Contents

Part I

Intro

Chapter 1

TODOs

- JoinerSplitter library class
- implement primitive specialisation (both for inputs and returns) across all applications. refer to java8lambdas.pdf, p56-59.
- how to stream a string?
- laboratory06, FirsFit class - return does not terminate forEach loop! why? Is it possible to escape a stream prematurely?
- what is better - summaryStatistics or Collectors single value methods (like averagingInt(), averagingDouble())?
- Optional class...
- UML Distilled textbook by Martin Fowler
- `http://agilemodeling.com/artifacts/crcModel.htm`
- create 'Arrays' section. put attention to initialisers
- what application it can be for java reflection?
- end-of-file indications
- exceptions combined with streaming. if possible to implement - turn back to laboratory 06, streamed version.
- generics in 'getting started'
- check try-with-resources in Java 9
- Constructors Avoid JavaBeans style of construction Beware of mistaken field redeclares Construct Object using class name Constructors in general Copy constructors Don't pass 'this' out of a constructor Initializing fields to 0-false-null is redundant

- https://www.ibm.com/developerworks/learn/java/index.html
- `http://www.javapractices.com/home/HomeAction.do;jsessionid=30B15108A7091DF24625AA77941`
- Coder's assertion that the annotated method or constructor body doesn't perform **unsafe ??** operations on its varargs parameter.
- Questions, lecture 08:
- private interface methods
- find more about static initialisers
- Check Absolute Java chapter 8, binding.
- serialisation and copy constructor. Why it might be better than cloning?
- hashCode() override when eqauls() overidden. put in reference.comparison section
- scanner... ongoing
  - look at links inside 'Question About Scanner Class and nextInt(), nextLine(), and next() Methods (Beginning Java forum at coderanch' on HDD
  - look at link to stackoverflow in 'Problem with .nextLine() in Java. — Treehouse Community'
  - study Pattern class docs
- *Java 8 Lambdas* continue from 'Chapter 7'
- *Java Pocket guide* continue from 'Chapter 10'
- why inner classes (also lambda) require external values to be immutable (final in java)
- specify methods of all core functional interfaces
- check .get(), .sorted(), forEach(a $\rightarrow$ doSth()).
- 'interfaces, unlike classes, don't have instance fields. Therefore the only way to modify child classes is to call methods on them.' - investigate it!
- static methods i interfaces, like Stream.of()
- optional as a replacement for null, p. 55 - 56 in 'Java8 Lambdas' book; applications
- test method refereces, including constructors
- some operations are more expensive on ordered colections...???

- documentation for minBy and maxBy (*Collectors* library),another method in that class, implementations in real codes
- Entry, entrySet() in javadoc
- IntStream
- mapToObj()
- check signature of Arrays.parallelSort, Arrays.parallelSetAll, Arrays.parallelPrefix, and update 'DataParalelism'Arrays'
- *Test-Driven Development* by Kent Beck
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce
- factory methods in concurrent context
- why forced autoboxing/unboxing is not recommended
- check recommendation to use final variables on *javapractices*, lec.4
- , testing, JUnit, tutorial, apply to existing codes
- difference between import and static import... ongoing

Chapter 2

Use It Later

- Compact Strings The compact strings feature is an optimization that allows for a more space-efficient internal representation of strings. It is enabled by default in Java 9. This feature may be disabled by using -XX:-CompactStrings, if you are mainly using UTF-16 strings.

Chapter 3

Resources

Notes based on:

- UoG java labs & classes
- *Java 8 Lambdas*, R. Warburton
- *Java Pocket Guide*, R. Liguori, P. Liguori
- *Effective Java,* 2nd Edition, J. Bloch
- *Absolute java*, 5th Edition, W. Savitch
- *Java in a Nutshell*, 6th Edition , B. J. Evans, D. Flanagan item *Abstraction in java, Ultimate Guide*, H. Hamill

Part II

Basics

3.1 Overview

- object oriented
- platform-neutral (due to JVM), *'write-once-run-anywhere'* philosophy
- **.java** (source code) → **.class** (bytecode) → (executable)
- statically typed (determined in compile-time)

3.2 Identifiers

- starts with letters or underscores
- composed of letters, underscores or numbers
- informal naming conventions:
  - initial capital letter for **classes, interfaces, enums, annotation** names. interfaces should be **adjectives**.
  - all-CAPS for **constant identifiers**
  - all lower letters for **packages modules**. Modules names should be the inverse internet domain name.
  - initial lower case for other identifiers
  - camel-case for multi-word identifiers

Informal Single-Letter Local Variables Naming

- **b**, byte
- **i, j, k**, integer
- **l**, long
- **f**, float
- **d**, double
- **c**, character
- **s**, String
- **o**, Object
- **e**, exception

3.2.1 Scope Rules

Block - part of code enclosed by a pair of corresponding curly brackets. Rules:

- A local variable is in scope from the point if its declaration to the end of the enclosing block.
- Method parameters are valid until the end of the method.
- iteration variables declared in a for-loop initializer are in scope until the end of the loop body.

3.2.2 Name Conflicts.

No two variables of the same scope and the same type can share the same identifier. However, variables in nested scope shadow the variable with the same identifier and type from enclosing scope. For instance, if global variable and local variable have the same identifier and type, then the value of local variable is used in local scope.

```
class MyClass{
int a = 1;
void print(){
int a = 2;
        System.out.println(a);
}
}
```

displays '2'.

3.3 Primitive Types

Since java is statically typed language types must be:

- declared:

  ```
  int var;
  ```

- ...then initialised with a value before use:

  ```
  var = 5;
  ```

- It can be done in one go:

  ```
  int var = 5;
  ```

3.3.1 List of Types

- integer values
  - expressed as decimals, octals 0.., hex-decimals 0x.., binaries 0b1001101..
  - byte → short → int → long ..L
  - 8-bit → 64-bit
- floating point values
  - float, 32-bit, 0,0F, 0,0f)
  - double, 64-bit, 0,0D, 0,0d)
  - **fractional part not reqiured when type suffix is applied)**
- boolean, 1-bit flag
- char, 16-bit (2 bytes) Unicode values

3.3.2 Conversion

Specification:
`https://docs.oracle.com/javase/specs/jls/se9/html/jls-5.html`

Implicit conversion - generally widening conversions, where little or no information is lost. Examples - byte to short, int to double, int to long.
Narrowing conversion - requires use of explicit cast operator. This conversion is done by truncation of excessive bits. Therefore, it can cause loss of information:

- Conversion of integer values:

  ```
  int a = 1025;
  byte b = (byte)a;
  ```

  In the example above '1' is assigned to 'b'
  - '0000..10 0000 0001' is truncated to '0000 0001'.

