Table of Contents

Topics, ingeschat naar prioriteit qua studeren en schatting van de hoofdstukken in het boek (ZIE OOK OVERIGE HOOFDSTUKKEN ONDERAAN)

Belankgrijkste hoofdstukken: 4, App. C, 3, 7, 8, 9, 5, 6. Doorlezen: 2

Paginas (exclusief oefenvragen):

H.4: 172 - 225 (53)
App. C.: 595 - 634 (39)
H.3: 103 - 161 (58)

Totaal 150

H.7: 325 - 393 (68)
H.8: 405 - 443 (38)
H.9: 453 - 496 (43)

Totaal 149

H.5: 233 - 275 (42)
H.6: 283 - 315 (32)

Totaal 74

(H.2: 47 - 94 (47))

LAMBDAS EXAM TOPICS (4, App. C)

- Functional interfaces, `java.util.function`
- Refactor code that uses inner anonymous classes to lambda expressions. Type inference and target typing.
- Develop code that uses the built in interfaces included in the `java.util.function` package such as `Function`, `Consumer`, `Supplier`, `UnaryOperator`, `Predicate`, and Optional APIs, including the primitive and binary variations of the interfaces.
- Develop code that uses a method reference, including refactoring a lambda expression to a method reference

JAVA COLLECTIONS EXAM TOPICS (3, 4, App. C)

- Diamond with generic declarations.
- Iterate collection, filter a collection, sort a collection using lambda expressions.
- Search for data using `findFirst()`, `findAny()`, `anyMatch()`, `allMatch()`, `noneMatch()`
- Calculations on Java Streams using `count`, `max`, `min`, `average`, `sum`. Save results of a collection by using the `collect` method and `Collector` class. `averagingDouble`, `groupingBy`, `joining`, `partitioningBy`
- Java SE 8 Collection improvements: `Collection.removeIf()`, `List.replaceAll()`, `Map.computeIfAbsent()` and `Map.computeIfPresent()` methods.
- `merge()`, `flatMap()` and `map()` methods on Java Streams

JAVA STREAMS EXAM TOPICS (4, 8, 7, 9)

- Stream interfaces and pipelines. Create a stream by using the `Arrays.stream()` and `IntStream.range()`. Identify the lambda operations that are lazy

- Develop code that uses parallel streams, including decomposition operation and reduction operation in streams.

CONCURRENCY EXAM TOPICS (7, App. C)

- `CyclicBarrier` and `CopyOnWriteArrayList` with a focus on advantages and differences from the `java.util` collections
- `Lock`, `ReadWriteLock`, `ReentrantLock` in `java.util.concurrent.locks` and `java.util.concurrent.atomic` packages
- `Executor`, `ExecutorService`, `Executors`, `Callable` and `Future` to execute tasks using Thread pools
- Parallel Fork/Join Framework

LANGUAGE ENHANCEMENTS EXAM TOPICS (4, 6, App. C)

- Switch statements (Strings, binary literals, numeric literals, underscores in literals)
- Try with resource statements, auto-closable resources.
- Multiple exception types in a single catch block
- Static and Default methods

LOCALIZATION EXAM TOPICS (5, App. C)

- Describe advantages of localizing an application and developing code that defines, reads and sets the locale with a `Locale` object.
- Resource bundles
- Time based events using `LocalData`, `LocalTime`, `LocalDateTime`, `Instant`, `Period` and `Duration`
- Formatting of dates, numbers and currency values with `NumberFormat`, `DateFormat`
- Work with dates and times across time zones and manage changes resulting from daylight savings

JAVA FILE I/O EXAM TOPICS (NIO.2) (8, 9)

- Operate on file and directory paths by using the `Paths` class
- Check, delete, copy, move with `File` class
- Recursively access a directory tree by using the `DirectoryStream` and `FileVisitor` interfaces
- Find a file by using the `PathMatcher` interface with the Java 8 specific improvements including `Files.find()`, `Files.walk()` and `lines()` method.
- Observe changes in a directory using the `WatchService` interface

EVENTUEEL DOORLEZEN AAN HET EINDE VAN DE STUDIE:

H.2 en H.6 -> komt wel op het examen maar vrij vaak gebruikt.

NIET LEZEN:

H.1 H. 10

Notes:

Algemeen:

Static inner classes can be instantiated. So this is fine:

```
public class MultipleNested {

  static class Nested() {}

  public static void main(String[] args) {

    Nested nested = new Nested();

  }

}
```

HOOFDSTUK 3 NOTITIES – GENERICS AND COLLECTIONS

There is a large amount of tables in this chapter that you need to memorize for the exam, some only contain explanation, others contain lists of methods that you need to know. The list with the most relevant content that you do not already know by hard:

- Table 3.3, page 121
- Table 3.4, page 130
- Table 3.5, page 135 – 136
- Table 3.6, page 139
- Table 3.7, page 140
- Table 3.8, page 141
- Table 3.9, page 142
- Table 3.10, page 148
- Table 3.11, page 159
- Table 3.12, page 159

Arrays and `ArrayList`

- An `ArrayList` can not contain primitives.
- An `array` can contain objects and primitives.
- Get an item out of a `List` with `get()` and check the size with `size()`
- Access items in the array with brackets and check the size with the `length` variable.
- Convert an array to a list with `Arrays.asList(array);`. The implementation returned by this method is **not resizable**! Strictly the list is still mutable (because you can still invoke .set()), but you can't alter the size of the list by removing and adding elements. You can in that case not call `list.remove(1)`, for example.
- Set an item in a particular index of the list with `list.set(1, "test");` where list is a list of `String`.
- `Arrays.sort(numbers);` sorts the array called numbers in their natural order.
- Strange behavior with `Arrays.binarySearch(numbers, 3)` on page 105: if the item where you search for doesn't exist in the array, it returns one less than the negated index of where the item should be inserted.
- `Collections.sort(list)` sorts the list and `Collections.binarySearch` has the same behavior as described above.

- `List` contains a an overloaded `remove()` method. One takes the `int` of the index to remove, the other one an object to match it with. If you have a list with `Integer` objects and you have an `Integer` with value 5 in it, then you can remove it by invoking: `list.remove(new Integer(5));` This looks strange but apparently the equals method is used here to search for an object that is equal to the given object.

Generics

- Formal type parameter – a generic type at the class level. A class can have multiple formal type parameters. E.g.:

```
public class SizeLimitedCrate<T, U> {

  private T contents;
  private U sizeLimit;

  public SizeLimitedCrate(T contents, U sizeLimit) {
    this.contents = contents;
    this.sizeLimit = sizeLimit;
  }
}
```

  You can use any name you want for the type parameter although a single upper-case letter is a common choice.
- Type erasure – generics are just for compile time. Behind the scenes, the compiler replaces all references to the formal type parameters of a class with `Object`. Allows the code to be compatible with older versions of Java.
- Interfaces can also have formal type parameters. There are three ways you can implement an interface with a formal type parameter:
    - You can specify the type of the parameter in the implementing class
    - You can implement the interface by using a formal type parameter. This way, you can let the calling class decide which type to use
    - You can omit the type parameter altogether. This shows a warning about *raw types* but the code compiles.

Generic methods:

- Can be used for static and normal methods
- Specify the type before the return type. You can return the type itself or a type that takes the type parameter as its type parameter. Example of the first:

```
public static <T> T identity(T t) { return t; }
```

  Example of the second:

```
public static <T> Crate<T> ship(T t) { return new Crate<T>();}
```

- You can call generic methods normally or you can explicitly specify the type like this:

```
Box.<String>ship("packages");
```

Bounds

- A *bounded parameter type* is a generic type that specifies a bound for the generic.
- A *wildcard generic type* is an unknown generic type represented with a question mark.
- Unbounded wildcards: `?`
- Wildcard with an upper bound: `? extends type`
- Wildcard with a lower bound: `? super type`

Unbounded wildcards

- Difference between generics in `ArrayLists` and arrays:
    - You can't assign a `List<Object>` to a `List<String>` but you **can** do this:

      ```
      Object[] o = new String[0];
      ```

    - If you try to put an invalid object in the array, it will throw a `ArrayStoreException` but the declaration succeeds.
    - Because of type erasure the same can not be done with Lists.

Upper-Bounded wildcards

- Take note of the difference between this:

  ```
  ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE
  ```

  And this:

  ```
  ArrayList<? extends Number> list = new ArrayList<Integer>(); // OK
  ```

  You can't just use a subclass with generics but with upper-bounded wildcards you can specify that any subclass of a class is fine (including the class itself).
- In this context "extends" is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).
- When you work with upper bounds or unbounded wildcards, **the list becomes logically immutable (although you can remove items from the list)**. So this doesn't work:

  ```
  static class Sparrow extends Bird {}
  static class Bird{}

  List<? extends Bird> birds = new ArrayList<Bird>();
  birds.add(new Sparrow()); // DOES NOT COMPILE
  birds.add(new Bird()); // DOES NOT COMPILE
  ```

  In other words: a compiler error results from code that attempts to add an item to a list with an unbounded or upper-bounded wildcard.

Lower bounded wildcards

- With lower bound wildcards a list is no longer immutable
- Note the table on page 121, **table 3.3**: you have to know what compiles and what doesn't:
    - If you use an upper bound or a unbounded wildcard, a list become immutable (so you can's add items to the list in the method that is shown on that page as an example)

- If you use a lower bounded wildcard you can add items to the list that you pass as a parameter to the method that takes a list with a lower bound wildcard

- Confusing rules with generics:
  - This is illegal: `List<Object> list = new ArrayList<String>();`
  - However this works fine:

```
public static void addSound(List<Object> sounds) {
    sounds.add("quack");
}

public static void main(String[] args) {
    addSound(new ArrayList<Object> list);
}
```

    In other words: the elements of the list can be of a subclass but the list instantiation must match exactly with the declaration.
  - Very tricky example on page 122: If you declare a lower bound as such:

```
List<? super IOException> exceptions = new
ArrayList<Exception>();
exceptions.add(new Exception()); // DOES NOT COMPILE
exceptions.add(new IOException()); // OK
exceptions.add(new FileNotFoundException()); // OK(!)
```

    Although `FileNotFoundException` is a sub-class of `IOException` it does fit in the list. However, the `Exception` does not fit. It is unclear why these rules are present but seem to have something to do with type erasure and Java not knowing which type to expect. *We could have a `List<IOException>` and `Exception` doesn't fit in there* is the explanation. Maybe clearer with the following explanation:

```
public void m1 (List<? super Number> list) {
    Number n = list.get(0); // WILL NOT COMPILE
}
```

    Read it aloud like this: A List containing instances of `Number` or its super class(es). Thus, this will allow you to do this: `list.add(new Integer(10));` **because `Integer` can be assigned to a variable of type `Number` or its super class**, but it will not allow you to retrieve anything other than `Object`: `Object obj = list.get(i);` because the compiler doesn't know the exact class of objects contained by list.

Nowhere explained in the book but pops up as an example. The formal type parameter of a method can also contain a concrete type with the extends keyword in it. So this is possible:

```
<T extends A> T method(List<T> list) {
    return new B();
}
```

The example above is a generic method with a bounded type parameter. Here you don't have to use the '`?`' in the upper bound declaration of the return type. You specify that you return a subclass of the type parameter. This is also described here:

https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html

But seems to be unrelated to upper bound wildcards. Remember that upper bounded wildcards are used for variables. It is described as "*You can use an upper bounded wildcard to relax the restrictions on a variable*". See:

https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html

So if you use wildcards in generics in a parameter, for example, it must have a `?` in it so this is **not** correct:

```
<X> void method(List<X super B> list) { // DOES NOT COMPILE
}
```

Another example that is legal because we use upper bounded wildcards here:

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;  // OK. List<? extends Integer>
is a subtype of List<? extends Number>
```

Using Lists, Sets, Maps and Queues

The Java collections framework is a set of classes in `java.util` for storing collections.

- `List`: a list is an ordered collection that allows duplicate entries. Elements in a list can be accessed by an int index.
- `Set`: a set is a collection that does not allow duplicate entries.
- `Queue`: a queue is a collection that orders its element in a specific order for processing. Typically firs-in-first-out but other orderings are possible.
- `Map`: a map is a collection that maps keys to values, with no duplicate keys allowed. The elements are key/value pairs.

`List`, `Queue` and `Set` are sub-interfaces of `Collection`. `Map` is not a sub-interface of `Collection` but is part of the collections framework.

Generic methods of the Collection interface that are shared with the implementing classes (note: not `Map`):

- `boolean` **`add`**`(E element)` – some collections always return true after an add (eg. a list allows duplicates so would return true every time you add an instance, even though there is already an existing one) and some have logic to determine if the add was successful. Note that if you add an element to a `Set` that's duplicate, no exception will be thrown. The add will in that case simply return `false`.
- `boolean` **`remove`**`(Object object)` – removes only one match. Returns false in case a match was not found. Is overloaded, can also take the index of the object to remove, that version throws an `IndexOutOfBoundsException` in case you try to refer to an index that is not in the collection.
- `int` **`size`**`()`
- `boolean` **`isEmpty`**`()`
- `void` **`clear`**`()`

- `boolean` **`contains`**`(Object object)` – uses the equals method of the object.

Big O notation

- *O(1)* – constant time. It doesn't matter how large the collection is, the answer will always take the same time to return.
- *O(log n)* – logarithmic time: A logarithm is a mathematical function that frows much more slowly than the data size. Logartihmic time is better than lineair time.
- *O(n)* – linear time: The performance will grow linearly with respect to the size of the collection.
- *O(n^2)* – n squared time: code that has nested loops where each loop goes through the data takes n squared time.

`List` implementations

- `ArrayList`
    - Main benefit is that you can lookup elements in constant time
    - Adding or removing an alement is slower than accessing an element.
    - Good choice in case you are reading more often than (or the same amount as) writing to the `ArrayList`.
- `LinkedList`
    - Main benefit is that you can access, add, remove from beginning and end of the list in constant time.
    - Tradeoff is that dealing with an arbitrary index takes lineair time
    - Implements `List` and `Queue`
    - Has all of the methods of a `List`
    - Has additional methods to facilitate adding or removing from the beginning/end of the list
- `Vector`
    - Replaced by `ArrayList`
    - Slower than `ArrayList`
    - Thread safe (but currently there are better ways to make sure a `List` is thread safe)
    - Shouldn't be used in new code.
- `Stack`
    - Data structure where you add and remove elements from the top of the stack
    - Extends `Vector`
    - Shouldn't be used in new code, use an `ArrayDeque` instead.

`List` methods:

- Deal with indexes
- In addition to the inherited `Collection` methods, table 3.4 on page 130 are the method signatures that you need to memorize (`add`, `get`, `indexOf`, `lastIndexOf`, `remove`, `set`)
- Methods that List redefines are other versions of the same method. E.g. `List` has a add method that doesn't return `boolean` (`void add(E element)`) and a version that doesn't return a `boolean` and takes an additional `int` parameter to indicate where the item must be added (`void add(int index, E element)`)

`Set` implementations:

- `HashSet`
    - Unordered set of elements that does not allow duplicates.
    - Adding elements and checking if an element exists costs constant time

    - Uses the hashcode of the elements to retrieve them to make it more efficient
- `TreeSet`
    - Implements `NavigableSet`
    - The order in which the elements are added is maintained
    - Adding and checking whether an element exists costs logarithmic time (*O(log n)*)

`Set` methods (behavior):

- The `Set` interface doesn't add methods that you need to know for the exam but you do need to know the difference and the behavior
        - The `equals()` method is used to determine equality
        - The `hashCode()` method is used to know in which bucket to look so that Java doesn't have to look through the whole set to find out if an object is there.
        - Best case is that hash codes are unique and Java has to call `equals()` on only one object, worst case is that all implementations return the same `hashCode()` in which case Java has to call `equals()` on every element in the set.
- Memmorize the `NavigableSet` interface methods and what they do on page 134. In short:
    - `E lower(E e)` – returns the largest element that is < e or null if no such element
    - `E floor(E e)` – returns the largest element that is <= e or null if no such element
    - `E ceiling(E e)` – returns the smallest element that is >= e or null if no such element
    - `E higher(E e)` – returns the smallest element that is > e or null if no such element

`Queue` implementations

- `LinkedList`
    - See above.
    - The `LinkedList` is a double ended queue
        - Benefit is that you can add and remove items from both ends of the queue
    - Implements both `List` and `Queue`
    - Tradeoff is that it isn't as efficient as a "pure" `Queue`
- `ArrayDeque`
    - is a "pure" double-ended queue
    - main benefit is that it is more efficient than a `LinkedList`

`Queue` methods (`ArrayDeque`)

- Memmorize table 3.5 on page 135 – 136 which sums up the methods of `ArrayDeque` that were added on top of the already existing `Collection` methods. Table 3.5 is somewhat strangely ordered. To summarize you can group the following methods
    - `boolean add(E e), void push(E e), boolean offer(E e)`. These methods add elements to the queue.
        - `add()` is for queue, adds an element to the back of the queue and returns true or throws and exception.
        - `offer()` is for queue and adds an element to the back of the queue, returns whether it was successful.
        - `push()` adds an element to the front of the queue and is for queue and stack.

    - `E element()`, `E remove()`, `E poll()`, `E peek()`, `E pop()`. All of these methods return an element from the queue but the behavior differs
        - `element()` returns next element or trows an exception and is for queue.
        - `remove()` returns and removes the next element or throws an exception in case the queue is empty. Is for queue.
        - `poll()` returns and removes the next element or returns null if empty queue. Is for queue.
        - `peek()` returns next element or null in case empty queue. Is for queue and stack.
        - `pop()` remove and returns next element or throws an exception if empty queue. Is for stack.
- Also note the difference between using the `ArrayDeque` as a queue or a stack
- offer/poll/peek are more common and standard for use with queues.
- Queues can contain duplicate elements

`Map` implementations

- `HashMap`
    - Stores the keys in a hash table (uses `hashCode()` of the keys to retrieve values more efficiently).
    - Main benefit is that retrieving and adding by key have constant time.
    - Does not maintain order
- `TreeMap`
    - Stores the keys in a sorted tree structure.
    - Main benefit is that keys are always in sorted order.
    - Tradeoff is that adding and checking if a key is present are both O(log n) (logarithmic)
- `Hashtable`
    - Old and thread safe. Don't use it.

`Map` methods

- Table 3.6 on page 139 lists the methods that you need to know for `Map`. Doesn't extend `Collection`, more methods are specified on than that interface.

Comparing collection types – see table 3.7 and 3.8. Note that `Set` doesn't have to be added/removed in a specific order and that **`Queue` can have duplicate elements**.

Tricky: the chapter doesn't mention sorted and whether a type uses the `compareTo` method up to the page where table 3.8 pops up (page 141). None of the types are sorted by default (while you might think that this is the case for `List`) except for `TreeSet` and `TreeMap`. Those are also the only two implementations that use the `compareTo` method. Whether an implementation uses compareTo or not is different from whether an implementation uses the `equals` method in case it was unable to find an element in the `Set` by `hashCode`. They are different concepts. A `Queue` is said to be unsorted but the elements are ordered in a specific way for processing. `List` orders the items in which you add them but they are not sorted by, for example, their natural order. Only Tree implementations immediately sort the elements as they get added.

Data structures that **do not** allow nulls:

- `TreeMap` (null values are ok but not null keys) because of the sort functionality.

- `TreeSet` because of the sort functionality.
- `ArrayDeque`
- `Hashtable` (no null keys or values)

Comparator vs. Comparable

- For string objects, order is defined according to the Unicode character mapping: numbers sort before letters and uppercase letters sort before lowercase letters.
- Sort objects you define by implementing `Comparable`.
- You can sort a list of objects by invoking `Collections.sort()`
- `Comparator` – a class to specify that you want to sort in a different order than the object itself provides.
- Comparable (`java.lang.Comparable`)
    - Interface with one method:

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

    - Returns < 0 if current object is smaller than the argument;
    - Returns > 0 if current object is larger than the argument;
    - Returns 0 if current object is equal to the argument.
    - Remember that in case you implement order for an int that implementing it this way:

      ```
      id - a.id
      ```

      .. sorts in ascending order. Implementing it shis way:

      ```
      a.id - id
      ```
- Comparator (`java.util.Comparator`)
    - Interface that has a method called `int compare(T o1, T o2)` (as opposed to `compareTo` in `Comparable` interface)
    - Used to define a sort order different from what is implemented in `Comparable` or for classes that do not implement the `Comparable` interface.
    - Can be implemented by using an anonymous class or by using lambda expressions (since it is a functional interface).

- `Comparator` makes sense to implement as a lambda, for `Comparable` it doesn't.
- It is common to sort nulls before any other values
- Comparator contains default and static methods to chain the comparison if you want to compare multiple fields:

  ```
  Comparator<Squirrel> c = Comparator.comparing(s -> s.getSpecies());
  c = c.thenComparingInt(s -> s.getWeight());
  return c.compare(s1, s2);
  ```

- It also offers a reverseOrder method to sort in reverse order (`Comparator.reverseOrder()`).

Searching and Sorting

- `Arrays` and `Collections` have overloaded `sort()` and `binarySearch()` methods.
- They take an optional `Comparator` parameter.
- The list or array must be sorted before it is searched using the same definition of order for both.
- `Collections.sort` version with single parameter requires objects in the collection to sort to implement the `Comparable` interface.
- Both `sort` and `binarySearch` allow you to pass in a `Comparator` when you don't want to use the natural order.
- `Collections.binarySearch()` allows you to search for an item using a `Comparator` implementation. If you pass in a comparator that sorts in reverse order and the list you provide is sorted ascending, the result when you call `binarySearch` is undefined:

```
List<String> names = Arrays.asList("Fluffy", "Hoppy");
Comparator<String> c = Comparator.reverseOrder();
int index = Collections.binarySearch(names, "Hoppy", c);
System.out.println(index); // Returns -1 ie. undefined.
```

- If you use add objects to a sorted collection like TreeSet, the compiler doesn't check if the added object implements Comparable but throws an exception at runtime.
- You can instantiate a `TreeSet` with a comparator:

```
Set<Rabbit> rabbit = new TreeSet<>(new Comparator<Rabbit>() {
    public int compare(Rabbit r1, Rabbit r2) {
        return r1.i = r2.id;
    }
});
rabbit.add(new Rabbit());
```

Additions in Java 8

- Method references (actually something that belongs to chapter 4 but somehow described in chapter 3 too)
    - `::` tells java to automatically pass the parameters to a method
    - Four formats for method references:
        - Static methods:

```
Consumer<List<Integer>> methodRef = Collections::sort;
```

The lambda version is:

```
Consumer<List<Integer>> lambda = l ->
Collections.sort(l);
```

`Collections.sort` is overloaded (there is also a version that accepts a `Comparator`) but because we are implementing a `Consumer` in this example, which takes only one parameter, Java knows that we want to use the single parameter version of the sort method in this case.

- Instance methods on a specific instance:

```
String str = "abc";
Predicate<String> methodRef2 = str::startsWith;
```

The lambda version would be:

```
Predicate<String> lambda2 = s -> str.startsWith(s);
```

- Instance methods on an instance to be determined at runtime:

```
Predicate<String> methodRef3 = String::isEmpty
```

In this case Java will invoke the `isEmpty` method on the string that is passed to the functional interface implementation at runtime. The lambda version would look like this:

```
Predicate<String> lambda3 = s -> s.isEmpty();
```

Tricky: you can use method references only in case the parameter that is used is the instance variable itself. So this implementation:

```
list.removeIf(s -> s.startsWith("A"));
```

Does not allow you to use a method reference (ie. `String::startsWith`) because the parameter to the `startsWith` method is not the parameter that is passed to the test method of the `Predicate` interface.

- Constructors (constructor reference):

```
Supplier<ArrayList> methodRef4 = ArrayList::new;
```

Uses the `Supplier` functional interface to instantiate an `ArrayList`. The lambda expression would look like:

```
Supplier<ArrayLIst> lambda4 = () -> new ArrayList();
```

- Removing conditionally:
  - Specify what needs to be removed with:

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

  - Can be called on implementations of the Collections interface like `List`.

- Updating all elements:
  - New method on the Lists interface
  - Method signature is:

```
void replaceAll(UnaryOperator<E> o);
```

- Looping through a `Collection`
    - Use the `forEach` construct with a lambda expression.
    - Takes a `Consumer` implementation
    - Example: `cats.forEach(System.out::println)`
- New Java 8 Map APIs
    - `putIfAbsent()` puts an item in a `Map` if its key is absent or if the value of the key is null. E.g. `favorites.putIfAbsent("Sam", "Tram");` (given that favorites is a `Map<String, String>`)
    - `merge()` allows adding logic to the problem of what to choose. It needs a mapping implementation of `BiFunction` to add this logic.
        - Example looks as follows:

```
BiFunction<String, String, String> mapper = (v1, v2) ->
v1.length() > v2.length ? v1 : v2;

Map<String, String> favorites = new HashMap<>();
favorites.put("Jenny", "Bus tour");
favorites.put("Tom", "Tram");

String jenny = favorites.merge("Jenny", "Skyride",
mapper);
String tom = favorites.merge("Tom", "Skyride", mapper);

System.out.println(favorites); // {Tom = Skyride, Jenny =
Bus tour}
System.out.println(jenny); // Bus Tour
System.out.println(tom); // Skyride
```

        The merge is called on the `Collection` type, in this case a `HashMap`. The map is updated only in case the entry has a value that is larger than the given value ("Skyride").

        - Also has logic for what happens in case values in the Map are `null` or missing keys are present. In this case it doesn't use the `BiFunction` at all but simply uses the new value.

        Remember that the value of the key is replaced with the result of the mapping function that is provided in the implementation of the `BiFunction`. The `BiFunction`'s `apply` method parameters are the existing value (that is looked up using the key that is also provided as a parameter to the merge method) and the new value (which is the second parameter of the merge method). The rules are best described in the Javadoc of the function:

        "If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null."

        Strange: if the key is not in the map and the function's result is null, the key

is added to the Map.

- So as stated above, if the mapping function is called and returns `null` the key on which the mapping was called is removed. Note that keys that do not exist are still added to the map. For example if you have a map with two keys ("Jenny" and "Tom") and a mapper that returns `null` where you call the `map.merge` method with a key that doesn't exist: `favorites.merge("Sam", "Skyride", mapper)`, a key "Sam" with value "Skyride" is added to the `Map`).

- `computeIfPresent` and `computeIfAbsent`
  - `computeIfPresent()` if the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
  - `computeIfAbsent()` runs only if the key isn't present or its associated value is `null`. Uses a `Function` since there is no value already in the map. Only the key is passed as the input. If the method returns `null` it doesn't add the key (since the key was never added in the first place).
- For completeness the signatures of the methods:

  default V **computeIfAbsent**(K key, Function<? super K,? extends V> mappingFunction)

  default V **computeIfPresent**(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)

  default V **merge**(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)

- Tables 3.11 and 3.12 give an overview of the methods and what happens in certain scenarios.

HOOFDSTUK 4 NOTITIES – FUNCTIONAL PROGRAMMING

Tables in chapter 4:

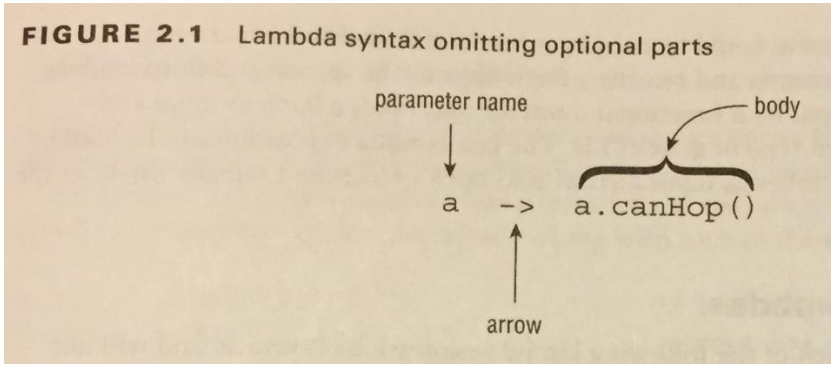
- tabel 4.1 blz. 173 (MEMORIZE)
- tabel 4.2 blz. 184
- tabel 4.3 blz. 187
- tabel 4.4 blz. 190
- tabel 4.5 blz. 204
- tabel 4.6 blz. 207 (MEMORIZE)
- tabel 4.7 blz. 208 (MEMORIZE)
- tabel 4.8 blz. 209 (MEMORIZE)
- tabel 4.9 blz. 210 (MEMORIZE)
- tabel 4.10 blz. 212
- tabel 4.11 blz. 218 (?)

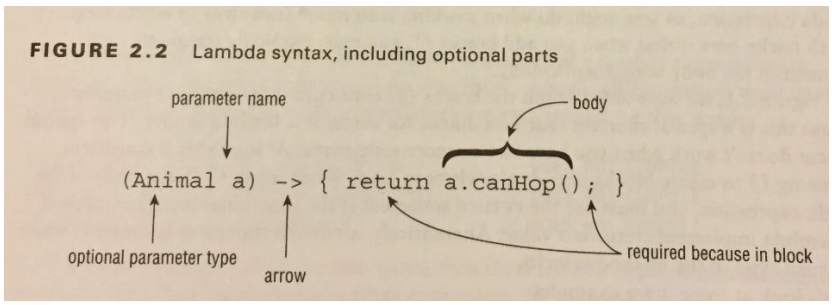
Lambda content from chapter 2:

- Functional interface has exactly one abstract method. Can be implemented by a concrete class or by a lambda expression.
- Is optionally annotated with `@FunctionalInterface`
    - Not mandatory but recommended
    - Adding methods in addition to the existing abstract method will cause a compilation error in case the interface is annotated but this will not be the case if it is not annotated, possibly breaking code that already uses it as a functional interface
    - Sub-interfaces (extending the functional interface) may contain:
        - No method at all (since they extend the functional interface which already contains a method)
        - Override the method
        - Static methods with an implementation (note that you can declare and implement static methods at te interface level)
        - Default methods
- A lambda expression "*is a block of code that gets passed around, like an anonymous method*"
- Additional explanation to example on page 55: with a lambda expression you implement the functional interface. Methods that accept a functional interface can have the implementation of the interface passed to it. So:

```
a -> a.canHop();
```

  …is the implementation of the `CheckTrait` interface and was passed to the `print(…)` method which accepted a `CheckTrait` interface as parameter. The return type of the implementation must be compatible with the method that is defined in the functional interface.
- Two variants of the same:





- Parentheses `()` can be omitted in a lambda expression:
    - if there is exactly one input parameter
    - if the type is not explicitly stated in the expression.

- Expressions that have zero or more than one input parameters therefore require parentheses.
- Multiple lines of code require braces: `{}` and a semicolon to terminate statements.
- Some examples with 0 or 1 parameters (look strange but are all valid):

```
() -> true                          // 0 parameters
a -> { return a.startsWith("test");}   // 1 parameter
(String a) -> a.startsWith("test")     // 1 parameter
(int x) -> {}                       // 1 parameter
(int y) -> {return;}                // 1 parameter
```

- Other examples with 2 parameters (in these examples, one of the parameters is not used):

```
(a, b) -> a.startsWith("test");
(String a, String b) -> a.startsWith("test");
```

- Providing the data types in lambda expression parameters is optional but if you provide it for one parameter, you must provide them for the rest too.
- You can't re-declare a parameter that is provided with a lambda expression (you can declare other variables). So this is false:

```
(a, b) -> { int a = 0; return 5; } // DOES NOT COMPILE
```

  But this is fine:

```
(a, b) -> {int c = 0; return 5;} // OK
```

- Deferred execution: code is specified now but runs later.
- Java provides a bunch of functional interfaces for common problems. In chapter 2 an example was implemented where a `test()` method was defined. This logic, however, is also provided in the `java.util.function.Predicate` functional interface.
- You can also cast in lambda expressions. This can come in handy if there is an ambiguity in the lambda implementations (see page 342 – 343):

```
use((Callable<Integer>)() -> {throw new IOException("");});
```

Chapter 4 notes:

- Effectively final – if you can add the final modifier to a local variable, it is "effectively final" (so the variable didn't change in the rest of the code)
- Lambda's use the same access rules as inner classes. Lambda expressions can access:
  - Static variables
  - Instance variables
  - Effectively final method parameters
  - Effectively final local variables
- Strange remark about the interface `Runnable` on page 174: "(…) *All you need to know is that Runnable doesn't take any parameters, return any data, or use generics.*" Looking in the Javadoc it says that `Runnable` is a functional interface. It has a `run()` method that has a return type of `void`.
- Something about the `Supplier` functional interface and other functional interfaces. If you see this:

