Modul- Fortgeschrittene Programmierkonzepte

05- Mixins and Generics (II)

Prof. Dr. Marcel Tilly

Bachelor Wirtschaftsinformatik, Fakultät für Informatik

Technische Hochschule Rosenheim

Mixins in Java

Last week, we showed how we can use default methods in interfaces to augment classes by certain functionality:

Use `default` methods!

```
public `interface` Escalated {
    String getText();
    `default` String escalated() {
        return getText.ToUpper();
    }
}
public class Message `implements Escalated` {
    private String m;
    public Message(String m) { this.m = m; }

    public String getText() {
        return m;
    }
}
Message m = new Message("Hello World");
System.out.println(`m.escalated()`);       // "HELLO, WORLD"
```

Stateful Mixins

- Conceptually, the difference between inheritance and mixins is that the latter are meaningless on its own and can be attached to other unrelated classes.
- On the other hand, inheritance is used if there is a strong relation between the classes.

The main issue with the above realization of mixins is the lack of *state*: since the iterfaces cannot have attributes, the only way to read/write data would be through (`public`) setters/getters.

Example: Let's say, you want to gradually escalate your shouting.

```
1 Message m = new Message("Hello, World");
2 m.escalate();  // "HELLO, WORLD"
3 m.escalate();  // "HELLO, WORLD!"
4 m.escalate();  // "HELLO, WORLD!!"
5 m.escalate();  // "HELLO, WORLD!!!"
```

Using State

This would require your `escalate` method to remember how often it was called, and add more bangs each time. Well fair enough, we'll use the same mechanis as for the `text`:

```
interface Escalatable {
    String text();   // to get the string
    int howOften();  // implementing class must handle counting!
    default String escalate() {
        int n = howOften();

        // n bangs in a row
        String bangs = Stream.generate(() -> "!")
                .limit(n)
                .reduce("", (a, b) -> a + b);

        return text + bangs;
    }
}
```

How does it work?

```
class Message implements Escalatable {
    private String t;
    Message(String t) { this.t = t; }

    public String text() { return t; }
    // counter
    private int n = 0;
    public int howOften() {
        return n++;
    }
}
```

```
class App {
    public static void main(String[] args) {
        Message m = new Message("Hello, World");
        m.escalate();  // "HELLO, WORLD"
        m.escalate();  // "HELLO, WORLD!"
        m.escalate();  // "HELLO, WORLD!!"
        m.escalate();  // "HELLO, WORLD!!!"
    }
}
```

But!

Unfortunately, we require the implementation (and logic) of the `howOften` in the *class* although it belongs to the *mixin*. What we need is a way for the mixin to store and retrieve its state with the object. This is where we can make use of inheritance and interfaces as well as generic methods.

First, we specify an interface `Stateful`, that specifies generic methods to store and retrieve the state. We use the `Class` object as key to store the state information and provide an `initial` value for the get method. The generic method allows us to avoid casts from `Object` to our actual state object.

```
1 interface Stateful {
2     <T> T getState(Class clazz, T initial);
3     <T> void setState(Class clazz, T state);
4 }
```

Stateful Object

Next, we create a `StatefulObject` that implements the `Stateful` interface, marking the methods `final` (since the mechanism is to be kept fixed).

```
1  class StatefulObject implements Stateful {
2      // note: we store the state for each mixin as Object!
3      private HashMap<Class, Object> states
4              = new HashMap<>();
5
6      public final <T> T getState(Class clazz, T initial) {
7          // cast necessary, since internally we store Object!
8          return (T) states.getOrDefault(clazz, initial);
9      }
10
11     public final <T> void setState(Class clazz, T s) {
12         states.put(clazz, s);
13     }
14 }
```

Extend the Mixin

For our mixin, we now `extend` the `Stateful` interface to access the state:

```
interface Escalatable extends Stateful {
    String text();

    default String escalated() {
        // generics magic!
        int n = getState(StatefulEscalate2.class, 0);
        setState(StatefulEscalate2.class, n+1);

        // generate n bangs, or empty strings for n=0
        String bangs = Stream.generate(() -> "!")
                .limit(n)
                .reduce("", (a, b) -> a + b);

        return text().toUpperCase() + bangs;
    }
}
```

