New Features in Java

There are many new features that have been added in java. There are major enhancement made in Java5, Java6 and Java7 like **auto-boxing**, **generics**, **var-args**, **java annotations**, **enum**, **premain method** etc.

Most of the interviewers ask questions from this chapter.

#### J2SE 4 Features

The important feature of J2SE 4 is assertions. It is used for testing.

* [Assertion](http://www.javatpoint.com/assertion-in-java) (Java 4)

#### J2SE 5 Features

The important features of J2SE 5 are generics and assertions. Others are auto-boxing, enum, var-args, static import, for-each loop (enhanced for loop etc.

* [For-each loop (Java 5)](http://www.javatpoint.com/for-each-loop)
* [Varargs (Java 5)](http://www.javatpoint.com/varargs)
* [Static Import (Java 5)](http://www.javatpoint.com/static-import-in-java)
* [Autoboxing and Unboxing (Java 5)](http://www.javatpoint.com/autoboxing-and-unboxing)
* [Enum (Java 5)](http://www.javatpoint.com/enum-in-java)
* [Covariant Return Type (Java 5)](http://www.javatpoint.com/covariant-return-type)
* [Annotation (Java 5)](http://www.javatpoint.com/java-annotation)
* [Generics (Java 5)](http://www.javatpoint.com/generics-in-java)

#### JavaSE 6 Features

The important feature of JavaSE 6 is premain method (also known as instrumentation).

* Instrumentation (premain method) (Java 6)

#### JavaSE 7 Features

The important features of JavaSE 7 are try with resource, catching multiple exceptions etc.

* String in switch statement (Java 7)
* Binary Literals (Java 7)
* The try-with-resources (Java 7)
* Caching Multiple Exceptions by single catch (Java 7)
* Underscores in Numeric Literals (Java 7)

# Assertion:

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

## **Advantage of Assertion:**

It provides an effective way to detect and correct programming errors.

## **Syntax of using Assertion:**

There are two ways to use assertion. First way is:

**assert** expression;

and second way is:

**assert** expression1 : expression2;

### **Simple Example of Assertion in java:**

**import** java.util.Scanner;

**class** AssertionExample{

**public** **static** **void** main( String args[] ){

  Scanner scanner = **new** Scanner( System.in );

  System.out.print("Enter ur age ");

**int** value = scanner.nextInt();

**assert** value>=18:" Not valid";

  System.out.println("value is "+value);

 }

}

|  |
| --- |
| If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used. |
| Compile it by: **javac AssertionExample.java** |
| Run it by: **java -ea AssertionExample** |

Output: Enter ur age 11

Exception in thread "main" java.lang.AssertionError: Not valid

### **Where not to use Assertion:**

There are some situations where assertion should be avoid to use. They are:

1. According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g. IllegalArgumentException, NullPointerException etc.
2. Do not use assertion, if you don't want any error in any situation.

# For-each loop (Advanced or Enhanced For loop):

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

## **Advantage of for-each loop:**

* It makes the code more readable.
* It elimnates the possibility of programming errors.

## **Syntax of for-each loop:**

**for**(data\_type variable : array | collection){}

### **Simple Example of for-each loop for traversing the array elements:**

**class** ForEachExample1{

**public** **static** **void** main(String args[]){

**int** arr[]={12,13,14,44};

**for**(**int** i:arr){

     System.out.println(i);

   }

 }

}

Output:12

13

14

44

### **Simple Example of for-each loop for traversing the collection elements:**

**import** java.util.\*;

**class** ForEachExample2{

**public** **static** **void** main(String args[]){

   ArrayList<String> list=**new** ArrayList<String>();

   list.add("vimal");

   list.add("sonoo");

   list.add("ratan");

**for**(String s:list){

     System.out.println(s);

   }

 }

}

Output:vimal

sonoo

ratan

# Variable Argument (Varargs):

The varrags allows the method to accept zero or muliple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

## **Advantage of Varargs:**

We don't have to provide overloaded methods so less code.

