Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Generics (Part 1)

Object Oriented Programming

Tiziano Leidi

15.11.2021

Strong Typing

We know that Java is a strong typed language.

Compared to programming language that are weak typed, the advantage is that the compiler is able to detect data-type incompatibilities.

In weak typed programming languages, if these types of incompatibilities are not fixed before program execution, run-time errors potentially occurs.

Strong Typing

However, strong-typing also has drawbacks.

In particular, it takes longer to write the program, with consequent larger and more complex source code.

In addition, when OOP techniques such as inheritance and polymorphism are not properly used, the source code might fill with type casts that potentially limit the reusability of the code.

Use of explicit casts should be avoided. For example, we already know how to prevent hard-coded sections of code.

Strong Typing

But, in some situations this goal might be difficult to achieve.

For example, consider the following assignment: you have to develop a reusable class for managing coordinates in the 3D space (x, y, z).

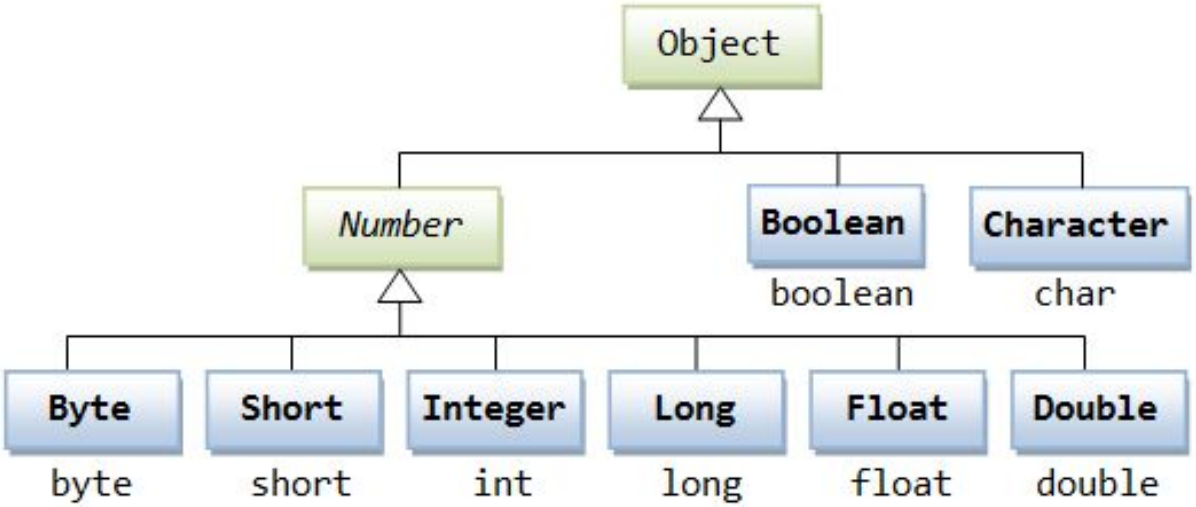
For reusability purposes, it should be possible to define the type of the coordinates as shirt, int, long, float, double, or even String, depending on the context of use.

How do you propose to implement the class?

```
public class Coordinate {
    private float x;
    private float y;
    private float z;

    public Coordinate(float x, float y, float z) {
        this.x = x;  this.y = y;  this.z = z;
    }

    public float getX() {
        return x;
    }
    public float getY() {
        return y;
    }
    public float getZ() {
        return z;
    }

// ...
}
```

Is this solution reusable?

We are experts in Java … let's take advantage of OOP techniques, Java's wrapper classes and auto-boxing and -unboxing:



```
public class Coordinate {
    private Number x;
    private Number y;
    private Number z;

    public Coordinate(Number x, Number y, Number z) {
        this.x = x;  this.y = y;  this.z = z;
    }

    public Number getX() {
        return x;
    }
    public Number getY() {
        return y;
    }
    public Number getZ() {
        return z;
    }

// ...
}
```

Is there sufficient flexibility?

