Object-Oriented Programming Guide

By Programming Club VPCI™

Date : 2018-12-28

Table of Contents

Abstract

This guide can be considered the winter break sessions of "Programming Club" at VPCI! The topic that we would have covered at the club meetings are stated above in the Table of Contents.

Object-Oriented Programming (OOP for short) is an essential skill for all programmers, especially in the workforce. The basic premise of OOP is to emulate certain parts of reality and the interactions going on between the objects in that reality. For example, all the inner machinations and subsections of a car can be emulated on a computer through the use of OOP. The wheels, frame, engine, driving wheel, and other such components would be created via various **classes**, all coming together through various relations. Each **object** of the class combine to create a unique car each and every time.

OOP is an important skill to learn for the workforce as many applications for companies involve taking something from reality, and making it virtual. This action is simplified using OOP, hence why it is an extremely desirable skill in the workforce, and why it is taught in schools across Ontario.

This guide will guide you through the working of OOP using **Java** as the main language. If Java is not your main language, do not be alarmed—the concepts of OOP can be transferred to other languages as well.

The first 2 weeks back from the winter break will be covering OOP. The first week will be review of this entire document, while the second week will be submission and take-up of the assignment.

We hope that everyone finds this guide as useful as possible, and that the winter assignment is not too difficult for those who are reading! Good luck to everyone on their programming endeavours!

Sincerely~
Programming Club Leaders (Rashad, Michael, Dave, Constance)

1. Introduction to Object-Oriented Programming (OOP)

1.1 Classes and Objects

Classes are user-created blueprints from which objects are created. Classes define the data which an object knows (called **fields)**, and the actions used to manipulate that data (called **methods**). A **class signature**—the act of declaring a class—include the following components, **in order :**

1. **Modifiers :** the access a program has to the class (can be **public**, **protected**, or **private)**
2. **Class Name :** the name should begin with a capitalized letter
3. **Superclass (if any) :** the name of the class' parent (superclass), preceded by the keyword (**extends)**; a class can **only extend one superclass**
4. **Interfaces (if any) :** a comma-separated list of interfaces, preceded by the keyword (**implements);** a class can **implement more than one interface**
5. **Body :** the class boy, surrounded by brace brace brackets, { }

Objects are the basic unit of OOP, and represent each individual entity. A typical program creates many objects, which interact with each other using various methods. An object consists of the following :

1. **State :** the attributes/properties of the object
2. **Behaviour :** the methods of an object
3. **Identity :** the unique name given to an object

When an object of a class is created, the class is said to be **instantiated.** Each **instance** of the class share the attributes and behaviours of the class, however the values of those attributes are unique to each instance.

```
// Class Declaration

public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
```

```
    public Dog(String name, String breed, int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName()
    {
        return name;
    }

    // method 2
    public String getBreed()
    {
        return breed;
    }

    // method 3
    public int getAge()
    {
        return age;
    }

    // method 4
    public String getColor()
    {
        return color;
    }

    @Override
    public String toString()
    {
        return("Hi my name is "+ this.getName()+
               ".\nMy breed,age and color are " +
               this.getBreed()+"," + this.getAge()+
```

```
                ","+ this.getColor());
    }

    public static void main(String[] args)
    {
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

Example code outlining the declaration and initialization of a class and an object

This class contains a single **constructor,** as recognized by the declaration utilising the same name of the class and lack of a return type. All classes have at least **one** constructor, whether explicitly created by the programmer, or implicitly defined by the program.

There is only one method for creating an object of a class : by using the **new** keyword.

```
// creating object of class Test
Test t = new Test();
```

Example creation of an object from a class

There also exists other types of objects called **anonymous objects**. They are objects that are instantiated but not stored in a reference variable. They are used for immediate method calling, are destroyed after method calling, and are widely used in various different libraries.

```
btn.setOnAction(new EventHandler()
{
      public void handle (ActionEvent event)
      {
            System.out.println("Hello World!");
      }
});
```

Example of an anonymous object being utilised by one of the classes in a Java library

1.2 Encapsulation

Encapsulation is considered to be wrapping up of data under a single unit. It is what binds together code and the data it manipulates. It is also known as **data-hiding**. Encapsulation is achieved through the use of **access modifiers.** They help to restrict the scope of a class, constructor, variable, method, or any other such thing. There are three well known modifiers :

Access To :	**private**	**protected**	**public**
Same Class	Yes	Yes	Yes
Same package subclass	No	Yes	Yes
Same package non-subclass	No	Yes	Yes
Different package subclass	No	Yes	Yes
Different package non-subclass	No	No	Yes

