



UNIVERSITY
OF COLOGNE

Software & System Engineering / Software Technology
Abteilung Informatik, Department Mathematik / Informatik

Object-Oriented Software Engineering

Modular Programming and Generic Programming

Adrian Bajraktari | bajraktari@cs.uni-koeln.de | SoSe 2024 | 29.04.2024



Modularization and Namespaces

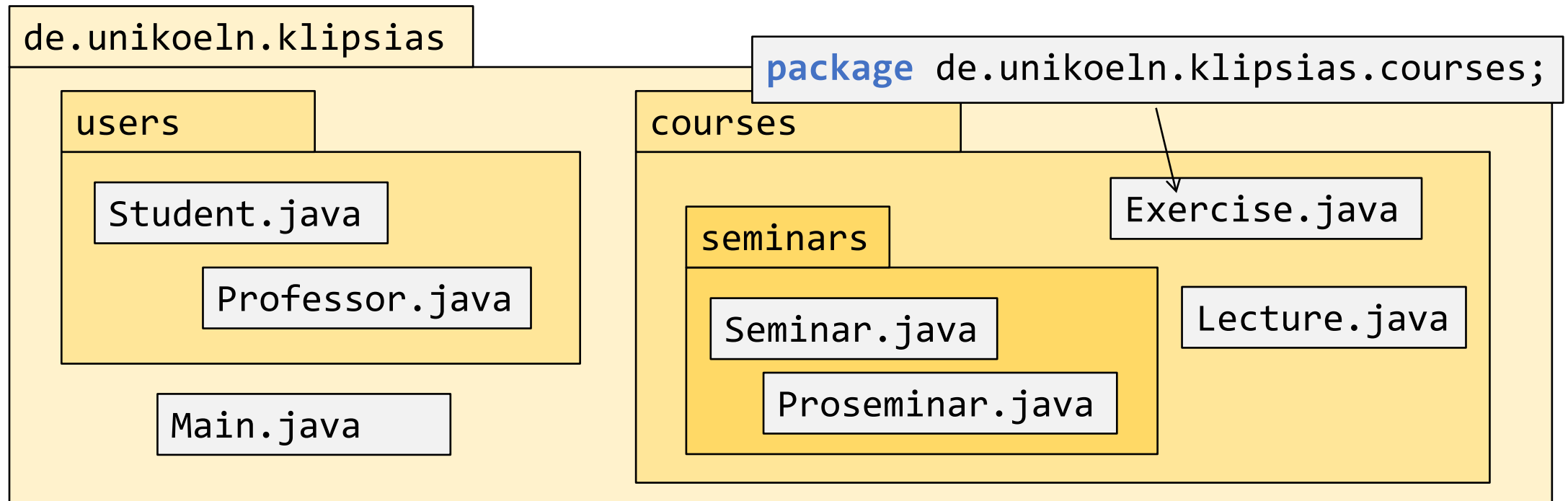
Modularization

Monoliths

Non-trivial software projects quickly become very big. Instead of putting all code in one file, or all files in one directory, developers should use [modularization](#).

Modularization

Many languages define different means to organize code in different levels of modules.



Namespaces and Packages

Namespace

Namespaces are scopes in which the name of a class is known. Classes in different namespaces can have the same names.

Name Qualifier

Classes can be used either with their

- fully qualified name, .e.g,
data.users.Student,
- unqualified name, e.g., Student.

Import

To use classes via unqualified names that lie in different packages, we must import the class.

Heuristic

- Rather import than qualify.
- Never put classes in default package.
- Never import unnecessary things, e.g., via *

Package

In Java, classes lie in **packages**. Packages define namespaces. Packages correspond to paths in the source file system.

