Java Enhanced

Java 5 "Tiger"

- Autoboxing und Unboxing
- Erweiterte for-Schleife
- Variable Anzahl Methoden-Argumente und printf
- Enumerationen
- Statische Imports
- Generics
- Streams

Java Erweiterungen

Deklarative Programmierung

- Java 5 Generics, Annotationen
  - Erste Java (1992) Spracherweiterung im Jahr 2004 (nach 12 Jahren!)
- Erweiterungen wurden so vorgenommen, dass bestehende Programme weiter funktionieren
- Aber: im JDK z.T. jetzt neue und alte Variante ("deprecated") nebeneinander vorhanden

Parallele Programmierung

- Java 6 keine Spracherweiterungen
- Java 7 parallele Verarbeitung und kleine Erweiterung bez. Sprache
  - JDK: Thread Pool (+ java.util.concurrent)

Funktionale Programmierung

- Java 8 Lambda Ausdrücke und Stream Bibliothek
  - JDK: parallele Verarbeitung à la Map-Reduce (+ java.util.streams)

Modulare Programmierung

- Java 9 Module -> PSPP Vorlesung
  - und "Modularisierung" des JDKs für z.B. Embedded Devices

Verschiedene neue Konzepte

Autoboxing und Unboxing

- Sehr oft Umwandlung von Wertetyp in Referenztyp und zurück: int<-> Integer
- vereinfacht, automatisiert

```
Integer i  = new Integer(5656);
int j = i.intValue();

Object o = new Integer(4711);
int y = (Integer)o.intValue();

void setValue(Object o){
  …
}
p.setValue(new Integer(34));
```

```
Integer i = 5656;
int j = i;


Object o = 4711;
int y = (Integer)o;


void setValue(Object o) {
   …
}
p.setValue(43);
```

Foreach-Scheife

- Sehr oft Iteration durch alle Elemente einer Collection, Arrays (in aufsteigender Reihenfolge)

```
int[] a = {1,2,4};   int sum=0;
for (int i = 0; i < a.length; i++) {
   sum+=i;
}


List numbers = new ArrayList(a);
// numbers is a list of Numbers.
Iterator it = numbers.iterator();
while (it.hasNext() ) {
   Integer number = (Integer)
   it.next();
   // Do something with the number...
}
```

```
int[] a = {1,2,4};   int sum=0;
for (int i : a) {
   sum +=i;
}


List numbers = new
   ArrayList(a);
for (int n : numbers) {

   // Do something with the
   number...
}
```

Enumerationen

- Typen, deren Wertebereich einer endlichen, kleinen Menge von Werten entspricht
- bisher als int oder static "Object"-Konstante)

```
public class Color {
   public static int Red = 1;
   public static int White = 2;
   public static int Blue = 3;
}
int myColor = Color.Red;
int myColor = 999;     // no compile time error !!!
```

oder:

```
public class Color {
    public static Color Red = new Color(255,0,0);
}
Color c = Color.Red;
```

Enumerationen


```
// The color enumeration
public enum Color {
    Red,
    White,
    Blue
}

Color c = Color.Red;
// Cycling through the values of an enumeration
// (static method values()
for (Color c : Color.values()) {System.out.println(c);}
```

Output:

```
Red
White
Blue
```

Variable Methoden Argumente und printf


```
// An example method that takes a
// variable number of parameters


// Old
int sum(int[] numbers){
   int mysum = 0;
   for (int i=0; i<numbers.length;
   i++)
          mysum += numbers[i].;
   return mysum;
}

// Code fragment that calls
// the sum method
sum(new int[] {12,13,20});
```

```
// An example method that takes a
// variable number of parameters


// New
int sum (int ... numbers) {
   int mysum = 0;
   for (int i: numbers) mysum += i;
   return mysum;
}

// Code fragment that calls the
// sum method
sum(12,13,20);
```

Variable Methoden Argumente und printf

- Es gibt neu eine printf Methode in System.out für die formatierte Ausgabe
- Erster String definiert ein Template mit Platzhaltern
  - %d,%i -> Integer
  - %f ->Float
  - %s ->String
  - %n -> New Line

```
// Pre Java 5 print statement
System.out.println(x + " + " + y + " = " + sum);
```

```
// Using Java 5 printf statement
System.out.printf("%d + %d = %d%n", x, y, sum);
```

http://alvinalexander.com/programming/printf-format-cheat-sheet

Static Imports


```
// x is a number of type double such as 5.345
double y = Math.sin(x);
```

```
With static imports in Java 5, you can ask the Java compiler to import only
a class's static portions, as shown, for example, in the rewritten code
fragment below:

// Import declaration along with the other imports
import static java.lang.Math.sin;

// And then somewhere in the code...
// "floating functions" arise from the ashes!!
double y = sin(x);

// Import all static methods from Math
import static java.lang.Math.*;
```

School of Engineering

Java Generics und Datentypen

outlook

Nutzen von (statischer) Typisierung

outlook

- Datentypen geben den Wertebereich einer Variablen an
- z.B. short -> Ganzzahlen [$-2^{15}..2^{15}-1$], int -> Ganzzahlen [$-2^{31} .. 2^{31}-1$]

Bei Variablen und Parametern sollte der Datentyp so einschränkend (i.e.strikt) wie möglich definiert werden -> *Object* oder *var* Typ vermeiden

- Wichtig um Fehler frühzeitig d.h. bereits bei der Übersetzung zu erkennen

- Bei Methoden bestimmen sie die formal (durch Compiler) überprüften Vorbedingungen:

Frage: darf ich die Wurzelfunktion einfach durch eine erweiterte mit komplexen Definitions- und Wertebereich austauschen?

- {Pre} T_{out} = foo(T_{in} a) {Post}
- z.B. {a >= 0 $\wedge$ a $\in$ R} b = $\sqrt{a}$ {b >= 0 $\wedge$ b $\in$ R}

Nutzen: Vererbung und neue Versionen

outlook

- Liskovsches Substitutionsprinzip
  - Eines der SOLID Kriterien

gilt auch für neue Versionen einer Klasse/Methode

> Ein Programm, das Objekte einer Basisklasse T verwendet, auch mit Objekten der davon abgeleiteten Klasse S korrekt funktionieren muss, ohne dabei das Programm zu verändern.