```
//Java program to illustrate error while
//using class from different package with
//private modifier
package p1;

class A
{
   private void display()
    {
        System.out.println("GeeksforGeeks");
    }
}

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

```
        A obj = new A();
        //trying to access private method of another class
        obj.display();
    }
}
```

Example code showing capability of private access modifier

In the above example, an error message will be shown as the object is trying to access the private method of another class. The object is restricted from manipulating the data and methods of the class, and therefore will not be able to accomplish the line, hence why an error message will be shown.

```
//Java program to illustrate
//protected modifier
package p1;

//Class A
public class A
{
   protected void display()
    {
        System.out.println("GeeksforGeeks");
    }
}

//Java program to illustrate
//protected modifier
package p2;
import p1.*; //importing all classes in package p1

//Class B is subclass of A
class B extends A
{
   public static void main(String args[])
   {
       B obj = new B();
       obj.display();
   }
```

```
}
```

Example code showing how access via protected is different

Protected allows the object of class B to access the methods of class A due to their inheritance relationship.

```
//Java program to illustrate
//public modifier
package p1;
public class A
{
   public void display()
      {
          System.out.println("GeeksforGeeks");
      }
}
package p2;
import p1.*;
class B
{
    public static void main(String args[])
      {
          A obj = new A;
          obj.display();
      }
}
```

Example code showing how public is much less restrictive compared to protected

Public allows access to all methods and fields regardless of which class is calling the method

The following are a list of advantages of encapsulation :

- **Data Hiding :** hide important information from the client that is not needed for them
- **Increased Flexibility :** change the access levels based on our requirements
- **Reusability :** easy to change with new requirements
- **Testing code is Easy :** easy to test for unit testing

2. Methods

A **method** is a specific action or a way in which an object can manipulate its data. The following is a list of the different types of methods one can create :

- **Accessor Method** : a method that **returns** the information of a certain field
- **Mutator Method :** a method that **changes** the information of a certain field
- **Constructor :** a special method that instantiates a class, creating a unique object
- **Utility Method :** a generic method that can have many different functionalities depending on the specifications of the class
- **Main Method :** a **static method** where all the interactions between objects occur

2.1 Parameter Passing

A **parameter** is the variable holding the value passed by the user to the system via the declaration of the parameter variable in the method header. One of the main things to note is that a copy of the variable being passed into the method is created, meaning that the method does the data manipulation on that. This means that there are no direct changes to the data stored inside the original variable.

```
public class Main
{
    public static void main(String[] args)
    {
        int x = 5;
        change(x);
        System.out.println(x);
    }
    public static void change(int x)
    {
        x = 10;
    }
}
```

Example code showing how the parameter does not affect the original variable

Since a copy of the variable ‘x’ was passed to the method ‘change’, the original variable of ‘x’ did not undergo a change, and would remain as 5.

2.2 Passing and Returning Objects

There is a significant difference when passing a **primitive data type** and a **reference type** to a method. A primitive data type passes the value, while an object passes the reference to the memory location containing its information. This means that changing the parameter containing the object does not change the object, but calling the methods of that object using the paramter allows changes to occur.

```
// Java program to demonstrate objects
// passing to methods.
class ObjectPassDemo
{
    int a, b;

    ObjectPassDemo(int i, int j)
    {
        a = i;
        b = j;
    }

    // return true if o is equal to the invoking
    // object notice an object is passed as an
    // argument to method
    boolean equalTo(ObjectPassDemo o)
    {
        return (o.a == a && o.b == b);
    }
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
```

```
        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

Example code showcasing the use of an object as a parameter

This program initially creates three objects. The method ‘compareTo’ compares the parameter object to another object. Notice how the function can access the information from the original object, even though a parameter was used.

2.3 Private and Final Methods

When using the keyword **final**, the method cannot be overridden by any of the inheriting classes. Since private methods can not be accessed, they are implicitly considered final, so using the final keyword serves no purpose.