/users/Student.java

```
package users;  
public class Student {  
    ...  
}
```

/data/users/Student.java

```
package data.users;  
public class Student {  
    ...  
}
```

Module

Package

Class



Header Dateien

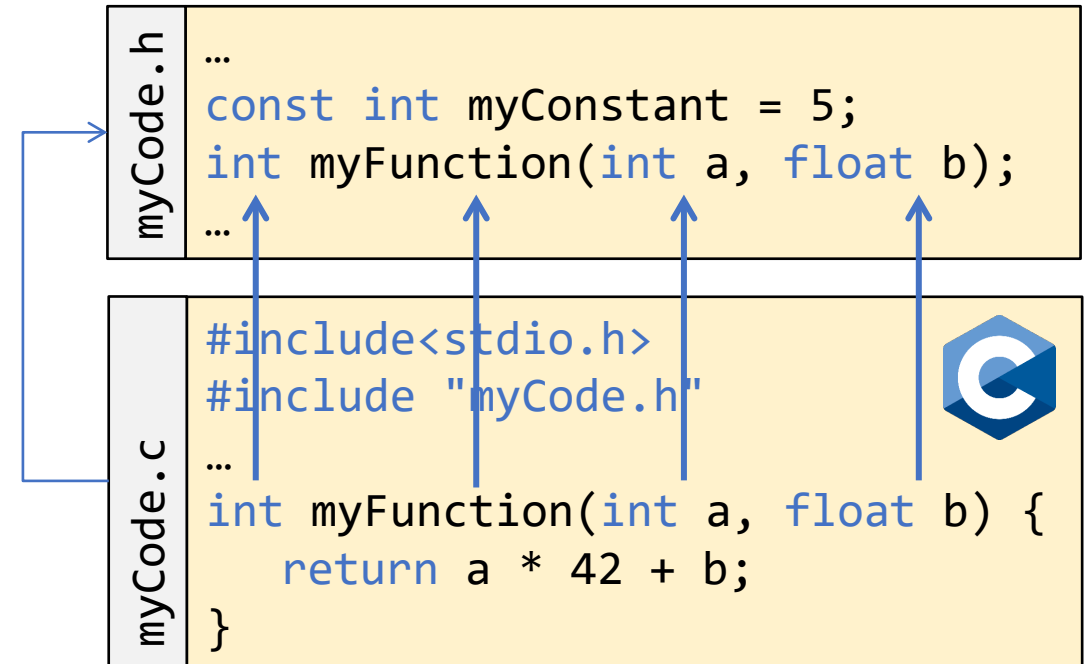
Headers in C/C++

C and C++, in contrast to most modern languages, split declaration and definition of procedures (as procedure prototypes), structs, global constants, etc. in **headers**.

The header file is included in the source file (.c / .cpp).

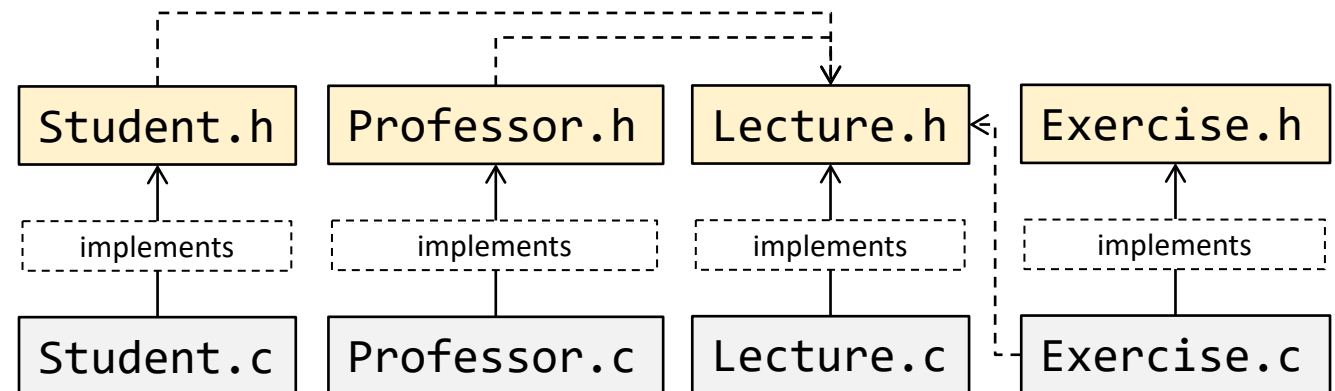
To include header files of the standard library (STL), use `#include<*.h>`.

