Java Generics

André Restivo

Index

Introduction

Java without Generics

Without generics, we would write code like this:

```
List wordList = new ArrayList();
wordList.add("ABCD");
String word = wordList.get(0);
```

Only to find the compiler complaining about that last line.

This happens because, unqualified Lists, store **Objects**. So, `wordList.get(0)` returns an **Object**.

This means we need to cast it to a String first:

```
String word = (String)wordList.get(0);
```

Why we need Generics

Without generics, the following code will throw a **ClassCastException**:

```
List wordList = new ArrayList();
wordList.add(123);
String word = (String) wordList.get(0);
```

It would be much nicer if we could detect this kind of bugs in **compile-time**.

What generics provide is **Type-safety**:

```
List<String> wordList = new ArrayList<>();
wordList.add(123); // add (String) in List cannot be applied to (int)
String word = wordList.get(0);
```

Now, besides **not** needing the explicit **cast**, trying to add an integer to the list, results in a **compile-time** error.

Type Variables

Type Variables

- A **type variable** is an unqualified identifier. Unlike String, which is qualified, an unqualified identifier doesn't tell explicitly which class we are talking about.
- A **class**, **interface**, **method** or **constructor** is generic if it declares one or more type variables.

Type variables are declared using the **diamond operator**:

```
<T>
```

Type Variable Names

There are some naming conventions usually associated with the use of generics.

Normally, type variable names are single, uppercase letters to make it easily distinguishable from java variables:

- E – **Element** (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – **Key** (Used in Map)
- N – **Number**
- T – **Type**
- V – **Value** (Used in Map)
- S,U,V etc. – 2nd, 3rd, 4th types

Generic Methods

Methods without Generics

Consider that we want to create a function that adds an **Array** of **Strings** to a **List** of **Strings**:

```
public void stringArrayToList(String[] a, List<String> l) {
  for (String s : a) l.add(s);
}
```

Now imagine we want to extend this method to **all types** of objects.

Methods without Generics

Without generics, we would do it like this:

```
public void arrayToList(Object[] a, List l) {
  for (Object o : a) l.add(o);
}
```

Two problems:

- We can't guarantee in **compile-time** the type of the **Array**.
- There are no guarantees about the types inside the **List** we are getting back.

So if we want to retrieve the values from the **List**, we have to cast them to **String** and risk a **ClassCastException**.

```
String[] a = {"ABC", "DEF"}; List l = new ArrayList();

arrayToList(a, l);

for (Object s : l)
  System.out.println((String)s);
```

Methods without Generics

And even if we swear to always use parameterized **Lists**:

```
String[] a = {"ABC", "DEF"}; List<String> l = new ArrayList<>();

arrayToList(a, l);

for (String s : l)
  System.out.println(s);
```

Something like this would still be possible:

```
String[] a = {"ABC", "DEF"}; List<Integer> l = new ArrayList<>();

arrayToList(a, l);

for (Integer i : l)
  System.out.println(i);
```

And we would get a **ClassCastException** again!

Generic Methods

Using generics, we can solve this problem by creating a parameterized function:

```
public <T> void arrayToList(T[] a, List<T> l) {
    for (T o : a) l.add(o);
}
```

Notice how we declared that the function is **parameterized** by a **type variable** called **T**. Now we can call this method with any kind of **List** whose element type is a **supertype** of the element type of the **Array**.

```
String[] a = {"ABC", "DEF"}; List<String> l = new ArrayList<>();

arrayToList(a, l);

for (String s : l)
    System.out.println(s);
```

And if we try to use **incompatible** types for the **Array** and **List**, we will get a **compile-time** error.

Generic Classes

Generic Classes

We can define our **own classes** with generics type.

A generic type is a **class** or **interface** that is **parameterized** over types:

```
public class Box<T> {
  T something;

  void put(T something) {
      this.something = something;
  }

  T get() {
      return this.something;
  }
}
```

Using it:

```
Box<String> box = new Box<>();
box.put("ABC");               // Only works with strings
String something = box.get(); // Only returns strings
```

Extending Generic Types

We can also create classes that extend generic types:

```
class NamedBox<T> extends Box<T> {
    String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

We can even create non-parameterized types based on parameterized types:

```
class StringBox extends Box<String> { }
```

Variance

Or a closer look at subtyping

Variance

Variance refers to how **subtyping** between more **complex types** relates to subtyping between their **components**:

- **Covariance**: if it preserves the ordering of types (accepts subtypes).
- **Contravariance**: if it reverses the ordering of types (accepts supertypes).
- **Invariance**: if it only accepts the specific type.

Basic Types

Basic Java types are **covariant**. So this is possible:

```
abstract class Animal { }
class Zebra extends Animal { }

Zebra zebra = new Zebra();
Animal animal = zebra;
```

All Zebras are Animals. But this is not:

```
abstract class Animal { }
class Zebra extends Animal { }
class Giraffe extends Animal { }

Animal animal = new Giraffe(); // Ok.
Zebra zebra = animal; // Incompatible types.
```

Not all Animals are Zebras...

Arrays

Arrays are also **covariant**. So this is possible:

```
abstract class Animal { }
class Zebra extends Animal { }

Zebra[] zebras = new Zebra[10];
Animal[] animals = zebras;
```

All groups of Zebras are a group of Animals. But this is not:

```
abstract class Animal { }
class Zebra extends Animal { }
class Giraffe extends Animal { }

Animal[] animals = new Giraffe[10]; // Ok.
Zebra[] zebras = animals; // Incompatible types.
```

Not all groups of Animals are a group of Zebras.

Arrays

But we need to be careful:

```
abstract class Animal { }
class Zebra extends Animal { }
class Giraffe extends Animal { }

Zebra[] zebras = new Zebra[10];
Animal[] animals = zebras;

animals[0] = new Zebra(); // Ok
animals[1] = new Giraffe(); // ArrayStoreException
```

This will throw an **exception** at **runtime**, as the **animals** array is still just an array of zebras.

Generics

With generic types, due to something called **"type erasure"**, Java has no way of knowing at runtime the type information of the type parameters.

Because of this, **generics are invariant**.