```
Supplier<LocalDate> s1 = LocalDate::now;
```

… then `LocalDate::now` is the implementation of the functional interface's method `get()`. So after this you can call:

```
LocalDate localDate = s1.get();
```

… to instantiate an instance of `LocalDate`. Obviously quite useless but just an example.

Another notation for the implementation is:

```
Supplier<LocalDate> s2 = () -> LocalDate.now();
```

Of course this can also be done directly in the lambda expression where it is appropriate. Nested generics are also possible:

```
Supplier<ArrayList<String>> s1 = ArrayList<String>::new;
ArrayList<String> list = s1.get();
```

Note that you can't use the diamond operator in the above example (`ArrayList<>::new` is **invalid however** it can be used if you explicitly define a parameterless lambda expression: `() -> new ArrayList<>();`)

- Also note situations where a functional interface takes two parameters, like the `BiConsumer` interface. This:

```
Map<String, Integer> map = new HashMap<>();
BiConsumer<String, Integer> b1 = map::put;
```

…infers that the parameters to put are going to be provided when you call the `accept()` method of the `BiConsumer` like this:

```
b1.accept("test", 1);
```

… this puts an item in the `map` variable. The static kind of call that we saw earlier to a factory method is not allowed here (e.g. `Map::put` – it's the implementation of the `accept()` method that doesn't return anything and gets two parameters in this interface. It has access to the local variable: `map` that we instantiated before declaring the `BiConsumer b1`).

If a functional interface returns a value, then you can use the `::` kind of calling a method. So this is allowed:

```
Predicate<String> p = String::isEmpty;
```

… becaue the functional interface's method `test()` returns a value (of type `boolean`). Another way of writing the above is:

```
Predicate<String> p = x -> x.isEmpty();
```

- If a functional interface has two parameters, Java can infer their values when you call it later on. The implementation can be as simple as:

  ```
  BiFunction<String, String, String> b1 = String::concat;
  ```

  This tells Java to implement the `apply()` function by taking the two parameters and catting them together and finally return a `String`. The alternative form of writing the above interface implementation would be:

  ```
  BiFunction<String, String, String> b1 = (aString, toAdd) ->
  aString.concat(toAdd);
  ```

  Calling this would look as follows:

  ```
  System.out.println(b1.apply("put", "together")); // puttogether
  ```

- The `UnaryOperator` and `BinaryOperator` expect the parameter(-s) type and the return type to be the same. That's why when you declare these function interfaces, you only specify one generic type like this:

  ```
  BinaryOperator<String> b1 = String::concat
  ```

  The above does the same as the `BiFunction` example above.

- Watch out for trick questions:

  Which functional interface would you use if you want to return a `Boolean` and take an `Integer` as a parameter?

  The answer is `Function`, **not** `Predicate` because `Predicate` returns a `boolean` (the primitive type) and not a `Boolean`.

Optionals

- `Optional.ofNullable(value);` <- the value may be `null`, if it is this method returns an `Optional.empty()` optional otherwise an `Optional.of(value)`.
- Note that `Optional.orElse()` doesn't take a supplier but a type `T`. So this does **not** work:

  ```
  opt.orElse (IllegalStateException::new); // DOES NOT COMPILE
  ```

  `IllegalStateException::new` is the implementation of a `Supplier` while `orElse` doesn't take a `Supplier`. You would have to use the `orElseThrow(Supplier s)` instead.
- Three methods that take functional parameters:
  - `ifPresent(Consumer c)`
  - `orElseGet(Supplier s)`
  - `orElseThrow(Supplier s)`

Streams

- Stream (`java.util.stream.Stream`) concepts:

    - Data is created when needed (lazy)
    - Pipeline – all operations in a stream occur in a pipeline
    - Stream operations – operations that occur on the elements of the stream
        - Intermediate operations – transforms the stream into another one. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.
        - Terminal operations – produces a result. Since streams can be used only once, the stream is no longer valid after a terminal operation completes. Only one terminal operation can exist per stream. Not before a terminal operation is reached will the pipeline start.
    - Infinite stream – the stream doesn't end unless there is a terminal operation
    - Not all terminal operations terminate the stream in case the stream is infinite
    - The three parts of a pipeline:
        - Source: where the stream comes from
        - Intermediate operations
        - Terminal operation
- Beware of questions in the exam where they are using streams a second time. You can only use a stream once so this would throw an exception:

```
Predicate<? super String> predicate = s -> s.length() > 3;
Stream<String> stream = Stream.iterate("-", (s) -> s + s);
boolean b1 = stream.noneMatch(predicate);
boolean b2 = stream.anyMatch(predicate); // INVALID: stream was already
terminated
System.out.println(b1 + " " + b2);
```

Stream Sources:

- Create a *finite* `Stream` using factory methods like:
    - `Stream<String> empty = Stream.empty();`
    - `Stream<Integer> single = Stream.of(1);`
    - `Stream<Integer> from Array = Stream.of(1, 2, 3);`
- Or convert from `List` to `Stream`:

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> fromList = list.stream();
Stream<String> fromListParallel = list.parallelStream();
```

- Create an infinite `Stream` using generate or iterate:
    - `Stream<Double> randoms = Stream.generate(Math::random);`
    - `Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);`
- `iterate()` is a method that takes the seed, the starting point, as its first parameter and a lambda expression as its second. The lambda expression gets passed in the previous value and generates the next value.
- Infinite streams may be turned into a finite stream using operations.

Terminal operations:

- Can be performed without any intermediate operations, not the other way around (note: with this they mean that you can only get a result out of a stream if you have at least a terminal operator. Of you only have an intermediate operator, the code still compiles but nothing happens).

- *Reductions* is a special kind of terminal operation where all of the contents of the stream are combined into a single primitive or `Object` (e.g. an `int` or a `Collection`).
- `long` **`count`**`()`
    - Determines the number of elements in a finite stream
    - Hangs for an infinite stream
    - Is a reduction because it looks at each element and returns one value
- `Optional<T>` **`min`**`(Comparator <? super T> Comparator)` / `Optional<T>` **`max`**`(Comparator <? super T> Comparator)`
    - Find the smallest or largest value in a finite stream according to the given sort order.
    - Hang on infinite streams
    - Are reductions, return a single value after looking at the entire stream.
    - Returns an `Optional` because there are streams where no min/max could be found.
- `Optional<T>` **`findFirst`**`() / Optional<T>` **`findAny`**`()`
    - Return an element in the stream unless the stream is empty
    - If the stream is empty return empty `Optional`
    - Work with infinite streams and finite streams.
    - `findAny()` is useful when you are working with a parallel stream because it gives Java the flexibility to return to you the first element it comes by rather than the one that needs to be first in the stream based on the intermediate operations.
    - Note that the example in the book is a little bit strange:

      ```
      Stream<String> infinite = Stream.generate(() -> "chimp");
      infinite.findAny().ifPresent(System.out::println); // chimp
      ```

      Normally you would use it in combination with intermediate operations like `filter()`
    - Are not reductions
- `boolean` **`anyMatch`**`(Predicate <? Super T> predicate)` / **`allMatch`** / **`noneMatch`**
    - Search a stream and return information about how the stream pertains to the predicate.
    - May or may not terminate for infinite streams (depends on data)
    - Are not reductions
- `void` **`forEach`**`(Consumer<? Super T> action)`
    - Loop construct for stream, does not terminate on infinite streams
    - You can call forEach on a `Collection` or on a stream
    - Traditional for loops are <u>not</u> possible on a `Stream` (they don't implement `Iterable`)
- Reduce has multiple variants with the following signatures:

  ```
  T reduce(T identity, BinaryOperator<T> accumulator)
  Optional<T> reduce(BinaryOperator<T> accumulator)
  <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator,
  BinaryOperator<U> combiner)
  ```

    - Most common way to use a reduction is to start with an initial value and keep merging it with the next value.

- Example for multiplying the integers in a stream:

```
Stream<Integer> s = Stream.of(1, 5, 4, 2);
Integer sum = s.reduce(1, (a, b) -> a*b);
```

- Identity is not always necessary. If you don't specify one, an `Optional` is returned as no data may be present (you are using the second form of the reduce method as described). Three choices for what could be in the optional:
  - If the stream is empty, an empty `Optional` is returned
  - If the stream has one element, it is returned
  - If the stream has multiple elements, the accumulator is applied to combine them.
- The parameter of the first two reduce methods is a `BinaryOperator` and the third form of the reduce method also takes a `BinaryOperator` as its combiner. So what you specify as a parameter in the reduce method is the implementation of that functional interface. Recall that the `BinaryOperator` takes two parameters and returns a value that has the same type as the parameters.
- The third method signature is used when we are processing collections in parallel but it doesn't mean that you have to use it on a parallel stream.
- Note that the accumulator parameter is a `BiFunction` and the combiner is a `BinaryOperator`. The accumulator operates on the stream elements and this may be done in parallel. The combiner then combines the parts of the reduction that may have been handled by parallel streams into one value.

  The example in the book operates on a non-parallel stream and basically just multiplies all the entries again. It also uses the same `BinaryOperator` for the two parameters. Apparantly a sub-interface of `BiFunction` is also allowed as the first parameter:

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> stream = Stream.of(1, 4, 2, 3, 4);
stream.reduce(1, op, op);
```

- `collect()` is a special type of reduction called a *mutable reduction*. The signatures are:

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, <? super T>
accumulator, BiConsumer<R, R> combiner)
<R, A> R collect(Collector <? super T, A, R> collector)
```

- It is more efficient than a regular reduction because we use the same mutable object while accumulating.
- Common mutable objects include `StringBuilder` and `ArrayList`.
- Lets us get data out of streams and into another form
- The first signature:
  - Here, too, the combiner is used to combine parallel stream outcomes to one collection. The example of the book uses a `StringBuilder` to generate a specific `String` using the collect method:

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
```

```
StringBuilder word = stream.collect(StringBuilder::new,
StringBuider::append, StringBuilder::append);
```

Again the stream is not a parallel stream but demonstrates how you can use the collect method.

- Note that the book mentions that it is important to know how to implement your own collector (see page 195). In the second signature the collect method expects a `Collector`
- Java provides an interface with common collectors. Examples:

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> strings =
Stream.collect(Collectors.toCollection(TreeSet::new));
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.of(Collectors.toSet()); // makes no
guarantees as to which set you will get.
```

- Common intermediate operations:
  - `filter()`
  - `distinct()`
  - `limit()` and `skip()`
  - `map()`
  - `flatMap()`
  - `sorted()`
  - `peek()`

Descriptions and explanations below – you don't have to memorize the signature but understand what the methods do.

- `Stream<T>` **`filter`**`(Predicate<? super T> predicate)` – returns a `Stream` with elements that match the given expression.
- `Stream<T>` **`distinct`**`()` – returns a stream with duplicate items removed.
- `Stream<T>` **`limit`**`(int maxSize)` and `Stream<T>` **`skip`**`(int n)`
  - make a stream smaller or make an infinite stream finite

```
Stream.iterate(1, n -> n +
1).skip(5).limit(2).forEach(System.out::println); //67
```

Makes an infinite stream finite by skipping the first five and limit them by 2.

- `<R> Stream<R>` **`map`**`(Function <? super T, ? extends R> mapper)`
  - For transforming data
  - Example (convert list of string to list of integers):

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.map(String::length).forEach(System.out::println); // 676
```

- `<R> Stream<R>` **`flatMap`**`(Function<? super T, ? extends Stream<? extends R>> mapper)`

    - Method takes each element in the stream and makes any elements it contains top-level elements in a single stream
    - Helpful when you want to remove empty elements in a stream or want to combine a stream of lists

      The example creates three lists and puts them in a stream like this:

      ```
      Stream<List<String>> animals = Stream.of(zero, one, two); //
      zero, one and two are instances of List<String>
      animals.flatMap(l -> l.stream()).forEach(System.out::println);
      ```

      Notice how the `Stream` (`animals`) was combined and the elements were put in one stream and iterated with a `forEach`.
- `Stream<T> sorted()`
  `Stream<T> sorted(Comparator<? Super T> comparator)`
    - Returns a stream with the elements sorted.
    - Uses natural ordering unless we provide a comparator
    - Can take a `Comparator` implementation
    - **Warning** in the book about compatibility of how you provide the comparator. You can't use `Comparator::reverseOrder` (the method reference) if you want to use `Comparator.reverseOrder()` in the sorted method because the signature of the sorted method that takes a `Comparator` lets you implement the `Comparator` functional interface. This method takes two `String` parameters and returns an `int`. As `Comparator.reverseOrder()` doesn't do that, you can't pass it as a method reference (see page 199)
- `Stream<T> peek(Consumer<? super T> action)`
    - Useful for debugging because it allows us to perform a stream operation without changing the content
    - Note that although peek is not meant to change the content, Java does not prevent you from writing a consumer (the parameter of the peek method) that changes the content.
- Keep in mind that there are a lot of questions about how the stream is processed. Step 1 to 8 in the book on pages 201 and 202 illustrate that well. Basically, all of the intermediate operations look at each element in the stream. If you instantiate an infinite stream and then tell it to sort, it will never stop running, even if you have a limit thereafter:

  ```
  Stream.generate(() -> "Elsa")
  .filter(n -> n.length() == 4)
  .sorted()
  .limit(2)
  .forEach(System.out::println) // WILL HANG
  ```

  If you switch the sorted and limit then the stream iterates two times and turns into a finite stream (doesn't hang).

Primitive streams

- Add common functionality that would otherwise be less easy to do (like calculating the average of int elements)
- Three types of primitive Streams:

    - `IntStream` – for `byte`, `char`, `short`, `int`
    - `DoubleStream` – for `double` and `float`
    - `LongStream` – for `long`

- Creating a primitive type stream can be done in several ways:
    - Similar to Stream:
        - `DoubleStream emptyStream = DoubleStream.empty()`
        - `IntStream intStream = IntStream.of(1, 2, 3);`
        - `DoubleStream random = DoubleStream.generate(Math::random);`
        - `DoubleStream fractions = DoubleStream.iterate(.5, d -> d/2);`
    - Taking a range of numbers:
        - `IntStream.range(1, 6); // 1, 2, 3, 4, 5, not including 6`
        - `IntStream.rangeClosed(1, 5) // 1, 2, 3, 4, 5, 6 (so inclusive range)`
    - Creating from another Stream type:
        - See table 4.6 p. 207. Basically, all stream types contain `mapTo…` methods. If you map to the same type, you use the `map()` method.
        - A mapping function must be provided as the parameter to these functions.
        - See table 4.7 p.208 for these functions. The functions have predictable names:
            - From `Stream` to another `Stream` requires a `Function` parameter
            - From primitive type to `Stream` requires a `DoubleFunction`, `IntFunction` and `LongFunction` for the corresponding types
            - From `Stream` to primitive type `Stream` requires a `ToDoubleFunction`, `ToIntFunction` and `ToLongFunction`
            - From primitive type to the same type requires a `DoubleUnaryOperator`, `IntUnaryOperator`, `LongUnaryOperator`
            - From one primitive type `Stream` to another requires `DoubleToIntFunction`, `DoubleToLongFunction`, `IntToDoubleFunction`, `IntToLongFunction`, `LongToDoubleFunction` and `LongToIntFunction`.

Primitive optionals

- There are separate `Optional` classes for primitives:
    - `DoubleOptional`
    - `IntOptional`
    - `LongOptional`
- Are for primitive optionals (so `Optional<Double>` is for `Double`, `OptionalDouble` is for `double`)
- Terminal operations like `min()` or `findAny()` return the corresponding `Optional` type for primitives (`sum()` doesn't return an `Optional`). `avg()` always returns an `OptionalDouble`.
- Summarizing statistics can be done with the summary statistics classes:
    - `IntSummaryStatistics`
    - `LongSummaryStatistics`
    - `DoubleSummaryStatistics`
- Book shows an example with `IntSummaryStatistics`. You can declare it by calling a method on a primitive stream:

```
IntSummaryStatistics stats = ints.summaryStatistics();
```

Where ints is a `IntStream`. Then you can invoke methods like

- `getMax()`
- `getMin()`
- `getCount()`

Functional interfaces for primitives

- `BooleanSupplier` with one method: boolean `getAsBoolean()`
- `BooleanSupplier` is taken as a separate paragraph in the book because it doesn't apply to the primitive streams that were mentioned earlier. They have separate functional interfaces in table 4.9 (p. 211) which you have to memorize.
- There are no `BiConsumer`, `BiFunction` and `BiPredicate` interface for the primitive streams.
- Additionally, primitive functional interfaces exist which you must also memorize (table 4.10 p. 212 – 213)

Advanced stream pipeline concepts

- Remember that the `Optional.map()` method takes a `Function` as the parameter and therefore returns a specific type of `Object`. It does not work if that method returns an `Optional`. In that case you can use `flatMap()` instead
- There is a huge list of available collectors that you can pass to the `collect()` method (by using the `Collectors` class and pass that to collect). Example:

```
stream.collect(Collectors.joining(", ");
```

The collectors are summed up in table 4.11 on page 218.

- You have to know the `Collectors` methods that are mentioned in the book. Question 11 uses the `Collectors.joining()` method which expects the elements that it gets to be of type `String`.

- Basic collectors allow you to…
  - …implement joining functional interfaces (using the `Collectors.joining()` method),
  - …collect the average and other reductions by using collectors like `Collectors.averagingInt()`,
  - … collect the elements to a specific type of collection with methods like `Collectors.toCollection()` supplying the type of collection you want the elements in the stream to be put in.
- Collecting elements in Maps
  - `Collectors.toMap()` -> you need to provide the implementation for the functions to collect the keys and the values for the map like:

```
Stream<String> stream = Stream.of("lions", "tigers", "bears");
Map<String, Integer> map = stream.collect(Collectors.toMap(s->s, String::length); // creates a map with string keys and the length of the strings as values
```

- Returning the same value in a lambda as the parameter (like in the above example: `s-> s`) can also be done by invoking `Function.identity()`
- The `toMap()` method also provides an overloaded version where you can provide a merge implementation and a version where you can provide the type of map to create. See examples on page 220.

- Collecting using grouping, partitioning and mapping
  - `groupingBy()` tells the `collect()` method that it should group all of the elements of the stream into lists and organize them by the function that you provide. The keys in the map are the function value and the result are the values.
  - So you declare a `Map` with a certain type as the key and a `List` with a certain type as the values. For example:

    ```
    Map<Integer, List<String>> map =
    stream.collect(Collectors.groupingBy(String::length));
    ```

    This example takes a stream of strings and groups them by their length. The strings are separated by a comma.
  - Another version of the `groupingBy` method allows you to provide a so called downstream collector which can do something special with the values. You can, for example to use a `Set` rather than a `List` as the results in the map's values you can invoke:

    ```
    stream.collect(Collectors.groupingBy(String::length,
    Collectors.toSet()));
    ```

    This results in a `Map<Integer, Set<String>>`.
  - Another version lets you provide the type of map you want to create :

    ```
    stream.collect(Collectors.groupingBy(String::length,
    TreeMap::new, Collectors.toSet()));
    ```

    This results in a `TreeMap<Integer, Set<String>>`

  - **Partitioning** results in two groups: true and false. It splits a list into two parts. It takes a `Predicate` implementation like:

    ```
    Stream<String> stream = Stream.of("lions", "tigers", "bears");
    Map<Boolean, List<String>> map =
    stream.collect(Collectors.partitioningBy(s -> s.length <= 5);
    ```

  - If there are no values for a particular `Boolean` key, it is still inserted in the map (but with an empty value)
  - You can change the type of `List` by using an overloaded version of the `partitioningBy` method that accepts a `Collector`:

    ```
    Map<Boolean, Set<String>> map =
    stream.collect(Collectors.partitioningBy(s -> s.length <= 5,
    Collectors.toSet()));
    ```
  - There is no possibility to change the type of map in the `partitioningBy` method.

- For methods that take a *downstream collector* we can use any of the collectors. For example grouping by the length of strings grouping by how many we have of each length:

  ```
  Map<Integer, Long> map =
  stream.collect(Collectors.groupingBy(String::length,
  Collectors.counting()));
  ```

  This creates a map with two keys (5 and 6) and values 2 and 1 (assuming we collect on the stream of animals above).

- The **`mapping()`** collector provides the functionality to go down a level and add another collector. Most complicated example you should know for the exam:

  ```
  Stream<String> stream = Stream.of("lions", "tigers", "bears");
  Map<Integer, Optional<Character>> map =
  stream.collect(Collectors.groupingBy(String::length,
  Collectors.mapping(s -> s.charAt(0),
  Collectors.minBy(Comparator.naturalOrder()))));
  ```

  The above would contain a map with keys 5 and 6 and an `Optional` with value 'b' for 5 and an `Optional` with value 't' for 6. First it groups by the lengths of the strings (two are 5 in size and one is 6 in size). Then it takes the first character of the elements and returns the smallest character by natural order.

Additional notes from Enthuware tests:

- An interface can have abstract methods (among static and default methods). Keep in mind that it is legal to put "abstract" in the declaration of the method but that it is redundant. So this is fine:

  ```
  interface F {
      default void m() {}
      abstract void n();
  }
  ```

- If you use the primitive suppliers (`DoubleSupplier`, `IntSupplier`, `LongSupplier`) they can automatically unbox primitives as well. So this is fine:

  ```
  DoubleSupplier ds1 = b1::getPrice; // Assuming that getPrice returns the
  Double wrapper
  ```

  Of course, if `getPrice()` would return a null value it would throw a `NPE`.

Additional notes from the questions:

- At one point they ask which methods in the `Stream` class are reductions. In one of the answers they have `sum()`. Although this is a reduction, it is not of the `Stream` class but the `IntStream` class. Beware of these kinds of pitfalls.
- Watch out with the iterate function. If you use a post increment operator, this operator is not applied to the incrementor so this would print infinite 1s:

```
Stream.iterate(1, x -> x++);
```

If you want them to be incremented then you have to use `++x`.

HOOFDSTUK 7 NOTITIES - CONCURRENCY

Tables: 7.1 page 329; table 7.2 page 338 – 339; table 7.3 page 340; table 7.4 page 341; table 7.5 page 346; table 7.6 page 348; table 7.7 and 7.8 page 353 – 354; table 7.9 page 360; table 7.10 and 7.11 page 362 – 363; table 7.12 page 365

- A Thread is the smallest unit of execution that can be scheduled by the OS
- A task is a single unit of work performed by a thread.
    - Will commonly be implemented as a lambda expression
    - A thread can complete multiple independent tasks but only one task at a time
- Types of Threads:
    - System thread – created by the JVM, runs in the background (e.g. the `gc`). When a system thread encounters a problem it can not recover from it generates an `Error` and not an `Exception`
    - User-defined thread – created by the developer to accomplish a specific task.
- Most Java applications are multi-threaded, even if they only contain one user defined thread that calls the `main()` method.
- If an application only contains one user-defined thread we call it a single threaded application
- Thread scheduling is done by the Thread scheduler.
- Context switch – when the threads allotted time is complete but the thread has not finished processing
- A thread can interrupt or supersede another thread if it has a higher thread priotiy than the other thread. The most important priorities are stored in the `Thread` class:
    - `Thread.MIN_PRIORITY` – 1
    - `Thread.NORM_PRIORITY` – 5 (default)
    - `Thread.MAX_PRIORITY` – 10

`java.lang.Runnable`

- A functional interface that takes no parameters and doesn't return a value (method is `void run()`)
- Beware that lambda expressions that return a value are incompatible with the `Runnable` interface. The following examples are therefore **invalid**:

```
() -> "" // INVALID
() -> 5 // INVALID
() – {return new Object();} // INVALID
```

The following examples are **valid**:

```
() -> System.out.println("Hello world"); // OK
() -> {inti=10; i++;} // OK
() -> {return;} // OK
() -> {} // OK
```

Creating a `Thread`

- Executing a task with a `Thread` is a two step process:
    - Define the `Thread` with the corresponding task to do
    - Start the Thread by calling `Thread.start()`
- Defining a task can be done in two ways:
    - Provide a `Runnable` object or lambda expression to the `Thread` constructor
    - Create a class that extends `Thread` and overrides the `run()` method
- Providing a `Runnable` object: you can define a class that implements the `Runnable` object and instantiate that and pass it to a `Thread` like this:

```
public static void main(String[] args) {
    (new Thread(new PrintData())).start();
}
```

  Where `PrintData` implements the `Runnable` interface. Obviously the above could have also been done by passing a lambda expression to the Thread constructor.

- Invoking `start()` on the `Thread` starts the `Thread` in a separate thread on the operating system.
- Extending the `Thread` class and overriding the run method is less common.
- Important to realize that the `run()` method (with which you can also start a `Thread`) **does not** start the thread in a separate processing thread on the operating system. Instead, it runs in the same thread as the one that called it. So this will run sequentially:

```
new PrintData().run();
(new Thread(new PrintData())).run();
(new ReadInventoryThread()).run();
```

  Invoking `start()`, however, will run them in a separate processing thread making the order of when the threads are run indeterminate. The order within a single `Thread` is still linear, of course (e.g. the `for()` loop in a `Thread` would still be ordered).
- It is encouraged to use the Concurrency API instead of creating your own threads.

Polling with `sleep()`

- `Thread.sleep()` pauses the current thread for a specified number of milliseconds.
- If you start a separate `Thread` in the main method and then call `Thread.sleep()`, it pauses the `Thread` that is associated with the `main()` method while the user defined thread continues to work
- Throws `InterruptedException`.

The `ExecutorService` (`java.util.concurrent`)

- Is an interface that can be instantiated using the `Executors` factory.
- `ExecutorService.newSingleThreadExecutor()` gives you a version that handles tasks in the order in which they were added to the executor.
- `execute()` lets it perform asynchronous tasks. Takes a `Runnable` so can be implemented with a lambda expression and returns void (fire-and-forget):

```
ExecutorService service = null;
```

```
try {
    service.execute(() -> {for (int i = 0; I <3; i++)
System.out.println("Pringing record: " +i);});
    (… other calls to execute…)
} finally {
    if(service!=null) service.shutdown();
}
```

- You **must** call the `shutdown()` method after use. If you don't the application will never terminate. Upon calling `shutdown()` the tasks that have already been submitted will still be executed until there are no more tasks. If tasks are submitted while the `ExecutorService`'s `shutdown()` is called, a `RejectedExecutionException` will be thrown.
- Note that the `main()` method runs in a separate thread so in case additional logic is present after calling the `execute()` method on the `ExecutorService`, they may be processed earlier than the tasks that are submitted to the `ExecutorService`.
- `shutdownNow()` *attempts* to stop all running tasks and returns a `List<Runnable>` of tasks that were submitted to the thread executor but never startd.
- `ExecutorService` does not implement `AutoClosable`

Submitting tasks to the `ExecutorService`

- `execute()` doesn't return anything so results are not directly available to the calling thread.
- Table 7.2 to memorize – 5 methods to execute tasks
- The methods that take a collection object containing a list of tasks to execute, execute synchronously. Unlike the other methods these methods will wait until the results are available before returning control to the enclosing program.
    - `invokeAll()` returns a `List` of ordered `Future` objects with one `Future` object corresponding to each submitted task in the order they were in the original collection.
    - Even though `Future.isDone()` returns trye for each element in the returned `List`, a task could have completed normally or thrown an exception.
    - Will wait indefinitely until all tasks are complete.
    - `invokeAny()` executes a collection of tasks and returns the result of one of the tasks that successfully completes execution, cancelling all unfinished tasks
    - Any completed task may be returned (although the first task to finish is often returned)
    - Method will wait indefiniteily until at least one task completes.
    - Overloaded versions of these methods exist that take a `timeout` value and a `TimeUnit` parameter.

Waiting for results

- Table 7.3, page 340: `Future` methods. The `get()` method on a `Future` object obtained from the `submit()` method that takes a `Runnable` will always return null. Calling `get()` on a `Future` method that comes back from a `submit()` used with a `Callable` returns the actual result or `null` in case there is no result.

`Callable`

- Functional interface similar to `Runnable` but its `call()` method returns a value and throws a checked exception.

- The result is actually the result of the task so if you define something like this:

```
Future<Integer> result = service.submit(() -> 30 +11);
```

And then print the result like this:

```
System.out.println(result.get()); // Prints 41.
```

Waiting for all tasks to finish

- If you don't need the results of the `Future` objects, you can also call `awaitTermination(long timeout, TimeUnit unit)` on the `ExecutorService` object. You do this after calling shutdown on the `ExecutorService`.
- You can call `isTerminated()` on the `ExecutorService` to determine if all the tasks have been finished.

Scheduling tasks

- `ScheduledExecutorService` (sub-interface of `ExecutorService`) can be used to schedule (repeated) tasks.
- Instantiate using the `Executors` factory:

```
ScheduledExecutorService service =
Executors.newSingleThreadScheduledExecutor();
```

- Memmorize methods in table 7.5 page 346. They return a `ScheduledFuture<V>` which is identical to `Future` except that it also has a `getDelay()` method that returns the delay when it was created. The `scheduleAtFixedRate()` and `scheduleAtFixedDelay()` take a `Runnable` instance as a parameter because it would otherwise endlessly generate `Future` objects if it would be a `Callable`.

Increasing concurrency with pools

- Memmorize table 7.6, page 348.
- Pooled thread executors can execute tasks in parallel until the pool is exhausted. In that case the task will be queued by the thread executor and wait to e executed.
- Use `newCachedThreadPool()` only for many short lived asynchronous tasks, not for long-lived processes.
- With a `ScheduledExecutorService` obtained with the `newScheduledThreadPool()`, the creation of an endless set of tasks can be avoided if the pool size is sufficiently large. A thread is returned to the thread pool when a thread finishes and result in new threads being available for the next tasks as they come up.

Synchronizing data access

- `Concurrency` API added a new package: `java.util.concurrent.atomic` to help coordinate access to primitive values and object references.
- Table 7.7 lists the atomic classes and table 7.8 lists the common acomic methods that you must know.
- To synchronize access: use a *monitor* aka *lock*.
    - Use an Object a a monitor with the `synchronized` keyword

```
SheepManager manager = new SheepManager();
```

```
synchronized(manager) {
    // Work to be completed by one thread at a time
}
```

- Using the current object as the monitor can be done with (see page 355):

```
synchronized(this) {
…
}
```

- You can even lock on any object, just as long as it is the same object:

```
private final Object lock = new Object();
private void incrementAndReport() {
    synchronized(lock) {
      System.out.print((++sheepCount)+" ");
    }
}
```

- Automatically synchronizing on the object itself can be done with the synchronized keyword on the method itself, which is equivalent to using `synchronized(this)`:

```
private synchronized void incrementAndReport() {
   …
}
```

- You can also synchronize on static methods in which case the class is used as the monitor. This can also be done on the method and with the synchronized keyword but using the class instead. These two are equivalent:

```
public static void printDaysWork() {
    synchronized(SheepManager.class) {
      ..
    }
}

public static synchronized void printDaysWork() {
  …
}
```

Using concurrent Collections

- Concurrent collections allows developers to use collections in multiple threads without having to manage synchronizing access to them. Instead of using the synchronized keyword on methods that access a certain `Collection` class, `Map` for example, you could use the `ConcurrentHashMap` implementation instead.
- Purpose of the concurrent collection classes is to solve memory consistency errors.
- Note that `ConcurrentModificationExceptions` may also be thrown even if you don't access objects with multiple Threads. See example on page 359 where a key of a map is removed in a for loop. This throws a `ConcurrentModificationException` because the corresponding Iterator is not updated when the remove occurs. If you would use a `ConcurrentHashMap` instead, however, this would be the case and no exception will be thrown.
- Memmorize table 7.9 on page 360.

- The `BlockingQueue` interface, implemented by `LinkedBlockingQueue` and `LinkedBlockingDeque` introduces two methods that allows to wait for an item to be added and to be retrieved (table 7.10 page 361 - 362):
    - `offer(E element, long timeout, TimeUnit unit)` – adds an item to the queue waiting the specified time, returning `false` if time elapses before space is available.
    - `poll(long timeout, TimeUnit unit)` – Retrieves and removes an item from the queue, waiting the specified time, returning `null` if the time elapsed before the item is available.
- Each of these methods throw a checked `InterruptedException` so must be handled.
- A `LinkedBlockingDeque` maintains a linked list between elements.
- A `LinkedBlockingDeque` can obviously also use the above methods without the timing aspect (so just the one that is offered by the `Queue` interface – `offer(E element)`)
- The `BlockingDeque` interface, implemented by `LinkedBlockingDeque`, also offers methods with time parameters (table 7.11 page 362 – 363):
    - `offerFirst(E e, long timeout, TimeUnit unit)` – Adds an item to the front of the queue, waiting a specifie time, returning `false` if time elapses before space is available.
    - `offerLast(E e, long timeout, TimeUnit unit)` – Adds an item to the tail of the queue, waiting a specified time, returning `false` if time elspses before space is available.
    - `pollFirst(long timeout, TimeUnit unit)` – Retrieves and removes an item from the front of the queue, waiting the specifid time, returning `null` if the time slapses before the item is available.
    - `pollLast(long timeout, TimeUnit unit)` – Retrieves and removes an item from the tail of the queue, waiting the specified time, returning `null` if the time elapses before the item is available.
- Each also throw a `InterruptedException` and must therefore be handled.
- Remember that a `LinkedBlockingDeque`, as it implements both the `BlockingDeque` and `BlockingQueue` interfaces, can use the all the methods that are summed up in the tables on pages 361 – 363.
- The `SkipList` classes `ConcurrentSkipListSet` and `ConcurrentSkipListMap` are counterparts of `TreeSet` and `TreeMap` resp.
- Are sorted concurrent collections.
- The `CopyOnWriteArrayList` and `CopyOnWriteArraySet` copy all of their elements to a new underlying structure anytime an element is added, modified or removed from the collection.
    - *Modified*: the **reference** is changed (not the contents)
    - Useful in multi-threaded environments that need to iterate the collection. Used in situations where reads are far more common than writes (writes are very expensive)
    - Any iterator established prior to a modification will not see the changes but will iterate over the original elements prior to the modification.
- If you know at the time of creation that your object requires synchronization, then you should use one of the concurrent collection classes from table 7.9. If you are working with an existing collection that is not a concurrent class and need to access it among multiple threads, you can wrap them using the methods in table 7.12 on page 365.
    - Are methods from the `Collections` class
    - Do not synchronize iterators that you may create from the synchronized collection.

- You need to use a synchronization block if you need to iterate over any of the returned collections in table 7.12:

```
List<Integer> list = Collections.synchronizedList(new
ArrayList<>(Arrays.asList(4, 3, 53)));
synchronized(list) {
  for(int i : list) {
    System.out.print(" " + i);
  }
}
```

- Throw an exception if they are modified within an oterator by a single thread (unlike the concurrent collections): `ConcurrentModificationException`

Working with parallel streams

- *Serial stream* - a stream in which the results are ordered with only one entry being processed at a time.
- *Parallel stream* – a stream that is capable of processing results concurrently, using multiple threads.
- Two ways of creating a parallel stream:
  - Using `parallel()` on an existing stream. Is an intermediate operation on an existing stream. Example:

```
Arrays.asList(1, 2, 4, 6, 8).stream().parallel();
```

  - Using `parallelStream()`. This creates a stream out of a `Collection`. The interface `Collection` offers this method. Example:

```
Arrays.asList(1, 2, 3, 6, 8).parallelStream();
```

- Independent operations are operations that do not require or impact the results of another element of the stream.
- Many common streams including `map()`, `forEach()`, and `filter()` can be processed independently although order is never guaranteed.
- Note that this is especially visible when you introduce side effects in a lambda expression. For example:

```
Arrays.asList("jackal", "kangaroo", "lemur").parallelStream().map(s ->
{System.out.println(s);return
s.toUpperCase();}).forEach(System.out::println);
```

This can result in a list of output that is unpredictable. Like this:

```
kangaroo
KANGAROO
lemur
jackal
JACKAL
LEMUR
```

When using streams, you should avoid any lambda expressions that can produce side effects.

- *You should remember that parallel streams can process results independently, although the order of the results cannot be determined ahead of time.*
- A *stateful lambda expression* is one whose result depends on any state that migt change during the execution of a pipeline. A *stateless lambda expression* is one whose result does not depend on any state that might change during the execution of a pipeline.

  See interesting example on page 371 where it is demonstrated that the `forEachOrdered` method maintains the order in which the list was before entering the stream even though it is a parallel stream but that as soon as you print the collection it created that the order may be completely different. The key here is that by modifying the data in parallel causes unpredictable results.

- It is strongly recommended to avoid stateful operations when using parallel streams (and in general also when you use serial streams).
- Any time you are working with a collection with a parallel stream, it is recommended that you use a concurrent collection.
- Reduction operations on parallel streams are referred to as *parallel reductions*. The results can be different from what you expect when working with serial streams.
- Reduction example with `findAny` on page 372 -373:
    - if you invoke `findAny()` on a `parallelStream`, the JVM selects the first thread to finish the task and retrieves its data while on a serial stream the return value is consistent.
- Stream operations based on order (`findFirst()`, `limit()`, `skip()` etc.) may perform more slowly due to the coordination that has to take place. The results, however, will be consistent with a serial stream. Interesting example on page 373 with a combination of reductions in a chain `skip(5).limit(2).findFirst()`: will return the same result on ordered serial and parallel streams.
- Combining results with `reduce()` – recall that the first parameter is the ***identity***, the second is the ***accumulator*** and the third parameter is the ***combiner***.
    - In the three-argument version of `reduce()`, the accumulator is a `BiFunction`, while the combiner is a `BinaryOperator`.
    - The `Streams` API prevents the problem where the order of the stream elements are not maintained when performing reductions as long as the arguments to the `reduce()` operation adhere to certain principles:
        - The *identity* must be defined such that for all elements in the stream `u`, `combiner.apply(identity, u)` is equal to `u`.
        - The *accumulator* operator `op` must be associative and stateless such that `(a op b) op c` is equal to `a op (b or c)`.
        - The *combiner* operator must also be associative and stateless and compatible with the identity, such that for all `u` and `t` `combiner.apply(u, accumulator.apply(identity, t))` is equal to `accumulator.apply(u, t)`

          Notes to explain the above in own words:

        - The first statement says that if you use the identity and feed them to the implementation of the combiner function (the `BinaryOperator`), then the identity should not change the element. So in the example on page 374 (top)

the identity was an empty string. The combiner was a concatenation of the element and the string. It doesn't matter which element you picked, adding an empty string to it doesn't change the element.
        - The second statement talks about operators between the elements and their behavior. It doesn't matter which values a, b or c have if the operator is counting them up, for example. The operator is said to be associative. This is not the case when you use the subtraction operator. If a = 2, b = 3 and a – b = c so that c in this case is -1, the same can not be true for a – b or c because 2 – (-1) would be 3.
        - For the combiner the rules are the same. The parts of the streams and the operations to combine them to one result must be predictable.
    - Although one and two argument versions of the `reduce()` method do support parallel processing, it is recommended that you use the three-argument version of `reduce()` when working with parallel streams.
- `collect()` also has a three argument version that takes accumulator, and combiner operators, along with a supplier operator instead of an identity.
    - The accumulator and combiner operations must also be associative and stateless, with the combiner operation compatible with the accumulator operator. In this manner, the `collect()` method can be performed as a parallel reduction.
    - You should use a concurrent collection to combine the results, ensuring that the result of concurrent threads do not result in a `ConcurrentModificationException`.
- Using the one argument `collect()` method
    - The one-argument version of `collect()` takes a collector argument: `Set<String> set = stream.collect(Collectors.toSet());`
    - If you want to ensure that a parallel reduction will be performed efficiently in Java using a collector, the following requirements apply (note: the book only talks about the fact that if you do not abide the rules that this only reduces the performance. It doesn't state that it doesn't work – see page 376 - 377)
        - The stream is parallel
        - The parameter of the collect operation has the `Collector.Characteristics.CONCURRENT` characteristic.
        - Either the stream is unordered, or the collector has the characteristic `Collector.Characteristics.UNORDERED`
    - Any class that implements the `Collector` interface includes a `characteristics()` method that returns a set of available attributes for the collector.
    - The book shows two examples of a parallel reduction with a parallel stream. The first uses `Collectors.toConcurrentMap()` and the second a `Collectors.groupingByConcurrent()` method.

Managing concurrent processes

- Two classes that allow you to coordinate tasks among a group of related threads that you must know for the exam: `CyclicBarrier` and `ForkJoinPool`
- `CyclicBarrier`
    - Is used with a limit and an optional `Runnable`, which is executed as soon as the limit is reached.
    - You instantiate a `CyclicBarrier` with its constructor

    - You call the `await()` method on the `CyclicBarrier` instance to let it wait until the threads have handled the code before that until the limit is reached.
    - After a `CyclicBarrier` is broken, all threads are released and the number of threads waiting on the `CyclicBarrier` goes back to zero.
    - If you use a thread pool, make sure that you set the number of available threads to at least the amount of the limit of the `CyclicBarrier`, otherwise the code will hang waiting on the limit of the `CyclicBarrier` to be reached.
- `ForkJoinPool`
    - Use when you don't know upfront how many thread and tasks we needed to perform.
    - Relies on the concept of recursion
        - *Base case*: a non-recursive method that is used to terminate the recursive path.
        - *Recursive case*: a recursive method that may call itself one or multiple times to solve a problem.
        - Always make sure that a recursive solution arrives at its base case at some point (not doing so would result in a infinite loop and ultimately a `StackOverflowError`)
    - Applying the fork/join framework requires us to perform three steps:
        1. Create a `ForkJoinTask` – requires defining the recursive process.
        2. Create a `ForkJoinPool`
        3. Start the `ForkJoinTask`
    - For the exam you must know how to implement a ForkJoin solution by extending one of two classes (both implement `ForkJoinTask` interface):
        - `RecursiveAction`
            - The example on page 383 starts off by defining arguments on which the task will operate.
            - They then override the `compute()` method in which they define the base case and the recusive case.
            - The base case in this example tests if the `end - start <= 3` because they want each task to perform at most three recursions (the example is implementing a weighing action for zoo workers). In the example, they also perform the actual task of the action.
            - In the example the recursive path will only take place if there are more than three iterations left, in which case the task is split up.
            - In the recursive part they define how to separate these tasks. They call `invokeAll` which is an inherited method with a new instance of the implemented `RecursiveAction` (in this example the `WeighAnimalAction`). In the example they pass two instances, one task handles the task from the start to the middle and the other task from the middle to the end.
            - In the main method they actually start the task by instantiating the `Task` itself and a `ForkJoinPool`:

```
ForkJoinTask<?> task = new
WeightAnimalAction(weights, 0, weights.length); //
weights is a Double array of size 10.
ForkJoinPool pool = new ForkJoinPool();
```

```
pool.invoke(task);
```

- `RecursiveTask`
  - `RecursiveTask` uses the `fork()` and `join()` methods to perform actions in separate threads and combine the results.
  - The `RecursiveTask`'s `compute()` method returns a value so the example in the book updates the used example for the `RecursiveAction` by returning a `Double` in the `compute()` method. The base case just returns the `Double` value of the operation it did (as an example counted the sum of the weights so far.
  - In that method, the recursive case is quite different because that uses the `fork()` and `join()` methods to divide and combine the tasks:

    ```
    RecursiveTask<Double> otherTask = new
    WeighAnimalTask(weights, start, middle);
    otherTask.fork();

    return new WeighAnimalTask(weights, middle,
    end).compute() + otherTask.join();
    ```

  - The **`fork()`** method instructs the fork/join framework to perform that action in a separate thread while the **`join()`** method causes the current thread to wait for the results.
  - The implementation of instantiating the pool and invoking the tasks is identical except that the `invoke()` method now returns a value (a `Double` in this case). So that is used as follows:

    ```
    Double sum = pool.invoke(task);
    ```

  - Be careful that the `join()` method instructs the current thread to wait for the results and that the order in which you call `fork()` and `join()` are important. The **following example** executes **with single threaded performance**:

    ```
    RecursiveTask<Double> otherTask = new
    WeighAnimalTask(weights, start, middle);
    Double otherResult = otherTask.fork().join();
    return new WeighAnimalTask(weights, middle,
    end).compute() + otherResult;
    ```

    Here, the current thread first waits on the start – middle task and then proceeds with the task that handles the middle – end part.

- Tips from the book to identify Fork/Join issues copied here:
  - The class should extend `RecursiveAction` or `RecursiveTask`

- If the class extends `RecursiveAction`, it should override a `protected void compute()` method that takes no arguments and returns `void`.
- If the class extends `RecursiveTask`, then it should override a `protected V compute()` method that takes no arguments and returns a generic type listed in the class definition.
- The `invokeAll()` method takes two instances of the fork/join class and does not return a result.
- The `fork()` method casuses a new task to be submitted to the pool and is similar to the thread executor `submit()` method.
- The `join()` method is called after the `fork()` method and causes the current thread to wait for the results of a subtask.
- Unlike `fork()`, calling `compute()` within a `compute()` method causes the task to wait for the results of the subtask.
- The `fork()` method should be called before the current thread performs a `compute()` operation, with `join()` called to read the results afterward.
- Since `compute()` takes no arguments, the constructor of the class is often used to pass instructions to the task.

Identifying threading problems

- Liveness is the ability of the application to be able to execute in a timely manner.
- Three types of liveness problems that you should know for the exam:
    - *Deadlock* – occurs when two or more threads are blocked forever, each waiting on the other. Common strategy to solve this is for threads to order their resource requests.
    - *Starvation* – occurs when a single thread is perpetually (voortdurend) denied access to a shared resource or lock. The thread is still active, but it is unable to complete its work as a result of other threads constantly taking the resource that they are trying to access.
    - *Livelock* – occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. Livelock is a special case of resource starvation in which two or more threads actively try to acquire a set of locks, are unable to do so, and restart part of the process.
- A race condition is an undesirable result that occurs when two tasks, which should be completed sequentially, are completed at the same time.
- Race conditions lead to invalid data if they are not properly handled.

Notes from tests of Enthuware:

- Although not specifically mentioned in the book, the Enthuware exams stress that it is important to know the contents of the `ConcurrentMap` interface of which `ConcurrentHashMap` is an implementation. It says:

  It is a `Map` providing additional atomic `putIfAbsent`, `remove`, and `replace` methods.

  Note that they also expect you to know that `putIfAbsent`, `remove` and `replace` methods

are already in the `Map` interface and are not specifically added to the `ConcurrentMap` interface.

Questions:

2 – both `Callable call()` and `Runnable run()` can throw unchecked exceptions. Tricky about this question is that it is confusing because only `call()` throws a checked exception (which doesn't mean that both methods can not throw unchecked exceptions).

3 – here, a `ScheduledExecutorService` was assigned to `ExecutorService` while `Executors.newSingleThreadScheduledExecotur()` returns in instance that implements `ScheduledExecutorService`. You must know this difference, you **can't** assign a `ScheduledExecutorService` to an `ExecutorService`.

4 – keep in mind that if you use a non-atomic variable, like the `long[]` in this question, that the outcome of incrementing it using several threads at the same time can not be determined.

7 – Confusion here about the call to `Arrays.asList()` in the code because in chapter 3 it is said that if you construct a `List` out of an array, that the list is not structurally mutable (you can't add and remove items). The `CopyOnWriteArrayList` in that question is backed by the list that is constructed using the `Arrays.asList()` call but apparently you can still add items to the `CopyOnWriteArrayList` because the list itself (the original list) is not actually modified here (since upon every modification a copy is created). Taking that into account option A is correct.

8 – although `sorted()` is called, the `findAny()` just returns the value value from the first thread that retrieves a record. Therefor the outcome can not be predicted.

9 – there are so many details about this question that it is unreastic that you can memorize this but…:

- …although `compute()` is a protected method you are allowed to make it public as you are allowed to make the overridden method more accessible (not the other way around).
- Because the compute method in this example returns a value, MyTask extends RecursiveTask (the only thing that you memorized correctly when you filled in the questions)
- Because the code calls `join()` immediately after fork(), the process waits until the compute has returned. It therefore produces single-threaded performance time.

10 – here, the three argument version of the reduce method is used. You must memorize the fact that the second argument is a `BiFunction` and how this behaves if you see implementations. Here, the statement is that c1 is a `String` and c2 is an int. So the first parameter of the `BiFunction` is an element of the stream and the second parameter is part of the result. In this case, the code tries to invoke "`length()`" on an `int`.

12 – because the base condition is never met, in infinite loop will be the result causing an error at runtime (`StackOverflowError`).

13 – the call to `flatMap()` should have triggered you that this returns a new stream that is single threaded by default. So the `collect()` is always execute in a single-threaded fashion here.

14 – the `offerFirst`, `offerLast`, `pollFirst` and `pollLast` methods with timing parameters all throw the checked exception InterruptedException which should have handled (or declared to be

thrown) here. Also note that even if the method declared that it threw that exception that the outcome can not be determined beforehand.

21 – I doubt that this is coming back in the exam but beware that the question is "*…. which of the following are **possible results**…*" so what is in the method is really up to your imagination and can therefore have any content whatsoever. What is key here is that the rest of the implementation runs fine without compilation errors and that the rest of the options can all be true (because the method that is part of the task that is submitted can have any possible implementation within the boundaries of the signature that is provided).

22 – the `execute()` method returns `void`.

HOOFDSTUK 8 NOTITIES - IO

- Memmorize table 8.1, page 409. Probably only worth memorizing what the methods do instead of memorizing them all. Table 8.2 on page 417 -> note: `BufferedInputStream` and `BufferedOutputStream` are not listed but are being described in the chapter. You have to know them too: are the high level versions of the `FileInputStream` and `FileOutputStream` for reading and writing in a buffered manner. From an existing `InputStream`/`OutputStream`. Example:

  ```
  InputStream is = new BufferedInputStream(new
  FileInputStream("filename"))
  ```

  Then reading and writing as shown on page 423.

  `InputStreamReader` and `OutputStreamReader` are listed but deemed to be "out-of-scope" for the exam. Furthermore, figure 8.3 on page 436 shows additional classes (`FilterInputStream`, `FilterOutputStream`) but you **don't have to know them**.

- Differences between stream classes and readers/writers:
  - The stream classes are used for inputting and outputting all types of binary or bytes data (so also strings, for example).
  - The reader/writer classes are used for reading and writing characters or strings only.

- They are also streams although they do not carry the word in the name.
- For the exam you only need to be familiar with the low-level stream classes that operate on files. The rest of the non-abstract stream classes are all high-level streams.
- Always wrap a file stream with a Buffered class in practice.
- Four abstract classes that are the parents of all stream classes defined within the API:

1. `InputStream`
2. `OutputStream`
3. `Reader`
4. `Writer`

- Review of `java.io` class properties:

    - A class with the word `InputStream` or `OutputStream` in its name is used for reading or writing binary data, respectively
    - A class with the word `Reader` or `Writer` in it is used for reading or writing character or string data, respectively
    - Most, but not all, input classes have a corresponding output class (exception you need to know for the exam: `PrintWriter` has no accompanying `PrintReader` class)
    - A low-level stream connects to the source directly
    - A high-level stream is built on top of another stream using wrapping (operations are filtered and applied to the original instance)
    - A class with `Buffered` in its name reads or writes data in groups of bytes or characters and often improves performance in sequential file systems.

- `close()`, `flush()`, `markSupported()`, `mark()`, `mark(int)`, `reset()`, `skip(long)`:
    - always close streams preferably with try-with-resource statements *
    - `flush()` to write cache to disk
    - `mark()` to go back to a point in the stream with `reset()`
    - first check if mark is supported with `markSupported()`, failing to do so may throw an exception
    - calling `reset()` while mark limit has passed, an exception at runtime may be thrown (may, because it depends on the implementation)
    - `skip(long)` returns number of skipped bytes or 0/-1 in case no bytes are skipped (end of stream). Skip can be faster than read for large values

* try-with-resource statements can also contain multiple resources:

```
try (InputStream is = new FileInputStream(source); OutputStream os = new
FileOutputStream(destination)) // BOTH RESOURCES AUTOMATICALLY CLOSED {…}
```

- `FileInputStream`
    - `read()` -> return an `int` but read bytes so that it can return -1 if at end of file.
    - Overloaded versions exist e.g. to read a byte array used by `Buffered` classes.
- `FileOutputStream`
    - `write()` -> write successive bytes using `write(int)` method.
    - Overloaded versions of the write method exists that allow a byte array to be passed and can be used by `Buffered` classes.
- `BufferedInputStream`, `BufferedOutputStream`
    - With `BufferedInputStream` you use the `read(byte[])` method. Returns the number of bytes that have been read or `0` if end of file.
    - `BufferedOutputStream` is called with `write(byte[], int, int)` method -> `byte[]` to write, `offset` and `length`.
- `FileReader`, `FileWriter`
    - Have `read()`/`write()` methods that work with `char` values although use an `int` to hold the data so that read can return `-1` to indicate end-of-file.
    - `close()` and `flush()` methods which work the same as the `InputStream`/`OutputStream` classes.
    - Writer class offers a `write(String)` method to write a `String` immediately to the stream.
- `BufferedReader`/`BufferedWriter`

    - `BufferedReader` offers a `readLine()` method. `BufferedWriter` has the `write(String)` method and the `newLine()` method.
- `ObjectInputStream`/`ObjectOutputStream`
    - Serialize and deserialize objects of classes that implement the `Serializable` interface (marker interface, good practice but not mandatory is to provide a `serialVersionUID` and maintain it, updating it if the class changes). `transient` and `static` members are skipped in to serialization process. Marking as `transient` will also avoid throwing a `NotSerializableException`.
- `ObjectOutputStream`
    - `writeObject(Object)`: throws a `NotSerializableException` in case the object or an embedded reference to the object is not serializable.
- `ObjectInputStream`
    - `readObject()` returns `Object` (mean that it must be cast). Throws `EOFException` if end of file is reached.
    - Voorbeeld chainen van streams voor het serializen van objecten uit een bestand:

      ```
      try(ObjectInputStream in = new ObjectInputStream(new
      BufferedInputStream(new FileInputStream(filename)))) {...} //
      in.readObject -> lezen van object.
      ```

    - Never use the "`available()`" method on the `InputStream` to check for the end of the stream. Rather, catch `EOFException` if relevant. `available()` can also return 0 even though there are still contents to be read.
    - `Object` stream classes support reading and writing `null` values so if you read from a serialized data stream it is always important to check for `null` (or use the `instanceof` operator for `Object` streams)
    - Upon deserialization the constructor and any default initializations are ignored (along with `transient` and `static` fields).

      When deserializing Java calls the first non-argument constructor for the first nonserializable parent class, skipping the constructors of any serialized class in between.

- `PrintStream`/`PrintWriter`
    - `System.out` and `System.err` are `PrintStream` objects
    - Methods to memorize (are in both `PrintStream` and `PrintWriter`):
        - `write() throws IOException`
        - `print()` – overloaded with all primitive types and `Object`. In general calls `valueOf` and calls `write()` method.
        - `println()` - similar to print but adds a system dependent line break.
        - `format()` – convenience method for writing formatted text directly to the stream.
        - `printf()` - identical to format (in fact, `printf` passes through to `format`)

Interacting with users

- `java.io.Console` (more evolved form of `System.in`/`System.out` stream classes).
- `Console` is a singleton class created automatically by the JVM and can be accessed by calling `System.console()`. Returns `null` where text interactions are not supported.
- Methods:

- `reader()`, returns the `Reader`
- `writer()` returns the `PrintWriter`.
- `format(String, Object...)`, `printf()` are identical and output data to the users
- `flush()`, forces buffered output to be written immediately. Call before any `readLine()`, `readPassword()`
- `readLine()`, overloaded `readLine(String format, Object... args)` -> write text to user before reading from input.
- `readPassword()`: similar to `readLine()` but input is not echoed back to user. Overloaded `readPassword(String format, Object... args)` to prompt user before input. Returns character array to prevent it from going to shared memory pool. Recommended to clear array immediately after reading.

From the questions:

`c.writer()` (where `c` is an instance of `Console`) returns a `Writer` object. You can then call `w.append(line)` -> see question 5 on page 446

`new File("c://book//java")` -> this is not a right format of the file location. Windows requires '\\', for Linux based file systems use '/'. Also recall that you can use the system property System.getProperty(System.getProperty("file.separator")) or java.io.File.separator to retrieve the OS dependent separator. Question 12 has two invalid ways to instantiate a file. \ and // are both not valid separators.

Any `class`, `abstract`, concrete or `final` can be marked as `Serializable`

Question 21: note that although static fields are not taken into account when serializing a class, they will not be null if they are assigned a value at the class level since they are available again when the object is deserialized. So in this case, "birthPlace" might not be null upon deserialization.

Question 23: "reset puts the stream back into the position before mark was called..." -> mark was called with 3 as the argument but the stream was at position 0 at that point. Reset would therefore put it back to the 0th position after the loop.

HOOFDSTUK 9 NOTITIES – NIO.2

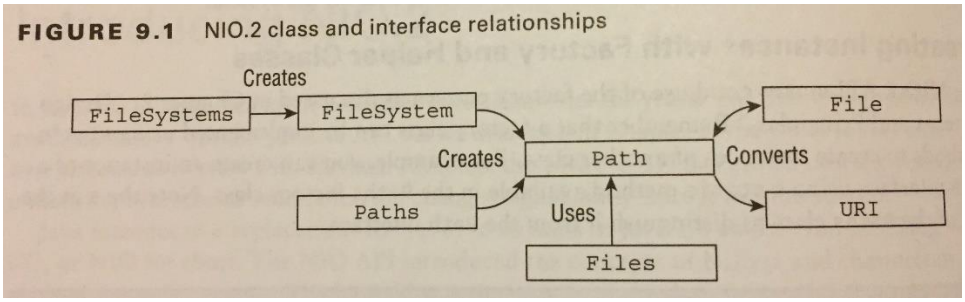
- Tabellen uit het hoofd leren: 9.1 blz. 461. Tabel 9.5 blz. 494 – niet uit het hoofd leren maar weten wat de genoemde methoden doen.
- `java.nio.file.Path` interface - basic entry point and replacement of `java.io.File` + support for navigating, detecting and creating symbolic links.

- `Paths` factory to create instance of platform specific instance (see below).

- Watch out for `Paths` and `Path` mixups in the exam: `Path` is the interface and `Paths` is the factory class to retrieve an instance of `Path`.
  `Paths path = Paths.get(...)` obviously is **false** and `Path path = Path.get(...)` **too**. To get a Path instance you can invoke: `Path path = Paths.get(…)`

- NIO.2 offers helper classes like `java.nio.file.Files` - mainly operates on `Path` objects.

- Figure 9.1 (blz. 456): not clear to what extend it is relevant for the exam - overview of the relationships of classes and interfaces in NIO.2.



- Use `Paths.get(String)` method to obtain a `Path` instance.
- Rules to learn for the exam:
    - *If the path starts with a forward slash it is an absolute path*
    - *If the path starts with a drive letter it is an absolute path*
    - *Otherwise, it is a relative path (e.g. ..\eagle)*

- `Paths.get(String, String...)` -> generates path based on parameters where NIO puts the platform specific separator between the strings.
- `Paths.get(URI)` (`Paths.get(new URI(file:///home/zoodirectory)`) **always requires an absolute path** otherwise throws **exception** at runtime (like for `Paths.get(new URI(file://pandas/cuddly.png")`) <-- relative path).
- `Path` interface contains a `toUri()` method to convert a path back to a URI.

- `Paths.get()` is shorthand for `java.nio.file.FileSystem.getPath()`.

- `FileSystem` - get instance using `FileSystems`. `FileSystems.getDefault().getPath("pandas/cuddly.png")` etc.
- `FileSystem` remote filesystem:

```
FileSystem fileSystem = FileSystems.getFileSystem(new
URI("http://www.selikoff.net"));
Path p = fileSystem.getPath("duck.txt");
```

- `java.io.File` now has a `.toPath()` method to convert a legacy `java.io.File` object to a `java.nio.file.Path`. `Path` also has a `toFile()` method that returns a legacy `File` instance.

Enum constants are left out in most method descriptions in the book but you must recognize them if they are provided to methods in the exam (table 9.1 page 461)

- Viewing the path:
    - `java.nio.Path.toString()`,
    - `.getName(int)` -> get name of path element at index. Is zero indexed and doesn't include the root segment (`/land/hoppy/harry.happy` -> 0 = `land`)
    - `.getNameCount()` returns 0 if `Path` represents root.

- Accessing Path components:

    - `getFileName()` – returns `Path` representing the filename, the fartest element from the root or start of the relative path.
    - `getParent()` – returns a `Path` instance representing the path or null if there is no such parent.
    - `getRoot()` – will stop at top-level element in case the path is relative. `getRoot()` returns the root or null if the `Path` is relative.

- Checking path type:
    - `isAbsolute()`
    - `toAbsolutePath()` - converts to an absolute path or returns a copy of the `Path` in case it was already absolute. Called on a relative path object, `toAbsolutePath()` preprends directories from the perspective of the working directory.

- `subPath(int, int)` – create new `Path` instance based on the subPath indexes. **Inclusive start** index, **exclusive end** index. *Warning: root is not included*. So example on page 466 if you have this:

  ```
  Path path = Paths.get("/mammal/carnivore/raccoon.image");
  ```

  And then invoke:

  ```
  System.out.println("Subpath from 0 to 3 is: " + path.subPath(0, 3));
  ```

  It will return: "`Subpath from 0 to 3 is: mammal/carnivore/raccoon.image`" (so without the '`/`' in the first position). Throws `IllegalArgumentException` when values are invalid (e.g. `path.subPath(1, 1);` or `path.subPath(1, 4);`)

- `relativize(Path)`
    - returns a `Path` instance relative to the given path.
    - When both paths are relative, relativize computes the paths as if both `Paths` are in the same working directory.
    - *Both the path and the given path must be relative or both absolute*, will throw `IllegalArgumentException` if you mix them.
    - When on Windows file systems, they must also both be in the same root directory (have the same drive letter).
    - Does not operate on the file system, just takes the elements of the paths to compute the relative path.
    - Does not clean up the path (so '.' characters are left in tact)

- `resolve(Path)`
    - Joins two paths with each other.
    - Returns a new path that has the current path joined with the given path.
    - *Relative and absolute paths can be mixed.* `path1.resolve(path2);` -> if `path1` is absolute and `path2` is relative, a path is returned with both paths combined.
    - If *both paths are absolute*, `path1` will be *ignored* and a copy of `path2` is returned.
    - If `path1` is relative and `path2` is absolute, then `path1.resolve(path2)` will ignore `path1` and return a copy of `path2`.

- If you would turn them around, then `path2` is appended to `path1`:

```
Path one = Paths.get("temp\\absolute.txt");
Path two = Paths.get("D:\\development\\why.blaat");
System.out.println("Resolved they become: " +
one.resolve(two));
System.out.println("Or the other way around: " +
two.resolve(one));
```

Prints:

```
Resolved they become: D:\development\why.blaat
Or the other way around:
D:\development\why.blaat\temp\absolute.txt
```

- Does not clean up the path (so '.' characters are left in tact)

- `normalize(Path)`
  - removes redundancies in a path. Does not operate on the actual file system.
  - Watch out for normalize rules:
    - It removes the redundant '..' and '.' characters but *also resolves* what the directory would be. So this: `E:\data\..\user\home` becomes `E:\user\home` because the '`..`' denotes one directory up.

- `toRealPath()`
  - may throw an IOException
  - checks the file system for existence
  - **implicitly calls `normalize()`**
  - the only method that accepts `NOFOLLOW_LINKS`
  - getting real path of current working directory: `Paths.get(".").toRealPath();`

Paths method for the exam (as described above):

Getting a `Path` instance:

- `Paths.get(String)`
- `Paths.get(String, String...)`
- `Paths.get(URI)`
- `FileSystems.getDefault().getPath(String)`
- `FileSystems.getFileSystem(URI).getPath(String)`
- `java.io.File.toPath` (legacy class extended to support conversion to `java.nio.file.Path`)

Working with `Path`s:

- `Path.toFile()` (converts to legacy `java.io.File` instance)
- `Path.toString()`
- `Path.getName(int)`
- `Path.getNameCount()`
- `Path.getFileName()`

- `Path.getParent()`
- `Path.getRoot()`
- `Path.isAbsolute()`
- `Path.toAbsolutePath()`
- `Path.subPath(int, int)`
- `Path.relativize(Path)`
- `Path.resolve(Path)`
- `Path.toRealPath()`

Interacting with files -> `java.nio.file.Files`

- `Files.exists(Path)`
- `Files.isSameFile(Path, Path)`:
    - returns true if files are same.
    - Throws `IOException` if either file does not exist.
    - Does not check content of the files.
    - *Follows symbolic links*.
    - First checks equals methods, if true then returns true; if false then checks location on disk.
- `Files.createDirectory(Path)` and `Files.createDirectories(Path)`
    - same as `java.io.File.mkdir()` and `java.io.File.mkdirs()`.
    - Throws `IOException` if parent does not exist in the case of the `createDirectory` method.
    - Also accepts optional `FileAttribute<?>` instance to set attribute on the directories.
- `Files.copy(Path, Path, CopyOption… options) throws IOException`
    - Copies files or directory from one place to another.
    - Directory copies are shallow: subdirectories and their contents are not copied (if the file is a directory, creates an empty directory in the target).
    - By default: does not overwrite existing files (fails if target already exists or is a symbolic link), follows symlinks, and does not copy file attributes.
    - Options can be configured by providing the additional options "`NOFOLLOW_LINKS`", "`REPLACE_EXISTING`" and "`COPY_ATTRIBUTES`".
    - Overloaed version of copy takes stream from the `java.io` package to read and write from and to a stream.
    - Allows to copy file contents to an external file system, a website etc. Takes two parameters, one of them being a `Path` instance. Depending on reading or writing the other parameter is an `InputStream` or `OutputStream` respectively.
- `Files.move()`
    - throws `IOException` and optionally a `AtomicMoveNotSupportedException`.
    - By default follows symbolic links, doesn't do an atomic move, throw an exception if the file already exists.
    - Behavior can be changed using the "`NOFOLLOW_LINKS`", "`ATOMIC_MOVE`" and "`REPLACE_EXISTING`".
- `Files.delete(Path)` and `Files.deleteIfExists(Path)`
- `Files.newBufferedReader(Path, Charset)`,`Files.newBufferedWriter(Path, Charset)` - reading and writing using a stream. By default the writer overwrites existing files.
- `Files.readAllLines(Path)` - reads all lines in the provided file and returns them in an ordered list of `String` values. An overloaded version lets you specify the charset.

Methods from `Files` class for exam:

- `Files.isDirectory(Path)`
- `Files.isRegularFile(Path)` – if sym. link points to regular file: also returns true
- `Files.isSymbolicLink(Path)`
- `Files.isHidden(Path) throws IOException`
- `Files.isReadable(Path)`
- `Files.isExecutable(Path)`
- `Files.size(Path) throws IOException` - in bytes. Result is undefined and system dependent if called on a directory.
- `FileTime Files.getLastModifiedTime(Path) throws IOException`
- `Files.setLastModifiedTime(Path, FileTime) throws IOException`
- `UserPrincipal Files.getOwner(Path) throws IOException`
- `Files.setOwner(Path, UserPrincipal) throws IOException`
- `Files.readAttributes(Path) throws IOException`
- `Files.getFileAttributeView(Path, Class<V>) throws IOException` – provides a direct resource for modifying file information. Can be used to update the file system-dependent attributes but also to read the attributes. You can use the `readAttributes()` method on the view to do so. This returns a `BasicFileAttributes` object. Class can be `BasicFileAttributeView.class`, for example.
- `Files.walk(Path)throws IOExcecption`
- `Files.find(Path, int, BiPredicate)throws IOException`
- `Files.list(Path) throws IOException` – operates on items that are direct children of the parent (alternative to using Files.walk with depth of 1)
- `Files.readAllLines(Path) throws IOException` – reads all lines of a file at once.
- `Files.lines(Path) throws IOException` – with Stream (lazy)

Additional methods

- Get a `UserPrincipal` from the filesystem is done with the `FileSystem` class. An instance can be instantiated in two ways:
    1. By using the path instance – `path.getFileSystem()`
    2. By using the factory class `FileSystems` – `FileSystems.getDefault()`

You can then get the `UserPrincipalLookupService` instance and lookup a `UserPrincipal` by name:

```
path.getFileSystem().getUserPrincipalLookupService().lookupPrincipalByName(
"jane");
```

Attributes and view class you need to know for the exam:

`BasicFileAttributes`, view class: `BasicFileAttributeView`. They represent a basic set of file attributes that is present on all file systems. Common methods you should know for the exam:

- `creationTime()`
- `lastModifiedTime()`
- `setTimes(FileTime lastModifiedTime, FileTime lastAccessTime, FileTime createTime)`

The new Stream methods

- Two common strategies to walk a directory tree:

    - *Depth-first search* – directory is traversed from the root to an arbitrary leaf and then navigated back to the root. Search depth is the distance from the root to current node. Mostly a max. depth is maintained.
    - *Breadth-first search* – starts at root and processes all elements of each particular depth, or distance from the root, before proceeding to the next depth level.

- *Depth-first search* is used by the Streams API with a default max. depth of `Integer.MAX_VALUE`

NIO.2 stream based methods:

- `Files.walk(Path) throws IOException` – returns a `Stream<Path>` object that traverses the directory in a *depth-first*, *lazy* manner (the contents have not yet been loaded).
    - Overloaded method that takes an additional int for the depth is also present where 0 means the current path record.
    - Overloaded method that take the `FOLLOW_LINKS` option exists. If you provide that, the `walk()` method will traverse the paths it has visited and throws a `FileSystemLoopException` if a cycle is detected. It is recommended to specify depth.
- `Files.find(Path, int, BiPredicate)` – More advanced method. `BiPredicate` in the form of `(T, U) -> Boolean` object types are `Path` and `BasicFileAttributes`. `int` specifies the max depth. Allows you to search directories in a more advanced way like:

```
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 exception
}
```

- `Files.list(Path)` – returns a stream of `Path` objects that are direct children of the parent (similar to the `java.io.File.listFiles()` method but then on a `Stream`)
- `Files.lines(Path)` – returns a `Stream<String>` object. Alternative to `Files.readAllLines()` which might result in an `OutOfMemoryError`.

Addditional notes about Enthuware tests:

- Page 489 explains that you would normally require an entire class definition to do what they did in one line and mentions the `FileVisitor`. Although the book mentions that you were required to know that interface for OCP7 and doesn't further explain that interface or any implementing classes, the test exams of Enthuware requires you to be familiar with it. Note that Appendix C does have a complete chapter about the interface. To complement these notes therefor: if you want to walk through a directory structure recursively, which interface do you need to implement for this purpose and which class do you need to extend to avoid implementing all the methods in that interface:
    - `FileVisitor` and `SimpleFileVisitor`

- There is not much information about the `PathMatcher` interface although knowledge of this interface is required:

  `java.nio.file.PathMatcher` is an interface that declares only one method:

  `boolean matches(Path path)`

  Tells if given path matches this matcher's pattern.

  Parameters: path - the path to match. Returns: true if, and only if, the path matches this matcher's pattern
- You need to know the `WatchService` which is described in Appendix C in the book. In short:

  `WatchService` has two types of methods for retrieving the `WatchKeys` - poll and take. The two `poll` methods return null if no key is present and do not wait for ever. The `take` method doesn't return null but keeps waiting until a key is available.

Vragen:

3. `setTimes()` is enkel beschikbaar op de `BasicFileAttributeView`. Met `BasicFileAttributes` kun je alleen lezen.

9. "performance is not often the reason to prefer one to the other": lijkt me onjuist maar wordt hier toch als toelichting gebruikt waardoor A onjuist zou zijn.

12. hier wordt over de regel heen gestapt dat je bij het meegeven van een URI een absolute path moet gebruiken. Dit mag dus wel: `Paths.get(new URI("cheetah.txt"))`

17. '..' resolved naar de root path. '..' en `getParent()` zijn conceptueel gelijk. Let erop dat je dus gewoon '..' kunt gebruiken in de `getPath()` methode.

HOOFDSTUK 5 NOTITIES – DATES, STRINGS, AND LOCALIZATION

Tabellen uit het hoofd leren: table 5.2, blz. 243 (maar wijst eigenlijk zichzelf); table 5.9 page 267

`java.time.*`

`LocalDate` – date without time and timezone information
`LocalTime` – time without date and timezone information
`LocalDateTime` – date and time without timezone information
`ZonedDateTime` – date and time with timezone information.

All have a `.now()` static method representation (`ZonedDateTime.now()` – returns a `ZonedDateTime` of now with date, time and timezone information)

All the creation methods below use static methods (factories), the **constructor** can **not** be used.

Creating a `LocalDate` instance:

- `public static LocalDate of(int year, int month, int dayOfMonth);`
- `public static LocalDate of(int year, Month month, int dayOfMonth);`

Months start with 1! Watch out for exam questions where `Month` is compared with an `int` (`Month.JANUARY = 1` does not compile).

Creating a `LocalTime` instance (multiple overloaded static methods varying in detail):

- `public static LocalTime of(int hour, int minute);`
- `public static LocalTime of (int hour, int minute, int second);`
- `public static LocalTime of(int hour, int minute, int second, int nanosecond);`

Creating a `LocalDateTime` instance: a lot of different variations possible including a method that takes a `LocalDate` and `LocalTime` as a parameter to combine them to a `LocalDateTime`. Entire list is on page 238.

Creating a `ZonedDateTime` requires you to create a `ZoneId` first using the static method: `of`, like so:

```
ZoneId zone = ZoneId.of("US/Eastern");
```

Then you can use one of three `ZonedDateTime` static methods to create a `ZonedDateTime` instance:

- `public static ZonedDateTime.of(int year, int month, int dayOfMonth, int hour, int minute, int second, int millisecond, ZoneId zone);` -> no version that accepts a `Month` enum.
- `public static ZonedDateTime.of(LocalDate date, LocalTime time, ZoneId zone);`
- `public static ZonedDateTime.of(LocalDateTime dateTime, ZoneId zone);`

Note that exceptions may be thrown in case invalid values are used in the create methods e.g. when you pass in 32 as a month value.

Date and time instances are **immutable**. Manipulating them therefor returns a new instance:

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
date = date.plusDays(2);
date = date.plusWeeks(1);
date = date.plusMonths(1)
date = date.plusYears(5);
```

Note that the exam might try to trick you not (re-) assigning a date instance after calling a manipulation method and printing the date instance:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date.plusDays(20);
System.out.println(date);
```
← **prints January 20, 2020…**

Similar methods exist for subtracting date and time elements and the methods can also be used with `LocalDateTime`. Furthermore, date and time methods can be chained e.g.:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date,
time).minusDays(1).minusHours(10).minusSeconds(30);
```

When printing, Java hides seconds and milliseconds if we aren't using them but when they become relevant it might be printed.

Adding and subtracting can also be done using `Period` and `Duration` instances (`date.minus(Period.ofWeeks(1)); date.plus(Duration.ofDays(23));`)

You have to use Durations and Periods that are compatible with the instances:

```
LocalDate date = LocalDate.of(2015, 1, 20);
date.plus(Duration.ofHours(1)); <- UnsupportedTemporalException
```

Warning: even though you have a `Duration` and a `Period` of the same lentgth:

```
Duration d = Duration.ofDays(1);
Period p = Period.ofDays(1);
```

Only `Period` is compatible with a `LocalDate`. So, this is **not possible** (assuming te above declarations):

```
LocalDate.of(2015, Month.JANUARY, 10).plus(d);
```

… while this is:

```
LocalDate.of(2015, Month.JANUARY, 10).plus(p);
```

Period: a day or more of time. Ways to create a `Period` instance (5):

- `Period.ofYears(1);`
- `Period.ofMonths(2);`
- `Period.ofWeeks(1);`
- `Period.ofDays(1);`
- `Period.of(1, 0, 7);` <- every year and seven days

Warning: you cannot chain methods when you create a `Period` (`Period.ofYears(1).ofWeeks(2);`)

Displaying a period always prepended with a 'P' and then the years, months and days (**P1Y1M1D** for one year, one month and one day). In case no value is present, it is omitted so `Period.ofMonths(3)` would print:

P3M

`Period.ofWeeks(3);` prints P21D

Periods can only be used by compatible types so **trying** to **add** a **period** of a **month** to a **LocalTime fails** with an exception at runtime.

Duration: for smaller units of time (days, hours, minutes, seconds, nanoseconds). Used with objects that have time and printed with 'PT' (Period of Time).

- `Duration.ofDays(1); // PT24H`
- `Duration.ofHours(1); // PT1H`
- `Duration.ofMinutes(1); //PT1M`
- `Duration.ofSeconds(1); // PT1S`
- `Duration.ofMillis(1); // PT0.001S`
- `Duration.ofNano(1); //PT0.000000001S`
- `Duration.of(1, ChronoUnit.DAYS);`
- `Duration.of(1, ChronoUnit.HOURS);`

- `Duration.of(1, ChronoUnit.MINUTES);`
- `Duration.of(1, ChronoUnit.SECONDS);`
- `Duration.of(1, ChronoUnit.MILLIS);`
- `Duration.of(1, ChronoUnit.NANOS);`

(`Duration` doesn't have a factory method that takes multiple units)

`ChronoUnit` has a way to calculate how far apart two temporal parts are, like:

`ChronoUnit.HOURS.between(LocalTime.of(5, 15), LocalTime.of(6, 30);` -> 1 (rounds)

`Instant` class represents a specific moment in time in the GMT time zone.

- Can be used to calculate a duration between two instants like:

```
Instant now = Instant.now();
// do something
Instant then = Instant.now();
Duration duration = Duration.between(now, then);
System.out.println(duration.toMillis());
```

- `ZonedDateTime` instances can be turned into an instance (and **only** `ZonedDateTime` instances since they contain timezone information and an Instance is a point in time at GMT).

  ```
  ZonedDateTime.toInstant();
  ```

  Note that the `ZonedDateTime` automatically translates the `ZonedDateTime` to the date-time calculated to GMT (Z – Zulu).

- Allows to add any unit of day or smaller only!

```
Instant tomorrow = instant.plus(1, ChronoUnit.DAYS);
Instant nextHour = instant.plus(1, ChronoUnit.HOURS);
// THIS FAILS:
Instant nextWeek = instant.plus(1, ChonoUnit.WEEKS);
```

Daylight savings is also on the exam and you need to know what happens at times when the hour is jumped one hour ahead (Spring) and one hour back (Fall) and remember that GMT doesn't participate. **See examples on page 252**.

Note that in case you create a time that doesn't exist because of DST, Java rolls it forward – **see page 253**.

String review just for refreshing:

- String class is final and String objects are immutable
- String pool
- Strings can be compared with the == operator but may not always be equal in case they are created with a constructor or method call. Reason is that if you assign a String like this: `String s1 = "test";` creates a string in the pool of strings automatically and another assignment: `String s2 = "test";` would point to the same instance. `String s3 = new String("test");`, however, creates a new object explicitly and puts that in the pool.
- Better to use the `equals()` method.

- Cocatenating with the + operator creates a new string instance
- “1” + 2 + 3 -> “123” but 1 + 2 + “3” -> “33”
- If you update a `String` value in a loop: use `StringBuilder` (`StringBuilder` is mutable)
- If multiple Threads are updating the same object: use `StringBuffer` (is Thread safe, just like `String` and unlike `StringBuilder` which is not)

Internationalization and localization

`java.util.Locale` – get the default locale: `Locale.getDefault();`

Formats:

- Either only language code (must be lowercase): `fr`
- Or language code, underscore, countrycode (most be uppercase): `en_US`

Three main ways to create `Locale`:

- Using constants:
    - `Locale l = Locale.GERMAN; //de`
    - `Locale l = Locale.GERMANY; //de_DE`
- Using the constructor and passing in the language or language and country code:
    - `Locale l = new Locale(“fr”));`
    - `Locale l = new Locale(“fr”, “FR”));`
- Using the Builder pattern:
    - `Locale l = new Locale.Builder().setLanguage(“en”).setRegion(“US”).build();`
    - `Locale l = new Locale.Builder().setRegion(“US”).setLanguage(“en”).build();`

Java will let you create a `Locale` with an invalid language and region setting but it will not match the `Locale` you want to use and doesn’t behave as expected.

You can set the default locale for just the application with the `Locale.setLocale(Locale)` method (will return that locale if you call `Locale.getDefault()`).

