OOP through JAVA (JNTUK-R19-2-1-ECE)

UNIT II: INHERITANCE AND POLYMORPHISM

Syllabus:

Inheritance in java, Super and sub class, Overriding, Object class, Polymorphism, Dynamic binding, Generic programming, Casting objects, Instance of operator, Abstract class, Interface in java, Package in java, UTIL package.

1. INHERITANCE IN JAVA:

- Inheritance is the way of producing new classes from already existing classes.

OR

- Inheritance is the process by which objects of one class acquire the properties of objects of another class
- Inheritance supports the concept of hierarchical classification.
- The newly created class is also called as sub class or child class or derived class.
- The old or existing class is also called as super class or parent class or base class.

Types of Inheritance:

- Inheritance is of 5 types

i) Single Inheritance
ii) Multi-level Inheritance
iii) Multiple Inheritance
iv) Hierarchical Inheritance
v) Hybrid Inheritance

- Java does not support Hybrid Inheritance.

i) Single Inheritance:

- Single Inheritance representing one super class and one sub class.

Example:

```
class A
 {
 int a=10;
 void display()
  {
   System.out.println("a="+a);
  }
 }
class B extends A
 {
 }
class Single
{
public static void main(String args[])
 {
 B obj=new B();
```

```
    obj.display();
   }
 }
```

Output:

```
a=10
```

ii) Multi-level Inheritance:

- Multi-level inheritance representing a sub class derived from a sub class derived from a super class

Example:

```
class A
 {
  int a=10;
  void display1()
   {
    System.out.println("a="+a);
   }
 }
class B extends A
 {
  int b=20;
  void display2()
   {
    System.out.println("b="+b);
   }
 }
class C extends B
 {
 }
class Multilevel
 {
  public static void main(String args[])
   {
    C obj=new C();
    obj.display1();
    obj.display2();
   }
 }
```

Output:

```
a=10
b=20
```

iii) Multiple Inheritance:

- Multiple inheritance representing multiple super classes and one sub class.
- Java does not support multiple inheritance directly.
- Java provides an interface concept to support the concept of multiple inheritance.

Example:

```
 interface Car
 {
    int  speed=60;
    public void distanceTravelled();
 }
 interface Bus
 {
    int distance=100;
    public void speed();
 }
 public class Vehicle  implements Car,Bus
 {
    int DT;
    int ASP;
    public void distanceTravelled()
    {
       DT=speed*distance;
       System.out.println("Total Distance Travelled is : "+DT);
    }
   public void speed()
   {
      int ASP=DT/speed;
      System.out.println("Average Speed maintained is : "+ASP);
   }
  public static void main(String args[])
  {
     Vehicle v1=new Vehicle();
     v1.distanceTravelled();
     v1.speed();
  }
}
```

Output:

```
Total Distance Travelled is : 6000
Average Speed maintained is : 100
```

iv) Hierarchical Inheritance:

- Hierarchical inheritance representing one super class and multiple sub classes.

Example:

```
class A
 {
  int a=10;
  void display()
   {
    System.out.println("a="+a);
   }
 }
class B extends A
 {
 }
class C extends A
 {
 }
class D extends A
 {
 }
class Hierarchical
 {
  public static void main(String args[])
   {
    B b1=new B();
    C c1=new C();
    D d1=new D();
    b1.display();
    c1.display();
    d1.display():
   }
 }
```

Output:

```
a=10
a=10
a=10
```

2. SUPER CLASS AND SUB CLASS:

- The newly created class is also called as sub class or child class or derived class.
- The old or existing class is also called as super class or parent class or base class.

i) Super Keyword in Java:

- super is a keyword in java which refers to the immediate super class object
- super can be used to refer immediate parent class instance variable
- super can be used to invoke immediate parent class method

- super() can be used to invoke immediate parent class constructor

Example1:

```
class A
{
    A()
    {
      System.out.println("Super class");
    }
}
class  B extends A
{
    B()
    {
      super();
      System.out.println("Current class");
    }
}
class SuperDemo
{
   public static void main(String args[])
   {
      B obj = new B();
   }
}
```

Output:

```
Super class
Current class
```

Example2:

```
class Person
{
 int id;
 String name;
 Person(int id,String name)
 {
  this.id=id;
  this.name=name;
 }
}
class Emp extends Person
{
 float salary;
 Emp(int id,String name,float salary)
```

```
  {
   super(id,name);//reusing parent constructor
   this.salary=salary;
  }
 void display()
  {
   System.out.println(id+" "+name+" "+salary);
  }
}
class SuperDemo2
{
 public static void main(String[] args)
  {
   Emp e1=new Emp(1,"RGV",30000f);
   e1.display();
  }
}
```

