**JAVA PROGRAMMING**

**E BOX**

**A TRAINING REPORT**

Submitted in partial Fulfillment of the requirements for the award of degree of

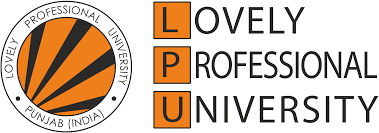
**B.TECH**

**(CSE)**

**Submitted to**

**LOVELY PROFESSIONAL UNIVERSITY**

**PHAGWARA, PUNJAB**



**From : 1ST JUNE, 2020 TO 25TH JUNE,2020**

SUBMITTED BY

**Name of student: SHUBHAM KUMAR**

**Registration Number: 11804686**

**Signature of the student: Shubham kumar**

|  |  |  |
| --- | --- | --- |
| S.NO | TOPIC | PAGE |
| 1 | DECLARATION OF STUDENT | 3 |
| 2 | Chapter-1 INTRODUCTION OF THE PROJECT UNDERTAKEN | 4 |
| 3 | CHAPTER 2: CLASSES AND OBJECTS , CONSTRUCTOR , RELATIONSHIP , COLLECTIONS | 5 |
| 4 | CHAPTER 3:INHERITANCE , POLYMORPHISM , ABSTRACT AND INTERFACE | 27 |
| 5 | CHAPTER 4: STRING , STRING BUFFER AND STRING TOKENIZER | 34 |
| 6 | CHAPTER 5:EXCEPTION HANDLING IN JAVA | 41 |
| 7 | CONCLUSION AND FUTURE SCOPE | 44 |
| 8 | REFERENCE | 47 |

**DECLARATION OF THE STUDENT**

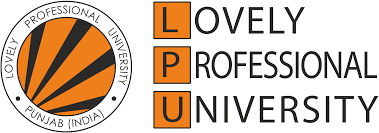
**To whom so ever it may concern**

**I, Shubham kumar, 11804686**, hereby declare that the work done by me on **“java programming”** from **may,2020 to June ,2020**, is a record of original work for the partial fulfillment of the requirements for the award of the degree, **B.TECH.**

Name of the Student : SHUBHAM KUMAR(11804686)

Signature of the student :Shubham kumar

Dated:02/09/2020

 **CHAPTER 1**

**INTRODUCTION OF THE PROJECT UNDERTAKEN**

**Objectives of the work undertaken:**

The main objective of this course is to have the complete knowledge of object oriented programming.

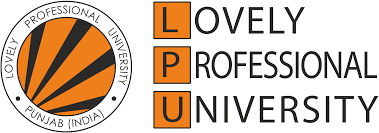
Object-oriented design centers on finding an appropriate set of classes and defining their contents and behaviour . It involves determining the proper use set of classes and then filling in the details of their implementation.

**Scope of the work:**

object-oriented design is fundamentally a three-step process: identifying the classes, characterizing them, and then defining the associated actions. So , this course was mainly focused on the knowledge of object oriented programming and it’s hands on practice by providing different challenges including skills problems and knowledge problems.

**Importance and applicability:**

In this internship,  I had discovered how to create modular, flexible and reusable software, by applying object oriented design principles and guidelines.  It provides me solid knowledge of methods and techniques in Object Oriented Design and Programming. Since JAVA is the most demandable language for developers and Industry , It will help me in solving different problem based questions , data structures , can be used in android app development .

 **CHAPTER 2**

**[Classes and Objects](javascript:void(0);)**

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

* **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
* **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

## **Objects in Java**

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

**Classes in Java**

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

Example

public class Dog {

String breed;

int age;

String color;

void barking() {

}

void hungry() {

}

void sleeping() {

}

}

A class can contain any of the following variable types.

* **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
* **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

**Creating an Object**

As we know that , a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

* **Declaration** − A variable declaration with a variable name with an object type.
* **Instantiation** − The 'new' keyword is used to create the object.
* **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

Example

public class Puppy {

public Puppy(String name) {

// This constructor has one parameter, name.

System.out.println("Passed Name is :" + name );

}

public static void main(String []args) {

// Following statement would create an object myPuppy

Puppy myPuppy = new Puppy( "tommy" );

}

}

If we compile and run the above program, then it will produce the following result −

**Output**

Passed Name is :tommy

**Accessing Instance Variables and Methods**

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path −

/\* First create an object \*/

ObjectReference = new Constructor();

/\* Now call a variable as follows \*/

ObjectReference.variableName;

/\* Now you can call a class method as follows \*/

ObjectReference.MethodName();

Example

This example explains how to access instance variables and methods of a class.

public class Puppy {

int puppyAge;

public Puppy(String name) {

// This constructor has one parameter, name.