- Um dies einzuhalten, muss *jede Methode,* die eine der Basisklasse überschreibt folgende Bedingungen einhalten:
  - Für die überschneidenden Werte-/Definitionsbereiche müssen sie sich **gleich** verhalten
  - Der Definitionsbereich der Aufrufparameter darf nur erweitert werden: man sagt **"abschwächen"**
  - Der Wertebereich der Rückgabe darf nur verkleinert werden: man sagt **"verstärken"**

… Nutzen: Vererbung und neue Versionen

outlook

```
public int foo (short x, @NotNull String s) throws AnyException;
```

Compiler hilft bei der Fehlererkennung: Compilerfehler sind "gute" Fehler: schnell gefunden, schnell korrigiert

Datentyp definiert "Menge von gültigen Werten"

Vorbedingung (Precondition)

- die Anzahl und Wertebereiche der Parameter, die durch Datentyp formal (von Compiler überprüfbar) bestimmt sind → Clean Code: so exakt bzw. einschränkend wie möglich (JavaScript, Basic, Python ☹)
- Weitere formale Einschränkungen z.B. @NotNull
- Weitere informelle Einschränkungen des Wertebereichs im Kommentar

Vorsicht mit "Spezialwerten" z.B. null, 31.12.1999 oder 42: dürfen nicht Teil des Rückgabe-Wertebereichs sein/werden.

Nachbedingung (Postcondition)

- die Wertebereiche der Rückgabeparameter
- Fehlerfälle: formal durch checked Exceptions oder Spezialwert signalisiert, informell durch Kommentar

Programme werden sehr viel häufiger erweitert als neu geschrieben

Implementationen von neuen Versionen und überschreibe der Methoden dürfen

- Vorbedingung *abschwächen* i.e. Definitionsbereich erweitern
- Nachbedingung *verstärken* i.e. Wertebereich verkleinern

☹ https://bernat.tech/posts/the-state-of-type-hints-in-python/

Anwendung von Java Generics

Java Generics im Collection-Framework

- Bisher:

```
List list = new LinkedList();
list.add(new Integer(99));
list.add(new String("a"));
Integer i = (Integer)list.get(1); -->Runtime-Error
```

- Gefährlich:
  - kein Schutz gegen Einfügen von Elementen vom "falschen" Typ
  - ClassCastException zur Laufzeit möglich bei falschem Typcast

- Lösung:

  try-catch, Befüllung der List absichern oder sonstige Klimmzüge

- Wünschenswert: Collection Klassen spezialisiert auf bestimmte Elementtypen

Java Generics im Collection-Framework

- Java 5

```
List<Integer> list = new LinkedList<Integer>();
list.add(new Integer(99)); // oder dank Boxing: list.add(99);
Integer i = list.get(0);   // oder dank Unboxing: int i = list.get(0);
list.add(new Double(3.1415)); -->Compile-Error
```

- Vorüberlegung über den Datentyp, den die Collection aufnehmen soll.

- Hier: Der *Typparameter* ist vom Typ Integer.
  - erhöht die Aussagekraft und erleichtert Verständnis des Codes
  - Einfügen von anderen Typen werden zur **Compile-Zeit** schon abgelehnt
  - Cast beim Lesen kann entfallen, da garantiert keine anderen Typen enthalten sind.

Java Generics im Collection-Framework


```
List<Integer> list = new LinkedList<Integer>();
list.add(99);
```

- Generischer Datentypen, `List<Integer>, LinkedList<Integer>`
- Duden: Generisch = "Die Gattung betreffend", "Gattungs..."
- Man sieht auch "parametrisierter Datentyp" oder "parametrischer Datentyp"

- Regeln sind sehr restriktiv, z.B.

  `list.add(new Object());`

  ergibt Compile-Error
- Boxing wird unterstützt, d.h. int <-> Integer

Java Generics - Vorteile

- Erweiterung des Java 5 Collection Frameworks
  - erhöht die Effizienz von Collections, macht sie vielseitiger einsetzbar
  - macht deren Einsatz komfortabler, sicherer und aussagekräftiger
  - senkt die Gefahr von Fehlern bei Typprüfungen zur Laufzeit

- Erleichtert so die Implementierung von Datenstrukturen wie Containern
- Möglichkeit zur Erstellung von eigenen Klassenschablonen und Methodenschablonen
- Erleichtert das Schreiben von Algorithmen, die mit verschiedenen Datentypen funktionieren.

Entwicklung von generischen Klassen

Generizität

- Datentypen wurden in die Programmiersprachen eingeführt, um die Anwendung von Programmiersprachen sicherer zu machen.
  - Assembler und frühe C Versionen kannten keine Datentypen-> oft Quelle von Fehlern
- Mit dem Datentyp wird die Menge der Operationen (Operatoren), die auf Werte dieses D.T. angewandt werden können, eingeschränkt (constraint).
  - z.B. 6 * 8 aber nicht "hallo" * "world" oder int i = "hallo"
- Es soll aber trotzdem möglich sein, eine Datenstruktur / einen Algorithmus auf Werte verschiedener Datentypen anzuwenden
  - z.B. Datenstrukturen: Liste, Stack, Queue
  - z.B. Algorithmen: Sortieren

Einen Algorithmus, der auf Werte von unterschiedlichen Datentypen angewandt werden kann, nennt man *generisch.*

Generizität bisher (bis Java 1.4)

School of Engineering

- Generizität erreicht durch:
- 1) überladen von Methoden

```
int max(int i, int j);
double max(double d, double e);
```

- 2) Object als Parameter (als Oberklasse aller Klassen)

```
class Box {
    private Object val;
    void setValue(Object val ) {
        this.val = val;
     }
    Object getValue() {
        return val;
     }
}
intBox.setValue(new Integer(32));
int i = (Integer)intBox.getValue();
```

© Java ist auch eine Insel
Christian Ullenboom

- beim Lesen muss ein Cast zum gewünschten Typ durchgeführt werden
- Es können Werte von beliebigen D.T. eingefügt werden obwohl das u.U. keinen Sinn macht: Laufzeitfehler beim Lesen (TypeCastException)

Generizität bisher (bis Java 1.4)

