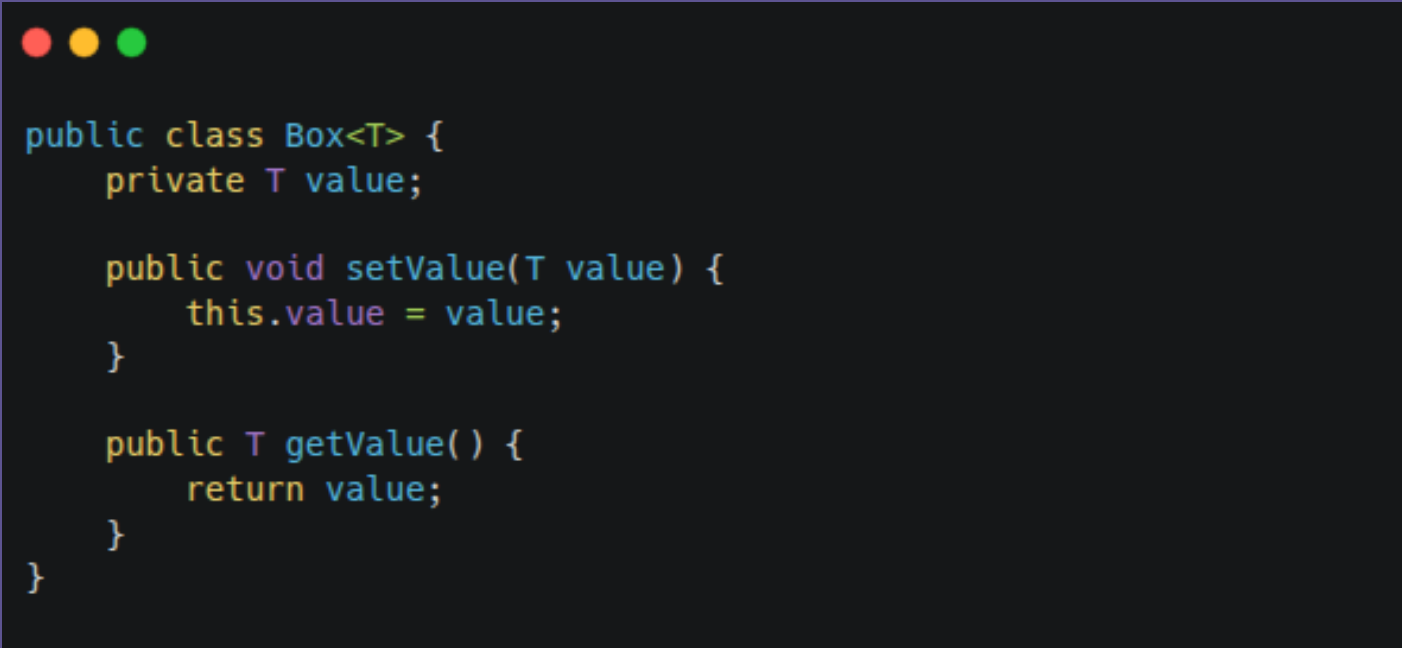


What is **Generics** `<T>` in Java?



Introduction

- Generics in Java allow you to write a class, interface, or method that works with any type of data, while providing compile-time type safety.
- Instead of specifying the exact type of an object, you can use a placeholder (a generic type) that will be replaced by the actual type when the code is compiled.



```
public class Box<T> {  
    private T value;  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
  
    public T getValue() {  
        return value;  
    }  
}
```

Here, the class **Box** can store any type of object, and the type **T** is determined when an object of the class is created (e.g., **Box<Integer>** or **Box<String>**).

Why do we need Generics?

- **Type Safety:** Generics enforce compile-time type checking, ensuring that the correct type is used and reducing runtime errors.

```
List<Integer> list = new ArrayList<>();  
list.add("Hello"); // Compile-time error, because list expects Integer, not String.
```

- **Code Reusability:** With generics, you can write more generalized code. Instead of writing multiple versions of the same method or class for different types, you can use a single generic method or class.

```
public static void main(String[] args) {  
    // Create a Box to hold Integer values  
    Box<Integer> integerBox = new Box<>();  
    integerBox.setValue(10); // Store an Integer  
    System.out.println("Integer value: " + integerBox.getValue());  
  
    // Create a Box to hold String values  
    Box<String> stringBox = new Box<>();  
    stringBox.setValue("Hello Generics!"); // Store a String  
    System.out.println("String value: " + stringBox.getValue());  
}
```

- **Eliminates Explicit Type Casting:** When using generics, you don't need to cast objects to their expected types. The compiler handles it for you, making the code cleaner and less error-prone.

```
List<String> list = new ArrayList<>();  
list.add("Hello");  
String value = list.get(0); // No cast needed!
```

- **Enhanced Maintainability:** Since the type of data is known at compile-time, it makes code easier to understand and maintain. It also makes refactoring safer.

