

$X\langle T \rangle$

$\langle \text{Number} \rangle$

$\langle T \rangle$

Generics

$\langle ? \rangle$

LinkedList $\langle T \rangle$

$\langle ? \text{ extends Fish} \rangle$

Listen ohne Generics

```
List myStringList = new LinkedList();  
myStringList.add(new String(0));  
String s = (String)myStringList.get(0);
```

Cast

“Ich *meine*, der Typ *sollte* hier
zum Typ String passen.”

Listen **mit** Generics

```
List<String> myStringList = new LinkedList<String>();  
myStringList.add(new String(0));  
String s = myStringList.get(0);
```

Generics

“Das passt *immer* zum Typ
String.”

Listen mit Generics

```
List<String> myStringList = new LinkedList<String>();  
myStringList.add(new String(0));  
String s = myStringList.get(0);
```

Generics

Typüberprüfung zur
Compilezeit

```
public class Pair {  
    Number first;  
    Number second;  
  
    public Pair (Number first, Number second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
public class Pair {  
    Number first;  
    Number second;  
  
    public Pair (Number first, Number second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
public class Pair {  
    String first;  
    String second;  
  
    public Pair (String first, String second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
public class Pair {  
    Number first;  
    Number second;  
  
    public Pair (Number first, Number second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
public class Pair <T> {  
    T first;  
    T second;  
  
    public Pair (T first, T second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```



```
public class Pair <T> {  
    T first;  
    T second;  
  
    public Pair (T first, T second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
Pair<String> pS = new Pair<>("s", "t"); // Kurzform  
Pair<String> pS = new Pair<String>("s", "t"); //  
Langform
```

“

```
public class Pair <T> {  
    T first;  
    T second;
```

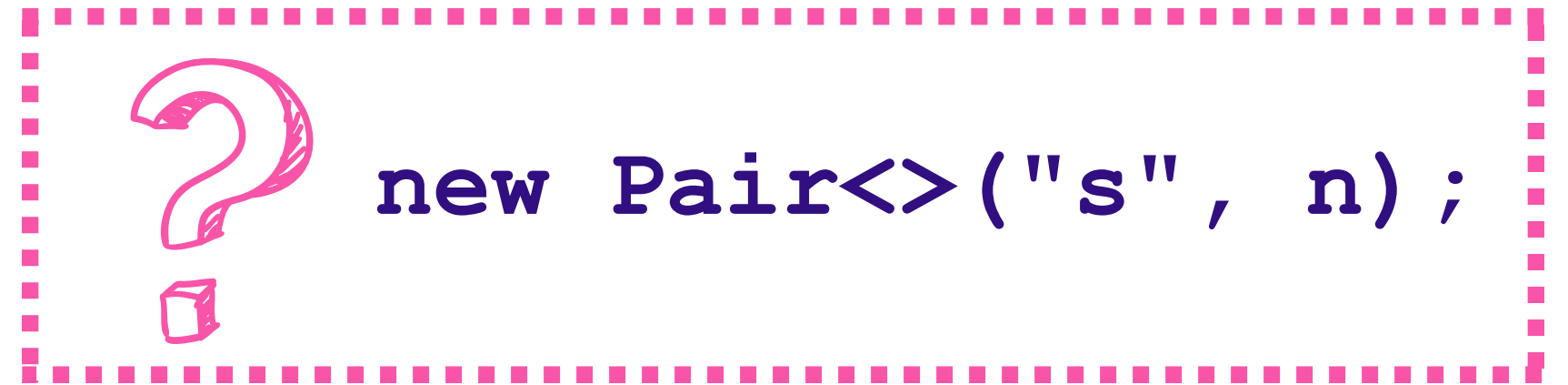
Schreiben Sie eine generische public-Klasse Pair mit einem generischen Typparameter T.