Stateful-Class

For the actual class to attach the mixin to, we `extend StatefulObject`, since the mechanism to store and retrieve state is the same.

```
1 public class StatefulMessage
2         extends StatefulObject    // manages state
3         implements Escalatable {  // uses state
4     private String m;
5
6     public StatefulMessage(String m) { this.m = m; }
7
8     public String text() { return m; }
9 }
```

Finally

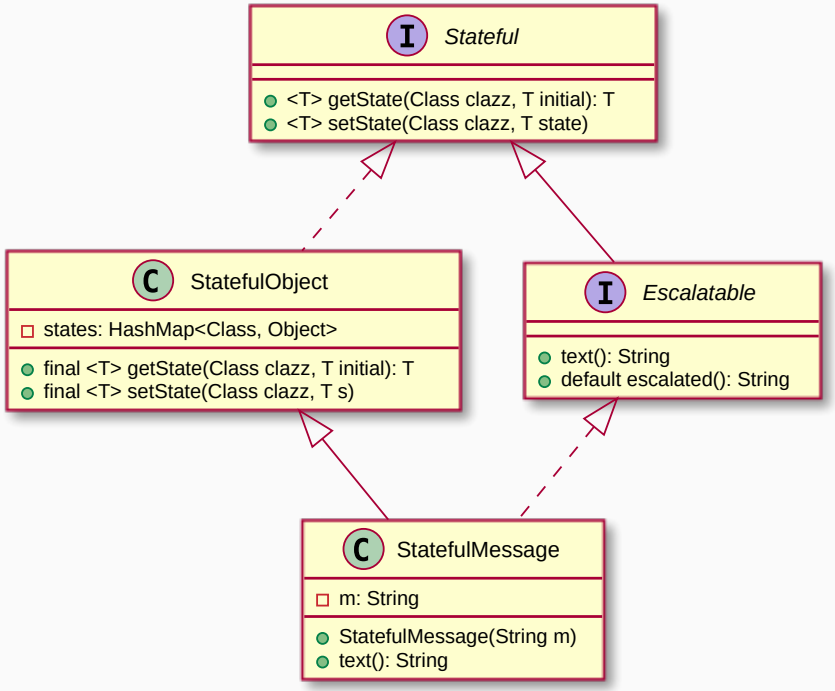
```
class App {
    public static void main(String[] args) {
        StatefulMessage m1 = new StatefulMessage("Hans");
        StatefulMessage m2 = new StatefulMessage("Dampf");

        System.out.println(m1.escalated());  // HANS
        System.out.println(m1.escalated());  // HANS!
        System.out.println(m1.escalated());  // HANS!!
        System.out.println(m2.escalated());  // DAMPF
        System.out.println(m2.escalated());  // DAMPF!
        System.out.println(m2.escalated());  // DAMPF!!
    }
}
```

UML-Diagram

How does it look as a *class diagram*?

Inheritance of classes (and interfaces) and generic methods, with relatively little impact on the overall class hierarchy:



Generics, part two.

- Generics and inheritance
- Bounds on type variables
- Wildcards
- Bounds on wildcards

Generics and Inheritance

Speaking of inheritance and generics. Recall a principal property of inheritance: an instance of a subclass (e.g. `java.lang.Integer`) can be assigned to a reference of the base class (e.g. `java.lang.Number`); the same holds for arrays:

```
Number n;
Integer i = 5;
n = i;  // since Integer extends Number

Number[] na;
Integer[] ia = {1, 2, 3, 4};
na = ia;  // ditto
```

Similarly, one would expect that the following works:

```
ArrayList<Number> as;
ArrayList<Integer> is = new ArrayList<>();
as = is;  // compiler error: incompatible types!
```

