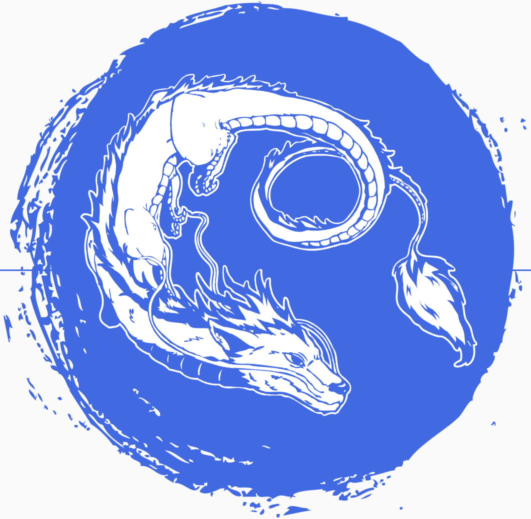
Genericity in Java

YAKA 2022 Team



Copyright

Rules

- You must have downloaded your copy from the Assistants' Intranet <https://intra.assistants.epita.fr>.
- This document is strictly personal and must **not** be passed on to someone else.
- Non-compliance with these rules can lead to severe sanctions.

Genericity

- Introduced in Java 5.
- See C++ templates?
- It’s totally different.
- However, they serve the same purpose.
- Like in C++, you can make methods and classes generic.

Why should I use genericity?

- Avoid code duplication;
- Preserve type safety;
- Avoid unnecessary casts.

How to make a method generic

```java
public class Main {
    public static <ELEMENT_TYPE> List<ELEMENT_TYPE> makeList(ELEMENT_TYPE ...  elements) {
        return Arrays.asList(elements);
    }

    public static void main(String[] args) {
        final var listInt = makeList(1, 2, 3, 4, 5);
        System.out.println(listInt.toString());

        final var listString = makeList("Constatez maître comme moi?", "Je ne vois plus de tache.");
        System.out.println(listString.toString());
    }

    /* Output:
        [1, 2, 3, 4, 5]
        [Constatez maître comme moi?, Je ne vois plus de tache.]
     */
}
```

How to make a class generic

```java
public class Main {
    public static class GenericClass<PRINTED_TYPE> {
        private final PRINTED_TYPE toPrint;

        public GenericClass(PRINTED_TYPE toPrint) {
            this.toPrint = toPrint;
        }

        public void print() {
            System.out.println(toPrint);
        }
    }
    // To be continued...
```

How to make a class generic - main

```java
    // Append this to the end of the previous code block!
    public static void main(String[] args) {
        final var genericOnInteger = new GenericClass<Integer>(47);
        final var genericOnString = new GenericClass<String>("Oh, to be a generic class in Java...");

        genericOnInteger.print();
        genericOnString.print();
    }

    /* Output:
        47
        Oh, to be a generic class in Java...
     */
}
```

Primitive types

- A generic type has to be a class or interface: **you cannot use a primitive type as a generic type**.

```
    // Replace the previous main with this one!
    public static void main(String[] args) {
        final var genericOnInteger = new GenericClass<Integer>(47);
        final var genericOnInt = new GenericClass<int>(1); /* Error: type argument cannot be of
                                                             * primitive type
                                                             */

        genericOnInteger.print();
        genericOnInt.print();
    }
}
```

Bounds - Problem setup

```java
public class Main {
    public static abstract class ParentClass {
        public abstract void myPrint();
    }

    public static class FirstChildClass extends ParentClass {
        @Override
        public void myPrint() {
            System.out.println("First child class!");
        }
    }

    public static class SecondChildClass extends ParentClass {
        @Override
        public void myPrint() {
            System.out.println("Second child class!");
        }
    }
    // To be continued...
```

Bounds - The problem itself

```
    // Append this to the end of the previous code block!
    public static class GenericClass<PRINTED_TYPE> {
        PRINTED_TYPE toPrint;

        public GenericClass(PRINTED_TYPE toPrint) {
            this.toPrint = toPrint;
        }

        public void myPrint() {
            toPrint.myPrint(); // Error: We don't know if PRINTED_TYPE has a method named 'myPrint'
        }
    }
}
```

- To solve this problem, you can put restrictions on generic types!

