DEVELHOPE

#codeforimpact

Generics and Wildcards

Generics

Generics are a concept introduced in Java 5.0 and they let the developer write more *type-safe* collections.

Before Java 5.0, the *compiler* didn't care about what *type* of *elements* you could put into a *collection*.

For example, if you had an `ArrayList` that was supposed to contain just `Car` objects, the *compiler* would let you put `Human` *objects* into the *collection*.

Thanks to the *generics*, it's now possible to put inside a *collection* only *objects* of the same *type* defined by the *generics*.

In the following example, the generics `<Car>` has the `Car` *type argument*:

```
List<Car> carsList = new ArrayList<Car>();
```

Classes before Generics

Let’s consider the following example:

```
public class Box {
      private Object object;

      public void set(Object object) {
            this.object = object;
      }

      public Object get() {
            return object;
      }
}
```

Problem

The `set()` method accepts an `Object`, so you are free to pass *whatever* `Object` you want.

The problem is that you could mistakenly pass a `String` parameter to the `set()` method.

There is no way to verify - at *compile time* - how the `set()` method is called.

So you will spot the error only at a *runtime*.

Classes with Generics

Let's consider the previous example code with *generics*:

```
public class Box<T> {
      private T t;

      public void set(T t) {
            this.t = t;
      }

      public T get() {
            return t;
      }
}
```

Solution

You can see that all occurrences of `Object` are replaced by `T`.

`T` is a *type parameter* and it that can be used anywhere inside the class (as *type variable*).

A *type variable* can be any *non-primitive type* you specify.

DEVELHOPE

#codeforimpact

Classes with Generics

If you want to use the `Box` class in the code, you have to do a *generic type invocation*, which replaces `T` with some concrete *value.*

```
Box<Integer> integerBox = new
Box<Integer>();



Box<String> stringBox = new Box<String>();
```

`integerBox` is related to a *generic type invocation* where `T` is replaced with `Integer`.

`stringBox` is related to a *generic type invocation* where `T` is replace with `String`.

Type Parameter Naming Conventions

By convention, *type parameter* names are single, uppercase letters.

The most commonly used type parameter names are:

- **`E`** - *Element* (used a lot for the *Java Collections* like `List`, `ArrayList`, `Map`, etc.)
- **`K`** - *Key*
- **`N`** - *Number*
- **`T`** - *Type*
- **`V`** - Value

DEVELHOPE

#codeforimpact

Wildcards

The question mark `?` is known as *wildcard* and in Java it represents an *unknown type.*

A *wildcard* can be really useful with methods that don't depend on the type parameter.

Let's see a practical example:

```
void printList(List<?> list) {
          System.out.println(list);
}
```

This way you can pass both a `List<Integer>` and a `List<Double>` to the method `printList()`.

Bounded Wildcards

In Java there are two types of *bounded wildcards*:

- *Upper Bounded Wildcards*
  - for *relaxing* restrictions on a variable
  - syntax: `<? extends Type>`
  - example: `List<? extends Number>`

- *Lower Bounded Wildcars*
  - for *imposing* restrictions on a variable
  - syntax: `<? super Type>`
  - example: `List<? super Integer>`