Technische Hochschule Rosenheim

Modul - Fortgeschrittene Programmierkonzepte

Bachelor Informatik

04 - Generics

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Agenda

Technische Hochschule Rosenheim

- Generic classes and interfaces
- Type erasure, raw type and instantiation
- Generic methods

Generic Classs/Interfaces

Technische Hochschule Rosenheim

The example data structure for this class will be a *map*. Unlike a *list* which stores data in a (fixed) sequential order, a *map* is an associative container that stores a certain *value* for a certain (unique) *key*.

Compare the basic interfaces:

```
interface List {
    void add(Object o);  // appends to the list
    Object get(int i);   // retrieves the i-th element
}
```

```
interface Map {
    void put(Object key, Object value);  // stores object for key
    Object get(Object key);              // retrieves object for key
}
```

Map

Technische Hochschule Rosenheim

There are many ways to implement a map, and they differ greatly in complexity. Here, let's consider a very basic implementation that consists of map entries (key and value) that are stored as a list.

Remember, *inner classes* are an excellent means to keep the class hierarchy neat and organized:

```
class SimpleMapImpl implements Map {
    private class Entry {
        Entry(Object key, Object value) {
            this.key = key;
            this.value = value;
        }
        Object key;
        Object value;
        Entry next;
    }
    // ...
}
```

Map-*put*

Technische Hochschule Rosenheim

- Use a `head` element and the `Entry.next` reference to build up the list.

```
class SimpleMapImpl implements Map {
    // ...
    private Entry head;
    @Override
    public void put(Object key, Object value) {
        if (head == null) {
            head = new Entry(key, value);  // easy: first Entry in
            return;
        }
        Entry it = head, prev = null;
        while (it!= null) {
            if (it.key.equals(key)) {  // key exists, update value
                it.value = value;
                return;
            }
            prev = it;
            it = it.next;
        }
        prev.next = new Entry(key, value);  // append at the end
    }
```

Map-*get*

Technische Hochschule Rosenheim

Simiarly, when `get`-ting an element, iterate from the `head` to the end, and return that `value` where the `key` matches, or `null`.

```
class SimpleMapImpl implements Map {
    // ...
    @Override
    public Object get(Object key) {
        Entry it = head;
        while (it != null) {
            if (it.key.equals(key))  // found it!
                return it.value;
            it = it.next;
        }

        return null;  // no value for this key
    }
}
```

Good or bad design?

Usage

Technische Hochschule Rosenheim

Here's how you would use it:

```
class App {
    public static void main(String... args) {
        Map map = new SimpleMapImpl();

        // the type conversion to Object is automatic
        map.put("Grummel Griesgram", 143212);
        map.put("Regina Regenbogen", 412341);

        // since the return type is Object,
        // explicit type conversion is required
        Integer grummel = (Integer) map.get("Grummel Griesgram");
        // > 143212
        Integer schleichmichl = (Integer) map.get("Schleichmichl");
        // > null
    }
}
```

What can happen?

Technische Hochschule Rosenheim

Prior to Java 1.5: via `Object` and explicit type casts.

```
Map map = new SimpleMapImpl();
map.put("Hans", 123);
map.put("Peter", "Pan");

Integer i1 = (Integer) map.get("Hans");   // forced type cast
Integer i2 = (Integer) map.get("Peter");  // ClassCastException!
```

- Implementation is not type safe!
- Problems can occur during *runtime*!

How can we solve this?

Runtime vs. Compile Time

Technische Hochschule Rosenheim

If you run it, you will get a *runtime* error:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String canno
    at SimpleStringIntMap.get(SimpleStringIntMapImpl.java:14)
    at App.main(App.java:24)
```

> Note: Even worse, if you were to use a `key` different from `String`, the `get` call would bind to the super class' `get(Object)` method. Use the `@Override` annotation to have the compiler warn you of unintentional overloading instead of overwriting.

Generic Classes/Interfaces

Technische Hochschule Rosenheim

```
class SimpleStringIntMapImpl extends SimpleMapImpl {
    public Integer get(String key) {
        Object val = super.get(key);
        if (val == null)
            return null;
        else
            return (Integer) val;
    }
}
```

Is this a *good* or a *bad* solution?

Generic Classes (Interfaces) to the Rescue

Technische Hochschule Rosenheim

> Note: The following works equally for classes and interfaces.

On top of the runtime issues, one would have to `extend` for each key-value type combination to be used, i.e. the `Map` should be literally generic. Clearly, this is an all but ideal situation which can be fixed using Java *generics* (introduced in Java 1.5).

Generics

Technische Hochschule Rosenheim

Instead of using `Object` along with type casts, use type parameters (type variables) as placeholder for actual types, i.e. instead of `Object`, use `T`. The type parameters need to be declared in the signature of the class, in `<...>` and between the name and the opening curly parenthesis:

```
interface Map<K, V> {

    void put(K key, V value);

    V get(K key);
}
```

While you could use any identifier for the type parameters, it is customary to use single letters. Use `T` for a single type, `K` and `V` for key and value, `R` and `S` for unrelated types (see towards the end of this class). If you need to use multiple type parameters, separate them with comma.

1. Examples

Technische Hochschule Rosenheim

When implementing or extending a generic interface or class, you may either set actual types for the parameters ...