```
class Base {
    private void foo() {}
}

class Derived extends Base {
    public void foo() {}
}

public class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.foo();
    }
}
```

Example code showing the use of private and implicit definition of final

The above code will produce a compiler error as ‘foo()’ has private access in class Base.

3. Constructors

3.1 Constructors and its Types

Constructors are used to initialize an object's state. They are a unique type of method that is executed at the time of object creation. A constructor is called whenever the keyword **new** is used. The following are some rules for writing constructors :

- Constructors of a class must have the **same name as the class name**
- A constructor cannot be abstract, final, and static
- Access modifiers can be used in constructor declaration to control which class can call it

```
// Java Program to illustrate calling a
// no-argument constructor
import java.io.*;

class Geek
{
    int num;
    String name;

    // this would be invoked while object
    // of that class created.
    Geek()
    {
        System.out.println("Constructor called");
    }
}

class GFG
{
    public static void main (String[] args)
    {
        // this would invoke default constructor.
        Geek geek1 = new Geek();

        // Default constructor provides the default
        // values to the object like 0, null
```

```
        System.out.println(geek1.name);
        System.out.println(geek1.num);
    }
}
```

Example of no-argument constructor

This type of constructor is called a no-argument constructor as the constructor has no parameter. Another name for this is the **default constructor.** If there is no constructor defined in the class, then the compiler creates a default constructor with no arguments. The compiler does this only when no constructor has been created.

```
// Java Program to illustrate calling of
// parameterized constructor.
import java.io.*;

class Geek
{
    // data members of the class.
    String name;
    int id;

    // contructor would initialized data members
    // with the values of passed arguments while
    // object of that class created.
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}

class GFG
{
    public static void main (String[] args)
    {
        // this would invoke parameterized constructor.
        Geek geek1 = new Geek("adam", 1);
        System.out.println("GeekName :" + geek1.name +
```

```
                            " and GeekId :" + geek1.id);
    }
}
```

Example code showing a constructor with parameters

This type of constructor is called a parameterized constructor, and is called such as it contains parameters. This type of constructor is useful for when the program calls for the fields to be initialized by the user.

```
// Java Program to illustrate constructor overloading
// using same task (addition operation ) for different
// types of arguments.
import java.io.*;

class Geek
{
    // constructor with one argument
    Geek(String name)
    {
        System.out.println("Constructor with one " +
                        "argument - String : " + name);
    }

    // constructor with two arguments
    Geek(String name, int age)
    {

        System.out.print("Constructor with two arguments : " +
                " String and Integer : " + name + " "+ age);

    }

    // Constructor with one argument but with different
    // type than previous..
    Geek(long id)
    {
        System.out.println("Constructor with one argument : " +
                                            "Long : " + id);
```

```
    }
}

class GFG
{
    public static void main(String[] args)
    {
        // Creating the objects of the class named 'Geek'
        // by passing different arguments

        // Invoke the constructor with one argument of
        // type 'String'.
        Geek geek2 = new Geek("Shikhar");

        // Invoke the constructor with two arguments
        Geek geek3 = new Geek("Dharmesh", 26);

        // Invoke the constructor with one argument of
        // type 'Long'.
        Geek geek4 = new Geek(325614567);
    }
}
```

Example code showing constructor overload

Constructors can be **overloaded**, meaning that you can create multiple constructors with the same name, but different parameter lists. This allows for different levels of information to be given to an object, based on the circumstances of the user.

The following are some differences between regular methods and constructors :

- Constructors must have same name as the class
- Constructors do not have a return type while methods have a return type, or **void**
- Constructors are called only once at the time of object creation, while methods can be called any number of times

3.2 Constructor Chaining

Constructor chaining is when one constructor is called from another constructor with respect to the current object. It can be done in two ways :

- **Within same class :** through the use of the **this()** keyword

- **From base class :** through the use of the **super()** keyword

This occurs through **inheritance.** A subclass constructor calls the super-class's constructor first, ensuring that the creation of the subclass object starts with the initialization of the fields from the super-class.

```
// Java program to illustrate Constructor Chaining
// within same class Using this() keyword
class Temp
{
    // default constructor 1
    // default constructor will call another constructor
    // using this keyword from same class
    Temp()
    {
        // calls constructor 2
        this(5);
        System.out.println("The Default constructor");
    }

    // parameterized constructor 2
    Temp(int x)
    {
        // calls constructor 3
        this(5, 15);
        System.out.println(x);
    }

    // parameterized constructor 3
    Temp(int x, int y)
    {
        System.out.println(x * y);
    }

    public static void main(String args[])
    {
        // invokes default constructor first
        new Temp();
```

```
    }
}
```

Example code showing constructor chaining via this()

The following are some rules for constructor chaining using the **this()** strategy :

- The this() expression should always be the first line of the constructor
- There should be at least one constructor without that keyword
- It can be achieved in any order of constructors in a class

```
// Java program to illustrate Constructor Chaining to
// other class using super() keyword
class Base
{
    String name;

    // constructor 1
    Base()
    {
        this("");
        System.out.println("No-argument constructor of" +
                                            " base class");
    }

