TRAINING & CONSULTING

Advanced Java

ABIS Training & Consulting
www.abis.be
training@abis.be

Document number: 1149_04.fm
2 March 2023

Address comments concerning the contents of this publication to:
ABIS Training & Consulting, Diestsevest 32 / 4b, B-3000 Leuven, Belgium
Tel.: (+32)-16-245610

Table of Contents

PREFACE

In this course, you will deepen your knowledge of the Java programming language to build well structured, high performant, distributed applications. We will focus on topics like String handling and formatting, i18n, NIO2, functional programming, logging, testing and multi threading.

The main topics are:

- Advanced Functional Programming: pre defined functional interfaces, advanced stream topics, use of Optional
- Collection enhancements and generics, Object equality
- Working with Strings: formatting, tokenizing, regular expressions, StringBuilder
- Java SE8 DateTime API, I18N, Date and Number formatting
- Advanced I/O topics: I/O in Java 8, File Attributes, directory walking and searching
- Multithreading and Concurrency: concurrent collections, thread interaction, Executors and ThreadPools, Fork/Join framework, parallel streams
- Logging and Testing basics

All these aspects are illustrated with practical examples based on OO implementation principles.

More Functional Programming

Objectives :

- Predefined Functional Interfaces
- Streams
- Optional

Predefined Functional Interfaces

1

- Functional Interface = interface with exactly 1 abstract method (but can have default and static methods)
- use @FunctionalInterface to check
- you can build them yourself, but many predefined functional interfaces available
- interfaces like Comparable, Comparator and Runnable are in fact functional interfaces
- interface method implementations will be used by lambda expressions (e.g. streams)
- variables used in lambdas are local, and have to be effectively final

List of predefined functional interfaces

- **defined in java.util.function package**
- **common functional interfaces**

functional interface	# parameters	return type	abstract method
Supplier<T>	0	T	get
Consumer<T>	1 (T)	void	accept
BiConsumer<T,U>	2 (T,U)	void	accept
Predicate<T>	1 (T)	boolean	test
BiPredicate<T,U>	2 (T,U)	boolean	test
Function<T,R>	1 (T)	R	apply
BiFunction<T,U,R>	2 (T,U)	R	apply
UnaryOperator<T>	1 (T)	T	apply
BinaryOperator<T>	2 (T,T)	T	apply

- **many others available (e.g. those working with primitives)**

Supplier

- provide an instance of T (cf. factory)
- does not accept arguments!
- method: **T get()**
- examples

```
Supplier<Person> sup = () -> new Person();
Person p1 = sup.get();
p1.set....();

// return constant value
Supplier<String>  hello = () -> "Hello";
// or value captured from lexical environment
String wlcm = "Welcome to this course.";
Supplier<String> welcome = () -> wlcm;
System.out.println(hello.get() + welcome.get());
```

- can also use method references

```
Supplier<Person> sup = Person::new
```

More Functional Programming

1. Predefined Functional Interfaces
2. Streams
3. Optional

Consumer

- action performed on T object passed as single argument
- returns no result (i.e. void)
- method: void accept(T object)
- example

  Consumer<Person> cons = p -> System.out.println("Hi " + p.getLastName());

  cons.accept(p1) ;
- default methods (to chain multiple consumers together)
  - `Consumer andThen(Consumer<T> after)`
  - performs, in sequence, this operation followed by the after operation
- Variations
  - primitive (wrapper) passed as argument

    IntConsumer, DoubleConsumer, LongConsumer
  - action performed on 2 objects, no result returned

    BiConsumer <T,R>, ObjIntConsumer(T, int), ,
    ObjLongConsumer(T, long), ObjDoubleConsumer(T, double)

Predicate

- (property of) object T passed as argument
- returns boolean
- method: **boolean test(T object)**
- examples

```
// wordlength > 3
Predicate<String> pred = word -> word.length() > 3 ;
pred.test("Java");    // returns true

// empty string
Predicate<String> isEmpty = String::isEmpty ;
isEmpty.test( p1.getFirstName() ) ;

// constant - always true
Predicate alwaysTrue = x -> true;
```

Predicate (..)

- default methods
  - `Predicate and(Predicate p)`
  - `Predicate negate()`
  - `Predicate or(Predicate p)`
- variations
  - primitive passed as argument
    DoublePredicate, IntPredicate, LongPredicate
  - 2 arguments passed
    BiPredicate

Function

- convert object T to other object R
- input and output types may be different
- method: **R apply(T object)**
- examples

```
// conversion from String -> int -> String
Function<String, Integer> toInt = s -> Integer.valueOf(s) ;
Function<String, String> backToStr = toInt.andThen( num -> String.valueOf(num) ) ;
String targets = backToStr.apply("245");
```

- default methods (to chain multiple functions together)
  - `Function andThen(Function<R, V> after)`

    returns a composed function that first applies this function to its input, and then applies the after function to the result
  - `Function compose(Function<R, V> before)`

    returns a composed function that first applies the before function to its input, and then applies this function to the result

Function (..)

- additional static method
  - `identity()`

    returns its input parameter
- Variations
  - convert primitive to object

    DoubleFunction `applyAsDouble()`
    IntFunction `applyAsInt()`
    LongFunction `applyAsLong()`
  - convert to primitive (wrapper)

    (Long | Int | Double)To(Int | Long | Double)Function
  - convert 2 objects to 1 object or primitive

    BiFunction, To(Int | Long | Double)BiFunction

Function subinterfaces

- UnaryOperator - input and output of same type
  - primitive inputs (Int | Long | Double)UnaryOperator
- BinaryOperator - 2 input and output of same type
  - primitive inputs (Int | Long | Double)BinaryOperator

Streams 2

Objectives:

- what are streams?
- stream operations: intermediate and terminal
- lambda expressions and streams
- building streams

More Functional Programming

1. Predefined Functional Interfaces
2. Streams
3. Optional

What are Streams?

- **stream**
  - **represents a flow of objects**
  - **on which operations can be performed**
  - **operations are non-interfering (do not modify the stream) and are typically stateless**
- **primary aim of streams is to make the (aggregate) operations easy on collections**

  Note: streams don’t store elements; they are computed on demand

- **stream can be seen as consumable sequence of elements that are accessed one at a time (sequential or parallel)**

Difference: Collection - Stream

- **Collection: contains elements/data, external iteration (via *iterator*)**
- **Stream: computes elements on demand, internal iteration**
- **collection -> stream:**
  sequence of elements accessed through pipeline
- **Example:**

```
// external iteration of Collection
List<String> listOfPersonNames = new ArrayList<>();
for (Person p : courseParticipants)
    listOfPersonNames.add(p.getName());

// internal iteration of Stream
List<String> listOfPersonNames =
        courseParticipants.stream()
                          .map(Person::getName)
                          .collect(Collectors.toList());
```

Stream operations

- java.util.Stream
- convert collection into stream -> `stream()`
- *internal iteration*
- streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

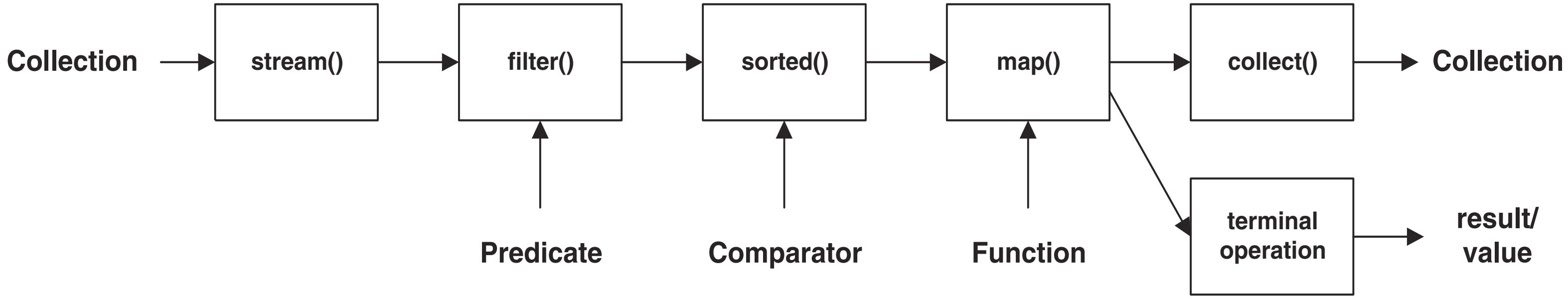
  `forEach( Consumer c )`

  ~= void accept(c)
- Example

  ```
  ArrayList<Person> courseParticipants = ...;
  courseParticipants.stream().forEach( p -> System.out.println(p.getName()) );
  ```
- Note: convert into parallel streams to increase performance
  -> `parallelStream()`
  -> `stream().parallel()`

Stream operations - *intermediate* operations

- **Additional manipulations/transformations are possible,**
  return of (new) stream (*pipelining - chaining*)
  lazy operations



- **filter() -> pick up element(s), based on Predicate**
- **sorted() -> arrange element(s), based on Comparator**
- **map() -> transformation, based on Function**
- **distinct() -> return stream of unique elements, based on equals()**
- **limit(n) -> return stream with maximum size n**
- **skip(n) -> return stream with first n elements discarded**

Filter

- **pick up element(s), based on Predicate**
  - **returns true/false, based on values of variables**
  - **resulting element can be used in another stream operation, e.g. `forEach`**
- **example**

```
courseParticipantNames.stream()
            .filter(name -> name.startsWith("J"))
            .forEach(s -> System.out.println(s));
```

Sorted

- arrange element(s), based on **Comparator**
  - returns sorted view of the stream elements
  - ordering of backed collection is NOT changed
  - **stateful** operation
- Example

```
courseParticipantNames.stream()
        .sorted()    // natural order
        .forEach(s -> System.out.println(s));

// reverse order using custom comparator
courseParticipantNames.stream()
        .sorted( (n1,n2) ->  n2.compareTo(n1) )
        .forEach(s -> System.out.println(s));
```

- Note:

`unordered()` returns (internally) unordered stream
-> better performance on `distinct()` or `groupingBy()`

Map

- transformation/conversion of each element, based on **Function**

  e.g. extract/convert information from each element of stream

- example

```
courseParticipantNames.stream()
            .map( s -> s.toUpperCase() )
            .forEach( s -> System.out.println("name = " + s) ) ;

courseParticipantNames.stream()
            .map(s -> s.length())
            .forEach( l -> System.out.println("length = " + l) ) ;
```

- notes:
  - map converts each element into 1 other element
  - **flatMap** converts each element into stream of elements (1 to many) (also available on Optional)

More Functional Programming

1. Predefined Functional Interfaces
2. Streams
3. Optional

Map (..)

