Java Programming Masterclass for Software Developers

Udemy Course : https://www.udemy.com/java-the-complete-java-developer-course/learn/v4/overview

Grr, he added lectures (and exercises) in parts of the course already finished, changing lecture numbers further on. Lecture numbers don't match any more.

14-01-2019 - ... Go to Bottom

Section	Description	Finished
Section 1	Course Introduction	14-01-2019
Section 2	Setup and First Steps	14-01-2019
Section 3	Variables, Datatypes and Operators	16-01-2019
Section 4	Expressions, Statements, Code blocks, Methods and more	17-01-2019
Section 5	Control Flow Statements	24-01-2019
Section 6	OOP Part 1 - Classes, Constructors and Inheritance	25-01-2019
Section 7	OOP Part 2 - Composition, Encapsulation, and Polymorphism	28-01-2019
Section 8	Arrays, Java inbuilt Lists, Autoboxing and Unboxing	31-01-2019
Section 9	Inner and Abstract Classes & Interfaces	13-05-2019
Section 10	Java Generics	14-05-2019
Section 11	Naming Conventions and Packages, 'static' and 'final' Keywords.	21-05-2019
Section 12	Java Collections	03-06-2019
Section 13	JavaFX	
Section 14	Basic Input & Output including java.util	
Section 15	Concurrency in Java	
Section 16	Lambda Expressions	
Section 17	Regular Expressions	
Section 18	Debugging and Unit Testing	
Section 19	Databases	
Section 20	Java Networking Programming	
Section 21	Java 9 Module System	
Section 22	Migrating Java Projects to Java 9	
Section 23	Archived Videos	
Section 24	Bonus Material	

Section 1, Course Introduction

14-01-2019, finished 14-01-2019 Go to Top Go to Bottom

Section 2, Setup and First Steps

14-01-2019, finished 14-01-2019 Go to Top Go to Bottom

We'll be using JDK11, and IntelliJ IDEA as IDE (Integrated Development Environment).

He says he recorded this video in October 2019 :-)

JDK (Java Development Kit) helps you write and compile Java programs, run programs with JVM (Java Virtual Machine, with JRE (Java Runtime Edition).

JDK is needed to write Java programs.

Which version? Oracle has started to charge for Java. There is something called open(-source) JDK, so you don't need to pay to use Oracle JDK. Oracle JDK is business orientated, they charge businesses. Just use the Oracle version.

Blog Java 11 : https://learnprogramming.academy/programming/java-11-has-arrived-is-it-time-to-panic/

Installing JDK 11 on a Linux Machine

Oracle website : https://www.oracle.com/technetwork/java/index.html

Download JDK from : https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html

```
$ cd /Downloads/Jdk11
$ tar -xvf jdk-11.0.1_linux-x64_bin.tar.gz
$ sudo mkdir -p /usr/lib/jvm/jdk-11
$ sudo mv jdk-11.0.1/* /usr/lib/jvm/jdk-11
$ sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/jdk-11/bin/java"
1010
$ sudo update-alternatives --install "/usr/bin/javac" "javac" "/usr/lib/jvm/jdk-
11/bin/javac" 1010
```

I already had Java8 installed, so had to do some `update-alternatives`-magic to activate Java11.

Installing and Configuring IntelliJ IDEA on a Linux Machine

Jetbrains website : https://jetbrains.com

Download from : https://www.jetbrains.com/idea/download/#section=linux

Open file explorer, go to Downloads/IntelliJ, right click *ideaIC-2018.3.3.tar.gz*, choose *Extract here*.

Move the folder *idea-IC-183.5153.38* anywhere you want it, into */Software* in my case.

Open terminal

```
$ cd /Software/idea-IC-183.5153.38/bin
$ ./idea.sh &
```

This starts IntelliJ IDEA installation wizard

- first window press OK
- then Accept
- then Usage Statistics

- then choose Theme
- then check *Create a desktop entry*
- then check *Create a script for opening files and projects* from the command line, leave *path* as is
- then leave Default plugins as is
- then choose Featured plugins, I chose *Scala* and *IDE Features trainer*
- then enter system password
- IntelliJ IDEA starts.

Tim gives lots of configuration tips, click *Configure* in bottom Welcome screen.

Project Defaults - Project Structure

- Project SDK : 11
- Project language level : 11

Settings - Editor - General - Auto Import

- check *Add unambiguous ...* and *Optimize imports ...*

Settings - Editor - General - Code folding

- uncheck *imports* , *One-line methods*, *Closures* and *Generic constructors ...*

Convention is Java class names start with capital.

Created first HelloWorld Java class, it runs in IntelliJ and in terminal in ./out/production/S02-HelloWorld

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

`$ java Hello` returns : `Hello World!`

`$ java Hello.class` returns an error! It will try to run a function wrapped in Hello (?)

`main(String[] args)` means that any given arguments are strings,. When you run `$ java myProgram multiply 2 3`, `args[0] == "multiply"`, `args[1] == "2"` and `args[2] == "3"`.

Section 3, Variables, Datatypes and Operators

15-01-2019, finished 16-01-2019 Go to Top Go to Bottom

IntelliJ has shortcuts to templates, if you type `sout + <Tab>`, it expands to `System.out.println()`

Variables (Lecture 16)

Variable names are case sensitive!

When declaring a variable for the first time, you start with the data type. When reassigning a value, the data type does not need to be given again.

There are 8 primitive data types (byte, short, int, long, float, double, char, boolean), `char` is not same as `String`, there is no primitive data type for a string.

`String` is a Java class with special support. Normally you would have to write `String myString = new String("myString")`, but `new String()` can be left out.