- Float to integer conversion:

3.3.3 Primitive Type Specialisation (from java8)

The use of boxed numbers creates overheads:

- memory overhead, object require additional metadata to be stored in heap
- time overhead (for memory allocation)

Naming Conventions

- interface **return** type is primitive → To*Type*Function
  ToIntFunction, ToLongFunction..
- **parameter** type is primitivenction → *Type*Function
  IntFunction, LongFunction..
- **higher-order function** using primitive type → *Function*To*Type*
  MapToLong..
- **streams** → look 'Aggregate Operations/Specialised Primitive Types'

3.4 Switch Statement

```
switch(some){
  case x:
    do-sth;
    break; \\ control flows to the next case when there's no break
  case y:
    do-sth;
    ....
  \\ optionally
  default:
    do-sth;
    break;
}
```

The use of 'default' makes a code less error prone.

3.5 Labels

We use them in order to exit from a nested loop more than one level up. The label marks the loop to exit.

```
exitHere
while{
while{
break exitHere;
}
} // the break exits here.
```

Works in the same way for *continue*;

3.6 Text-based User Input

- program input argument

  ```
  public static void main (String[] args){
  String a = args[0];
      String b = args[1];
      ...
  }
  ```

- using scanner - `https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html`

3.7 Modifiers

3.7.1 Access Modifiers

Those modifiers specify where **fields** can be **accessed from**, where **methods** can be **called from**.

- **public** - any class
- **protected** - package & subclasses,
- **default** - *package protected*, package only
- **private** - the same class only

3.7.2 Other modifiers

- abstract, static, final, default
- native - to merge C and C++ with java (method declaration without body)
- strictfp - floating-point arithmetic as defiind by IEEE
- synchronized - one thread at the time allowed (thread-safe)
- transient - this data member is not serialised when a class is serialised
- volatile - multithreading: no caching, last actual value (from registers)

3.8 Reference Types

> 'Reference types hold references to objects and provide a means to access those objects stored somewhere in memory. The memory locations are irrelevant to programmers. All reference types are a subclass of type java.lang.Object.'

3.8.1 List of Reference Types

- classes
- interfaces
- enumerations
- arrays
- annotations

3.8.2 Reference vs Primitive Types

	Reference Type	Primitive Type
amount	**unlimited**, user defined	boolean and numeric only
memory content	**references** to data	**atual** data
multivariables	all variables that holds **the same reference** point **the same** object	all variables hold **a copy** of primitive data value. a modification to data stored in one variable **doesn't affect** the values stored in another variables
as a method parameter	**by reference** the object they point to **can be modified**	**by value** original values are **unaffected**

3.8.3 Default Values

they are the values assigned implicitly to not initialised variables.

Objects

Instance Variables

They are the variables defined at class level. **They have a value of null when not initialised**.

Local Variables

When not initialised, they have **no value at all, not even null value!**. Actually, they are assigned a NaN bit pattern at binary level not a garbage value, as it is commonly believed.

Arrays

Always given null as a default value! Both the array itself and their componnents as well, if they are objects.

3.8.4 Conversions

Widening

- convertion to **parent** class
- **always legall**
- **no explicit cast** is necessary

Narrowing

- From more general to more specific type, from superclass to subclass.
- **explcit cast is mandatory!**
- illegal conversion throws **ClassCastException**
- may result in loss of data/precision

3.8.5 Comparison Objects

Using Equality operators

Compares **references!** Return true only when references are the same - point to **the same object**.

Using equals()

Inherited from Object class, By default it uses == for comparison. Really useful when overridden.**hashCode()** should be overridden, too, for compatibility reasons (if it is expected to use a collection tat uses hash functions, HashMap, HashSet, etc).

3.8.6 Comparing String Objects

The rules are similar - **equality operators** compare references and **equals()** compares values character-wise. Therefore, the result depends on how the string object was created.

Strings Created from Literals

A reference points to a string object taken from pool. All String objects initialised with the same literal point to the same String object in the pool. Therefore **in opposite to common belief** comparison using equality operator returns **true** each time when underlying character string is the same, because the references are the same, as well as the object they point to.
Obviously **equaks()** returns **true**, too.

String Object Created Using 'new' Operator

Therefore the rules are exactly the same as for any other object - equality operators ompare references, check, whether or not they point to the same string object. equals() compers underlying character string.

3.8.7 Copying

Two options:

- to copy a reference
- to create a copy of an object (**cloning**).

Copying a Reference

New reference to the same object is created. Any change applied to an object using one reference is to see when the object is accessed by another reference.

Cloning

New object is created - a copy of existing one. There are two options - shallow and deep cloning.

Shallow Cloning

Is done by **clone()** method. **Cast is mandatory** - clone() returns type **object**. Can throw **ClonenotSupportedException**.
As a result of shallow cloning new object is created, with the same values of primitive types and copies of reference types. **Copies of the objects referred to by those references are not done**, they point to the same objects as original object.

Deep Cloning

Not supported by Java API, programmer must provide a copying method. Additionally to shallow cloning it makes recursively copies of objects that are referred by reference variables.

Serialisation and Copy Constructor

An alternative approach to cloning. TODO..

3.9 Constructors

> 'Classes implicitly have a no-argument constructor if no explicit constructor is present. (..) if a constructor with arguments is added, there will be no no-argument constructor unless it is manually added.' (*Java Guide*)

3.9.1 Constructor Chaining

Parent class constructor can be called within subclass constructor by using **super(..)** keyword. As parent class variables are initialised when parent constructor is called, this call is always done, whether explicitly or implicitly. Also another constructor can be called from within another constructor of the same class using **this().**

- **This call (super() or this()) must be the first statement in the constructor.**
- **this() and super(..) in the same constructor are not allowed.**
- If there is not an explicit call to the constructor of the superclass, an automatic call to the no-argument constructor of the superclass is made. Therefore:
  - there should be always **an explicit call to superclass constructor**
  - **no-arg constructor should be always present unless the class is final - not intended to be extended.**
  - otherwise **an error may be raised**.