To include header files of the project, use `#include "*.h"`.



Heuristic

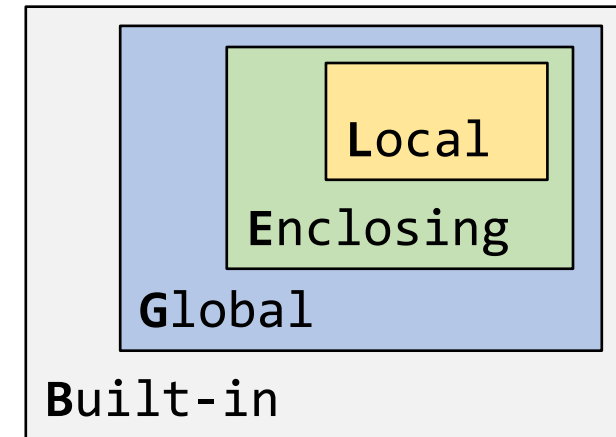
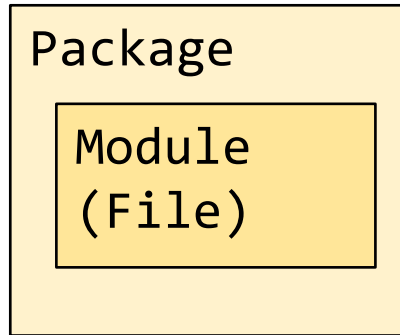
- Only include header files, never include source files.
- Only include files you need.
- Do not rely on includes in included files. Always include everything you need yourself.



Python Packages

Packages in Python

In Python, **packages** group modules in a directory hierarchy. Python has 4 namespaces: Local, Enclosing, Global, and Built-in. When looking up an identifier, the compiler checks through these namespaces, in the same order (called LEGB rule).



Importing in Python

In Python, modules from other packages can be imported

```
from numpy import linalg
import numpy.linalg as linalg

...

linalg.dot(a, b)
```

```
x = 42

def enclosingScope():
    x = 13

    def localScope():
        x = 7
        print(x)

    localScope()

    enclosingScope()
```





