11 - Java Generics

It is the parametric polymorphism explicit bounded (invariant) of Java.

They enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. They provide:

- Stronger type checks at compile time;
- Elimination of casts;
- Possibility to implement generic algorithms;

Generic Types

A generic type is a generic class or interface that is parameterized over types. Example:

```java
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

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

The section delimited by angle brackets (<>) defines the type parameters, which can be used arbitrarily in the definition.

When referencing the generic `Box` class, we perform a generic type invocation, which replaces `T` with some concrete value, such as `Integer`:

```java
Box<Integer> integerBox;
```

In this example, `Integer` is called type argument, which can be any object, but not primitive types.

Every type parameter must be instantiated, either explicitly or implicitly (like a form of type inference).

Generic Methods

Similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Example:

```java
public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
```

````
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}

public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}```

The generic method in `Util` can then be invoked either by explicitly passing the parameters or by
letting the compiler infer the types:
```java
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");

// Explicit
boolean same = Util.<Integer, String>compare(p1, p2);

// Type-inferred
boolean same = Util.compare(p1, p2);
````

Bounded Type Parameters

Bounded type parameters are type parameters with some restrictions on which type argument can be passed. In particular, you can enforce upper/lower bounds:

- Upper bound: `<TypeVar> extends SuperType`
  - `SuperType` and any of its subtype are ok;
- Lower bound: `<TypeVar> super SubType`
  - `SubType` and any of its supertype are ok;

Example:

language-java

```java
class NumList<E extends Number> {
        void m(E arg){
                arg.intValue();
        }
}
```

When we want to instantiate a `NumList` object we have to use as type parameter a class that implements `Number`. This way we can invoke methods that are surely defined (since inherited).

Type Erasure

Type erasure is a process done by the Java compiler:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods;
- Insert type casts if necessary to preserve type safety;
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types, thus at runtime all the instances of the same generic type have the same type. Example:
```

```java
List<Integer> intList = new ArrayList<Integer>();
List<String> strList = new ArrayList<String>();

intList.getClass() == strList.getClass() // True
```
language-java

This means that generics do not introduce runtime overhead (the only overhead is caused by castings introduced at runtime, which are negligible). Also, type erasure ensure backward compatibility with legacy code.

Invariance, Covariance and Contravariance

In Java we know that `Integer` is a subtype `Number`, but *it is not true* that `List<Integer>` is a subtype of `List<Number>`.

By abstracting it, we can say that given two concrete types `A` and `B`, `Class<A>` has no relationship with `Class<B>`, regardless of any relationship between `A` and `B`. Formally, we can say that subtyping in Java is invariant for generic classes: the relationship between `A` and `B` do not influence the relationship between `Class<A>` and `Class<B>`.

On the other hand, as expected, if `A` extends `B` and they are generic classes then for each type `C` we have that `A<C>` extends `B<C>`. E.g.: `ArrayList<Integer>` *is a subtype* of `List<Integer>`.

NOTE: One of the common parents of `Class<A>` and `Class<B>` is `Class<?>`, where ? is called [wildcard](#).

Covariance and Contravariance

2 other notions are:

- Covariant: if `A` is a subclass of `B` then `Class<A>` is a subclass of `Class<B>`
- Contravariant: if `A` is a subclass of `B` then `Class<B>` is a subclass of `Class<A>`, i.e. there is an inversion of relation.

Why has Java decided to go for invariance rather than one of those? Let's say that we have the interface:

```java
interface List<T> {
        boolean add(T elt);
        T get(int index);
}
```
language-java

Hence, when instantiated with `Number` and `Integer` we have:

- `List<Number>`:
  - `boolean add(Number elt)`;
  - `Number get(int index)`;
- `List<Integer>`:
  - `boolean add(Integer elt)`;
  - `Integer get(int index)`;

Consider now the following code:

```java
List<Integer> intList = new ArrayList<>();
List<Number> numList - new ArrayList<>();

// Statement 1
numList = intList;

// Statement 2
numList.add(new Number(...));
```
language-java

```java
// Statement 3
intList = numList;