The `extends` keyword

- The type on the left of this keyword either inherits from the type on its right
- Extended types/interfaces can be chained with &

```
// Let's go back to the previous example...
public static class GenericClass<PRINTED_TYPE extends ParentClass> {
    private final PRINTED_TYPE toPrint;

    public GenericClass(PRINTED_TYPE toPrint) {
        this.toPrint = toPrint;
    }

    public void myPrint() {
        toPrint.myPrint(); // Fine: ParentClass defines a myPrint method!
    }
}
// To be continued...
```

The `extends` keyword - main

```java
// Append this to the end of the previous code block!
public static void main(String[] args) {
    final var firstChildInstance = new FirstChildClass();
    final var secondChildInstance = new SecondChildClass();

    final var firstGenericInstance = new GenericClass<FirstChildClass>(firstChildInstance);
    final var secondGenericInstance = new GenericClass<SecondChildClass>(secondChildInstance);

    firstGenericInstance.myPrint();
    secondGenericInstance.myPrint();
}

/* Output:
    First child class!
    Second child class!
 */
```

Type erasure

- The type of a generic class containing its generic type parameter is actually its **static type**.
- Its **dynamic type** (called *raw type*) does not include its generic type parameter!
- Legacy from Java versions beneath 5, kept for bytecode retrocompatibility.

Type erasure - Example

```
public class Main {
    public static void main(String[] args) {
        final var myIntegerList = new ArrayList<Integer>();
        final var myStringList = new ArrayList<String>();

        System.out.println("Type of myIntegerList: " + myIntegerList.getClass().toString());
        System.out.println("Type of myStringList: " + myStringList.getClass().toString());
        System.out.println(myIntegerList.getClass().equals(myStringList.getClass()));
    }

    /* Output:
        Type of myIntegerList: class java.util.ArrayList
        Type of myStringList: class java.util.ArrayList
        true
     */
}
```

Type erasure - Workaround of choice

- Store the generic type in a class attribute

```java
public class Main {
    public static class GenericClass<STORED_TYPE> {
        public final Class genericType;
        // Other attributes...

        public GenericClass(final Class genericType) {
            this.genericType = genericType;
            // Other assignations...
        }

        // Methods...
    }
    // To be continued...
```

Type erasure - Workaround of choice - main

```
    // Append this to the end of the previous code block!
    public static void main(String[] args) {
        final var genericIntegerInstance = new GenericClass<Integer>(Integer.class);
        final var genericStringInstance = new GenericClass<String>(String.class);

        System.out.println(genericIntegerInstance.genericType.toString() + " stored at runtime!");
        System.out.println(genericStringInstance.genericType.toString() + " stored at runtime!");
    }

    /* Output:
        class java.lang.Integer stored at runtime!
        class java.lang.String stored at runtime!
     */
}
```

Type erasure - Static methods and attributes

- Beware: static members of a generic class being related to its raw type, they cannot use its generic type.
- Calling a static method or accessing a static attribute of a generic class must be done through its raw type.

Type erasure - Static methods and attributes - Example

```
public class Main {
    public static class GenericClass<PLACEHOLDER_TYPE> {
        public static void printGenericType() {
            PLACEHOLDER_TYPE randomVariable; // Error: Cannot access 'this' from a static context
        }

        public static void myStaticPrint() {
            System.out.println("Generic static print!");
        }
    }

    public static void main(String[] args) {
        GenericClass<Integer>.myStaticPrint(); // Error: Cannot resolve myStaticPrint
        GenericClass.myStaticPrint(); // Good!
    }
}
```