## **Syntax of varargs:**

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

return\_type method\_name(data\_type... variableName){}

### **Simple Example of Varargs in java:**

**class** VarargsExample1{

**static** **void** display(String... values){

  System.out.println("display method invoked ");

 }

**public** **static** **void** main(String args[]){

 display();//zero argument

 display("my","name","is","varargs");//four arguments

 }

}

      Output:display method invoked

display method invoked

### **Another Program of Varargs in java:**

**class** VarargsExample2{

**static** **void** display(String... values){

  System.out.println("display method invoked ");

**for**(String s:values){

   System.out.println(s);

  }

 }

**public** **static** **void** main(String args[]){

 display();//zero argument

 display("hello");//one argument

 display("my","name","is","varargs");//four arguments

 }

}

   Output:display method invoked

display method invoked

hello

display method invoked

my

name

is

varargs

## **Rules for varargs:**

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

* There can be only one variable argument in the method.
* Variable argument (varargs) must be the last argument.

## **Examples of varargs that fails to compile:**

**void** method(String... a, **int**... b){}//Compile time error

**void** method(**int**... a, String b){}//Compile time error

### **Example of Varargs that is the last argument in the method:**

**class** VarargsExample3{

**static** **void** display(**int** num, String... values){

  System.out.println("number is "+num);

**for**(String s:values){

   System.out.println(s);

  }

 }

**public** **static** **void** main(String args[]){

 display(500,"hello");//one argument

 display(1000,"my","name","is","varargs");//four arguments

 }

}

Output:number is 500

hello

number is 1000

my

name

is

varargs

# Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

## **Advantage of static import:**

* Less coding is required if you have access any static member of a class oftenly.

## **Disadvantage of static import:**

* If you overuse the static import feature, it makes the program unreadable and unmaintainable.

### **Simple Example of static import**

**import** **static** java.lang.System.\*;

**class** StaticImportExample{

**public** **static** **void** main(String args[]){

   out.println("Hello");//Now no need of System.out

   out.println("Java");

 }

}

Output:Hello

Java

### **What is the difference between import and static import?**

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

# Autoboxing and Unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

## **Advantage of Autoboxing and Unboxing:**

|  |
| --- |
| No need of conversion between primitives and Wrappers manually so less coding is required. |

### **Simple Example of Autoboxing in java:**

**class** BoxingExample1{

**public** **static** **void** main(String args[]){

**int** a=50;

        Integer a2=**new** Integer(a);//Boxing

        Integer a3=5;//Boxing

        System.out.println(a2+" "+a3);

 }

}

Output:50 5

### **Simple Example of Unboxing in java:**

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

**class** UnboxingExample1{

**public** **static** **void** main(String args[]){

    Integer i=**new** Integer(50);

**int** a=i;

        System.out.println(a);

 }

}

Output:50

### **Autoboxing and Unboxing with comparison operators**

|  |
| --- |
| Autoboxing can be performed with comparison operators. Let's see the example of boxing with comparison operator: |

**class** UnboxingExample2{

**public** **static** **void** main(String args[]){

    Integer i=**new** Integer(50);

**if**(i<100){            //unboxing internally

        System.out.println(i);

        }

 }

}

Output:50

### **Autoboxing and Unboxing with method overloading**

|  |
| --- |
| In method overloading, boxing and unboxing can be performed. There are some rules for method overloading with boxing:   * **Widening beats boxing** * **Widening beats varargs** * **Boxing beats varargs** |

### **1) Example of Autoboxing where widening beats boxing**

|  |
| --- |
| If there is possibility of widening and boxing, widening beats boxing. |

**class** Boxing1{

**static** **void** m(**int** i){System.out.println("int");}

**static** **void** m(Integer i){System.out.println("Integer");}

**public** **static** **void** main(String args[]){

**short** s=30;

   m(s);

 }

}

Output:int

### **2) Example of Autoboxing where widening beats varargs**

|  |
| --- |
| If there is possibility of widening and varargs, widening beats var-args. |

**class** Boxing2{

**static** **void** m(**int** i, **int** i2){System.out.println("int int");}

**static** **void** m(Integer... i){System.out.println("Integer...");}

**public** **static** **void** main(String args[]){

**short** s1=30,s2=40;

   m(s1,s2);

 }

}

Output:int int

### **3) Example of Autoboxing where boxing beats varargs**

|  |
| --- |
| Let's see the program where boxing beats variable argument: |

**class** Boxing3{

**static** **void** m(Integer i){System.out.println("Integer");}

**static** **void** m(Integer... i){System.out.println("Integer...");}

**public** **static** **void** main(String args[]){

**int** a=30;

   m(a);