Oracle says :

> `String` objects are *immutable*, which means that once created, their values cannot be changed.

but also says :

> **Note:** The `String` class is immutable, so that once it is created a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

so reassigning a value is no problem.

Declare a variabale : `<datatype> <varName> = <value> ;`

Some types have extra character after value to indicate type

- `L`, `long`, `long myLong = 0L;`
- `f`, `float`, `float myFloat = 0f;`
- `d`, `double`, `double myDouble = 0d;`

Declare multiple variables of same type: `int one = 1, two = 2;`

Declare multiple variables of different types in one statement is not possible.

Reassigning a value : `<varName> = <anotherValue> ;`

- char myChar = "a" ; myChar = "b";
- int myInteger = 5 + 7 - (3 * 2) ; myInteger = 10;
- String myString = "abcdefg" ; myString = "hijklmnop";
- boolean myBoolean = false ; myBoolean = true ;
- double myDouble = 3.1415d ; myDouble = (double) (10 / 3) ;

Primitive Data Types - The Byte, Short, Int and Long (Lecture 17)

For integers, you would usually use `int` as data type (32-bit). Because of memory concerns or if `int` is not big enough, you can also use other primitive types.

- `byte`, default `0`, value -2^8 to $2^7 - 1$, -128 to 127 (inclusive)
- `short`, default `0`, value -2^{16} to $2^{16} - 1$, -32,768 to 32,777 (inclusive)
- `int`, default `0`, value -2^{31} to $2^{31} - 1$, -2,147,483,648 to 2,147,483,647 (inclusive)
- `long`, default `0L`, value -2^{63} to $2^{63} - 1$, -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (inclusive)
- long is assigned a value with l or L behind number, use L for readability. `long myLong = 100L;`

In Java, you can use underscores as seperators to make large numbers better readable. Seperator can not be placed at beginning, or end, or before or after a decimal point.

```
int  myLargeNumber  = 2_147_483_647 ;
int  mySocialNumber = 1234_56_789 ;
long myCreditCardNr = 1234_5678_1234_5678L ;
```

Value wrapping : If you give an integer the maximum value and add 1, the result is its min value. And vice versa with subtracting 1 from minimum value. So if an integer exceeds the boundaries, it will wrap to the opposite.

Casting: Java converts results of calculations with all integers to `int`, to make sure the result fits in `byte` and `short` use `(datatype)` just before calculation :

```
byte myByteValue = 100 ;
// No problem, int goes in to int
int myIntvalue = myByteValue / 2 ;
// int maybe does not fit into byte, cast with (byte),
//  tells Java to cast result to type byte
// When casting, put expressions between parentheses
byte myByteCalc = (byte) (myByteValue / 2) ;
short myShortCalc = (short) (100000 + 50 * (myByteValue / 2)) ;
// Not needed for long, because int fits easily into long
long  myLongCalc = 100000L + 50L * (myByteValue / 2) ;
```

That's also a reason why just using `int` is easier.

Primitive Data Types - The Float and Double (Lecture 18)

Two data types to deal with numbers that decimals, `float` and `double`. `double` has double the precision that `float` has.

- `float`, 32-bit precision floating point, default `0f`
- `double`, 64-bit precision floating point, default `0d`, takes up twice the space in memory

`float` is less precise than `double`, the conversion cannot be performed implicitly.

like `long`, `float` and `double` have a symbol behind the number when assigning a literal.

When using a decimal point in a number or calculation, Java treats the number or result as `double`.

```
int    myIntValue     = 5;
float  myFloatValue1  = 5f;
float  myFloatValue2  = 5.25f;
float  myFloatValue3  = (float) (5.25); // Decimal point -> double, cast to float
float  myFloatValue4  = (10f / 3f);     // No decimal point -> int, cast to float
double myDoubleValue1 = 5d ;
double myDoubleValue2 = (double) (5)
double myDoubleValue3 = 5.25 ;          // Decimal point -> double
double myDoubleValue4 = (10d / 3d);
double myDoubleValue5 = 10.0d / 3d;     // Decimal point -> double
```

`double` is often faster on newer computers.

`double` is used in internal Math methods (cos(), sin(), ...).

`double` is far more precise.

So, use `double`s :-)

Primitive Data Types - The Char and Boolean (Lecture 19)

- `char`, 16-bit, default `'\u0000'`, 1 character or Unicode code character
- value assigned with single quotes!

```
char myChar1 = 'a';
char myChar2 = '\u00A9'; // (c)
char myChar3 = '\u00AE'; // (tm)
char myChar4 = '\u20AC'; // euro sign
```

- `char` is internally an int!
  - `char c = 'A'; c++; // -> c is now 'B'`
  - `char c = 'A' + 1; // -> c is now 66`
- `boolean`, 1-bit, default `false`, `true` or `false`

Unicode-table:

https://unicode-table.com/en

Unicode is extended (24-bit?), now holds more than 16-bit `char` can hold, so `char` can not represent all Unicode characters.

Strings and Finish up Primitives (Lecture 20)

`String` is not a primitive data type, it is a `class`. But it is so strongly integrated in Java language that it is used like a primitive. You can declare a `String` type variable the same way as primitives.