System.out.println("Name chosen is :" + name );

}

public void setAge( int age ) {

puppyAge = age;

}

public int getAge( ) {

System.out.println("Puppy's age is :" + puppyAge );

return puppyAge;

}

public static void main(String []args) {

/\* Object creation \*/

Puppy myPuppy = new Puppy( "tommy" );

/\* Call class method to set puppy's age \*/

myPuppy.setAge( 2 );

/\* Call another class method to get puppy's age \*/

myPuppy.getAge( );

/\* You can access instance variable as follows as well \*/

System.out.println("Variable Value :" + myPuppy.puppyAge );

}

}

If we compile and run the above program, then it will produce the following result −

**Output**

Name chosen is :tommy

Puppy's age is :2

Variable Value :2

**CONSTRUCTORS IN JAVA**

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

**Syntax**

Following is the syntax of a constructor −

class ClassName {

ClassName() {

}

}

Java allows two types of constructors namely −

* No argument Constructors
* Parameterized Constructors

**No argument Constructors**

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

**Example**

Public class MyClass {

Int num;

MyClass() {

num = 100;

}

}

You would call constructor to initialize objects as follows

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

System.out.println(t1.num + " " + t2.num);

}

}

This would produce the following result

100 100

**Parameterized Constructors**

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

**STATIC AND FINAL KEYWORD**

static means there is only one copy of the variable in memory shared by all instances of the class.

The final keyword just means the value can't be changed. Without final, any object can change the value of the variable.

The classes and interfaces defined within the Collection Framework act as a container for objects; each contained object is called an element of the collection. In essence, an element is a container for group objects. However, in that sense the simple array is an efficient way to store and retrieve group of object references and primitive values. Then, why do we need a collection, after all? There are two primary reasons. Firstly, arrays are fixed in size and cannot expand or shrink dynamically according to the content. Secondly, compile-time type checking does not allow an array to store values of different kinds. In fact, we can pick up numerous advantages/disadvantages if we pursue more on the comparative review. Let's not do it here; instead, let's explore the core concepts behind a few of the collection objects in the [Java](https://www.developer.com/java/) framework.

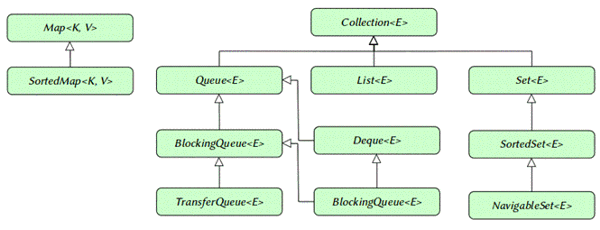
**The Collection Framework**

The Collection Framework is composed of three main components, such as:

* **Interfaces**, which represent a specific type of collection in the framework. There is typically one collection interface defined for every collection type. For example, the List interface, which represents an ordered collection, is the parent interface of all the sub classes and sub interfaces, such as ArrayList, LinkedList, and so forth. Similarly, the Set interface defines the set that does not allow duplicate elements. Its sub implementations are Hashset, TreeSet, and so on. The Map interface represents the collection of elements that map keys to values in a one-to-one mapping fashion.
* **Implementation classes** represent the concrete implementation of some collection interfaces, such as ArrayList, LinkedList, and the like, which are nothing but the sub implementation of the List interface according to various standard data structures.
* **Algorithms classes** provide the means to apply different types of algorithms to the collection classes. For example, we often need to search, sort, or copy elements in the collection or maybe convert a collection of one type to another, and so forth. Realize that this type of operation requires a standard implementation of some frequently used algorithms in the library. Algorithms classes supply exactly the required methods that are used readily with collection instances.

**The Collection Hierarchy**

The collection hierarchy is maintained in the following manner. Figure 1 illustrates a few of the common interfaces in the hierarchy.

  
**Figure 1.1:** The collection hierarchy

The framework also contains many concrete classes that provide the implementation of the collection interfaces. Suppose we want to create an object of List interface. We may instantiate ArrayList in the following manner.

List<String> countries=new ArrayList<>();

countries.add("India");

countries.add("USA");

countries.add("Japan");

We may iterate over the contents in the following manner.

for(String c:countries)

System.out.println(c);

Or, we may use an iterator to iterate over the content as follows.

Iterator<String> iter=countries.iterator();

while(iter.hasNext())

System.out.println(iter.next());

**Iterator<E> and Enumeration<E> Interfaces**

The Iterator<E> interface acts as a cursor to traverse through collection elements. There is another similar interface, called Enumeration<E>. Iterator<E> is a replacement of Enumeration<E> in the collection framework since version 1.2. Enumeration<E> primarily provides a nextElement() method to return successive elements of the series, similar to the Iterator<E> example code above. Iterator<E> is rather a compact version of it. It not only shortened the method names, like hasNextElement(), nextElemen() of Enumeration<E> to hasNext() and next() respectively in Iterator<E>,s but also added another method, called remove(). The remove() method removes the last element returned by the iterator from underlying collection. It is recommended to use Iterator<E> in place of Enumeration<E> in the collection framework.