 }

}

Output:Integer

### **Method overloading with Widening and Boxing**

|  |
| --- |
| Widening and Boxing can't be performed as given below: |

**class** Boxing4{

**static** **void** m(Long l){System.out.println("Long");}

**public** **static** **void** main(String args[]){

**int** a=30;

   m(a);

 }

}

Output:Compile Time Error

# Java Enum

**Enum in java** is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

## **Points to remember for Java Enum**

* enum improves type safety
* enum can be easily used in switch
* enum can be traversed
* enum can have fields, constructors and methods
* enum may implement many interfaces but cannot extend any class because it internally extends Enum class

### **Simple example of java enum**

**class** EnumExample1{

**public** **enum** Season { WINTER, SPRING, SUMMER, FALL }

**public** **static** **void** main(String[] args) {

**for** (Season s : Season.values())

System.out.println(s);

}}

Output:WINTER

SPRING

SUMMER

FALL

### **What is the purpose of values() method in enum?**

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

### **Internal code generated by the compiler for the above example of enum type**

The java compiler internally creates a static and final class that extends the Enum class as shown in the below example:

**public** **static** **final** **class** EnumExample1$Season **extends** Enum

{

**private** EnumExample1$Season(String s, **int** i)

    {

**super**(s, i);

    }

**public** **static** EnumExample1$Season[] values()

    {

**return** (EnumExample1$Season[])$VALUES.clone();

    }

**public** **static** EnumExample1$Season valueOf(String s)

    {

**return** (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);

    }

**public** **static** **final** EnumExample1$Season WINTER;

**public** **static** **final** EnumExample1$Season SPRING;

**public** **static** **final** EnumExample1$Season SUMMER;

**public** **static** **final** EnumExample1$Season FALL;

**private** **static** **final** EnumExample1$Season $VALUES[];

**static**

    {

        WINTER = **new** EnumExample1$Season("WINTER", 0);

        SPRING = **new** EnumExample1$Season("SPRING", 1);

        SUMMER = **new** EnumExample1$Season("SUMMER", 2);

        FALL = **new** EnumExample1$Season("FALL", 3);

        $VALUES = (**new** EnumExample1$Season[] {

            WINTER, SPRING, SUMMER, FALL

        });

    }

}

## **Defining Java enum**

The enum can be defined within or outside the class because it is similar to a class.

### **Java enum example: defined outside class**

**enum** Season { WINTER, SPRING, SUMMER, FALL }

**class** EnumExample2{

**public** **static** **void** main(String[] args) {

Season s=Season.WINTER;

System.out.println(s);

}}

Output:WINTER

### **Java enum example: defined inside class**

**class** EnumExample3{

**enum** Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here

**public** **static** **void** main(String[] args) {

Season s=Season.WINTER;//enum type is required to access WINTER

System.out.println(s);

}}

Output:WINTER

### **Initializing specific values to the enum constants**

The enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors and methods.

### **Example of specifying initial value to the enum constants**

**class** EnumExample4{

**enum** Season{

WINTER(5), SPRING(10), SUMMER(15), FALL(20);

**private** **int** value;

**private** Season(**int** value){

**this**.value=value;

}

}

**public** **static** **void** main(String args[]){

**for** (Season s : Season.values())

System.out.println(s+" "+s.value);

}}

Output:WINTER 5

SPRING 10

SUMMER 15

FALL 20

#### Constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

**enum** Season{

WINTER(10),SUMMER(20);

**private** **int** value;

Season(**int** value){

**this**.value=value;

}

}

### **Internal code generated by the compiler for the above example of enum type**

**final** **class** Season **extends** Enum

{

**public** **static** Season[] values()

    {

**return** (Season[])$VALUES.clone();

    }

**public** **static** Season valueOf(String s)

    {

**return** (Season)Enum.valueOf(Season, s);

    }

**private** Season(String s, **int** i, **int** j)

    {

**super**(s, i);

        value = j;

    }

**public** **static** **final** Season WINTER;

**public** **static** **final** Season SUMMER;

**private** **int** value;

**private** **static** **final** Season $VALUES[];

**static**

    {

        WINTER = **new** Season("WINTER", 0, 10);

        SUMMER = **new** Season("SUMMER", 1, 20);

        $VALUES = (**new** Season[] {

            WINTER, SUMMER

        });

    }

}

### **Can we create the instance of enum by new keyword?**

|  |
| --- |
| No, because it contains private constructors only. |

### **Can we have abstract method in enum?**

Yes, ofcourse! we can have abstract methods and can provide the implementation of these methods.