Output:

```
1   RGV   30000.0
```

3. METHOD OVERRIDING:

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java.**

- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding:

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Example:

```
class Parent
{
  void show() { System.out.println("Parent's show()"); }
}
class Child extends Parent
{
  void show() { System.out.println("Child's show()"); }
}
class Main
```

```
{
   public static void main(String[] args)
   {
      Parent obj1 = new Parent();
      obj1.show();
      Parent obj2 = new Child();
      obj2.show();
   }
}
```

Output:

```
Parent's show()
Child's show()
```

4. OBJECT CLASS IN JAVA:

- The Object class is the parent class of all the classes in java by default.
- In other words, it is the topmost class of java.
- **Object** class is present in **java.lang** package.
- Every class in Java is directly or indirectly derived from the **Object** class.
- If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived.
- Therefore, the Object class methods are available to all Java classes.
- Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Methods in Object class:

- The Object class provides many methods depicted in a table as,

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes

InterruptedException	notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is beir garbage collected.

5. POLYMORPHISM IN JAVA:

- Polymorphism is a concept by which we can perform a *single action in different ways*.
- Polymorphism is derived from 2 Greek words: poly and morphs.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java:
    - compile-time polymorphism and runtime polymorphism.
- compile-time polymorphism supports overloading and runtime polymorphism supports overriding

6. DYNAMIC BINDING:

- Connecting a method call to the method body is known as binding.
- There are two types of binding
    i) Static Binding (also known as Early Binding).
    ii) Dynamic Binding (also known as Late Binding).
- When type of the object is determined at compiled time(by the compiler), it is known as static binding.
- If there is any private, final or static method in a class, there is static binding.
- When type of the object is determined at run-time, it is known as dynamic binding.

Example (dynamic binding)

```
class Animal{
void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
void eat(){System.out.println("dog is eating...");}

public static void main(String args[]){
Animal a=new Dog();
a.eat();
}
}
```

Output:

```
dog is eating...
```

7. CASTING OBJECTS AND The instanceof OPERATOR:

- One object reference can be typecast into another object reference. This is called casting object.

Example:.

```
        m(new Student());
```

- It assigns the object **new Student()** to a parameter of the **Object** type.
- This statement is equivalent to

```
        Object o = new Student(); // Implicit casting
        m(o);
```

- The statement **Object o = new Student()**, known as implicit casting, is legal because an instance of **Student** is an instance of **Object**.
- Suppose you want to assign the object reference **o** to a variable of the **Student** type using the following statement:

```
        Student b = o;
```

- In this case a compile error would occur. Why does the statement **Object o = new Student()** work but **Student b = o** doesn't?
- The reason is that a **Student** object is always an instance of **Object**, but an **Object** is not necessarily an instance of **Student**.
- Even though you can see that **o** is really a **Student** object, the compiler is not clever enough to know it.
- To tell the compiler that **o** is a **Student** object, use explicit casting.
- The syntax is similar to the one used for casting among primitive data types.
- Enclose the target object type in parentheses and place it before the object to be cast, as follows:

Student b = (Student)o; // Explicit casting

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting), because an instance of a subclass is always an instance of its superclass.
- When casting an instance of a superclass to a variable of its subclass (known as downcasting),
- Explicit casting must be used to confirm your intention to the compiler with the **(SubclassName)** cast notation.

The instanceof Operator:

- The instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right hand side.
- This operator allows us to determine whether the object belongs to a particular class or not.

Example:

```
        person  instanceof  student
```

- It is true if the object person belongs to the class student, otherwise it is false.

8. ABSTRACT CLASS IN JAVA:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

- A method which is declared as abstract and does not have implementation is known as an abstract method.