- special Stream interfaces for primitive types (int, double, long) -> used for *reduce* operations later on
  - `mapToInt` -> IntStream
  - `mapToDouble` -> DoubleStream
  - `mapToLong` -> LongStream
- example

```
long nrOfParticipants = courseParticipantNames.stream()
            .mapToInt(s -> s.length())
            .count();
```

- notes:
  - specialized IntFunction and IntPredicate used
  - reverse mapping primitive to Stream: `mapToObj`

Other intermediate operations

- **limit(n)**

  -> return stream with maximum number of elements (size n)

- **skip(n)**

  -> return stream with first n number of elements discarded

- **distinct()**

  -> return stream of unique elements, based on equals()

- **peek()**

  -> return stream of elements, after processing (not modifying) each element with the provided action

- **important note**

  order of chained operations can be important for performance or number of operations actually executed

  i.e. first filter, before sort, before map

Stream operations - *terminal* operations

1. Predefined Functional Interfaces
2. Streams
3. Optional

- Close a stream pipeline,
  return void or a non-stream result of certain type
  or possible side-effect
  - collect() -> convert elements of stream into destination/collection
  - forEach() -> invoke action on each element, based on **Consumer**
  - xxxMatch() -> find and match elements, based on **Predicate**
  - reduce() -> reduce stream to single element or single value, based on **BinaryOperator**
- Notes:
  - as soon as a terminal operation is executed, the stream is closed. No other operations can be invoked -> **IllegalStateException**
  - Only at this point is any processing performed (**eager** operation), which allows for optimization of the pipeline

    lazy evaluation; merged operations; elimination of redundant operations; parallel execution

Collect

- `collect()`

  collect values of stream into target destination/collection

  based on Collector

  implementation via e.g. utility class Collectors methods

  - `toList()`
  - `toSet()`
  - `groupingBy()`
  - `averagingInt()`
  - `summarizingInt()` -> built-in statistics summary
  - `joining()` -> with optional prefix and suffix

- `toArray()`

  returns an array (of Objects) containing the stream elements

Collect examples

```
List<Person> allMales = courseParticipants.stream()
                                  .filter( p -> p.getGender().equals("M"))
                                  .collect(Collectors.toList());


//  grouping
Map<Integer, List<Person>> allGroupedByAge = courseParticipants.stream()
                                  .collect(Collectors.groupingBy( p -> p.age) );

// get statistics (count, sum, min, max, average)
IntSummaryStatistics stats = courseParticipants.stream()
                                  .collect(Collectors.summarizingInt( p -> p.age) );

// joining into String
allMales.stream().collect(Collectors.joining(" and " , "Participants ", " are present.") );
```

```
all males: [James Bond, Peter Dupont]

all by age
age 38: [James Bond, Mary Stones]
age 25: [Peter Dupont]

IntSummaryStatistics{count=3, sum=101, min=25, average=33,666667, max=38}

Participants Bond and Dupont are present.
```

For each element

- invoke an action on each element of the stream, return void
- `forEach()`

  accepts **Consumer** to be executed for each element
- example

  ```
  courseParticipantsNames.stream()
                      .forEach( name -> System.out.println(name) );

  courseParticipantsNames.stream()
                      .forEach( System.out::println );
  ```
- note

  `forEachOrdered()` respects the order of the stream in case of parallel processing

Find and match

- match elements, based on **Predicate**, return boolean
    - `anyMatch()` -> return true if predicate applies to 1 element
    - `allMatch()` -> return true if predicate applies to all elements
    - `noneMatch()` -> return true if predicate applies to no element

```
boolean allMale = courseParticipants.stream()
                .allMatch(p -> p.getGender().equals("M"));
```

- find elements, after **Predicate**, return Optional
    - `findFirst()` -> returns first element
    - `findAny()` -> returns any element

```
Optional<Person> firstFemale = courseParticipants.stream()
                .filter(p -> p.getGender().equals("F"))
                .findFirst();
```

More Functional Programming

1. Predefined Functional Interfaces
2. Streams
3. Optional

Reduce stream to single value

- based on the **map-reduce** pattern (*fold* operation)

  https://en.wikipedia.org/wiki/MapReduce

  - `reduce()` -> return 1 element, based on **BinaryOperator** accumulator
  - `max()` -> return maximum valued element, based on Comparator
  - `min()` -> return minimum valued element, based on Comparator
  - `reduce()`, `max()` or `min()` may return Optional, if stream is empty
  - `count()` -> return number of elements in the stream as *long*
- For primitive type streams
  - `sum()` -> return sum of (numeric) elements (primitive stream only)
  - `average()` -> return average value of elements (idem)

Reduce stream to single value (..)

- **reduce returns an Optional object**

  Optional<T> reduce(BinaryOperator<T> op)

  if collection is possibly empty, or contains possibly only 1 element

- **reduce with initial (default) value**

  T reduce(T id, BinaryOperator<T> op)

  for using a ‘default’ or initial (identity) value, and accumulator

- **reduce with initial value and BiFunction (fused map)**

  <U> U reduce(U identity, BiFunction<U,? super T,U> accum, BinaryOperator<U> combi)

  for using identity value, accumulator and combiner

  BinaryOperator is based on functional interface

  BiFunction.apply(Object, Object)

  returning an object of the same type

- **Example (with initial value)**

```
//  calculate sum of values
reduce( 0, (x, y) ->  x + y )
```

Reduce stream to single value - examples

```
List<Integer> evenNumbers = Arrays.asList(2, 4, 6, 8, 10);
// reduce to calculate the sum: initial value = 0
int sum = evenNumbers.stream()
                    .reduce(0, (m, n) -> m + n );

// reduce based on specialised stream IntStream
int totalAge = courseParticipants.stream()
                    .mapToInt(p -> person.getAge())
                    .sum();

long nrOfParticipants = courseParticipants.stream().count();
```

Optional

- class **Optional** represents the existence or absence of a value
  - is useful if result may be absent
  - used to avoid NullPointerException
- useful methods:
  - `isPresent()` to check whether an object is present
  - `ifPresent()` to execute a block of code if the value is present
  - `get()` to get the value of the object
  - `orElse()` to suggest a substitute for a missing object
  - `orElseThrow()` to throw an Exception when the object is not present
  - `filter()` if value is present, and matches the given Predicate, return Optional
  - `map()` transforms value, if present

Optional - Examples

```
Optional<String> opt = Optional.of("Test");
 // opt = Optional.empty();

 if (opt.isPresent())
      System.out.println("value present");
 else
      System.out.println("no value present");

try {
     System.out.println("value of optional is " + opt.get());
} catch (NoSuchElementException e) {
     System.out.println("no element available. Message: " + e.getMessage());
}

 System.out.println("value of optional is " + opt.orElse("alternative"));

 opt.ifPresent(s -> System.out.println("first char is " + s.charAt(0)));
```

Optional - Examples (..)

```
courseParticipants.stream()
                      .findAny(p -> p.getGender().equals("F"))
                      .ifPresent(Person::printInfo);

courseParticipants.stream()
                      .findAny(p -> p.getGender().equals("F"))
                      .filter(p -> p.getName().startsWith("A"))
                      .ifPresent(Person::printInfo);
```

More Functional Programming

1. Predefined Functional Interfaces
2. Streams
3. Optional

Collections and Generics

Objectives :

- **Comparing objects with equals() and hashCode()**
- **Collections and Functional programming**
- **Generic classes and methods**

Comparing objects with equals() and hashCode()

- like toString(), equals() and hashCode() are methods from the Object class that can be overridden by your own classes
- equals() will be used to check if 2 objects are meaningfully equal, in contrast to == which check object equality
- hashCode() is used to improve performance, but is also linked with equals() via a contract
- both methods are called automatically for checking equality when objects are added to a Set or Map

equals()

- **implementation in Object:**

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- **override in your own class**

```
class Moof {
    private int moofValue;
    Moof(int moofValue) { this.moofValue = moofValue;}
    public int getMoofValue() { return moofValue; }
    public void setMoofValue(int moofValue){this.moofValue = moofValue;}
    public String toString(){ return "Moof "+ moofValue;}
    public boolean equals(Object o) {
        if ((o instanceof Moof) && (((Moof)o).getMoofValue()== this.moofValue)) {
            return true;
        } else {
            return false;
        }
    }
}
public class EqualsTest {
    public static void main (String [] args) {
        Moof one = new Moof(8);    Moof two = new Moof(8);
        if (one.equals(two)) {
            System.out.println("one and two are equal");
        }
    }
}
```

Collections and Generics

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

equals() contract

- **reflexive**: for any non-null reference value x, x.equals(x) should return true
- **symmetric**: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- **transitive**: For any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
- **consistent**: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified
- for any non-null reference value x, x.equals(null) should return false
- if class implements Comparable, compareTo() should return 0 if equals returns true

hashCode()

- numeric (int) value
- hashcode value of an object is used to determine how the object should be stored, and is used again to help locate the object
- mainly important for performance when object is stored in a collection
- “bucket” system
- different objects could have same hashCode()
- by default: each object has a unique hashCode() equal to the internal address of the object on the heap
- if equals() method overridden, hashCode() should also be overridden! (see hashCode() contract)
- algorithm often uses prime numbers combined with variables on which the equals() is based
- to check if 2 objects are equal (in a collection), Java first checks if hashCode() is the same, before it checks the equals() method!