```
ArrayList<Zebra> zebras = new ArrayList<>();
ArrayList<Zebra> theSameZebras = zebras; // Ok.
ArrayList<Animal> animals = zebras; // Incompatible types.
```

Wildcards

Wildcards

Instead of a **type variable** like <T>, we can use a wildcard like <?> to represent an unknown type:

```
ArrayList<Zebra> zebras = new ArrayList<>();
ArrayList<Zebra> theSameZebras = zebras; // Ok.
ArrayList<?> animals = zebras;
```

Or create a method that receives a list of **"unknowns"**.

```
public void printList (List<?> someList) {
  for (Object o : someList)
    System.out.println(o);
}
```

We can use it like this:

```
List<Zebra> zebras = new ArrayList<>();
printList(zebras);
```

Wildcards

However, this does not work:

```
List<?> zebras = new ArrayList<>();
zebras.add(new Zebra()); // add (<?>) in List cannot be applied
```

A list of **"unknowns"** is still a list of something.

We just don't know what it is.

We can't just start stuffing zebras inside one.

Bounded Wildcards

We can, however, use extends and super to create more flexible lists.

A list of animals or any subtype (upper-bounded):

```
List<? extends Animal> animals = new ArrayList<>();
```

A list of animals or any supertype (lower-bounded):

```
List<? super Animal> animals = new ArrayList<>();
```

Upper-bounded wildcards are **variant**, while lower-bounded wildcards are **contravariant**.

Read and Write

Covariant types are **"read-only"** (kind of):

```
List<? extends Animal> animals = new ArrayList<>();
for (Animal a : animals)
    System.out.println(a);
animals.add(new Zebra());  // add (<? extends Animal>) in List
                           // cannot be applied to (Zebra)
((List<Zebra>)animals).add(new Zebra()); // Ok.
                                         // But we can get a ClassCastException
```

We cannot be **certain** that the list is of **Zebras**. This is not a list of **Animals**, it is a list of some **"unknown"** Animal type.

Contravariant types are **"write-only"** (kind of):

```
List<? super Animal> animals = new ArrayList<>();
animals.add(new Zebra());
for (Animal a : animals)
    System.out.println(a); // Incompatible types.
for (Object a : animals)
    System.out.println(a); // Ok. But we don't know the real type.
```

Examples

If we don't care about the data inside a collection:

```
public int getSize(Collection<?> collection) {
    return collection.size();
}
```

If we just need a list where we can put animals. Even if it is a List<Object>.

```
public void addToList(Animal animal, List<? super Animal> list) {
    list.add(animal);
}
```

If we want to make sure the List contains animals. It can be a List<Zebra>.

```
public void makeEat(List<? extends Animal> list) {
    for (Animal animal : list)
        animal.eat();
}
```

Observer Example

Generic Observer Interface

```
public interface Observer<T> {
    public void changed(T observable);
}
```

Generic Observable Class

```
public class Observable<T> {
    private List<Observer<T>> observers;

    public Observable() {
        this.observers = new ArrayList<>();
    }

    public void addObserver(Observer<T> observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer<T> observer) {
        observers.remove(observer);
    }

    public void notifyObservers(T subject) {
        for (Observer<T> observer : observers)
            observer.changed(subject);
    }
}
```

Light

```
public class Light extends Observable<Light> {
    private boolean turnedOn;

    public Light(boolean turnedOn) {
        this.turnedOn = turnedOn;
    }

    public boolean isTurnedOn() {
        return turnedOn;
    }

    public void setTurnedOn(boolean turnedOn) {
        this.turnedOn = turnedOn;
        this.notifyObservers(this);
    }
}
```

Client

```
public class Application {
    public static void main(String[] args) {
        Light light = new Light(false);

        light.addObserver(new Observer<Light>() {
            @Override
            public void changed(Light light) {
                System.out.println("LIght: " + light.isTurnedOn());
            }
        });

        light.setTurnedOn(true);
        light.setTurnedOn(false);
        light.setTurnedOn(true);
    }
}
```

Transformation Example

Generic Transformation Interface

Transforms one variable into another having the same type.

```
interface Transformation<T> {
    public T execute(T argument);
}
```

Generic Transformation Line

Does a series of transformations in sequence:

```
public class TransformationLine<T> {
    private List<Transformation<T>> transformations;

    public TransformationLine() {
        this.transformations = new ArrayList<>();
    }

    public void addTransformation(Transformation<T> transformation) {
        this.transformations.add(transformation);
    }

    public T execute(T value) {
        T current = value;
        for (Transformation<T> transformation : transformations)
            current = transformation.execute(current);
        return current;
    }
}
```

Client

```
TransformationLine<Integer> tl = new TransformationLine<>();

tl.addTransformation(new Transformation<Integer>() {
    @Override
    public Integer execute(Integer argument) {
        return argument + 1;
    }
});

tl.addTransformation(new Transformation<Integer>() {
    @Override
    public Integer execute(Integer argument) {
        return argument * 2;
    }
});

tl.addTransformation(new Transformation<Integer>() {
    @Override
    public Integer execute(Integer argument) {
        return argument + 5;
    }
});

System.out.println(tl.execute(10));
```