There is a specific iterator for List, called ListIterator<E>. ListIterator<E> provides methods for bidirectional traversing, element insertion and replacement, apart from normal operations of Iterator<E>. These features are absent in both Iterator<E> and Enumeration<E>.

**List<E> and Set<E> Interfaces**

Both the interfaces represent a collection of elements but with some specific differences.

List<E> maintains an ordered sequence of elements with a precise control over element insertion point in the collection. Note that, like the arrays list index, it also begins with 0. For example, lists of elements say,

List<String> colors=new ArrayList<>();

colors.add("Red");

colors.add("Green");

colors.add("Green"); //Duplicate? OK

colors.add("Blue");

colors.add("violet");

To insert elements at a specific index, we may use an overloaded add method as follows.

colors.add(4,"purple");

colors.add(2,"maroon");

There is a similar method, called set. It can be used as follows.

colors.set(1, "light green");

The main difference between these two methods is that the eadd method inserts elements at a specified index and, if an element exists at that index, it simply shifts all the elements one step forward in the collection sequence and then inserts the new element. The set method, on the other hand, is used specifically to replace an existing element at the specified place in the index with a new element.

Searching an element in an unknown index in the list can be done with the following method:

int indexOf(Object o)

Searching is a costly operation because, in many operations, it will implement a linear search. A linear search in a large collection can bog down performance.

The set<E> maintains a collection of elements that are unique, with no control over the insertion point by the index.

Set<String> aves=new HashSet<>();

aves.add("Hornbill");

aves.add("swallow");

aves.add("sparrow");

aves.add("kite");

Unique elements in the collection means that following two add operation with duplicate elements will actually add only one element in the collection. The duplicate one will simply be ignored. This also means that there can be only one null element in the collection.

aves.add("macaw");

aves.add("macaw"); //Duplicate? No effect

Observe, List<E>, however, allows duplicate elements in the collection.

Because Set<E> is modeled upon mathematical set abstraction, the set operations such as union, intersection, and set-difference (minus) are applicable as follows:

Set<String> A=new HashSet<>();

aves.add("A");

aves.add("B");

aves.add("C");

aves.add("D");

aves.add("E");

Set<String> B=new LinkedHashSet<>();

aves2.add("A");

aves2.add("B");

aves2.add("F");

aves2.add("G");

**Union Operation**

A.addAll(B); // A = AUB

### **Intersection Operation**

A.retainAll(B); // A = A∩B

### **Set Difference Operation**

A.removeAll(B); // A = A-B

To understand the difference, the keywords to remember List<E> and Set<E> are as follows:

* List<E>: **index-able** collection
* Set<E>: **no-duplicate** collection

**Map<K, V> Interfaces**

It's a key, value pair collection. That means that every element in the collection is uniquely identified/mapped by a key. This collection can be visualized easily as a table with two columns. The first column represents the keys and the second column contains the values associated with the keys. For example, the key value map of a collection of employee names and their phone numbers can be represented as follows.

|  |  |
| --- | --- |
| **Key** | **Value** |
| Tom | (111)123-4567 |
| Dick | (222)123-7890 |
| Harry | (333)373-3703 |

Each key is mapped to exactly one value. The key must be unique, although there is no restriction on duplicate values. There can be only one null value as the key but multiple values as the null value. A simple implementation can be as follows.

Map<String,String> map = new HashMap<>();

map.put("Tom", "(111)123-4567");

map.put("Dick", "(222)123-7890");

map.put("Harry", "(333)373-3703");

String dp = map.get("Dick");

System.out.println("Map: " + map);

System.out.println("Map Size: " + map.size());

System.out.println("Map is empty: "

+ map.isEmpty());

System.out.println("Map contains Dick key: "

+ map.containsKey("Dick"));

System.out.println("Dick Phone: " + dp);

System.out.println("Dick key is removed: "

+ map.remove("Dick"));

map.clear(); // removes all elements in the map

**Conclusion**

The collection framework contains numerous interfaces and classes that are used with a wide range of collection types. The List<E>, Set<E>, and Map<K, V> are the basic building blocks in the collection hierarchy. All the collection-related interfaces and classes are grouped together in the java.util package. The utility class called Collections contains many static methods of different types of algorithms that can be applied to collection elements on a specific purpose. As always, the best source of more information on collection framework is the Java Documentation itself.

