OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Streams and Lambdas in Java 8

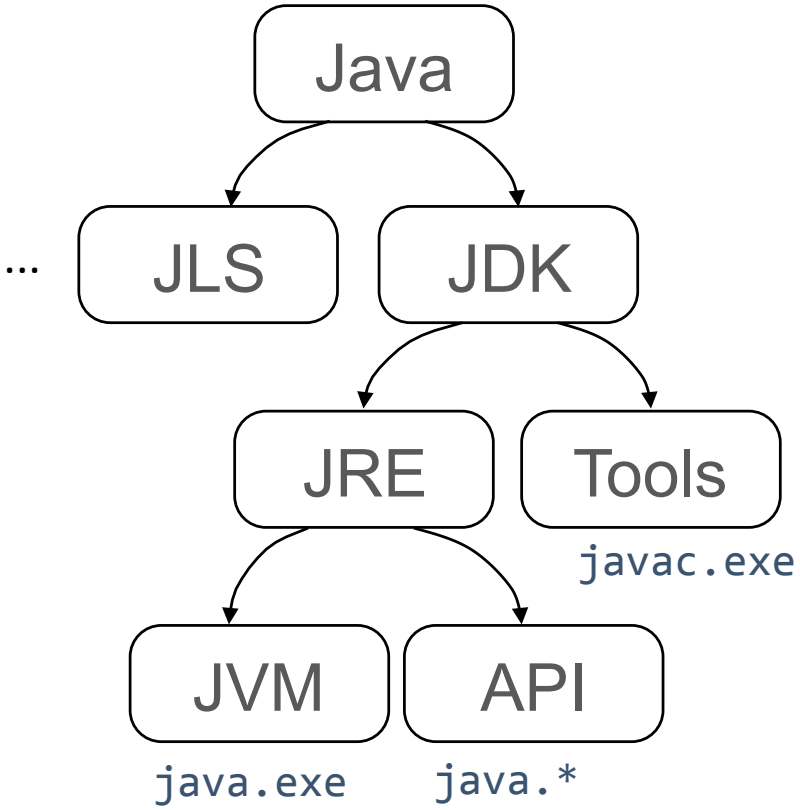
Working Environment

- Integrated Development Environment (IDE)
- JDK 8

Integrated Development Environments

NetBeans	NetBeans by Oracle `https://netbeans.org/downloads/` Any bundle will do current version: 8.1
Eclipse	by Eclipse Foundation (formerly by IBM) `http://www.eclipse.org/downloads/` current version: Eclipse Mars (4.5); with Neon available Choose bundle „...Java EE Developers“ or „...Java Developers“
IDEA	Intellij IDEA by JetBrains (commercial) `https://www.jetbrains.com/idea/` current version: 15 Students can get „Ultimate Edition“ for free See `https://www.jetbrains.com/student/`

JDK 1.8 (JDK 8)