Visibility

Visibilities: Overview

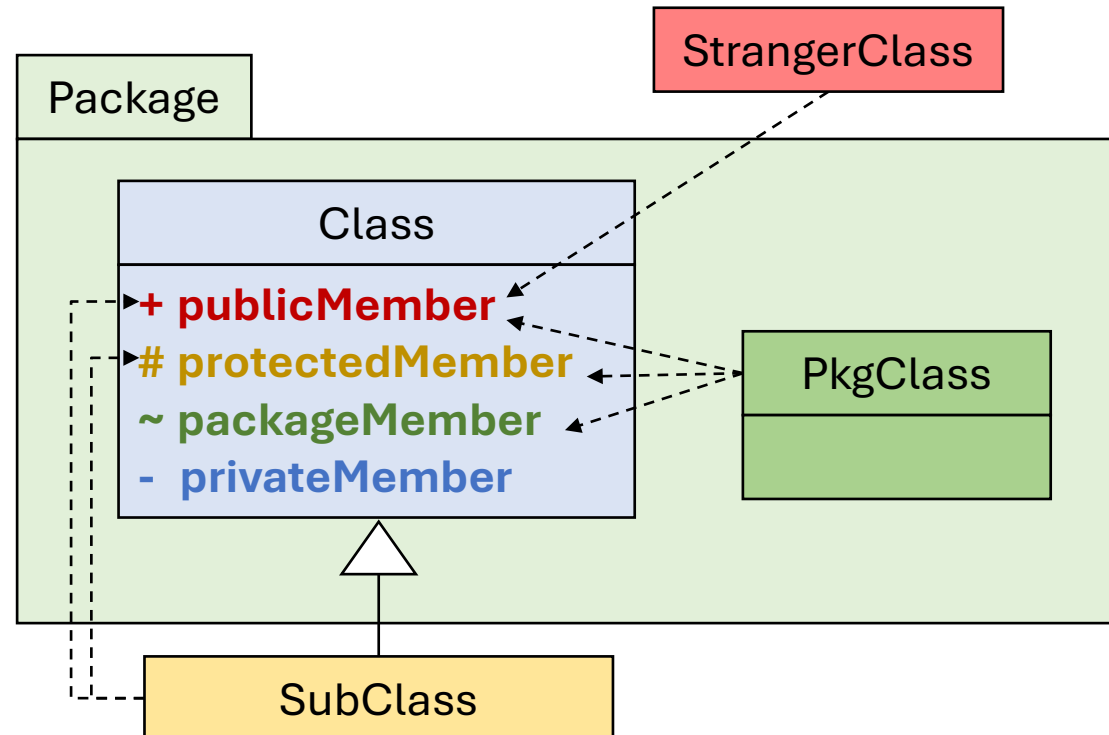
Visibility

Visibility modifiers allow to specify to whom a member should be visible (i.e., usable).

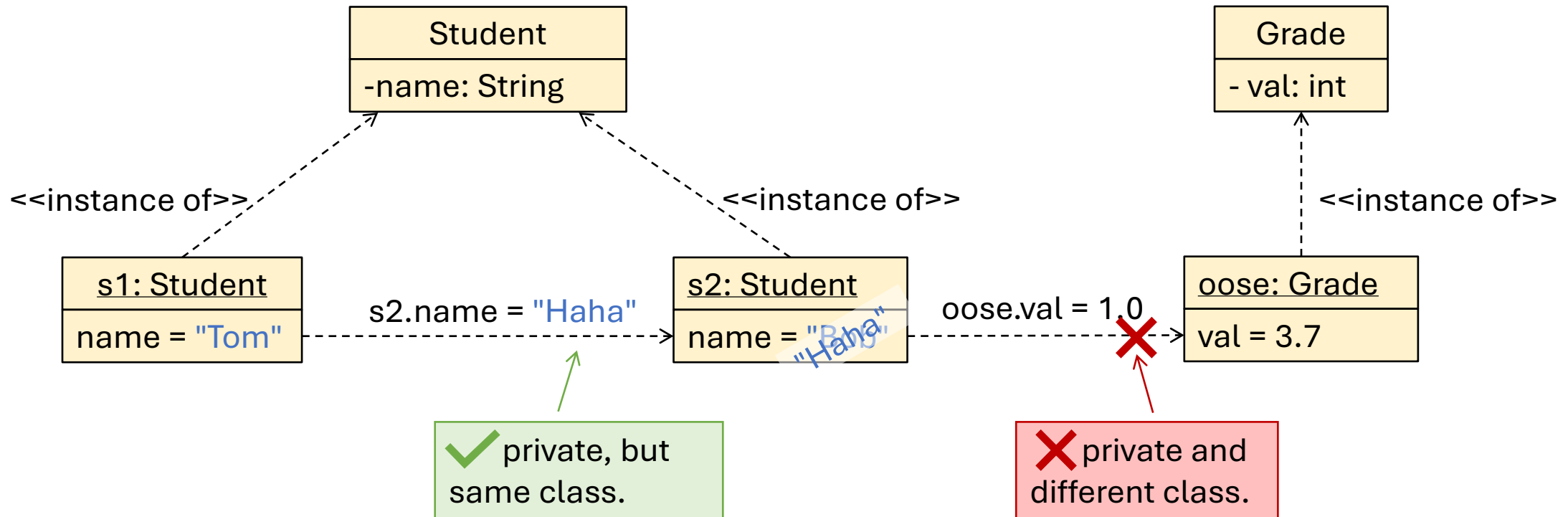
Purpose: Avoid other code making itself dependent on members that might be subject to change.

Attention! Visibility is neither access control, nor mutability!

Visible in:	public	protected	package-private	private
Own class	✓	✓	✓	✓
Own package	✓	✓	✓	✗
Subclasses	✓	✓	✗	✗
Globally	✓	✗	✗	✗



Visibility is on Class Level



Visibility in other Languages


Friends

In C++, methods, procedures and other classes can be defined as friends to allow access to protected or private members.