    // constructor 2
    Base(String name)
    {
        this.name = name;
        System.out.println("Calling parameterized constructor"
                                                + " of base");
    }
}

class Derived extends Base
{
    // constructor 3
    Derived()
    {
        System.out.println("No-argument constructor " +
```

```
                            "of derived");
    }

    // parameterized constructor 4
    Derived(String name)
    {
        // invokes base class constructor 2
        super(name);
        System.out.println("Calling parameterized " +
                           "constructor of derived");
    }

    public static void main(String args[])
    {
        // calls parameterized constructor 4
        Derived obj = new Derived("test");

        // Calls No-argument constructor
        // Derived obj = new Derived();
    }
}
```

Example code showing constructor chaining via super()

Similar to constructor chaining in the same class, super() should be the first line of the constructor as the super-class's constructor is invoked before the subclasses.

4. Inheritance

4.1 Inheritance Primer

Inheritance is when one class is allowed to inherit (gain) the features (fields and methods) of another class. The following are some important terminology for understanding this :

- **Super class :** the class whose features are inherited
- **Sub class :** the class that inherits the other class, with the additional capability of adding its own fields and methods
- **Reusability :** fields and methods from existing classes are reused to prevent redundancy in programs

The keyword used for inheritance is **extends**.

```
//Java program to illustrate the
// concept of inheritance

// base class
class Bicycle
{
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
```

```
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return("No of gears are "+gear
                +"\n"
                + "speed of bicycle is "+speed);
    }
}

// derived class
class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override
```

```
    public String toString()
    {
        return (super.toString()+
                "\nseat height is "+seatHeight);
    }

}

// driver class
public class Test
{
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());

    }
}
```

Example code showing how extends() works

When an object of MountainBike is created, a copy of the super-class's fields are also stored in the object. The methods of the super-class can also be used by the subclass.

There are multiple different types of inheritance, which is as follows :

- **Single Inheritance :** subclasses inherit the features of one superclass
- **Multilevel Inheritance :** a derived class will be inheriting a base class, as well as acting as a base class for another class
- **Hierarchical Inheritance :** one class serves as a super-class for more than one subclass
- **Multiple Inheritance (Through Interfaces) :** one class can have more than one superclass (**for languages other than Java)**; for Java, interfaces are used instead
- **Hybrid Inheritance :** a mixture of two or more of the above mentioned types of inheritance

Every class is implicitly a subclass of **Object class**. A subclass inherits all members from its superclass; this does **NOT** include constructors, and private members. However, the constructor of the superclass can be invoked from the subclass.

4.2 Inheritance and Constructors

Constructors of a base class with no argument automatically gets called in the derived class's constructor.

```
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}

class Derived extends Base {
  Derived() {
    System.out.println("Derived Class Constructor Called ");
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }
}
```

Code example of a no-argument constructor inheritance

For this program, "Base Class Constructor Called " will get printed before "Derived Class Constructor Called ".

```
class Base {
  int x;
  Base(int _x) {
    x = _x;
  }
}

class Derived extends Base {
  int y;
  Derived(int _x, int _y) {
```

```
    super(_x);
    y = _y;
  }
  void Display() {
    System.out.println("x = "+x+", y = "+y);
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived(10, 20);
    d.Display();
  }
}
```

Code example of a parameterized constructor inheritance

The parameterized constructor of the base class is called using **super()**. The main purpose of this is so that the base class constructor is the first line in the derived class constructor.

5. Polymorphism

5.1 Overloading

Overloading is when multiple method can have the same name, but different signatures. This means that the overall name for the method is the same, but the number of parameters or types of parameters are different. This is considered a type of compile time (**static) polymorphism.**

```
// Java program to demonstrate working of method
// overloading in Java.

public class Sum
{

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
```

```
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Example code of static polymorphism

It is useful as we will not have to create multiple different method names for functions that operate in the same manner. It is **not** possible to overload by return type. It is possible only when the data type of the function being called is explicitly specified.

```
// Java program to demonstrate working of method
// overloading in static methods
public class Main
{
    public static int foo(int a) { return 10; }
    public static char foo(int a, int b) { return 'a'; }

    public static void main(String args[])
    {
        System.out.println(foo(1));
        System.out.println(foo(1, 2));
    }
}
```

Example code showing method overload with different return types

It is also not possible to differentiate two methods simply by the fact that one has **static** in their method signature. Java does not support operator overloading (although Java already has overloaded operators, such as + for concatenation and addition). **C++ allows operator overloading.**

Overloading is when the same function has different method signatures. **Overriding** is having the same function with the same signature, but different classes (and therefore different method bodies) through inheritance.