```
    public Pair (T first, T second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

”

```
Pair<String> pS = new Pair<>("s", "t"); // Kurzform  
Pair<String> pS = new Pair<String>("s", "t"); //  
Langform
```

```
public class Pair <T> {  
    T first;  
    T second;
```

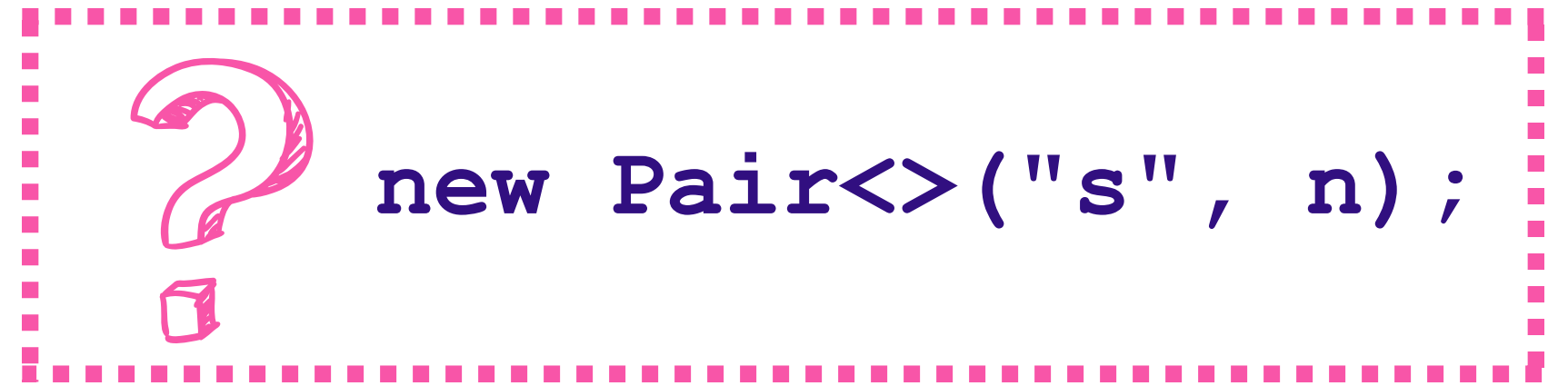


? new Pair<>("s", n);

```
    public Pair (T first, T second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
Pair<String> pS = new Pair<>("s", "t"); // Kurzform  
Pair<String> pS = new Pair<String>("s", "t"); //  
Langform
```

```
public class Pair <S, T> {  
    S first;  
    T second;
```



```
new Pair<>("s", n);
```

```
    public Pair (S first, T second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

```
Number n = new Number(3.0);
```

```
Pair<String, Number> pSN = new Pair<>("s", n);
```



```
Pair<Number> p =  
    new Pair<Double>(d) ;
```

Nein!



```
Pair<Number> p =  
    new Pair<Double>(d) ;
```

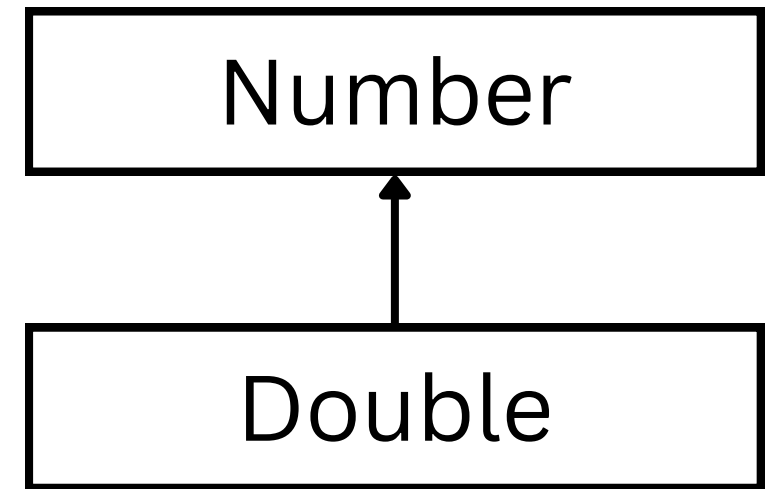
List<Double> ist **kein** Subtyp von List<Number>



**Wir wollen einen gemeinsamen Obertyp:
“etwas was zu Number und nur zu
Number passt”.**

Wie setzen wir das um?

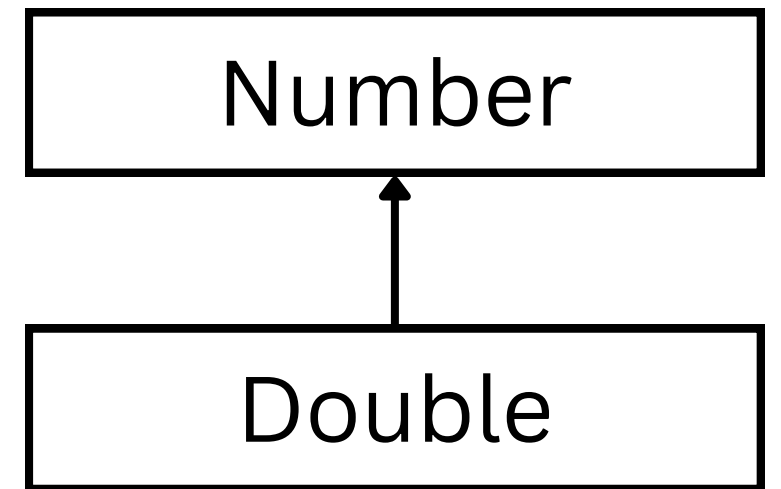
```
public class Pair <T extends Number> {  
    T first;  
    T second;  
  
