Generics

You should not mix-up arrays and generics together. They don't go well together. There are differences in how arrays and generic types enforce the type check. We say that arrays are reified, but generics are not. As a result of this, you see these differences working with arrays and generics.

Arrays are covariant, Generics are not:

What that means? You must be knowing by now that the following assignment is valid:

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:

Generics were introduced in Java to enforce stronger type check at compile time. As such, generic types don't have any type information at runtime due to [type erasure](http://docs.oracle.com/javase/tutorial/java/generics/erasure.html). So, a List<String> has a static type of List<String> but a dynamic type of List.

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.

So, what's the issue with Generic Array Creation?

Creation of array whose component type is either a *type parameter*, a *concrete parameterized type* or a *bounded wildcard parameterized type*, is ***type-unsafe***.

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.

What about creating an array of type List<String>[]?

The issue is the same. Due to type erasure, a List<String>[] is nothing but a List[]. So, had the creation of such arrays allowed, let's see what could happen:

List<String>[] strlistarr = new List<String>[10]; // Won't compile. but just consider it

Object[] objarr = strlistarr; // this will be fine

objarr[0] = new ArrayList<Integer>(); // This should fail but succeeds.

Now the ArrayStoreCheck in the above case will succeed at runtime although that should have thrown an ArrayStoreException. That's because both List<String>[] and List<Integer>[] are compiled to List[] at runtime.

So can we create array of unbounded wildcard parameterized types?

Yes. The reason being, a List<?> is a reifiable type. And that makes sense, as there is no type associated at all. So there is nothing to loose as a result of type erasure. So, it is perfectly type-safe to create an array of such type.

List<?>[] listArr = new List<?>[10];

listArr[0] = new ArrayList<String>(); // Fine.

listArr[1] = new ArrayList<Integer>(); // Fine

Both the above case is fine, because List<?> is super type of all the instantiation of the generic type List<E>. So, it won't issue an ArrayStoreException at runtime. The case is same with raw types array. As raw types are also reifiable types, you can create an array List[].

So, it goes like, you can only create an array of reifiable types, but not non-reifiable types. Note that, in all the above cases, declaration of array is fine, it's the creation of array with new operator, which gives issues. But, there is no point in declaring an array of those reference types, as they can't point to anything but null (*Ignoring the unbounded types*).

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.

import java.util.LinkedList;  
**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.

import java.util.LinkedList;

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() { }