Generics in Java

AP Fall-1404 Dr. Mojtaba Vahidi Asl
by Danial Yaghooti

Generics

Generics represent parameterized types, which means we can write classes, interfaces, and methods that work with different data types without rewriting the same code for each type. Instead of fixing the data type in advance, we use a type parameter (such as <T>) that will be specified when the class or method is used.

By using generics, we can create reusable and flexible code that still remains type-safe. Type safety is important because it helps catch errors at compile time, reducing the chances of runtime errors like incorrect casting.

Types of Java Generics

1- Generic Class

- A generic class is like a regular class but uses type parameters (like <T>).
- It can accept one or more types.
- Makes the class reusable for different data types.
- To create objects of a generic class, we use the following syntax:

```
ClassName<Type> obj = new ClassName<Type>();
```

Note: In Parameter type, we can not use primitives like "int", "char" or "double". Use wrapper classes like Integer, Character, etc

Generic Class

Here is how we define a generic class:

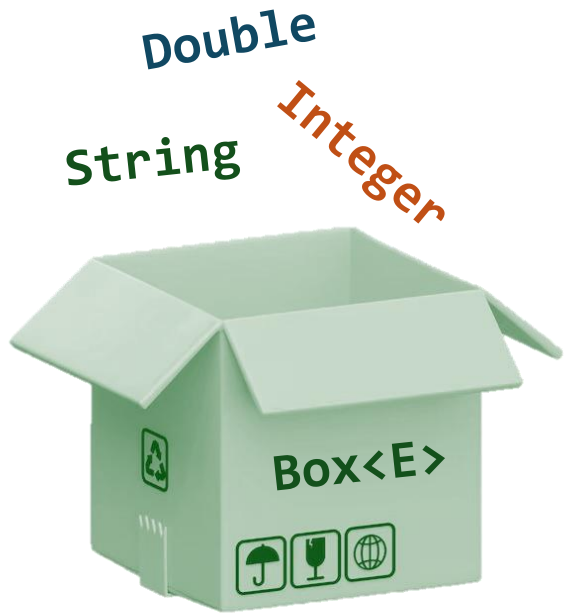
```
class Box<E>{
    E element;

    public Box(E element){
        this.element = element;
    }

    public E getElement(){ return this.element;}
}
```

Now you can store any reference value as:

```
Box<referenceType> box= new Box<>(Element);
```



Note:

- E can be replaced by anything: A, B, Element…
- You can add more generic types in your class: <T,V,…>

Types of Java Generics

2- Generic Methods

- A generic method is a method that can work with different data types using a type parameter.
- let's you write one method that works for all types, instead of repeating the same logic.
- The mains structure of a generic method is:

```
<type-parameters> returnType methodName(parameterList)
```

Generic Method

Let's see an example:

```
public class Test {

    static <T> void WhoIs(T element) {
        System.out.println(element.getClass().getName() + " = " + element);
    }

    public static void main(String[] args) {

        WhoIs(17); //java.lang.Integer = 17

        WhoIs("Java"); //java.lang.String = Java
    }
}
```

Limitations of Generics

1-

Generics Work Only With References Type

When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like int, char.

```
Box<int> b= new Box<>(7);
Box<Integer> b= new Box<>(7);
```

2-

Generic Types Differ Based on their Type Arguments

During compilation, generic type parameters are replaced with their bound types (or Object if unbounded), and casts are inserted as needed. This process is called type erasure.

Example:

```
//instance of Integer
Box<Integer> b1= new Box<>(7);
//instance of String
Box<String> s1= new Box<>("Java");
s1 = b1; // -> Compile Time Error
```

Let’s talk about Autoboxing

What is Autoboxing?

