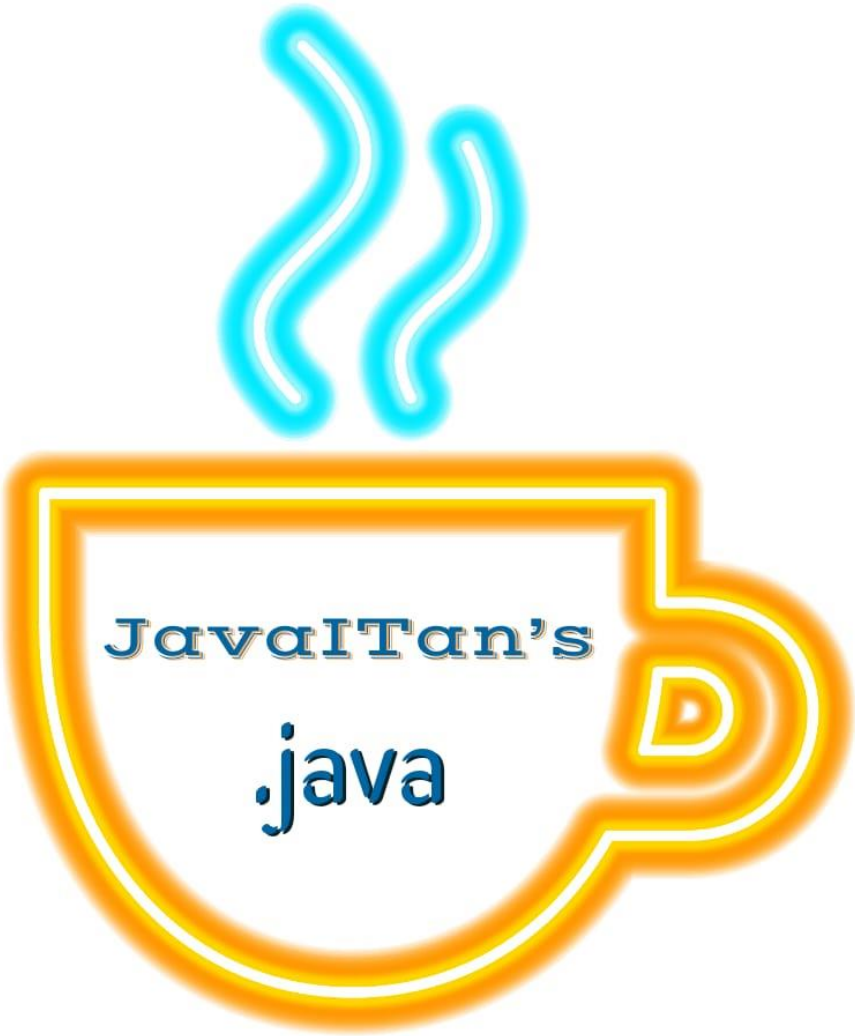
CORE JAVA INTERVIEW QUESTION'S

DAY - 7



---------*********---------

TEJAS PATIL

JAVA DEVELOPER

PURSUING PG – DAC FROM CDAC, PUNE

TABLE OF CONTENTS

1. What are Generics in Java? 3

2. How does Generic provide Type-safety? 3

3. What do you mean by type-erasure? 3

4. explain <? extends T> in java generics. 4

5. Explain <? super T> in java generics. 4

6. What is the rule of polymorphism and generics? 5

7. Why the following statement will not work? 5

8. Explain with reason which scenario is legal at compile-time and which is not. 6

9. What is the difference between List<?> mylist and List<Object> mylist? 7

10. Why can't we create Generic array? 7

1. What are Generics in Java?

Answer:- **Generics** are a facility of generic programming that were added to the Java programming language in 2004 within J2SE5.0. They allow "a type or method to operate on objects of various types while providing compile-time type safety.

2. How does Generic provide Type-safety?

Answer:- The following block of Java code illustrates a problem that exists when not using generics. First, it declares an ArrayList of type Object. Then, it adds a String to the ArrayList. Finally, it attempts to retrieve the added String and cast it to an Integer.

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // exception: ClassCastException
```

Although the code compiles without error, it throws a runtime exception (java.lang.ClassCastException) when executing the third line of code. This type of problem can be avoided by using generics and is the primary motivation for using generics.

Using generics, the above code fragment can be rewritten as follows:

```
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // (type error)  Compile time error
```

The type parameter String within the angle brackets declares the ArrayList to be constituted of String. With generics, it is no longer necessary to cast the third line to any particular type, because the result of v.get(0) is defined as String by the code generated by the compiler.

Compiling the third line of this fragment with J2SE 5.0 (or later) will yield a compile-time error because the compiler will detect that v.get(0) returns String instead of Integer.

3. What do you mean by type-erasure?

Answer:- - all the generic information applied in java application is removed by compiler while compiling the code. This is known as "type-erasure".

i.e. Type erasure is the technique using which the Java compiler translates generic / parameterized type to raw type in Java generics.

4. explain <? extends T> in java generics.

Answer:- - The wildcard declaration of List<? extends Number> foo3 means that any of these are legal assignments:

```
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends Number
```

Reading - Given the above possible assignments, what type of object are you guarenteed to read from List foo3:

- You can read a Number because any of the lists that could be assigned to foo3 contain a Number or a subclass of Number.
- You can't read an Integer because foo3 could be pointing at a List<Double>.
- You can't read a Double because foo3 could be pointing at a List<Integer>.

Writing - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for **all** the above possible ArrayList assignments:

- You can't add an Integer because foo3 could be pointing at a List<Double>.
- You can't add a Double because foo3 could be pointing at a List<Integer>.
- You can't add a Number because foo3 could be pointing at a List<Integer>.

You can't add any object to List<? extends T> because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only read from it and you'll get a T or subclass of T.

5. Explain <? super T> in java generics.

Answer:- - The wildcard declaration of List<? super Integer> foo3 means that any of these are legal assignments:

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a superclass of Integer
```

