Theory: Type Erasure

23 minutes 0 / 5 problems solved

Generics were introduced to implement generic programming concept and control type-safety at compile-time. The feature has been available since Java 5. To support backward compatibility with previous Java versions, information about types is erased by the compiler.

This means that at runtime all these objects will have the same type:

```
// Generic types
List<Integer> integers = new List<>();
List<String> strings = new List<>();

// Raw type
List objects = new List();
```

The transformation process is called **type erasure**. Let's take a closer look at what it actually does.

§1. Generics replacement

First of all, type erasure replaces parameters of generic types with their left bound. In the case of unbounded ones, they are replaced by `Object` type. It means that information about types is erased when a program is translated into byte code. As a result, byte code contains only ordinary non-generic classes and interfaces.

Let's consider the generic class `Data`:

```
class Data<T> {
    private T data;

    public T get() {
        return data;
    }

    public void set(T data) {
        this.data = data;

    }

}
```

Java compiler will replace the parameter `T` by `Object`, because `T` is unbounded. Below is a code that is effectively the same as `Data<T>` after compilation:

```
class Data {
    private Object data;

    public Object get() {
        return data;
    }

    public void set(Object data) {
        this.data = data;

    }

}
```

Now suppose `Data` is parameterized by `<T extends Number>`. In this case, transformed code looks similar to the last code snippet with one difference: `Object` will be replaced by `Number`.

If a value is assigned, a generic replacement can affect the accuracy of the program. If it is necessary to preserve type safety, the compiler inserts type casting. Let's look at the code:

Current topic:

Type Erasure

Topic depends on:

- ✕ Type Bounds
- ✕ Wildcards

Topic is required for:

- Generics and Reflection
- Reification

Table of contents:

```
Data<String> data = new Data("stored value");
String stored = data.get();
```

After type erasure is performed the code above is equivalent to:

```
Data data = new Data("stored value");
String stored = (String) data.get();
```

§2. Bridge methods

Sometimes to add type casting for preserving polymorphism the compiler has to generate synthetic methods. Let's now consider the extension of an already familiar `Data` class:

```
public class NumberData extends Data<Number> {
    public void set(Number number) {
        System.out.println("NumberData set");
        super.set(number);
    }
}
```

After type erasure `NumberData` method becomes `set(Number number)`, while the original `Data` method is `set(Object obj)`. As `NumberData` extends `Data`, it is possible to invoke `set(Object obj)` from `NumberData` instance and set objects of arbitrary type. To solve this problem and preserve the polymorphism of generic types after type erasure, Java compiler generates a so-called **bridge method** in the `NumberData` class. It overrides parameterized parent methods and provides type casting to specific parameters

```
public class NumberData extends Data {
    // Bridge method generated by the compiler
    public void set(Object object) {
        super.set((Number) object);
    }

    public void set(Number number) {
        super.set(number);
    }


    ...

}
```

A bridge method is a synthetic method created by the compiler as a part of the type erasure process. It is presented in byte code only and not available for direct usage from java code. Normally you don't face bridge methods explicitly, although sometimes they might appear in a stack trace.

§3. Conclusion

Type erasure is an operation that JVM runs during the compilation of source java code to byte code. It replaces the information about parameterization types by their upper bounds. It also inserts type casting and generates bridge methods. Type erasure transformation greatly influences the design of Java classes.

Report a typo

37 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

This content was created 6 months ago and updated 3 days ago. Share your feedback below in comments to help us improve it!

Comments (5) Hints (0) Useful links (0) Show discussion