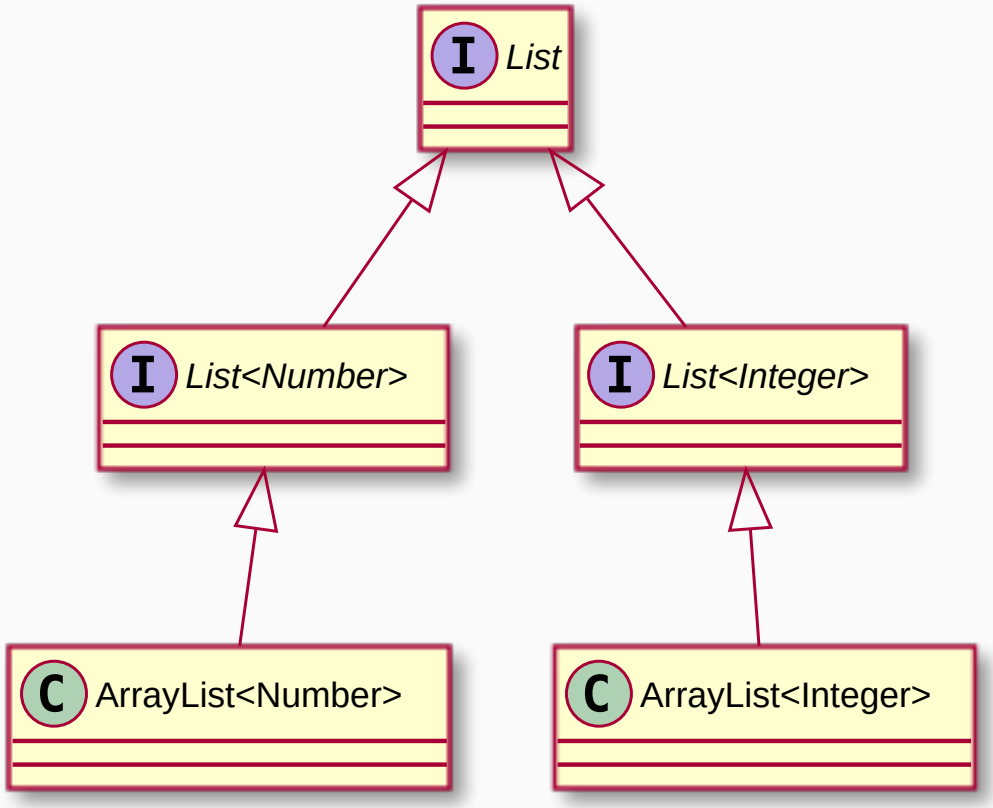
Compile Error, why?

It yields a compiler error, even if you try to type-cast it:
Incompatible types, required `ArrayList<Number>`, *found* `ArrayList<Integer>`.

In other words: two instances of the same generic classes are unrelated, even if their type arguments are related. The relation does however hold, if two generic classes are related and use the *same* type argument:

```
1 List<Integer> li;
2 ArrayList<Integer> al = new ArrayList<>();
3 li = al;  // ok, since ArrayList implements List!
```

Generics and Inheritance



What can happen?

Note that as a side effect of this relation, the following code compiles, but fails at runtime:

```
1  ArrayList rawL;  // raw type
2  ArrayList<Integer> intL = new ArrayList<>();
3  ArrayList<String> strL = new ArrayList<>();
4  // ok, since raw type is base (type erasure)
5  rawL = intL;
6  // compiler warning: unchecked assignment; raw to parameterized
7  strL = rawL;
8
9  intL.add(1337);
10 // exception: cannot cast Integer to String
11 System.out.println(strL.get(0));
```

Rules on Generics

The rules to remember are: 1. The type hierarchy works for generic classes if the type argument is the same, e.g. `List<Integer>` is super type of `ArrayList<Integer>`. 2. Types with different type arguments are not related, even if the type arguments are, e.g. `List<Number>` **is not** a super type of `ArrayList<Integer>`.

Read more about generics and inheritance in the Java docs.