Reading - Given the above possible assignments, what type of object are you guaranteed to receive when you read from List foo3:

- You aren't guaranteed an Integer because foo3 could be pointing at a List<Number> or List<Object>.
- You aren't guaranteed a Number because foo3 could be pointing at a List<Object>.
- The only guarantee is that you will get an instance of an Object or subclass of Object (but you don't know what subclass).Writing - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for **all** the above possible ArrayList assignments:
- You can add an Integer because an Integer is allowed in any of above lists.
- You can add an instance of a subclass of Integer because an instance of a subclass of Integer is allowed in any of the above lists.
- You can't add a Double because foo3 could be pointing at a ArrayList<Integer>.
- You can't add a Number because foo3 could be pointing at a ArrayList<Integer>.
- You can't add an Object because foo3 could be pointing at a ArrayList<Integer>.

6. What is the rule of polymorphism and generics?

Answer:- - Polymorphism applies to the "base" type of the collection and not the “generic” type.

e.g.

Following code does not work

```
class Parent{}
class Child extends Parent{}

List<Parent>mylist=new ArrayList<Child>();
```

Because the rule is , generic type of reference and generic type of the object to which it refers must be identical. Polymorphism applies to "base" type, and the meaning of "base" is collection class itself.

Here List and ArrayList are type “base” and Parent and Child are type “generics”.

7. Why the following statement will not work?

Answer:- List<Animal> mylist=new ArrayList<Dog>();

The reason the compiler won't let u pass an ArrayList<Dog> into a method that takes an ArrayList<Animal>, is because within the method, that parameter is of type ArrayList<Animal>, and that means u could put any kind of Animal into it. There would be no way for compiler to stop u from putting a Cat into a List that was originally declared as <Dog>, but is now referenced from the <Animal> parameter.

Given the following two scenarios:

```
a)      public void addAnimal(Animal arr[])
{
  System.out.println(arr[0]);
}
```

```
Dog d[]={new Dog(),new Dog()};
addAnimal(d);
```

b) public void addAnimal(List<Animal> ref)

{

System.out.println(ref);

}

addAnimal(new ArrayList<Dog>());

8. Explain with reason which scenario is legal at compile-time and which is not.

Answer:- - scenario a is legal at compile-time whereas b is not.

The reason the compiler won't let u pass an ArrayList<Dog> into a method that takes an ArrayList<Animal>, is because within the method, that parameter is of type ArrayList<Animal>, and that means u could put any kind of Animal into it. There would be no way for compiler to stop u from putting a Cat into a List that was originally declared as <Dog>, but is now referenced from the <Animal> parameter.

Now the question arises , why compiler allows it in case of arrays and not in collection ?

The reason u can get away with compiling this for arrays is because there is a runtime exception (ArrayStoreException) that will prevent u from putting the wrong type of object into an array. If u send a Dog array into the method that takes an Animal array, and u add only Dogs (including Dogs subtypes) into the array now referenced by Animal, no problem. But if u do try to add Cat to the object that is actually a Dog array, u will get the exception.

But there is no equivalent exception for generics, because of type erasure !

In other words, at runtime the JVM knows the type of arrays but does not know the type of collection. All the generic type information is removed during compilation, so by the time it gets to the JVM, there is simply no way to recognize the disaster of putting a Cat into an ArrayList<Dog>. that is at runtime the JVM would have no way to stop u from adding a Cat to what was created as a Dog collection.

9. What is the difference between List<?> mylist and List<Object> mylist?

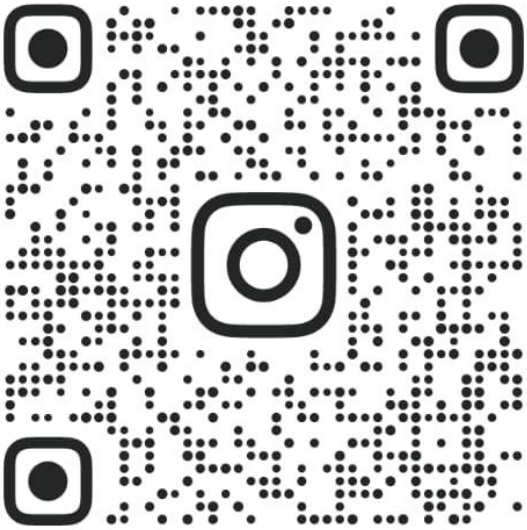
Answer:- - There is a huge difference. List<?>, which is the wildcard<?> without the keywords extends or super, simply means "any type". So that means any type of List can be assigned to the argument. That could be a List of <Dog>, <Integer> etc. whatever. And using the wildcard alone, without the keyword super(followed by a type),means that u cannot add anything to the list referred to as List<?>.

List<Object> is completely different from List<?>. List<Object> means that the method can take only a List<Object>. Not a List<Dog> or List<Cat>. It does, however, mean that u can add to the list, since the compiler has already made certain that u r passing only a valid List<Object> into the method.

10. Why can't we create Generic array?

Answer:- - It's because Java's arrays (unlike generics) contain, at runtime, information about its component type. So you must know the component type when you create the array. Since you don't know what T is at runtime, you can't create the array.

THANK YOU



Do Follow on Instagram for More Information……!