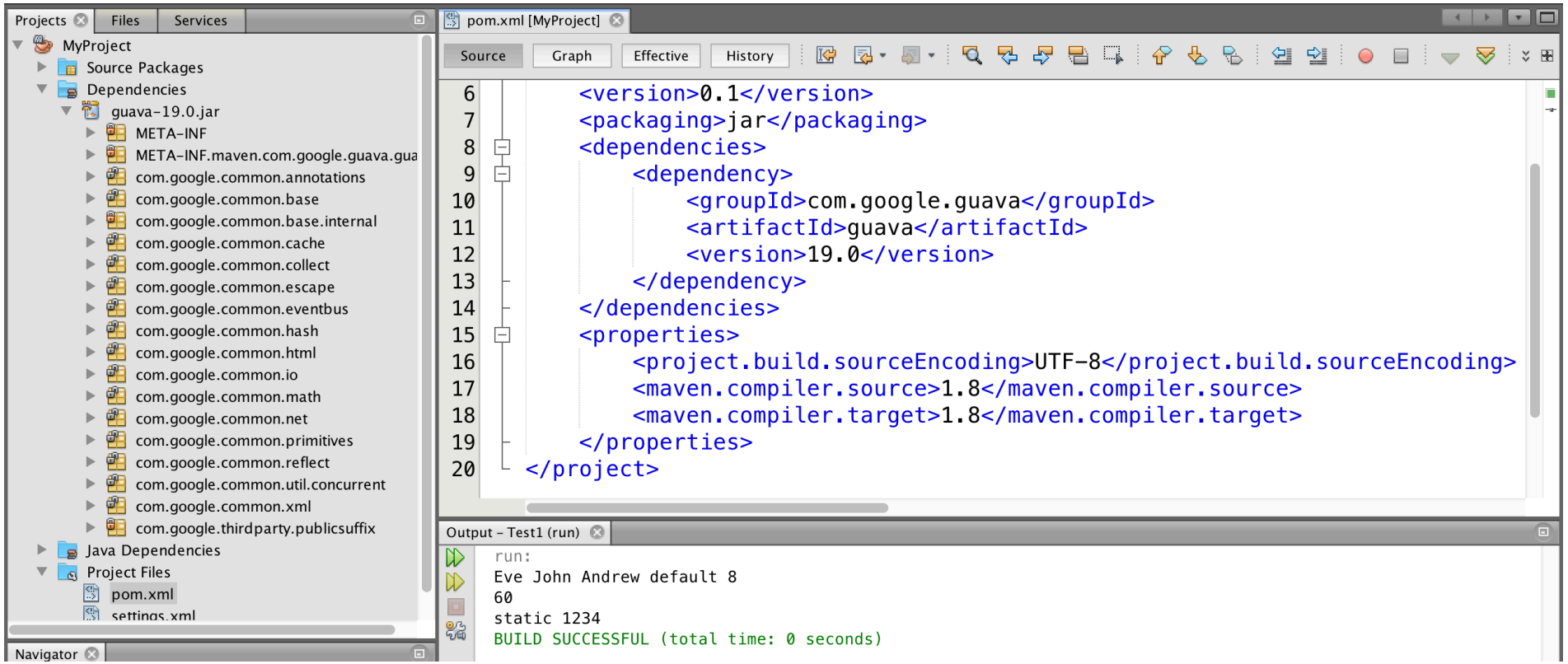



```
$ java -version

java version "1.8.0_65"

Java(TM) SE Runtime
Environment (build 1.8.0_65-
b17)

Java HotSpot(TM) 64-Bit
Server VM (build 25.65-b01,
mixed mode)
```

Dependency Management with Maven

Manage dependencies to 3rd party recursively and have your project build automatically

e.g. use Google Guava Libraries by simply adding one dependency.



Basics Revisited

- Interfaces
- Collections Framework
- Anonymous Inner Classes

Interfaces

"Ordinary" interfaces

Marker interfaces (e.g. `Serializable` or `Runnable`)

Functional interfaces (annotated with `@FunctionalInterface`)

`@interface` classes (Annotations)

New in Java 8:

`static` methods

`default` methods

Default methods[1]

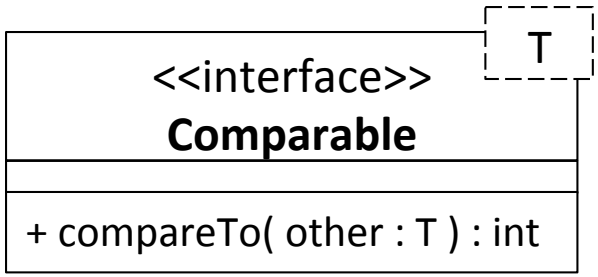
"A default method is a method that is declared in an interface with the `default` modifier; its body is always represented by a block. It provides a default implementation for any class that implements the interface without overriding the method. Default methods are distinct from concrete methods (§8.4.3.1), which are declared in classes."