hashCode() contract

- within the same program, the result of hashCode() must not change. This means that you shouldn’t include variables that change in figuring out the hash code
- if equals() returns true when called with two objects, calling hashCode() on each of those objects must return the same result.
- if equals() returns false when called with two objects, calling hashCode() on each of those objects does not have to return a different result. This means hashCode() results do not need to be unique when called on unequal objects
- summary

condition	required	not required (but allowed)
x.equals(y) == true	x.hashCode()==y.hashCode()	
x.equals(y) == false		no hashCode() requirements
x.hashCode()==y.hashCode()		x.equals(y) == true
x.hashCode()!=y.hashCode()	x.equals(y) == false	

Collections and Generics

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

hashCode() - Example

```
public class Company {
        private String name;
        public Company(String name) {this.name = name;}
        public String getName() {return name;}
        public void setName(String name) {this.name = name;}
        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + ((name == null) ? 0 : name.hashCode());
            return result;
        }
        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null) return false;
            if (getClass() != obj.getClass()) return false;
            Company other = (Company) obj;
            if (name == null) {
                if (other.name != null) return false;
            } else if (!name.equals(other.name)) return false;
            return true;
        }
    }
```

Remark: equals() and hashCode() generated by IntelliJ

Collections and Functional programming 2

Functional methods added to Collection 2.1

- **Iterable** interface provides default method

  ```
  void forEach(Consumer<? super T> action)
  ```

- remove element based on predicate

  ```
  default boolean removeIf(Predicate<? super E> filter)
  ```

  example

  ```
  listOfParticipants.removeIf( p -> p.getLastName().startsWith("X") );
  ```

New methods added to List 2.2

- replace all elements based on unary operator

  ```
  default void replaceAll(UnaryOperator<E> operator)
  ```

  example

  ```
  UnaryOperator<Person> uo = p-> {p.setFirstName(p.getFirstName().toUpperCase());
                                                          return p;};
   listOfPersons.replaceAll(uo);
  ```

- sort all elements according to comparator

  ```
  default void sort(Comparator<? super E> c)
  listOfParticipants.sort((p1, p2) ->  p1.getLastName().compareTo(p2.getLastName()));
  ```

Collections and Generics

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

Enhancements to Map

Map does not support streams. Hence the advent of various new useful methods.

- `putIfAbsent(key, value)`
- `forEach(consumer)`
- `computeIfPresent(key, function)`
- `computeIfAbsent(key, function)`
- `getOrDefault(key, defaultvalue)`
- `remove(key, value)` only removes if key is mapped to value

Examples

```
Map<Integer,String> map = new HashMap<>();
map.forEach( (key, value) -> System.out.println(value) );

map.putIfAbsent(key, "value" + key)

map.computeIfAbsent(25, key -> "value" + key)
```

Collections and Generics

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

Utility classes and interfaces

- **Arrays**
  - **void setAll(T[] array, IntFunction<? extends T> generator)**

    Set all elements of the specified array, using the provided generator function to compute each element.
- **Iterator**
  - **void forEachRemaining(Consumer<? super E> action)**

    Performs the given action for each remaining element until all elements have been processed
    (or the action throws an exception.)

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

Utility classes and interfaces (..)

- **Comparator**
  - **Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)**

    static method that accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.

  - **Comparator<T> reversed()**

    imposes the reverse ordering of this comparator.

  - **Comparator<T> thenComparing(Comparator<? super T> other)**

    Returns a lexicographic-order comparator with another comparator.
    If this Comparator considers two elements equal,
    i.e. compare(a, b) == 0, other is used to determine the order.

  - **other methods: `reverseOrder()`, `naturalOrder()`, `nullsFirst()`, `nullsLast()`**
  - **example:**

```
List<Person> personList = ... ;
personList.sort(Comparator.comparing(Person::getLastName).reversed()) ;
```

Generics

- generic = parameterized type
- type parameter names are single, uppercase letters
  - E - Element (often used in the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U, ... - 2nd, 3rd, ...th types
- widely used for collections
- uses angle brackets + “diamond” syntax: <>
- type only checked at compile time, not at runtime -> type erasure!
- possible to create your own generic classes and methods
- limitations when passing collections of subtypes to methods
  -> wildcards needed!

Generic classes

3.1

- **used when functionality could be applied to objects of totally different types**
- **type can be used in methods, constructor or as variable name**
- **note: “new” not allowed on the generic type!**
- **example:**

```
public class GenericMemoryRepository<T>{

    private List<T> components = new ArrayList<>();

    public List<T> findAll(){
        return components;
    }

    public void add(T component) {
        components.add(component);
    }

    public void delete(T component) {
        components.remove(component);
    }
}
```

Collections and Generics

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

Generic methods

- **method which uses a generic type, but that is not a property of the class**
- **parameter type has to be mentioned in the method header!**
- **example:**

```
public class RepositoryFactory {

    public <T> GenericMemoryRepository<T> createMemoryRepository(T type){
        return new GenericMemoryRepository<T>();
    }

}
```

Bounding generic types

- **Java disallows assigning lists of different types to each other (even not subtypes!)**

```
List<Number> numbers = new ArrayList<Integer>();    // NOT ALLOWED
```

OR

```
List<Person> persons = new ArrayList<>();
public void doSomething(List<Object> objects){
    System.out.println(objects);
}
this.doSomething(persons);   // NOT ALLOWED
```

- **solution: work with wildcards**
- **3 types**
  - **unbounded wildcards**
  - **upper-bounded wildcards**
  - **lower-bounded wildcards**
- **limitation: unbounded and upper-bounded wildcards will make the list functionally immutable -> can not add elements!**

Unbounded wildcards

- ? represents any data type
- can only be used for reference type of a parameter, not for the effective type
- diamond syntax can not be used in combination with the wildcard
- examples:

```
List<?> values = new ArrayList<String>();
List<?> numbers = new ArrayList<>();     //NOT ALLOWED
List<Number> numbers = new ArrayList<?>();     //NOT ALLOWED

List<Person> persons = new ArrayList<>();
public void doSomething(List<?> objects){
    System.out.println(objects);
}
this.doSomething(persons);
```

Upper-bounded wildcards

- to limit the type of object passed to the list
- syntax: ? extends T
- type passed has to be of type T or extend/implement T
- examples:

```
List<? extends Number> numbers = new ArrayList<Integer>();
numbers.add(1);   // NOT ALLOWED

public class Instructor extends Person{}
List<Instructor> instructors = new ArrayList<>();
public void doSomethingWithPersons(List<? extends Person> objects){
    System.out.println(objects);
}
this.doSomething(instructors);
```

- List<?> functionally (mostly) equivalent to List<? extends Object>

Collections and Generics

1. Comparing objects with equals() and hashCode()
2. Collections and Functional programming
3. Generics

Lower-bounded wildcards

- sets a lower boundary for the type of object being passed
- syntax: ? super T
- type passed has to be of type T or a supertype of T
- now allowed to add elements to the list, but only if added type is of type T or a subtype
- examples:

```
List<? super Integer> numbers = new ArrayList<Number>();
numbers.add(1);
numbers.add(2.0);   //NOT ALLOWED

public class Instructor extends Person{}
List<Person> persons = new ArrayList<>();
public void doSomethingWithPersons(List<? super Instructor> objects){
    objects.add(new Instructor());
    System.out.println(objects);
}
this.doSomething(persons);
```

String Handling, DateTime and I18N

Objectives :

- String vs. StringBuilder
- Tokenizing and Formatting
- Working with Date and Time
- Internationalization

String vs. StringBuilder

1

- **Strings are immutable - String pool (can lead to unexpected results)**

  String s = "hello";   -> creates the literal "hello" in the pool

  s.concat(" world");   -> adds "hello world" to the pool, but **s** doesn’t reference it

  System.out.println(s);   -> will print out "hello" and not "hello world" !
- **use String s = “hello” and not String s = new String(“hello”);**
- **performance could be improved by using StringBuilder**
  - **avoid extensive String manipulations, use StringBuilder instead**

    StringBuilder sb = new StringBuilder("hello");

    sb.append(" world");
  - **StringBuilder can also be instantiated with an initial capacity**

    StringBuilder sb = new StringBuilder(100);
- **watch out for differences in method names!**
- **StringBuffer is thread safe version of StringBuilder**

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Tokenizing 2

- **to split up bigger pieces of data, e.g. lines read from a text file via LineNumberReader**

StringTokenizer 2.1

- **splits up a String into tokens based on a delimiter, default is blank**
- **if multiple blanks -> multiple splits!**
- **can throw a NoSuchElementException**
- **deprecated -> better not to use in new code!!!**

```
String s2 = "John  47      Leuven  Belgium";
StringTokenizer tokenizer = new StringTokenizer(s2," ");
while(tokenizer.hasMoreElements()){
    String s= tokenizer.nextToken();
    System.out.println(s);
}
```

String.split() 2.2

- **takes a regular expression and produces a String array**
- **to tokenize relatively small pieces of data**

```
String s = "John  47      Leuven  Belgium";  String[] tokens = s.split("\\s+");
```

Tokenizing (..)