5.2 Multiple Ways of Method Overloading

There are three different ways of doing method overloading, all of which involve changing one small aspect of the method signature.

- **The number of parameters in two methods**

```
import java.io.*;

class Addition
{
    // adding two integer values.
    public int add(int a, int b){

        int sum = a+b;
        return sum;
    }

    // adding three integer values.
    public int add(int a, int b, int c){

        int sum = a+b+c;
        return sum;
    }

}

class GFG {
    public static void main (String[] args) {

        Addition ob = new Addition();

        int sum1 = ob.add(1,2);
        System.out.println("sum of the two integer value :" + sum1);
        int sum2 = ob.add(1,2,3);
        System.out.println("sum of the three integer value :" +
sum2);

    }
}
```

Code example of type 1 method overloading

- **The data types of the parameters of methods**

```
import java.io.*;
```

```
class Addition{

    // adding three integer values.
    public int add(int a, int b, int c){

        int sum = a+b+c;
        return sum;
    }

    // adding three double values.
    public double add(double a, double b, double c){

        double sum = a+b+c;
        return sum;
    }
}

class GFG {
    public static void main (String[] args) {

        Addition ob = new Addition();

        int sum2 = ob.add(1,2,3);
        System.out.println("sum of the three integer value :" +
sum2);
        double sum3 = ob.add(1.0,2.0,3.0);
        System.out.println("sum of the three double value :" + sum3);


    }
}
```

Code example of type 2 method overloading

- **The order of the parameters of methods**

```
import java.io.*;

class Geek{
```

```
    public void geekIdentity(String name,  int id){

        System.out.println("geekName :"+ name +" "+"Id :"+ id);
    }

    public void geekIdentity(int id, String name){

        System.out.println("geekName :"+ name +" "+"Id :"+ id);
    }
}

class GFG {
    public static void main (String[] args) {

        Geek geek = new Geek();

        geek.geekIdentity("Mohit", 1);
        geek.geekIdentity("shubham", 2);

    }
}
```

Code example of type 3 method overloading

5.3 Overriding

Overriding is a feature that allows a subclass or child class to provide their own versions of a pre-existing method belonging to a super-class or parent class. The method with the same method signature in the subclass **overrides** the same method signature in the super-class.

```
// A Simple Java program to demonstrate application
// of overriding in Java

// Base Class
class Employee
{
    public static int base = 10000;
    int salary()
    {
```

```
        return base;
    }
}

// Inherited class
class Manager extends Employee
{
    // This method overrides show() of Parent
    int salary()
    {
        return base + 20000;
    }
}

// Inherited class
class Clerk extends Employee
{
    // This method overrides show() of Parent
    int salary()
    {
        return base + 10000;
    }
}

// Driver class
class Main
{
    // This method can be used to print salary of
    // any type of employee using base class refernce
    static void printSalary(Employee e)
    {
        System.out.println(e.salary());
    }

    public static void main(String[] args)
    {
        Employee obj1 = new Manager();
```

```
        // We could also get type of employee using
        // one more overridden method.loke getType()
        System.out.print("Manager's salary : ");
        printSalary(obj1);

        Employee obj2 = new Clerk();
        System.out.print("Clerk's salary : ");
        printSalary(obj2);
    }
}
```

Code example of method overriding

The following are some rules for method overriding :

- **Access modifiers** : they can allow more transparency when going down subclasses (ie. protected method in the super-class can become public method in subclass, not private)
- **Final methods** : method signatures that include the keyword **final** cannot be overridden
- **Static methods** : method signatures that include the keyword **static** are not overridden, but rather are **hidden**
- **Private methods** : method signatures with the access modifier **private** cannot be overridden
- **Return types** : the overriding method in the same class must have the same return type
- **Invocation of methods** : it is possible to call the parent class method from the child class using the **super** keyword
- **Constructors** : it is **impossible** to override constructors as two related classes can never have constructors with the same name
- **Exception handling** : if the super-class method does not throw an exception, then the subclass method can only throw the unchecked exception. If the the super-class method does throw an exception, then the subclass method can only throw the same exception
- **Abstract methods** : abstract methods are always meant to be overridden in derived classes, otherwise there will be a compile-time error