- 3) für jeden Datentyp eine eigene Klasse

Container für eine einfache Zahl (Datentyp int)

```
class IntBox {
    private int val;
    void setValue( int val ) {
              this.val = val;
     }
    int getValue() {
             return val;
    }
}
```

Das Gleiche für String, Integer, Float, Double etc.

```
class StringBox {
    private String val;
    void setValue(String val ) {
        this.val = val;
    }
    String getValue() {
        return val;
}
```

© Java ist auch eine Insel
Christian Ullenboom

Generische Klassen (ab Java 5)

- für den Typ wird lediglich ein Platzhalter, z.B. T, E, V eingesetzt
- der Typ kann später bei der Instanzierung (Variablendeklaration) festgelegt werden: Konkretisierung des Typs

```
class Box<T> {
    private T val;

    void setValue(T val ) {
        this.val = val;
     }

    T getValue() {
        return val;
     }
}
```

Platzhalter durch < > gekennzeichnet

© Java ist auch eine Insel
Christian Ullenboom

- Deklarationen:
  - Box<**String**> box = **new** Box<**String**>();
  - Box<**Integer**> box = **new** Box<Integer>();
  - Box<**Point**> box = **new** Box<**Point**>();

Bei der Instanzierung wird der Typ konkretisiert

- Einschränkung: für T nur Referenztypen erlaubt, keine einfachen Datentypen

Verwendung von generischen Typen

School of Engineering

- Es sind keine Typenumwandlungen (Casts) notwendig
- Fehler werden während der Übersetzung erkannt.

```
Box<String> box = new Box<String)();
box.setValue("hallo");
String x = box.getValue();
```

Konkretiriserung dss Platzhaltertyp s

- funktioniert auch mit einfachen Typen dank Boxing

```
Box<Integer> box = new Box<Integer>();
box.setValue(42);
int x = box.getValue();
box.setValue("hallo"); // Compiler Fehler
```

automatisches Boxing

Generische Methoden

Methoden mit Typparameter

"<T>" vor dem Rückgabetyp

```
static <T> void foo(T arg) {

}}
foo(4);
```

- Der konkrete Typ muss nicht angegeben sondern wird anhand der Parameter hergeleitet (Type Inference)

```
static <T> T bar(T t) {
   return t;
}}
int i = bar(4);
int i = bar(4.3); --> Compile-Error
```

- Generische Methoden können auch in nicht-generischen Klassen verwendet werden.

Subtyping von generischen Klassen

Erben bei generischen Klassen

- Es kann auf drei Arten von generischen Klassen geerbt werden
- Die erbende Klasse ist generisch, geerbte nicht

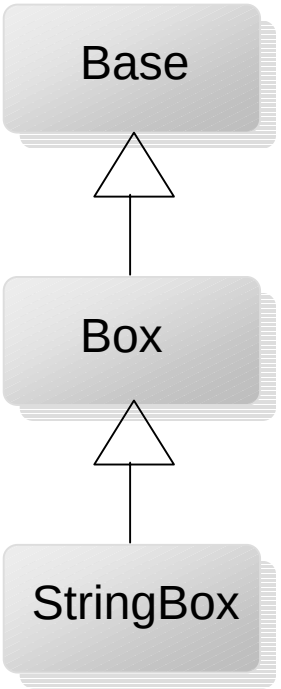
```
class Box<E> extends Base
```

- Die erbende Klasse bleibt weiterhin generisch

```
class Box<E> extends Base<E>
```

- Die erbende Klasse konkretisiert den Typ

```
class StringBox extends Box<String>
```




```
Base base;
Box box;
StringBox sbox;
base = box;  //ok
box = sbox;  //ok
```

Generische Interfaces

```
public interface List<E> {
        public void add(E e);
        public E get(int i);
        ...
}
```

```
public class LinkedList <E> implements List<E> {
        private E first= null;

        public void add(E e){
           ...
        }
        ...
}
```

```
List<Integer> list = new LinkedList<Integer>();
```

Mehrere generische Typen

- Mehrere Platzhaltertypen werden einfach durch "," abgetrennt

```
public interface Map <K,V> {
        public void put(K key, V value);
        public V get(K key);
        ...
}
```

im JDK Object wegen Abwärtskompatibilät

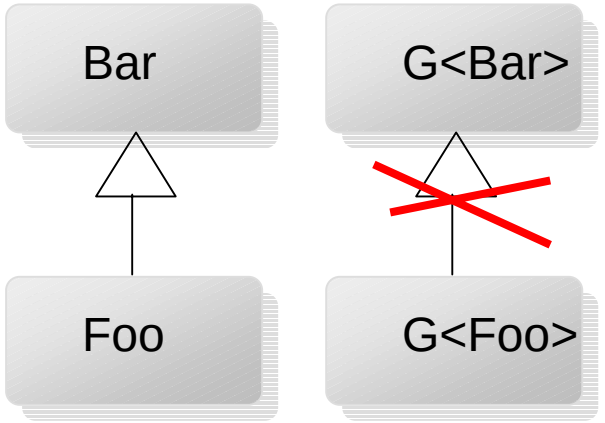
Generics und Subtyping

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```
ok?

nein! Compile time error.

- Problem

```
lo.add(new Integer(4));  // unsafe
String s = ls.get(0);  // run time error
```



Wenn Foo Oberklasse Bar ist und G eine generische Typendeklaration dann ist **G<Foo> keine Oberklasse von G<Bar>.**

Wildcards

Klassen mit Wildcards

- Ein bewusstes "Vergessen" der Typinformationen lässt sich durch das Wildcard Zeichen '?' erreichen.
- Es erlaubt, verschiedene Unterklassen zusammenzuführen.
- Wildcards dienen dazu, unterschiedliche parameterisierte Typen zuweisbar und damit in einer Methode benutzbar zu machen.