Scanner

2.3

- extra features compared to String.split
- scanners can be constructed using files, streams or strings
- tokenizing is performed in a loop
- tokens can be converted to their primitive types automatically
- can be used with a Locale (e.g. to find an int when decimal separator is comma instead of dot)
- Scanner is also used to read from the console
- example

```
Scanner scan1 = new Scanner(System.in);       //first scanner to read from console
System.out.println("give your data");
Scanner scan2=new Scanner(scan1.nextLine());  //second scanner scans input line
while (scan2.hasNext()){                      //per line
    if (scan2.hasNextInt()){
        System.out.println("integer found: "+ scan2.nextInt());
    } else {
        String next =scan2.next();
        System.out.println("text found: " + next );
    }
}
```

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Formatting

- **System.out.printf() writes to the standard console**
  - **syntax: printf(format-string, argument-list)**
  - **format-string defines the output format**

%[arg_index$][flags][width][.precision]conversion char

*arg_index: integer followed directly by $, indicates which argument should be printed at this position
optional if order of arguments matches printing order

*flags:

flag	result
-	left justify the argument
+	include a sign (+ or -) with this argument
0	pad this argument with zeroes
,	use locale-specific grouping separators
(	enclose negative numbers in parentheses

*width: minimum number of characters to print

*precision: number of digits after decimal point

*conversion: b (boolean), c (char), d (integer), f (floating point), s (string), e (exponential)

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Formatting - Examples

- **Example 1**

```
Person p1 = new Person("John",47);
Person p2 = new Person("Alexandra", 36);
Person p3 = new Person("Tim",1);
Person[] persons = {p1,p2,p3};
for (Person p: persons){
    System.out.printf("%1$-25S%2$-4d\n",p.getFirstName(),p.getAge());
}
```

returns:

```
MICHAEL        47
ALEX           36
TIM            1
```

- **Example 2**

```
Map<String, Double> map = new HashMap<String,Double>();
map.put("sql", 2340.5);
map.put("java", 230.887);
map.put("servlets&jsp", -20.244);
for ( String s: map.keySet()){
    System.out.printf("%-20s%10.2f\n",s,map.get(s));
}
```

returns:

```
java            230.89
sql            2340.50
servlets&jsp    -20.24
```

Localization

- locale defines the user's language and country
- used for internationalization
  - gui labels and text
  - formatted date and time
  - formatted numbers (currency character, decimal separator)
- java.util.Locale
  - constructor takes in language code and possibly a country code (regional differences exist)

    ```
    Locale locNL = new Locale("nl");
    Locale locFRBE = new Locale ("fr","BE");
    ```

  - pre defined locales as static variables of Locale

    ```
    Locale.FRENCH    (same as Locale("fr"))
    Locale.FRANCE    (same as Locale("fr","FR"));

    Locale.GERMAN
    Locale.GERMANY

    Locale.JAPAN

    Locale.SIMPLIFIED_CHINESE
    ```

    and others

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Localization (..)

- methods
  - getDefault(): gets the system’s locale
  - getAvailableLocales(): list of all possibilities
  - getLanguage(): gets the language code
  - getCountry(): gets the country code
  - getDisplayLanguage(): gets the language in full, optionally with Locale (default: English)
  - getDisplayCountry(): gets the country in full (if provided), optionally with Locale (default: English)

```
Locale locDef = Locale.getDefault();
System.out.println(locDef);                                 ->  en_GB
Locale locFR = new Locale("fr");
Locale locNL = new Locale("nl");
Locale locIT = Locale.ITALY;
System.out.println(locFR.getDisplayLanguage(locIT));        ->  francese
System.out.println(locIT.getDisplayCountry());              ->  Italy
Locale[] localeList = Locale.getAvailableLocales();
for (Locale l: localeList){
    System.out.println(l.getLanguage() + " " + l.getCountry() + " "
                                 + l.getDisplayLanguage(locNL));
}
```

Number Formatting

- printf

  see previous

- NumberFormat and DecimalFormat
  - possible to add a locale, currency symbols, percentages
  - pattern definition:

symbol	meaning
0	shows exactly 1 digit
#	digit, but leading zeroes not shown
.	decimal separator or monetary decimal separator
%	multiply by 100 and show as percentage
E	for exponent in scientific notation

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

NumberFormatting - Examples

- **example 1**

```
double d = 34.75;
DecimalFormat df = new DecimalFormat("#.0");
System.out.println(df.format(d));
                                output:  34.8
```

- **example 2**

```
NumberFormat nf = NumberFormat.getCurrencyInstance(new Locale("nl","BE"));
System.out.println(nf.format(d));
                                output: 34,75 €
```

- **example 3**

```
double v = 0.043;
NumberFormat nf2 =NumberFormat.getPercentInstance(new Locale("nl","BE"));
System.out.println(nf2.format(v));
                                output:  4%
```

- **example 4**

```
DecimalFormat df2 = new DecimalFormat("#.00%");
System.out.println(df2.format(v));
                                output: 4.32%
```

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Java SE8 Date/Time API

6

- in Java SE8, java.time package introduced (originally JSR 310)
- Java SE reaction to Joda time
- simplifies working with date / time
- accounts for Daylight Savings Time
- supports many chronologies:

  ISO, Hijrah-umalqura, Japanese, Minguo, ThaiBuddhist
- most important classes:
  - LocalDate, LocalTime, LocalDateTime
  - ZonedDateTime
  - Period, Duration
  - DateTimeFormatter
- DateTime(Parse)Exception is a runtime exception, but good practice to catch it

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Formatting DateTime

Letter	Date component	Letter	Time component
y	year	H	hour in day (0-23)
M	month in year	h	hour in am/pm (1-12)
w	week in year	K	hour in am/pm (0-11)
W	week in month	k	hour in day (1-24)
d	day in month	a	am/pm marker (text)
D	day in year	m	minute in hour
E	day in week (text)	s	second in minute

number of times a letter is repeated determines the width/type

e.g.: MM -> 10 MMM -> Oct MMMM -> October

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Java SE8 Date/Time API - Examples

- **example 1: construction and formatting of DateTime object**

```
    try {
        LocalDate localDate = LocalDate.now();
        LocalDate futureDate = LocalDate.of(2020, 2, 29);
        DateTimeFormatter fmt =
            DateTimeFormatter.ofPattern("dd MMMM yyyy", new Locale("nl"));
        System.out.println("Vandaag is het: " + fmt.format(localDate) + ",
                andere datum is: " + fmt.format(futureDate));
    } catch (DateTimeException e) {
        System.out.println(e.getMessage());
    }
```

- **example 2: constructing DateTime via String**

```
try {
        DateTimeFormatter fmt2 = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        LocalDate someDate = LocalDate.parse("31/13/2025", fmt2);
        System.out.println(someDate);
    } catch (DateTimeException dte) {
        System.out.println(dte.getMessage());
    }
```

- **example 3: using methods for calculations**

```
    LocalDate now = LocalDate.now();
    LocalDate calculated = now.plusMonths(3).minusDays(25);
    System.out.println(calculated);
```

Java SE8 Date/Time API - Examples (..)

- **example 4: time difference**

```
LocalDate now = LocalDate.now();
LocalDate christmas =LocalDate.of(2018,12,25);
Period diff = Period.between(now, christmas);
System.out.printf("Time to go:  %d months and %d days",
                                        diff.getMonths(), diff.getDays());
```

- **example 5: time zones**

```
TreeSet<String> zones =new TreeSet(ZoneId.getAvailableZoneIds());
System.out.println(zones);
ZonedDateTime timeInJapan = ZonedDateTime.now(ZoneId.of("Asia/Tokyo"));
System.out.println(timeInJapan);
```

- **example 6: chronologies**

```
JapaneseDate jdate = JapaneseDate.now();
System.out.println("today japanese: "+jdate);
LocalDate bday=LocalDate.of(1978,4,10);
JapaneseDate bd = JapaneseDate.from(bday);
System.out.println("bday japanese: "+bd);
```

Ingredients 7.1

- `ResourceBundle`

  J2SE class for resource bundles

- `Locale`

  Locale to use is stored in UI component tree

  Initialised to locale of browser

- Properties file

  standard mechanism to localize messages and paths to images

  messages can use parameters

  same format as standard Java properties file

Properties file - Definition

- properties file placed in package (in classpath)
- extension `.properties`

filename	locale
applicationResources.properties	default or unconfigured
applicationResources_nl.properties	dutch language
applicationResources_nl_BE.properties	dutch language in Belgium

- each message is a `key=value` pair
- possible to add parameters to message value
- example:

applicationResources_nl.properties

```
login.firstname = Voornaam
lastname= Familienaam
company= Bedrijf
message = Welkom {0} van {1}.
```

Controlling the locale and getting messages in Java code

7.3

- standard Java `Locale` class
- initialized with locale of system
- obtaining/changing the default locale:

```
Locale loc = Locale.getDefault();
Locale.setDefault(new Locale("nl"));
```

- getting locale specific messages from Java code

  needed:
  base name of properties file, locale to use, key in properties file

```
String messageKey = "login.firstname";
String baseName = "be.abis.i18n.resources.applicationResources";
ResourceBundle bundle = ResourceBundle.getBundle(baseName, loc);
String messageDefaultLoc = bundle.getString(messageKey);
ResourceBundle bundle2 = ResourceBundle.getBundle(baseName, new Locale("fr"));
String messageLocFR = bundle2.getString(messageKey);
String message =
    MessageFormat.format(bundle.getString("message"), "Sandy","Abis");
System.out.println(message);
```

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

String Handling, DateTime and I18N

1. String vs. StringBuilder
2. Tokenizing
3. Formatting
4. Localization
5. Number Formatting
6. Java SE8 Date/Time API
7. Internationalization (I18N)

Advanced I/O Topics

Objectives :

- Files and Paths
- File I/O enhancements since Java 8
- File Attributes
- Moving and copying files
- Directory Walking and Searching

Files and Paths

1

- java.nio.file.Path
  - system dependent file path
  - hierarchical, composed of a sequence of directory and file name elements
- java.nio.file.Files
  - access to files/directories
  - manipulation of file attributes
  - diagnosis of exceptions

Path

Creation of path

- via helper class Paths
  - convert String or URI to Path

    Path p1 = Paths.get("/home/training/team01/java/in.txt");
- convert a File to a Path

  File f2 = new File("C:\\temp\\java\\in.txt");

  Path p2 = f2.toPath();

Other path manipulations

- retrieve information (like filename or root)
- convert to absolute path, or to URI
- join (partial) paths
- compare two paths

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

Files

Control

- existence
- access allowed
- compare

Manipulation

- copy
- move
- delete

File attributes

- package java.nio.file.attributes

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

File I/O enhancements since Java 8

2

- **Files.newBufferedReader()**
  - takes path and charset as input
  - example

```
Path path = Paths.get("myFile.txt");
try (BufferedReader reader =
        Files.newBufferedReader(path,Charset.forName("US-ASCII"))) {
    String currentLine = null;
    while((currentLine = reader.readLine()) != null)
        System.out.println(currentLine);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

- **Files.newBufferedWriter()**
  - takes path and charset as input
  - example

