**Generics-2022**

**Arrays are covariant, Generics are not**

Object[] arr = new String[10];

Basically, an Object[] is a super type of String[], because Object is a super type of String. This is not true with generics. So, the following declaration is not valid, and won't compile:

List<Object> list = new ArrayList<String>(); // Will not compile.

Reason being, generics are invariant.

Enforcing Type Check:

However, arrays carry with them the runtime type information of the component type. At runtime, arrays use Array Store check to check whether you are inserting elements compatible with actual array type. So, the following code:

Object[] arr = new String[10];

arr[0] = new Integer(10);

will compile fine, but will fail at runtime, as a result of ArrayStoreCheck. With generics, this is not possible, as the compiler will try to prevent the runtime exception by providing compile time check, by avoiding creation of reference like this, as shown above.

**Why generic array creation is forbidden**

Consider the code as below:

public <T> T[] getArray(int size) {

T[] arr = new T[size]; // Suppose this was allowed for the time being.

return arr;

}

Since the type of T is not known at runtime, the array created is actually an Object[]. So the above method at runtime will look like:

public Object[] getArray(int size) {

Object[] arr = new Object[size];

return arr;

}

Now, suppose you call this method as:

Integer[] arr = getArray(10);

Here's the problem. You have just assigned an Object[] to a reference of Integer[]. The above code will compile fine, but will fail at runtime. That is why generic array creation is forbidden.

Why typecasting new Object[10] to E[] works?

Now your last doubt, why the below code works:

E[] elements = (E[]) new Object[10];

The above code have the same implications as explained above. If you notice, the compiler would be giving you an *Unchecked Cast Warning* there, as you are typecasting to an array of unknown component type. That means, the cast may fail at runtime. For e.g, if you have that code in the above method:

public <T> T[] getArray(int size) {

T[] arr = (T[])new Object[size];

return arr;

}

and you call invoke it like this:

String[] arr = getArray(10);

this will fail at runtime with a ClassCastException. So, no this way will not work always.

**Is there any workaround for E[]?**

Yes, you can create the array using [Array#newInstance()](http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Array.html" \l "newInstance%28java.lang.Class,%20int...%29) method:

public <E> E[] getArray(Class<E> clazz, int size) {

@SuppressWarnings("unchecked")

E[] arr = (E[]) Array.newInstance(clazz, size);

return arr;

}

Typecast is needed because that method returns an Object. But you can be sure that it's a safe cast. So, you can even use @SuppressWarnings on that variable.

Usage of Generics in Java

public class Person {

private String name;

public Person(String name) {

this.name = name;

}

public String getName() {

return name;

}

@Override

public String toString() {

return "Person{" +

"name='" + name + '\'' +

'}';

}

}

# **Type-1**

Let use create a generic type java class.

**public class GenericQ1<T> {**

private LinkedList<T> list = new LinkedList<>();  
  
 public void offer(T t) {  
 list.add(t);  
 }  
  
 public T poll() {  
 T t = list.removeFirst();  
 return t;  
 }  
  
 @Override  
 public String toString() {  
 return list.toString();  
 }  
  
 public static void main(String[] args) {  
 GenericQ1<Person> personQ = new GenericQ1<>();  
  
 for (int i = 0; i < 5; i++) {  
 Person p = new Person("Name-" + i);  
 personQ.offer(p);  
 }  
  
 System.*out*.println("All Persons : " + personQ);  
 Person person1 = personQ.poll();  
 System.*out*.println("Removed Person :::" + person1);  
 }  
}

In the above case the class has been defined with generic type <T>. The structure is given below.

**public class GenericQ1<T> {**

}

So the class is fully generic type.

# **Type – 2**

Let use create a general class and create generic method. The code is given below.

public class GenericQ2 {

private LinkedList<Object> list = new LinkedList<>();

public <T> void offer(T t) {

list.add(t);

}

public <T> T poll() {

T t = (T) list.removeFirst();

return t;

}

@Override

public String toString() {

return list.toString();

}

public static void main(String[] args) {

GenericQ2 personQ = new GenericQ2();

for (int i = 0; i < 5; i++) {

Person p = new Person("Name-" + i);

personQ.offer(p);

}

System.out.println("All Persons : " + personQ);

Person person1 = personQ.poll();

System.out.println("Removed Person :::" + person1);

}

}

Now you can mark the above highlighted area. Here the class is not of type generic. Let us see the basic difference.

public class GenericQ2<T>

public class GenericQ2

Always remember that , if the class is not generic type and if you are defining pure generic method, you have to define the method in the following manner.

**<accessModifier> <T> returnType(can be void) method() {**

**}**

Example is given below.

public **<T> void** offer(T t) { }

public **<T> T** poll() { }

**How to create a Generic Type Class**

public class Gen<T> {  
  
 public void doSomething(T t) {  
 if(t instanceof String) {  
 System.*out*.println("It is a String");  
 } else {  
 System.*out*.println("Not...");  
 }  
 }  
  
 public T getSomething(T t) {  
 return t;  
 }  
  
 public T get(T t) {  
 **T[] ts = (T[]) Array.*newInstance*(t.getClass(), 5);**  
 ts[0] = t;  
 T t1 = ts[0];  
 return t;  
 }  
}

Test

public static void main(String[] args) {  
 Gen<String> gen = new Gen<String>();  
 gen.doSomething("abcd");  
 String val1 = gen.get("PQRS");  
 System.*out*.println(val1);  
 String value2 = gen.getSomething("Hati");  
 System.*out*.println(value2);  
}