5.4 Constructor Overloading

Overloaded constructors are called based upon the parameters specified when **new** is executed. There exists a need to initialize an object in different ways based on the given data, hence why multiple constructors can exist.

```
// Java program to illustrate
```

```
// Constructor Overloading
class Box
{
    double width, height, depth;

    // constructor used when all dimensions
    // specified
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions
    // specified
    Box()
    {
        width = height = depth = 0;
    }

    // constructor used when cube is created
    Box(double len)
    {
        width = height = depth = len;
    }

    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}

// Driver code
public class Test
{
    public static void main(String args[])
```

```
    {
        // create boxes using the various
        // constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println(" Volume of mycube is " + vol);
    }
}
```

Example code of constructor overloading

The keyword **this()** can be used during constructor overloading to use the default constructor implicitly from the parameterized constructor. It should be the first statement inside the parameterized constructor.

```
// Java program to illustrate role of this() in
// Constructor Overloading
class Box
{
    double width, height, depth;
    int boxNo;

    // constructor used when all dimensions and
    // boxNo specified
    Box(double w, double h, double d, int num)
```

```
    {
        width = w;
        height = h;
        depth = d;
        boxNo = num;
    }

    // constructor used when no dimensions specified
    Box()
    {
        // an empty box
        width = height = depth = 0;
    }

    // constructor used when only boxNo specified
    Box(int num)
    {
        // this() is used for calling the default
        // constructor from parameterized constructor
        this();

        boxNo = num;
    }

    public static void main(String[] args)
    {
        // create box using only boxNo
        Box box1 = new Box(1);

        // getting initial width of box1
        System.out.println(box1.width);
    }
}
```

Example code of this() for constructor overloading

6. Interfaces and Abstract Classes

6.1 Abstract Classes

An **abstract class** is a class that cannot be instantiated, but instead acts as a template for creating other classes. The methods in this class which use the keyword **abstract** are not defined in the abstract class, but in each individual subclass.

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function in C++)
    abstract void draw();
}
```

Example code with an abstract class

An abstract class can contain constructors, but cannot be called to create the abstract class. Instead, they are called when an instance of an inherited class is called.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```

Example code with a constructor in the abstract class

Abstract classes can also have other types of methods, and final methods as well. These are inherited by the subclass, and can be overridden (in the case of the non-final methods).

6.2 Interfaces

An **interface** is similar to a class in that it can have methods and variables, but the methods in an interface are all considered abstract. They specify what a class must do, but leaves the body up to the class. If a class does not define every method within an interface, then the class must be declared **abstract.**

```
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface in1 {
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements interface.
class testClass implements in1 {
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("Geek");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
```

```
}
```

Example code of an interface

These are the following purposes of an interface :

- Achieve total abstraction
- Allows multiple inheritance (**only for Java)**

The following are some important notes regarding interfaces :

- Interfaces cannot be instantiated, but can be referenced to the object of its implementing class
- A class can have multiple interfaces
- A class must implement **all** the methods in an interface
- All methods are **public and abstract***
- All fields are **public, static, and final**

* In JDK 9 (Java version 9), interfaces can have static methods, private methods, and private static methods

6.3 Difference between Abstract Class and Interface

1. **Type of method**
    - Interfaces can have only **abstract methods (pre Java 9)**
    - Abstract classes can have both abstract and non-abstract methods
2. **Final Variables**
    - Variables in an interface are by default final
    - Abstract classes can have non-final variables
3. **Type of Variables**
    - Abstract classes can have all forms of variables
    - Interfaces can only have static and final variables
4. **Implementation**
    - Abstract classes provide implementation of interfaces
5. **Inheritance vs. Abstraction**
    - Java interfaces are implemented using **implements** while abstract classes can be extended using **extends**
6. **Multiple Implementation**
    - Interfaces can extend another interface only, while an abstract class can extend another Java class and implement multiple interfaces
7. **Accessibility of Data Members**
    - Members of a Java interface are public by default
    - Abstract classes can have various access modifiers

7. Exception Handling

7.1 Exceptions

Exceptions are unwanted or unexpected events which occur during the execution of a program that disrupt the normal flow of the code. This should not be mistaken for an **error**, which indicate serious problems that an application cannot catch.

The user can handle an exception by using a **try-catch-finally block.** The try block attempts to do code within it, until an exception occurs (if it does). When an exception occurs, the try block ends and the catch block is then executed, handling the occurrence of an error in some rational manner. If there is any code that must be executed after a try block is completed is placed within the finally block.

```
try {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// optional
finally {
// block of code to be executed after try block ends
}
```

Code template for exception handling

7.2 Exception Flow Control

The following are some combinations of control flow in **try-catch** or **try-catch-finally** blocks :

1. **Exception occurs in try block and handled in catch block :** the rest of the try block does not execute, and control goes towards to the corresponding catch block. After, it will be transferred to the finally block (if present), before finishing the rest of the program.