```
Path path = Paths.get("myFile.txt");
try (BufferedWriter writer =
        Files.newBufferedWriter(path,Charset.forName("UTF-8")
                                                        ,StandardOpenOption.APPEND)) {
            writer.write("hello world");
} catch (IOException e) {
    // Handle file I/O exception...
}
```

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

File I/O enhancements since Java 8 (..)

- start from file (or channel), using static methods in Files
- will take a Path as input
- java.nio.file.Files.lines() returns a Stream<String>, whereas Files.readAllLines() returns a List<String>
  - can use the standard stream methods
  - example

```
try (Stream<String> allLines = Files.lines(Paths.get("myFile.txt"))) {
    long nrOfLines = allLines.count();
    System.out.println("number of lines in file: "+nrOfLines);
    Path path = Paths.get("myFile.txt");
    Files.lines(path).filter(x->x.startsWith("a")).forEach(System.out::println);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

- Note: also BufferedReader has a method `lines()` that returns a Stream<String>

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

File Attributes

- metadata about files and directories
  - last modified time
  - accessibility
  - visibility
  - size
  - ...
- some methods are OS dependent
- some methods can throw IOException
- ways to retrieve metadata
  - Files class
  - BasicFileAttributes class

Getting File Attributes via the Files class

- type of object a path refers to
  - Files.isRegularFile(path), Files.isDirectory(path), Files.isSymbolicLink(path)
- visibility
  - Files.isHidden(path)
- accessibility
  - Files.isReadable(path), Files.isExecutable(path)
- file length
  - Files.size(path)
- file modifications
  - Files.getLastModifiedTime(path), Files.setLastModifiedTime(path, fileTime)
- ownership
  - Files.getOwner(path) returns a UserPrincipal
  - Files.setOwner(path, userPrincipal)

Getting File Attributes via BasicFileAttributes class

- **limitations of using Files to get metadata**
  - **one method per attribute -> many roundtrips to file system -> bad performance**
  - **difficult to make distinction between OS specific attributes**
- **BasicFileAttributes class solves this**
  - **different attributes can be read with one method**
  - **OS specific subclasses of BasicFileAttributes**
  - **limitation: read only**
- **subclasses:**
  - **DosFileAttributes (DOS/Windows-based systems)**
  - **PosixFileAttributes (POSIX systems, such as UNIX, Linux, Mac, and so on)**
- **BasicFileAttributeView (and subclasses) can be used to modify the attributes**

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

BasicFileAttributes - Example

```
public class BasicFileAttributesExample {

    public static void main(String[] args) throws IOException {
        Path path = Paths.get("myFile.txt");
        BasicFileAttributes data = Files.readAttributes(path,BasicFileAttributes.class);
        System.out.println("Is path a directory? "+data.isDirectory());
        System.out.println("Is path a regular file? "+data.isRegularFile());
        System.out.println("Is path a symbolic link? "+data.isSymbolicLink());
        System.out.println("Path not a file, directory, nor symbolic link? "+ data.isOther());
        System.out.println("Size (in bytes): "+data.size());
        System.out.println("Creation date/time: "+data.creationTime());
        System.out.println("Last modified date/time: "+data.lastModifiedTime());
        System.out.println("Last accessed date/time: "+data.lastAccessTime());
        System.out.println("Unique file identifier (if available): "+ data.fileKey());
    }

}
```

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

Moving and copying files

- Files.copy(path1,path2,StandardCopyOption... options)
- Files.move(path1,path2,StandardCopyOption... options)
- Options
  - REPLACE_EXISTING – replace a file if it exists
  - COPY_ATTRIBUTES – copy metadata to the new file
  - NOFOLLOW_LINKS – shouldn't follow symbolic links

Files.walk() 5.1

- traverses a directory in a depth-first, lazy manner
  - Set of elements is built and read while the directory is being traversed. E.g., until a specific subdirectory is reached, its child elements are not loaded.
  - Performance enhancement allows the process to be run on directories with a large number of descendants in a reasonable manner.
- returns a Stream<Path> object
- example

```
Path path = Paths.get("mydir");
try {
    Files.walk(path)
            .filter(p -> p.toString().endsWith(".java"))
            .forEach(System.out::println);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

- variation: Files.walk(path,int) takes into account search depth

Directory Walking and Searching (..)

Files.find() 5.2

- use instead of filter() in Files.walk()
- parameters:
  - path
  - search depth
  - BiPredicate (with parameters path and BasicFileAttributes)
- example

```
Path path = Paths.get("mydir");
long dateFilter = 1420070400000l;
try {
    Stream<Path> stream = Files.find(path, 10,
            (p,a) -> p.toString().endsWith(".java")
                    && a.lastModifiedTime().toMillis()>dateFilter);
    stream.forEach(System.out::println);
} catch (Exception e) {
    // Handle file I/O exception...
}
```

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

Directory Walking and Searching (..)

Files.list()

5.3

- like Files.walk(), but does not consider subdirectories
- example

```
Stream<Path> allFiles = Files.list(Paths.get("mydir")) ;
allFiles.filter(p -> !Files.isDirectory(p)).forEach(System.out::println);
```

Advanced I/O Topics

1. Files and Paths
2. File I/O enhancements since Java 8
3. File Attributes
4. Moving and copying files
5. Directory Walking and Searching

Concurrency

Objectives :

- Multithreading and synchronization
- Concurrency API

Multithreading

- Thread = single sequential flow of control / independent task
- Multi-threading = multiple concurrently executing threads

- Problems
  - synchronize information between concurrent threads
  - interaction between threads

    Be careful to avoid deadlocks!
  - priority
  - performance

Creating threads

- **2 possibilities**
  - **Implement the interface java.lang.Runnable**

    ```
    public class ThreadDemo implements Runnable
    ```
  - **Extend the class java.lang.Thread**

    ```
    public class ThreadDemo extends Thread
    ```
- **Implement method `run()`**

  ```
  public void run( ) {  }
  ```
- **Start new thread via invocation of method `start()`**

  ```
  ThreadDemo td = new ThreadDemo(); // in case of Runnable
  new Thread(td).start();

  ThreadDemo td = new ThreadDemo(); // in case of Thread
  td.start();
  ```
- **alternative: lambdas using Runnable as functional interface**

  ```
  new Thread(()->System.out.println("Hello")).start()
  ```

Common thread methods

- Thread information
  - method `currentThread()`
  - returns e.g. `Thread[thread1,5,main]`
- Named thread

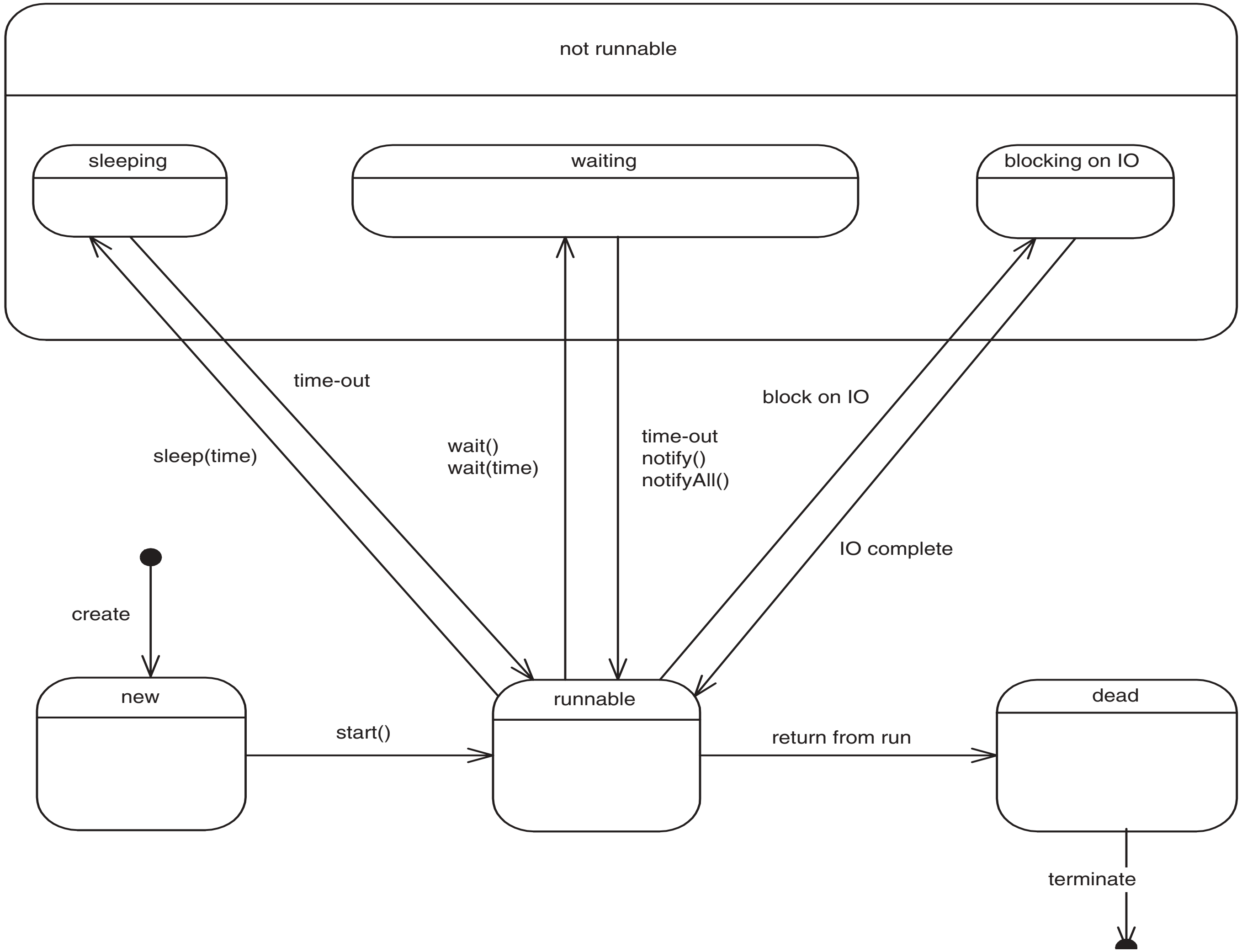
  Thread t = new Thread(runnable, "thread name")

  System.out.println(t.getName());
- Thread priority (default inherited from creator of thread)
  - minimum 1, maximum 10, normal 5
  - `thread.getPriority()`
  - `thread.setPriority(priority)`
- Thread group
  - put threads in (common) thread group

    ThreadGroup tg = new ThreadGroup("thread group name");

    Thread t = new Thread(runnable, "thread name", tg)

Thread States



- **check actual state with `thread.isAlive()`**

Starting Threads

- thread.`start()`
  - runnable thread with highest priority
  - *round-robin* mechanism for threads with same priority
- Thread runs until
  - thread with higher priority becomes runnable
  - run() method terminates
  - it is blocked or interrupted

Blocking Threads

- **sleep(long milliSeconds [, int nanoSeconds])**

  causes the thread to pause its execution for the specified amount of milliseconds

  locks are not relinquished

- **yield()**

  causes the thread to temporarily pause its execution and allow other threads of the same priority to execute

- **join([long milliSeconds [, int nanoSeconds]])**

  causes the current thread to wait (for the specified amount of time) for the specified thread to die

sleep() and join() can throw `InterruptedException`

Example