What about Strings?

```
public class Coordinate {
    private Object x;
    private Object y;
    private Object z;

    public Coordinate(Object x, Object y, Object z) {
        this.x = x;  this.y = y;  this.z = z;
    }

    public Object getX() {
        return x;
    }
    public Object getY() {
        return y;
    }
    public Object getZ() {
        return z;
    }

// ...
}
```

Problem solved!

Let's use it:

```
public class CoordianteTest {
    public static void main(String[] args) {
        Coordinate coordinate = new Coordinate(12, 4, 33);
        int x = coordinate.getX();
        int y = coordinate.getY();
        int z = coordinate.getZ();
    }
}
```

What's missing here?

```
public class CoordianteTest {
    public static void main(String[] args) {
        Coordinate coordinate = new Coordinate(12, 4, 33);
        int x = (Integer) coordinate.getX();
        int y = (Integer) coordinate.getY();
        int z = (Integer) coordinate.getZ();
    }
}
```

Do you like the cast?

And what about robustness?

```
public class CoordianteTest {
    public static void main(String[] args) {
        Coordinate coordinate = new Coordinate(20, 18.5, "7");
        int x = (Integer) coordinate.getX();
        int y = (Integer) coordinate.getY();
        int z = (Integer) coordinate.getZ();
    }
}
```

Exception in thread "main" java.lang.ClassCastException: class java.lang.Double cannot be cast to class java.lang.Integer

In order to correctly use the Coordinate class:

- We have to take care that the data type is consistent for the specified x, y and z. The compiler is not able to help us.
- If we enhance the class (for example we extend it) with additional methods to modify the values of x, y and z, things become even harder … we have to take care that the data type of x, y, z remains consistent during the whole program execution.

Is this what we expect from a strong typed language?

Generics

Generics is a programming tool that helps in this kind of situations ...

```java
public class Coordinate<T> {
    private T x;
    private T y;
    private T z;

    public Coordinate(T x, T y, T z) {
        this.x = x;  this.y = y;  this.z = z;
    }

    public T getX() {
        return x;
    }
    public T getY() {
        return y;
    }
    public T getZ() {
        return z;
    }

// ...
}
```

```java
public class CoordianteTest {
    public static void main(String[] args) {
        Coordinate<Integer> coordinate =
                                    new Coordinate<Integer>(6, 25, 2);
        int x = coordinate.getX();
        int y = coordinate.getY();
        int z = coordinate.getZ();
    }
}
```

With this version of the class, if you try to add a wrong type of value, a compilation error occurs:

Error: java: incompatible types: double cannot be converted to java.lang.Integer

The problem is detected by the compiler!

Generic Types

The *Coordinate* class has been transformed into a generic type. It's a type of data that can be "parameterized" to make it context-specific at compile time.

A generic type is composed of a class or interface and a section of formal type parameters <T1, T2, ... Tn>.

Each formal type parameter must be replaced by an actual type argument before compilation. The result will be a parameterized type.

Formal Type Parameters

By convention, the names of the formal type parameters are single capital letters.

The most commonly used ones are:

E - Element

K - Key

N - Number

T - Type

V - Value

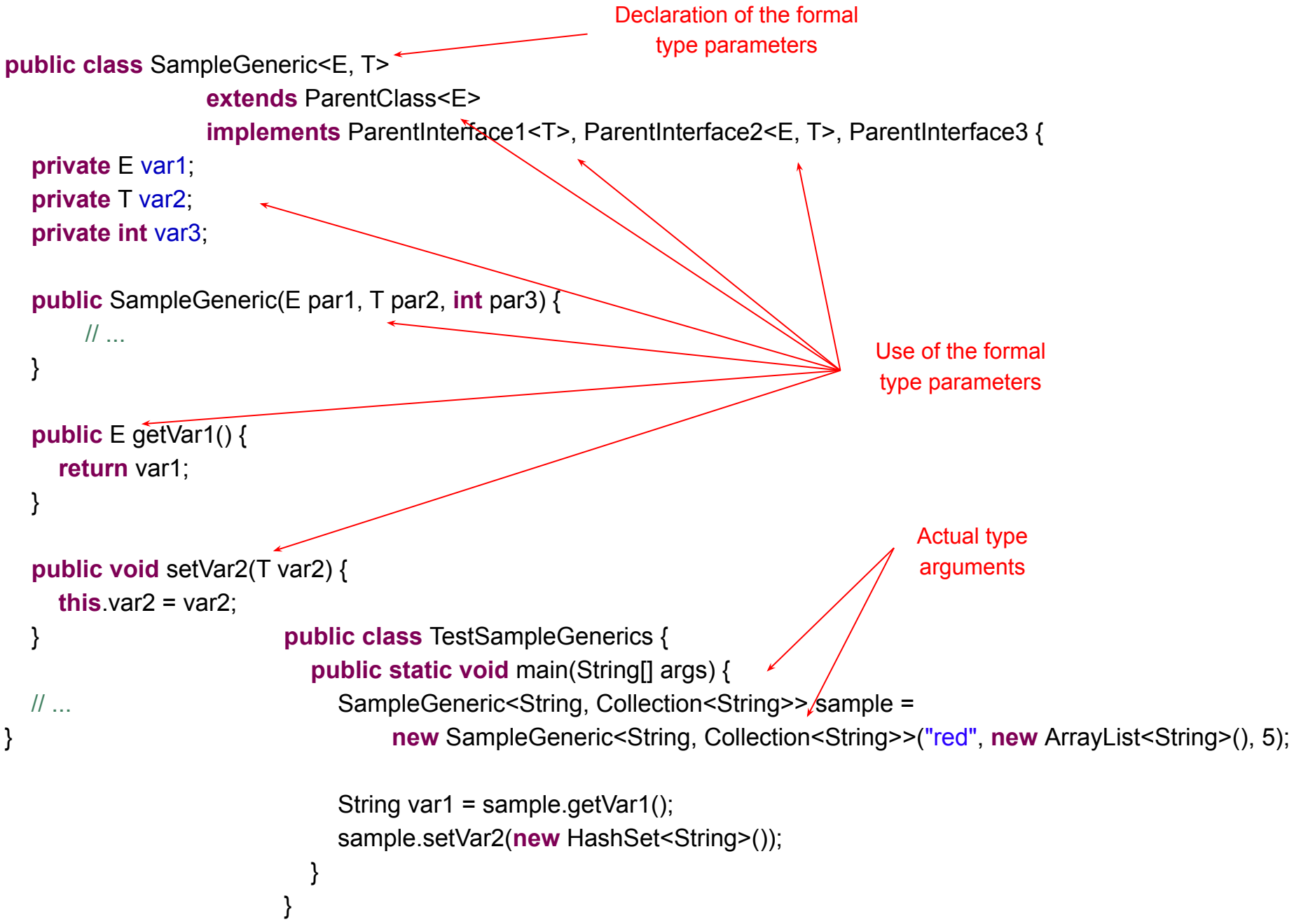
S, U, etc. - 2nd, 3rd, ... types

Formal Type Parameters

Formal type parameters have to be declared in angle brackets "<" and ">" after the class definition name. Then, they can be used in the body of the class as a type for fields, local variables, parameters and method return types.

Actual type arguments have to be provided also in angle brackets after the type name in field, variable and parameters declarations, in method return types as well as during instantiation.

This syntactic solution provides flexibility ...




```
public class SampleGeneric<E, T>
                    extends ParentClass<E>
                    implements ParentInterface1<T>, ParentInterface2<E, T>, ParentInterface3 {
   private E var1;
   private T var2;
   private int var3;

   public SampleGeneric(E par1, T par2, int par3) {
         // ...
   }

   public E getVar1() {
      return var1;
   }

   public void setVar2(T var2) {
      this.var2 = var2;
   }                          public class TestSampleGenerics {
                                 public static void main(String[] args) {
   // ...                            SampleGeneric<String, Collection<String>> sample =
}                                          new SampleGeneric<String, Collection<String>>("red", new ArrayList<String>(), 5);

                                    String var1 = sample.getVar1();
                                    sample.setVar2(new HashSet<String>());
                                 }
                              }
```

RAW types

If needed, generic types can be used without specifying type arguments:

```
Coordinate<Integer> coordinateA =  new Coordinate<Integer>(6, 25, 2);
Coordinate coordinateB =  new Coordinate(6, 25, 2);
```

The result of this operation is called a RAW type. It's similar as using *Object* as a type argument.