- all primitives have a corresponding default value
- as String is an object (instance of class String), it's default value is `null`
- value assigned with double quotes!
- char and String are surprisingly incompatible

```
char a = 'a'; char b = 'b';
String ab = a + b;      // -> Error
String ab = "" + a + b; // -> Works fine
```

```
String myString1 = "This is a string";
System.out.println("myString1 : " + myString1);
myString1 += ", it has one or more characters";
System.out.println("myString1 : " + myString1);
// Other primitives concatenated and transformed to String by Java
int myInt = 50
myString2 = "The price is \u20AC " + myInt;
boolean myBoolean = false;
myString3 = "My dog is " + myBoolean;
```

Operators and Operator Precedence in Java (Lecture 21/22)

- value assignment, `=`, `int sum = 1; String string = "";`
- concatenate, `+`, `String result = "res" + "ult";`
    - concatenating String with primitive will cause primitive to be stringified.
    - int + false + String + double will also be converted to String.
- arythmic, `+-*/%`, `%` is modulus (remainder), `int sum = 10; int rem = sum % 4; // rem = 2`
- quick add 1, `++`, `int myIndex = 10; myIndex++; // now 11`
- quick subtract 1, `--`, `int myIndex = 10; myIndex--; // now 9`
- quick concatenate, `+=`, `String result = "res"; result += "ult"; // now "result"`
- quick add/subtract n, `+=`, `-=`, `int myIndex; myIndex += 7; myIndex -= 5; // now 2`
- quick multiply/divide/remainder n, `*=`, `/=`, `%=`
- logical not, `!`, `!(<comparison>)`, `boolean isDone = false; isDone = !isDone; // now true`
- logical and/or, `&&`, `||`
- is equal, is not equal, `==`, `!=` (learned later: to compare object values use `.equals()`)
- (Java does not know is not equal `<>`)
- is greater than/is greater than or equal to, `>`, `>=`
- is less than/less than or equal to, `<`, `<=`
- when comparing expressions, put expressions between parentheses (= round brackets)
- as `=` assigns a value, make sure you use `==` in comparisons
    - `boolean isCar = false; if (isCar = true) ...;`
    - will set `isCar` to `true` and expression evaluates to `true`
    - `int balance = 100000; if (balance = 0) ...;`
    - will set `balance` to `0` and expression evaluates to `true`
- ternary operator, `<expression> ? <value-if-true> : <value-if-false>;`
- precedence, `int result = 50 + 50 * 50;` value is 2550, `int result = (50 + 50) * 50;` value is 5000

Summary of operators : https://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html

Java Operator Precedence Table : http://cs.bilkent.edu.tr/~guvenir/courses/CS101/op_precedence.html

Section 4, Java Tutorial: Expressions, Statements, Code blocks, Methods and more.

16-01-2019, finished 17-01-2019 Go to Top Go to Bottom

Keywords and Expressions (Lecture 24)

Java keywords : https://en.wikipedia.org/wiki/List_of_Java_keywords

> In the Java programming language, a keyword is one of 61 reserved words that have a predefined meaning in the language; because of this, programmers cannot use keywords as names for variables, methods, classes, or as any other identifier.

`const` is reserved, but is not used and has no function, for defining constants use `final`.

`goto` is reserved, but is not used and has no function.

`void`, was wondering about that. It is used to declare that a method does not return any value.

- `double kms = (100 * 1.609344);` : `kms = (100 * 1.609344)` is an expression
- `int miles = 100;` : `miles = 100` is an expression, `100` is a literal
- `String myString = "This is a literal";` : `myString = "This is a literal"` is an expression, `"This is a literal"` is a literal
- `System.out.println("This is an expression");` : `"This is an expression"` is an expression, because it's an argument in a method call.
- `if (score > 99) {...}` : `score > 99` is an expression

Statements, Whitespace and Indentation (Code organisation) (Lecture 25)

`int myInt = 50;` is a statement, `myInt = 50` is an expression, `50` is a literal.

Semi-colon is used to end Java statement.

- you can use multi-line statements and end with a semi-colon
- you can put multiple statements on one line as long as you end each one with a semi-colon.

`if (<expression>); <do something>;` `if` statement will check nothing, it has finalized.

Whitespace is needed between keywords and identifiers. Many times extra whitespace makes code more readable. Unnecessary whitespace is ignored by Java, use as much as you want.

Indentation also helps making code readable, Java does not need it. Do it as much as you like

Code Blocks And The If Then Else Control Statements (Lecture 26/27)

With one-line statements after an if statement, you can just finish with a colon.

```
int  myInt = 500;
if (myInt > 450) System.out.println("Jeez, your Int is high!");
```

When multiple statements follow the if statement, you make a code block with { }.

```
int myInt = 500;
if (myInt > 450) {
  System.out.println("Jeez, your Int is high!" + " I will give you a bonus");
  int myIntNow = myInt + 100;
  System.out.println("Your Int was " myInt + ". It is now " + myIntNow);
}
```

Recommended is to use code blocks always, also when 1 statement after `if` test.

```
if (<logical-expression1>) {
```

```
    // <logical-expression1> evaluates to true
  ; // Statement
} else if (<logical-expression2>) {
    // <logical-expression1> evaluates to false
    // <logical-expression2> evaluates to true
  ; // Statement
} else if (<logical-expression3>) {
    // <logical-expression1/2> evaluate to false
    // <logical-expression3> evaluates to true
  ; // Statement
  ; // Statement
} else {
    // <logical-expression1/2/3> evaluate to false
  ; // Statement
}
```

After one of the code blocks is executed, code jumps to first line after last curly brace (`}`).

Within the code block variables declared outside the code block are accessible.

Variables declared inside code block are not accessible outside the code block, *scope* is limited to code block.

Methods in Java (Lecture 28/29/30)

Repeating code can be put in another method, so it can be called multiple times (DRY = Don't Repeat Yourself). Also for readability it is useful to put blocks of code in another method. Methods need meaningful names, so code blocks they are called from remain readable without too many comments.