Nur lokale Variable und Methoden-Parameter können mit Wildcards benutzt werden!

```
Box<Object> b;
Box<Integer> bI = new Box<Integer>();
Box<Double> bD = new Box<Double>();
b = bI;    // --> Compile-Error
b = bD;  // --> Compile-Error
```

- Variablendeklaration mit Wildcards

```
Box<?> bw;
bw = bL;  // ok
bw = bI;   // ok
bw = b;   //  ok
```

Methoden mit Wildcards

- Eine Methode die alle enthaltenen Elemente einer Collection ausgibt:

früher:

```
public void printCollection(Collection c) {
    iterator iter = c.iterator();
    while(iter.hasNext()) {
        System.out.println(iter.next());
    }
}
```

Methoden mit Wildcards

- Eine Methode die alle enthaltenen Elemente einer Collection ausgibt:
- naiver Ansatz

Java 5

```
public void printCollection(Collection<Object> c) {
    for(Object e:c) {
        System.out.println(e);
    }
}
```

- Funktioniert aber nur mit Collections mit dem Parametertyp <Object>
- List<String> kann ja nicht List<Object> zugewiesen werden

Methoden mit Wildcards

- Eine Methode die alle enthaltenen Elemente einer Collection ausgibt:

Java 5

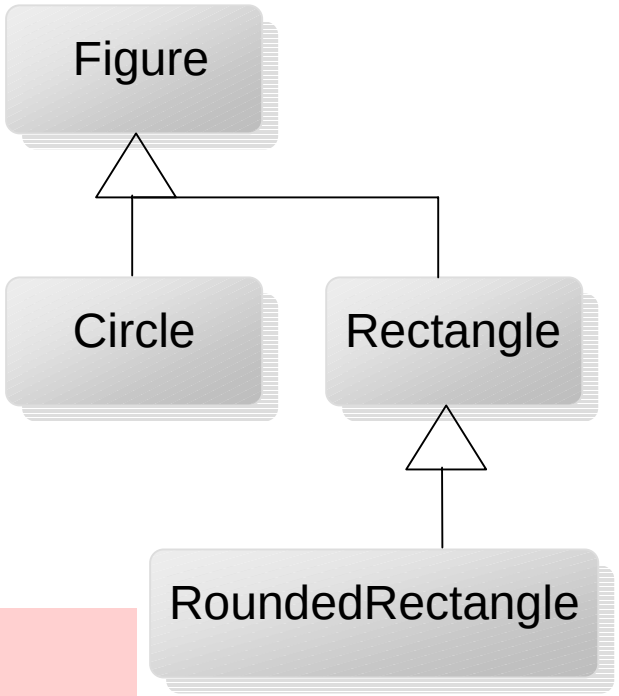
```
public void printCollection(Collection<?> c) {
    for(Object e:c) {
        System.out.println(e);
    }
}
```

- **Collection<?>** ist der Oberklasse aller Collections

Bounds

Generics, Bounded Wildcards

```
abstract class Figure {
    public abstract void draw();
}
class Circle extends Figure {
    public void draw() {};
}
class Rectangle extends Figure {
    public void draw() {}
}
class RoundedRectangle extends Rectangle
{
    public void draw() {}
}
```




```
public void drawAll(List<Figure> figures) {
        for (Figure f : figures) f.draw();
}

List<Circle> lc = new LinkedList<Circle>();
drawAll(lc);  // ok ?
```

nein! Compile time error.

Generics, Upper Bounded Wildcards

```
 public void drawAll(List<?> figures) {
        for (Figure f : figures) f.draw();
}

List<Circle> lc = new LinkedList<Circle>();
drawAll(lc);  // ok ?
```

nein! Compile time error.:
draw Methode nicht bekannt

- Problem List<Figure> ist zu restriktiv List<?> zu offen (keine Information über Typ)
- Wir wollen ausdrücken: dass der Listentyp eine **Oberklasse von Figure** ist
- **Upper Bound Wildcard**

```
 public void drawAll (List<? extends Figure> figures) {
        for (Figure f : figures) f.draw();
}

List<Circle> lc = new ArrayList<Circle>();
drawAll(lc);
```

Generics, Lower Bounded Wildcards

outlook

- Methode die ein RoundedRectangle zu einer generischen Liste hinzufügt

```
public void addRectangle (List<? extends Figure> rects) {
        rects.add (new RoundedRectangle());   // OK ?
}
List<Figure> lf = new ArrayList<Figure>();
addRectangle(lf);
```

Nein: Listentype könnte ein Circle sein und wir fügen ein Rechteck dazu

- Wir wollen ausdrücken, dass der Listentyp eine Basisklasse von RoundedRectangle ist

```
public void addRectangle (List<? super RoundedRectangle> rects) {
        rects.add (new RoundedRectangle);  // OK?
}
```

Ja: Liste von Figures, Rectangle und RoundedRectangle erlaubt

Bounded Wildcards bei Methodenparameter

School of Engineering

- Abhängigkeit zwischen zwei Typen in Kombination mit Wildcards

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src) { ... }
}
```

- auch möglich

```
class Collections {
   public static <T, S extends T> void copy(List<T> dest, List<S> src) {...}
}
```

Bounded WildCards bei Rückgabetyp

School of Engineering

- Upper bound wildcards (**extends**) erlauben Funktionen eine bounded parametrisierten Typ zurückgeben.

```
public List<? extends Figure>  getFigures () { }
```

Raw Types & Type-Erasure

Raw Type

- Wenn man eine Variable ohne "<>" (wie 1.4) deklariert, dann spricht man von einem Raw Type

- Raw Types und Generic Types sind Zuweisungskompatibel; die statische Typensicherheit geht aber verloren; es werden deshalb vom Compiler Warnungen generiert

- Mittels **SuppressWarnings("unchecked")** lassen sich diese ausschalten

```
Box<String> bs = new Box<String>
Box raw;   //Raw Type

raw = bs;  // ok
@SuppressWarnings("unchecked")
bs = raw; // unchecked Warning
bs = (Box<String>)raw;    // unchecked Warning
```

Type Erasures

School of Engineering

- Java entfernt die generische Typeninformation vollständig zur Laufzeit
  - Grund: man wollte **neuer Code** auch mit **alter JRE** noch laufen
    - *wurde auch beim Übergang C -> C++ so umgesetzt*
  - i.d.R. lediglich verlangt dass **alter Code** auf **neuer JRE** noch lauffähig
    - *für Python 2.7 -> 3.x nicht mal das erfüllt :-(*

- Entscheid des Java Design Teams zur Implementierung der Aufwärtskompatibilität
- Aus `Box<T>` wird zur Laufzeit `Box<Object>` und

Lösungen dazu auf den nächsten Folien

- Störend: Daraus ergeben sich Einschränkungen
  - **Typenprüfung** nich möglich: `if (e instanceof List<Integer) …`
  - **Cast** sind nicht überprüfbar -> Warnung `E e = (E)o;`
  - kein **Instanzierung** möglich `E e = new E();`
  - keine Erstellung von **Arrays** von `E[] a = new E[10];`

Type Erasures und Typenprüfung und Cast

- Beim Ablauf des Programms kann nicht mehr von z.B. `LinkedList<String>` und `LinkedList<Integer>` unterschieden werden.
  Beide habe zur Laufzeit den Typ `LinkedList<Object>`

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

Java liefert True zurück

- Auch Casts funktionieren nicht wirklich

