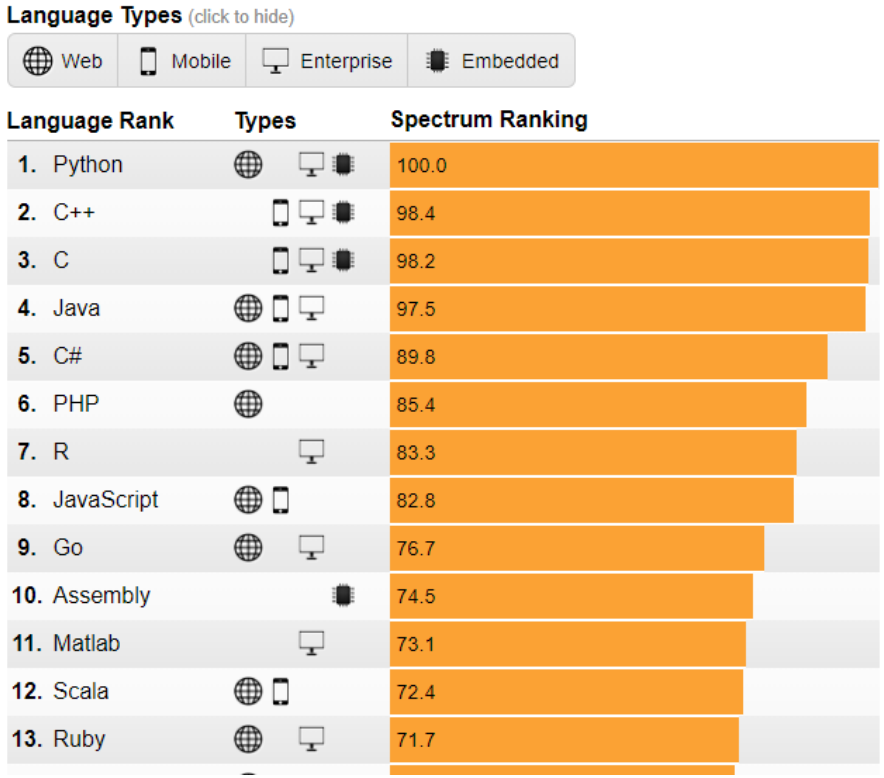
Objektorientierte Programmierung

Hochschule Bochum
WS 19/20
Dr.-Ing. Darius Malysiak

III. Vertiefung Java 1-8

Derzeitige Stellung von Java:



https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019

III. Vertiefung Java 1-8

Entwicklung von Java:

JDK Version 1.1
- JDBC (Java Database Connectivity)
- Inner Classes
- Java Beans
- RMI (Remote Method Invocation)
- Reflection (introspection only)

J2SE Version 1.2
- Collections framework.
- Java String memory map for constants.
- Just In Time (JIT) compiler.

J2SE Version 1.4
- XML Processing
- Logging API
- Java Web Start
- Assertions
- Chained Exception
- IPv6 Support
- Regular Expressions
- Image I/O API

J2SE Version 5.0
- Generics
- Enhanced for Loops
- Autoboxing/Unboxing
- Typesafe Enums
- Varargs
- Static Import
- Metadata (Annotations)
- Instrumentation

III. Vertiefung Java 1-8

Entwicklung von Java:

Java Version SE 7

- Strings in Switch Statement
- Type Inference for Generic Instance Creation
- Multiple Exception Handling
- Java nio Package
- Diamond Syntax

Java SE 8

- Lambda Expressions
- Pipelines and Streams
- Date and Time API
- Type Annotations

III. Vertiefung Java 1-8

Generics:

```java
import java.util.ArrayList;
import java.util.List;
public class Container {
    private List l = new ArrayList();
    public void addToContainer(Object s) {
        l.add(s);
    }
    public Object get(int i) {
        return l.get(i);
    }
}
```

```java
public class Main {
    public static void main(String [] args)
    {
        Container c = new Container();
        c.addToContainer("test");
        Integer i = (Integer)c.get(0);
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
        at test.Main.main(Main.java:10)

Process finished with exit code 1
```