Methods can receive arguments

```
public static void myMethod1(boolean myBoolean, int myInt, String myString) {
  // Code goes here
}
```

`public static void`, void means no result send back from method.

In stead of `void`, you can specify what data-type is returned, so you do need `return` in code block.

```
public static boolean higherThan5( int intArg ) { return intArg > 5; }
boolean higher = higherThan4(8);
public static int sumInts(int intOne, int intTwo) { return intOne + intTwo; }
int sum = sumInts(1, 2);
```

According to the teacher, in programming terms, return -1; is conventionally used to indicate an error, and in searching algorithms, -1 indicates `invalid value` or `value not found`. Hmmm, OK.

Procedure as a term can also be used to indicate a method that returns nothing (`void`).

In general, *function* as a term can also be used to indicate a method.

DiffMerge Tool Introduction/Installation/Usage (Lecture 31/32/33)

> DiffMerge is a program that will help you to visually compare and merge files on any operating system.

> It can be very helpful with programming as well.
>
> Some other tools like code repositories use similar merge operations, so this will also help you in the future when you start working with code repositories

It is easy to make a typo when coding, and get stuck. This is where DiffMerge can help.

By downloading the teachers code, and comparing it to own code with DiffMerge, it is easier to find errors/typos. You can compare a single file, or a whole folder.

Website : https://sourcegear.com/diffmerge

Download from : https://sourcegear.com/diffmerge/downloads.php

Does not seem to be really maintained any more, download for *Ubuntu **12.04** LTS*, *copyright site **2017***.

At download location in Terminal :

```
$ sudo dpkg -i diffmerge_4.2.0.*.deb
```

When loading files/folders to compare in DiffMerge, load the troublesome code in the right window, DiffMerge only allows for applying changes Left-to-Right. When starting DiffMerge and choosing folders, DiffMerge remembers the last chosen folders, and has a button *Swap*.

Once code is loaded, and through other program code has changed, DiffMerge will notice and ask whether Reload is desired.

Some files and folders should be excluded from DiffMerge, some can not even be read by DiffMerge (e.g. `.class`)

Start DiffMerge, go to Tools - Options, select Folder Windows - Folder Filters.

- Make sure *Use Filename Filters* is checked, add `*.class` to field under *Use Filename Filters.*
  - and/or (`out` until now has only had `.class` files in it, maybe more later?)
- Make sure *Use Sub-folder Filters* is checked, add `out` to field under *Use Sub-folder Filters*
- Make sure *Use Sub-folder Filters* is checked, add `.idea` to field under *Use Sub-folder Filters*

Coding exercises (Lecture 34)

Udemy feature, enter your code in Udemy exercise, have your solution checked right away with click on a button.

While coding in IntelliJ we will need a *main* method for testing, when copying to exercise, leave this one out.

All exercises went well, result in *JavaPrograms/Section-04/S04-05-Exercises*.

Method Overloading (Lecture 35/36/37/38)

> Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

Using the method with the same name, but with a different number of parameters. Just specifying a method with the same name and a different number of parameters, is overloading the method. Java looks for a method with the given name and the given number of arguments.

```
public static void main(String[] args) {
  myMethod(1, " = one")
  myMethod(1, 1, " = two") }
public static void myMethod(int iOne, String sOne) {
  System.out.println(iOne + iOne) }
public static void myMethod(int iOne, int iTwo, String sOne) {
  System.out.println((iOne + iTwo) + iOne) }
```

Java will check whether given arguments are of the right data type. IntelliJ will also try and give an error indication when clearly wrong.

Unique method signature is method name and number and type of arguments.

Method overloading is commonly used in Java.

- Improves code readability and re-usability
- It's easier to remember one method name instead of multiple names.
- Achieves consistency in method naming.
- Gives programmers the flexibility to call similar methods with different types of data

Constants in class, use `final`. More on this later.

```
public class Main {
  private static final String INVALID_VALUE_MESSAGE = "Invalid Value";
  public static void main(Strings[] args) {
  /* write your code here, or not, whatever */   } } // end class
```

Section 5, Control Flow Statements

17-01-2019, finished 24-01-2019 Go to Top Go to Bottom

CFS's dealt with here :

```
switch ( <value> ) { case <value>: <code> break; default: <code> } //  (break;)
for    ( <exp> )   { <code> }      //  (break; - continue;)
while  ( <exp> )   { <code> }      //  (break; - continue;)
do     { <code> }  while ( <exp> ) //  (break; - continue;)
```

Switch Statement (Lecture 40/41)

Can replace `if () {} else if() {} else if() {} else {}` statement, if same value is compared to specific single values in each expression.

```
switch (<checkValue>) {
  case <value1>:
    // code for checkValue == value1
    break;
  case <value2>:case <value3>:
    //code for checkValue == value2/value3
    break;
```

```
  case <value4>:
  case <value5>:
    //code for checkValue == value4/value5
    break;
  default:
    // code if none of above conditions is met
    break; /* Not really necessary, but recommended to do it anyway*/ } // end switch
```

For Loop (Lecture 42/43/44)

Enables executing a code block (0-n) times.

```
for ( <init> ; boolean <loop-while-expression> ; <increment-by> ) { /* code */ }
// Repeat 10 times
for ( int i = 0; i < 10; i++ ) { /* code */ }
// Increment by 2, repeat 5 times, int = 0, 2, 4, 6, 8
for ( int i = 0; i < 10; i+=2 ) { /* code */ }
for ( int i = 0; i < 10; i++ ) {
  if (i == 3) continue; // jump to end loop, code below not executed
  /* code, not executed if i == 3 */
  if (i == 7) break;    // jump to end loop and exit loop
  /* code, not executed if i == 3 or i == 7 */  } // end for
```

The `<init>` will be incremented with `<increment-by>` each time the loop reaches the end of the code block. Then the loop returns to the top and validates the `<loop-while-expression`. When it evaluates to `false`, the loop is exited.

