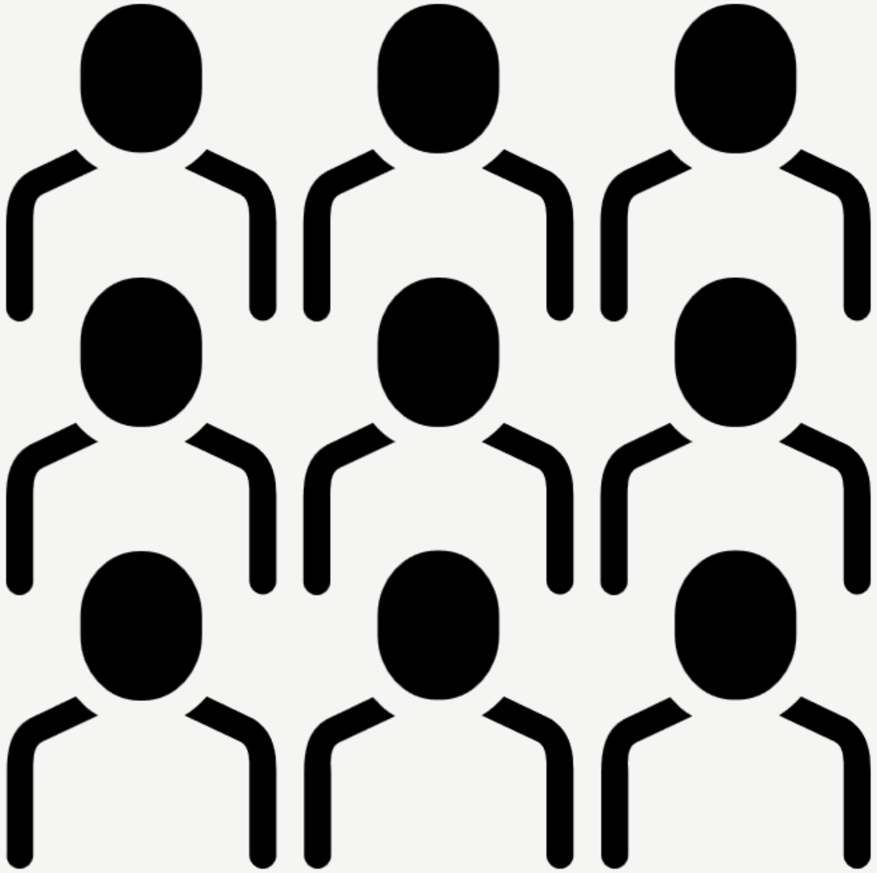
Generics in Java



Imagine you've submitted a request to a company to address an issue you're experiencing with their product. The company has responded by indicating that they will send someone to address your issue. You will anticipate that a person will arrive to solve your problem, and this individual could be of any race, gender, or name.

In a similar manner, when creating a generic business logic in Java that can be applied to any type of object passed, we can leverage Generics available in Java.

"Generics in Java is a feature that allows you to create reusable classes and methods that can work with different data types.

Generics let you write true polymorphic code that works with any data type."

Why we need Generics in Java ?

Let's consider a scenario where you want to create a generic container for a pair of values. To handle this scenario, you may resort to a non-generic class named Pair to store two values of different types. Here's a non-generic implementation:

```java
public class Pair {
    private Object first;
    private Object second;

    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() {
        return first;
    }

    public Object getSecond() {
        return second;
    }

    public void setFirst(Object first) {
        this.first = first;
    }

    public void setSecond(Object second) {
        this.second = second;
    }
}
```

Now, let's see how this class is used:

```java
Pair stringIntPair = new Pair("Hello", 42);
String myString = (String) stringIntPair.getFirst(); // Requires casting
int myInt = (Integer) stringIntPair.getSecond(); // Requires casting
```

While this code is functional, it requires explicit casting when retrieving values, and it doesn't provide compile-time type safety. For instance, the following code would compile but result in a ClassCastException at runtime:

```java
Pair stringIntPair = new Pair("Hello", 42);
stringIntPair.setFirst(16);
String myString = (String) stringIntPair.getFirst(); // ClassCastException
```

Generic class in Java

To enhance type safety and avoid such runtime issues, you can use generics to create a Pair class that enforces the type of its components. Here's how you could implement a generic Pair class:

```java
public class GenericPair<T, U> {
    private T first;
    private U second;

    public GenericPair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }

    public void setFirst(T first) {
        this.first = first;
    }

    public void setSecond(U second) {
        this.second = second;
    }
}
```

We can instantiate the generic class by substituting types for the type variables:

```java
GenericPair<String, Integer> stringIntPair = new GenericPair<String, Integer>("Hello", 42);
stringIntPair.setFirst("Another string");
stringIntPair.setSecond(123);

String myString = stringIntPair.getFirst();  // No casting needed
int myInt = stringIntPair.getSecond();         // No casting needed
```

Using generics in this way provides compile-time type checking and avoids the need for explicit casting, leading to cleaner and safer code.

