1. **How generics are typesafe?**

Ans:

before java 5

List mylist=new ArrayList();

mylist.add(new Integer(100));

mylist.add("hello");

mylist.add(new Double(3.4));

because "add()" method argument used to be java.lang.Object

while reading

String str=(String)mylist.get(1);

because "get()" method return type used to be java.lang.Object.

what is the risk involved in case of above code?

if "Integer" is there at 1 position then we will get ClassCastException.

java 5 onwards

Generics

List<String> mylist=new ArrayList<String>();

now compiler will see to it that mylist will be used with String only or else it will give error which is much better than ClassCastException.

e.g.

mylist.add("hello");

mylist.add("welcome");

mylist.add(100); // compilation error

at the time of reading

String str=mylist.get(1); // no typecasting required

if u say

Integer ob=mylist.get(0); // u will get compilation error

Hence we can say that Generics provide "type-safety".

1. **What is Type-Erasure?**

Ans:

Suppose we’ve written following code:

public class Generic1<T>

{

private T first;

void setVal(T first)

{

this.first=first;

}

T getVal()

{

return first;

}

when u compile above class, compiler will remove all the generic information because JVM can't understand Generics. This is known as "Type Erasure". So after compilation the above class will be as follows:

public class Generic1

{

private Object first;

void setVal(Object first)

{

this.first=first;

}

Object getVal()

{

return first;

}

1. **What is ? extends T in Java Generics?**

Ans:

In Java Generics, ? extends T represents a **wildcard with an upper bound**, meaning it can accept **T or any subclass of T**.

1. **Consider Animal is a parent of Dog class**

**What is the difference between List<? extends Animal> and List<? super Dog> as method parameters?**

Ans:

**List<? extends Animal> (Upper Bounded Wildcard)**

**Represents a list that can hold objects of Animal or any of its subclasses** (e.g., Dog, Cat, Labrador).

**We cannot add new elements** (except null) because the exact type is unknown at runtime.  
 **We can read elements as Animal**.

**We can pass:** List<Dog>, List<Cat>, or List<Animal>.

**We cannot add new elements** because list might be of type List<Dog> or List<Cat>, and inserting another type (new Cat()) could break type safety.

**We can read elements as Animal**, since every element in the list is at least an Animal.

**List<? super Dog> (Lower Bounded Wildcard)**

**Represents a list that can hold objects of Dog or any of its superclasses** (e.g., Animal, Object).  
**We can add Dog and its subclasses (Labrador)** because we know that Dog (or a parent) is a valid type.  
**We can only read elements as Object**, because the actual type is unknown.

**We can pass:** List<Dog>, List<Animal>, or List<Object>.

**We cannot assume elements are of type Dog when reading** because the list might be of type List<Animal> or List<Object>, so we can only safely retrieve Object.

1. **What is the difference between List<?> and List<Object>?**

Ans:

**List<?> (Unbounded Wildcard)**

**Represents a list of unknown type.**  
**Can hold any type of list (List<String>, List<Integer>, List<Object>, etc.).**  
**Cannot add new elements because the exact type is unknown.**  
**Can read elements, but only as Object.**

**We can pass:** List<Integer>, List<String>, List<Object>, etc.

**We cannot add elements** because list could be of any type (List<String>, List<Integer>), and adding a different type would break type safety.

**List<Object> (Explicit Object Type)**

**Represents a list that can hold Object and any of its subclasses.**  
**Can store elements of any type (String, Integer, Double, etc.).**  
**Can read elements as Object.**  
**Cannot accept List<Integer>, List<String>, etc., due to type mismatch.**

**We can add elements of any type** because Object is the parent of all classes.

**We cannot pass List<String> or List<Integer>** because generics are **invariant** (i.e., List<String> is not a subtype of List<Object>).

1. **Given:**

**List<Animal>mylist=new ArrayList<Dog>();**

**Why above statement causes compilation error?**

Ans:

If Java allowed this assignment:

List<Animal> mylist = new ArrayList<Dog>(); // Compilation error

It would mean that mylist can accept **any Animal**, but internally, it's actually a List<Dog>.  
Thus, the following scenario would break type safety:

mylist.add(new Cat()); // This would be allowed if the above assignment worked

* Cat is a subclass of Animal, so theoretically, adding a Cat to mylist should be valid.
* But since mylist actually refers to ArrayList<Dog>, this would corrupt the list because Cat is not a Dog.