```
<T> T badCast(T t, Object o) { return (T) o; // unchecked warning
```

- Lösung: Laufzeitsystem überprüft und löst Fehler aus
  - ist unschön, da dadurch statische Typsicherheit verloren geht.

Type Erasures und Erzeugung von Instanzen

School of Engineering

- Erzeugung von Objekten von Platzhaltertyp geht nicht

```
E e = new E(); // not allowed
```

- Lösung: man gebe beim Aufruf noch die Class<T> mit
- dann kann man mit newInstance ein neues Objekt erzeugen

```
<T> T foo(Class<T> clazz, String s) {
    return (T)clazz.newInstance();
}

String s = foo(String.class,"hallo");
```

Type Erasures und Erzeugung von Arrays

School of Engineering

- Eine Array von generischen Typen kann nicht angelegt werden:
- z.B. Array von T

```
T[] a = new T[10]; // not allowed
```

- Lösung man verwendet Object Array und führt Cast durch

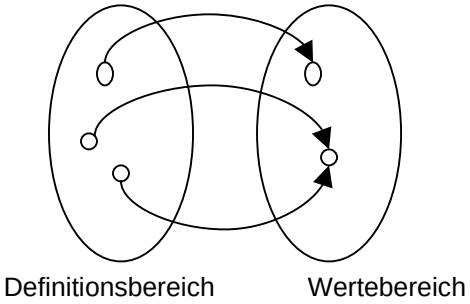
```
T[] a = (T[])new Object[10]; // ok, array of unbounded type
```

- Eine Reihe von weiteren Problemen wegen dem Type Erasure,
  - z.B. bei Reflection
- Vermutlich würden die Java Entwickler das heute nicht mehr so machen

Funktionale Programmierung und Lambda Ausdrücke

λ Calculus

- Funktionen sind fundamental in Mathematik und Informatik
- in der Mathematik
  - Werte werden vom Definitionsbereich in den Wertebereich abgebildet
  - f x -> y



in Informatik übliche Notation

- Beispiel
  - I (x) = x -> x
  - Sqr(x) = x -> x^2
- Wenn man keinen Namen "erfinden" will, nennt man sie einfach Lambda λ
  - λx.x^2

in Mathematik (Lambda Kalkül) übliche Notation

- λ Kalkül: Grundlage der math. Theorie bez. Berechenbarkeit (Church 1940)
  - z.B. Higher Order Functions = Funktionen, die Funktionen als Argument haben
- In der Informatik
  - z.B. Datenstrom kann mittels Lambda Ausdruck in andern umgewandelt werden

Klassen, Annonyme Klassen, Lambda

- Comparator als separate (innere) Klasse

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
class TrimmingComparator implements Comparator<String> {
    @Override public int compare( String s1, String s2 ) {
        return s1.trim().compareTo( s2.trim() );
     }
}
...
Arrays.sort( words, new TrimmingComparator() );
System.out.println(Arrays.toString(words) );
```

- Als anonyme Klasse (inline)

```
Arrays.sort(words, new Comparator<String>() {
   @Override public int compare(String s1, String s2 ) {
      return s1.trim().compareTo( s2.trim() );
   }});
System.out.println(Arrays.toString(words) );
```

- Als Lambda Ausdruck

```
Arrays.sort(words,(s1, s2) -> s1.trim().compareTo(s2.trim()));
System.out.println( Arrays.toString( words ) );
```

Beispiel für Lambda Expressions

- Definition eines Function Interfaces

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void apply(int d);
}
```

In Java werde Funktionen durch Interfaces mit i.d.R. einer Methode beschrieben

- Methode die Funktion als Parameter erlaubt

```
void forEach(int[] a, MyFunctionalInterface func) {
    for (int i = 0; i < a.length; i++) {
        func.apply(a[i]);
    }
}
```

- Übergabe eines Lambda Ausdrucks beim Aufruf

```
int[] a = {2,4,5};
forEach(a,x -> System.out.println(x));
```

Lambda Expressions - Details

- **Allgemeine Form**

```
Parameters  "->"  (Expr | Block)
```

Lambdas können 0, 1 oder n Parameter haben

```
()      -> ...
x       -> ...
(x, y)    -> ...
(x, y, z) -> ...
```

```
// no parameters
// 1 parameter
// 2 parameters
// 3 parameters
```

Parameter können typisiert sein

```
(int x) -> ...
(String s, int x) -> ...
```

```
// must be in brackets although just 1
parameter
```

Macht man aber i.d.R. nicht, sondern nutzt Typeninferenz aus

```
@FunctionalInterface
public interface Func {
    boolean apply(int x, int y);
}
```

```
Func f =  (x, y) -> x > y;
```

must be *int* must be *boolean*

… Lambda Expressions - Details

```
Parameters  "->"  (Expr | Block)
```

Rechte Seite ist i.d.R. ein Ausdruck

```
x ->  x * x                 // returns x * x
(x, y) ->  x + y            // returns x + y
```

Rechte Seite kann auch ein Block sein

```
n  -> {int sum = 0;
       for (int i = 1; i <= n; i++) sum += i;
       return sum;
      }
```

Rechte Seite kann auch ein Block ohne Rückgabe Wert sein

```
x -> { System.out.Println(x); };
```

Rechte Seite kann auf lokale Variablen zugreifen (Erstellungskontext, Closure)
Schreiboperationen müssen Threadsafe sein -> Atomic Typen verwenden