// Statement 4
Integer n = intList.get(0);
```

If covariance was allowed:

- Statement 1 would be possible, since `Integer` is a subtype of `Number`;
- Statement 2 however, would throw an error, as it is trying to add a `Number` to a list of `Integer`, but `Number` is not a subtype of `Integer`. On dynamic dispatching (selection of which polymorphic form to use at runtime) of `numList`, it would be invoked the `add` method of `List<Integer>`, and a type error would be raised;

If contravariance was allowed:

- Statement 3 would be possible, since `Integer` is a subtype of `Number` and thus `List<Number>` is a subtype of `List<Integer>`;
- Statement 4 however, would throw an error, as `intList.get(0)` returns a `Number`, but we expect an `Integer`, which is one of its subtypes.

These examples shows that the substitution principle ([Polymorphism > Inclusion]) is violated by both covariance and contravariance: `List<Number>` is neither a *supertype* nor a *subtype* of `List<Integer>`. Hence, Java only use invariance.

There are some cases however, where covariance and contravariance would be safe to allow:

- Covariance is safe if the type is read-only;
- Contravariance is safe if the type is write-only;

Indeed, some programming languages allows them:

- In C#, the type parameter of a generic class can be annotated out (covariant) or in (contravariant);
- In Scala, the type parameter of a generic class can be annotated - (covariant) or + (contravariant);

Java Arrays

In Java, if `Type1` is a subtype of `Type2`, then `Type1[]` is a subtype of `Type2[]`. Thus, Java arrays are covariant. This is done because Java Arrays are widely used, and it was needed a simple way to use them.

Consider the method: `void sort(Object[] array)`.

Since `Object[]` is the supertype of any other array, it exploit the covariance. Without covariance, we would need to define a new method `sort` for every array! But since sorting only make use of read-only operations, covariance is safe to be used in this case.

Problems with Array Covariance

Even if the array covariance works for sort it may still cause type errors in other cases. Consider:

```language-java
// Defined classes Fruit, Apple, Pear (with Apple and Pear subtypes of Fruit)

Apple[] apples = new Apple[1];
Fruit[] fruits = apples; // ok by covariance

fruits[0] = new Pear();
```

This code breaks a general Java rule: for every object, its dynamic type (type of the value of the object, known at runtime) must be a subtype of its static type (type given in the declaration). But in the example, `Pear` is not a subtype of `Apple`: every array update includes a runtime check, and assigning to an array element an object of a non compatible type throws a `ArrayStoreException`.

This scenario also shows why Java do not support generic arrays: the dynamic type of an array is known only at runtime, but the generics at runtime are casted to `Object` due to type erasure (and thus, it would not be possible to perform runtime checks). This would result in non-detectable errors. Since Java is type-safe, generics array are not supported.

Wildcard "?"

The wildcard represents an unknown type, and it is written as a question mark (?). It can be used as:

- Type of a parameter, field, or local variable;
- Return type (though it is better programming practice to be more specific);

We will see that they become handy in 2 particular scenarios:

- When a type is used exactly once and the name is unknown;
- For use-site variance: they provide on-spot covariance/contravariance.

When should Wildcards be used?

Follow the "PECS" Principle (Producer Extends, Consumer Super):

- Use `? extends T` when you want to get values (from a producer): supports covariance;
- Use `? super T` when you want to insert values (in a consumer): support contravariance;
- Do not use `?` (`T` is enough) when you both obtain and produce values.

Wildcards for Covariance

Invariance of generic classes is correct, but it limits code reusage. Wildcards can alleviate the problem.

Consider the following code:

```java
interface Set<E>{
        //adds to this all the elements in c (discarding duplicates)
        void addAll(??? c);
}
```

What is a "general enough" type for the method `addAll`?

- `void addAll(Set<E> c)`: ok, but what if I want to add to the set all the elements of a list `List<E> c`, or what if I want to add all the elements of a `Queue<E>`?
- `void addAll(Collection<E> c)`: better solution since `List`, `Queue` ... are all subtypes of `Collection`. The problem is that it do not take into account the elements accepted by covariance. The elements of a `List<T>` with `T` subtype of `E` should be allowed to be added to the set
- best solution: `void addAll(Collection<? extends E> c`
    - match any collection of elements of type subtype of `E`

Wildcards for Contravariance

Consider the following code (most general copy method):