Resource bundles:

- Name followed by underscore followed by language and the extension of the file:

```
Zoo_en.properties
Zoo_fr.properties
```

- Files contain key/value pairs separated by:
    - ‘=’
    - ‘ ‘ (an empty space)
    - ‘:’
    - !, # -> comment
    - White spaces before and after the lines are ignored
    - Escape characters can be used
    - Break a line using a backslash and continue on the next line (‘\’)

- Get the resource bundle based on a locale:

```
ResourceBundle.getBundle(“Zoo”, new Locale(“en”, “US”));
```

- … or just by name:

```
ResourceBundle.getBundle("Zoo"); // uses default locale
```

- Get a string based on a key:

```
resourceBundle.getString(String key);
```

- Get all the keys in a resource bundle:

```
Set<String> keys = resourceBundle.keySet();
```

- Convert the resource bundle to a `Properties` instance with:

```
Properties properties = new Properties();
resourceBundle.keySet().stream().forEach(k -> props.put(k,
resourceBundle.getString(k));
```

Properties:

- Call `getProperty("key");` to get a key, returns null if it doesn't exist
- Call `getProperty("key", "defaultValue");` to get a key and provide a default value in case "key" doesn't exist.
- `get(Object key)` – inherited. Also allows to get the property value but doesn't allow to provide a default in case key is not found.

Java class resource bundles:

- Allows any type for the value part (key remains a `String`)
- Class name is name, underscore language and extends `ListResourceBundle`
- Implement `getContents()` that returns a two dimensional array
- Main advantages:
    - You can use a value that is not a `String`
    - You can create the values at runtime.

When you request a resource bundle, Java tries to find the most specific one. A class resource bundle get preference over file resource bundle (in case there is a tie).

Rules to match resource bundles:

- Look for the property file after the matching Java class (Java class has precedence)
- Drop one thing at a time if there are no matches. First drop the country and then the language.
- Look at the default locale and the default resource bundle last.

Mind the hierarchy rules as well. Rules are to match a resource bundle that acts as a base. It can get them from any parent of that resource bundle. Going from most specific to least specific. See **table 5.8 page 265**.

Format numbers – `java.text` package

Several factory methods to get a `NumberFormat`:

- General purpose formatter:
    - `NumberFormat.getInstance();`
    - `NumberFormat.getInstance(Locale locale);`
- Same as getInstance:
    - `NumberFormat.getNumberInstance();`
    - `NumberFormat.getNumberInstance(Locale locale);`
- Formatting monetary amounts:
    - `NumberFormat.getCurrencyInstance();`
    - `NumberFormat.getCurrencyInstance(Locale locale);`
- Formatting percentages:
    - `NumberFormat.getPercentageInstance();`
    - `NumberFormat.getPercentageInstance(Locale locale);`

`.parse()`, turn a `String` into a number (throws `ParseException` if parsing fails). Also, it parses until it finds a character that it can not parse and returns only the part it can parse. If the first character is a character it can not parse, it throws `ParseException`. The return value of `parse()` is a `Number` object which can subsequently be cast to its appropriate data type. Behavior depends on the type of formatter. If you instantiate a `NumberFormat` using the `getCurrencyInstance()` method, you can't just call parse with a number (like `.parse("1")`) if you are working in the Dutch locale.

Comparing `DecimalFormat` from Appendix C to the `NumberFormat` class:

- `DecimalFormat` is a subclass of `NumberFormat`
- It is instantiated with a constructor, taking a string of the format of the decimal number
- `NumberFormat` also has a constructor but that's typically not used to instantiate an instance of `NumberFormat`.

`.format()`, turn a number into a `String`.

The format classes are **not tread safe**, do not store them in instance variables or `static` variables.

`java.time.format.DateTimeFormatter` to format any type of date and/or time object:

- Can be used in `LocalDate`, `LocalTime`, `LocalDateTime` object's `.format()` methods
- Instantiated by invoking methods that take a value of `FormatStyle` (`FormatStyle.SHORT`, `FormatStyle.MEDIUM`), rest is not relevant for exam):
    - `ofLocalizedDate(FormatStyle style);`
    - `ofLocalizedTime(FormatStyle style);`
    - `ofLocalizedDateTime(FomatStyle style);`
- Or instantiated with a constant:
    - `DateTimeFormatter.ISO_LOCAL_DATE`
    - `DateTimeFormatter.ISO_LOCAL_TIME`
    - `DateTimeFormatter.ISO_LOCAL_DATE_TIME`
- Or instantiated with a custom format:
    - `DateTimeFormatter.ofPattern("MMMM DD, yyyy hh:mm);`
- Formatting can be done by:
    - Calling the formatter's `.format()` method passing in an instance of `LocalDate`, `LocalTime` or `LocalDateTime` where the allowed parameter is dependent on the chosen `DateTimeFormatter` (e.g. passing a `LocalTime` instance to `.format()` on a `DateTimeFormatter` that is created with the `ofLocalizedDate` method will fail)

    - Calling `.format()` on the Date/Time/DateTime instance itself (where the formatter is passed as a parameter).
- Trying to format dates in a format of which the corresponding formatter doesn't have a pattern for will result in an exception:

```
DateTimeFormatter df = DateTimeFormatter.ofPattern("hh:mm");
f.format(time);
f.format(dateTime);
f.format(date); // WILL THROW EXCEPTION
```

The parse methods can also be invoked on the instances of date/time/datetime and with a formatter. Two examples:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
```

Additional notes following the test exams of Enthuware:

- You are expected to know that `LocalDate`, `LocalTime`, `LocalDateTime` and `ZonedDateTime` implement the `TemporalAccessor` interface.

  The entire description from that test:

  Here are some points that you should keep in mind about the new Date/Time classes introduced in Java 8 -
  1. They are in package java.time and they have no relation at all to the old java.util.Date and java.sql.Date.

  2. java.time.TemporalAccessor is the base interface that is implemented by LocalDate, LocalTime, and LocalDateTime concrete classes. This interface defines read-only access to temporal objects, such as a date, time, offset or some combination of these, which are represented by the interface TemporalField.

  3. LocalDate, LocalTime, and LocalDateTime classes do not have any parent/child relationship among themselves. As their names imply, LocalDate contains just the date information and no time information, LocalTime contains only time and no date, while LocalDateTime contains date as well as time. None of them contains zone information. For that, you can use ZonedDateTime.

  These classes are immutable and have no public constructors. You create objects of these classes using their static factory methods such as of(...) and from(TemporalAccessor). For example, LocalDate ld = LocalDate.of(2015, Month.JANUARY, 1); or LocalDate ld = LocalDate.from(anotherDate); or LocalDateTime ldt = LocalDateTime.of(2015, Month.JANUARY, 1, 21, 10); //9.10 PM

  Since you can't modify them once created, if you want to create new object with some changes to the original, you can use the instance method named with(...). For example, LocalDate sunday = ld.with(java.time.temporal.TemporalAdjusters.next(DayOfWeek.SUNDAY));

4. Formatting of date objects into String and parsing of Strings into date objects is done by java.time.format.DateTimeFormatter class.

This class provides public static references to readymade DateTimeFormatter objects through the fields named ISO_DATE, ISO_LOCAL_DATE, ISO_LOCAL_DATE_TIME, etc. For example -

LocalDate d1 = LocalDate.parse("2015-01-01", DateTimeFormatter.ISO_LOCAL_DATE);

The parameter type and return type of the methods of DateTimeFormatter class is the base interface TemporalAccessor instead of concrete classes such as LocalDate or LocalDateTime. So you shouldn't directly cast the returned values to concrete classes like this

LocalDate d2 = (LocalDate) DateTimeFormatter.ISO_LOCAL_DATE.parse("2015-01-01"); //will compile but may or may not throw a ClassCastException at runtime.

You should do like this

LocalDate d2 = LocalDate.from(DateTimeFormatter.ISO_LOCAL_DATE.parse("2015-01-01"));

5. Besides dates, java.time package also provides Period and Duration classes. Period is used for quantity or amount of time in terms of years, months and days, while Duration is used for quantity or amount of time in terms of hour, minute, and seconds.

Vragen:

7: Hier worden uren opgeteld bij een date wat niet kan maar je krijgt dan al een compiler error end us is vraag E fout.

10: als je een `DateTimeFormatter` hebt aangeroepen met `ofLozalized`**`Time`** en een style `FormatStyle.SHORT` en die formatteert op een `LocalDateTime` object wordt enkel de tijd informatie getoond (let op “ofLocalizedTime”).

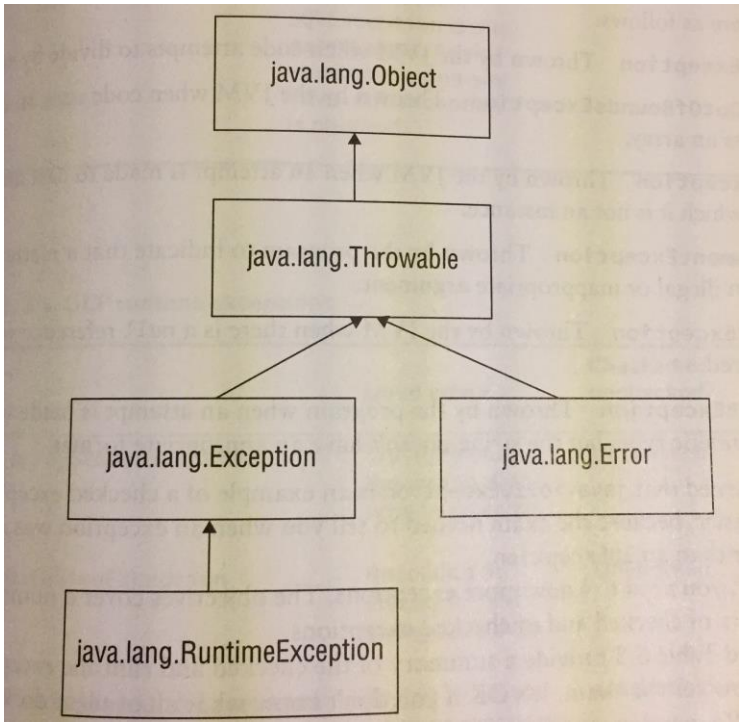
11: `Period` staat geen chaining toe maar het **compileert en draait wel gewoon**: alleen de **laatste methode** die je aanroept **telt**.

17: get is de methode die werd overge-erft en verwacht een key waarde. getProperty ondersteunt daarnaast de key waarde en een default waarde. In de vraag warden de twee methoden door elkaar gehaald. In dit geval zou het een compilatieprobleem opleveren.

HOOFDSTUK 6 NOTITIES – EXCEPTIONS AND ASSERTIONS

Errors – fatal, should not be caught by the program. Is legal to catch it, but not good practice.
Unchecked exception – or runtime exception can optionally be caught but don’t have to be caught or declared.
Checked exception – have to follow the *handle or declare* rule. Is any class that extends `Exception` but is not a runtime exception.

Memmorize:



Runtime exceptions you need to memorize:

- `ArithmeticException` – thrown by the JVM when code attempts to divide by zero.
- `ArrayOutOfBoundsException` – thrown by the JVM when the code uses an illegal index to access an array
- `ClassCastException` – thrown by the JVM when an attempt is made to case an object o a subclass of which it is not an instance .
- `IllegalArgumentException` – thrown by the program to indicate that a method has been passed an illegal or inappropriate argument.
- `NullPointerException` – Thrown by the JVM when there is a `null` reference where an object is required
- `NumberFormatException` – thrown by the program when an attempt is made to convert a string to a numeric type, but the string doesn't have an appropriate format.
- `java.lang.ArrayStoreException` – trying to store the wrong data type in an array
- `java.time.DateTimeException` – receiving an invalid format string for a date
- `java.util.MissingResourceException` – trying to access a key or resource bundle that does not exist.
- `java.lang.IllegalStateException`, `java.lang.UnsupportedOperationException` – attempting to run an invalid operation in collections and concurrency.

Checked exceptions you need to memorize:

- `java.text.ParseException` – may be thrown when converting a string to a number
- `java.io.IOException`, `java.io.FileNotFoundException`, `java.io.NotSerializableException` – Dealting with IO and NIO.2 issues. `IOException` is the parent class, there are a number of subclasses. You can assume any `java.io` is checked.
- `java.sql.SQLException` – Dealing with database issues, `SQLExceptoin` is the parent class. You can assume any subclass of `java.sql` is checked.

`Try` statement needs to have either or both of the `catch` and `finally` clauses **except** when it is a try-with-resources statement.

- Java checks the exceptions in the order in which they appear
- It is illegal to declare a sub-class exception in a `catch` block that is lower down in the list than a superclass exception because it will be unreachable code.
- Java will not allow you to declare a `catch` block for a checked exception type that cannot potentially be thrown by the `try` clause body.
- In the declaration of a method you can also declare that it throws a runtime exception (but is not necessary).
- In a multi-catch block: the exceptions can appear in any order (but with one variable name)
- You can't use subclasses in the multi-catch block (because they would match both:

```
try{
  throw new IOException();
} catch (FileNotFoundException | IOException e) {
  …
}
```

← **does not compile**

- Re-assigning the exception in the catch block is legal in a single catch block (although bad practice) but illegal in a multi-catch block (in which case the exception is effectively final)

- In a try-with-resources/auto resource management block the implicit `finally` block runs before a defined one. You can also not use a resource created in the `try` clause in the catch or `finally` block (its scope is only within the `try` block).
- You can only use this functionality on classes that implement the `AutoClosable` interface
    - Has one method:

      ```
      public void close() throws Exception;
      ```

    - If you would throw an exception in the implementation of `close()`, you would have to handle that exception also in an try-with-resource block (by handling it in a `catch` block or by declaring it in the method)
- A method that overrides another method is allowed to declare more specific exceptions than the parent or even none at all.
- There is also a `Closable` interface that existed before the `AutoClosable` interface. It restricts the type of exception thrown to `IOException` and requires the implementations to be idempotent (which the `AutoClosable` doesn't but it is encouraged to make it idempotent)
- Java closes resources in the reverse order from which it created them:

  ```
  try (JammedTurkeyCage t1 = new JammedTurkeyCage(); JammedTurkeyCage t2
  = new JammedTurkeyCage();)
  ```

  ← t2 is closed before t1…

- If multiple exceptions are thrown of the same type, only the primary exception is caught. The so called *suppressed* exception(-s) can be retrieved by calling `getSupressed()` on the primary exception. Warning: if the primary exception is a `RuntimeException` and a sub-class is thrown thereafter, this will still be suppressed. So in this example:

  ```
  try (JammedTurkeyCage t = new JammedTurkeyCage()) {
  ```

```
    throw new RuntimeException("turkeys ran off");
} catch (IllegalStateException e) {
    System.out.println("caught: " + e.getMessage();
}
```

…assuming that the `close()` method of the auto-closable `JammedTurkeyCage` would throw an `IllegalStateException` the exception would not be caught in the above exception. Instead it is suppressed and the `RuntimeException` is thrown.

- Exceptions are only suppressed if they are thrown in the same `try` clause. If you, for example, throw a new exception in the `finally` block, it will not be suppressed (even though an exception has been thrown in the `try` clause).

Assertions:

- `assert boolean_expression;`
  `assert boolean_expression: error_message;`
- Throws a `java.lang.AssertError` if result is false (so an error which you are not supposed to catch or throw)
- When switched off, skipped.
- Enable by:
  - Using the `-enableassertions` flag on the command line or the shortcut flag `-ea`
  - Without parameters: enables for all Java classes except the System classes (classes that are part of the Java runtime)
  - With parameters:
    - Enable for classes in the `com.wiley.demos` package:

      ```
      java -ea:com.wiley.demos… my.programs.Main
      ```

    - Enable for just one class:

      ```
      java -ea:com.wiley.demos.TestColors my.programs.Main
      ```

    - Enable for package and disable for a specific class:

      ```
      java -ea:com.wiley.demos… -da:com.wiley.demos.TestColows
      my.Programs.Main
      ```

Vragen:

- Multi-catch block does not allow redundancy (so you can't put subclasses of exceptions in the same multi-catch block).
- Try-with-resources closes resources in reverse order
- If a single exception is caught, the same type or a sub-class can be assigned to the catch variable (question 12)
- You can declare a superclass of an Exception to be thrown in a method and throw a subclass
- In a multi-catch the variable in the catch block is effectively final and cannot be reassigned.
- A runtime exception can be thrown by any method. `Closable` has `IOException` in its signature but that doesn't mean at can't throw `RuntimeException` (and any sub class thereof).

- Keep in mind that you can’t use an exception in the catch block that is never thrown in the try block (question 19).