**RELATIONSHIPS IN JAVA**

**Association, Composition and Aggregation in Java**



Figure 1.2

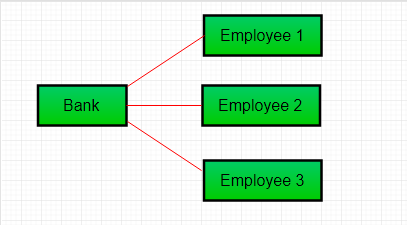
**Association**

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.  
In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.

|  |
| --- |
| Java program to illustrate the  concept of Association  import java.io.\*;    class Bank  {      private String name;        // bank name      Bank(String name)      {          this.name = name;      }        public String getBankName()      {          return this.name;      }  }    // employee class  class Employee  {      private String name;        // employee name      Employee(String name)      {          this.name = name;      }        public String getEmployeeName()      {          return this.name;      }  }    // Association between both the  // classes in main method  class Association  {      public static void main (String[] args)      {          Bank bank = new Bank("Axis");          Employee emp = new Employee("Neha");            System.out.println(emp.getEmployeeName() +                 " is employee of " + bank.getBankName());      }  } |

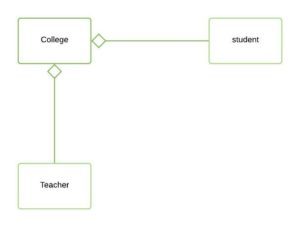
Output:

Neha is employee of Axis

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.  
[](https://media.geeksforgeeks.org/wp-content/uploads/Aggre.png)

Fgure 1.3

**Aggregation**



*Figure 1.4 Aggregation*

It is a special form of Association where:

* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

|  |
| --- |
| // Java program to illustrate the concept of Aggregation.  import java.io.\*;  import java.util.\*;    // student class  class Student  {      String name;      int id ;      String dept;        Student(String name, int id, String dept)      {            this.name = name;          this.id = id;          this.dept = dept;        }  }    /\* Department class contains list of student  Objects. It is associated with student  class through its Object(s). \*/  class Department  {        String name;      private List<Student> students;      Department(String name, List<Student> students)      {            this.name = name;          this.students = students;        }        public List<Student> getStudents()      {          return students;      }  }    /\* Institute class contains list of Department  Objects. It is asoociated with Department  class through its Object(s).\*/  class Institute  {        String instituteName;      private List<Department> departments;        Institute(String instituteName, List<Department> departments)      {          this.instituteName = instituteName;          this.departments = departments;      }        // count total students of all departments      // in a given institute      public int getTotalStudentsInInstitute()      {          int noOfStudents = 0;          List<Student> students;          for(Department dept : departments)          {              students = dept.getStudents();              for(Student s : students)              {                  noOfStudents++;              }          }          return noOfStudents;      }    }    // main method  class UMESH  {      public static void main (String[] args)      {          Student s1 = new Student("Mia", 1, "CSE");          Student s2 = new Student("Priya", 2, "CSE");          Student s3 = new Student("John", 1, "EE");          Student s4 = new Student("Rahul", 2, "EE");            // making a List of          // CSE Students.          List <Student> cse\_students = new ArrayList<Student>();          cse\_students.add(s1);          cse\_students.add(s2);            // making a List of          // EE Students          List <Student> ee\_students = new ArrayList<Student>();          ee\_students.add(s3);          ee\_students.add(s4);            Department CSE = new Department("CSE", cse\_students);          Department EE = new Department("EE", ee\_students);            List <Department> departments = new ArrayList<Department>();          departments.add(CSE);          departments.add(EE);            // creating an instance of Institute.          Institute institute = new Institute("BITS", departments);            System.out.print("Total students in institute: ");          System.out.print(institute.getTotalStudentsInInstitute());      }  } |

Output: Total students in institute: 4

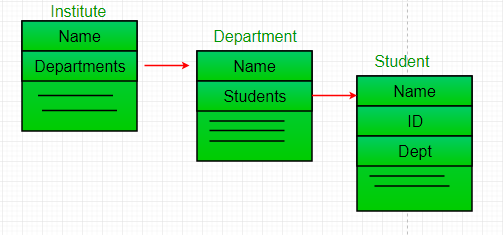
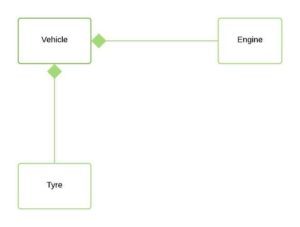
In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).  
It represents a **Has-A** relationship.  
[](https://www.geeksforgeeks.org/media.geeksforgeeks.org/wp-content/uploads/Reference.png)

Figure 1.5

**When do we use Aggregation ??**  
Code reuse is best achieved by aggregation.

**Composition**



*Figure 1.5 Composition*

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist** without the other entity.

Lets take example of**Library**.

|  |
| --- |
| // Java program to illustrate  // the concept of Composition  import java.io.\*;  import java.util.\*;    // class book  class Book  {        public String title;      public String author;        Book(String title, String author)      {            this.title = title;          this.author = author;      }  }    // Libary class contains  // list of books.  class Library  {        // reference to refer to list of books.      private final List<Book> books;        Library (List<Book> books)      {          this.books = books;      }        public List<Book> getTotalBooksInLibrary(){           return books;      }    }    // main method  class UMESH  {      public static void main (String[] args)      {            // Creating the Objects of Book class.          Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");          Book b2 = new Book("Thinking in Java", "Bruce Eckel");          Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");            // Creating the list which contains the          // no. of books.          List<Book> books = new ArrayList<Book>();          books.add(b1);          books.add(b2);          books.add(b3);            Library library = new Library(books);            List<Book> bks = library.getTotalBooksInLibrary();          for(Book bk : bks){                System.out.println("Title : " + bk.title + " and "              +" Author : " + bk.author);          }      }  } |

**Output**

Title : EffectiveJ Java and Author : Joshua Bloch

Title : Thinking in Java and Author : Bruce Eckel

Title : Java: The Complete Reference and Author : Herbert Schildt

In above example a library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That’s why it is composition.