```
// Java program to demonstrate
```

```java
// control flow of try-catch-finally clause
// when exception occur in try block
// and handled in catch block
class GFG
{
    public static void main (String[] args)
    {

        // array of size 4.
        int[] arr = new int[4];

        try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }

        catch(ArrayIndexOutOfBoundsException ex)
        {
            System.out.println("Exception caught in catch block");
        }

        finally
        {
            System.out.println("finally block executed");
        }

        // rest program will be executed
        System.out.println("Outside try-catch-finally clause");
    }
}
```

Example code of an exception being handled

2. **Exception occured in try-block and is not handled in catch block :** the default handling mechanism is followed. If finally block present, it proceeds after the default handling mechanism.

```
// Java program to demonstrate
// control flow of try-catch-finally clause
// when exception occur in try block
// but not handled in catch block
class GFG
{
    public static void main (String[] args)
    {

        // array of size 4.
        int[] arr = new int[4];

        try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }

        // not a appropriate handler
        catch(NullPointerException ex)
        {
            System.out.println("Exception has been caught");
        }

        finally
        {
            System.out.println("finally block executed");
        }

        // rest program will not execute
        System.out.println("Outside try-catch-finally clause");
```

```
    }
}
```

Example code of an uncaught exception

3. **Exception does not occur in try-block :** the catch block never runs, however the finally block runs after the try-block, followed by the rest of the code.

```
// Java program to demonstrate try-catch-finally
// when exception doesn't occurred in try block
class GFG
{
    public static void main (String[] args)
    {

    try
    {

        String str = "123";

        int num = Integer.parseInt(str);

        // this statement will execute
        // as no any exception is raised by above statement
        System.out.println("try block fully executed");

    }

    catch(NumberFormatException ex)
    {

        System.out.println("catch block executed...");

    }

    finally
    {
        System.out.println("finally block executed");
    }
```

```
    System.out.println("Outside try-catch-finally clause");
    }
}
```

Example code of no exception occurring

If there is only a **try-finally block**, then the finally block will occur before the default control mechanism if an exception were to be raised.

7.3 Multi-Catch

Starting from Java version 7, a single catch block was capable of catching multiple exceptions by separating each exception with **| (pipe symbol)** in the catch block.

```
// A Java program to demonstrate multicatch
// feature
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
        catch (NumberFormatException | ArithmeticException ex)
        {
            System.out.println("Exception encountered " + ex);
        }
    }
}
```

Example code showing multi-catch

If multiple exceptions belong to a base exception type, it would be easier to catch the base exception type. When doing multi-catch, note that it is not possible to catch exceptions that related via inheritance (ie. cannot catch both NumberFormatException and Exception).

7.4 Throwing Exceptions

The **throw** keyword is used to explicitly throw an exception from a method or any block of code. It is also possible to throw either checked or unchecked exceptions. It is used mainly to throw custom exceptions. The one main issue is that the exception must be of type **Throwable** or a subclass of it. **Exception** is a subclass of Throwable, hence why all user defined exceptions are throwable. The flow of the code stops there, and is then sent down to the nearest **try-catch** block, if found.

```
// Java program that demonstrates the use of throw
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

Code example using keyword throw

The **throws** keyword is used in the signature of a method to indicate this method might throw one of the listed exceptions. The caller to these methods handles the exception in their own try-catch block.

```
// Java program to demonstrate working of throws
class ThrowsExecp
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalAccessException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

Example code using throws

7.5 Types of Exceptions

There are two main types of Java exceptions : **built-in exceptions, and user-defined exceptions.** Built-in exceptions are ones which already exist within the Java libraries. The following are some important exceptions to be wary of in Java :

- **ArithmeticException :** thrown when an exceptional condition has occurred in an arithmetic operation (ie. dividing by 0)
- **ArrayIndexOutOfBoundException :** thrown to indicate that an array has been accessed with an illegal index (either negative, or above the range of the array)
- **ClassNotFoundException :** occurs when we try to access a class whose definition is not found

- **FileNotFoundException :** occurs when we try to access a file that does not exist, or is impossible to open
- **IOException :** thrown when an input-output operation fails or is interrupted
- **InterruptedException :** thrown when a thread is processing, and is interrupted
- **NoSuchFieldException :** thrown when a class does not contain a specified field or variable
- **NoSuchMethodException :** thrown when accessing a method which is not found
- **NullPointerException :** occurs when referring to members of a **null** object
- **NumberFormatException :** occurs when a method could not convert a string into numbers
- **RuntimeException :** represents every exception that can occur during runtime
- **StringIndexOutOfBoundsException :** thrown by String class methods to indicate that an index is either negative or larger than the size of the string

7.6 User-Defined Exceptions

Programmers in Java can create their own exceptions, which are all derived from the class **Exception.** All user-defined exceptions **extend** Exception. The constructor of an exception can have parameters, as long as **super()** is called in the first line.

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
```

```
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

Example code of an user-defined exception

It is also possible to call the constructor of exception class without the use of any parameter.