```
public class JoinDemo extends Thread{
    public JoinDemo(String name) {
        super(name);
    }
    public static void main(String[] args) {
        Thread t1 = new JoinDemo("Thr1");
        Thread t2 = new JoinDemo("Thr2");
        t1.start(); t2.start();
        try {
            System.out.println("Wait for the child threads to finish.");
            t1.join();
            if (!t1.isAlive())
                System.out.println("Thread Thr1 is not alive.");
            t2.join();
            if (!t2.isAlive())
                System.out.println("Thread Thr2 is not alive.");
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Exit from main thread.");
    }
```

Concurrency

1. Multithreading
2. Synchronization
3. Thread interaction
4. Concurrency API
5. Parallel Streams

Example (..)

```
    public void run() {

        System.out.println(Thread.currentThread().getName() + " is going to sleep 5s");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("exception " + e.getMessage());
        }
    }
}
```

Synchronization

- control the access to an object, shared by multiple threads
- method 1: use **synchronized methods** -> keyword `synchronized`
  - lock flag associated with each object, implements **mutual exclusion**
  - *synchronized* method takes the lock flag (i.e. locks the object)
  - *synchronized* method can only be executed if lock flag not already used by another thread
  - *synchronized* prevents the other synchronized methods in the class from being executed
  - *synchronized* doesn’t prevent non-synchronized methods in the class from being executed
  - lock flag automatically released at the end of the *synchronized* block
  - example:

    public **synchronized** double getSalary() { ... }

Synchronization (..)

- method 2: synchronized blocks/statements
  - lock specific object, not the containing object

    synchronized(objectToLock) { ... }
  - lock limited to the block of following statements
  - example

    ```
    private Boolean salaryLock = new Boolean(false);
    public double getSalary() {
        synchronized(salaryLock) { ... }
    }
    ```
  - Note: synchronized methods use `synchronized(this)` on entire mthod
- method 3: static synchronization
  - static methods use lock on the class

    static synchronized void method() { ... } or static synchronized (<class name>.class)
  - lock used only by (other) static methods
- if a lock cannot be obtained, the thread enters the blocked state

Concurrency

1. Multithreading
2. Synchronization
3. Thread interaction
4. Concurrency API
5. Parallel Streams

Thread interaction

implement “producer - consumer” concept

only works in synchronized blocks!

Interaction control methods (in class Object)

- **wait(long milliSeconds [, int nanoSeconds])**
  - **wait for notification of a free resource**
  - **wait for the completion of another thread and release current lock**
- **notify()**
  - **wakes up a single thread**
  - **relinquish locked object**
- **notifyAll()**
  - **wakes up all waiting threads**
  - **relinquish locked object**

Interrupting Threads

- interrupt()
  - awaken (sleeping or waiting) thread
  - throws `InterruptedException`

    use try catch block for the method invoking the `wait()/sleep()` to start running (depending on priority)
- time-out

  waiting thread continues
- notify() or notifyAll()

Example of thread interaction

```
public class DemoQueue {
    private Message head, tail;
    public synchronized void appendMessage(Message p) {
        if (tail == null)
            head = p;
        else
            tail.next = p;
        p.next = null;
        tail = p;
        notifyAll();
    }
    public synchronized Message getMessage() {
        try { while (head == null)
                wait();
        } catch(InterruptedException e) {   }
        Message p = head;
        head = p.next;
        if (head == null)
            tail = null;
        return p;
    }  }
```

Concurrency

1. Multithreading
2. Synchronization
3. Thread interaction
4. Concurrency API
5. Parallel Streams

Concurrency API

- package `java.util.concurrent`
- provides a lot of additional features
- low-level synchronization utilities
  - locking and atomic variables
- higher-level, thread-safe, high-performance concurrency classes
  - ExecutorServices and thread pools
  - concurrent collections
  - based on semaphores, mutexes, latches, and barriers
  - fork/join framework

Executors and ThreadPools

- **ExecutorService creates and manages threads for you**
- **includes numerous features, such as thread pooling and scheduling**
- **types of ExecutorServices**
  - **single thread executor**
    - uses one single thread

      ExecutorService es = Executors.newSingleThreadExecutor()
  - **cached thread pool**
    - will create new threads as they are needed and reuse threads that have become free (java decides how many)

      ExecutorService es = Executors.newCachedThreadPool()
  - **fixed thread pool (most common)**
    - pool with fixed number of threads (chosen by you)

      ExecutorService es = Executors.newFixedThreadPool(5)
  - **scheduled thread pool**
    - can schedule commands to run after a given delay or to execute periodically

      ScheduledExecutorService es = Executors.newScheduledThreadPool(5)

Executors and ThreadPools (..)

- **check number of available cpus via:**

  int cpus = Runtime.getRuntime().availableProcessors();
- **task to be executed**
- **execute via**
  - **es.execute(new MyRunnableTask()) (returns void)**
  - **es.submit(new MyRunnableTask()) (returns Future<?>)**
  - **es.submit(new MyCallableTask()) (returns Future<ReturnType>)**
- **Future**
  - **object that can be used to determine if the task is complete**
  - **can also be used to return a generic result object after the task has been completed**
  - **methods: isDone(), isCancelled(), cancel(), get()**
- **shutdown ExecutorService via es.shutdown() in finally block**
- **es.awaitTermination() is counterpart of myThread.join()**

Callable vs. Runnable

- limitations of Runnable’s run() method
  - returns void
  - can’t throw checked exception
- use Callable instead (implement call() method)
- example

```
public class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws MyCheckedException {
        int count = ThreadLocalRandom.current().nextInt(1,11);
        for(int i = 1; i <= count; i++) {
            System.out.println("Running..." + i);
        }
        return count;
    }
}
```

Concurrency

1. Multithreading
2. Synchronization
3. Thread interaction
4. Concurrency API
5. Parallel Streams

Atomic Variables and ThreadLocalRandom

4.2

Concurrency

1. Multithreading
2. Synchronization
3. Thread interaction
4. Concurrency API
5. Parallel Streams

Atomic Variables

- **increment operator ++ is not thread-safe (actually 2 operations)**
- **java.util.concurrent.atomic package provides classes like AtomicInteger, AtomicLong and AtomicBoolean**
- **example**

```
AtomicInteger ai = new AtomicInteger(5);
ai.incrementAndGet();
ai.addAndGet(-4);
System.out.println(ai);
```

ThreadLocalRandom

- **Math.random() and shared Random instances are thread-safe, but suffer from contention when used by multiple threads**
- **ThreadLocalRandom is unique to a thread and will perform better**
- **provides several convenient methods such as nextInt(int,int) that allow you to specify the range of possible values returned**
- **example**

```
int random = ThreadLocalRandom.current().nextInt(150,450)
```

Concurrent Collections

- accessing collections from across multiple threads is so common that the writers of Java thought it would be a good idea to have alternate versions of many of the regular collections classes just for multi-threaded access
- accessor methods synchronized by default
- avoids ConcurrentModificationException
- many classes available, e.g. CopyOnWriteArrayList, LinkedBlockingQueue and ConcurrentHashMap
- BlockingQueue is like a regular Queue, except that it includes methods that will wait a specific amount of time to complete an operation
- only use when necessary, since they can affect performance
- example

```
try {
    BlockingQueue<Integer> blockingQueue = new LinkedBlockingQueue<>();
    blockingQueue.offer(39);
    blockingQueue.offer(3, 4, TimeUnit.SECONDS);
    System.out.println(blockingQueue.poll());
    System.out.println(blockingQueue.poll(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException e) {// Handle interruption}
```

Parallel Streams

- Streams API has built-in concurrency support
- *parallel stream* is a stream that is capable of processing results concurrently, using multiple threads
- creation
  - parallel(): on an existing stream

    ```
    myList.stream().parallel()
    ```

  - parallelStream(); on Collection object

    ```
    myList.parallelStream()
    ```

- improves performance, but could change expected result!

```
Arrays.asList(1,2,3,4,5,6)
    .stream()
    .forEach(s -> System.out.print(s+" "));
```

vs.

```
Arrays.asList(1,2,3,4,5,6)
    .parallelStream()
    .forEach(s -> System.out.print(s+" "));
```

vs.