**Aggregation vs Composition**

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.
3. **Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak** Association.

|  |
| --- |
| // Java program to illustrate the  // difference between Aggregation  // Composition.    import java.io.\*;    // Engine class which will  // be used by car. so 'Car'  // class will have a field  // of Engine type.  class Engine  {      // starting an engine.      public void work()      {            System.out.println("Engine of car has been started ");        }    }    // Engine class  final class Car  {        // For a car to move,      // it need to have a engine.      private final Engine engine; // Composition      //private Engine engine;     // Aggregation        Car(Engine engine)      {          this.engine = engine;      }        // car start moving by starting engine      public void move()      {            //if(engine != null)          {              engine.work();              System.out.println("Car is moving ");          }      }  }    class UMESH  {      public static void main (String[] args)      {            // making an engine by creating          // an instance of Engine class.          Engine engine = new Engine();            // Making a car with engine.          // so we are passing a engine          // instance as an argument while          // creating instace of Car.          Car car = new Car(engine);          car.move();        }  } |

Output:

Engine of car has been started

Car is moving

In case of aggregation, the Car also performs its functions through an Engine. but the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That’ why we make The Engine type field non-final.

Shape

Description automatically generated  **CHAPTER 3**

**INHERITANCE , POLYMORPHISM , ABSTRACT AND INTERFACE**

**Inheritance**

* Inheritance is the mechanism by which an object acquires the some/all properties of another object.
* It supports the concept of hierarchical classification.

For example: Car is a four wheeler vehicle so assume that we have a class FourWheeler and a sub class of it named Car. Here Car acquires the properties of a class FourWheeler. Other classifications could be a jeep, tempo, van etc. FourWheeler defines a class of vehicles that have four wheels, and specific range of engine power, load carrying capacity etc. Car (termed as a sub-class) acquires these properties from FourWheeler, and has some specific properties, which are different from other classifications of FourWheeler, such as luxury, comfort, shape, size, usage etc.

A car can have further classification such as an open car, small car, big car etc, which will acquire the properties from both Four Wheeler and Car, but will still have some specific properties. This way the level of hierarchy can be extended to any level.

[Java Swing](https://beginnersbook.com/2015/07/java-swing-tutorial/) and [Awt](https://beginnersbook.com/2015/06/java-awt-tutorial/) classes represent best examples for inheritance.

**Polymorphism**

* Polymorphism means to process objects differently based on their data type.
* In other words it means, one method with multiple implementation, for a certain class of action. And which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the object)
* This can be implemented by designing a generic interface, which provides generic methods for a certain class of action and there can be multiple classes, which provides the implementation of these generic methods.

Lets us look at same example of a car. A car have a gear transmission system. It has four front gears and one backward gear. When the engine is accelerated then depending upon which gear is engaged different amount power and movement is delivered to the car. The action is same applying gear but based on the type of gear the action behaves differently or you can say that it shows many forms (polymorphism means many forms)

Polymorphism could be static and dynamic both. [Method Overloading](https://beginnersbook.com/2013/05/method-overloading/) is static polymorphism while, [Method overriding](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) is dynamic polymorphism.

* **Overloading** in simple words means more than one method having the same method name that behaves differently based on the arguments passed while calling the method. This called static because, which method to be invoked is decided at the time of compilation
* **Overriding** means a derived class is implementing a method of its super class. The call to overriden method is resolved at runtime, thus called runtime polymorphism

**Abstract class vs Interface**

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables:**Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can’t provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”.
6. **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