    public Pair (T first, T second) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```



```
Pair<Number> pN = new Pair<Number>(n);  
Pair<Number> pD = new Pair<Double>(d);
```




```
public class Pair <T extends Number> {  
    T first;  
    T second;  
  
    public Pair (T first, T second ) {  
        this.first = first;  
        this.second = second;  
    } ...  
}
```

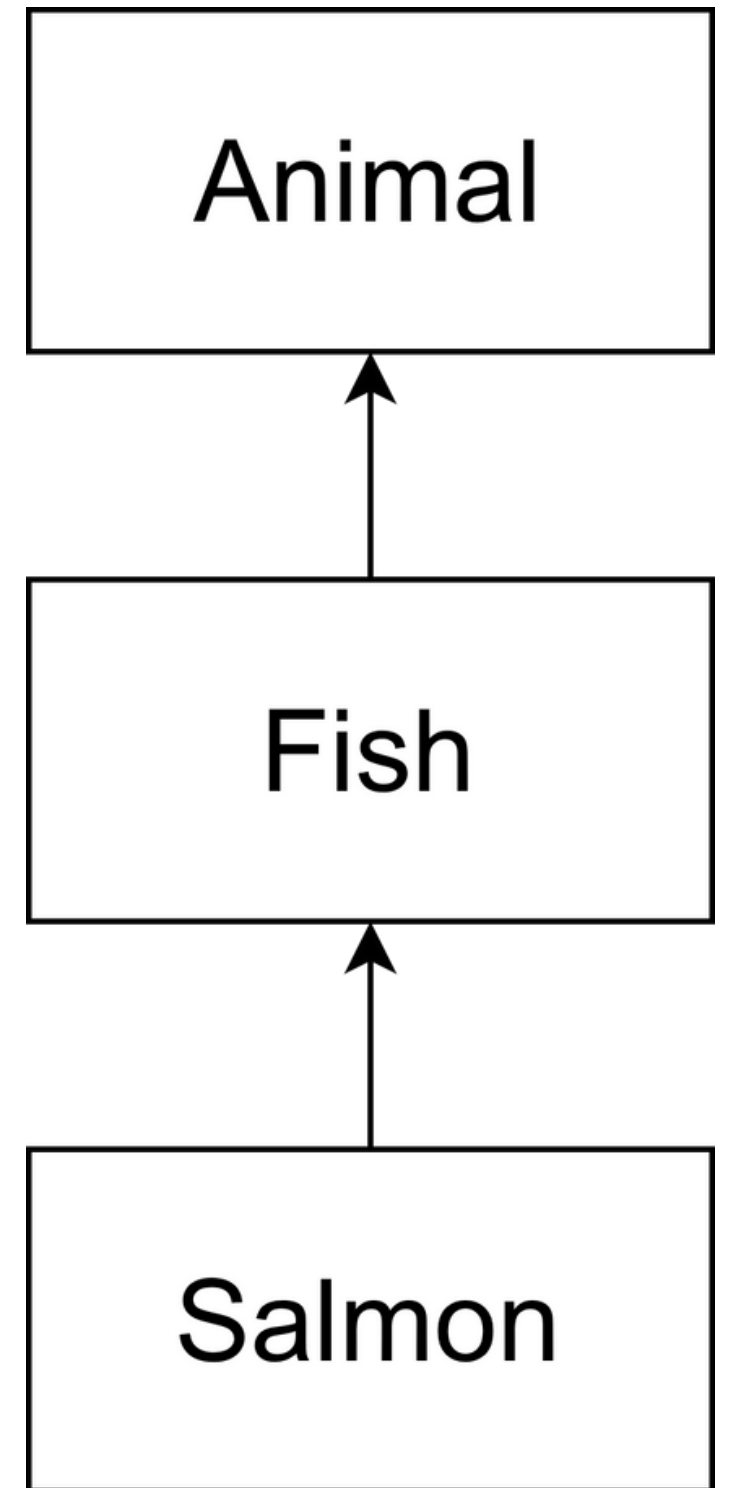


““

Schreiben Sie eine generische public-Klasse Pair mit einem generischen Typparameter T, der sowohl auf die Klasse Number und alle Subtypen von Number beschränkt ist.