```
// (1) define actual types: all type parameters bound
class SimpleStringIntMapImpl implements Map<String, Integer> {

    public void put(String key, Integer value) {
        // ...
    }

    // ...
}
```

2. Example

Technische Hochschule Rosenheim

... carry over the parameter list ...

```
// (2) carry over type list: still two type parameters
class SimpleMapImpl<K, V> implements Map<K, V> {

    public void put(K key, V value) {
        // ...
    }

    // ...
}
```

3. Example

Technische Hochschule Rosenheim

... , or a mix of the two.

```
// (3) partially carry over; here: one type parameter remains
class SimpleStringMapImpl<V> implements Map<String, V> {

    public void put(String key, V value) {
        // ...
    }

    // ...
}
```

Map- Generics

Technische Hochschule Rosenheim

```
public class SimpleMapImpl<K, V> implements Map<K, V> {

    class Entry {
        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
        K key;
        V value;
        Entry next;
    }

    Entry head;
    @Override
    public void put(K key, V value) {
        // ...
    }
    @Override
    public V get(K key) {
        // ...
    }
}
```

Usage

Technische Hochschule Rosenheim

```
class App {
    public static void main(String[] args) {

        // note: type inferred!
        Map<String, Integer> map = new SimpleMapImpl<>();
        map.put("Hans", 14235);
        Integer i = map.get("Hans");  // > 14235

        map.put("Peter", "Willi");  // compile time error!
    }
}
```

As you can see, `SimpleMapImpl` now features *type safety*: instead of failing at runtime with a `ClassCastException`, it now fails at compile time with a type error.

Generics and Static

Technische Hochschule Rosenheim

Note that the inner class was declared non-static:

```
public class SimpleMapImpl<K, V> implements Map<K, V> {

    class Entry {

        public Entry(K key, V value) { /* ... */ }
        Entry next;
        // ...
    }

    // ...
}
```

Generics and Static

Technische Hochschule Rosenheim

Accordingly, we can use anything of the enclosing instance, including the type arguments (since it can only exist with the outer instance). If you were to use a static inner class, you need to make it generic as well:

```
public class SimpleMapImpl<K, V> implements Map<K, V> {

    static class Entry<K, V> {

        public Entry(K key, V value) { /* ... */ }
        Entry<K, V> next;
        // ...
    }

    // ...
}
```

> Note: Here, the static inner class uses the same names `<K, V>`; this is arbitrary, they could also be named `<Y, Z>`. Since the class is static, `K` and `V` of the outer class are not visible.

Type Erasure and Raw Type

Technische Hochschule Rosenheim

Internally, the Java compiler actually removes the generic parameters after compilation. If you look at the compiled classes, you will find

```
- Map.class
- SimpleMapImpl$Entry.class
- SimpleMapImpl.class
```

The type validation is completely done at **compile time** by replacing the type parameters with `Object`.

On completion, there is no need to retain the generic parameters, and only the "basic" .class file is stored. This has three side effects.

- You can't distinguish between types based their type parameters.
- There is no (direct) way to instantiate an *array* of generic elements.
- You can instantiate the so-called *raw type* of a generic class.

What works and what not!

Technische Hochschule Rosenheim

```
class Example<T> {
    private T inst = new T();           // compiler error!
    private T[] wontWork = new T [10];  // compiler error!

    public static void main(String... args) {
        Example e0 = new Example();  // raw type; effectively T := Object
        Example<String> e1 = new Example<>();   // T := String
        Example<Integer> e2 = new Example<>();  // T := Integer

        System.out.println(e0.getClass() == e1.getClass());  // > true!
        System.out.println(e1.getClass() == e2.getClass());  // > true!
    }
}
```

If you must instantiate from a generic type, you need to pass in the type parameter information ("the class") at runtime, and the type must have a default constructor.

Under the hood ...

Technische Hochschule Rosenheim

This is done using Java's reflection mechanism (which will be covered next week!). The runtime type information is stored in the `.class` attribute of a class or interface; it is of type `Class<T>` where the `T` is bound to the type itself. We can exploit that to generate instances of generic types at runtime:

```
class Example<T> {
    private T inst;     // definition ok-- no new yet!
    private T[] array;

    Example(Class<T> clazz)
            throws IllegalAccessException, InstantiationException {
        inst = clazz.newInstance();  // uses default constructor
        array = (T[]) Array.newInstance(clazz, 5);  // size of 5
    }

    public static void main(String... args) {
        // pass in actual type!
        Example<String> e = new Example<>(String.class);
    }
}
```

Generic Methods

Technische Hochschule Rosenheim

Similar to classes and interfaces, type parameters can be attached to methods to make them generic:

```
class Example {
    static Object[] reverse(Object[] arr) {
        Object[] clone = arr.clone();
        for (int i = 0; i < arr.length/2; i++)
            swap(clone, i, arr.length - 1 - i);
        return clone;
    }
    private static void swap(Object[] arr, int i, int j) {
        Object h = arr[i];
        arr[i] = arr[j];
        arr[j] = h;
    }
    public static void main(String... args) {
        Integer[] arr = {1, 2, 3, 4, 5};
        Integer[] rev = (Integer[]) reverse(arr);  // explicit type cast
        // will produce ClassCastExecption at runtime!
        Integer[] oha = (Integer[]) reverse(
            new String[] {"Hans", "Dampf"});
    }
}
```

Generic Methods

Technische Hochschule Rosenheim

For methods, the type parameters are specified prior to the return type, and type parameters can be used both for arguments and return types.

```
class Example {
    static <T> T[] reverse(T[] in) {
        T[] clone = in.clone();
        for (int i = 0; i < in.length/2; i++)
            swap(clone, i, in.length - 1 - i);
        return clone;
    }
    private static <T> void swap(T[] arr, int i, int j) {
        T h = arr[i];
        arr[i] = arr[j];
        arr[j] = h;
    }
    public static void main(String... args) {
        Integer[] arr = {1, 2, 3, 4, 5};
        Integer[] rev = reverse(arr);  // type safety at compile time!
        // will produce error at compile time! (Integer[] and String[] incompatible)
        Integer[] oha = (Integer[]) reverse(
            new String[] {"Hans", "Dampf"});
    }
}
```

Generics, part two.

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

Generics and Inheritance

Technische Hochschule Rosenheim

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;  // what happens here?
```

Compile Error, why?

Technische Hochschule Rosenheim

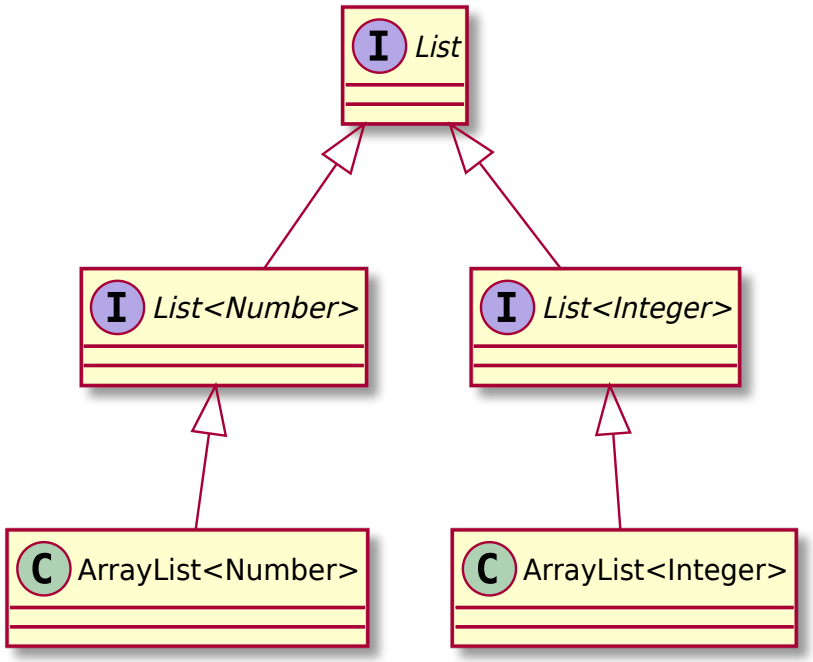
It yields a compiler error, even if you try to type-cast it:

> 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:

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

Generics and Inheritance



What can happen?