Inheritance can be misleading

- Beware: if a class B inherits from a class A and C is a generic class, it **doesn’t mean that C<B> inherits from C<A>**.

```java
public class Main {
    public static class GenericClass<PLACEHOLDER_TYPE> {
    }

    public static void main(String[] args) {
        final Object polymorphicString = new String(); /* Good: String inherits from Object,
                                                        * polymorphism applies
                                                        */
        final GenericClass<Object> tryPolymorphicGenericInstance = new GenericClass<String>();
        // Error: GenericClass<String> doesnt inherit from GenericClass<Object>!
    }
}
```

Generics and methods

- What happens when you want a method to take as argument an instance of a generic class which generic type we don’t know?

```
public class Main {
    public static List<Object> getMatchingElements(final List<Object> list, Object o) {
        final var returnedList = new ArrayList<Object>();
        for (final var elm : list) {
            if (elm.equals(o)) {
                returnedList.add(o);
            }
        }
        return returnedList;
    }

    public static void main(String[] args) {
        final var listString = List.of("This", "will", "not", "work");
        final var subList = getMatchingElements(listString, "See?");
        // Error: as we saw before, List<String> cannot be cast to List<Object>.
    }
}
```

Wildcards

- A generic class which generic type is undefined.
- Use it when you don’t know the generic type of a class passed as argument to or returned by a method.

Wildcards - Example

```java
public class Main {
    public static List<?> getMatchingElements(final List<?> list, Object o) {
        final var returnedList = new ArrayList<Object>();
        for (final var elm : list) {
            if (elm.equals(o)) {
                returnedList.add(o);
            }
        }
        return returnedList;
    }
    // To be continued...
```

Wildcards - Example - main

```
    // Append this to the end of the previous code block!
    public static void main(String[] args) {
        final var listString = List.of("This", "will", "do", "better", "will", "it", "not?");
        final var subList = getMatchingElements(listString, "will");
        System.out.println(subList.toString());
    }

    /* Ouptut:
        [will, will]
     */
}
```

Wildcard - Problem

- But having a completely unknown type in our method can raise some issues:

```java
public class Main {
    public static abstract class ParentClass {
        public abstract void myPrint();
    }

    public static class FirstChildClass extends ParentClass {
        @Override
        public void myPrint() {
            System.out.println("First child class!");
        }
    }

    public static class SecondChildClass extends ParentClass {
        @Override
        public void myPrint() {
            System.out.println("Second child class!");
        }
    }
    // To be continued...
```

Wildcard - Problem

```java
    // Append this to the end of the previous code block!

    public static void addNewElement(final List<?> list) {
        list.add(new FirstChildClass()); /* Error: We don't know what type of elements list is
                                          * supposed to contain
                                          */
    }

    public static void printAllElements(final List<?> list) {
        for (final var elt : list) {
            elt.myPrint(); // Error: We don't know if elt's type defines a myPrint method.
        }
    }
}
```

Bounded wildcards

- You can use the `extends` keyword, which we have previously seen, to restrict the generic type of a wildcarded class.
- You can also use the `super` keyword.
- The type on the left of the `super` keyword is a supertype of the one on its right.

Bounded wildcards - Example

```java
// Let's go back to the previous example...
public static void printAllElements(final List<? extends ParentClass> list) {
    for (final var elt : list) {
        elt.myPrint(); // Much better!
    }
}

public static void addNewElement(final List<? super FirstChildClass> list) {
    list.add(new FirstChildClass()); /* Good: FirstChildClass can be cast to its supertype through
                                      * polymorphism
                                      */
}
// To be continued...
```

Bounded wildcards - Example

```java
// Append this to the end of the previous code block!
public static void main(String[] args) {
    final var myFirstChildClassList = new ArrayList<ParentClass>();
    addNewElement(myFirstChildClassList);
    addNewElement(myFirstChildClassList);
    final var mySecondChildClassList = List.of(new SecondChildClass(), new SecondChildClass(),
                                               new SecondChildClass());

    printAllElements(myFirstChildClassList);
    printAllElements(mySecondChildClassList);
}
```

Bounded wildcards - Rule of thumb

- If you only read from your class, use `extends`.
- If you only write in your class, use `super`.
- If you want to do both... Don't use wildcards.

Questions?

- Any questions?