3.9.2 Practices

- always put no-arg constructor definition when there is at least one constructor taking argument. The minimum - a constructor with no body. It just creates an object and initialises instant variables with default values:

  ```
  public myClass();
  ```

  The reason:

  ```
  myClass mc = new myClass();
  ```

  raises an error. **Implicit no-arg constructor is not present**, it is implicitly created only when **no other constructor is present**.
  Also a subclass constructor may make automatic call to superclass constructor.
- for the same reason there should always be an explicit call to a parent constructor from within child constructor when a class extends another class. If there is no explicit call to parent constructor **implicit call to no-arg constructor is done**, and this can be not present.
- **never call overridable method from within a constructor**. This method may call a variable, which is initialised within child constructor, but parent constructor is called implicitly first, and child method is called from within this parent constructor - ERROR! See OOP.Inheritance.Pitfalls.Constructor_calls_an_overrida

3.10 this

Common uses:

- to refer to current object
- to pass a reference of current object as a method argument
- to call a constructor from within another constructor in the same class.
- **this() and super(..) in the same constructor is not allowed.**

3.11 super

Refers to superclass. Common usage:

- to call superclass constructor from within subclass constructor.
- to call overridden superclass methods from within a subclass method. **This must be the first statement in the constructor.**
- **this() and super(..) in the same constructor is not allowed.**

3.12 Exceptions

3.12.1 Types

- RuntimeExceptions - **unchecked**, fi:
  - null-pointers
  - out-of-bounds
  - divide-by-zero
  - system resources exhaustion
- **checked** Exception - must be handled:
  - by *throws* in a method signature (**propagates to a caller**)
  - caught by *try*, *try-catch(-finally)*, *try-with*
- errors

3.12.2 Blocks

- *try* block requires:
  - **catch** block, or
  - **finally** block, or
  - that the **method throws the exception**
- *finally*
  - is commonly used to cleanup resources - Scanner, (Buffered)Reader, etc
  - an exception thrown in *finally* clause **must be always handled!** Propagation is prohibited.
- *catch* blocks should handle exceptions ordered from the **most** specific to the **most** general.
- *try-with-resources* removes the need for **finally** block. The resources must implement **AutoCloseable**.

```
AutoCloseable ac = new AutoCloseable(..);
try(ac){
  ac.read(...)
}
```

Autocloseable examples:

- FileWriter
- FileReader
- BufferedReader

3.12.3 Important Points

- **All types of exceptions can be caught! - checked, unchecked, errors.**
- **Exceptions are objects!**
- **catch clause should never be empty - it hides thrown exceptions making debuging harder.**
- **exception acronym is a common naming convention**

3.12.4 Catch or Declare Rule

Exceptions that might be trown by a method must be either

- caught in a *catch* block, or
- declared in the method signature in *throws* clause and **propagated to a caller**.

Errors and **Runtime exceptions (unchecked)** are excluded from the rule. but still can be handled in the same way if needed.

3.12.5 Overriding a method Throwing an Exception

- **We cannot add an exception!**
- we can remove some exception
- we can replace it by its descendant

It makes sense - it allows that an object of a child class can be put in all those places when an object of parent class can be used..

3.12.6 Custom Exception

In order to create your own exception we need :

- **a class extending Exceptions!**
- two constructors - no-arg (String msg, int ID)

```
//custom exception must extend Exception class
public class CusomException extends Exception{

//required no-arg constructor
    public CustomException(){
        // the main purpose to define custom exception class:
        // custom exception message.
        // The essage is send to parent constructor and
        // it initialises String message variable
        super("custom exception message)

        // this constructor takes a string parameter containing
        // exception message. This constructor allows to instantiate
        // exception object with custom defined message
        public CustomException(String message){
            super(message);
        }
    }

}
```

3.12.7 Useful methods

At least one of the following methods should be present in a **catch** clause!

- getMessage() - detailed message about exception
- printStackTrace() as getMessage(), also includes stack trace **from where** the exception was thrown all the way back **to where** it was thrown
- e.toString - returns exception type.

```
try{...
} catch (Exception e) { ...
} finally { ...
}
```

3.13 Abstract Classes & Methods

Abstract methods is a method declaration without implementation. A subclasses are intended to provide an implementation to all derived abstract methods. If a class contains an abstract method, then:

- it must be marked as an **abstract class**, too.
- it can't be instantiated.

3.13.1 Usage

To enforce subclasses to conform to some API.

3.13.2 Abstract Classes vs interfaces

See: Basics.Interfaces

3.14 Varargs

- marked by **elipsis** - (...)

  ```
  public static void main(String... args)
  ```

- must be **the last or the only** parameter in a parameter list

3.15 Static

Static members belong to class, they are not instantiated.

3.15.1 Usage

- to store values shared across all instances
- constants (marked as **final**)

3.15.2 Static Initialisers

```
static int var1, var2;
static{
var1 = ...;
var2 = ...;
}
```

3.16 Interfaces

- **Can extend multiple interfaces**.
- can have concrete methods
- can have private methods
- **we can ommit 'abstract' keyword defining methods - they are abstract by definition as interface methods**

3.16.1 Interfaces vs Abstract Classes

- abstract classes
  - let inherit fields, prevent multiple inheritance
  - define some **functionality - states & behaviour**, **which must be present in entire family of objects**.
- interfaces
  - allows multiple inheritance, have no fields (no states)
  - ensure some **behaviour** **to be present in all objects implementing an interface**

3.17 Enumerations

Special kind of class with final amount of predefined instances. It can have fields, methods, constructors, mgetters, asf, as regular class have:

```
enum myEnum{
// a call to constructor initialising object myEnumVAR1
 VAR1 (1) ;
 // another constructor call.
 VAR2 (2);

//  enum variable
 private int var;

 // a constructor(int var){
 this.var = var;
 }

 // getter
 private int var(){
  return this.var;
 }
}
```

3.17.1 values()

Returns an array of enum type:

```
enum myEnum{..};
...
myEnum[] enumArr = myEnum.values();
```

3.18 Annotations

3.18.1 Built-in

- @Override
- @Deprecated item @FunctionalInterface
- @SuppressWarnings
- SafeVarargs - Coder's assertion that the annotated method or constructor body doesn't perform unsafe operations on its varargs parameter.