[1][Gosling 2015, p. 288]

Generics with <>

Generic class:

Definition



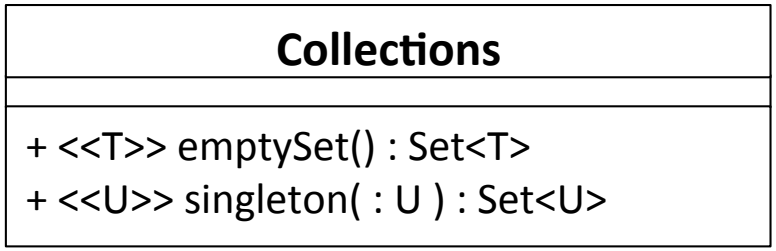

```
public interface Comparable<T> {
   public int compareTo(T other);
}
```

Usage

```
class Student
implements Comparable<Person>
{
   public int compareTo(Person other)
   {
      // ...
   }
}
```

Generic method:

Definition




```
public class Collections {
   public <T> Set<T> singleton(T obj)
   { ... }
}
```

Usage

```
Set<String> name =
    Collections.singleton("Max");

Set<Person> persons =
    Collections.<Person>emptySet();
```

Collections Framework

Two things to remember:

1. There are Lists, Sets, and Maps:

```
List<T>:          ArrayList<T>  or  LinkedList<T>
Set<T>:           TreeSet<T>  or  HashSet<T>
Map<K, V>:        TreeMap<K, V>  or  HashMap<K, V>
```

2. Use “loosly coupled” references:

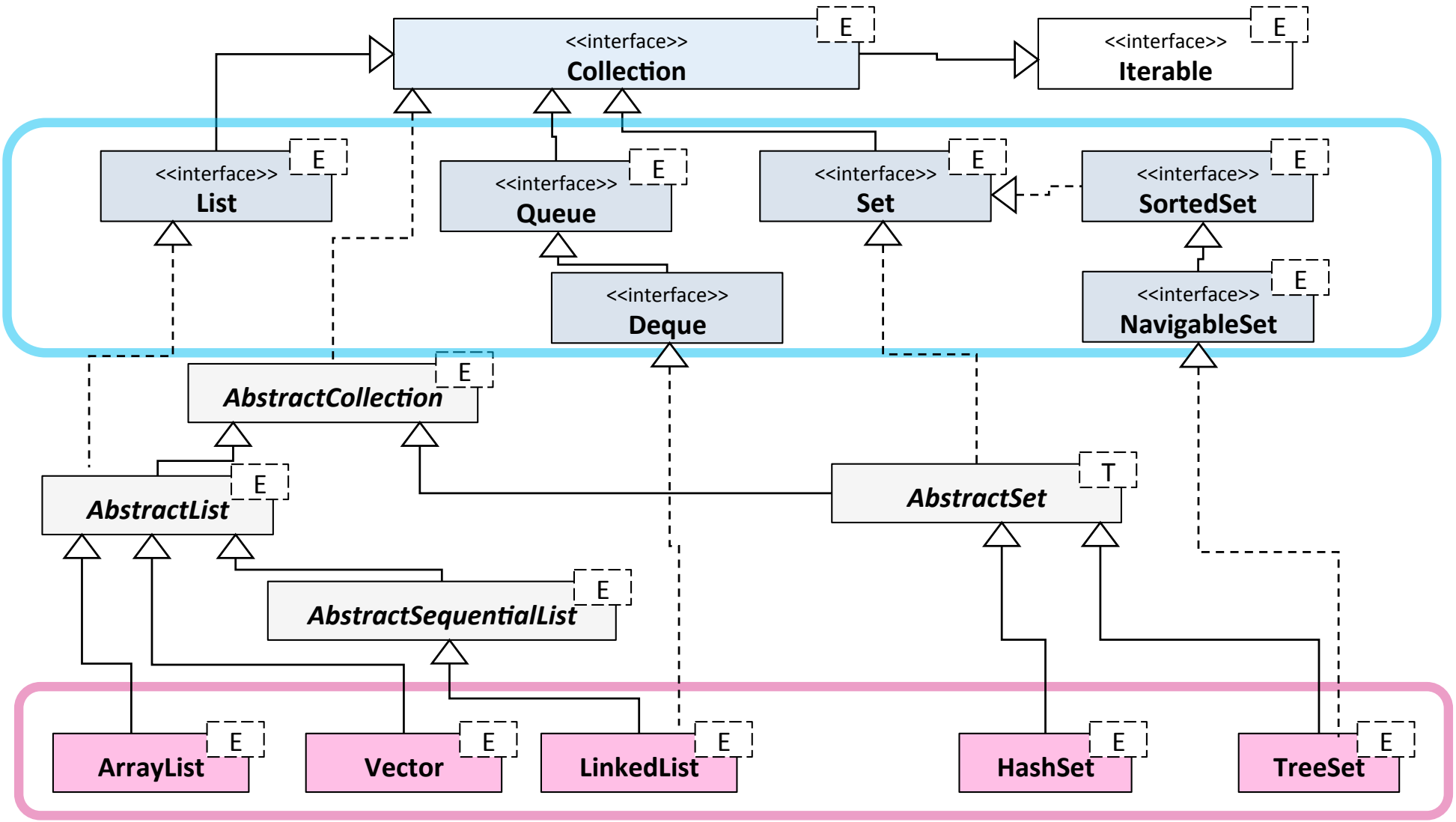
```
List<String> = new ArrayList<>();
```

