Contents

Part I

Intro

Chapter 1

TODOs

gh

1.1 Readings

- notes for reference type conversions, using a chapter in 'java in nutshell'
- Thinking Java, IO, Passing Objects.Clonning
- *Java 8 Lambdas* continue from 'Chapter 7'
- *Java Pocket guide* continue from 'Chapter 10'
- https://www.ibm.com/developerworks/learn/java/index.html
- javapractices.com
- UML Distilled textbook by Martin Fowler
- `http://agilemodeling.com/artifacts/crcModel.htm`
- Check Absolute Java chapter 8, binding.
- *Test-Driven Development* by Kent Beck
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce

1.2 Research Topics and Tasks

- add examples to 'primitive type optimisation'
- more about copy constructor, examples & make notes (under constructors ot referencetypes.clonning?)

- format spacing starting from Basic.Switch_Statement
- review implemented setters to add error-checks. Add setters when such error checks are reasonable, for instance to avoid negative ages, sizes, etc.
- investigate IntStream.peek()
- investigate further application of 'characteristic' parameter of Collectors.of() - custom collector
- investigate StringJoiner library class
- investigate streamSupplier, streamSupplier.get()
- IntStream and Arrays.stream(), difference
- fix 'placeholder indicators in Time-Date chapter(Library Classes
- 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...
- 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
- 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
- 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

- why inner classes (also lambda) require external local values to be immutable (final in java)
- optional as a replacement for null, p. 55 - 56 in 'Java8 Lambdas' book; applications
- 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
- check signature of Arrays.parallelSort, Arrays.parallelSetAll, Arrays.parallelPrefix, and update 'DataParalelism'Arrays'
- 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

Resources

Notes based on:

- UoG java labs classes
- books
  - *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
  - *Thinking in Java*, B Eckel

Part II

Basics

Chapter 3

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)

Chapter 4

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

4.1 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

4.2 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.

4.2.1 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'.

Chapter 5

Modifiers

5.1 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

5.2 Recommendation on How to Use Access Modifiers

It is safe to start with restrictive access level and then to relax it. **The opposite approach can break some existing code**.

5.2.1 public

Only for **methods** and **constants** that form **part of public API**.

5.2.2 protected

For members that are supposed to be used outside of the package.

- **documment them!**
- **don't change them!** It can break the external code.

5.3 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)

Chapter 6

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;
  ```

6.1 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

6.2 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:

6.3 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'

Chapter 7

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.'

7.1 List of Reference Types

- classes
- interfaces
- enumerations
- arrays
- annotations

7.2 Reference vs Primitive Types

	Reference Type	Primitive Type
amount	**unlimited**, user defined	boolean and numeric only
memory content	**references** to data	**actual** 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**

7.3 Default Values

they are the values assigned implicitly to not initialised variables.

7.3.1 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.

7.3.2 Arrays

Default values depends on array's elements type:

- **reference types - null**
- **primitive numerical types - 0**

7.4 Object Instantiation

7.4.1 Class Loading

- Classloader finds and loads the class from which an object is to instantiate. It also loads the class when there is an attempt to use a static member of the class.
- Classloader goes upward the inheritance tree and loads all classes yet not loaded.

7.4.2 Static Initialsation

It starts in the class highest in the inheritance tree. Then initialisation continous going downward the tree.

7.4.3 Class Variables Initialisation

They are set to their default values

- primitive type variables are set to 0, 0.0, false respectively
- reference type variables are set to **null**

7.4.4 Constructor Calls

It follows the procedure described in Constructor/'Constructor Chaining'.

7.5 Cloning

7.5.1 Overview

The most common usage is when we need to manipulate an object and the original object must stay intact. For instance we want to make the original object **immutable** - *read-only*.

All classes inherites **clone()** method from *Object* class. In order to avoid default cloneability the method is **protected** - can be used only within the package and by subclasses. As an example, most classes in java standard library don't override **clone()**, they inherit its protectd version. That's why we can't call clone() on an object of *Integer* type.

There are two options - shallow and deep cloning. Default option of clonning is the **shallow clonning** - only the 'surface' of the cloned object is new created, however both objects share those fields, which are of reference type. This is necessery because there is no guarantee, that those are *cloneable*. There is no programmatical support in Java for **deep clonning - all fields of reference types all the way downward are also cloned**. Deep clonning must be implemented manually.

7.5.2 Switching Cloneability

Turning ON

In order to expose clone() in the public API we need:

- implement **Cloneable** interface
- override inherited ***clone()***
  - extend its visibility to **public**
  - call ***super.clone()*** within
    - surround it with *try-catch* and catch **CloneNotSupportedException**
    - **cast** the return from *super.clone()* to the destination type

Once a class is made *cloneable*, everyting what derives from the class is also cloneable - the public visibility of clone() can't be reduced.

clone() Overriding

The simplest way:

```
public T clone(){
    try{
        return  (T)super.clone();
    } catch (CloneNotSupportedException cnse) {}
}
```

The Purpose of Cloneable Interface

This is an empty interface, there is no method inside. The interface works as a kind of flag - it indicates that the class is intendent to be cloneable. This is checked when super method in Object class is invoked.**Object.clone()** tests whether Cloneable is implemented or not. If not - it throws **CloneNotSupportedException**.

7.5.3 Shallow Cloning

Is done by **clone()** method. **Cast is mandatory** - clone() returns an object of **Object** type. 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.

7.5.4 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.

7.5.5 Serialisation and Copy Constructor

An alternative approach to cloning. TODO..

Chapter 8

Arrays

8.1 Permitted Element types

- primitive types
- reference types
- **structered types are not allowed!**

  Arrays reserve fixed memory size. Collections require dynamic memory size defined in runtime. Therefore defining an array of Collection type causes an error.

8.2 Definition, Instantiation, Initialisation

8.2.1 Definition

```
<T>[] arrName;
```

Specifies:

- elements' type
- variable identifier

8.2.2 Instantiation

```
array = new <T>[size];
```

Assignes memory space to the array acording to its elements' size and number. Can be combined together with a definition:

```
<T>[] array = new <T>[size];
```

8.2.3 Initialisation

Arrays are initialised with default values during instantiation **in runtime**:

- **0** for **numerical primitive types**
- **null** for **reference types**.

Options to assign custom values:

Manually

- using **loops**
- using **literals** (**surrounded with curly braces, size is ommitted!**)

  ```
  int[] array = new int[]{1, 2, 3, 4};
  or
  int[] array = {1, 2, 3, 4};
  ```

- **copying existing array using clone()**. It creates a shallow copy - equal values for primitive types, copy of references for reference types, not copy of objects! This combines definition, initialisation and instantiatin in one-go:

  ```
  int[] array = (int)anotherArray.clone()
  ```

- using **System.arraycopy(...)**:

  ```
  System.arraycopy(src, idx, trget, idx, numberOfElements);
  ```

- using streams. Example below

  ```
  int[] intArray = IntStream.of(intArray)
      .mapToInt(i -> 44)
      .toArray()
      .clone();
  ```

8.3 Anonymous Arrays

Arrays can be instantiated anonymously, inline, as an expression. Example:

```
System.out.println("This is a string array: " +
                  new String[]{"first item", "second item")});
```

8.4 Converting a List to an Array

8.4.1 list.toArray()

Returns **Object**[]. Any attempt to cast to some other type will cause that an exeption will be thrown.

8.4.2 list.toArray(T[])

Example:

```
void makeArray(ArrayList<String> as){
  String[] array = as.toArray(new String[0]);
}
```

Following documentation - toArray(T[]) specifies a type of returned array and returns the array is big enough. Otherwise the method creates the array with the proper size. So, we can:

- invoke the method with an array with **size equal to 0** - new array is always created.
- invoke with the array with the **size equal to the size of converted list**:

  ```
  void makeArray(ArrayList<String> as){
      String[] array = new String[as.size()];
      as.toArray(array);
  }
  ```

8.4.3 list.stream().toArray(T::new);

8.5 Type Conversion

Arrays of primitive types are **invariant** - an attempt to cast raises compile error. Reference type conversion is allowed.

Chapter 9

if/else

9.1 Nested if's problem

If curly brackets are missing then the rule is that **'else'** statement is connected with **the nearest 'if' statement**. **The indentation is ognored!**:

```
if(cond1) print("first);
     if(cond2) print("second");
else print("else");
```

'else' is printed when **cond2** in nested **if** is **false**.

9.2 else if

```
if cond1
else if cond2
```

is equivalent to:

```
if cond1...
else {
     if cond2....
}
```

Chapter 10

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.

Chapter 11

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*;

Chapter 12

Text-based User Input

12.1 Program Arguments

- program input argument

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

12.2 Using an Object of Scanner Class

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

12.2.1 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

12.2.2 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).

12.2.3 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.

12.2.4 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.

Chapter 13

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*)

13.1 Constructor Chaining

Constructors are not inherited. They are chained instead!// 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().**

- **(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**.

13.2 Practices

13.2.1 Define no-arg Constructor when a Constructor Taking Arguments Is Present

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();
```

Otherwise:

- **variable instantiation**:

  ```
  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**.

- **implicit call to superclass constructor** will cause an error.

13.2.2 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_overridable_method

Chapter 14

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.**

Chapter 15

super

Refers to superclass. Snatx:

`super.superClassMember`

Common usage:

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

15.1 super.a() vs. ((<T>)this).a()

Using super we refer to **the most immediately** implementation of overridden method. Using cast we can refer to **any implementation** of overridden method up in the inheritance tree.

Chapter 16

Exceptions

16.1 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

16.2 Try 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

16.3 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**

16.4 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.

16.5 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..

16.6 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 CustomException 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);
        }
    }

}
```

16.7 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 { ...
}
```

Chapter 17

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.

17.1 Usage

To enforce subclasses to conform to some API.

17.2 Abstract Classes vs interfaces

See: Basics.Interfaces

Chapter 18

Varargs

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

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

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

Chapter 19

Static

Static members belong to class, they are not instantiated.

19.1 Usage

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

19.2 Static Initialisers

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

Chapter 20

Interfaces

- **multiple interfaces can be implemented**.
- **all method are implicitly public**
- can have concrete methods
- can have private methods(
- can have varuables - **final static** (constanses)
- **we can ommit 'abstract' keyword defining methods - they are abstract by definition as interface methods**

20.1 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**

Chapter 21

Enumerations

Special kind of class with final amount of predefined instances.

- It can have fields, methods **(main() included)**, constructors, getters, asf, as regular class have
- it can be **standalone** or be **nested**, too
- **toString()** default behaviour is to return enum object identifier
- we can define **abstract** classes as well; we must provide method definition for each enumerated object obviously
- Java enums extend the java.lang.Enum class implicitly, so your enum types cannot extend another class
- If a Java enum contains fields and methods, the definition of fields and methods must always come after the list of constants in the enum. Additionally, the list of enum constants must be terminated by a semicolon;

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

 // another constructor call.
 VAR2 (2);

//  enum variable
 private int var;

 // a constructor
 myEnum(int var){
     this.var = var;
 }
 // getter
 private int var(){
  return this.var;
 }
}
```

21.1 Iteration)

values() returns an array of enum type:

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

Then the values can be accessed in this way:

```
 for{myEnum m : myEnum.values(){
    m.toString();
 }
```

Chapter 22

Annotations

22.1 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 23

Inner Classes

23.1 Overview

- Inner class can directly acces all members of sorrounding class.
- An object of inner class can be instantiated only in association with an object of the sorrounding outer class.
- **cant have a static class as its member**

23.2 localisation within a Code

We can declare inner class within:

- a class
- method
- .. and any other arbitrary scope.

23.3 Motivation

The intention of inner classes is:

- to group classes which logically belong together
- to control the visibility of one within the other.
- to completel prevent type-coding dependencies
- to completely hide implementation details
- **only inner classes can be private or protected!**

The best use of inner clases is when the outer class is uocasted. In such case the inner classes become completely invisible - perfect implementation hiding.

Chapter 24

Inner Static Class

24.1 Overview

- cannot acces members of sorrounding outer class
- no need to instantiate the outer class

Part III

IO

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

Chapter 25

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

Chapter 26

Writing to a File

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

Chapter 27

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**.

