TUTORIALS — Generics

version **#1.0.0**

YET ANOTHER KIND OF ASSISTANTS 2026 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.forge.epita.fr

1 Introduction

Generics were introduced with *Java 5*. They allow us to create type-safe classes, interfaces and methods working with multiple kinds of data.

Java always offered the possibility to create classes that operate on various types of objects by using the root of the class hierarchy: the `Object` class. Yet, this technique did not ensure type safety. One could do the following using this idea:

```java
public class MyObject {
    private Object obj;

    public void setObj(Object obj) {
        this.obj = obj;
    }

    public Object getObj() {
        return obj;
    }
}
```

The idea is that this `MyObject` class will accept any kind of type. You could insert a `String` when an `Integer` was expected somewhere else in your code for example. However, by using generics, you guarantee which type the generic class will operate on.

```java
public class MyGenericObject<T> {
    private T obj;

    public void setObj(T obj) {
        this.obj = obj;
    }

    public T getObj() {
        return obj;
    }
}

public class Main {
    public static void main(String[] args) {
        // <String> not needed because it is infered from the type on the left
        MyGenericObject<String> str = new MyGenericObject<>();
        // ...
    }
}
```

2 Overview

As you have probably noticed, you have already been using generics with Lists. Indeed, when you write List<Double>, you are using a special case of List<T> with T = Double. Generics are very common and can have multiple type parameters like a map taking a key and a value Map<K,V>.

2.1 Example

The following example shows how to use generics in *Java*.

```java
// The class takes a generic parameter ELEMENT_TYPE
public class Stack<ELEMENT_TYPE> {
    private List<ELEMENT_TYPE> list = new ArrayList<>();

    // The elements we can push on the stack can only be of type ELEMENT_TYPE
    void push(ELEMENT_TYPE element) {
        list.add(0, element);
    }

    ELEMENT_TYPE pop() {
        ELEMENT_TYPE result = list.get(0);
        list.remove(0);
        return result;
    }

    // Does not compile because the method is static
    static ELEMENT_TYPE myStaticMethodNotWorking(ELEMENT_TYPE t) {
        return t;
    }

    // Generic method: U is declared between chevrons at the beginning of the prototype
    public static <U> U myStaticMethod(U u) {
        return u;
    }
}
```

You can create generic classes and interfaces but also generic methods, that introduce their own type parameters, limited to the method's scope, like the last method in the example above. The type parameters are specified between angle brackets before the return type of the method.

Be careful!

Static methods can be called even if no object of the class has been instantiated. Because of this, you cannot rely on generic types declared with the class. You have to redeclare them on the method prototype like in the example above.

2.2 Bounded generics

You may want to write a generic class that uses the methods of an interface or an abstract class.

Generics can be written with bounds, meaning that you can restrict the use of the generic class to a limited set of classes that satisfy an inheritance relation.

For instance, you can limit the use of a generic list to *Number* types only by writing this:

```java
public class MyNumberList<T extends Number & Serializable> {
    // etc...
}
```

When MyNumberList is used, *T* must be a type inheriting from *Number* and *Serializable*, otherwise the program will not compile.

```java
// Does not compile
MyNumberList<String> myList = new MyNumberList<>();
// Compiles because Integer inherits from both classes
MyNumberList<Integer> myList = new MyNumberList<>();
// Does not compile because int is a primitive type
MyNumberList<int> myList = new MyNumberList<>();
```

2.3 Wildcards

To avoid the use of a concrete type when defining a generic, wildcards *?* can provide a shortcut to define a generic type upper bound or lower bound.

For instance, *List<? extends Number>* is a list that has values of a type inheriting of *Number* like *Integer* or *Double*. On the contrary, *List<? super Number>* defines a list that contains values of a type that is *Number* or one of its parent (*Object*).

```java
List<? super Number> list = List.of(1,2);

// Does not compile because Integer is not the lowest bound of Number
Integer value = list.get(0);

// Compiles because Object is the lowest bound of Integer
Object value = list.get(0);

List<? extends Number> otherList = List.of(1.0d, 2.0d);

// Does not compile because the call to get() will return a value of type
// "? extends Number", which *can* be a Float, but we don't know the actual
// type behind it.
Double otherValue = otherList.get(0);

// Compiles because a Number extends Number
Number myNumber = otherList.get(0);

// Compiles because doubleValue is a method of Number
// Beware that it may cause loss of precision
Double otherValue = myNumber.doubleValue();
```

Going further...

Feel free to check the documentation of generics.

Being a hero means fighting back even when it seems impossible.