```language-java
<T> void copy(List<? super T> dst, List<? extends T> src);
```

We exploit covariance for `src` and contravariance for `dst`.

The Price of Wildcards

A wildcard type is anonymous/unknown, and it reduce the possibility of reading/writing of objects due the restriction of covariance and contravariance.

Consider the following code:

```language-java
List<Apple> apples = new List<Apples>();
List<? extends Fruit> fruits = apples; // Covariance

fruits.add(new Pear());  // A
Fruit f = fruits.get(0); // B
fruits.add(new Apple()); // C
fruits.add(null);        // D
```

- A: compile-time error:
  - `? extends Fruit` match every subclass of `Fruit` (and thus, it doesn't know its static type), and being covariance the compiler won't allow the modification of `fruits`;
- B: ok:
  - Covariance, so read is ok: elements of `fruits` are surely subtype of `Fruit` and hence they can be assigned to `Fruit f`;
- C: compile-time error:
  - `? extends Fruit` match any subclass of `Fruit`, in this case is possible to infer that `Apple` is matched and it the statement would be correct
  - but what if there was an if statement where in the "then" branch we assigned `List<Apple>` to `fruits` and in the "else" branch we assigned `List<Pear>` to `fruits`?
  - also, can't add anything due to covariance constraints
- D: ok:
  - Lists that exploit covariance (as in this case) can't be modified, we can only add `null`;

The Price of Wildcard Contravariance

```language-java
List<Fruit> fruits = new ArrayList<Fruit>();
List<? super Apples> apples = fruits; //contravariance

apples.add(new Apple());     // A
apples.add(new FujiApple()); // B
apples.add(new Fruit());     // C
Fruit f = apples.get(0);     // D
Object o = apples.get(0);    // E
```

- The assignment `apples = fruits` is legit: `? super Apples` match every class from `Apple` to `Object` and in particular it matches `Fruit`:
- A: ok
  - we can add an `Apple` since it is surely subclass of a superclass of `Apple` and this check can be done statically
- B: ok
  - as for A
- C: compile-time error

- if `? super Apple` is `Apple` than we can't add a new `Fruit` since it is not a subclass of `Apple` (instead it is a superclass of `Apple`)
    - if `? super Apple` is `Fruit` or some higher object than the types are correct but this can't be known at static time (actually not even at runtime: in Java we have type erasure)
- D: compile-time error
    - while the types here are correct we see that `? super Apple` can match any type from `Apple` to `Objet`. In this case it matches `Fruit` but as before there could be some branching and what is matched would be known only at runtime (actually not even at runtime: in Java we have type erasure)
    - the only way where the types are surely allowed is to assign to the most general type of all, `Object`, as in E
    - mind that the compiler signal error no matter what since we are reading with contravariance
- E: ok
    - check D

Limitations of Java Generics

Mostly due to the concept of type erasure:

- Cannot instantiate generics types with primitive types:
    - `ArrayList<int>` do not compile: primitive types are not under `Object`, it would not be possible to perform the type erasure which is key to retro compatibility;
- Cannot create instances of type parameters:
    - In `List<T>` we can't do `new T()` since the `new` is done at runtime, and at runtime `T` is `Object`, we would only instantiate `Object`;
- Cannot declare static fields whose types are type parameter:
    - `public class C<T>{ public static T local; ... }`: do not know the type until instantiation;
- Cannot use `casts` or `instanceof` with parameterized types:
    - `list instanceof ArrayList<Integer>` do not compile;
    - `list instanceof ArrayList<?>` is ok;
- Cannot create arrays of parameterized types;
- Cannot `create`, `catch`, or `throw` objects of parameterized types (Type erasure + Run Time check);
- Cannot overload a method where the formal parameter types of each overload erase to the same raw type.