III. Vertiefung Java 1-8

Generics:

- Obwohl der Code kompiliert kommt es zu einem Laufzeitfehler.
- Prüfung durch Compiler nicht möglich. **Warum?**
- Casting von Object zu Kinderklassen syntaktisch immer möglich.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
        at test.Main.main(Main.java:10)

Process finished with exit code 1
```

III. Vertiefung Java 1-8

Generics:

```java
import java.util.ArrayList;
import java.util.List;
public class Container<T> {
    private List<T> l = new ArrayList<T>();
    public void addToContainer(T s) {
        l.add(s);
    }
    public T get(int i) {
        return l.get(i);
    }
}
```

```java
public class Main {
    public static void main(String [] args)
    {
        Container<String> c = new
        Container<String>();
        c.addToContainer("test");
        Integer i = (Integer)c.get(0);
    }
}
```

Compiler-Error

Error:(10, 35) java: incompatible types: java.lang.String cannot be converted to java.lang.Integer

III. Vertiefung Java 1-8

Generics:

```
public class Main {
    public static void main(String [] args) {
        Container<String> c = new Container<>();
        c.addToContainer("test");
        String s = c.get(0);
    }
}
```

Diamond-Operator

Der Compiler ist **meist** in der Lage Typen zu inferieren.

III. Vertiefung Java 1-8

Generics:

Generics können beschränkt (bounded) sein

```
import java.util.ArrayList;
import java.util.List;
public class Container<T extends String> {
    private List<T> l = new ArrayList<T>();
    public void addToContainer(T s) {
        l.add(s);
    }
    public T get(int i) {
        return l.get(i);
    }
}
```

```
public class Main {
    public static void main(String [] args)
    {
        Container<Double> c = new
        Container<Double>();
        c.addToContainer(new Double(1));
        Double i = c.get(0);
    }
}
```

Upper-Bounded

Compiler-Error

Error:(6, 19) java: type argument java.lang.Double is not within bounds of type-variable T

III. Vertiefung Java 1-8

Generics:

Generics gelten auch für Variablen

```
public class Main {
    public static void main(String [] args) {
        Container<? extends Double> c = new Container<Double>();
        c.addToContainer(new Double(1));
        Double i = c.get(0);
    }
}
```

Nur Lese-Zugriff möglich!

Compiler-Error

Error:(10, 26) java: incompatible types: java.lang.Double cannot be converted to capture#1 of ? extends java.lang.Double

III. Vertiefung Java 1-8

Generics:

Generics gelten auch für Variablen

```
public class Main {
    public static void main(String [] args) {
        Container<? super Double> c = new Container<Double>();
        c.addToContainer(new Double(1));
        Double i = c.get(0);
    }
}
```

Lower-Bounded

Nur Schreib-Zugriff möglich!

Compiler-Error

Error:(12, 25) java: incompatible types: capture#1 of ? super java.lang.Double cannot be converted to java.lang.Double

III. Vertiefung Java 1-8

Generics:

Generics gelten auch für Variablen