Chapter 4

Interfaces

4.1 Comparable

Contains

`compareTo()`

which used by:

- Collections.sort()

Chapter 5

IO

General strategy is to wrap byte-wise IO acces into Buffered objects (readers/writers) performing IOs line-by-line.

5.1 Reading a File - BufferedReader

- constructor throws **IOException**
- need to call **.close()**
  - in **finally**
  - throws **IOException**, too. Hence should be within another try-catch
  - code-smell avoidable by using **try-with-resources**.

```
String line;
BufferedReader br = new BufferedReader(new FileReader(
                   "file-url"));
try(br){
while((line = br.readLine()) != null)
...
} catch(IOException ioe) { ... }
```

Mind:

- resources closed in ordinary brackets
- null-checked assignment enclosed within its own brackets

5.2 Writing to a File

```
try(BufferedWriter bw = new BufferedWriter(new FileWriter(
          "file-url"))){
bw.write("bla bla bla...");
bw.newLine();
}
```

5.3 Serialisation

It is a process of writing java objects as binary data in order to:

- send them over network
- store them on file system.

Individual fields can be excluded from serialisation by making them **transient**.

5.3.1 Writing - ObjectOutputStream.writeObject(Object o)

```
try(ObjectOutputStream oos = new ObectOutputStream(new FileOutputStream("file-url))){
oos.writeObject(Object o);
} catch (IOException e)
```

5.3.2 Reading - ObjectInputStream.readObject(Object o)

```
try(ObjectInputStream oos = new ObectInputStream(new FileInputStream("file-url))){
while(true){
Object o = oos.readObject();
}
} catch (ClassNotFoundException e)
```

Chapter 6

Collections

6.1 Iterator

Collections are **Iterable** - all of them use **Iterator<T>** objet. It means that the iterator object is created each time we want to loop over collections' items in order to control the iteration proces. This is ***external iteration***. The core iterator methods are **hasNext()** and **next()**.
Issues:

- hard to abstract away different behavioural operations
- inherent **serial nature**.

6.2 Common Methods

They are defined in java.util.Collection

- .add(), addAll()
- clear() - removes all elements
- contains() containsAll()
- equals()
- hashCode()
- isEmpty()
- iterator()
- parallelStream()
- remove(), remooveAll()

- size()
- sort()
- toArray()

6.3 Map

6.3.1 Literal Initialisation

Map.of():

```
someMap<K, V> map = Map.of(k, v, k, v...);
```

6.3.2 HashMap

Part III

Library Classes

Chapter 7

Scanner

Parses primitive types and strings using regular expressions. Breaks an input into tokens using whitespace as a default delimiter. Custom delimiter can be specified by useDelimiter(Pattern pattern). Pattern is a compiled representation of a regular expression.(see: Docs)
close() frees resources which implement Closeable (see: *Docs, Specs*).
Example:

```
import java.util.scanner
.....
Scanner sc = new Scanner(System.in)
if(sc.hasNextInt())
int i = sc.nextInt();

if(sc.hasNext())
String a = sc.next();
...
sc.close();
```

7.1 Key Points:

- parses primitives and Strings
- nextInt(), nextLong(), nextBool(), next()...
- delimiter - whitespace & **endline**(default)
- read object must implement **Readable**
- **must be closed to release resources**
- **not safe for multithreading!**

7.2 InputMismatchException

When thrown, the token causing the exception is not passed through; instead, it can be retrieved using another method. We can get rid of them by calling .nextLine().

7.3 hasNext(), eof

hasNext() and its methods family (hasNextInt/Double, asf), may block waiting for a next token. It happens when the last token was parsed. **These methods return false only when next token is of a different type or next token is 'end-of-file'!**. So, naive using it to terminate loops:

```
while(scanner.hasNext()){
...
}
```

..may not work as expected. It terminates (returns **false**) only when next token **is not a String!** - as almost everything is a string then the only thing (I know) that terminates is end-of-file marker. It is ctrl-D (ctrl-Z on Windows).

7.4 nextLine()

Looping over multiline input may cause unexpected behaviour because of end-of-line marker '**\n**', **It is NOT consumed by nextInt(), nextDouble()**, asf. It may make a use of the next call to next() method and generate unexpected behaviour. In this case, next() consumes '\n' and returns an **empty string**.

One of the ways to deal with this is to use **nextLine()** - it takes all tokens that left in the current line, end-of-line including, and then sets the marker on starting position of the next line (if it exists). **nextLine() consumes \n and discards it, but it doesn't return it:**

```
 String line = sc.nextLine();
 System.out.println("all this " + line + " is on single line
 - NO NEW LINE IS HERE!" );
```

Another - to use **second scanner** and feed it with a line fetched by first scanner.
/sectionTips

- always prompt a user for an input
- always echoe user input to discover early problems, and to make the user a feedback, what was input.