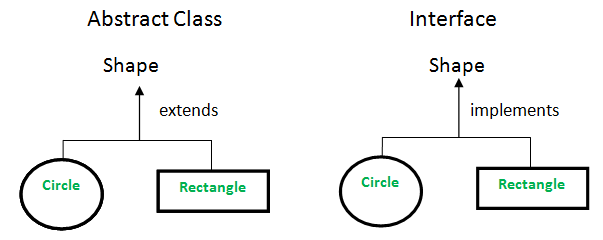


Figure 2.1 Figure 2.2

|  |
| --- |
| Java program to illustrate the concept of abstract class    import java.io.\*;    // abstract class  abstract class Shape  {      // declare fields      String objectName = " ";        Shape(String name)      {          this.objectName = name;      }        // declare non-abstract methods      // it has default implementation      public void moveTo(int x, int y)      {          System.out.println(this.objectName + " " + "has been moved to"                                     + " x = " + x + " and y = " + y);      }        // abstract methods which will be      // implemented by its subclass(es)      abstract public double area();      abstract public void draw();  }    class Rectangle extends Shape  {        int length, width;        // constructor      Rectangle(int length, int width, String name)      {            super(name);          this.length = length;          this.width = width;      }        @Override      public void draw()      {          System.out.println("Rectangle has been drawn ");      }        @Override      public double area()      {          return (double)(length\*width);      }  }    class Circle extends Shape  {        double pi = 3.14;      int radius;        //constructor      Circle(int radius, String name)      {            super(name);          this.radius = radius;      }        @Override      public void draw()      {            System.out.println("Circle has been drawn ");      }        @Override      public double area()      {          return (double)((pi\*radius\*radius)/2);      }  }    class UMESH  {      public static void main (String[] args)      {            // creating the Object of Rectangle class          // and using shape class reference.          Shape rect = new Rectangle(2,3, "Rectangle");          System.out.println("Area of rectangle: " + rect.area());          rect.moveTo(1,2);            System.out.println(" ");            // creating the Objects of circle class          Shape circle = new Circle(2, "Cicle");          System.out.println("Area of circle: " + circle.area());          circle.moveTo(2,4);        }  } |

Output:

Area of rectangle: 6.0

Rectangle has been moved to x = 1 and y = 2

Area of circle: 6.28

Cicle has been moved to x = 2 and y = 4

|  |  |
| --- | --- |
| Java program to illustrate the concept of interface  import java.io.\*;    interface Shape  {      // abstract method      void draw();      double area();  }    class Rectangle implements Shape  {      int length, width;        // constructor      Rectangle(int length, int width)      {          this.length = length;          this.width = width;      }        @Override      public void draw()      {          System.out.println("Rectangle has been drawn ");      }        @Override      public double area()      {          return (double)(length\*width);      }  }    class Circle implements Shape  {        double pi = 3.14;      int radius;        //constructor      Circle(int radius)      {            this.radius = radius;      }        @Override      public void draw()      {          System.out.println("Circle has been drawn ");      }        @Override      public double area()      {            return (double)((pi\*radius\*radius)/2);      }    }    class UMESH  {      public static void main (String[] args)      {       // creating the Object of Rectangle class          // and using shape interface reference.          Shape rect = new Rectangle(2,3);          System.out.println("Area of rectangle: " + rect.area());            // creating the Objects of circle class          Shape circle = new Circle(2);          System.out.println("Area of circle: " + circle.area());      }  }  OUTPUT:Area of rectangle is 6.0 , area of circle is 6.28  Shape  Description automatically generated **CHAPTER 4**    **STRING , STRING BUFFER AND STRING TOKENIZER**  **String**  In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:  char[] ch={'j','a','v','a','t','p','o','i','n','t'};  String s=new String(ch);  is same as:  String s="java";  Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.  The java.lang.String class implements Seriazable , Comparabale  and Char Sequencce [interfaces](https://www.javatpoint.com/interface-in-java).  String in Javafigure 3.1  What is String in java  Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.  How to create a string object?  There are two ways to create String object:  By string literal  By new keyword  1) String Literal  Java String literal is created by using double quotes. For Example:  String s="welcome";  Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:  String s1="Welcome";  String s2="Welcome";//It doesn't create a new instance  Java string literalFigure 3.2  In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.  Note: String objects are stored in a special memory area known as the "string constant pool".  Why Java uses the concept of String literal?  To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).  2) By new keyword  String s=new String("Welcome");//creates two objects and one reference variable  In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).  Java String Example  public class StringExample{  public static void main(String args[]){  String s1="java";//creating string by java string literal  char ch[]={'s','t','r','i','n','g','s'};  output  java  strings  example  System.out.println(s1);  System.out.println(s2);  System.out.println(s3);  }}  The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.  **StringBuffer and StringBuilder**  Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.  A list of differences between StringBuffer and StringBuilder are given below:  StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.  StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.  StringBuffer is less efficient than StringBuilder.  StringBuilder is more efficient than StringBuffer.    StringBuffer Example  Java Program to demonstrate the use of StringBuffer class.  public class BufferTest{      public static void main(String[] args){          StringBuffer buffer=new StringBuffer("hello");          buffer.append("java");          System.out.println(buffer);      }  }  hellojava  StringBuilder Example  //Java Program to demonstrate the use of StringBuilder class.  public class BuilderTest{      public static void main(String[] args){          StringBuilder builder=new StringBuilder("hello");          builder.append("java");          System.out.println(builder);      }  }  hellojava  Performance Test of StringBuffer and StringBuilder  Let's see the code to check the performance of StringBuffer and StringBuilder classes.  Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.  public class ConcatTest{      public static void main(String[] args){          long startTime = System.currentTimeMillis();          StringBuffer sb = new StringBuffer("Java");          for (int i=0; i<10000; i++){              sb.append("Tpoint");          }          System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) + "ms");          startTime = System.currentTimeMillis();          StringBuilder sb2 = new StringBuilder("Java");          for (int i=0; i<10000; i++){              sb2.append("Tpoint");          }          System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) + "ms");      }  }  Time taken by StringBuffer: 16ms  Time taken by StringBuilder: 0ms  **String tokenizer**  StringTokenizer class in Java is used to break a string into tokens.  A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the StringTokenizer object.  Constructors:  StringTokenizer(String str) :  str is string to be tokenized.  Considers default delimiters like new line, space, tab,  carriage return and form feed.  StringTokenizer(String str, String delim) :  delim is set of delimiters that are used to tokenize  the given string.  StringTokenizer(String str, String delim, boolean flag):  The first two parameters have same meaning. The flag  serves following purpose.  If the flag is false, delimiter characters serve to  separate tokens. For example, if string is "hello geeks"  and delimiter is " ", then tokens are "hello" and "geeks".  If the flag is true, delimiter characters are  considered to be tokens. For example, if string is "hello  kk" and delimiter is " ", then tokens are "hello", " "  and "kk".  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | A Java program to illustrate working of StringTokenizer    import java.util.\*;  public class NewClass  {      public static void main(String args[])      {          System.out.println("Using Constructor 1 - ");          StringTokenizer st1 =               new StringTokenizer("Hello kk How are you", " ");          while (st1.hasMoreTokens())              System.out.println(st1.nextToken());            System.out.println("Using Constructor 2 - ");          StringTokenizer st2 =               new StringTokenizer("JAVA : Code : String", " :");          while (st2.hasMoreTokens())              System.out.println(st2.nextToken());            System.out.println("Using Constructor 3 - ");          StringTokenizer st3 =               new StringTokenizer("JAVA : Code : String", " :",  true);          while (st3.hasMoreTokens())              System.out.println(st3.nextToken());      }  } |   Output :  Using Constructor 1 -  Hello  kk  How  are  you  Using Constructor 2 -  JAVA  Code  StringUsing Constructor 3 –  JAVA    :    Code    :    String |