When to Use Generics?

Data Structures:

When creating custom data structures (e.g., linked lists, stacks, queues) that should work with any type of object.

Utility Classes:

When writing utility methods that operate on collections or other types, generics help provide type safety and reduce code duplication.

APIs:

When you design reusable APIs or libraries that must work with a variety of types without knowing them in advance.

Wildcards in Generics

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

<?> means any type can be used, and no restrictions are applied. but it doesn't allow you to modify the elements of the collection. You can read the elements, but you can't insert new elements because you don't know the exact type.

```
public void printList(List<?> list) {  
    for (Object obj : list) {  
        System.out.println(obj);  
    }  
}  
//This method will accept a list of any type (List<Integer>, List<String>, etc.).|
```

- **Upper Bounded Wildcard (<? extends T>)**

The upper-bounded wildcard restricts the type to be either the specified type or a subtype of it.

```
public void printNumbers(List<? extends Number> list) {  
    for (Number num : list) { //You can safely read as Number or any subclass of Number  
        System.out.println(num);  
    }  
}
```

- **Lower Bounded Wildcard (<? super T>)**

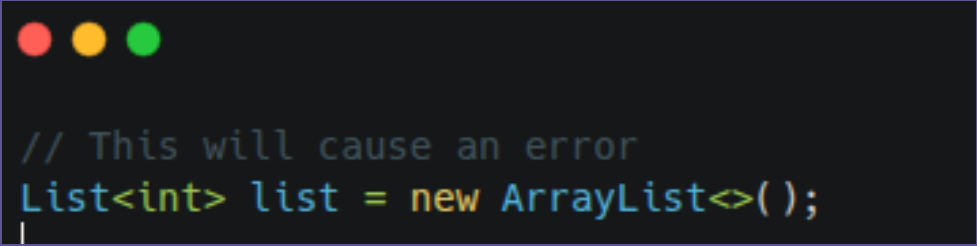
The lower-bounded wildcard allows you to pass a type that is either the specified type or a supertype of it.

```
public void addNumbers(List<? super Integer> list) {  
    list.add(10); // You can safely add Integer, because the list is of type Integer or a  
                 // supertype  
}
```

When to Not Use Generics?

- **When You Need to Work with Primitive Types**

Generics cannot work directly with primitive types like `int`, `char`, `double`, etc. You need to use their wrapper classes (e.g., `Integer`, `Character`, `Double`) instead.

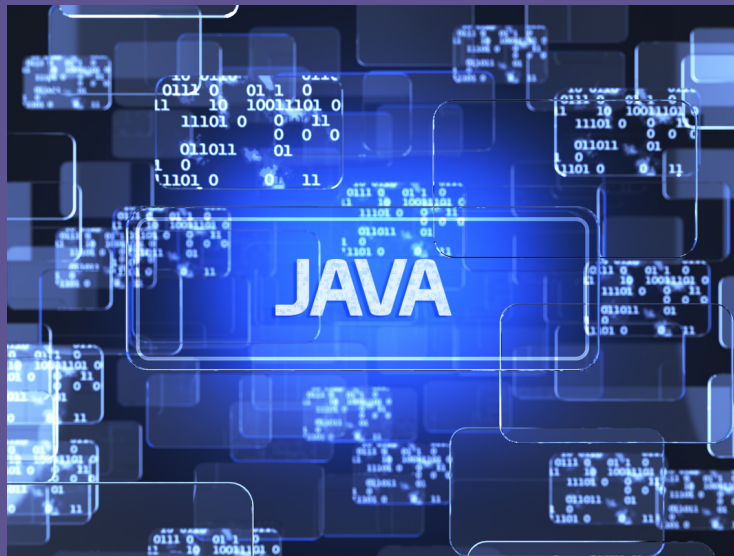


```
// This will cause an error
List<int> list = new ArrayList<>();
|
```

- **When You Don't Need Type Safety**
- **When the Code is Not Intended for Reusability**

Thank you for reading

Please share this with your friends



Manjul Tamang

Software Developer

manjultamang.com.np