Chapter 8

Integer

8.1 Boxing

- Using int value:

  ```
  Integer var = Integer.valueOf(int a);
  ```

- Using String:

  ```
  Integer var = Integer.valueOf(String s);

  // This is an equivalent of:
  Integer var = Integer.valueOf(Integer.parseInt(String s));
  ```

8.2 Unboxing

```
Integer var = integer.valueOf(4);
int i = var.intValue();
```

8.3 Conversion from String

```
int i = Integer.parseInt(String s);
```

Chapter 9

Date & Time

9.1 Date

Textual output:

```
(new Date).toString();
```

returns format

May 04 09:51:52 CDT 2009

9.2 Date + String.format(..)

Syntax:

- placeholders starts with 't':

  ```
  Date date = new Date();
  String.fomat("%tc %ty" )
  ```

- 't' indicates a date/time placeholder

  ```
  "%tx..."
  ```

- useful String.format() syntax:
  - explicit positionning, **$** terminates

    ```
    String.format("%1$ %2$.....");
    ```

  - shortcut for repeated args, <:

    ```
    String.format("%tc %<te %<tm.... ", date);
    ```

Placeholders Indicators

Character Output height	Description
	Full date formater
c	default
Mon May 04 09:51:52 CDT 2009 F	ISO
2004-02-09 D	US formatted
02/09/2004 height	
	24/12 Hours
T	24-hour
18:05:19 R	24-hour, no secs
18:05 r	12-hour
06:05:19 pm height	
	Year
Y	full
2018 y	last 2 digits
18 C	century
20 height	
	Month
B	full
January b	abbreviated
Jan m	2-digit with leading zero
01 height	
	Day
d	with leading zero
07 e	without leading zero
7 height	
	Weekday
A	full
Friday a	abbreviated
Fri height	
	AM/PM
P	uppercase
PM p	lowercase
pm height	
	Hours
H	24h with leading zeros
21, 07 k	24h without leading zeros
21, 7 I (uppercase 'i')	12h with leading zero
09 l (lowercase 'l')	12h without leading zero
9 height	
	Min's, Sec's
M	minutes
25 S	seconds
58 height	
	Varia
s	Seconds since 1970-01-01 00:00:00 GMT
1078884319 Z	timezone
EET, GMT z	offset from GMT
+0200 height	

9.2.1 SimpleDateFormat

In order to format date/time:

- initialise a pattern object:

  ```
  SimpleDateFormat f = new SimpleDateFormat("yyyy/mm/dd");
  ```

- apply the pattern to a date object using format() method:

  ```
  f.format(date);
  ```

9.3 String

9.3.1 String.format(pattern, args)

- use

  ```
  String.format("%2$s %1$s , arg1, arg2);
  ```

- use < to use **the same** arg for the **next** placeholder

  ```
  String.format({%s %<s"}, arg);
  ```

9.3.2 Splitting a String

`stringObject.split(String delimiter)`

The method splits the input string using the delimiter stringtaken as the method argument. No-arg overloded version of the method takes **spaces as a default delimiter**.

9.4 java.utils.Arrays

- asList(T.. a) - creates a list from some collection of type T
- binarySearch(..)
- copytOf(..)
- equals(..)
- .fill()
- toArrayList()
- toArray()
- sort(..)
- .toString()

9.5 Random

9.5.1 Math.random()

Returns a radom value x such that

```
0.0 <= x < 1.0
```

Idiom to get int x from a Custom Range 0..y

Remember to:

- add 1
- cast to int

```
int x = (int)(Math.random() * y + 1)
```

9.5.2 Random class

An object of Random class encapsulates **a stream** of random values. Individual random values can be extracted using various next..() methods - nextInt(..), nextLong(..) asf.

- custom range
- custom numeric type

```
int val = (new Random()).nextInt(range);
```

9.5.3 Random class vs Math.random()

- two Random objects with the same seed will produce the same random values. We need to define distinct seeds in order to get distinct random values.
- Random class objects are **more efficient** and **less biased**.

Part IV

OOP

- UML Distilled textbook by Martin Fowler
- **is-a** relationship - inheritance
- **has-a** relationship - association
- **CRC** - Class-Responsibility-Collaboration (`http://agilemodeling.com/artifacts/crcModel.htm`)
  - **responsibility** - states and behaviours
  - **collaboration** - other classes that this class is somewhat awared of (association, aggregation, composition).

9.6 Abstraction

There are different meanning of abstractin. one of them is the ability to capture real world entities as classes. Two types of abstractions in Java:

- **interfaces**, used to define expected behaviour.Implementation **is hidden from a client**.
- **abstract classes**, used to define incomplete functionality.

9.7 Inheritance

The ability of subclass to derive members (fields and methods) from ascendands. In java only single parent class is allowed. It is an **'is-a'** relationship.

9.7.1 Accessing Members

It is valid to instantiate an object with a subtype. The instantiated reference variable allows an access to those members (and their variations) which are present in their type. It is still possible to access subtype members using cast:

```
Parent childParent = new Child();
\\ access to a member as it is defined in the Parent class.
childParent.field...

\\ access to a member as it is defined in the Child class
((Child)childParent).field
```

Pay attention to the syntax of above cast!

types	fields	cast	method calls
Parent pc = new Child()	parent	**to child**	**child** (*)
Parent p = new Parent()	parent	**error!**	parent
Child c = new Child()	child	**to parent**	child
Child c = new Parent()	**error!**		

*** Those method which are overridden are accessible as usual. In order to call methods that are not overridden the reference variable used to access the methods (here - *pc*) must be cast to (Child)**

9.7.2 Pitfalls

Constructor Calls an Overridable Method

1. call to constructor in **child class**
2. it calls **parent class constructor** first
3. if there is a call to overridable method it calls **textchild version of the method**
4. **ERROR!** The call **will fail** if the method references some uninitialised variable. **The variable can be initialised only when the control returns to child constructor - in steps which will folow!**

9.8 Encapsulation

ineer details of classes can be hidden by making them private and acceptible through public API only - getters (accessors) and setters (mutators).

9.9 Polymorphism , Method Overriding

'Many forms'. implemented by

- subclass specialisation (*is-a'* relationship
- Liskov substitution principle
- virtual method invocation)

Polymorphism is usually achieved by method overriding.

9.9.1 Liskov Substitution Principle

Whenever an instance of some class is expected in a program, one can suply an instance of subclass of the class

9.9.2 Virtual method Invocation

Method calls are dynamically dispatched based on runtime type of the receiver object.

9.10 Method Overloading

Overloading means that two or more methods have the same name but different signature. They **must have different parameters** (number of them and/or types), they **may have different return type and access modifiers** (private, protected, etc) - see Rules.

9.11 Method Overriding

It means that new implementation is provided to an inherited method. This is annotated by @Override. The overridden method in a superclass can be still called when invoked using **super.** keyword.