```
Arrays.asList(1,2,3,4,5,6)
    .parallelStream()
    .forEachOrdered(s -> System.out.print(s+" "));
```

Concurrency

1. Multithreading
2. Synchronization
3. Thread interaction
4. Concurrency API
5. Parallel Streams

Parallel Streams (..)

- many common streams including map(), forEach(), and filter() can be processed independently, although order is never guaranteed
- since order is not guaranteed, methods such as findAny(), findFirst(), limit() and skip() may result in unexpected behaviour
- some operations require special handling to be able to be processed in a parallel manner
  - reduce()
  - collect()

Introduction to JUnit and Logging

Objectives :

- **Testing principles and Unit testing**
- **Logging**

Testing principles

Introduction to JUnit and Logging

1. Testing principles
2. JUnit Overview
3. JUnit 5 implementation
4. Logging concepts
5. Log4j

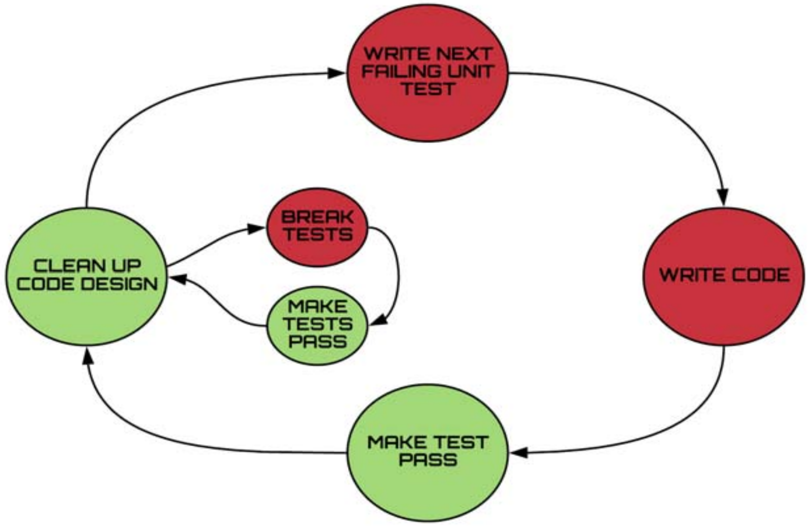
types of tests:

- code testing
  - **unit test**
  - integration test
- regression testing
- performance testing
- defect tracking
- ...

Testing based on test plans/scenarios, derived from use cases

Test Driven Design (TDD)

Red-Green-Refactor



Test for:

- **boundary cases**
- **exceptions**

Why use a unit testing framework?

Advantages

- easier to write tests
- easier to run tests
- easier to rerun tests after change

  +

  consistency, maintenance, ramp-up time, automation

JUnit Overview

web site: www.junit.org

open source testing framework used to develop and execute unit tests in Java

What are unit tests?

- **low-level**
- **investigate the behaviour of a single component (=unit) within a class, servlet, EJB,...**
- **based on component specification**
- **written before the component is developed (test-driven development)**

Why use unit testing?

- **improvement in productivity and overall code quality**

JUnit overview

Instantiate object -> invoke method -> verify assertions

JUnit 4 (2006): refactored to take advantage of Java SE 5 features

- **annotations** (no inheritance, no naming conventions)
  - increase of flexibility
  - more lightweight
- **new functionality**
  - parameterized tests
  - simplified exception testing
  - timeout tests
  - flexible fixtures
  - easy way to ignore tests
  - new way to logically group tests

-> packages `org.junit.*`
(and `junit.framework.*` for compatibility with JUnit 3)

JUnit 5

- **released September 2017, requires Java 8 or higher**
- **JUnit 5 is composed of several different modules from three different sub-projects**
- **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**
  - **JUnit Platform**
    - serves as a foundation for launching testing frameworks on the JVM
    - defines the TestEngine API for developing a testing framework that runs on the platform
    - provides a Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven as well as a JUnit 4 based Runner for running any TestEngine on the platform
  - **JUnit Jupiter**
    - combination of the new programming model and extension model for writing tests and extensions
    - provides a TestEngine for running Jupiter based tests
  - **JUnit Vintage**
    - provides a TestEngine for running JUnit 3 and JUnit 4 based tests

JUnit 5 (..)

- new in JUnit 5
  - lambda support
  - test interfaces with default methods
  - nested unit tests
  - conditional test execution
  - parameterized tests
  - possibility to write custom extensions
  - repeated tests
  - dynamic tests
- watch out: some names of annotations have changed!
- JUnit 4.13 integrated some features of JUnit 5!

JUnit 5 implementation

junit-jupiter-*.jar in classpath

Most important classes of the `org.junit.jupiter.api.*` framework

- **Assert** with several static test methods
- Write test methods in test classes with annotations

Execution of the test cases

- with the command line
  - `org.junit.runner.`JUnitCore
- or via IDE tooling (Eclipse, IntelliJ, Netbeans,...)

JUnit Assert

- The Assert class contains only static methods to be invoked in the test methods (use static import)
- General principle:
  - AAA: Arrange - Act - Assert
  - if the test fails ---> report the failure

Example

```
import static org.junit.jupiter.api.Assert.*;
Person p = new Person("John","Travolta");     //arrange
String firstName = p.getFirstName();           //act
assertEquals(firstName,"Johan");               //assert
```

Two types of reports:

- Failures: failures of anticipated test conditions

```
org.junit.ComparisonFailure: expected:<Joh[a]n> but was:<Joh[]n>
```

- Errors: unexpected error or exceptions

JUnit Assert (..)

Different tests

Type of test	Normal	Negated
Condition returns true	assertTrue(boolean)	assertFalse(boolean)
Object does not exist	assertNull(Object)	assertNotNull(Object)
Both objects refer to the same instance	assertSame(Object, Object)	assertNotSame(Object, Object)
Both objects are equal	assertEquals(Object, Object)	assertNotEquals(Object,Object)
Exception is thrown	assertThrows()	assertDoesNotThrow()
Fail unconditionally	fail()	-

Note:

- **assertEquals is overloaded to compare**

  Objects, booleans, longs, doubles,....

  https://junit.org/junit5/docs/5.8.2/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

JUnit test class

1. Testing principles
2. JUnit Overview
3. JUnit 5 implementation
4. Logging concepts
5. Log4j

Create 1 test class for each class to be tested with

- annotation `@Test` before test method
- name test methods after the goal to be reached
  e.g. nameOfPerson1IsJohn()
- initialize and finalize the fixture (test context)
  - defined via annotations `@BeforeEach` and `@AfterEach`

    -> `setUp()` / `tearDown()` methods
  - called before and after each test

  to make sure there are no side effects between test runs

Example

```
import static org.junit.jupiter.api.Assert.*;
import org.junit.jupiter.api.AfterEach;
import org.junit..jupiter.api.BeforeEach;
import org.junit..jupiter.api.Test;
import be.abis.demo.Person;

public class PersonTest {

    Person p;

    @BeforeEach
    public void setUp() throws Exception {
        p = new Person("John","Travolta");
    }

    @AfterEach
    public void tearDown() throws Exception {p = null;}

    @Test
    public void firstNameIsJohah() throws NameTooLongException {
        String firstName = p.getFirstName();
        assertEquals("First name incorrect", "Johan", firstName);
    }

    @Test
    public void testGetFirstName() {
        Person p2 = new new Person("SomeVeryLongFirstName","Travolta");
        assertThrows(NameTooLongException.class, () -> p.getFirstName());
    }

}
```

Logging concepts

Capture relevant information for testing/debugging/tracing

- use of `System.out.println()`
  - quick and dirty
  - remove code before going into production
- logging code in separate class
  - use *boolean* for triggering debug information
- use logging framework
  - priority levels
  - standardised, formatted information
  - configurable
  - *handlers* for console, files, streams, sockets, OS logs,...
- controlled by logging manager

Log4j

Open source framework from Apache (*http://logging.apache.org/log4j*)

- easy to use, yet flexible framework for application logging with minimal performance impact
- runtime configurable: enable logging at runtime without modifying the application binary
- *hierarchical loggers*: selectively control which log statements are output at arbitrary granularity
- direct its output to a file, the console, a remote server using TCP, a remote Unix Syslog daemon, a remote listener using JMS, the EventLog or even send e-mail
- logging levels TRACE, DEBUG, INFO, WARN, ERROR and FATAL
- formatting is done via (extension of) Layout class
- Versions
  - Log4j (2001)
  - Log4j 2 (2014)

Log4j 2 - What changed?

- improved reliability
- extensibility
- simplified configuration syntax
- support for xml, json, yaml and properties configurations
- improved filters
- property lookup support for values in config file, system properties, environment variables, ThreadContext Map, and data in the event
- support for multiple APIs: can be used with apps using the Log4j 1.2, SLF4J, Commons Logging and java.util.logging (JUL) APIs
- custom log levels
- Java 8-style lambda support for "lazy logging"
- markers
- support for user-defined Message objects
- "garbage-free or low garbage" in common configurations
- improved speed

Log4j2 concepts

- base packages: org.apache.log4j and org.apache.logging.log4j
- base components: *loggers*, *appenders*, *layouts and filters*
- log messages according to message type and level
- control at runtime how messages are formatted and where they are reported
- ***Logger*** is named (case sensitive) entity, organised in hierarchy. Logger inherits level from ancestor if no level is assigned.

  hierarchy is based on dot prefix

  java
  java.util
  java.util.List

  root level is always defined (not named)
- Example

```
Logger myLog = LogManager.getLogger("be.abis.ch5logging");
Logger rootLog = LogManager.getRootLogger();
```

Logging levels and methods

Level	Description	Method
FATAL	non-recoverable or catched runtime (unchecked) exceptions (highest level)	fatal()
ERROR	recoverable or checked exceptions - **default**	error()
WARN	warning messages	warn()
INFO	informational messages	info()
DEBUG	used for debugging/diagnosing problems (lowest level)	debug()
TRACE	used for tracing information	trace()

generic method: `log(Level lvl, Object message)`

A logging request is enabled if its level >= the level of its logger; otherwise, the request is disabled.