Example:

```
abstract class shape
{
abstract double area();
}
class rectangle extends shape
{
double l=12.5,b=2.5;
double area()
{
return l*b;
}
}
class triangle extends shape
{
double b=4.2,h=6.5;
double area()
{
return 0.5*b*h;
}
}
class square extends shape
{
double s=6.5;
double area()
{
return 4*s;
}
}
class shapedemo
{
public static void main(String[] args)
{
rectangle r1=new rectangle();
triangle t1=new triangle();
square s1=new square();
System.out.println("The area of rectangle is: "+r1.area());
System.out.println("The area of triangle is: "+t1.area());
System.out.println("The area of square is: "+s1.area());
}
}
```

Output:

```
The area of rectangle is: 31.25
The area of triangle is: 13.65
The area of square is: 26.0
```

9. THE FINAL KEYWORD:

- The **final keyword** in java is used to restrict the user.
- The java final keyword can be used in many context. Final can be:
    a) final variable
    b) final method
    c) final class
- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.
- It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only.

a) final variable:

- If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable:

- There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
 class Bike9
 {
 final int speedlimit=90;//final variable
 void run()
 {
 speedlimit=400;
 }
 public static void main(String args[])
 {
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output:

```
Compile Time Error
```

b) final method:

- If you make any method as final, you cannot override it.

Example of final method:

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

 public static void main(String args[]){
    Honda honda= new Honda();
    honda.run();
    }
  }
```

Output:

Compile Time Error

c) final class:

- If you make any class as final, you cannot extend it.

Example of final class:

```
  final class Bike{}

  class Honda1 extends Bike{
    void run(){System.out.println("running safely with 100kmph");}

    public static void main(String args[]){
    Honda1 honda= new Honda1();
    honda.run();
    }
  }
```

Output:

Compile Time Error

Is final method inherited?

- Yes, final method is inherited but you cannot override it. For Example:

```
  class Bike{
    final void run(){System.out.println("running...");}
  }
  class Honda2 extends Bike{
     public static void main(String args[]){
      new Honda2().run();
     }
  }
```

Output:

running...

10. INTERFACES:

- An interface is a collection of abstract methods and final variables
- By default, the methods declared inside interface are abstract and variables are final.
- By default, all variables and methods inside an interface are public.
- Java provides an interface concept to support the concept of multiple inheritance.

Defining Interfaces:

```
interface  InterfaceName
{
```

```
Variable declarations;
Method declarations;
}
```

Variable declaration in an Interface:

```
        static final type variablename=value;
```

Method declaration in an interface:

```
        returntype methodname(parameter-list);
```

Example:

```
interface Area
{
final static float pi=3.14F;
float compute(float x,float y);
void show();
}
```

Extending Interfaces:

Syntax:

```
interface SubInterfaceName extends SuperInterfaceName
{
Variable declarations;
Method declarations;
}
```

Example:

```
interface A
{ }
interface B extends A
{ }
```

- Interface can be extended from more than one interface also

Example:

```
interface A
{
}
interface B
{
}
interface C extends A,B
{
}
```

Implementing Interfaces:

Syntax1:

```
class classname implements interfacename
{
body of classname;
```

}

Syntax2:

```
class classname extends superclassname implements interface1,interface2,……..
{
body of classname;
}
```

Example Program:

Class implementing Claculator interface

```
interface Calculator
{
 int add(int a,int b);
 int subtract(int a,int b);
 int multiply(int a,int b);
 int divide(int a,int b);
}
class Normal_Calculator implements Calculator
{
 public int add(int a,int b){
 return a + b; }
 public int subtract(int a,int b) {
 return a – b; }
 public int multiply(int a,int b) {
 return a * b; }
 public int divide(int a,int b)
{
 return a / b;
}
 public static void main(String args[])
{
 Normal_Calculator c = new Normal_Calculator();
 System.out.println("Value after addition = "+c.add(5,2));
 System.out.println("Value after Subtraction = " +c.subtract(5,2));
 System.out.println("Value after Multiplication = " +c.multiply(5,2));
 System.out.println("Value after division = " +c.divide(5,2));
}
}
```

Output

```
C:\javabook>java Normal_Calculator
Value after addition = 7
Value after Subtraction = 3
Value after Multiplication= 10
Value after division = 2
```

Implementing Multiple Inheritance:

```
interface Car
{
  int  speed=60;
  public void distanceTravelled();
}
interface Bus
{
  int distance=100;
  public void speed();
}
public class Vehicle  implements Car,Bus
{
  int DT;
  int ASP;
  public void distanceTravelled()
  {
    DT=speed*distance;
    System.out.println("Total Distance Travelled is : "+DT);
  }
  public void speed()
  {
    int ASP=DT/speed;
    System.out.println("Average Speed maintained is : "+ASP);
  }
  public static void main(String args[])
  {
    Vehicle v1=new Vehicle();
    v1.distanceTravelled();
    v1.speed();
  }
}
```

Output:

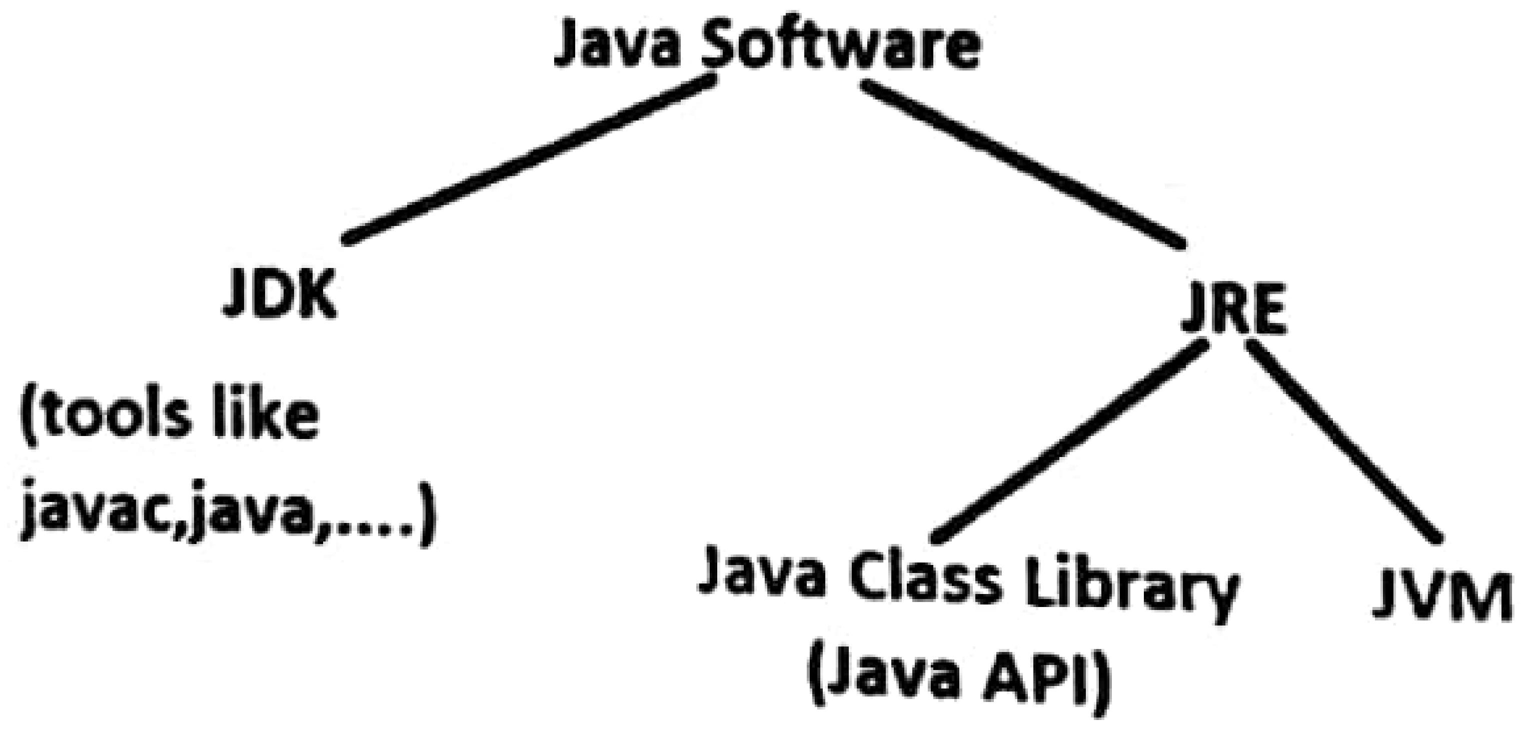
```
Total Distance Travelled is : 6000
Average Speed maintained is : 100
```

11. Abstract Class Vs Interface:

Interface	Abstract Class
Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
implements keyword is used to inherit an interface.	**extends** keyword is used to inherit a class.
By default, all methods in an interface are **public** and **abstract**; no need to tag it as **public** and **abstract**.	Methods have to be tagged as **public** or **abstract** or both, if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default **public**, **static**, or **final**.	Variables, if required, have to be declared as **public, static,** or **final**.
Interfaces do not have any constructors.	Abstract classes can have constructors.
Methods in an interface cannot be static.	Non-abstract methods can be static.

12. PACKAGES:

- A package is a collection of classes, interfaces and sub-packages.
- By using packages, we can reuse the code already we created.
- Packages are java's way of grouping a variety of classes and/or interfaces together.
- **Java API** contains a set of classes and interfaces that are in the form of packages.



Types of packages:

- Java contains two types of packages
  - i) Predefined Packages
  - ii) User-defined Packages

i) Predefined Packages

- Java has many predefined packages.
- A package is contained many predefined classes and interfaces and all these are given by compressing into a single jar file called RT.jar file
- RT.jar file is located at,

C:\jdk1.5\jre\lib\RT.jar

- Commonly used predefined packages are given in a table as,

Package	Functionality
java.lang	Basic language fundamentals
java.util	Utility classes and collection data structure classes
java.io	File handling operations
java.math	Arbitrary precision arithmetic
java.net	Network programming
java.sql	Java Database Connectivity (JDBC) to access databases
java.awt	Abstract window toolkit for native GUI components
javax.swing	Lightweight programming for platform-independent rich GUI components

- The smallest package in java is java.applet
- The biggest package in java is java.awt

Using System or Predefined Packages:

- We use predefined packages using import keyword.

Syntax:

```
import packagename.*;
OR
import packagename.classname;
```

Example:

```
import java.io.*;
import java.lang.Math;
```

ii) User-defined Packages:

- The general form of creating user-defined packages is

package packagename;

Steps to create user-defined packages:

- Create a folder where directory name and package name to be created. Both must be same.

```
c:\>md pack1
```

- Change into created folder

```
c:\>cd pack1
```

- Define the classes and interfaces required in each application or program and write first statemen as package statement

```
package packagename;
```

Example:

One.java

```
package pack1;
public class One
{
```

```
--------
--------
}
```

Two.java

```
package pack1;
public class Two
{
--------
--------
}
```

- Compile all the applications to get .class files. Now the package is created

```
c:\pack1>javac *.java
```

- Finally import this package into the other programs.
  This is called **accessing a package**

Accessing a user-defined package:

Syntax:

```
        import packagename.*;
```

Example:

Sample.java

```
import pack1.*;
class Sample
{
        public static void main(String args[])
        {
                -----------
                -----------
        }
}
```

13. java.lang PACKAGE:

- java.lang is a special package, as it is imported by default in all the classes that we create.
- There is no need to explicitly import the lang package.
- It contains the classes that form the basic building blocks of Java.
- Remember we have been using String and the System class, but we have not imported any package for using these classes, as both these classes lie in the java.lang package.
- Commonly used classes and interfaces are given in a table as,

java.lang	
Interfaces	**Classses**
Comparable	Boolean
Clonable	Byte
Runnable	Class
	Object
	Integer
	Long
	Float
	Enum
	String
	StringBuffer
	StringBuilder
	Thread
	Throwable

- The wrapper classes for primitive types are given in a table as,

Primitive	Wrapper
`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`
`void`	`java.lang.Void`

14. The java.util PACKAGE:

- The package java.util contains a number of useful classes and interfaces.
- Java util package contains collection framework, collection classes, classes related to date a time, event model, internationalization, and miscellaneous utility classes.
- On importing this package, you can access all these classes and methods.
- The classes and interfaces in java.util include:

S.NO	CLASS	PURPOSE
1	Hashtable class	for implementing hashtables, or associative arrays
2	Vector class	which supports variable-length arrays
3	Enumeration interface	for iterating through a collection of elements
4	StringTokenizer class	for parsing strings into distinct tokens separated by delimiter characters
5	EventObject class and the EventListener interface	which form the basis of the new AWT event model in Jav 1.1.
6	Locale class	which represents a particular locale for internationalizatio purposes
7	Calendar and TimeZone classes	interpret the value of a Date object in the context of a particular calendar system
8	ResourceBundle class, ListResourceBundle and PropertyResourceBundle	which represent sets of localized data

15. GENERIC PROGRAMMING IN JAVA:

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.
- It makes the code stable by detecting the bugs at compile time.
- Before generics, we can store any type of objects in the collection, i.e., non-generic.
- Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics:

There are mainly 3 advantages of generics,

i) Type-safety:

- We can hold only a single type of objects in generics.
- It doesn't allow to store other objects.
- Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

ii) Type casting is not required:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

iii) Compile-Time Checking:

- It is checked at compile time so problem will not occur at runtime.
- The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection:

```
ClassOrInterface<Type>
```

Example to use Generics in java:

```
ArrayList<String>
```