Rules

- **final, static, private** methods can't be overridden.
- access modifier of overriding method **must not be more restrictive**.
- **no new checked exception** can be thrown
- if return type is a reference type, then it can be original type or any descendant of this type (**covariant return type**).

Private methods can't be overriden because they are excluded from inheritance (not visible from within subclasses).
Static methods reside in static context, they belong to class, not to objects. Therefore they are not inherited, too. Hence, they can't be overriden.
However, they can be **shadowed** - subclass can have static method with the same name, as its parent class. A method call is bind to first method with a proper signature - starting from the class in current context and going up the inheritance tree.

9.12 UML modelling

9.12.1 Domain Modelling

Focuses on **multiplicities** and **directionalities**:

- no details about class content
- details about inheritance direction
- details about type of relationship - 1:1, 1:M, M:M

Inheritance

White arrows pointing from children classes to parent class.

Association

An arrow pointing from *knowing* class to *known-about* class.

9.12.2 Class Diagram

More internal details about classes. It includes:

- attributes
- behaviours
- types
- visibility - public (+), private(-), protected ()

attributes and behaviours.

Part V

Lambda expressions and Aggregate Operations

Chapter 10

Lambda Expressions

10.1 Motivation

Abstraction in OOP is an abstraction over data. Lambda expressions add an abstraction over beaviour.
Functional programming - problem domains expressed in term of **immutable** values & functions that translate between them.

10.2 Lambda Expressions - Overview

- nameless method
- intended to pass around behaviour
- **allows to call method direct on interfaces**.

Traditionally, behaviour is wrapped into an inner class in order to send - **code as data**. A class has to implement a functional interface.
In lambda expression, we don't have to provide types explicitly.They are inferred from a context - a signature of the method in corresponding functional interface.
Lambda expressions are statically typed!

Every time an object implementing functional interface is in use - as method parameter or return value, lambda expression can be used insted.

Examples

- no argument, no return

  ```
  FuncIt fi = () -> ();
  ```

  Example:

```
Runnable noArgument = () -> System.out.rintln("this lambda takes no argument");
```

- no argument, 1 return

  ```
  FuncInt fi = () -> a;
  ```

- 1 argument, no return

  ```
  FuncInt fi = () -> a;
  ```

  Example:

  ```
  ActionListener oneArgument = event - > System.out.println("this is an event");
  ```

- no arguments, 2 returns

  ```
  FuncInt fi = () -> {
  do1();
  do2()
  }
  ```

  Example:

  ```
  Runnable multiStatement = () -> {
  System.out.println("Multi-return);
  System.out.println("lambda expression");
  }
  ```

- 1 argument, 1 return

  ```
  FuncInt fi = (a) -> b;
  ```

- 2 arguments

  ```
  FuncInt<Long> fi = (a, b) -> a + b;
  ```

  Example:

```
BinaryOperator<Long> add = (x, y) -> x + y;
```

the same with explicit types:

```
FuncInt<Long> fi = (Long a, Long b) -> a + b;
```

Example:

```
BinaryOperator<Long> add = (Long x, Long y) -> x + y;
```

10.3 Target Type, Type Inference

The type of an expression is define by the context in which lambda appears:

- method parameter
- variable assignment

They are used to **infer** lambda type.

Type Inference Rules

- **single possible target type**
  The lambda expression infers its type from the corresponding parameter on the functional interface
- **several possible types can be inferred**, all belong to the same inheritance tree
  the most specific type is inferred
- **several possible types, none of them is the most specific**
  compile error!

10.4 Use of External values by Lambda

As functional programming is about transiton from one **immutable** value to another one, hence:

- external values have to have similar properties
- they must be **effectively** final *the external variable used by lambda need not to be declared as final as long as it is not reassigned later in the code*

Examples:
1. This compiles:

```
String name = "Joe";
FunctInt.do(input -> "hi " + name);
```

2. This not:

```
String name = "Joe";
name = "Kate"; // NOT effectively final!
FuncInt.do("Hi " + name);
```

Compile error - variable *name* is reassigned, hence not effectively final.

10.5 Lambda Graphically

```
FunctInt fi = a -> b;
```

Graphically equivalent:

```
a -> FunctInt -> b
```

10.6 SAMs, Core Functional Interfaces

Functional interface (aka SAM - Single Abstract method) - an interface, which contains exactly one **abstract** method, and which is intended to support lambda expressions. Predefined functional interfaces (part of java API):

- **Predicate**<T>, T $\rightarrow$ boolean, **test()**
- **Consumer**<T>, T $\rightarrow$ void, **accept()**
- **Function**<T, R>, T $\rightarrow$ R, **apply()**
- **Supplier**<T>, () $\rightarrow$ T
- **UnaryOperator**<T>, T $\rightarrow$ T
- **BinaryOperator**<T>, (T, T) $\rightarrow$ T, **apply()**

10.7 Custom Functional Interfaces

- require **@FunctionalInterface** annotation to be applied
- the annotation signals that the interface **is intended** to use for lambda expression
- annotation compels javac to check whether the interface meets the criteria to be a functional interface. **helpful by refactoring!**
- just **single method** in an interface **doesn't make it a functional interface!**

  - *Comparable*
    The author explanation - because functions are not comparable (have no fieldsm no states). **I'm not sure, that this is good explanation**
  - *Closable*
    Has to do with resources, which opens and closes. Clearly - they mutate, hence this is not **pure function**.

10.8 Method References

This is a shortcut notation for some lambda expressions which include **a call to existing method**. **Argument types and their number are inferred**. Four kinds:

- **static** method reference
- **instance** method reference
- **arbitrary object of specified type** method reference
- **constructor call**

10.8.1 Static Method Reference

```
(args..) -> T.call(args..)
```

is equivalent to:

```
T:call
```

..where **T is a class name**

10.8.2 Instance Method Reference

```
(args..) -> t.call(args..)
```

can be replaced by ('t' is an object identifier):

```
t::call
```

10.8.3 Referencing a Method of an Arbitrary Object of a Particular Type

```
(T a) -> a.call()
```

is equivalent to

```
T::call
```

The syntax similar to static method referencing.