## **Java enum in switch statement**

We can apply enum on switch statement as in the given example:

### **Example of applying enum on switch statement**

**class** EnumExample5{

**enum** Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}

**public** **static** **void** main(String args[]){

Day day=Day.MONDAY;

**switch**(day){

**case** SUNDAY:

 System.out.println("sunday");

**break**;

**case** MONDAY:

 System.out.println("monday");

**break**;

**default**:

System.out.println("other day");

}

}}

Output:monday

# Java Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

## **Built-In Java Annotations**

There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.

## Built-In Java Annotations used in java code

* @Override
* @SuppressWarnings
* @Deprecated

## Built-In Java Annotations used in other annotations

* @Target
* @Retention
* @Inherited
* @Documented

## **Understanding Built-In Annotations in java**

Let's understand the built-in annotations first.

## **@Override**

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

**class** Animal{

**void** eatSomething(){System.out.println("eating something");}

}

**class** Dog **extends** Animal{

@Override

**void** eatsomething(){System.out.println("eating foods");}//should be eatSomething

}

**class** TestAnnotation1{

**public** **static** **void** main(String args[]){

Animal a=**new** Dog();

a.eatSomething();

}}

Output:Comple Time Error

## **@SuppressWarnings**

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

**import** java.util.\*;

**class** TestAnnotation2{

@SuppressWarnings("unchecked")

**public** **static** **void** main(String args[]){

ArrayList<String> list=**new** ArrayList<String>();

list.add("sonoo");

list.add("vimal");

list.add("ratan");

**for**(Object obj:list)

System.out.println(obj);

}}

Now no warning at compile time.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

## **@Deprecated**

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

**class** A{

**void** m(){System.out.println("hello m");}

@Deprecated

**void** n(){System.out.println("hello n");}

}

**class** TestAnnotation3{

**public** **static** **void** main(String args[]){

A a=**new** A();

a.n();

}}

## At Compile Time:

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

## At Runtime:

hello n

## **Custom Annotation**

To create and use custom java annotation, visit the next page.

# Java Custom Annotation

**Java Custom annotations** or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

**@interface** MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

## Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

## **Types of Annotation**

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

## **1) Marker Annotation**

An annotation that has no method, is called marker annotation. For example:

**@interface** MyAnnotation{}

The @Override and @Deprecated are marker annotations.

## **2) Single-Value Annotation**

An annotation that has one method, is called single-value annotation. For example:

**@interface** MyAnnotation{

**int** value();

}

We can provide the default value also. For example:

**@interface** MyAnnotation{

**int** value() **default** 0;

}

## How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

@MyAnnotation(value=10)

The value can be anything.

## **3) Mulit-Value Annotation**

An annotation that has more than one method, is called Multi-Value annotation. For example:

**@interface** MyAnnotation{

**int** value1();

String value2();

String value3();

}

}

We can provide the default value also. For example:

**@interface** MyAnnotation{

**int** value1() **default** 1;

String value2() **default** "";

String value3() **default** "xyz";

}

## How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

@MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")

## **Built-in Annotations used in custom annotations in java**

* @Target
* @Retention
* @Inherited
* @Documented

## **@Target**

**@Target** tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

|  |  |
| --- | --- |
| Element Types | Where the annotation can be applied |
| TYPE | class, interface or enumeration |
| FIELD | fields |
| METHOD | methods |
| CONSTRUCTOR | constructors |
| LOCAL\_VARIABLE | local variables |
| ANNOTATION\_TYPE | annotation type |
| PARAMETER | parameter |

## Example to specify annoation for a class

@Target(ElementType.TYPE)

**@interface** MyAnnotation{

**int** value1();

String value2();

}

## Example to specify annoation for a class, methods or fields

@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})

**@interface** MyAnnotation{

**int** value1();

String value2();

}

## **@Retention**

**@Retention** annotation is used to specify to what level annotation will be available.

|  |  |
| --- | --- |
| RetentionPolicy | Availability |
| RetentionPolicy.SOURCE | refers to the source code, discarded during compilation. It will not be available in the compiled class. |
| RetentionPolicy.CLASS | refers to the .class file, available to java compiler but not to JVM . It is included in the class file. |
| RetentionPolicy.RUNTIME | refers to the runtime, available to java compiler and JVM . |

## Example to specify the RetentionPolicy

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

**@interface** MyAnnotation{

**int** value1();

String value2();

}

## **Example of custom annotation: creating, applying and accessing annotation**

Let's see the simple example of creating, applying and accessing annotation.