Generics in Java empower developers to designate a type parameter for a type, whether it be a class or an interface. This type, known as a generic type, can be a generic class or a generic interface. When declaring a variable of the generic type and initializing an object, you have the flexibility to specify the type parameter value. While you are already familiar with specifying parameters for methods, the concept of generics extends this capability to types, such as classes or interfaces.

Generic class in Java

eazy bytes

Type parameters cannot be instantiated with primitive types. For example, GenericPair<String, int> is not valid in Java.

The type parameters T and U are specified inside angle brackets after the name of the class where as in the definitions of class members, they are used as types for instance variables, method parameters, and return values.

When you construct an object of a generic class, you can omit the type parameters from the constructor by using Diamond operator. For example, both the following declarations are considered same,

```
GenericPair<String, Integer> stringIntPair = new GenericPair<>("Hello", 42);

GenericPair<String, Integer> stringIntPair = new GenericPair<String, Integer>("Hello", 42);
```

Previously we used T, U inside Generics. It is not mandatory to use T, U alone. We can use any character. Below are the few character standards that's followed by Developers,

T – Type
E – Element
K – Key
N – Number
V – Value

Generics methods in Java

Here is an example of a Generic method in Java. In the code below, we have a method named printArray that takes an array of any data type T as a parameter. The method then loops through the array and prints out each element using the System.out.println() method.

When you declare a generic method, the type parameter <T> is placed after the modifiers (such as public and static) and before the return type.

When calling a generic method, you do not need to specify the type parameter. It is inferred from the method parameter and return types.

```java
public static <T> void printArray(T[] array) {
    for(T element: array){
        System.out.println(element);
    }
}
```

Here is an example of how to use the printArray method with different data types. In the code above, we create two arrays, intArray and stringArray, of type Integer and String, respectively. We then call the printArray method twice, passing in each array as a parameter. The method is able to print out the contents of each array, regardless of their data type, thanks to the use of Generics.

```java
Integer[] intArray = { 1, 2, 3, 4, 5 };
String[] stringArray = { "Hello", "World" };

printArray(intArray);
printArray(stringArray);
```

We can use Collections with out Generics !!!!

eazy bytes

We can create Collection objects with out generics as well. Thus a Collection object can store any kind of objects. Here is an example of an ArrayList without Generics:

```java
ArrayList list = new ArrayList();
list.add("Hello");
list.add(123);
list.add(true);

String greeting = (String) list.get(0);
int number = (int) list.get(1);
boolean flag = (boolean) list.get(2);
```

In the example, we have created an ArrayList named list without using Generics. We add three elements to the list, a String, an int, and a boolean.

When retrieving elements from the list using the get method, we must cast them to their appropriate data type. This can be error-prone, as we may accidentally cast an element to the wrong data type and cause a runtime exception.

With the use of Generics, we can ensure type safety and avoid runtime errors. Here is an example of the same code using Generics:

```java
ArrayList<String> list = new ArrayList<String>();
list.add("Hello");
// list.add(123); // Compilation error - can only add Strings to the list
// list.add(true); // Compilation error - can only add Strings to the list

String greeting = list.get(0); // No need for casting, as the list only contains Strings
```

In the code above, we have created an ArrayList with a type parameter of String, using Generics. We can only add Strings to this list, and we don't need to cast elements when retrieving them using the get method. This ensures type safety and prevents runtime errors.

Covariance

eazy bytes

At times, it is necessary to impose specific constraints on the type parameters of a generic class or method. This is achieved by specifying a type bound, which mandates that the type must either extend particular classes or implement specific interfaces.

Suppose you are tasked with implementing a method to handle an array of objects belonging to subclasses of the "Employee" class. In this scenario, you can straightforwardly declare the parameter to be of type Employee[].

```
public static void process(Employee[] employees) {

        // some logic

}
```

If "Developer" or "Manager" is a subclass of "Employee," you have the flexibility to pass a "Developer[]" or "Manager[]" array to a method because "Developer[]" or "Manager[]" are considered a subtype of "Employee[]." This characteristic, where arrays vary in the same way as the element types, is referred to as **covariance**.