- `continue;` jumps to the end of the loop, continues loop if `<loop-while-expression>` is still met after incrementing.
- `break;` exits the loop immediately.

Look out for Endless Loops.

There is als `forEach`, will be dealt with later.

While - Do While Loop (Lecture 45/46/47)

In stead of looping a maximum defined number of loops, you might want to loop until a condition is met.

```
// First checks, then executes
while (<loop-while-expression>) { /* code */ }
// First executes (at least once!) then checks
do { /*code */ } while (<loop-while-expression); // !! semi-colon !!
// Apparently often used :
while (true) { if (<loop-end-condition>) break; /* code */ }
```

No initialization or incrementation as `for() {}`-loop has, if needed must be done by code.

Look out for Endless Loops.

- `continue;` jumps to the end of the loop, continues loop if `<loop-while-expression>` is still met.
- `break;` exits the loop immediately.

`do { /* code */ } while (<loop-while-condition>);` **executes at least once**, then checks loop condition!

Parsing Values from a String (Lecture 48)

Data type classes have methods to convert one type into another. Class names start with a capital.

Converting a String into some other data type.

- e.g. from user input from console or user interface.

Convert into primitive data type

- `int number = Integer.parseInt("2019");` -> number == 2019
  - invalid content of argument will throw exception `NumberFormatException:`
- `double number = Double.parseDouble("2018.125");` -> number == 2018.125

Some very difficult exercises, but did them (S05-07-Exercises).

:-(Computer broke down

While upgrading Ubuntu to 18.04 LTS I botched it mid-install. I should not do this in the middle of the night. Took me days to try to fix it. All data salvaged, clean new install over the crashed one, data restored. Lets move on :-)

Aside : running from the shell prompt

Not (yet) in course :

IntelliJ stores a compiled class in `out/production/<ProjectName>`. To run a class from the shell prompt :

- if no package was used : `java <ClassName>`
  - `$ java Helloworld`
- if a package was used : `java package.path.<ClassName>`
  - `$ java com.masterclass.Main`

Reading User Input (Lecture 49/50/51/52)

Class called *Scanner*, simple text scanner, that can parse primitive types and strings.

It uses methods like `.parseInt()` internally. Method `.next()` returns result.

We've been outputting to screen with `System.out`, now will pass `System.in` to the class Scanner, and let and Scanner parse the input from the screen.

```
// Create an instance of class Scanner
Scanner scanner = new Scanner(System.in);
System.out.println(" Enter your name : ");
String name = scanner.nextLine();
scanner.close();
System.out.println("Your name is " + name);
```

`.nextLine()` returns the input as a string

It also has methods to handle other data types :

```
// In stead of
System.out.println(" Enter your year of birth : ");
int yearOfBirth = Integer.parseInt(scanner.nextLine());
// you can
System.out.println(" Enter your month of birth : ");
int monthOfBirth = scanner.nextInt();
scanner.nextLine();
```

! After using `.nextInt()` to retrieve an integer, there is still a *next line character* in the buffer, from pressing `<Enter>` confirming the input. We must clear the buffer by just calling `scanner.nextLine()`.

Entering text when being asked for input through `.nextInt()` will result in an error.

```
boolean hasNextInt = scanner.hasNextInt();
if (hasNextInt) { int number = scanner.nextInt(); scanner.nextLine(); }
```

```
int min = Integer.MIN_VALUE;
int max = Integer.MAX_VALUE;
```

Scanner can only be instantiated once!!

```
package com.masterclass;
import java.util.Scanner;
public class Main {
  public static void main (String[] args) {
    Scanner scanner = new Scanner(System.in);
    calcMinMax(scanner); /* runs fine */
    calcMinMax(scanner); /* runs fine as well */
    scanner.close();  }
  private static void calcMinMax(Scanner scanner) { /* code ... */  }  } // end class
```

Section 6, OOP Part 1 - Classes, Constructors and Inheritance

24-01-2019, finished 25-01-2019 Go to Top Go to Bottom

Classes (Lecture 54/55)

OOP, Object Oriented Programming, Classes, Objects, Constructors, Inheritance.

Real world objects have state and behavior.

Objects have fields(/properties) and methods.

In IntelliJ in project explorer, click on *src*, then right click on *package.name*, choose *New*, choose Java *class*, enter name (first letter capital). Now new `<Name>.java` is created along the original `Main.java`.

```
// Car.java
package com.masterclass;
public class Car {
  // State, encapsulated, not accessible
  private String brand, model; // defaults null, null
  private int doors, wheels;   // defaults 0, 0
  // State, accessible
  public  String description;  // default null

  public void   setModel(String model) { this.model = model; } /* setter */
  public String getModel()             { return this.model;  } /* getter */ } //<-class
```

`public` / `private` are *access modifiers*, what access do we allow others to this class.

- `public class`, unrestricted access.
- `private class`, no other class can access this method/variable.
- `protected class`, allows classes in same *package* access

Encapsulation, only allow object to access its fields `private int doors;`.

Java automatically adds extra functionality to created classes, because the new class automatically is sub-classed from class *Object* and gets all the functionality class *Object* has.