```
final k = 42;
AtomicInteger sum = new AtomicInteger();

forEach(a,x -> System.out.println(k + x));
forEach(a,x -> sum.addAndGet(x));
```

https://www.baeldung.com/java-8-lambda-expressions-tips

Vordefinierte (generische) Funktionsinterfaces

Function Interface in java.util.function

School of Engineering

- Interface mit einem Argument und einem Rückgabewert

```
public interface Function<T, R> {
      R apply(T t);
}
```

- Beispiel
  - eine Wertetransformation: Bereich [0..1[zu [1..6]

```
import java.util.function.*;
...
Function<Double, Integer> func = x -> (int) (x * 6 + 1);
System.out.println(func.apply(.5));
...
```

- Es gibt auch **BiFunction** mit 2 Parametern, für mehr verwendet man **Currying**

https://www.baeldung.com/java-currying

outlook

Predicate Interface

- Interface mit einem Argument und boolean Rückgabewert

```
public interface Predicate<T> {
      boolean test(T t);
}
```

- Beispiel

```
import java.util.function.*;
...
Predicate<Double> predicate = x -> x > .5;
if (predicate.test(.5)) System.out.println("ok");
...
```

- negate() liefert Negation des Prädikats

```
Predicate<Integer> isEven = i -> i % 2 == 0;
Predicate<Integer> isOdd = isEven.negate();
```

Supplier, Consumer Interface

School of Engineering

- Interface mit *keinem* Argument und einem Rückgabewert

```
public interface Supplier<T>
     T get();
}
```

- Interface mit einem Argument und *keinem* Rückgabewert

```
public interface Consumer<T> {
      void accept(T t);
}
```

- Beispiel

```
Supplier<Double> supplier = () -> Math.random();
Consumer consumer = x -> System.out.println(x);

double d = supplier.get();
d = func.apply(d);
consumer.accept(d);
```

- oder besser kaskadiert

```
consumer.accept(func.apply(supplier.get()));
```

Method Reference

- Vereinfachung für den Fall, dass der Lambda Ausdruck lediglich eine vordefinierte Methode mit Parameter aufruft

```
x -> System.out.println(x)
```

- kann einfacher geschrieben werden als

```
System.out::println
```

- Weitere Beispiele

Syntax	as lambda expression	Reference to
`Class::staticMethod`	`(args) -> Class.staticMethod(args)`	a static method
`obj::instanceMethod`	`(args) -> obj.instanceMethod(args)`	an instance method of a particular object
`Class::instanceMethod`	`(args) -> obj.instanceMethod(args)`	an instance method of an arbitrary object of a particular type
`Class::new`	`(args) -> new ClassName(args)`	a constructor

https://dzone.com/articles/methodreference

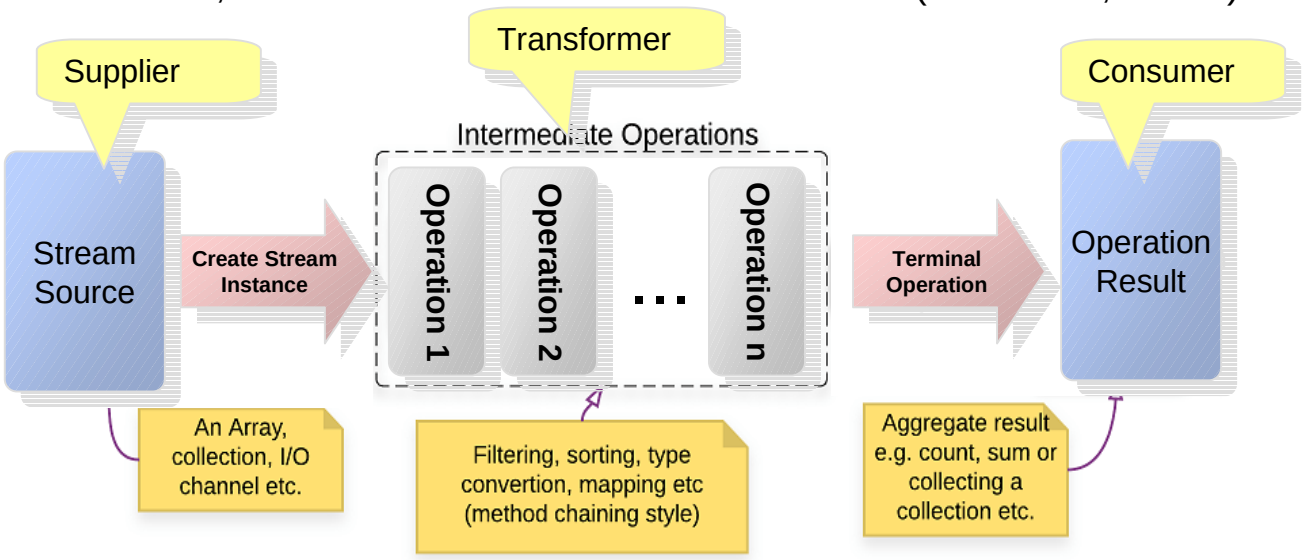
Streams

Stream Datenstruktur

- Datenstruktur die einen Strom von Daten (Referenzen) repräsentiert
- Implementieren das Interface **java.util.stream.Stream<T>**
  - NICHT kompatibel zu Input- und OutputStream von java.io
  - Collections sind ebenfalls KEINE Ströme, bieten aber Methoden an solche zu erzeugen
- Ströme von Daten, Eigenschaften:
  - Die Daten des Stroms selber sind **unveränderbar**
  - Kein indexierter Zugriff möglich
  - Es können **verkettete Operationen** auf Daten nacheinander (oder parallel) ausgeführt werden
    - *Resultat der Operation kann ein neuer Strom sein*
  - Eine Vielzahl von Operationen um
    - *Ströme zu **erzeugen***
    - *Ströme zu **verändern***
    - *Ströme zu **konsumieren** (i.e. terminierende Operation auf Strom)*
    - *Operationen sind i.d.R. lazy ausgewertet (erst wenn Daten benötigt)*
  - Viele Operationen können durch **Lambda Ausdrücke** parametrisiert sein

Stream Operationen

- Erzeugen einen Datenstrom anhand einer Quelle
  - z.B. array, Collection, generator function, I/O Kanal
- Intermediate Operations zum umwandlen in anderen Datenstrom
  - filter,sort, transform (i.e. map)
- Terminale Operation das ein Resultat erzeugt
  - z.B. zählen, aufsummieren oder eine neue Collection (i.e. reduce, collect)



https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-api-intro.html

Erzeuge Streams

Folgende Varianten um Streams zu erzeugen

- Direkte Angabe der Werte

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
```