27.1 Writing - ObjectOutputStream.writeObject(Object o)

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

27.2 Reading - ObjectInputStream.readObject(Object o)

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

Part IV

Collections

Chapter 28

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**.

Chapter 29

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()

Chapter 30

Map

30.1 Literal Initialisation

Map.of():

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

30.2 HashMap

Chapter 31

ArrayList

31.1 Converting a List to an Array

31.1.1 list.toArray()

Returns **Object**[]. Any attempt to cast to some other type will cause that an exeption will be thrown.

31.1.2 list.toArray(T[])

Example:

```
void makeArray(ArrayList<String> as){
String[] array = as.toArray(new String[0]);
}
```

Following documentation - toArray(T[]) specifies a type of returned array and returns the array is big enough. Otherwise the method creates the array with the proper size. So, we can:

- invoke the method with an array with **size equal to 0** - new array is always created.
- invoke with the array with the **size equal to the size of converted list**:

  ```
  void makeArray(ArrayList<String> as){
  String[] array = new String[as.size()];
  as.toArray(array);
  }
  ```

31.1.3 list.stream().toArray(T[]::new);

Clean & elegant.

Part V

Lambda Expressions

Chapter 32

Lambda Expressions

32.1 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.

32.2 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;
```

32.3 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!

32.4 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.

32.5 Lambda Graphically

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

Graphically equivalent:

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

32.6 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()**

32.7 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**.

32.8 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**

32.8.1 Static Method Reference

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

is equivalent to:

```
T:call
```

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

32.8.2 Instance Method Reference

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

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

```
t::call
```

32.8.3 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.

32.8.4 Referencing a Constructor

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

can be replaced by:

```
T::new
```

32.8.5 Applications

Instantiate an Array

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

32.9 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.

Part VI

Aggregate Operations (streams)

Chapter 33

Motivation

- reduces boilerplate code
- adds parallelism

Chapter 34

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()**.

Chapter 35

Stream methods

There are two types of stream methods:

- lazy, they **return another stream**.
  - .filter(a $\rightarrow$ b), b $\subseteq$ a
  - .map(element $\rightarrow$ mapFunc());
  - .flatMap(ListOfLists $\rightarrow$ concatenated_lists)
  - .sorted()
  - .findFirst()
- eager, they **return a value or void**
  - .collect(Collectors.toList(), ..ToSet(), see: "Collector" in "Support Classes, Interfaces and methods"
  - .count
  - .max(), .min()
  - .reduce(initVal, *reducer*)
    - **BinaryOperator<T, V>reducer = (acc, item) $\rightarrow$ func(acc, item)**
  - .isPresent() - returns **true** or **false**
  - .ifPresent() - performs some action when a value is present

35.1 min(), max()

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

35.2 .reduce()

It comes in three oveloaded version.

35.2.1 .reduce(accumulator)

Accumulator is of BinaryOperator<T, T>type. It is like a BiFuncion (but both parameters are of the same type) - it takes two objects as parameters and returns single object resulting from some operation performed on input objects. Example:

```
.reduce((Person p1, Person p2) -> p1.getAge() > p2.getAge() ? p1 : p2)
.ifPresent(System.out::println);
```

...prints the oldest person.

35.2.2 .reduce(identity, accumulator)

Identity is an initial value used by the accumulator. Example:

```
.reduce(new Person(), (Person p1, Person p2) -> p1.getAge()  += p2.getAge());
.ifPresent(System.out::println);
```

...prints the person whose age is a sum of all ages.

35.2.3 .reduce(identity, accumulator, combiner)

This version is intended for parallel operations. The combiner is a BinaryOperation used to merge partial values returned by accumulator function executed in paralel. Example:

```
.reducent(0, (Person p1, Person p2) -> totalAge = p1.getAge()  + p2.getAge(),
(a, b)) -> a + b);
.ifPresent(System.out::println);
```

...prints a sum of all ages.

35.3 .collect(Collector..)

.collect(Collector....) accepts a Collector which consists of four different operations:

- a supplier
- an accumulator
- a combiner
- a finisher

There are three general types of Collectors:

- reducers/summarisers, they return a single value
- grouping collectors
- partitioners

.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));
```