```
// A Class that represents use-defined expception
class MyException extends Exception
{

}

// A Class that uses above MyException
public class setText
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException();
        }
        catch (MyException ex)
        {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}
```

Example code of an exception using the default constructor

8. Citations

"Abstract Classes in Java." *GeeksforGeeks*, 7 Dec. 2018, www.geeksforgeeks.org/abstract-classes-in-java/.

"Access Modifiers in Java." *GeeksforGeeks*, 7 Dec. 2018, www.geeksforgeeks.org/access-modifiers-java/.

"Access Specifier of Methods in Interfaces." *GeeksforGeeks*, 7 Dec. 2018, www.geeksforgeeks.org/g-fact-73/.

"Classes and Objects in Java." *GeeksforGeeks*, 5 Sept. 2018, www.geeksforgeeks.org/classes-objects-java/.

"Constructor Chaining In Java with Examples." *GeeksforGeeks*, 26 Apr. 2018, www.geeksforgeeks.org/constructor-chaining-java-examples/.

"Constructor Overloading in Java." *GeeksforGeeks*, 3 Sept. 2018, www.geeksforgeeks.org/constructor-overloading-java/.

"Constructors in Java." *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/constructors-in-java/.

"Copy Constructor in Java." *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/copy-constructor-in-java/.

"Default Constructor in Java." *GeeksforGeeks*, 10 July 2018, www.geeksforgeeks.org/g-fact-50/.

"Differences between Abstract Class and Interface in Java." *GeeksforGeeks*, 22 Nov. 2018, www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/.

"Different Ways of Method Overloading in Java." *GeeksforGeeks*, 3 Sept. 2018, www.geeksforgeeks.org/different-ways-method-overloading-java/.

"Dynamic Method Dispatch or Runtime Polymorphism in Java." *GeeksforGeeks*, 7 Sept. 2018, www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/.

"Encapsulation in Java." *GeeksforGeeks*, 7 Sept. 2018, www.geeksforgeeks.org/encapsulation-in-java/.

"Exceptions in Java." *GeeksforGeeks*, 6 Sept. 2018, www.geeksforgeeks.org/exceptions-in-java/.

"Flow Control in Try Catch Finally in Java." *GeeksforGeeks*, 17 Sept. 2018,

www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/.

"How Are Parameters Passed in Java?" *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/g-fact-45/.

"Inheritance and Constructors in Java." *GeeksforGeeks*, 17 July 2018, www.geeksforgeeks.org/g-fact-67/.

"Inheritance in Java." *GeeksforGeeks*, 5 Sept. 2018, www.geeksforgeeks.org/inheritance-in-java/.

"Interfaces in Java." *GeeksforGeeks*, 25 June 2018, www.geeksforgeeks.org/interfaces-in-java/.

"Java and Multiple Inheritance." *GeeksforGeeks*, 19 Oct. 2018, www.geeksforgeeks.org/java-and-multiple-inheritance/.

"Multicatch in Java." *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/multicatch-in-java/.

"Overloading in Java." *GeeksforGeeks*, 10 Sept. 2018, www.geeksforgeeks.org/overloading-in-java/.

"Overriding in Java." *GeeksforGeeks*, 30 Nov. 2018, www.geeksforgeeks.org/overriding-in-java/.

"Passing and Returning Objects in Java." *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/passing-and-returning-objects-in-java/.

"Private and Final Methods in Java." *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/private-and-final-methods-in-java/.

"Returning Multiple Values in Java." *GeeksforGeeks*, 13 Sept. 2018, www.geeksforgeeks.org/returning-multiple-values-in-java/.

"Types of Exception in Java with Examples." *GeeksforGeeks*, 7 Sept. 2018, www.geeksforgeeks.org/types-of-exception-in-java-with-examples/.

"User-Defined Custom Exception in Java." *GeeksforGeeks*, 27 May 2017, www.geeksforgeeks.org/g-fact-32-user-defined-custom-exception-in-java/.

"Throw and Throws in Java." *GeeksforGeeks*, 29 Oct. 2018, www.geeksforgeeks.org/throw-throws-java/.