10.8.4 Referencing a Constructor

```
(T a) -> new T()
```

can be replaced by:

```
T::new
```

10.8.5 Applications

Instantiate an Array

```
String[] :: new;
```

10.9 Overloading

If lambda expression is backed by overloaded methods, then it means that several types can be inferred. A compiler will try to pick the method with the most specific type which suits. It fails to compile when it is not clear which type is the most specific, for instance **lambda expression is ambiguous** and two or more non-related types are possible to choose.

Chapter 11

Aggregate Operations (streams)

11.1 Motivation

- reduces boilerplate code
- adds parallelism

11.2 for-loop vs stream

This is **external iteration**:

- calls constructor → iterator(),
- check next → hasNext()
- take next → next(), nextInt(), asf..

Streams are **internal** iterations invoked by calling **stream()**.

11.3 Stream methods

There are two types of stream methods:

- lazy, they **return another stream**.
  - .filter(a → b), b $\subseteq$ a
  - .map(element → mapFunc());
  - .flatMap(ListOfLists → concatenated_lists)
- eager, they **return a value or void**

  - .count
  - .collect(Collectors.toList(), ..ToSet(), asf..
  - .max(), .min()
  - .reduce(initVal, *reducer*)
    - **BinaryOperator<T, V>reducer = (acc, item) → func(acc, item)**

11.4 Stream<T>object

We can use it to pass stream over our code - for instance as an argument to call a method or as a return type.

11.5 Ordering

Encounter order - the order streamed items appears. Depends on source data and operationson Stream:

- Collection with a defined order
  - *the order is preserved.*
- unorder Collection
  - *undefined order*
- the order is propagated through intermediate operations

11.6 min(), max()

```
List list....
list.stream()
      .min(Comparator.comparing(ListItem -> getFieldToCompare()))
```

min(), max() are eager functions. Takes Comparator argument, which is returned by a static ***comparing()*** method of ***Comparator*** class. This method takes a lambda expression of type **Function** (x ->y), which specifies which field is used to compare items from the list. See Aggregate Operations/.collect(..) for a different way to get min or max

11.7 .collect(..)

.collect(Collector.toList()), .collect(Collector.toSet()) are just some examples. All collectors can be found in *java.util.stream.Collectors* class; Appropriate implementation (specific type) is picking under the hood. In order to specify the type explicitly pass a constructor as an argument, f.i):

```
stream.collect(toCollection(BinaryTree::new));
```

Can be used to get a single value, too:

```
stream.collect(maxBy(comparing(someValue)));
```

11.7.1 Partitoning, Grouping

```
.collect(partitioningBy(Predicate))
```

... splits into two group using Predicate formula as a dscriminator.

```
.collect(groupingBy(Classifier))
```

Similar to *partitioningBy()*, but splits to arbitrary number of groups using *classifier.//* **Classifier** is a *Function*, which can specify which fied to use as a discriminator.

11.8 Strings

11.8.1 Streaming a String

```
stringObject.chars()
```

returns a stream of integers values equal to characters streamed from input string. To convert back to character representation we need to cast stream items to char.

Example:

```
strObject.chars()   \\ converts the string into an Integer stream.
         .mapToObj(i -> (char) i)     \\ casts Integer stream
                    \\ into Char stream
         .collect(Collectors.toList());
```

Parallel version doesn't preserve the order!

11.8.2 String Representation of Collections

```
someCollection.stream()
                              .map(Collection::getField)
                              .collect(Collectors.joining(", ", "[", "]"));
```

11.9 Primitive Type Optimisation

The goal of the optimisation is to remove overhead caused by boxing/unboxing operations when numerical operations are performed over some collection. Naming conventions depends on a location of primitive type (argument or return):

- **return**: to-type-name, like *toIntFunction*
- **argument** - type-name, like *LongFunction*
- **higher order functions** - function-To-type, like *mapToLong*
- **specialised stream** - type-Stream, like *longStream*

Example: We can use mapToObj(Function) to convert integer stream (primitive type) into Character list:

```
IntStream(22, 23, 24)
    .mapToObj(i -> (char) i)
    .collect(Collectors.toLost());
```

11.10 .summaryStatistics()

Works on specialised stream of some primitive type only. Hence, we need to convert objects' stream, first, for instance using *mapToInt(), mapToLong() or mapToDouble()*
.summaryStatistics() allows to get single value as a return from a stream, like:

- getmax()
- getMin()
- getAverage()
- getSum()

11.11 Patterns & Idioms Using Aggregate operations

Filter Pattern

Traditionally:

```
for(item : List){
if(conditionOn(item))
filteredList.add(item);
}
```

With aggregate methods:

```
FilteredList fl = List.stream()
.filter(item -> conditionOnItem())
.collect(toList();)
```

Reduce Pattern

Traditionally:

```
var accu = initVal;
for(item : List){
combine(accu, item);
}
```

Aggregate version:

```
.reduce(initVal, (acc, item) -> func(acc, item));
```

'func' is of BinaryOperator type - $(T, T) \rightarrow T$.

When initVal is missing, then first two items are used instead.

Chapter 12

Advanced Topics

12.1 Reflection

Each class is represented by an object, which is created when the class is loaded. The object is accessible by getClass(). It encapsulates:

- properties of class members
  - access level
  - type, etc
- behaviours of class members
  - getters & setters
  - method invocations

12.2 Immutability

- safe to used in multithreading
- perfect values for keys in Map

12.2.1 Implementing Immutability

- all fields private
- no setters
- constructor sets all the internal state of te object
- if field refers to some reference type, then associated getter returns a reference to copy, not to the original object

12.3 Default Methods

Introduced to java o preserve **backward binary compatibility**. it means, that programs compiled in previous java version will still compile in new java versions. **Default methods** are a solution to maintain backward binary compatibility after streaming methods were added to Collections. Those methods provide Collection descendants (without stream methods) whith missing methods.

Inheritance/Overriding Rules

Generally - **class methods wins over interface (default) methods**. Otherwise - standard overriding rules.

ATTENTION!