For *encapsulated* fields, you need *setters* to set a value and *getters* to get a value. Public fields can be directly set and gotten.

Setter-methods are a good place to do validation (and manipulate other encapsulated data).

Now the *Car* class is available.

```
// Main.java
package com.masterclass;
public class Main {
  public static void main(String[] args) {
    Car porsche = new Car();
    porsche.setModel("Carrera");              /* needs a 'setter'    */
    String myModel = porsche.getModel();      /* needs a 'getter'    */
    porsche.description = " This a nice car"; /* directly accessible */
    String myDescription = porsche.description /* directly accessible */ } } //<- class
```

Constructors (Lecture 56/57)

With a constructor you can set fields immediately when creating the object or initialize fields with default values.

Constructor method is `public`, does not need a return data type/`void` or `static` and has exactly the same name as the class, case-sensitive.

Constructor methods can also be *overloaded*, and can call other constructors through `this()`.

Calling another constructor **must** be the first statement in the constructor code block.

```
// Main.java
  // ...
  Car porsche = new Car("Porsche", "911", 4, 3); // "Porsche", "911", 4, 3
  Car lincoln = new Car("Lincoln", "Town Car");  // "Lincoln", "Town Car", 4, 4
  Car other   = new Car()                        // "Brand", "Model", 4, 4
```

```
// Car.java
  // ...
  private String brand, model; // defaults null, null
  private int doors, wheels;   // defaults 0, 0
  public Car() { /* Empty constructor called*/
    this(4, 4); } /* calls constructor with params */
  public Car(int doors, int wheels) {
    this("Brand", "Model", doors, wheels); }
  public Car(String brand, String model, int doors, int wheels) {
    this.brand = brand; this.model  = model;
    this.doors = doors; this.wheels = wheels; }
```

In the constructors the setter methods `setValue(value);` can be used. There is discussion in the community about what should be used. There are scenario's where setter methods do not work. By setting the value directly using `this.value = value;`, that will not be a problem.

Inheritance (Lecture 58/59)

Objects often share common characteristics (state and/or behavior). By creating an umbrella class for them you only have to code for the common characteristics once, and then code for the characteristics of the specific object.

To use the constructor of the parent class in the sub class, we use `super()`.

> If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`.
>
> Invocation of a superclass constructor must be the first line in the subclass constructor.

If you want to execute a method in the superclass, you use `super.methodName()`. Both methods can **not** be `static`.

```
public class Person {
  private String name; private int age; private String occupation;
  public Person(String name, int age, String occupation) {
    this.name = name; this.age = age; this.occupation = occupation; }
  public void printData() {
    System.out.println(occupation + " " + name + " is " + age + " years old."); }
} // end class
public class Student extends Person {
  private String studies;
  public Student(String name, int age, String studies) {
```

```
    super(name, age, "Student");   /* sets name, age, occupation */
    this.studies = studies; }
  public void printData() {
    super.printData();             /* prints occupation, name, age */
    System.out.println(name + " studies " + studies) }
} // end class
public class Teacher extends Person {
  private String teaches;
  public Teacher(String name, int age, String teaches) {
    super(name, age, "Teacher");   /* sets name, age, occupation */
    this.teaches = teaches; }
  public void printData() {
    super.printData();     /* prints occupation, name, age */
    System.out.println(name + " teaches " + teaches) }
  public void printDataReverse() {
    System.out.println(name + " teaches " + teaches)
    super.printData();     /* prints name, age, occupation */  }
} // end class
```

You can call any method in the superclass with `super.`. If a method is defined in the superclass, but not in the subclass, you don't have to use `super.`.

And if you would like to alter the functionality, you can just create a method with the same name in the subclass and that will automatically be called in stead of the method in the superclass (where ofcourse you can call `super.methodName()`).

Reference vs Object vs Instance vs Class (Lecture 60)

A *class* is a blueprint, using this blueprint we can make as many *objects* as we want based on that *class*.

An *object* is built based on a *class* (*instantiated* using the `new` operator), also known as an *instance* of that *class*.

Each object has a location (in memory), this is known as a *reference*.

References can be copied, they will still point to the same *object*.

References can be passed as *parameters* to *constructors* and *methods*.

```
  Object myObj1 = new Object();
  Object myObj2 = myObj1;  // myObj1 and myObj2 reference to the same object
```

this vs super (Lecture 60)

Keyword `super` is used to access/call *parent* class members (variables/fields and methods).

In a *constructor* `super()` is called to call the superclass *constructor*, it **must** the first statement.

Anywhere else the syntax is `super.someField` / `super.someMethod()`

Keyword `this` is used to access/call *current* class' members. `this` is required when we have a parameter with the same name as an instance variable (field).

```
public void add2X (int x) { this.x += x; }
```

! Both can be used anywhere in a class, **except** `static` areas (the static block or a static method). More on `static` later.

`this` commonly used in *constructors* and *setters*, optionally in *getters*.

`this()` is used to call a *constructor* from another overloaded *constructor* in the same *class*, can only be used in a *constructor* and **must** be the first statement.

The Java compiler puts a default call to `super()` (no arguments) if we don't add it ourselves.

Even *abstract classes* have *constructors*, but can not be *instantiated* using the `new` keyword.

An *abstract class* is still a *super class*, so its *constructors* run when someone makes an *instance* of a *concrete subclass*.

! A *constructor* can have a call to `super()` or `this()`, but never both.