Bounds on Type Arguments

- Using a linked list to store key-value pairs is inefficient
- A better way to organize the entries is to use a binary tree that stores the current key as well as links to subtrees with elements that are smaller (“left”) and larger (“right”).

```
1  public class SortedMapImpl<K, V> implements Map<K, V> {
2      class Entry {
3          public Entry(K key, V value) {
4              this.key = key;
5              this.value = value;
6          }
7          K key;
8          V value;
9          Entry left, right;  // two successors!
10     }
11     Entry root;
```

… cont'd

```
1    public void put(K key, V value) {
2        if (root == null) {
3            root = new Entry(key, value);
4            return;
5        }
6        Entry it = root;
7        while (it != null) {
8            // unchecked cast, runtime hazard: ClassCastException
9            int c = ((Comparable<K>) key).compareTo(it.key);
10           if (c == 0) {
11               it.value = value;
12               return;
13           } else if (c < 0) {
14               if (it.left == null) {
15                   it.left = new Entry(key, value);
16                   return;
17               } else { it = it.left;}
18           } else {
19               if (it.right == null) {
20                   it.right = new Entry(key, value);
21                   return;
22               } else {it = it.right;}
23           }
```

... cont'd

```
    public V get(K key) {
        Entry it = root;
        while (it != null) {
            // unchecked cast, runtime hazard: ClassCastException
            int c = ((Comparable<K>) key).compareTo(it.key);

            if (c == 0) return it.value;
            else if (c < 0) it = it.left;
            else it = it.right;
        }

        return null;
    }
}
```

About Binary Trees

- **Note**: This (unbalanced) binary tree has a worst case of O(n).
- Can you think of such a degenerate case?
- To make this implementation more efficient, use an AVL tree.

The explicit cast of the `K` type to a `Comparable<K>` results in a warning (*unchecked cast*) which can result in a `ClassCastException` at runtime.

Bounds

To enforce that a certain class is either a subclass or implements a certain interface, use the following syntax with `extends`:

```
class SortedMapImpl<K extends Comparable<K>, V>
    implements Map<K, V> {
    // ...
}
```

This has two effects:

- First, the type to be used for K is checked at compile time if it implements `Comparable<K>`.
- Second, since K implements the interface, you can call any method inherited from `Comparable` on a reference of K without an explicit cast.

Comparable

```
class SortedMapImpl<K extends Comparable<K>, V> implements Map<K, V> {
    // ...
    public V get(K key) {
        Entry it = root;
        while (it != null) {
            // no cast necessary!
            int c = key.compareTo(it.key);
            if (c == 0) return it.value;
            else if (c < 0) it = it.left;
            else it = it.right;
        }
    }
}
```

- To enforce more than one class or interface, use the `&` symbol (since the `,` is already reserved for multiple type arguments), for example:
  `<T extends Comparable<T> & Serializable>`.
- As you can see in the example above, you may set these *bounds* on type variables also when extending an interface or class.
- Read more on type bounds in the Java docs.

Wildcards and Bounds

Consider this routine that prints out all elements of a `java.util.Collection`.

```
void print(Collection c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

Using the raw type is not advised, so we change the signature to

```
void print(Collection<Object> c) {
    // ...
}
```

Wildcard: ?

- Which is not the supertype for all kinds of `Collection`s?
- What is the supertype of all `Collection`s?

It is a `Collection` with *unknown* type, which is denoted using the wildcard `?`:

```
void print(Collection<?> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

Wildcard as Bound

- inside `print()`, we can *read* the objects
- but we cannot add to the collection:

```
1 void print(Collection<?> c) {
2     for (Object o : c) {
3         System.out.println(o);
4     }
5     c.add(new Object());  // compile time error: type error
6 }
```

Generics ‘extends’

- Since we don’t know what the element type of `c` stands for, we cannot add objects to it.
- The `add()` method takes arguments of type the collection is bound to.
- Any parameter we pass to `add` would have to be a subtype of this now unknown (`?`) type.
- Since we don’t know what type that is, we cannot pass anything in.
- An exception is `null`, which is a member of every type.

Similar to type variables, wildcards can be bound.

```
class Klass {
    void method() { /* ... */ }
}
```

```
void apply(Collection<? extends KlassA> c) {
    for (Klass k : c) {
        k.method();
    }
}
```

Upper Bound

An *upper* bound, defining that the class is unknown, but *at least* satisfies a certain class or interface.