```
Configurator.setLevel("be.abis.ch5logging",Level.WARN);
Configurator.setrootLevel(Level.ERROR)

myLog.error(...)  -> enabled
rootLog.warn(...) -> disabled
```

A logger without an assigned level will inherit one from the hierarchy.

Introduction to JUnit and Logging

1. Testing principles
2. JUnit Overview
3. JUnit 5 implementation
4. Logging concepts
5. Log4j

Output specification

- ***Appender*** **specifies the output destination (25 possibilities in log4j2!):**
  - **console, file, socket, cassandra, JPA HTTP, Async,...**
  - **each enabled logging request for a given logger will be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy.**
- ***Layout*** **makes it possible to customize and format the output**
  - **Example: *PatternLayout* (cf. printf)**
    - default set to the pattern:

      %-4r [%t] %-5p %c %x - %m%n

      time - [thread] - log level - logger name - message
    - example

      123 [main] FATAL be.abis.test.MyClass - my message
  - **other layouts: XML, JSON, YAML, CSV,...**

Output specification (..)

Conversion pattern

- **composed of literal text and format control expressions called**
- **conversion specifiers**
  - **start with a percent sign (%), followed by**
  - **(optional) format modifiers:**
    field width, padding, left/right justification
  - **a conversion character: category, priority, date, thread name,...**

Examples:

p: priority

t: thread

m: message

c: category   x: nested diagnostic context

d: date or r: milliseconds

https://logging.apache.org/log4j/2.x/manual/layouts.html#PatternLayout

Output specification (..)

- ***Filter*** **allow Log Events to be evaluated to determine if or how they should be published**
  - **result is an enum that has one of 3 values:**
ACCEPT, DENY or NEUTRAL
  - **types: Regex, Time, Composite, Burst,...**

Configuration

- **default configuration specified in DefaultConfiguration class**
  - **appender: ConsoleAppender**
  - **layout: PatternLayout linked to ConsoleAppender**

    %d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n
  - **log level: Level.ERROR**
- **configuration of Log4j 2 can be accomplished in 1 of 4 ways**
  - **configuration file written in XML, JSON, YAML, or properties format**
  - **programmatically, by creating a ConfigurationFactory and Configuration implementation**
  - **programmatically, by calling the APIs exposed in the Configuration interface to add components to the default configuration**
  - **programmatically, by calling methods on the internal Logger class**

Configuration (..)

- when log4j starts, ConfigurationFactory searched in following order of precedence:
  - "log4j.configurationFile"
    -> load configuration that matches the file extension
  - if no system property is set
    -> log4j2-test.properties in the classpath
  - log4j2-test.yaml or log4j2-test.yml in the classpath
  - log4j2-test.json or log4j2-test.jsn in the classpath
  - log4j2-test.xml in the classpath
  - if a test file cannot be located the properties
    -> log4j2.properties on the classpath
  - log4j2.yaml or log4j2.yml on the classpath
  - log4j2.json or log4j2.jsn on the classpath
  - log4j2.xml on the classpath
  - DefaultConfiguration

Configuration files

- **log4j2.properties**

```
name=PropertiesConfig
property.filename = logs
appenders = console
appender.console.type = Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss}
                                   [%t] %c{1} - %msg%n
rootLogger.level = info
rootLogger.appenderRefs = stdout
rootLogger.appenderRef.stdout.ref = STDOUT
```

- **JSON**

```
{"Configuration": {
    "Appenders": {
        "Console": {
            "PatternLayout": {
                "pattern": "%d{dd-MMM-yyyy HH:mm:ss} [%t] %-5level  - %msg%n"
            },"name": "Console", "target": "SYSTEM_OUT"
        }},
    "Loggers": {
        "Root": {
            "AppenderRef": {"ref": "Console"},
            "level": "trace"
        }
    }}}
```

Configuration files

- **XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36}
                                    %msg%n" />
        </Console>
        <File name="MyFile" fileName="all.log" immediateFlush="false"
               append="false">
            <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
                                    %logger{36} - %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="Console" />
            <AppenderRef ref="MyFile"/>
        </Root>
    </Loggers>
</Configuration>
```

- **initialize these by**
    - **adding them on the classpath in the src(/main/resources) folder**
    - **-Dlog4j.configurationFile=
      file:/c:/logging/exampleconfigs/log4j2.xml (or properties/json)**

Programmatic configuration

- create a class that extends ConfigurationFactory

```
@Plugin(name="CustomConfigurationFactory",
            category=ConfigurationFactory.CATEGORY)
@Order(50)
public class CustomConfigurationFactory extends ConfigurationFactory {
    static Configuration createConfiguration(final String name,
                                            ConfigurationBuilder<BuiltConfiguration> builder) {
        builder.setConfigurationName(name);
        builder.setStatusLevel(Level.WARN);
        AppenderComponentBuilder appenderBuilder
                = builder.newAppender("file", "FILE").addAttribute("fileName", "mylog.log");
        appenderBuilder.add(builder.newLayout("XmlLayout"));
        builder.add(appenderBuilder);
        builder.add(builder.newLogger("be.abis.ch5logging", Level.DEBUG).
        add(builder.newAppenderRef("file")).addAttribute("additivity", true));
        builder.add(builder.newRootLogger(Level.ERROR).
                                        add(builder.newAppenderRef("file")));
        return builder.build();
    }
    @Override
     // some other methods
}
```

- call the configuration via

```
ConfigurationFactory.setConfigurationFactory(new CustomConfigurationFactory())
 or
-Dlog4j.configurationFactory=be.abis.ch5logging.config.CustomConfigurationFactory
```

Introduction to JUnit and Logging

1. Testing principles
2. JUnit Overview
3. JUnit 5 implementation
4. Logging concepts
5. Log4j

APPENDIX A. EXERCISES

Exercise Setup

An exercise skeleton can be found at
L:\Java\ExerciseSkeletons\JavaAdvanced

Create a new IntelliJ project called *JavaAdvancedExercises*. Copy the provided directories under a package called *be.abis.exercise*. Put the *courses.csv* file under the *c:\temp\javacourses* directory. Check out the code. Run the *TestMain* application, to check whether everything works.

1 Functional Programming

1. Implement the “find” methods in the MemoryPersonRepository by using streams. Throw a *PersonNotFoundException* if no person was found.
2. Call the findAllPersons() method in the PersonRepository class. Execute the following “queries” via a stream based on this list. Create a separate Test class for this.
   a. Select all persons whose last name starts with S. Sort them on alphabetical order of first name.
   b. Print a list of all distinct companies.
   c. How many persons are there in the list that work in Leuven?
   d. Who is the youngest person?
   e. Group all persons per company.
   f. How many persons are there per company?
   g. What is the average number of employees per company?

2 Collections and Generics

1. Implement equals() and hashCode() in the Person class. On which properties will you base the methods? Use this in the addPerson() method in the MemoryPersonRepository, to check whether the person was not already present. In case it would, throw a *PersonAlreadyExistsException*.
2. Add a class *TestCollectionLambda*. Call the findAllCourses() method in the CourseRepository class. Perform following actions directly on the list, not via streams:
   a. Sort all courses by title.
   b. Sort all courses by duration and price.
   c. Remove all courses that take less than 3 days.
   d. First create a Map<String,Double> which contains the title of the course as the key, and the price per day as value.
   Use computeIfPresent() to increase the price for each course by 10% .
   Print the contents of the map using the forEach() method.
3. Create a generic class *Printer<T>* with a method *print(T t)*, which calls the toString() methods of the corresponding objects.
4. Add a class *Laptop* with a generic method *callPrinter(),* which takes in a List of objects to print.

3 String Handling, DateTime and I18N

1. In the CourseRepository, add a method *String formatCourse(Course c)*, which prints a course in the following form:
   DB2, an overview;5;550.0;30/4/1986
   Use StringBuilder to achieve this.
2. Add a method printAllCourses(), which prints the courses in a more readable format (as follows):

```
-------------------------------------------------------------------------------------------
                                     Course Overview
-------------------------------------------------------------------------------------------
Course Title                    Total Price with VAT            Release Date
-------------------------------------------------------------------------------------------
Maven                            €544,50                         Jun 11, 2007
Workshop SQL                    €1149,50                         Jan 09, 1990
Programming with Spring         €1905,75                         Mar 21, 2008
Java Programming                €3025,00                         May 27, 1997
DB2, an overview                €3327,50                         Apr 30, 1986
```

3. Add a method *calculateAge()* to the Person class that returns the age of a Person.
4. Use a Test class with a main method to execute the following:
   a. What day is it 3 years, 2 months and 15 days from now?
   b. Which day of the week were you born? Print in your native language.
   c. How many days to go until your next birthday?
   d. How many days old are you today?
5. Internationalize the output of exercise 3.2, such that you can change the language of the labels + the date format.

4 Advanced I/O topics

1.
   a. Create a new class *FileCourseRepository*, that implements the CourseRepository interface. Add a method *Course parseCourse(String s)* to map each line onto a course object.
   b. In the constructor, read in the file using Files.lines. Use the method created above for the mapping.
   c. Implement the addCourse(Course method) by means of Files.newBufferedWriter.
2. Have a look at the file attributes of the courses.csv file.
3. Create a copy *courses2.csv* file in the original directory. Then move that copy in a subdirectory called *inputfiles*.
4. Create a class to find all java source files in your project containing the word *Course*.

5 Concurrency

1. Create a class Counter with a method count() that counts from 1 to 1000. Make the class a thread, and start 3 threads. Use the “implement Runnable” technique.
2. Create 4 threads, but let them perform different tasks (of which 2 are the counter from previous exercise). Implement via lambdas.
   Remove the “implements Runnable” from the Counter class.
3. Make sure the count() method is synchronized.
4. Let the count() method notify one of the other tasks when it is finished. Only then that task can start. Use thread interaction to achieve this.
5. Adjust previous exercise to use Executors and Thread Pools.
6. Add a task to previous executor, which reads in the courses.csv file, and returns the number of lines read. Use a Callable to achieve this.
7. Increase the file size of the courses.csv file (copy the lines at least 50 times). Parallellize the reading, and check what happens. You can drop the counter task while testing.
8. Adapt the count() method of in the Counter class to use AtomicInteger and ThreadLocalRandom.

6 JUnit and Logging

Libraries for Logging can be found at L:\Courses Library\Java\JavaAdvanced

1. Create a JUnit test class for the MemoryPersonRepository. Make sure to create tests for the exceptions as well.
2.
   a. Set-up/configure a logging environment based on Log4J2. The configuration file should be written in XML. Messages for Level.ERROR, should be logged in an exceptions.json file, whereas messages on Level.INFO should be shown on the console.

   b. Add logging to the MemoryPersonRepository class. Add a message on Level.ERROR when an exception occurs (e.g. person by email/pwd not found). An info message should be logged when the login went correct.