Constructor chaining, the last constructor has the *responsibility* to initialize the fields. No matter with how many arguments we *instantiate* the *class*, the last *constructor* will do the initialization.

```
class  ClassName {
  private int a, b;
  public ClassName()             { this(0); }
  public ClassName(int a)        { this(a, 0): }
  public ClassName(int a, int b) { this.a = a; this.b = b; } }
```

```
class Shape {
  private int x, y;
  public Shape(int x, int y) { this.x = x; this.y = y; } } // end class
class Rectangle {
  private int width, height;
  public Rectangle(int width, int height) { this(width, height, 0, 0); }
  public Rectangle(int width, int height, int x, int y) {
    super(x, y);
    this.width = width; this.height = height; } } // end class
```

Method Overloading vs Overriding Recap (Lecture 61)

Method Overloading

- providing multiple methods with same name but different arguments (type/count), .
- return type may be different, allows for reuse of the same method name
- very handy, reduces duplicated code, no need to remember different method names
- nothing to do with *polymorphism*, but often referred to as *Compile Time Polymorphism*
- compiler decides which method to be called, based on method name, return type and arguments
- `static` and *instance methods* can be overloaded. More on `static` / *instance method* later
- happens usually inside a single *class*, but a method can also be treated as *overloaded* in the *subclass* of that *class*, because *subclass inherits* one version of the method from the *superclass* and can then have another *overloaded* version of the *method*
- overloaded methods **must**

    - have same name
    - have different arguments (type/count)
- overloading methods **may**
    - have different return types
    - have different access modifiers (`private` / `protected` / `default` / `public`)
    - throw different checked or unchecked exceptions (more on this later)

Method overriding

- by extending the superclass, the subclass gets all the superclass' methods (these are called derived methods)
- defining a method in a subclass that already exists in the superclass, with the same *signature* (name, arguments type/count)
- also known as *Runtime Polymorphism* and *Dynamic Method Dispatch*, because method to be called is decided at runtime by the *JVM* (Java Virtual Machine)
- recommended to put `@Override` immediately above method definition. Annotation read by compiler, will show us if overriding rules are not followed correctly
- `static` methods can **not** be overwritten, only *instance* methods
- overriding methods **must**
    - override an *inherited* method
    - override a method that is **not** a *constructor*, `private` method or `final` method
    - have same or higher *access modifier* (`public` etc)
    - have same *signature* (name, arguments type/count)
    - have same *return type* or *subclass* thereof
- overriding methods **may**
    - use `super.methodName()` to call the *superclass* version of the overridden method

Example of *subclass thereof*, covariant return type :

```
class Burger {} /* ... */ }
class HealthyBurger {} /* ... */ }
class BurgerFactory {
  public Burger createBurger() { return new Burger();  }  } // end class
class HealthyBurgerFactory {
  @Override
  public HealthyBurger createBurger() { return new HealthyBurger();  }  } // end class
```

Static vs Instance methods (Lecture 63)

Static methods

- are declared using a `static` modifier
- can not access *instance methods* and *instance fields* directly
- are usually used for operations that don't require any data from an instance of the class (`this`, current *instance* of the *class*)
- have no access to `this` keyword

Whenever there is a method that does not use *instance fields*, that method should be declared `static`. Example : `main` is `static` method, called by the *JVM* when it starts an application.

`static` methods are invoked as `methodName()` when in the same class, ClassName is inferred by JVM. Otherwise call as `ClassName.methodName()`.

```
class Calculator { public static int sum(int a, int b) { return a + b; } }
public class Main {
  public static void main(String[] args) {
    printResult("4 + 5", Calculator.sum(4,5)); /* short for Main.printResult */ }
  public static void printResult(String arguments, int result) {
    System.out.println(arguments + " = " + result) } } // end class
```

`static` methods do not need an *instance* of it's class to be called.

Instance methods

- belong to an *instance* of a *class* (an *object*), usually created by `new` keyword
- can only be used if that *instance* exists
- can access *instance methods* and *instance fields* directly
- can access `static` methods and `static` fields directly

Does a method use fields or instance methods? Yes -> *instance method*. No -> `static`.

Static variables vs Instance variables (fields) (Lecture 64)

Static variables

- declared by keyword `static`
- also known as *static member variables*
- shared by every *instance* of that *class*
- changed by one *instance* also changes for other *instances*

- not used very often, but can be very useful
- e.g. when reading user input using *Scanner*, we declare scanner as a `static` variable, so `static` methods can access it directly

```
class Person {
  private static String name;  /* static */
  public Person(String name) { Person.name = name; }
  public String getName()    { return name; }  } // end class
public class Main {
  public static void main {
    Person frank = new Person("Frank"); /* frank.getName() => "Frank" */
    Person bob   = new Person("Bob");   /* frank.getName() => "Bob"   */ } } // <-class
```

Instance variables (fields)

- every *instance* has it's own copy of an *instance variable*, so it can have a different value (*state*) for each instance
- represent the *state* of an *instance*

```
class Person {
  private String name;  /* non-static */
  public Person(String name) { this.name = name; }
  public String getName()    { return name; }  } // end class
public class Main {
  public static void main {
    Person frank = new Person("Frank"); /* frank.getName() => "Frank" */
    Person bob   = new Person("Bob");   /* frank.getName() => "Frank" */ } } // <-class
```

Section 7, OOP Part 2 - Composition, Encapsulation, and Polymorphism

28-01-2019, finished 28-01-2019 Go to Top Go to Bottom

Composition Part 1 (Lecture 68/69)

Inheritance deals with a *Is-a relationship*.

```
public class Car extends Vehicle { /* Car IS a Vehicle */ }
```

Composition deals with a *Has-a relationship*.

```
public class Engine {
  private int cylinderCount; }
public class Car extends Vehicle { /* Car IS a Vehicle */
  private Engine engine;           /* Car HAS an Engine */ }
```

`private` / `public`

If you declare the *engine* private, you will have to access it with a getter

- `myCar.getEngine().capacity`

If you declare the *engine* public, you can access it as a field

- `myCar.engine.capacity`

Encapsulation (Lecture 70/71)

Mechanism that allows for restricting external access to certain components in the objects.