What is Covariance ?

Given that "String" is a subclass of "Object" in Java, it follows the language rules that allow assigning a child object reference to a parent object. This is called Covariance. For example:

```
String s = "Generics";
Object o = s;
```

Covariance

Similarly, applying covariance rules, the following assignment is also valid:

```
String[] sArray = { ”Generics”, ”Collections” };
Object[] objArray = sArray; // This is valid in Java due to covariance
```

Now let’s try similar covariance rules with the collections,

```
List<String> s = new ArrayList<>();
List<Object> o = new ArrayList<>();
o = s; // Compilation fails as collections doesn’t support covariance
```

If we try to implement a generic method to handle an collection of objects belonging to subclasses of the "Employee" class like shown below, it will not work.

```
public static void process(List[] employees) {

        // some logic

}
```

The type ArrayList<Developer>/ArrayList<Manager> is not a subtype of ArrayList<Employee> as collections doesn’t support covariance.

Why covariance not supported by Collections ?

The decision to make arrays covariant in Java was a deliberate design choice, providing a significant degree of polymorphic behavior. This design allowed the storage of various business objects inside an "Object[]" array, enabling the creation of generic code. However, this choice introduced the potential for bugs that could only be identified at runtime. Consider the following scenario:

```
Number[] numArray = { 1, 2, 3 };
Object[] objArray = numArray;
objArray[0] = "Hello"; // Compilation will be fine but at runtime ArrayStoreException will be throwed
```

To avoid the same pitfalls that rendered arrays risky, a similar approach is not permitted in collections. However, it's crucial to note that there are valid technical use cases for such behavior, and Java engineers were aware of this. If you intend to achieve similar functionality in Collections, it is necessary to declare our Collections as covariant.

Subtype or Upper Bound Wildcards

Subtype or Upper Bound Wildcards : <? extends T> means a type that is either T or a child of T, where T is the type used as the generics of a collection.

```
public static void printEmployeeNames(ArrayList<? extends Employee> employees) {
    for (Employee e: employees){
        System.out.println(e.getName());
    }
}
```

The wildcard type ? extends Employee indicates some unknown subtype of Employee. You can call this method with an ArrayList<Employee> or an array list of a subtype, such as ArrayList<Developer>.

When we are using Subtype or Upper Bound Wildcards, we can't add a new element/object inside the collection. For example, inside the above method, if we try to add an object of Employee or it's subtypes, we are going to get a compilation error.

The reason is very simple, the ? inside ? extends Employee could refer to any subclass, perhaps Developer or Manager. Since we can't add a Developer to an ArrayList< Manager>, adding a new element is not allowed while using Subtype or Upper Bound Wildcards.

Supertype or Lower Bound Wildcards

Contravariance is the opposite of covariance, and it refers to the ability of a type to accept more generic (broader) types. The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword. In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>

You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> — anything that can hold Integer values.

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>. The term List<Integer> is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer. The following code adds the numbers 1 through 10 to the end of a list:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

Unbounded Wildcards

eazy bytes

The unbounded wildcard type in Java is denoted by the "?" character, as seen in expressions like List<?>. This is referred to as a list of unknown type. There are two scenarios where using an unbounded wildcard is useful:

1) When writing a method that relies on Object class functionality: If you are creating a method that can use features provided by the Object class, employing an unbounded wildcard is beneficial.
2) When using methods in a generic class independent of the type parameter: If the code is using methods in a generic class that don't depend on the type parameter, such as List.size or List.clear, employing an unbounded wildcard like List<?> is appropriate.

Let's consider the following example method, printList:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>.

Unbounded Wildcards

To write a generic printList method, we must use unbounded wildcards like List<?>. Below is the updated example with unbounded wildcards,

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Because for any concrete type A, List<A> is a subtype of List<?>, you can use printList to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

Advantages of Java Generics

eazy bytes

Type-safety

Generics allow us to store only a single type of object and prevent the storage of other types.In contrast, without Generics, we can store any type of object, regardless of its data type.

No Type casting required & Code Reusability

Using Generics we can eliminate the need for explicit type casting while working with collections etc.

With the help of generics, we can write code that will work with different types of objects.

Compile-Time Check

Using Generics in Java, we can check for errors at compile time, which helps in preventing potential problems that may occur at runtime. As a best practice, it is recommended to handle errors and resolve them during the compilation phase rather than at runtime, in order to minimize the risk of unexpected errors or behavior in the program.