Generally when we've got interface reference to some class, we're interested only on this part of class API which is defined by the interface (which the class must inherit):

```
SomeInterface si = new ClassImplement();
```

Here *si* references only the methods which are specified by underlying interface. But:

```
ChildInterface ci = new ChildClass();
```

The same default method ParentInterface is overrided by ChildInterface *(which is the most specific)* and ParentClass. **ParentClass has a priority** over ChildInterface! This prevents default methods from breaking existing inheritance trees in older libraries.

Multiple inheritance (from multiple interfaces)

If a method can be inherited from more than 1 interface default method - **compile error!**. To solve:
- **specify the interface to inherit from** using keyword *super*:

```
SomeClass implements Interface1, Interface2{
  @Override
  void defaultMethod(){
    return InterfaceX.super.defaultMethod();
  }
}
```

Keyword *super* can be used to address parent class API, but to address implemented interface API as well

Chapter 13

Data Parallelism

13.1 Intro

13.1.1 Notions

Concurrency

More than one task is done at the same time exploiting **processor time slices**.

Parallelism

Splits a task in couple of chunks, each assigned to a different processor core, then chunk results are combined into single final result.

Task Pararellism

Each individual thread of execution can do totally different task.

13.2 Impact on Performance

Depends on:

- size, usually
  - <100 → **serial** execution is faster
  - ∼100 → **both** equally fast
  - >100 → **parallel** execution is faster

  Those results are due the overhead of decoposing the problem and merging the result.
- source data structure

    - **ArrayList, array, Stream.ranges()** are **cheap** to split
    - costs of splitting **HashSet,, TreeSet** is **acceptable**
    - **LinkedList** is **expensive** to split, usually O(n)
    - **Streams.iterate, BufferedReader.lines** are **hard to estimate** - unknown length at the beginning.

- primitives are faster to operate on than boxed values
- cost per element
    - the more expensive execution of single element, the more likely performance increase
- **stateless** are faster than **stateful** no need to maintain state
    - stateless - **.map, .flatmap, .filter**
    - statefull - **.sorted, .distinct, .limit**

Exact numbers can vary on different machines.

13.3 Parallel Stream Operations

.parallelStream() - parallel version of stream; *.parallel()* - makes a stream parallel at custom point in method chain.

13.4 Caveats

The use of locks can lead to nondeterministic deadlocks!

13.4.1 Parallel .reduce()

- initVal must be the **identity** of combining function *(0 for adding, 1 for multiplying, asf)*
- combining function must be **associative** *the order of execution doesn't matter.*

13.5 Arrays

Some methods:

- *parallelPrefix*
  - calculates totas given an arbitrary function
- *parallelSetAll*
  - updates values using lambda
- *parallelSort*

Example

```
int[] arrayInitialiser(int size){
  int[] values = new int[size];
  for(i = 0; i < values.length; i++){
    int[i] = i;
  }
  return values;
}
```

Parallel:

```
int[] parallelArrayinitialiser(int size){
  int[] values = new int[size];
  Arrays,setAllParalel(values, i -> i);
  return values;
}
```

Application of *parallelPrefix()*. **simplemovingAverage** It takes 'rolling window' of n array items an replace values with an average of n previous values:

```
double[] simplemovingAverage(double[] values, int size){
  double[] sums = Arays.copyOf(values, values.length);

  //new array holds runnung totals of the sums so far. Hence, sum of last n-items will be th
  // sum[i - n] is hoeld in prefix.
  Arrays.parallelPrefix(sums, Double :: sum);
  int start = n - 1;
  return IntStream.range(start, sums.length)
                 .mapToDouble(i -> {
                   // 0 assigned to prefix initially, as start - n equals (n - 1) - n = -1
                   double prefix = i == start? 0 : sums;
                   return (sum[i] - prefix) / 2;
                 }
                 .toArray();
                 )
}
```

Is cleaner code possible?

Part VI

no title yet..

Chapter 14

Idioms

14.1 Exceptions

14.1.1 Exception-Driven Loop

```
boolean terminateLoop = false;
while(!terminateLoop){
  try{
    /some code throwing an exception/    // jumps to catch clause
    terminateLoop = true;                 // terminates this loop
  } catch (SomeException e){
    /some code/
  }
}
```

Chapter 15

Design Patterns

15.1 Singleton

15.1.1 code

```
class Singleton{
  // private & static instance variable.
  private static Singleton obj;

  // private constructor
  private Singleton(..){
    ...
  }

  // the only publicly exposed member - instance getter
  public Singleton getObj(){
  if(obj == null) obj = new Singleton();
  return obj;
  }
}
```

15.1.2 Description

Object getter is the only publicly exposed API. It returns private static instance variable holding the singleon object. If the object is null, it calls private constructor. There is no other way to call the constructor (and to instantiate another object).

15.1.3 Usage

All cases, when **exactly one instance** is required:

- network session

- solar system, Milky Way

Chapter 16

Antipatterns, Code Smells

16.1 AutoBoxing, Unboxing

Recommended

Unwrapping primitive types from collections.

```
ArrayList<Integer> integer;
.....
int i = integer.get(..);
```

Not Recommended

This is not recommended because there is no point to force autoboxing:

```
Integer i = 5;
```

Better:

```
Integer i = Integer.valueOf(5);
```

Forced unboxing is also not recommended:

```
Integer a = Integer.valueOf(1);
int b = 2;
int c = a + b;
```

Better:

```
Integer a = Integer.valueOf(1);
int b = 2;
int c = a + b.intValue();
```

16.2 Constructors

16.2.1 Constructor Calls an Overridable Method

1. call to constructor in **child class**
2. it calls **parent class constructor** first
3. if there is a call to overridable method it calls **textchild version of the method**
4. **ERROR!** The call **will fail** if the method references some uninitialised variable. **The variable can be initialised only when the control returns to child constructor - in steps which will folow!**

Chapter 17

Testing

17.1 JUnit

17.1.1 Unit testing

It could be a function class, package, subsystem.

Part VII

Appendix

Chapter 18

Vocabulary

18.1 M

members -

fields and methods

Chapter 19

Docs, Specs

- Oracle JDK 9 Documentation
  `https://docs.oracle.com/javase/9/`
- The Java® Language Specification
  `https://docs.oracle.com/javase/specs/jls/se9/html/index.html`

19.1 Java Doc

- Pattern - `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`
- Scanner - `https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html`