"Package visibility"

With friends, we can introduce a more flexible "package" visibility. The package consists of all friends of the class.


```
class Student {  
    private:  
        int matNr;  
    public:  
        friend int main();  
        friend Professor::enterGrade(Student& s);  
        friend class Tutor;  
}
```



Visibility in Python?

Python does not offer visibility. Instead, *per convention*, members intended as private start with an `_` underscore. However, this is not checked by the compiler nor the runtime system.


```
class Student:  
    _firstName  
    _lastName  
    _age  
    _matNr
```

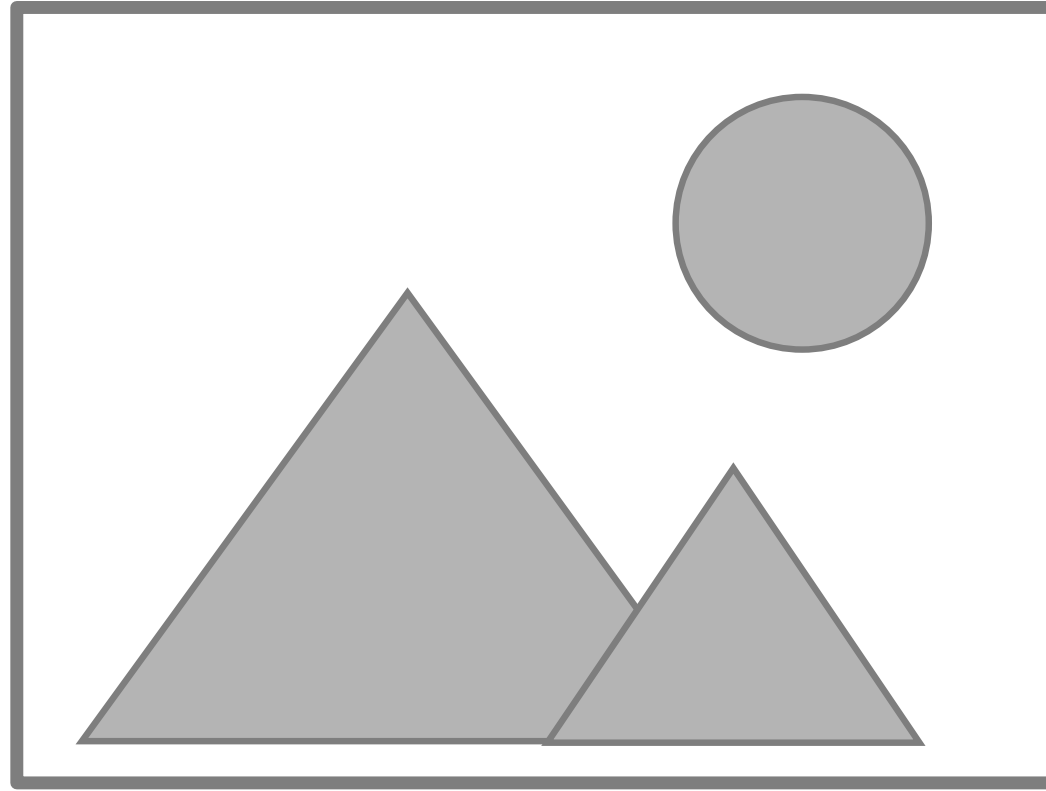


Visibility in JavaScript

In JavaScript, private members start with a `#` hashtag. This is checked by the runtime system. (Constructors cannot be private in JS)

```
class Student {  
    #firstName  
    #lastName  
    #age  
    #matNr  
}
```





Generic Types and Generic Programming

Generic Programming

Generic Programming

Code is generic if parts of it are defined in terms of a placeholder that is later replaced with an arbitrary type.

Observation

Languages with no (mandatory) static typing are inherently generic.

```
class IntList {  
    IntNode head;  
    ...  
}  
  
class IntNode {  
    int data;  
    IntNode next;  
    ...  
}
```



```
class XList {  
    XNode head;  
    ...  
}  
  
class XNode {  
    X data;  
    XNode next;  
    ...  
}
```

Redundancy

Approach 1: Clone it!

Copy code and replace all occurrences for each data type.