Collections Framework[1]

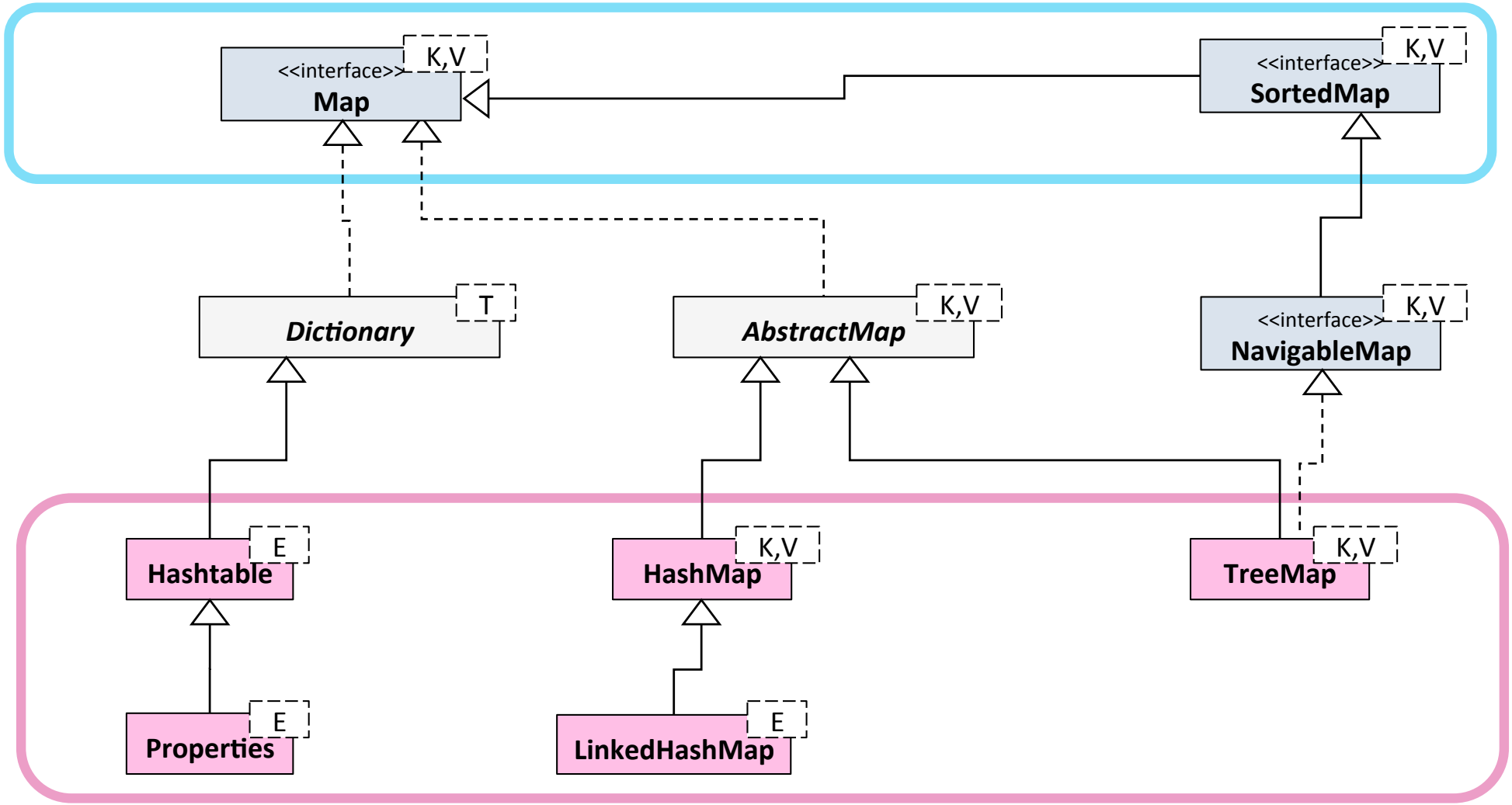
“The collections framework is a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. […]”