For example, `List<Integer>` fits as a `List<? extends Number>`.

What is the difference between a wildcard bound and a type parameter bound?

1. A wildcard can have only one bound, while a type parameter can have several bounds (using the & notation).
2. A wildcard can have a *lower* or an upper bound, while there is no such thing as a lower bound for a type parameter.

Lower Bounds

So what are *lower* bounds?

A lower bounded wildcard restricts the unknown type to be a specific type or a supertype of that type.

In the previous examples with *upper bounds*, we were able to *read* (`get()`) from a collection, but not *write* (`add()`) to a collection. If you want to be able to *write* to a collection, use a *lower* bound:

```
1 void augment(List<? super Klass> list) {
2     for (int i = 1; i <= 10; i++) {
3         list.add(new Klass());  // this works
4     }
5     // compile time error: can't resolve type
6     Klass k = list.iterator().next();
7     // runtime hazard: ClassCastException
8     Klass k = (Klass) list.iterator().next();
9 }
```

Good or Bad?

The drawback is, that we can't (safely) *read* from the collection anymore, since the compiler is unable to resolve the type to be used:

The actual instance could be `Klass` or any supertype (up to `Object`), thus a forced cast could lead to a `ClassCastException`.

This is where generics reach their limits: you can specify an upper bound for a wildcard, or you can specify a lower bound, but you *cannot specify both*.

Liskov Substitution

Definition: Liskov Substitution Principle:

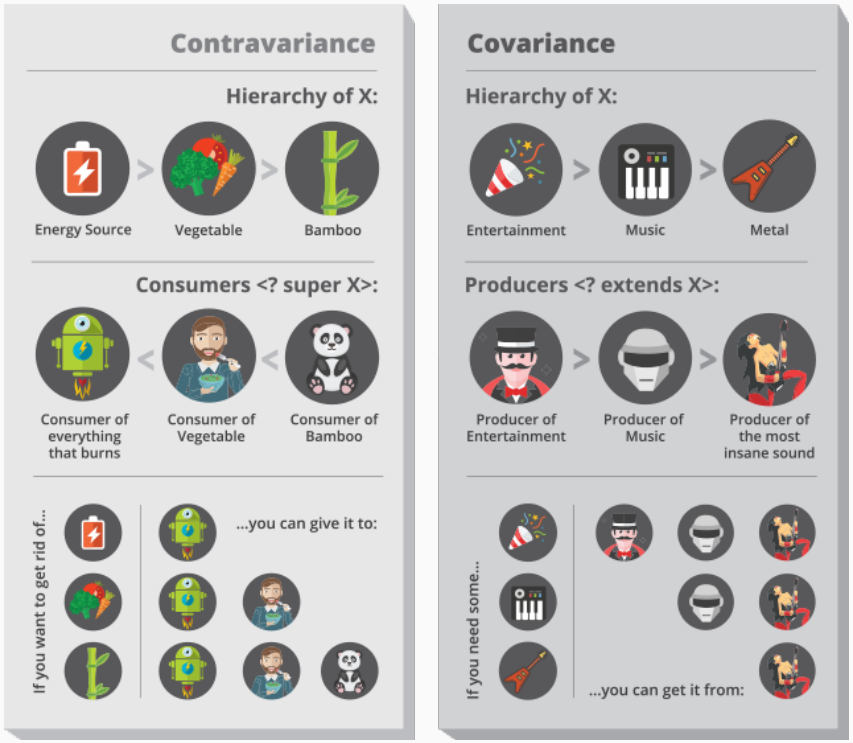
```
if S is a subtype of T, then objects of type T may be replaced with objects of
    type S.
```

Within the type system of a programming language, a typing rule

- **Covariant**: if it preserves the ordering of types (≤), which orders types from more specific to more generic
- **Contravariant**: if it reverses this ordering;
- **Invariant or nonvariant**: if neither of these applies.

PECS

Languages supporting generics (such as Java or Scala, and to some extent C++), feature *covariance* and *contravariance*, which are best described in the following diagram by Oleg Shelajev at RebelLabs (based on a diagram by Andrey Tyukin available under the CC-BY-SA).