```
public class Main {
    public static void main(String [] args) {
        Container<? super Double> c = new Container<Double>();
        c.addToContainer(new Double(1));
        Double i = (Double)c.get(0);
    }
}
```

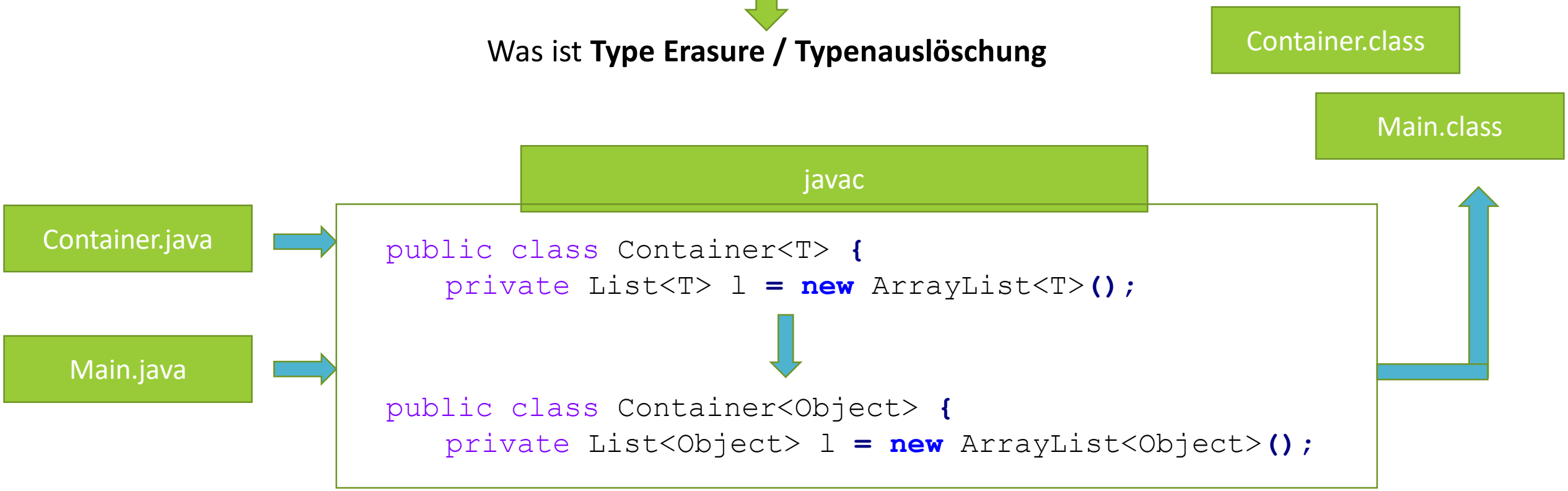
Lower-Bounded

III. Vertiefung Java 1-8

Generics:

Was ist das eigentliche technische System hinter Generics?

Was ist **Type Erasure / Typenauslöschung**



III. Vertiefung Java 1-8

Generics – Type Erasure:

Unbounded Generic

javac

```
import java.util.ArrayList;
import java.util.List;
public class Container {
    private List<Object> l = new ArrayList<Object>();
    public void addToContainer(Object s) {
        l.add(s);
    }
    public Object get(int i) {
        return l.get(i);
    }
}
```

Der Generic-Parameter wird durch Object ersetzt

III. Vertiefung Java 1-8

Generics – Type Erasure:

Bounded Generic

javac

```
import java.util.ArrayList;
import java.util.List;
public class Container {
    private List<String> l = new ArrayList<String>();
    public void addToContainer(String s) {
        l.add(s);
    }
    public String get(int i) {
        return l.get(i);
    }
}
```

Der Generic-Parameter wird durch die obere Schranke ersetzt

III. Vertiefung Java 1-8

Generics – Type Erasure:

javac

```
public class Main {
    public static void main(String [] args) {
        Container<Object> c = new Container<Object>();
        c.addToContainer((Object)new String("test"));
        String i = (String)c.get(0);
    }
}
```

Explizite casts bei Verwendung von Generics

???? Wieso überhaupt Generics ????
Ist doch eh alles Object !!

III. Vertiefung Java 1-8

Generics – Type Erasure:

- Der Compiler ersetzt jeden Generic-Parameter durch eine konkrete Klasse.
- Unbounded Parameter werden immer durch Object ersetzt.
- Bounded Parameter werden durch die Klasse ersetzt, welche die Schranke darstellt.
- Der Compiler kann **vor** Type Erasure eine Typüberprüfung durchführen.
- Sofern es keine Konflikte gibt wird Type Erasure durchgeführt, gefolgt von Optimierungen und letztendlich Byte Assembling.
- Bei Klassen sind nur Upper-Bounds (‚extends') zulässig. Warum ist ‚class C<T super Some>{}' sinnlos? -> ‚C<Object>' prinzipiell möglich -> muss durch Object ersetzt werden!!
- Bei Variablen zusätzlich ‚implements' und ‚super' als Lower-Bounds.

Später mehr dazu!

III. Vertiefung Java 1-8

Generics:

```
public static <T> T get(T s) { return s; }
```

- Generics können ebenfalls für einzelne Methoden genutzt werden.
- Methoden müssen nicht statisch sein, die Klasse kann z.B. keine Generics enthalten

```
public class SomeClass {
public <T> T get(T s)
    {
        return s;
    }
}
```

III. Vertiefung Java 1-8

Generics:

Warum ist dies prinzipiell nicht möglich ?

```
public class Container<T>
{
    public static final T S;

    public T get()
    {
        return S;
    }
}
```

Compiler-Error

Error:(9, 25) java: non-static type variable T cannot be referenced from a static context

III. Vertiefung Java 1-8

Generics:

Warum ist dies prinzipiell nicht möglich ?

```
public class Container<Object>
{
    public static final Object S;

    public Object get()
    {
        return S;
    }
}
```

```
public void f1() {
    Container<Double> c = new Container<>();
    c.S = new Double(new Double(1));
}

public void f2() {
    Container<String> c = new Container<>();
    c.S = new String("12");
}
```

Compiler-Error

Error:(9, 25) java: non-static type variable T cannot be referenced from a static context

III. Vertiefung Java 1-8

Generics:

Warum kein „super“ oder „implements“ als Generic-Bounds bei Klassen?

```
public class Container<T super SomeClass>
```

```
public class Container<T implements SomeClass>
```

Hint: Was soll T sein? Kann ein Interface Upper- oder Lower-Bound sein?

III. Vertiefung Java 1-8

Lambdas

Definition:
Boilerplate Code ist die Bezeichnung für ein Codefragment welches unverändert an vielen Stellen im Projekt verwendet wird.

Bemerkung:
Boilerplate Code kann in der Regel nicht vollständig entfernt werden, es wird daher versucht diesen Code mit Hilfe von kurzen Ausdrücken zu ersetzen.

III. Vertiefung Java 1-8

Lambdas

Definition:
Boilerplate Code ist die Bezeichnung für ein Codefragment welches unverändert an vielen Stellen im Projekt verwendet wird.

```
for(int i=0;i<list.size();i++) {
        String s = list.get(i);
        StringFilter filter = StringFilter::getInstance();
        String[] data = filter.splitCanonical(s);
        //real business logic on data starts here
        //..
}
```

III. Vertiefung Java 1-8

Lambdas

Definition:
Boilerplate Code ist die Bezeichnung für ein Codefragment welches unverändert an vielen Stellen im Projekt verwendet wird.

```
for(String s : list) {
        StringFilter filter = StringFilter::getInstance();
        String[] data = filter.splitCanonical(s);
        //real business logic on data starts here
        //..
}
```

Gar nicht so viel besser ☹

III. Vertiefung Java 1-8

Lambdas

Definition:
Boilerplate Code ist die Bezeichnung für ein Codefragment welches unverändert an vielen Stellen im Projekt verwendet wird.

```
list.map( p -> filter.splitCanonical(p) ).
        collect(Collectors.toList()).forEach(s -> b(s));
```

Hier wurden Lambda-Ausdrücke und Streams verwendet, wir werden zunächst Lambda-Ausdrücke diskutieren.

III. Vertiefung Java 1-8

Lambdas: ???

Definition:

Ein Lambda-Ausdruck (Lambda Expression) ist ein in Java verfasster Ausdruck welcher eine Referenz auf eine konkrete Implementierung von funktionalen Schnittstellen repräsentiert.

Synonyme dazu sind häufig ‚Lambda-Funktion' , ‚anonyme Funktion' oder ‚lambda Implementierung'

III. Vertiefung Java 1-8

Lambdas: Implementierung von Interfaces