[1][Oracle Corp. 2016]

Collection interfaces (abridged)



Map interfaces (abridged)



Utilities in class Collections

```
public static void reverse(List<?> list)

public static <E> Collection<E> checkedCollection(Collection<E> c,
                                                  Class<E> type)

public static <T> List<T> nCopies(int n, T o)

public static int frequency(Collection<?> c, Object o)

public static void shuffle(List<?> list)

public static void rotate(List<?> list, int distance)

public static void reverse(List<?> list)

public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)

public static <T> T min(Collection<? extends T> coll,
                        Comparator<? super T> comp)

// others:

Arrays.asList(Object... o)

Arrays.stream(T[] array)

Stream.of(T[] array)
```

Anonymous Inner Classes

```
public interface Comparator<T> { int compare (T obj1, T obj2); }
```

```
public class Collections {
   public static <T> void sort(List<T> l, Comparator<? super T> c)
   {...}
}
```

```
List<String> names = Arrays.asList("John", "Andrew", "Eve");

Collections.sort(names, new Comparator<String>() {
   @Override public int compare(String s1, String s2) {
      return Integer.compare(s1.length(), s2.length());
   }
});

for(String name : names) System.out.print(name + " ");
// Eve John Andrew
```

Streams

Streams

A stream is "[…] a **sequence of elements** from a **source** that supports **aggregate operations**":[1]

- "**Sequence of elements**: A stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand.
- **Source**: Streams conamounte from a data-providing source such as collections, arrays, or I/O resources.
- **Aggregate operations**: Streams support SQL-like operations and common operations from functional programing languages, such as filter, map, reduce, find, match, sorted, and so on."

[1][Urma (2014a), section "Getting Started With Streams"]

Streams

- are not data structure
- do not contain storage for data
- are “pipelines” for streams of data (i.e. of objects)
- while in the pipeline data undergo transformation (without changing the data structure holding it)
- wrap collections (lists, set, maps)
- read data from it
- work on copies

How do data get into streams?

Streams are mainly generated based on collections:

```
List<String> names = Arrays.asList("John", "George", "Sue");


Stream<String> stream1 = names.stream();

Stream<String> stream2 = Stream.of("Tom", "Rita", "Mae");

Stream<String> stream3;
stream2 = Arrays.stream( new String[]{"Lisa", "Max", "Anna"} );
```

Or with builder pattern:

```
Stream<String> stream4 = Stream.<String>builder()
                              .add("Mike")
                              .add("Sandra").build();
```

How do data get out of streams?

- The Streaming API provides so called “finalizing” methods (i.e. methods that do not return stream objects)

```
forEach
toArray
collect
reduce
min
max
count
anyMatch
noneMatch
findFirst
findAny
```

Streaming example

"Take all names from the stream that start with the letter "J", map the names into capital letters, skip one, and collect them into a new set"

```
List<String> names = Stream.of("John","George","Joe","Sue","James");


Stream<String> stream1 = names.stream();

______ = stream1.filter( ___________ )
                .map( ___________ )
                .skip( ___________ )
                .collect( ___________ );
```

Lambda Expressions and Functional Interfaces

- Lambdas
- Functional Interfaces

Lambdas or Closures

"Lambda" = "closure" = record storing a function (functionality, method) and its environment (but without a class or method name)

Roughly: anonymous method

Lambdas represent source code – not data and not object state!