- Easy, but absolute no-go.
- "Rule of three".

```
class IntList {  
    IntNode head;  
    ...  
}  
class IntNode {  
    int data;  
    IntNode next;  
    ...  
}
```

```
class FloatList {  
    FloatNode head;  
    ...  
}  
class FloatNode {  
    float data;  
    FloatNode next;  
    ...  
}
```

```
class PersonList {  
    PersonNode head;  
    ...  
}  
class PersonNode {  
    Person data;  
    PersonNode next;  
    ...  
}
```

Manual Genericity

Approach 2: Subtyping

Also known as manual genericity (until Java 5).

Problems:

- Many casts and runtime type checks.
- Often, only common super type is `Object`.

```
List list = new List();
list.add("str");
String s = (String) list.get(0);    //cast necessary
...
Integer i = (Integer) list.get(0); //cast + runtime error
```

```
class List {
    Node head;
    ...
}

class Node {
    Object data;
    Node next;
    ...
}
```


Language-Level Genericity

Approach 3: Real Genericity

Language level genericity allows to specify the type of certain elements on the client side. Generic classes have one or many **type parameters**.

Generic and Parameterized Types

A **generic type** is a template for unlimited many **parameterized types**. Parameterized Types are created through **type instantiation**.

```
public class Node<T> {  
    T data;  
    Node<T> next;  
    setData(T data);  
    ...  
}
```

```
Node<Person> p = new Node<Person>();  
Node<Lecture> l = new Node<Lecture>();  
Node<Department> d = new Node<Department>();  
  
Node<List<Triple<Student, Examiner, Examiner>>>  
    listOfStudentExaminerMappings = new Node<>();
```

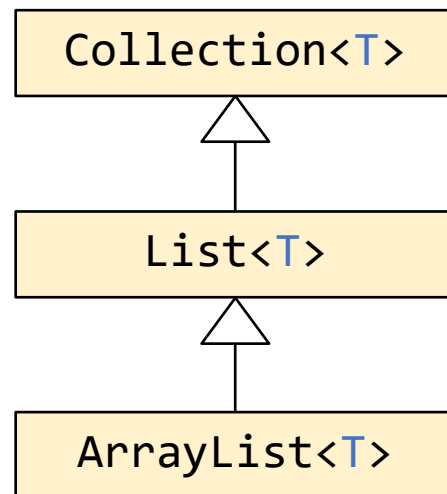
Generic Types and Subtyping

Subtyping in Generics

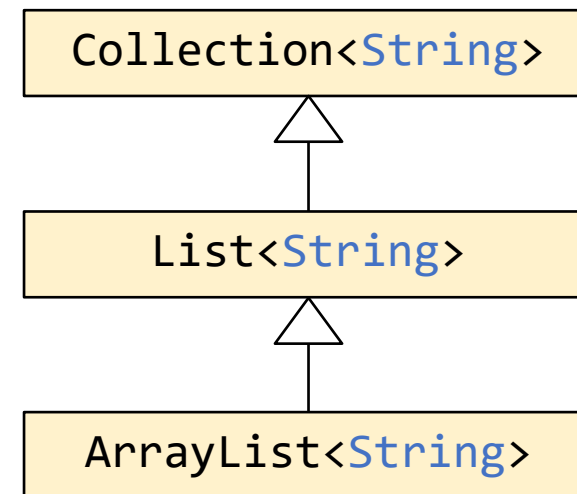
$\text{Sub}_{\text{Gen}}\langle X_1, \dots, X_n \rangle <: \text{Super}_{\text{Gen}}\langle Y_1, \dots, Y_n \rangle \Rightarrow \text{Sub}_{\text{Param}}\langle \mathbf{P}_1, \dots, \mathbf{P}_n \rangle <: \text{Super}_{\text{Param}}\langle \mathbf{P}_1, \dots, \mathbf{P}_n \rangle.$

Mind that the type arguments are exactly the same!