- Aus Array

```
Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4,5,6,7,8,9} );
Arrays.stream(new Integer[]{1,2,3,4,5,6,7,8,9} );
```

- Aus Liste

```
Stream<Integer> stream = list.stream();
```

- Generator Funktion: erzeugt Elemente mittels **Supplier**

```
Stream<Double> stream = Stream.generate(() -> Math.random());
```

- Aus File

java.nio.file.*

```
Stream<String> stream = Files.lines(Paths.get(fileName));
```

https://howtodoinjava.com/java8/java-streams-by-examples/

Verarbeite Streams (intermediate Operationen)

Resultat ist i.d.R. wieder ein Stream

- **filter()** filtert Elemente eines Streams anhand **Predicat**

```
stream.filter((s) -> s.startsWith("A"))
```

- **map()** konvertiert Elemente eines Streams mittels **Function**

```
Stream<String> words = Stream.of("a","b","c");
words.map(s -> s.toUpperCase());
```

- **sorted()** erzeugt sortierten Stream

```
words.sorted();
```

- **skip(n)** überspringe nächste n Elemente

```
words.skip(1);
```

- **limit(n)** n Elemente des Streams

```
words.limit(5);
```

Konsumiere Strom (terminal Operation)

School of Engineering

- Erzeugt Resultat eines definierten Typs
  - Die eigentliche Verarbeitung des Stroms wird erst jetzt angestossen (Lazy Auswertungskonzept)
  - nachher kann **keine weitere Operation** auf diesen Strom mehr angestossen werden
- **forEach()** iteriere über alle Elemente des Stroms und wende **Consumer** an

```
words.forEach(System.out::println);
```

- **collect()** wandelt Stream in Collection um, z.B. Liste

```
List<String> list= words.collect(Collectors.toList());
list.forEach(System.out::println);
```

- **toArray()** wandelt Stream in Array um

```
String[] arr = words.toArray(String[]::new);
Arrays.asList(arr).forEach(System.out::println);
```

- verschiedene weitere Collectors

https://www.baeldung.com/java-8-collectors

outlook

… Konsumiere Strom (terminal Operation)

- **match()** überprüft ob Bedingung erfüllt wird mittels **Predicat**

```
boolean m = words.anyMatch((s) -> s.startsWith("A"));
boolean m = words.allMatch((s) -> s.startsWith("A"));
boolean m = words.noneMatch((s) -> s.startsWith("A"));
```

- **count()** zählt die Elemente

```
int n  = words.count();
```

outlook

- **reduce()** Reduktion von Stream auf einzelnen Wert mittels **Akkumulator**

```
Stream<String> words = Stream.of("a","b","c");
String s = words.reduce("", (s1,s2) -> s1 + "#" + s2);
```

a#b#c

```
Stream<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
int result = numbers.reduce(0, (subtot, elem) -> subtot + elem);
```

21

Initialisierung

Akkumulator

https://www.baeldung.com/java-stream-reduce
https://openbook.rheinwerk-verlag.de/java8/04_011.html

Kaskadierung der Operationen

- Man führt i.d.R. mehrere Stream Operationen hintereinander aus

```
Object[] words = { " ", '3', null, "2", 1, "" };

Arrays.stream( words ) // " ", '3', null, "2", 1, ""
  .filter( Objects::nonNull ) // " ", '3', "2", 1, ""
  .map(Objects::toString ) // " ", "3", "2", "1", ""
  .map(String::trim ) // "", "3", "2", "1", ""
  .filter(s -> ! s.isEmpty() ) // "3", "2", "1"
  .map(Integer::parseInt ) // 3, 2, 1
  .sorted() // 1, 2, 3
  .forEach(System.out::println ); // 1 2 3
```

- Parallele Verarbeitung von Streams

outlook

https://www.baeldung.com/java-when-to-use-parallel-stream

Stream Operationen

Intermediate Operations

- filter()
- map()
- flatMap()
- distinct()
- sorted()
- peek()
- limit()
- skip()

Terminal Operations

- forEach()
- forEachOrdered()
- toArray()
- reduce()
- collect()
- min()
- max()
- count()
- anyMatch()
- allMatch()
- noneMatch()
- findFirst()
- findAny()

https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-cheat-sheet.html

Zusammenfassung

- Verschiedene Erweiterungen
  - Autoboxing und Unboxing
  - Erweiterte for-Schleife
  - Variable Anzahl Methoden-Argumente und printf
  - Enumerationen
  - Statische Imports
- Generics
  - Generische Typen und Methoden erhöhen die Sicherheit, da Typprüfungen zur Laufzeit reduziert werden
  - Viele falsche Typzuweisungen werden vom Compiler erkannt
  - Die Aussagekraft des Quelle wird erhöht
- Lambda Ausdrücke
  - Funktionale Verarbeitung
- Ströme
  - Verarbeitung in Datenströmen

Anhang: Oberklasse aller numerischen Typen

outlook

- `java.lang.Number` ist Oberklasse von Integer, Long, Double, ..

