Object Cloning in Java

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **rotected** Object clone() **throws** CloneNotSupportedException

### Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

### Advantage of Object cloning

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

* You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
* It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
* Clone() is the fastest way to copy array.

### Disadvantage of Object cloning

Following is a list of some disadvantages of clone() method:

* To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
* We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
* If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

### Example of clone() method (Object cloning)

Let's see the simple example of object cloning

**class** Student18 **implements** Cloneable{

**int** rollno;

String name;

Student18(**int** rollno,String name){

**this**.rollno=rollno;

**this**.name=name;

}

**public** Object clone()**throws** CloneNotSupportedException{

**return** **super**.clone();

}

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

**try**{

Student18 s1=**new** Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);

System.out.println(s2.rollno+" "+s2.name);

}**catch**(CloneNotSupportedException c){}

}

}

# Generics 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. They are as follows:

**) 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.

1. List list = **new** ArrayList();
2. list.add(10);
3. list.add("10");
4. With Generics, it is required to specify the type of object we need to store.
5. List<Integer> list = **new** ArrayList<Integer>();
6. list.add(10);
7. list.add("10");// compile-time error

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

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting
4. After Generics, we don't need to typecast the object.
5. List<String> list = **new** ArrayList<String>();
6. list.add("hello");
7. 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.

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

**Syntax** to use generic collection

1. ClassOrInterface<Type>

**Example** to use Generics in java

1. 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());

}

}

}

**import** java.util.\*;

**class** TestGenerics1{

1. **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());

}

}

}

## 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());

}

}}

## Generic class

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

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

### Creating a 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, and Employee). 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.

1. **class** TestGenerics3{
2. **public** **static** **void** main(String args[]){
3. MyGen<Integer> m=**new** MyGen<Integer>();
4. m.add(2);
5. //m.add("vivek");//Compile time error
6. System.out.println(m.get());
7. }}

## Type Parameters

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

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

## Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

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 );

    }

}