DON'T FORGET

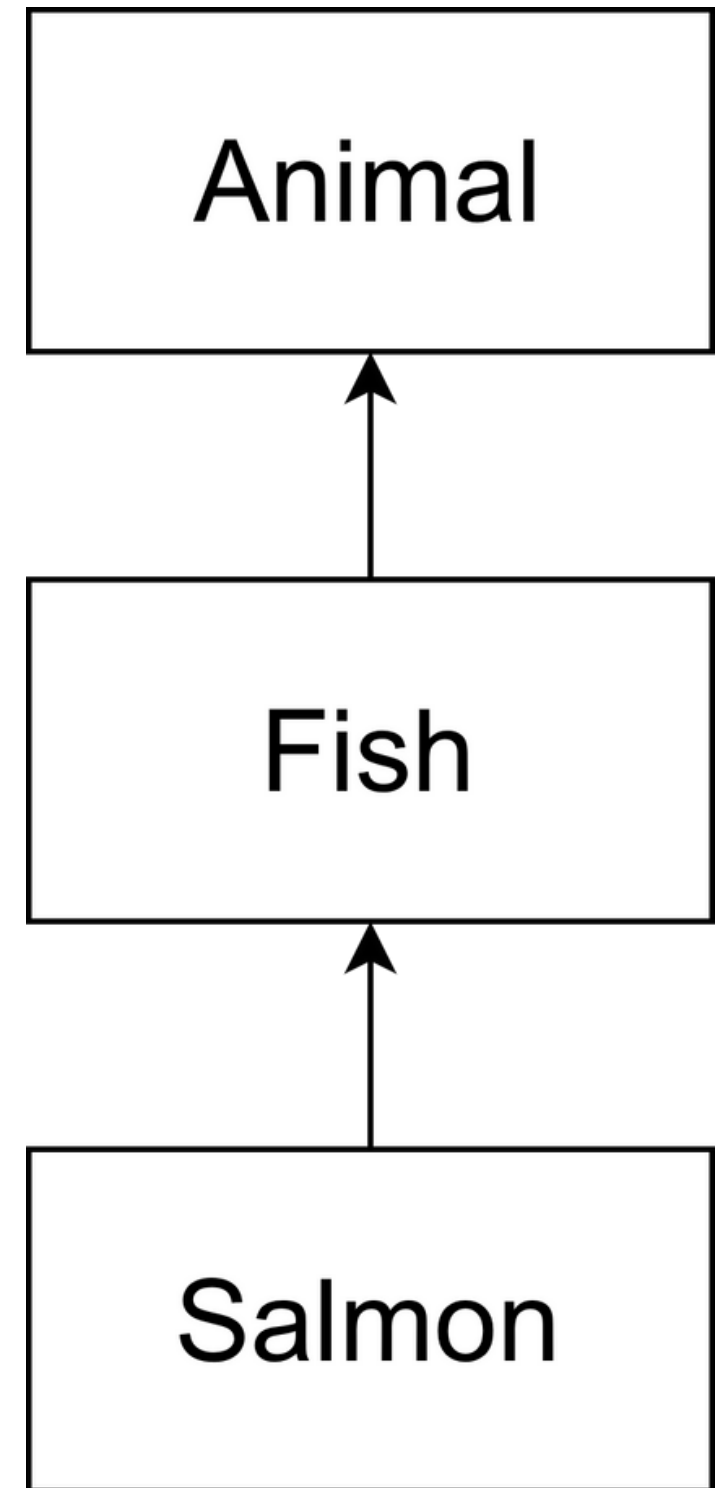
```
Fish f1 = new Animal(); ⚡  
Fish f2 = new Fish(); ✓  
Fish f3 = new Salmon(); ✓
```