.collect(Collection.toMap(..)) can throw an Exception if the provided key is not unique. In order to overcome it we can pass a merge function as an additional parameter to .toMap() function:

```
Map<Integer, String> map = persons
.stream()
.collect(Collectors.toMap(
p -> p.age,
p -> p.name,
(name1, name2) -> name1 + ";" + name2));

System.out.println(map);

// {18=Max, 23=Peter;Pamela, 12=David}
```

Can be used to get a single value, too:

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

35.3.1 Partitoning, Grouping

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

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

```
.collect(Collectors.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.

> The classification function maps elements to some key type K. The collector produces a Map¡K, List¡T¿¿ whose keys are the values resulting from applying the classification function to the input elements, and whose corresponding values are Lists containing the input elements which map to the associated key under the classification function.

The classifier returns a Map. So, it is a good idea to store the result in some Map variable for further processing!

35.3.2 Summarizing

- .collect(Collection.averagingInt(..)
- .collect(Colletion.summarizingIInt...))

35.3.3 Custom collector

We need to instantiate and use an object of Collector interface type - see:"Aggregate Operations"."Support Classes, Interfaces and methods"."Collector Interface"

35.4 Sorting

35.4.1 .sorted()

This method uses natural ordering.

35.4.2 .sorted(Comparator.comparing(a -¿ b))

Useful to sort reference types or to define custom ordering:

```
Person[] people;
Stream.of(people)
          .sorted(Comparator.comparing(p -> p.getAge()))
```

This sorts people according to their age.

Chapter 36

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

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

Chapter 37

Streamming Primitive Types

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

37.1 Key Ponts

- they used specialised lambdas:
  - IntFunction instead of Function
  - LongPredicate instead of Predicate
  - asf
- support additional methods, like
  - sum()
  - average()

37.2 Conversion to Primitive Stream and back

- to primitive stream:
  - mapToInt()
  - mapToLong()
  - mapToDouble()
- convertion from primitive type:
  - mapToObj()
  - .boxed() simply converts from IntStream to Stream<Integer>

37.3 .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()

Chapter 38

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!

38.1 String Representation of Collections

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

Chapter 39

Streaming Arrays

```
Stream.of(array)   //  converts an array to a stream
         .....
         .toArray(T[]::new);      // converts the stream back to an array
```

Chapter 40

Support Classes, Interfaces and Methods

40.1 Collector interface

```
Collector<T, A, R>
```

T - the type of input elements to the reduction operation
A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)
R - the result type of the reduction operation

In order to create new object of Collector class we need to supply four argument to the method Collector.of(...):

- a supplier
- an accumulator
- a combiner
- a finisher
- a characteristic (optional)

40.1.1 supplier

This parameter *supplies* **a container**

```
() -> container
```

A container to store accumulated values. For instance:

- an array

- collection
- builder (StringBuilder, StringJoiner)

We need to assign an initial value to it - new int[1], new StringJoiner(" — ").

40.1.2 accumulator

it is a lambda of **Consumer** type, which provides a function which defines the way to combine the values together. it returns nothing, just updates the state of the container:

```
(container, element) -> container.add(element);

or:

(container, element) -> container[0] += element;
```

40.1.3 combiner

The purpose of this parameter is to supply a function which allows to merge to containers. It supports **parallel execution** of the collector:

```
(container1, container2) -> container1.merge(container2);

or:

(container1, container2) -> {
container1[0] += container2[0];
return container1;
}
```

40.1.4 finisher

We use this parameter to define required output type returned by our collector - String, int. Colloquially - 'to take out the value from the container:

```
StringJoiner::toString

or

container -> container[0]
```

40.1.5 characteristic

Three types:

- CONCURRENT — Indicating that one result container can be used by multiple concurrent accumulators.

- IDENTITY_FINISH — Indicates that the finisher function is the identity function and then can be omitted.
- UNORDERED — Indicates that the collector doesn't depend on the ordering of the elements.

40.2 IntStream

40.2.1 .of(ints)

40.2.2 .range(lb, up)

Generates a stream of integers lb..(up-1). Similar to loops.

40.2.3 .rangeClosed(lb, up)

Generates a stream of integers lb..up. Similar to loops.

40.2.4 .iterator(lb, next).limit(n)

lb - value of first index next - lambda defining how to calculate consecutive index, fi:

```
i -> i + 2
```

returns every second value(all odds or all even).
.limit(n) defines a size of the stream. .iterator creates infinite stream.

40.2.5 .generate(intSupplier s)

This method generates custom integer stream. Example applications:

- stream of constances
- stream of random values

It takes an object of functional interface **intSupplier** type as a parameter . The signature of lthe ambda expression is:

```
() -> int;
```

The expression provides a definition of returned integer stream. Example:

```
int[] i = IntStream.generate({} -> (int)(Math.random() * 52 +1))
                                .distinct
                                .limit(13)
                                .sorted()
                                .toArray()
```

returns 13 random and distinct numbers in the range 1..52 sorted in the ascending order.

40.2.6 .boxed()

Converts back to Stream<Integer.

```
Stream<Integer) stream = IntStream.range(0, 21).boxed();
```

40.3 Stream

We can use an object of this class to pass stream over our code - for instance as an argument to call a method or as a return type. Tha class also contains useful static methods.

40.3.1 .of(objects)

'objects' can be a structure or literal (literal list of some objects). They can be of different type, they can replace them.

Chapter 41

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.

Part VII

Library Classes

Chapter 42

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();
```

42.1 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!**

42.2 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().

42.3 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).

42.4 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 43

Integer

43.1 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));
  ```

43.2 Unboxing

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

43.3 Conversion from String

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

Chapter 44

Date & Time

44.1 Date

Textual output:

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

returns format

May 04 09:51:52 CDT 2009

44.2 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);
    ```

44.3 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);
  ```

Chapter 45

String

45.1 CharSequence Interface Family

CharSequence interface family:

- String
- CharBuffer
- StringBuffer
- StringBuilder

45.2 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);
  ```

45.3 split()

```
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**. **If input string starts with the delimiter then first returned item is an empty string!**

Chapter 46

java.utils.Arrays

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

46.1 ArrayList from an Array

46.1.1 Sending Arrays.asList(array) to ArrayList constructor

```
ArrayList<Element> arrList = new ArrayList<>(Arrays.asList(array))
```

46.1.2 Using collections.addAll(arrayList, array).

```
ArrayList<T> arrList = new ArrayList<>();
Collections.addAll(arrList, array);
```

This is the best approach performance-wise.

46.1.3 Arrays.asList(array)

This is wrong:

```
ArrayList<Element> arrList = Arrays.asList(array)
```

arrList is backed by a fixed size array - adding or removing will cause to throw **UnsupportedOperationException**.

Chapter 47

Random

47.1 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)
```

47.2 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);
```

47.3 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**.

47.4 Comparable

Contains

`compareTo()`

which is used by:

- Collections.sort()

Part VIII

Advanced Topics

47.5 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

47.6 Immutability

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

47.6.1 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

47.7 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 48

Data Parallelism

48.1 Intro

48.1.1 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.

48.2 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.

48.3 Parallel Stream Operations

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

48.4 Caveats

The use of locks can lead to nondeterministic deadlocks!

48.4.1 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.*

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?`

g

Part IX

Idioms

Chapter 49

Exceptions

sectionException-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 50

Dependencies

50.1 Overview

Dependency means that some piece of code relies on another one. If an object of type A uses an object of type B, then the object A is called *dependant*, and the object B is called *dependency*. The classes A and B are **coupled**.

50.2 Dependency as a Type of Relationship

A dependencies is the weakest type of relationship (strongest to weakest):

- inheritance (is-a)
- composition (has-a, the same life-span)
- aggregation (as-a, independent life-span)
- association (objects works together)
- dependency(brief use of another object)

50.3 Types of dependencies

- class dependencies
- interface dependecies
- method/field dependecies (usually used by reflection)

Complicated dependency interfaces means tighter coupling. Therefore it is recommended to avoid adding new method to existing interfaces. it is better to add a new component with the required new functionality.