Syntax:

```
( parameter list ) -> { expression(s) }
```

Examples:

```
(int x, int y) -> { return x + y; }
```

```
(long x) -> { return x * 2; }
```

```
() -> { String msg = "Lambda"; System.out.println(msg); }
```

For details on "functional programming" cf. [Huges 1984] or [Turner 2013]

Lambdas and functional interfaces

Functional interfaces are so called *SAM* types (single abstract method)

A functional interface has exactly one abstract method, e.g. `Runnable`, `Comparator<T>`, or `Comparable<T>`

Functional interfaces can define 0..* default methods and 0..* static methods

Using the `@FunctionalInterface` annotation on classes the compiler is required to generate an error message if it is no interface and it doesn't define exactly one SAM.

```
@FunctionalInterface
public interface Counter<T> {
    int count(T obj);
}
```

```
Counter<String> strCount = (String s) -> { return s.length(); };
```

[https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html]

Your first lambda with Comparator<T>


```
@FunctionalInterface
public interface Comparator<T> { int compare(T obj1, T obj2); }
```

```
public class Collections {
   public static <T> void sort(List<T> l, Comparator<? super T> c)
   {...}
}
```

Your task:
Prepare your first lambda expression
to compare two String objects by length

```
List<String> names = Arrays.asList("John", "Andrew", "Eve");

Collections.sort(names,   ????   );

for(String name : names) System.out.print(name + " ");
// Eve John Andrew
```

Functional interfaces in JDK

Selection of most used interfaces from package `java.util.function`:

```
// Computes a single input with no result
public interface Consumer<T>  {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {...}
}

// Represents a supplier of results.
interface Supplier<T> {
   T get();
}

// Computes a single output, prodeces output of different type
interface Function<T,R>  {
   R apply(T t);
   default <V> Function<T,V> andThen(Function<? super R,? extends V> after)  {...}
   default <V> Function<V,R> compose(Function<? super V,? extends T> before) {...}
}

// Represents a predicate (boolean-valued function) of one argument
interface Predicate<T> {
   boolean test(T t);
   default Predicate<T> and(Predicate<? super T> other) {...}
   default Predicate<T> negate() {...}
   // ...
}
```

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
UnaryOperator<T>

[https://docs.oracle.com/javase/8/docs/api/java/util/function/package-amountmary.html]

Type inference

Lambda expressions allow for minimal syntax if compiler can deduct type information (so called *type inference*), e.g.:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

R = Integer could be inferred from return type of expression t.length() + u.length() → Integer (e.g. when used directly in typed method call)

```
BiFunction<String, Object, Integer> bf;
bf = (String txt, Object obj) -> { return t.length() + u.hashCode() ); }
// can be even shorter:
bf = (txt, obj) -> t.length() + u.hashCode(); // se below
```

compiler can infer types from declaration of f

Further syntax shortening examples:

for single return statements keyword return together with {}-pair can be dropped

```
(int x, int y) -> { return x * y; }  // shorter:   (x, y) -> x * y
(long x) -> { return x * 2; }        // shortest:  x -> x * 2
```

()-pair can be dropped with only one parameter

Lambda as parameters and return types

Further examples for type inference using `Comparator<T>` as functional interface:

```
List<String> names = Arrays.asList("Ahab", "George", "Sue");
Collections.sort(names, (s1, s2) -> s1.length() - s2.length() );
```

*`T` :: **`String`** can be deducted from `names` being a `List<`**`String`**`>`*

Signature of `Collections::sort` method is:
`public static <T> void sort(List<T> list, Comparator<? super T> c)`

```
Collections.sort(names, createComparator() );
```

```
public Comparator<String> createComparator() {
    return (s1, s2) -> s1.length() - s2.length();
}
```

Statement returned here is of functional interface type `Comparator<String>`

Method references ("function pointers")