DON'T FORGET

```
Fish f1 = new Animal(); ⚡  
Fish f2 = new Fish(); ✓  
Fish f3 = new Salmon(); ✓
```

**Der dynamische Typ muss ein
(in-)direkter Subtyp vom
statischer Typ sein!**



```
List<Fish> f = new LinkedList<>();
```

```
f.add(new Animal());
```

```
f.add(new Fish());
```

```
f.add(new Salmon());
```

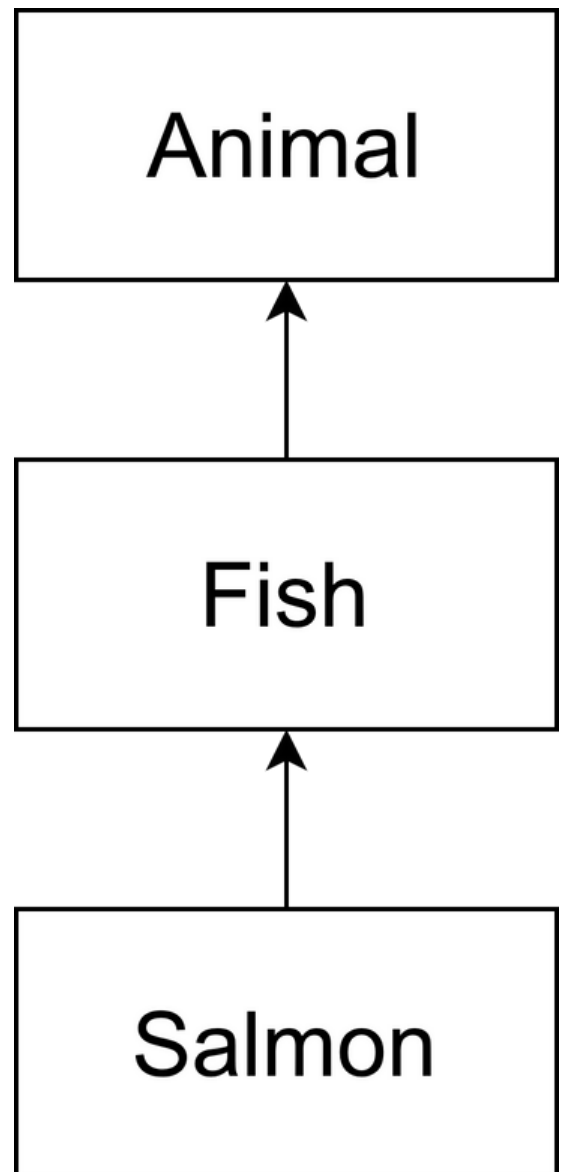


```
List<Fish> f;
```



```
f = new LinkedList<Fish>();
```

```
f = new LinkedList<Salmon>();
```



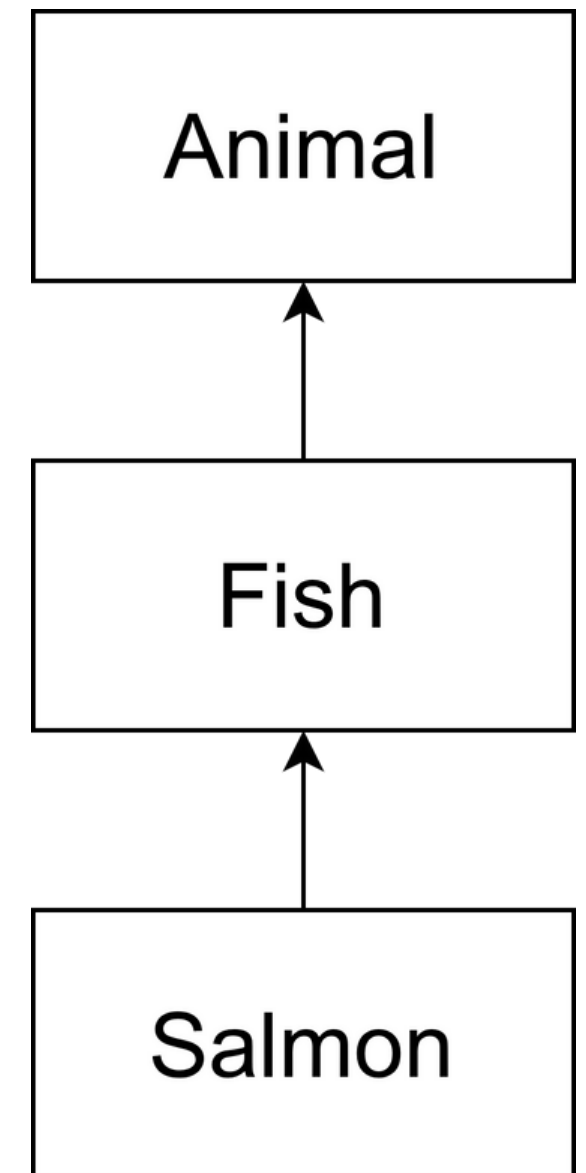
```
List<Fish> f;
```



```
f = new LinkedList<Fish>();  
f = new LinkedList<Salmon>();
```

```
List<? extends Fish> f;
```