Generic type hierarchy



Parameterized type hierarchy



$T = \text{String}$

$T = \text{String}$

$T = \text{String}$

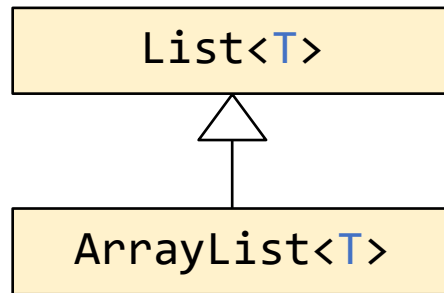
Generic Types and Subtyping

Subtyping in Generics

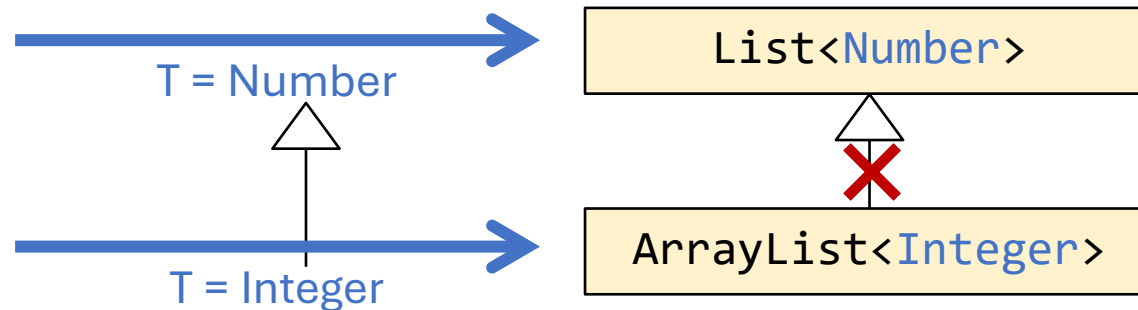
The parameterized type $\text{Sub}\langle P_1, \dots, P_n \rangle$ is never a subtype of the parameterized type $\text{Super}\langle Q_1, \dots, Q_n \rangle$.

- Though generic Sub is subtype of generic Super .
- Even if P_i is subtype of Q_i .
- Even if $\text{Sub} == \text{Super}$.

Generic type hierarchy



Parameterized type hierarchy



Why?

```
List<String> ls = new ArrayList<String>();           //okay
List<Object> lo = ls;                               //not allowed in Java
lo.add(new Object());                               //insert an Object instance into lo
String s = ls.get(0);                               //type error! Cannot assign Object to String
```

So, is there no subtyping possible for type arguments? It is, and it gets complicated.



Bounded Genericity

Bounded Genericity

When an upper bound is declared, the compiler knows that any actual type argument has at least the interface of the bound. Thus, it can allow access to members of the bound type.

Multiple Bounds

A generic can have multiple upper bounds. If one is a concrete or abstract class, it must be put first.

```
public class Example<T
    extends Person
    & Cloneable
    & Printable> {
    ...
}
```

```
public class Node<T extends Number> {
    T data;
    public MyClass(T v) {
        data = v;
    }
    public void setData(T v) {
        data = v;
    }
    public T getData() {
        return data;
    }
    public void print() {
        println(this.getData()
            .intValue());
    }
}
```



Generic Methods

```
public class Other {  
    public void print(Node<Number> n) {  
        println(n.getData().intValue());  
    }  
}  
  
print(new Node<Number>(42));
```

This method is not generic, as it does not deal with any type parameters. The expected parameter is a parameterized type, thus not generic.

```
public class Other <E extends Number> {  
    public void print(Node<E> n) {  
        println(n.getData().intValue());  
    }  
}  
  
Other o = new Other<Integer>();  
o.print(new Node<Integer>(42));
```

This method is not a *real* generic method, but it makes use of the classes type parameters.

```
public class Other {  
    public <E extends Number> void print(Node<E> n) {  
        println(n.getData().intValue());  
    }  
}  
  
Other o = new Other();  
o.<Integer>print(new Node<Integer>(42));
```

This method is a *real* generic method. It introduces its own type parameters, bound to its scope.



Wildcards

Wildcard

If a type parameter is used once only, a wildcard should be used instead. Wildcards hide the genericity of a method from clients. The wildcard `?` is an unnamed type, not a type variable. Wildcards are allowed as types of local variables, attributes, parameters and return types (though this should be avoided).