Technische Hochschule Rosenheim

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

```
ArrayList rawL;  // raw type
ArrayList<Integer> intL = new ArrayList<>();
ArrayList<String> strL = new ArrayList<>();

// ok, since raw type is base (type erasure)
rawL = intL;

// compiler warning: unchecked assignment; raw to parameterized
strL = rawL;

intL.add(1337);
// exception: cannot cast Integer to String
System.out.println(strL.get(0));
```

Rules on Generics

Technische Hochschule Rosenheim

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

Technische Hochschule Rosenheim

- 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").

```
public class SortedMapImpl<K, V> implements Map<K, V> {

    class Entry {
        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
        K key;
        V value;
        Entry left, right;  // two successors!
    }

    Entry root;
```

... cont'd

Technische Hochschule Rosenheim

```
public void put(K key, V value) {
    if (root == null) {
        root = new Entry(key, value);
        return;
    }
    Entry it = root;
    while (it != null) {
        // unchecked cast, runtime hazard: ClassCastException
        int c = ((Comparable<K>) key).compareTo(it.key);
        if (c == 0) {
            it.value = value;
            return;
        } else if (c < 0) {
            if (it.left == null) {
                it.left = new Entry(key, value);
                return;
            } else { it = it.left;}
        } else {
            if (it.right == null) {
                it.right = new Entry(key, value);
                return;
            } else {it = it.right;}
        }
```

... cont'd

Technische Hochschule Rosenheim

```
    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

Technische Hochschule Rosenheim

- **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

Technische Hochschule Rosenheim

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

Technische Hochschule Rosenheim

```
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;
        }
    }
}
```

- 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

Technische Hochschule Rosenheim

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: ?

Technische Hochschule Rosenheim

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

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

Technische Hochschule Rosenheim

- inside `print()`, we can *read* the objects
- but it is unspecific, we can call only methods on *Object*:

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

Generics 'extends'

Technische Hochschule Rosenheim

- Since we don't know what the element type of `c` stands for, we cannot do anyting specific.
- Let's assume we want to call a specific *method*.

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

Technische Hochschule Rosenheim

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

Technische Hochschule Rosenheim

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:

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

Multiple Bounds

Technische Hochschule Rosenheim

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Liskov Substitution

Technische Hochschule Rosenheim

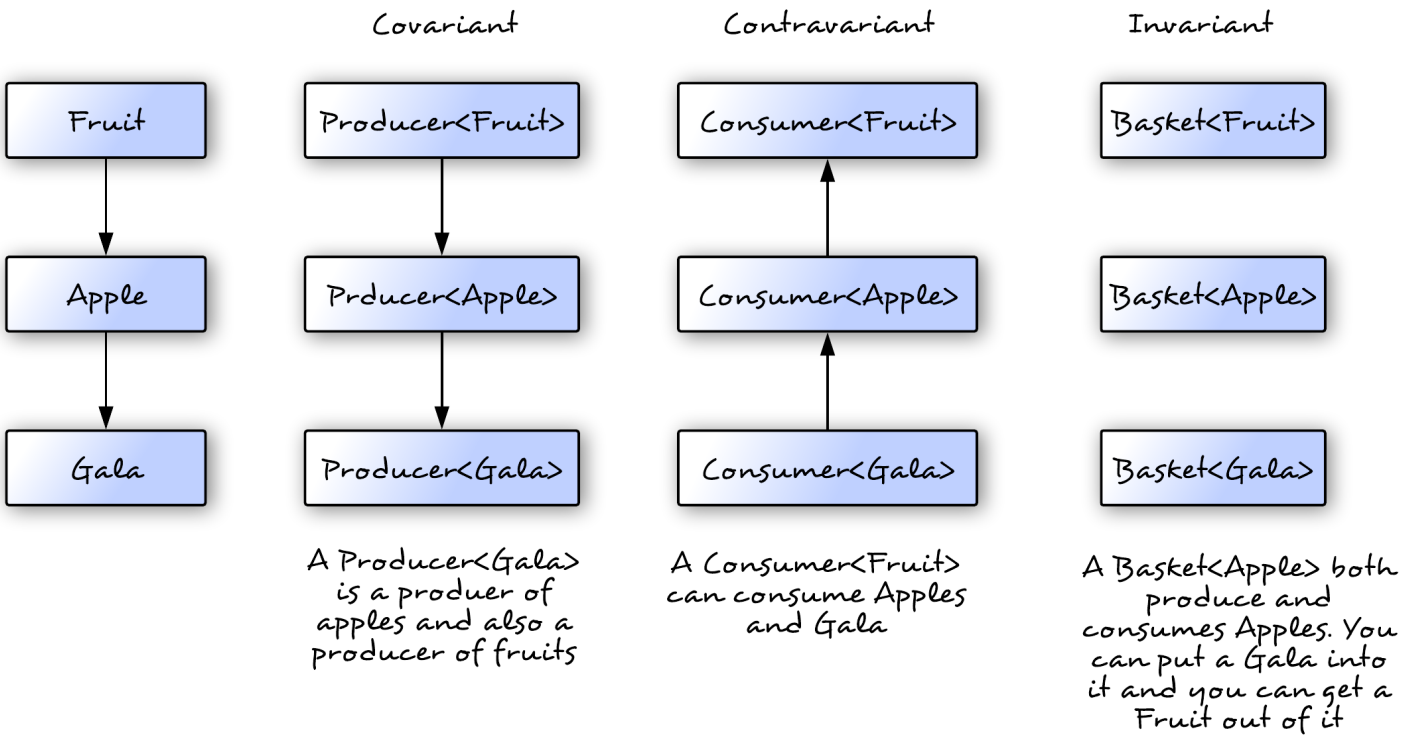
Definition: Liskov Substitution Principle:

```
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.

Covariance vs Contravariance



taken from https://dzone.com/articles/variance-immutability-and-strictness

PECS

Technische Hochschule Rosenheim

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.

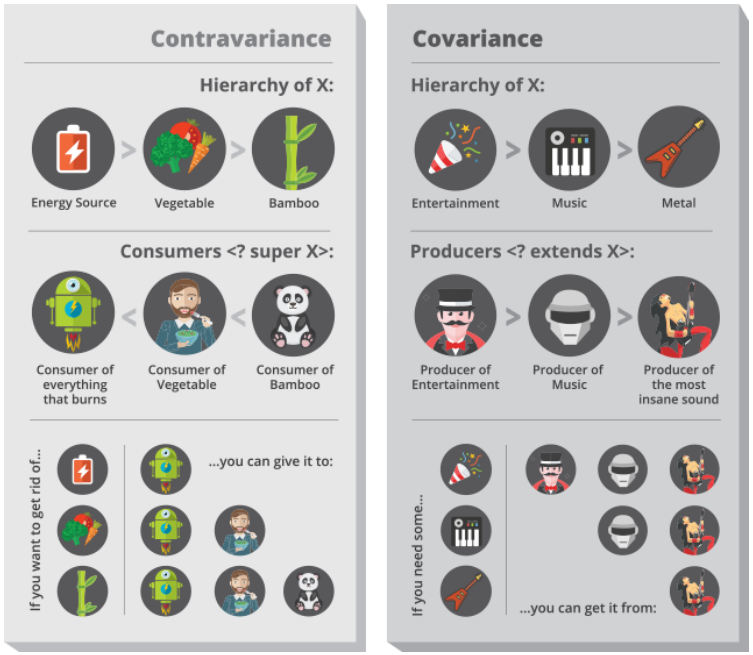
- If you are retrieving items from a generic collection, it is a producer and you should use `extends`.
- If you are adding items, it is a consumer and you should use `super`.
- If you do both with the same collection, you shouldn't use either `extends` or `super` (but a type variable, with bounds if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

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

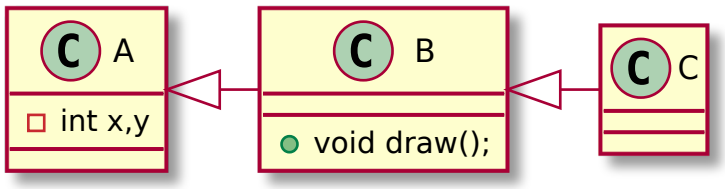
```
static <T> void copy(Collection<? extends T> producer,  Collection<? super T> consumer) {

    for(T n : producer) {
        consumer.add(n);
    }
}
```

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 - Example




```
    static void listRead(List<? extends A> l) {
        for (A a:l) {
            System.out.println(a);
        }
    }

    static void listWrite(List<? super A> l, A value) {
        l.add(value);
    }
...
        List<A> l = new ArrayList<>();
        l.add(new B("B"));
        l.add(new D("D"));
        l.add(new A("A"));

        listRead(l);
        listWrite(l, new C("C"));
```

PECS

Technische Hochschule Rosenheim

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

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!
```

PECS

Technische Hochschule Rosenheim

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));
```

Wildcards in short

Technische Hochschule Rosenheim

- 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

Technische Hochschule Rosenheim

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.

Summary

Technische Hochschule Rosenheim

Lessons today...

- Map- Implementation
- Generics
  - Compile time vs. Runtime
  - Classes
  - Methods
  - Bounds
  - PECS
  - Covariant vs Contravariant

Final Thought!