CODEGYM

Mentor-supported training program from Codegym

Java developer in 12 months

MODULE 2. JAVA CORE

Lesson #5

Generics

CODEGYM

Lesson plan

- Reasons for appearance of generics
- Type erasure
- Generics: wildcards
- Generics: type safe methods

CODEGYM

Java Generics

This is one of the most significant changes in the history of the Java language. “Generics" became available with Java 5 and made it easier, more convenient, and safer to use the Java Collection Framework. Type misuse errors are now detected at compile time.

In Java, Generics are classes which contain type parameters.

```
// Reasons for introduction of generics

ArrayList stringList = new ArrayList();
stringList.add("abc"); // add a string to the list
stringList.add("abc"); // add a string to the list
stringList.add( 1 ); // add a number to the list
for(Object o: stringList){
    String s = (String) o; // we will get an exception
here once we reach the number
}
```

```
// How Generics solve the problem:

ArrayList<String> stringList = new ArrayList<String>();
stringList.add("abc"); // add a string to the list
stringList.add("abc"); // add a string to the list
stringList.add( 1 ); // we will get a compilation error here
for(Object o: stringList){
    String s = (String) o;
}
```

CODEGYM

Type erasure

In fact, no information about their parameter types was added to the types, and all the magic happened at the compilation stage.
Inside the generic class, no information about its parameter type is stored. This approach was later called **type erasure**.

Code with generics	What actually happens

`List<String> strings = new ArrayList<String>();`
`strings.add("abc");`
`strings.add("abc");`
`strings.add(1); // compilation error`
 	
`for (String s : strings) {`
`  System.out.println(s);`
`}`
	`List strings = new ArrayList();`
	
	`strings.add((String) "abc");`
	`strings.add((String) "abc");`
	`strings.add((String) 1); // compilation error`
	
	`for (String s : strings) {`
	`  System.out.println(s);`
	`}`

CODEGYM

Example of your own class with a type parameter

```
class Zoo<T>{
    T t;

    T getT(){
        return t;
    }

    void setT (T t){
        this.t = t;
    }
}
```

The class does not contain information about the type which was passed

```
class Zoo<T>{
    ArrayList<T> pets = new ArrayList<T>();
    public T createAnimal(){
        T animal = new T();// we will have an
error here!
        pets.add(animal)
        return animal;
    }
}
```

CODEGYM

Parent type for type parameters

Java allows you to specify a parent type for parameter types. For this, the extends keyword is used.

```
class Zoo<T extends Cat>{
    T cat;
    T getCat(){
        return cat;
    }
    void setCat (T cat){
        this.cat = cat;
    }
}
```

Firstly, it is now only possible to pass the Cat type or one of its descendants as a type parameter.

Secondly, in the Zoo class, variables of type T can now call methods of the Cat class. Because instead of type T, the Cat type will be used everywhere.

CODEGYM

Generics: wildcards

Let's say that MagicWarrior is a child class of Warrior.

In that case List<MagicWarrior> will not be a descendant of List<Warrior>. To fix this situation a more complex design is used. It looks like this:

```
List<? extends Warrior>
```

«? extends Warrior» means «any type, inherited from Warrior».

If a List with any parameter is needed, then two versions can be used:

```
List<? extends Object>
```

```
List<?>
```

CODEGYM

Generics: type safe methods

Class methods can also have their own parameter types.

```
class Calculator {
    <T> T add(T a, T b); // add
    <T> T sub(T a, T b); // subtract
    <T> T mul(T a, T b); // multiply
    <T> T div(T a, T b); // divide
}
```

```
public void doSomething(List<? extends MyClass> list){
    for(MyClass object : list){
        System.out.println(object.getState()); // everything works fine here
    }
}
```

In general, you can pass a List to the doSomething method with an element type of not only MyClass, but of any of the descendants of MyClass. And in such a list (of heirs) it is not possible to add MyClass objects!

```
public void doSomething(List<? extends MyClass> list){
    list.add(new MyClass()); // error!
}
```

To solve this the following design must be used:

```
List<? super MyClass> list
```

«? extends T» means that the class must be a descendant of T.

«? super T» means that the class must be a parent of T.

CODEGYM

How to avoid type erasure

The value of the class field of type Integer (i.e., Integer.class) is actually an object of type Class<Integer>

Taking advantage of the fact that Class<T> is Generic, and the fact that a variable of its type can only store a value of type T, we can do this tricky combination:

```
class Zoo<T> {
    Class<T> clazz;
    ArrayList<T> animals = new ArrayList<T>();
    Zoo(Class<T> clazz){
        this.clazz = clazz;
    }
    public T createNewAnimal(){
        T animal = clazz.newInstance();
        animals.add(animal);
        return animal
    }
}
```

CODEGYM

How to use it

```
Zoo<Tiger> zoo = new Zoo<Tiger>(Tiger.class); // we pass the type here!
Tiger tiger = zoo.createNewAnimal();
```

This is not a mega-cunning combination - we just pass a reference to the desired type. But if we just used Class instead of Class<T>, then someone could mistakenly pass two different types - one as a T parameter, and the other to the constructor.

CODEGYM

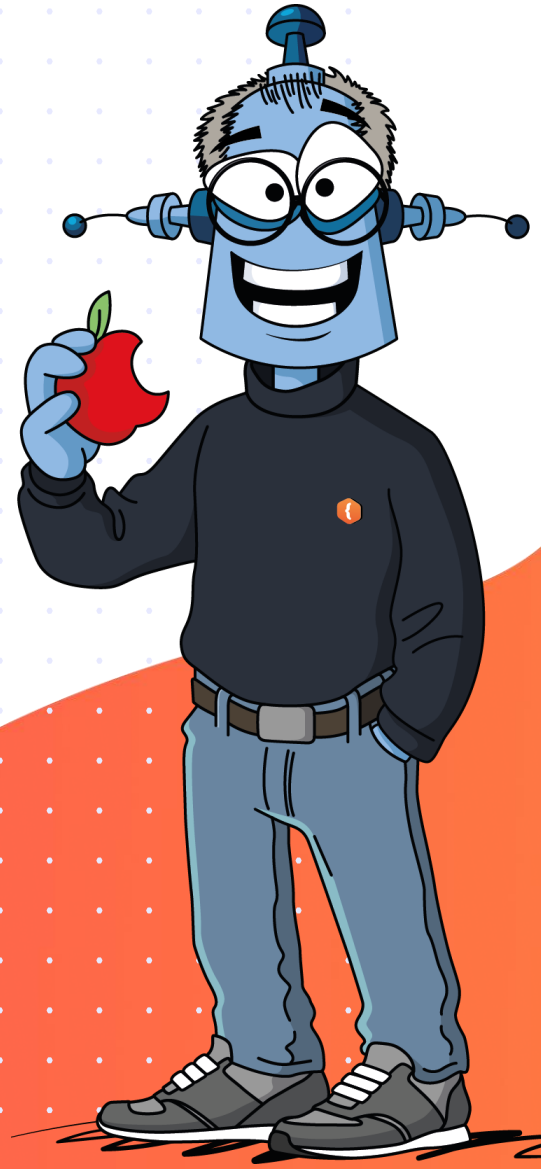
Homework

Module 2
Level 5. Generics

MODULE 2. JAVA CORE

CODEGYM

Questions and Answers



MODULE 2. JAVA CORE