Shape

Description automatically generated **CHAPTER 5**

**EXCEPTION HANDLING IN JAVA**

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions. In this guide, we will learn what is an exception, types of it, exception classes and how to handle exceptions in java with examples.

What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:  
An exception generated by the system is given below

Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:5)

ExceptionDemo : The class name

main : The method name

ExceptionDemo.java : The filename

java:5 : Line number

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

Advantage of exception handling

Exception handling ensures that the flow of the program doesn’t break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.  
By handling we make sure that all the statements execute and the flow of program doesn’t break.

Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:  
NullPointerException – When you try to use a reference that points to null.  
ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.  
ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

Diagram

Description automatically generated

FIGURE 4.1

EXAMPLE :

public class MyClass {

public static void main(String[ ] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

}

}

}

Types of exceptions

There are two types of exceptions in Java:  
1)Checked exceptions  
2)Unchecked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Compiler will never force you to catch such exception or force you to declare it in the method using throws keyword.

Shape

Description automatically generated **Final Chapter- CONCLUSION AND FUTURE PRESPECTIVE**

Java is everywhere, in laptops, scientific supercomputers, gaming consoles, mobile phones, etc. It’s been quoted that there are 9 million Java developers in the world. This lets you know the demand of Java and its evolution in software development in future. In this blog, I will take you through the following topics:

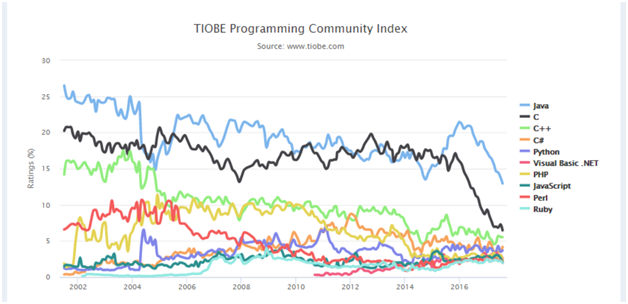
[Java](https://www.janbasktraining.com/blog/java-tutorial/) is not only easy to learn but is also is designed in such a way that it is easy to use, easy to write, compile, debug, and learn as compared to other programming languages. It also allows you to create reusable code and modular programs.  In the screenshot below, let’s have a look at the data trends which shows how Java has dominated this field from early 2000’s till the present.

FIGURE 7.1

As you can see in the above Java is widely used in the industry and is highly popular. Some of the different domains where Java is used widely are as follows:

Financial services: It is used in server-side applications.

Big Data: Hadoop MapReduce framework is written using Java.

Banking: To deal with transaction management.

Stock market: To write algorithms as to which company they should invest in.

Retail: Billing applications that you see in a store/restaurant are completely written in Java.

Android: Applications are either written in Java or use Java API.

Scientific and Research Community: To deal with huge amount of data

Some of the technologies use java as a vital core of their functionalities.