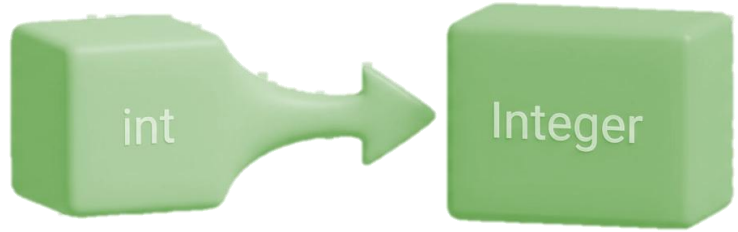
Autoboxing is the automatic conversion that the Java compiler makes between the primitive types (e.g., int, double) and their corresponding object wrapper classes (e.g., Integer, Double).

- Mechanism: It is “Syntactic Sugar.” The compiler injects valueOf() methods behind the scenes.
- Unboxing: The reverse process (Wrapper to Primitive) is called Unboxing.
- Purpose: Allows primitives to be used in Collections (like ArrayList) and simplifies syntax.

Example:

```
// What you write:
List<Integer> numbers = new ArrayList<>();
numbers.add(10); // Autoboxing happens here!!

// What the compiler generates:
numbers.add(Integer.valueOf(10));
```



Traps in Autoboxing

1- The Danger of Unboxing Null

Since Wrapper classes are objects, they can be null. Primitives cannot. If you try to unbox a null wrapper into a primitive, the JVM throws a NullPointerException.

Why it happens:

The compiler translates the unboxing to obj.intValue(). If obj is null, you are effectively calling a method on null.

Example:

```
Integer count = null;

// CRASH! Throws NullPointerException
// Because it tries to do: count.intValue()
int value = count;
```

⚠ Always validate that the wrapper object is not null before assigning it to a primitive.

Traps in Autoboxing

2- Reference vs. Value Comparison

When using the == operator on Wrapper objects, Java compares memory references, not values.

However, Java caches Integer objects between -128 and 127.

Inside Cache: == might return true because both variables point to the same cached object.

Outside Cache: == returns false because new objects are created in the heap.

Example:

```
Integer a = 100;
Integer b = 100;
System.out.println(a == b); // TRUE (Cached instances)

Integer x = 200;
Integer y = 200;
System.out.println(x == y); // FALSE (Different objects in Heap)
```

⚠ NEVER use `==` to compare Wrapper objects. Always use `.equals()`.

Traps in Autoboxing

3- The Hidden Cost of Convenience

Autoboxing is not free. It involves object creation, memory allocation, and eventual Garbage Collection.

The Issue:

Using Wrapper classes in heavy computation loops causes “object churn.” The JVM creates millions of short-lived objects, putting unnecessary pressure on the Garbage Collector and CPU.

Example:

```
Long sum = 0L;
// Wrapper Class
for (long i = 0; i < Integer.MAX_VALUE; i++) {
// Creates a new Long object in every iteration!
sum += i;
}
```

⚠ Use primitives (int, long, double) for calculations and loops.
Only use Wrapper classes when necessary.

Restricting Generic Types

You can restrict a generic class or method to accept only specific types using bounded type parameters.

This ensures type safety and prevents invalid type usage.

```
//Syntax
<T extends SuperClass>
```

Means must be `SuperClass` or subclass of it.

You can also use interfaces:

```
//Syntax
<T extends Comparable<T>>
```

Restricting Generic Types

Example:

```
public class Test {

    static <T extends Number> void showDouble(T n) {
        System.out.println(n.doubleValue());
    }

    public static void main(String[] args) {

        showDouble(51);           // Ok(Integer)
        showDouble(17.8);        //Ok(Double)
        showDouble("Java");    // Compile Time Error ✖
    }
}
```

Pair Class

A generic Pair class allows storing two related value of possibly different types, similar to `std::pair` in C++.

Here is Java's implementations:

```
public class Pair<K,V>{
    K first;
    V second;

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

    @Override
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
//Usage
Pair<String, Integer> user= new Pair<>("Negar", 21);
```

Resources

w3schools.com

Thank You for Your Time ([_])