- Somit

```
public List<? extends Number>  getValueList() { }
```

- Eine Liste von numerischen Werten

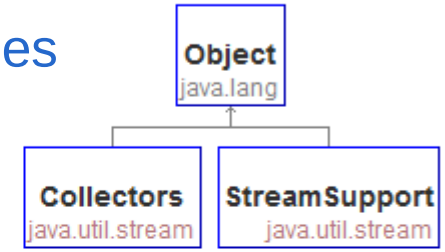
```
class Number {
  double doubleValue()  Returns the value of the specified number as a double.
  float  floatValue()   Returns the value of the specified number as a float.
  int    intValue()     Returns the value of the specified number as an int.
  long   longValue()    Returns the value of the specified number as a long.
  short  shortValue()   Returns the value of the specified number as a short.
}
```

Anhang: Stream Packages und Klassen

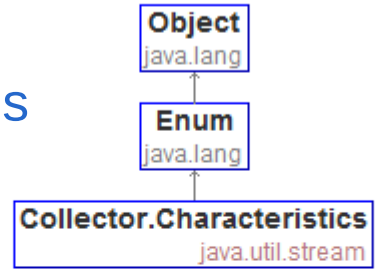
outlook

-

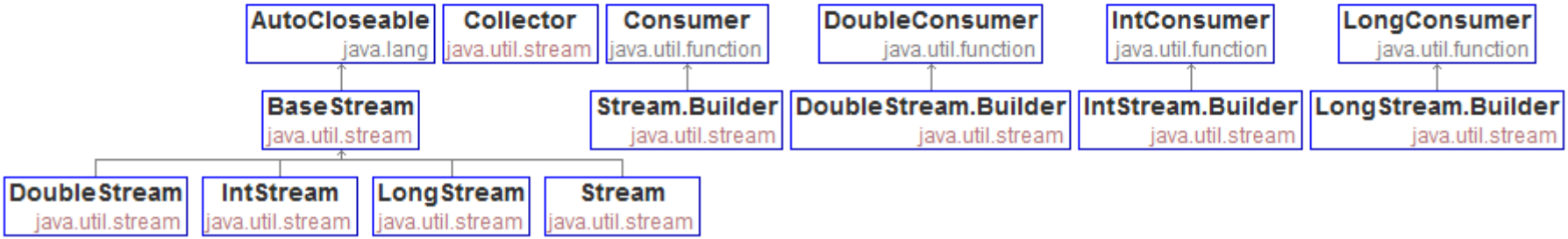
Classes



Enums



Interfaces



Anhang: Exception Handling in λ-Expressions

- Checked Exceptions have to be handled in λ-Expressions

```
files.map(file -> {
        try {return read(file);}
        catch (IOException e) {
            // handle the exception...
               return null;
        }
    })
    .filter(Objects::nonNull)
    .forEach(System.out::println);
```

- Simple Solution: Convert to unchecked Exception

```
@FunctionalInterface
public interface ExFunction<T, R, E extends Exception> {
    R apply(T t) throws Exception;
}
<T, R, E extends Exception> Function<T, R> uncheck(ExFunction<T, R, E> fn) {
    return t -> {
        try {
            return fn.apply(t);
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    };
}
```

```
// Compiler is happy
files.map(uncheck(this::read))
     .filter(Objects::nonNull)
     .forEach(System.out::println);
```

https://belief-driven-design.com/functional-programming-with-java-exception-handling-e69997c11d3/

Anhang: Threads, Events and λ-Expressions

outlook

- If Events are handled as λ-Expressions an UncaughtExceptionHandler may be set

```
public Thread getThreadByName(String threadName) {
    for (Thread t : Thread.getAllStackTraces().keySet()) {
        if (t.getName().equals(threadName)) return t;
     }
     return null;
}

....
button.addActionListener(e -> {VolumeControl.setInput("cd");});
....
getThreadByName("AWT-EventQueue-0").setUncaughtExceptionHandler((th,ex)->{
    JOptionPane.showMessageDialog(null, ex.getMessage(),
    "Error", JOptionPane.ERROR_MESSAGE);
});
```

or use Thread.setDefaultUncaughtExceptionHandler

- Invoke asynchronously in different thread

```
SwingUtilities.invokeLater(()-> {label.setText(count);})
```

- Start Thread

```
new Thread(() -> {...}).start();
```

Beispiele: Imperativer vs Funktionaler Stil

outlook

Internal iteration, using new default method `Iterable#forEach(Consumer<? super T> action)`

```
List<String> list =
 Arrays.asList("Apple", "Orange", "Banana");
for (String s : list) {
    System.out.println(s);
}
```

```
List<String> list =
 Arrays.asList("Apple", "Orange", "Banana");
//using lambda expression
list.forEach(s -> System.out.println(s));
//or using method reference on System.out instance
list.forEach(System.out::println);
```

Counting even numbers in a list, using `Collection#stream()` and `java.util.stream.Stream`

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int count = 0;
for (Integer i : list) {
    if (i % 2 == 0) {
        count++;
    }
}
System.out.println(count);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
long count = list.stream()
                 .filter(i -> i % 2 == 0)
                 .count();
System.out.println(count);
```

Retrieving even number list

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
List<Integer> evenList = new ArrayList<>();
for (Integer i : list) {
    if (i % 2 == 0) {
        evenList.add(i);
    }
}
System.out.println(evenList);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
List<Integer> evenList =
                list.stream()
                    .filter(i -> i % 2 == 0)
                    .collect(Collectors.toList());
System.out.println(evenList);
```

Or if we are only interested in printing:

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
list.stream().filter(i -> i % 2 == 0)
             .forEach(System.out::println);
```

Finding sum of all even numbers

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int sum = 0;
for (Integer i : list) {
    if (i % 2 == 0) {
        sum += i;
    }
}
System.out.println(sum);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int sum = list.stream()
              .filter(i -> i % 2 == 0)
              .mapToInt(Integer:tValue)
              .sum();
System.out.println(sum);
```

Alternatively

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
int sum = list.stream()
              .filter(i -> i % 2 == 0)
              .reduce(0, (i, c) -> i + c);
System.out.println(sum);
```

Finding whether all integers are less than 10 in the list

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
boolean b = true;
for (Integer i : list) {
    if (i >= 10) {
        b = false;
        break;
    }
}
System.out.println(b);
```

```
List<Integer> list =
        Arrays.asList(3, 2, 12, 5, 6, 11, 13);
boolean b = list.stream()
                .allMatch(i -> i < 10);
System.out.println(b);
```

Also look at `Stream#anyMatch(...)` method

Finding all sub-directory names in a directory. Using new static methods, `Arrays#stream(T[] array)`

```
List<String> allDirNames = new ArrayList<>();
for (File file : new File("d:\\").listFiles()) {
    if(file.isDirectory()){
        allDirNames.add(file.getName());
    }
}
System.out.println(allDirNames);
```

```
List<String> allDirNames =
    Arrays.stream(new File("d:\\")
          .listFiles())
          .filter(File::isDirectory)
          .map(File::getName)
          .collect(Collectors.toList());
System.out.println(allDirNames);
```