50.4 Hidden Dependency

Visible dependency is the dependency exposed in the API, for instance when we use dependecy as a method parameter. Hidden dependency can be a dependency used within a method body. **Hidden dependency can bad!**, because can be discovered by code inspection only.

50.5 Indirect Dependencies

Transitive dependencies, chained dependencies. A depends on B, B depends on C. A indirect depends on C.

50.6 Local and Context Depedences

General purpose component is a component which may be used byother applications, too. The classes which belong to this component are *local.*

Application specific components are not of any use outside the application. The classe belonging to the component are called *context*

Context dependencies means that general purpose classes depends on application specific classes. **This is bad!** It means that general purpose classes cannot be used outside the application.

Part X

Antipatterns, Code Smells

Chapter 51

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();
```

Chapter 52

Constructors

52.1 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!**

In order to avoid this kind of bug:

- don't call any method from within a constructor
- make a **class final** (non-extendable)
- or make the **method final** (non-overridable)
- or make the **method private** (which makes them automaticaly final)

Part XI

Testing

Chapter 53

JUnit

53.1 Unit testing

It could be a function class, package, subsystem.

Part XII

Varia

Chapter 54

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.

Part XIII

Appendix

Chapter 55

Vocabulary

55.1 B

55.1.1 binding

Binding refers to the process of associating a method **definition** with a method **invocation**

- **early binding, static binding**

  the method definition is associated with the method invocation when the code is compiled

- **late binding, dynamic binding**

  the method invocation is associated with the method invocation when the method is invoked (at run time)

55.2 C

55.2.1 composition

composing a new class from existing classes.

55.2.2 covariant

dependants depend on compatible types:
T_a -> T_b => $A(T_a)$ -> (T_b)

This is not an attribute of types, but only of *dependant types*. Implicit type compatibility implies implicit covariance, explicit compatibility implies explicit covariance. Intuitively - all types are implicitly compatible with Object class, in opposite direction compatibility is possible explicitly only, by the use of **cast**.

55.3 D

55.3.1 depedency

means that some piece of code relies on another one.

55.3.2 dynamic binding

polymorphism related consept, runtime binding, late binding. The specific callee method is determined during a runtime at the very last moment.

55.4 E

55.4.1 encapsulation

It is hidding data within a class and to make it accessible only through methods. Encapsulation hides implementation. I **combines characteristics and behaviours**

- **it is safe to make changes to the implementation** without worrying that existing code can be broken.
- it can preserve internal consistency of the class. For instance can ensure that modifying some data will modify also all related data (**update variables, call methods**)when it is necessary.
- prevent internal (private) utility methods to be called from outside.

55.5 H

55.5.1 has-a relationship

see: composition.

55.6 I

55.6.1 interface

set of all requests that can be done to an object. The type is what determines an interface.

55.6.2 invariant

'not changing'. For instance an attempt to cast invariant type can raise a compile error. See: Arrays.TypeConversion

55.7 L

55.7.1 late binding

polymorphism related consept, dynamic binding, runtime binding. The specific callee method is determined during a runtime at the very last moment.

55.8 M

55.8.1 members of class

Fields and methods

55.9 N

55.9.1 non-interfering

The underlying source data (list, array, values) are not modified.

55.10 P

55.10.1 polymorphism

dynamic binding, late binding, runtime binding. Different behaviour of objects of the same base type.

55.10.2 problem space

the place where the problem atually exists.

55.11 R

55.11.1 RTTI

Run-Time Type Identification

55.11.2 runtime binding

polymorphism related consept, late binding, dynamic binding. The specific callee method is determined during a runtime at the very last moment.

55.12 S

55.12.1 short circuiting

A logical expression is evaluated only until truth or falsehood of the entire expression can be unambiguously determined.

55.12.2 solution space

the place where the problem is modelled, such as computer.

55.12.3 stateful

The state can change and it is maintained during execution.

55.12.4 stateless

The execution of an operation is **deterministic**. It means that **it doesn't depend on any mutable value**.

55.13 U

55.13.1 upcasting

a process of treating a derived type as though it were its base type(class).

Chapter 56

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`

56.1 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`