```
public interface Test { boolean check(String text); }

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

            Test myInterface =
                (String text) -> {
                    if (text.contains("!")) { return true; }
                    return false;
                };

            System.out.println("" + myInterface.check("Hello world!"));
    }
}
```

III. Vertiefung Java 1-8

Lambdas: Implementierung von Interfaces

```
Test myInterface =
        (text) -> {
                if (text.contains("!")) { return true; }
                return false;
        };
```

Mit Hilfe von Inferenz der Typen ist auch hier eine kompaktere Schreibweise möglich.

```
public interface Test2 { boolean check(); }
```

```
Test2 myInterface =
        () -> {
                return false;
        };
```

Falls kein Parameter benötigt wird so kann „()“ verwendet werden

III. Vertiefung Java 1-8

Lambdas: Implementierung von Interfaces

```
public interface Test3 { void check(); }
Test3 myInterface = () -> System.out.println("check");
```

- Falls der Rumpf des Lambda-Ausdrucks aus einem einzigen zusammenhängenden Ausdruck besteht können die Klammern ausgelassen werden.
- Somit wäre bei der Rückgabe eines Wertes z.B. ‚return a > b' nicht korrekt, stattdessen reicht es ‚a > b' zu schreiben.

III. Vertiefung Java 1-8

Lambdas: Implementierung von Interfaces

Unterschied zu anonymen Interface Implementierungen:

Anonyme Interface Implementierungen können einen Zustand besitzen.

```
Test myInterface = new Test() {
    private boolean res = false;

    public boolean check(String text){
        if (text.contains("!")) res = true;
        else res = false;
        return res;
    }
};
```

Lambda Funktionen haben keinen Zustand.

III. Vertiefung Java 1-8

Lambdas: Implementierung von Interfaces

Ein Lambda-Ausdruck wird erfolgreich gegen ein Interface gematched g.d.w folgende Bedingungen erfüllt sind:

- Nur eine Methode des Interfaces ist nicht implementiert.
- Die Parameter des Lambda-Ausdrucks stimmen mit jenen der zu implementierenden Methode überein.
- Der Rückgabetyp stimmt mit jenem der zu implementierenden Methode überein.

III. Vertiefung Java 1-8

Lambdas: Zugriff auf Variablen

Ein Lambda-Ausdruck kann auf externe Elemente zugreifen:

- Lokale Variablen des Definitions-Kontexts

```
public interface Test2 { boolean check(); }


final boolean res = false;
Test2 myInterface = () -> { return res;};
```

III. Vertiefung Java 1-8

Lambdas: Zugriff auf Variablen

Ein Lambda-Ausdruck kann auf externe Elemente zugreifen:

- Statische Variablen des Instanz-Kontexts

```
public interface Test2 { boolean check(); }


private static final boolean res = false;

public void f()
{
        register(() -> { return res;})
}
```

III. Vertiefung Java 1-8

Lambdas: More Syntax Sugar

```
public interface Test3 { void print(String s); }
```

```
Test3 myInterface = System.out::println;
```

```
Test3 myInterface = (s) -> System.out.println(s);
```

Für beliebige statische public Methoden möglich.

III. Vertiefung Java 1-8

Lambdas: More Syntax Sugar

Sogar für Instanz-Methoden möglich!

```
public interface Test3 { void print(String s); }

SomeClass instance = new SomeClass();
Test3 myInterface =  instance::print;
```

Konstruktoren sind ebenfalls Methoden und können mit **class::new** referenziert werden

III. Vertiefung Java 1-8

Lambdas: More Syntax Sugar

Frage: Was passiert hier?

```
public Interface Test3 { Object whatAmI() }



public void main(String[] args) {
        Test3 myInterface = ContainerImpl::new;
        myInterface.whatAmI();
}
```