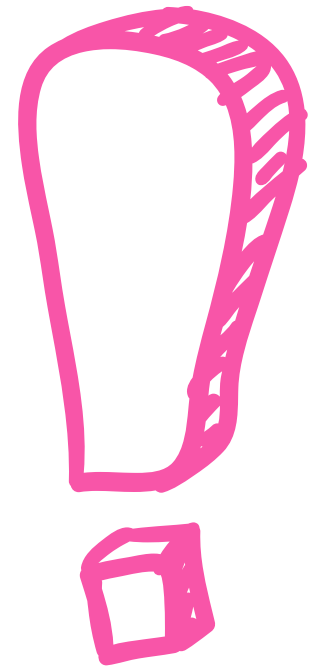
? als Wildcard





Typparameter T, der
auf Animal und alle
Subtypen von Animal
eingeschränkt ist

generischer
Typparameter, der auf
Animal und alle
Subtypen von Animal
eingeschränkt ist



Typparameter T, der
auf Animal und alle
Subtypen von Animal
eingeschränkt ist

<T extends Animal>

generischer
Typparameter, der auf
Animal und alle
Subtypen von Animal
eingeschränkt ist

<? extends Animal>

Interfaces und Generics

```
public interface X<T extends Number> { }
```

```
public class A implements X { }
```



```
public class A implements X<Number> { }
```



```
public class A<T extends Number>  
    implements X<T> { }
```



```
public class A<T extends Number>  
    implements X<T extends Number> { }
```



Interfaces und Generics

```
public interface X<T extends Number> { }
```

```
public class A implements X { }
```



```
public class A implements X<Number> { }
```



```
public class A<T extends Number>  
    implements X<T> { }
```



```
public class A<T extends Number>  
    implements X<T extends Number> { }
```



Interfaces und Generics

```
public interface X<T extends Number> { }
```

```
public class A<T extends Number>  
    implements X<T> { }
```

“

Schreiben Sie eine public-Klasse A, die das Interface X mit denselben Einschränkungen von A wie bei X und A instanziiert mit T instanziiert.

”

Interfaces und Generics

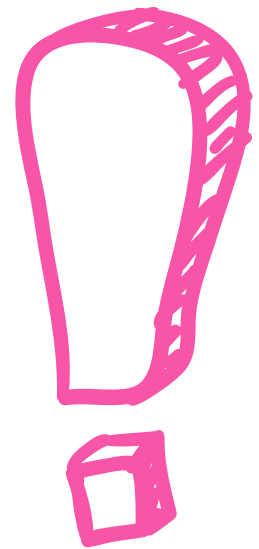
Int1 und Int2 sind Interfaces



```
public class A <T extends X & Int1 & Int2>{ }
```

Interfaces und Generics

```
public class A <T extends X & Int1 & Int2> {}
```



Mit dieser Schreibweise lässt sich erzwingen, dass der Typparameter T sowohl auf Klasse X als auch auf die zwei hier aufgezählten Interfaces eingeschränkt ist. Das heißt, erlaubt als Instanziierungen von T sind nur Klassen, die jedes der zwei Interfaces direkt oder indirekt implementieren und entweder gleich X oder von X direkt oder indirekt abgeleitet sind.

Zusammenfassung

Typangabe	Typ beim Lesen	Einfügen von ...	Zuweisen
List <Number>	Number	Number und Subtypen	nur List<Number>
List<? extends Number>	Number	nur null	List<Number> und List<X> (X Subtyp von Number)
List<? super Number>	Object	Number und Subtypen	List<Number> und List<X> (X Supertyp von Number)