FREE eBook

LEARNING generics

Free unaffiliated eBook created from
Stack Overflow contributors.

#generics

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: generics

It is an unofficial and free generics ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official generics.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with generics

Remarks

Generics allow you to define placeholders for exact types in definitions for classes, interfaces and/or methods.

Possible subtopics:

- Class (and structure) generics
- Interface generics
- Method generics
- Constraints
- Covariance and contravariance

Examples

Availability

Generics became available with:

- .NET Framework 2.0 (and version 2.0 of the compact framework).
- Java in version 5.
- Common Lisp since it was standardised ...

Read Getting started with generics online: https://riptutorial.com/generics/topic/4454/getting-started-with-generics

Chapter 2: Generics in Java

Syntax

- `class MyClass<T1, T2 extends CharSequence> implements Comparable<MyClass> //...`
- `interface MyListInterface<T extends Serializable> extends List<T> //...`
- `public <T1, T2 extends Instant> T1 provideClone(T1 toClone, T2 instant> //...`
- `public static List<CharSequence> safe(Collection<? extends CharSequence> l) { return new ArrayList<>(l);}`
- `Set<String> strings = Collections.singleton("Hello world");`
- `List<CharSequence> chsList = safe(strings);`

Remarks

Type erasure limits reflection, though that is not JVM specific, for example Ceylon uses reified generics.

Existential type support is not necessarily supported by other languages in this form: Kotlin supports it through type projections.

Examples

Introduction

Generics was introduced in Java in its version (1.)5. These are erased during compilation, so runtime reflection is not possible for them. Generics generate new types parametrized by other types. For example we do not have to create new classes in order to use type safe collection of `String`s and `Number`s, generic `ArrayList<T>` can be used in all cases, like: `new ArrayList<String>()`.

Example:

```
List<String> variable = new ArrayList<String>();
```

In Java 7 some syntactic sugar was introduced to ease the construction (`<>` aka. diamond):

```
List<String> variable = new ArrayList<>();
```

Interestingly it was also possible (from Java 5) to use type inference, when a static method had as a return value (often used in Google Guava for example):

```
List<String> singleton = Collections.singletonList();//Note the missing `<>` or `<String>`!
```

In Java existential types were used to provide polymorphism for the types, as the generic types are invariant (for example: `List<String>` is not a subtype, nor a supertype of `List<CharSequence>`, although in Java `String[]` is a subtype of `CharSequence[]`; note: `String` implements the `CharSequence` interface). Existential generic types can be expressed as:

```
List<? extends CharSequence> list = new ArrayList<String>();
Comparable<? super ChronoLocalDate> ccld = LocalDate.now();
ChronoLocalDate cld = JapaneseDate.now(); //ChronoLocalDate extends
Comparable<ChronoLocalDate>
ccld.compareTo(cld);
//cld.compareTo(ccld);//fails to compile because ccld is not a `ChronoLocalDate` (compile
time)
```

Both instances can be used in a list parametrized by the corresponding `Comparable`:

```
List<Comparable<? super ChronoLocalDate>> list2 = new ArrayList<>();
list2.add(cld);
list2.add(ccld);
```

Generic Methods

Generic *type parameters* are commonly defined at the class or interface level, but *methods* and (rarely) *constructors* also support declaring type parameters bound to the scope of a single method call.

```
class Utility // no generics at the class level
{
    @SafeVarargs
    public static <T> T randomOf(T first, T... rest) {
        int choice = new java.util.Random().nextInt(rest.length + 1);
        return choice == rest.length ? first : rest[choice];
    }

    public static <T extends Comparable<T>> T max(T t1, T t2) {
        return t1.compareTo(t2) < 0 ? t2 : t1;
    }
}
```

Notice the type parameter declarations, `T` and `<T extends Comparable<T>>` respectively, appear after the method modifiers and *before* the return type. This allows the type parameter `T` to be used within the scope of such methods, acting as:

- argument types
- return type
- local variable types

Though both methods above use the same type parameter name `T`, at the method level they are completely independent of each other. The compiler will *infer* the actual type based on the arguments passed to the method *at each call site* that invokes the method. Since the `max` method declares that `T extends Comparable<T>`, the compiler also enforces that the inferred types are compatible implementations of the `Comparable` interface.

```
Integer num1 = 1;
Integer num2 = 2;
String str1 = "abc";
String str2 = "xyz";
```

```
Integer bigger = Utility.max(num1, num2);
assert bigger == num2;

String later = Utility.max(str2, str1);
assert later == str2;

Utility.max(num1, str1); // compiler error: num1 and str1 are incompatible types

Utility.max(new Object(), new Object()); // compiler error: Object does not implement
Comparable
```

Java 8 significantly improved the compiler's ability to correctly infer the generic types at call sites. If the compiler fails to infer the proper type, developers can explicitly state the type as a part of the call:

```
Object obj = Utility.<Object>randomOf(str1, new Object(), num1);
```

Read Generics in Java online: https://riptutorial.com/generics/topic/6552/generics-in-java

Credits

S. No	Chapters	Contributors
1	Getting started with generics	Community, Mark Hurd
2	Generics in Java	Gábor Bakos, William Price