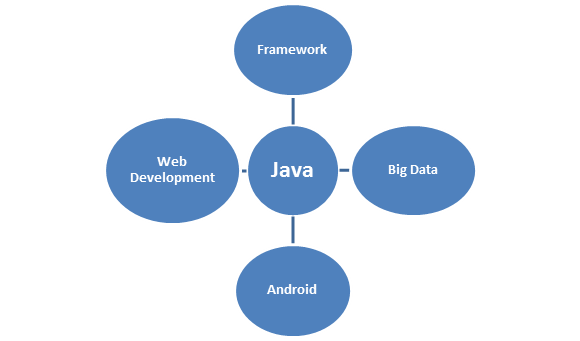


FIGURE 7.2

Java is most widely used programming language. It is present everywhere. It is the ocean of opportunities. Whichever domain you work in you would surely come across Java Programming.

Core Features Of Java Programming

1). Security:

Java is a secured language as there are no explicit pointers used and all the programs are run inside the sandbox to protect them from any un-trusted sources. Java develops a virus-free system and converts all the codes into bytecodes which are not easily readable by humans.

2). Portable:

The key feature of Java is its portability as it can run on all the operating systems without any dependencies. Java is platform independent which means that any application written on one platform can be easily ported to another platform.

According to SUN Micro System:

Portability = Platform independent + Architecture

3). Robust:

Java’s strong memory management system helps in eliminating errors by checking the codes during runtime. Java Programming is robust as it completely takes care of memory allocation and releasing.

4). Object-oriented:

All the functions in Java are performed using objects. Thus Java is an object-oriented programming language. All these objects possess certain behavior. Hence, it is the most used language as it supports [OOP’s concepts](https://www.janbasktraining.com/blog/what-is-java-oops-concept/). So, above mentioned are the few important features or services provided by the Java Programming language vendors to the industry programmers.

Exciting Career Opportunities With Java Programming

More than 22 years of Java and the success party continues. To all those who said that Java is slipping in popularity, job trends have responded saying ‘not yet’ Java continues to grow in popularity and usage and is believed to be the most reliable programming language in the developer community. Java is currently leading the pack in the Android mobile application development market and enterprise back-end market. When you consider the integration of modern language features the future for Java shines brightly. In accordance with the reports from Oracle, there are more than nine million Java developers in the world who continue to work with Java for reasons such as reliability, practicality, and compatibility.

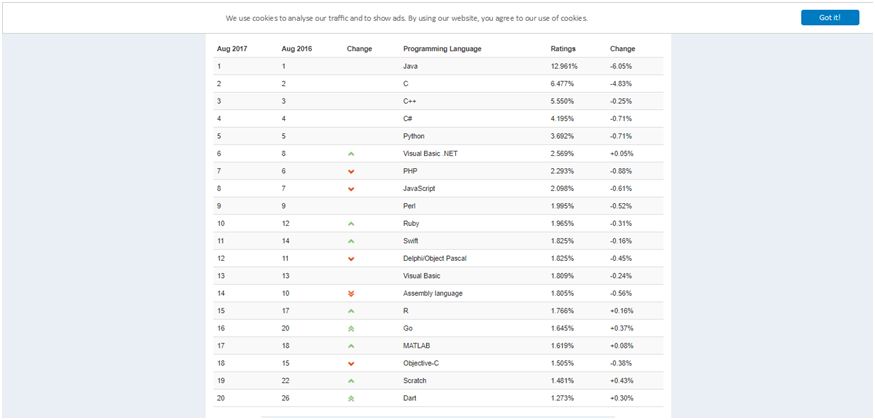
Have a look at the popularity of Java Programming as compared to other programming languages:

FIGURE 7.3

As you can see in the above screenshot, the popularity of Java among all other programming languages is the highest. The average salary for a Java Developer in the USA is around $102,000 with salaries for job postings nationwide being 77% higher than average salaries.

Java Job Roles and Java Programmer Skills

There are a lot of [Java career opportunities](https://www.janbasktraining.com/blog/java-developer-role/) all over the globe. A quick search reveals that are may giant enterprise looking for Java developers. This is the right to take advantage of the Java career opportunities that come your way.

Java Programmer Skills:

With high-level programming and devoting long working hours, a [Java programmer needs](https://www.janbasktraining.com/blog/java-skills-attractive-opportunity-employment-youth/) to be well versed with the following skills:

Enterprise Java Beans

Oracle database SQL and JDBC

XML,X query, XSL

J2EE framework

JSP

Service Oriented Architecture

Java based Web services

Java Servlet Technology

Applications for a wide range of environments, from consumer devices to heterogeneous enterprise systems are being developed by Java Programming. Java Programming helps you in developing, deploying and using all applications and services with 9 million developers worldwide.

**REFERENCE**

1.www.tutorialspoint.com/java

2.www.javatpoint.com/java

3. **Core Java Volume I:Advanced Features(horstmann cornell)**

1