Syntax: `Classname::methodName` `objectReferenceName::methodName`

Lambdas can be replaced by method references whenever there would not further actions within the the lambda

Examples:

Reference	Method reference…	…replacing Lambda
static method	`String::valueOf`	`obj -> String.valueOf(obj)`
instance method (via class)	`String::compareTo`	`(s1, s2) -> s1.comapreTo(s2)`
instance method (via object ref)	`person::getName`	`() -> person.getName()`
Constructor	`ArrayList::new`	`() -> new ArrayList<>()`

[Table taken from Inden 2015, p. 812]

Streaming API

- Creating Streams
- Fluently working with streams
- Finalize Streams

Stream ops I use most

Stream operations

$$Source \Rightarrow STREAM \Rightarrow OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n \Rightarrow Result$$

Create — *"lazy" Intermediate* — *not performed until* → *Terminal*

```
// interface Collection
stream()
parallelStream()

// Utiliy class
Arrays.stream(...)
Arrays.stream(...).parallel()
```

Return an object reference of Type Stream<E>

```
// stateless
filter
map
flatMap
peek
//stateful
distinct
sorted
limit
skip
```

```
forEach
toArray
collect
reduce
min
max
count
anyMatch
noneMatch
findFirst
findAny
```

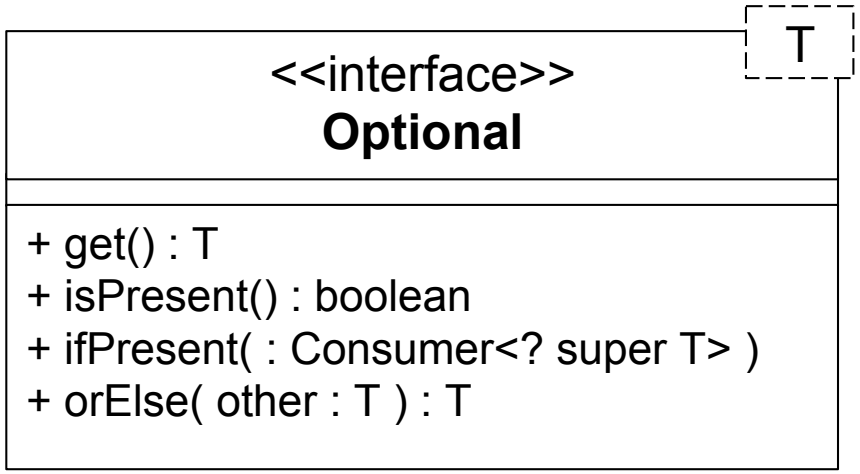
Check the factory methods in **`Collectors`** *for a set of* `Collector`*-Operations*

Check the java doc for more information on parameter types and return types;
intermediate operations return (of course) Stream<E> object references

[Inden 2015, p. 825]

Class `Optional<T>`

The class Optional<T> is a container wrapped around an object and is useful if you do not know whether its content is `null` or not (e.g. when using in fluent programming style).



Example

"old fashioned"

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t : transactions){
    if(t.getType() == Transaction.GROCERY) {
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator(){
    @Override
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t : groceryTransactions) {
    transactionsIds.add(t.getId());
}
```

Stream like

```
List<Integer> transactionsIds =
    transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

[taken from Urma (2014a)]

Bibliography

Gosling, J. et al. (2015): The Java® Language Specification – Java SE 8 Edition, March 2015, http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Huges, J. (1984): Why Functional Programming Matters, rev. ed., http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

Oracle Corp. (2016): The Collections Framework, Website, http://docs.oracle.com/javase/8/docs/technotes/guides/collections/

Turner, D. (2013): Some History of Functional Programming Languages. In: Loidl, H.-W. and Pena, R.: *Trends in Functional Programming*: 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers, p. 1-20. Springer: Berlin and Heidelberg

Urma, R.-G. (2014a): Processing Data with Java SE 8 Streams, Part 1, Java Magazine, March/April 2014, http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Urma, R.-G. (2014b): Part 2: Processing Data with Java SE 8 Streams, Java Magazine, May/June 2014, http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html