PECS

Case 1: You want to go through the collection and do things with each item.

- The list is a **producer**, so you should use a `Collection<? extends Thing>`.
- The reasoning is that a `Collection<? extends Thing>` could hold any subtype of Thing, and thus each element will behave as a Thing when you perform your operation.
- You actually cannot add anything to a `Collection<? extends Thing>`, because you cannot know at runtime which specific subtype of Thing the collection holds.

Case 2: You want to add things to the collection.

- The list is a **consumer**, so you should use a `Collection<? super Thing>`.
- The reasoning here is that unlike `Collection<? extends Thing>`, `Collection<? super Thing>` can always hold a Thing no matter what the actual parameterized type is.
- Here you don't care what is already in the list as long as it will allow a Thing to be added; this is what `? super Thing` guarantees.

PECS

The combination of the two principles (contravariance and covariance) is known as *PECS* – *producer* `extends`, *consumer* `super`. The mnemonic is seen from the collection's point of view.

- If you are retrieving items from a generic collection, it is a producer and you should use `extends`.
- If you are adding items, it is a consumer and you should use `super`.
- If you do both with the same collection, you shouldn't use either `extends` or `super` (but a type variable, with bounds if needed).

The classic example for PECS is a function that reads from one collection and stores them in another, e.g. copy:

```
1 static <T> void copy(Collection<? extends T> source,
2     Collection<? super T> dest) {
3     for(T n : source) {
4         dest.add(n);
5     }
6 }
```

As you can see, it combines a type variable (T) with bounded wildcards to be as flexible as possible while maintaining type safety.

Here is another example, adapted from a stackoverflow post. Consider this function that adds a Number to a list of Numbers.

```
static <T extends Number> void includeIfEven(List<T> evens, T n) {
    if (n.intValue() % 2 == 0) {
        evens.add(n);
    }
}
```

```
List<Number> numbers = new LinkedList<>();
List<Integer> ints = new LinkedList<>();
List<Object> objects = new LinkedList<>();
includeIfEven(numbers, new Integer(4));  // OK, Integer extends Number
includeIfEven(numbers, new Double(4.0)); // OK, Double extends Number
includeIfEven(ints, new Double(4.0));    // type error!
includeIfEven(objects, new Integer(4));  // type error!
```

As you can see, if the bounds for the type variable (`extends Number`) is satisfied, the same type is used for both arguments. But the container would actually be more flexible, e.g. a `List<Object>` could also hold those numbers. This is where the bounds come in:

```
static <T extends Number> void includeIfEven(List<? super T> evens, T n) {
    // ...
}
```

By using the wildcard with a lower bound on `T`, we can now safely call

```
includeIfEven(objects, new Integer(4));
includeIfEven(objects, new Double(4.0));
```

Summary

- A wildcard can have only one (upper or lower) bound, while a type parameter can have several bounds (using the `&` operator).
- A wildcard can have either a lower or an upper bound, while a type variable can only have an upper bound.
- Wildcard bounds and type parameter bounds are often confused, because they are both called bounds and have in part similar syntax:
  - type parameter bound: T extends Class & Interface1 & ... & InterfaceN
  - wildcard bound: `? extends SuperType` (upper) or `? super SubType` (lower)
- A wildcard can have only one bound, either a lower or an upper bound.
- A list of wildcard bounds is not permitted.
- Type parameters, in contrast, can have several bounds, but there is no such thing as a lower bound for a type parameter.
- Use upper and lower bounds on wildcards to allow type safe reading and writing to collections.

More Information

Effective Java (2nd Edition), Item 28, summarizes what wildcards should be used for:

> *Use bounded wildcards to increase API flexibility. [...]*
>
> *For maximum flexibility, use wildcard types on input parameters that represent producers or consumers. [...]*
>
> ***Do not use wildcard types as return types.*** *Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code. Properly used, wildcard types are nearly invisible to users of a class. They cause methods to accept the parameters they should accept and reject those they should reject. If the user of the class has to think about wildcard types, there is probably something wrong with the class's API.*

Read more on wildcards in the Java docs.

Final Thought!