completion.

By using well-structured try, catch, and finally blocks, you can create programs that fix exceptions and continue execution as if there were no errors. If there is a possibility of more than one exception, you can use multiple catch blocks to handle the different exceptions.

The following program generates two random integers in each iteration of the for loop and performs a division operation. If a division by zero error occurs, the exception is handled in the catch block. Thus, an arithmetic exception does not terminate the program and the for loop continues execution after the catch block is executed.

ExceptionExample.java

```
import java.util.Random;

class ExceptionExample {

    public static void main(String args[]) {
        int a = 0, b = 0, c = 0;
        Random r = new Random();

        for (int i = 0; i < 10; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b / c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero");
                a = 0;
            }
            System.out.println("a: " + a);
        }
    }
```

Easy Identification of Program Code and Error-Handling Code:

The use of try/catch blocks segregates error-handling code and program code making it easier to identify the logical flow of a program. The logic in the program code does not include details of the actions to be performed when an exception occurs. Such details are present in the catch blocks.

Unlike many traditional programming languages that include confusing error reporting and error handling code in between the program code, Java allows you to create well-organized code. Separating error handling and program logic in this way makes it easier to understand and maintain programs in the long run.

Propagation of Errors:

Java's exception handling mechanism works in such a way that error reports are propagated up the call stack. This is because whenever an exception occurs, Java's runtime environment checks the call stack backwards to identify methods that can catch the exception.

When a program includes several calls between methods, propagation of exceptions up the call stack ensures that exceptions are caught by the right methods.

Meaningful Error Reporting:

The exceptions thrown in a Java program are objects of a class. Since the Throwable class overrides the `toString()` method, you can obtain a description of an exception in the form of a string and display the description using a `println()` statement.

catch block

```
catch (ArithmeticException e) {
       System.out.println("Exception occurred: " + e);
}
```

The above catch statement displays the following output when an arithmetic exception occurs:

```
Exception: java.lang.ArithmeticException: / by zero
```

Traditional programming languages use error codes for error reporting. In the case of large programs, debugging errors using their error codes gets complex. The meaningful descriptions provided by Java's exception handling mechanism are helpful when you need to debug large programs or experiment with complex code.

Identifying Error Types:

Java provides several super classes and sub classes that group exceptions based on their type. While the super classes like IOException provide functionality to handle exceptions of a general type, sub classes like FileNotFoundException provide functionality to handle specific exception types.

A method can catch and handle a specific exception type by using a sub class object.

For example, FileNotFoundException is a sub class that only handles a file not found exception. In case a method needs to handle multiple exceptions that are of the same group, you can specify an object of a super class in the method's catch statement.

For example, IOException is a super class that handles all IO-related exceptions.

.