*File: Test.java*

//Creating annotation

**import** java.lang.annotation.\*;

**import** java.lang.reflect.\*;

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**@interface** MyAnnotation{

**int** value();

}

//Applying annotation

**class** Hello{

@MyAnnotation(value=10)

**public** **void** sayHello(){System.out.println("hello annotation");}

}

//Accessing annotation

**class** TestCustomAnnotation1{

**public** **static** **void** main(String args[])**throws** Exception{

Hello h=**new** Hello();

Method m=h.getClass().getMethod("sayHello");

MyAnnotation manno=m.getAnnotation(MyAnnotation.**class**);

System.out.println("value is: "+manno.value());

}}

Output:value is: 10

## How built-in annotaions are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

## **@Inherited**

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

@Inherited

**@interface** ForEveryone { }//Now it will be available to subclass also

**@interface** ForEveryone { }

**class** Superclass{}

**class** Subclass **extends** Superclass{}

## **@Documented**

The @Documented Marks the annotation for inclusion in the documentation.

# Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

#### Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

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);

**3) 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>

## **Full Example of Generics in Java**

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

**import** java.util.\*;

**class** TestGenerics1{

**public** **static** **void** main(String args[]){

ArrayList<String> list=**new** ArrayList<String>();

list.add("rahul");

list.add("jai");

//list.add(32);//compile time error

String s=list.get(1);//type casting is not required

System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:element is: jai

rahul

jai

## **Example of Java Generics using Map**

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

**import** java.util.\*;

**class** TestGenerics2{

**public** **static** **void** main(String args[]){

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

map.put(1,"vijay");

map.put(4,"umesh");

map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator

Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();

**while**(itr.hasNext()){

Map.Entry e=itr.next();//no need to typecast

System.out.println(e.getKey()+" "+e.getValue());

}

}}

Output:1 vijay

2 ankit

4 umesh

## **Generic class**

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let’s see the simple example to create and use the generic class.

**Creating generic class:**

**class** MyGen<T>{

T obj;

**void** add(T obj){**this**.obj=obj;}

T get(){**return** obj;}

}

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

**Using generic class:**

Let’s see the code to use the generic class.

**class** TestGenerics3{

**public** **static** **void** main(String args[]){

MyGen<Integer> m=**new** MyGen<Integer>();

m.add(2);

//m.add("vivek");//Compile time error

System.out.println(m.get());

}}

Output:2

## **Type Parameters**

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

## **Generic Method**

Like generic class, we can create generic method that can accept any type of argument.

Let’s see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

**public** **class** TestGenerics4{

**public** **static** < E > **void** printArray(E[] elements) {

**for** ( E element : elements){

            System.out.println(element );

         }

         System.out.println();

    }

**public** **static** **void** main( String args[] ) {

        Integer[] intArray = { 10, 20, 30, 40, 50 };

        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

        System.out.println( "Printing Integer Array" );

        printArray( intArray  );

       System.out.println( "Printing Character Array" );

        printArray( charArray );

    }

}

Output:Printing Integer Array

10

20

30

40

50

Printing Character Array

J

A

V

A

T

P

O

I

N

T

## **Wildcard in Java Generics**

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

**import** java.util.\*;

**abstract** **class** Shape{

**abstract** **void** draw();

}

**class** Rectangle **extends** Shape{

**void** draw(){System.out.println("drawing rectangle");}

}

**class** Circle **extends** Shape{

**void** draw(){System.out.println("drawing circle");}

}

**class** GenericTest{

//creating a method that accepts only child class of Shape

**public** **static** **void** drawShapes(List<? **extends** Shape> lists){

**for**(Shape s:lists){

s.draw();//calling method of Shape class by child class instance

}

}

**public** **static** **void** main(String args[]){

List<Rectangle> list1=**new** ArrayList<Rectangle>();

list1.add(**new** Rectangle());

List<Circle> list2=**new** ArrayList<Circle>();

list2.add(**new** Circle());

list2.add(**new** Circle());

drawShapes(list1);

drawShapes(list2);

}}

drawing rectangle

drawing circle

drawing circle