The possibility to instantiate RAW types is provided by Java for compatibility reasons.

In general, the use of RAW types is discouraged.

Type Inference

In some situations, the Java compiler is able to infer the type of the actual arguments from the context.

For example, this possibility is available at instationation time:

```
Coordinate<Integer> coordinate =  new Coordinate<>(6, 25, 2);
```

In this case, the type used for object instantiation is inferred from the variable declaration.

The empty pair of angle brackets “<>” is called the diamond.

Generics vs. Polymorphism

Generics and polymorphism are 2 different things:

- Polymorphism focus on data type compatibility.
- Generics is a tool for parameterizable (compile-time) data types.

If required, polymorphism and generics can be mixed.

```java
public class Coordinate<T> {
    private T x;
    private T y;
    private T z;

    public Coordinate(T x, T y, T z) {
        this.x = x;  this.y = y;  this.z = z;
    }

    public T getX() {
        return x;
    }
    public T getY() {
        return y;
    }
    public T getZ() {
        return z;
    }

// ...
}
```

```java
public class CoordianteTest {
    public static void main(String[] args) {
        Coordinate<Number> coordinate =  new Coordinate<>(6, 25, 2L);
        Number x = coordinate.getX();
        Number y = coordinate.getY();
        Number z = coordinate.getZ();
    }
}
```

Generics in the Java Library

Generics are supported by the Java libraries. You already know that many classes of the Collections framework feature Generics.

For example:

```
public class LinkedList<E>
        extends AbstractSequentialList<E>
        implements List<E>, Deque<E>, Cloneable, Serializable
```

It is possible to use it without generics (RAW version):

```
LinkedList list = new LinkedList();
list.add(new Integer(1));
Integer num = (Integer) list.get(0);
```

Or with generics:

```
LinkedList<Integer> list = new LinkedList<>();
list.add(1);
int num = list.get(0);
```

Generic Types and Arrays

In Java, arrays of parameterized data type are not allowed.

For example, it is not possible to define:

new HashMap<String, Object>[];

Generic Types and Arrays

This limitation can be worked around by instantiating an array of the RAW type (HashMap[]) and then cast to the needed parameterized type:

```
HashMap<String, Object>[] myArray =
                        (Map<String, Object>[]) new HashMap[10];
```

This solution produces a warning at compilation time, which can eventually be removed by annotating the code with:

```
@SupressWarnings("unchecked")
```

```java
public class HashMapArray {
    public static void main(String[] args) {
        String str = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam interdum malesuada urna, "
                + "vel ornare massa pellentesque id. Donec ultricies, mauris id rutrum porta, tellus sem accumsan "
                + "risus, eu porttitor urna arcu quis nulla. Etiam lacinia, lorem sit amet sagittis venenatis, "
                + "quam libero semper elit, vel pharetra lectus dolor sit amet risus. In libero ante, "
                + "feugiat eu finibus nec, fermentum ac.";
        String[] words = str.replaceAll("[.,]", "").split(" ");

        @SuppressWarnings("unchecked")
        HashMap<String, Integer>[] maps = (HashMap<String, Integer>[]) new HashMap[6];

        int cnt = 0;
        for (int i = 0; i < maps.length; i++) {
            maps[i] = new HashMap<>();
            for (int j = 0; j < words.length / maps.length; j++) {
                String tmp = words[cnt++];
                maps[i].put(tmp, tmp.length());
            }
        }

        for (HashMap<String, Integer> map : maps) {
            map.forEach((k, v) -> System.out.print(k + ": " + v + "\t"));
            System.out.println();
        }
    }
}
```

IsInstance and Cast Methods

In a standard situation, type check and class casting is performed with *instanceOf* and the *explicit cast* operator (round brackets):

```
public void feed(List<Animal> animals) {
    animals.forEach(animal -> {
        animal.eat();
        if (animal instanceof Cat) {
            ((Cat) animal).meow();
        }
    });
}
```