When and when not to use Wildcards

Wildcards are preferred if there are no interdependencies between two parameterized types. (B in example)

```
public class Other {  
    public void print(Node<? extends Number> n) {  
        println(n.getData().intValue());  
    }  
}  
  
print(new Node<Integer>(42));
```

```
public <A extends Number, B extends Integer>  
void print(Node<A> n, List<B> l);
```



```
public void print(Node<? extends Number> n,  
List<? extends Integer> l);
```

```
public <A extends Number, B extends Integer>  
void print(Node<A> n1, Node<A> n2, List<B> l);
```

Input and Output Parameter

Input Parameters

Generics with an **upper bound** are input parameters, i.e., we can only call methods on them that do not have the type parameter as parameter types, but as return types (we can "read from them", thus "input"). We also call them "**producers**".

Output Parameters

Generics with a **lower bound** are output parameters, i.e., we can only call methods on them that do not have the type parameter as return type, but only as parameter type (we can "write into them", thus "output"). We also call these "**consumers**".

```
<T> void copy(List<? extends T> source, List<? super T> destination) {  
    ...  
}
```

Implementation of Generics

Homogenous Translation

Java and many other languages implement generics via **homogenous translation**. Type parameters and arguments are checked by the compiler, but then removed (**type erasure**). Thus, a `List<Int>` and a `List<Person>` are just a `List` at runtime.

Restrictions on Generics in Java

- Object creation: no `"new T()"`,
- Type checking: no `"instanceof T"`,
- Declaration of class variables:
no `"static T var;"`,
- Create arrays of parameterized types:
no `"new List<Number>[10]"`,
- Exceptions: no `"class Exception<T>"`,
- Overloading complicated.
- Type arguments must be reference types.

Implementation in Java

Generic Type: "Type erasure". All occurrences of type parameters are replaced with their type argument.

- unbounded → `Object`
- bounded → `Bound`

"Bridge methods" (no details here) are added.

Parameterized Type: Compiler adds type checks and casts (like manual genericity).

- Fast compilation.
- Small byte code.

Raw Type

Because of backwards compatibility, raw types (generics that are instantiated without type arguments) are allowed to be used, but highly discouraged.



Coding Time!

Implementation of Generic and Parameterized Types in C++

Templates

Heterogenous Translation

C++ and others realize generics (called templates) via [heterogenous translation](#). When instantiating a generic type, a copy of the class is created where all occurrences of type arguments are replaced with their arguments ([type expansion](#)).

This quickly leads to huge code.

```
template<typename T> class Node {  
    T data;  
    Node<T> next;  
    void setData(T) ...  
    T getData() ...  
}
```



```
class Node_MyData {  
    MyData data;  
    Node_MyData next;  
    void setData(MyData) ...  
    MyData getData() ...  
}
```

```
Node<MyData> n = new Node<MyData>();
```



```
Node_MyData n = new Node_MyData();
```



Generics in Kotlin

Generics in Kotlin

Generics work as in Java, but Kotlin does not have wildcards. Instead, it provides **declaration site variance** and **type projections**.

Declaration-Site Variance

Declaration site variance is a way to tell the compiler that a type will only be a **producer** (**out**) or a **consumer** (**in**).

Use-Site Variance / Type Projection

Some classes can not be restricted to **in** or **out** at declaration site, e.g., `Array`. We can use **in** and **out** on the call site as well, restricting how we can interact with the object.

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }  
fun fill(dest: Array<in String>, value: String) { ... }
```

```
interface Source<out T> {  
    fun nextT(): T  
}  
  
interface Comparable<in T> {  
    operator fun compareTo(other: T): Int  
}  
  
fun demo(strs: Source<String>,  
        x: Comparable<Number>) {  
    val objects: Source<Any> = strs  
        //Okay, T is only in out-Position  
    ...  
    x.compareTo(1.0) /* 1.0 is Double <:  
        Number, thus, ... */  
    val y: Comparable<Double> = x // OK!  
}
```