IsInstance and Cast Methods

However there's an alternative way to perform a type check and cast, by using the *isInstance()* and *cast()* methods provided by the Class class (remember Reflection):

```
public void alternativeCast() {
    Animal animal = new Cat();

    // ...

    if (Cat.class.isInstance(animal)) {
        Cat cat = Cat.class.cast(animal);
        cat.meow();
    }
}
```

IsInstance and Cast Methods

This alternative approach is the commonly used solution in generic types because the specific type is unknown.

Note, that the Class instance should also be passed to the generic class as we can't get it from the type parameter T.

```java
public class AnimalFeederGeneric<T> {
    private Class<T> type;

    public AnimalFeederGeneric(Class<T> type) {
        this.type = type;
    }

    public List<T> feed(List<Animal> animals) {
        List<T> list = new ArrayList<T>();
        animals.forEach(animal -> {
            if (type.isInstance(animal)) {
                T objAsType = type.cast(animal);
                list.add(objAsType);

                // ...
            }
        });
        return list;
    }
}
```

```
List<Animal> animals = new ArrayList<>();
animals.add(new Cat());
animals.add(new Dog());
AnimalFeederGeneric<Cat> catFeeder
        = new AnimalFeederGeneric<Cat>(Cat.class);
List<Cat> fedAnimals = catFeeder.feed(animals);
```

Restrictions of Generics

Generics have some restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or `instanceof` With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

... for more information:

https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

Generic Methods

Java generics are mainly used to define generic data types (classes and interfaces).

But in Java, it is also possible to parameterize methods.

Usually, generic methods are used to express dependencies between parameter types and method return type.

Generics can be applied to static methods, instance methods as well as to constructors.

Generic Methods

The list of formal type parameters has to be provided within angle brackets “<” and “>” before the return type. They can then be used as the type for parameters, the method return type as well as in the body of the method.

```
public static <T> T addAndReturn(T element, Collection<T> collection) {
    T curElement = element;

    // ...

    collection.add(curElement);
    return curElement;
}
```

Generic Methods

```
String stringElement = "stringElement";
List<String> stringList = new ArrayList<String>();

String theString = addAndReturn(stringElement, stringList);

Integer integerElement = new Integer(123);
List<Integer> integerList = new ArrayList<Integer>();

Integer theInteger = addAndReturn(integerElement, integerList);
```

Generic Methods

Generic methods are invoked in the same way as non-generic methods.

The actual type argument is usually not specified. It's type is inferred by the compiler from the context use.

However, if needed, it can be explicitly specified:

```
String theString =
    TestClass.<String>addAndReturn(stringElement, stringList);
```

When inferred, the selected type is the most specific one in common to all the considered actual arguments.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
   // ...
}

    Object[] ao = new Object[100];
    Collection<Object> co = new ArrayList<Object>();
    fromArrayToCollection(ao, co); // T inferred to be Object
    String[] as = new String[100];
    Collection<String> cs = new ArrayList<String>();
    fromArrayToCollection(as, cs); // T inferred to be String
    fromArrayToCollection(as, co); // T inferred to be Object
    Integer[] ai = new Integer[100];
    Float[] af = new Float[100];
    Number[] an = new Number[100];
    Collection<Number> cn = new ArrayList<Number>();
    fromArrayToCollection(ai, cn); // T inferred to be Number
    fromArrayToCollection(af, cn); // T inferred to be Number
    fromArrayToCollection(an, cn); // T inferred to be Number
    fromArrayToCollection(an, co); // T inferred to be Object
    fromArrayToCollection(an, cs); // compile-time error
```

Summary

- Advantages and disadvantages of strong typing
- The problem of explicit casts
- Introduction to Generics and their advantages
- Generic types
- Formal type parameters and actual type arguments
- RAW types
- Generics vs. polymorphism
- Use of generics in the Java library
- Generic types and arrays
- Generic types and class cast
- Generic Methods