> **Encapsulation:**
>
> The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access a private data member (variable/method) of another class. However if we setup public getter and setter methods to read and set/update, then an outside class can access those private data fields via public methods. This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That's why encapsulation is known as **data hiding.**
>
> **Advantages of encapsulation:**

> 1. It improves maintainability and flexibility and re-usability: implementation code of *setters* and *getters* can be changed at any point in time. Since the implementation is purely hidden for outside classes they would still be accessing private fields using the same public methods. Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the re-usability of the underlying class.
> 2. The fields can be made read-only (If we don't define setter methods in the class) or write-only (If we don't define the getter methods in the class). For e.g. If we have a field which doesn't need to change at any cost then we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.
> 3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call `set method` and to read a field call `get method` but what these set and get methods are doing is purely hidden from them.
>
> Encapsulation is also known as "**data Hiding**".
>
> 1. Objects encapsulate data and implementation details. To the outside world, an object is a black box that exhibits a certain behavior.
> 2. The behavior of this object is what which is useful for the external world or other objects.
> 3. An object exposes its behavior by means of public methods or functions.
> 4. The set of functions an object exposes to other objects or external world acts as the interface of the object.

```
public class Player { public String name; public int health; }
Player player = new Player;
player.name = "Frank";  // fields directly accessible, .health is not set
```

- If code uses directly accessible fields, it makes it too easy to manipulate those values.
- And if the class definition changes (e.g. name -> fullName), any code directly setting that value must be updated.
- Possibility to forget to initialize fields.

By declaring the fields `private` it is impossible to access them directly, use *constructors* and *setters* and *getters*.

```
public class Player {
  private String name; private int health;
  public Player(name, health) { this.name = name; this.health = health; }
  public getName() { return this.name; }
  public setName(String name) { this.name = name; } } // end class
```

If a field name is changed, the *get* method name can stay the same, so calling code does not have to be changed.

We only have to make changes in the class where we refer to `this.<oldFieldName>` / `<oldFieldName>`

```
public class Player {
  private String fullName; private int health;
  public Player(name, health) { this.fullName = name; this.health = health; }
  public String getName() { return this.fullName; }
  public String setName(String name) { this.fullName = name; } } // end class
```

Internally we could keep the original *get* method and make it point to the right *get* method.

```
public String getName() { return getFullName(); }
public String getFullName { return this.fullName; }
```

Polymorphism (Lecture 72/73)

> **Polymorphism:**
>
> Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
>
> Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.
>
> It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
>
> The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
>
> A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

> Teacher : Mechanism that allows actions to act differently based on the actual object that the action is being performed on.

```
class SuperClass                  { public String myCur() { return "$"; } }
class SubClass1 extends SuperClass { public String myCur() { return "€"; } }
class SubClass2 extends SuperClass { /* no myCur() */ }
SuperClass class;
class = new SuperClass(); class.myCur(); /* from SuperClass -> "$"  */
class = new SubClass1() ; class.myCur(); /* from SubClass1  -> "€"  */
class = new SubClass2() ; class.myCur(); /* from SuperClass -> "$"  */
/* You can define the return type of a function as the type of the superclass
   and return a subclass of it ( IS-A relationship ) */
public SuperClass getAClass() { return new SubClass1(); }
class = getAClass()     ; class.myCur(); /* from SubClass1  -> "€"  */
```

> Teacher: It will automatically, if you're inheriting from another class, and you've got a method, and you overwrite that method, that's what polymorphism is, it's giving unique functionality, for the class that has inherited from a base class. It is incredibly usefull, and really enables writing quite generic code.

Huh?

Overriding is a type of polymorphism was mentioned in an answer somewhere.

It is not explained very well, or rather, I don't yet understand it very well. But lots of knowledge on the internet to be found.

- https://www.tutorialspoint.com/java/java_polymorphism.htm
- https://www.quora.com/What-are-encapsulation-inheritance-polymorphism-and-abstraction
- https://www.javatpoint.com/java-oops-concepts

Final OOP Master Challenge (Lecture 74/75)

Challenge to use all to principles of OOP learned so far.

Polymorphism:

```
class Burger { /* ... */ }
class DeluxeBurger extends Burger { /* ... */ }
Burger burger = DeluxeBurger();
/* Now, to access a method in DeluxeBurger : */
((DeluxeBurger) burger).<methodName>();
```

It is ofcourse like good old *casting*. We *cast* the *object* `burger` to the *class* `DeluxeBurger`.

Section 8, Arrays, Java inbuilt Lists, Autoboxing and Unboxing

29-01-2019, finished 31-01-2019 Go to Top Go to Bottom

Arrays (Lecture 76/77/78)

Arrays can hold multiple values of the same type. Arrays are *zero indexed*, start at 0.

```
/* Initialize integer array with 10 elements */
int[] myInts;  myInts = new int[10];
double[] myDoubles = new double[10];
/* Elements are initialized with there default values, null for String objects!
   myInts[0] == 0, myDoubles[1] == 0.0d */
/* Assign a value */
myInts[5] = (5 + 1); /*  arrays start at [0], 5 is in position 6 */
/* Assign values with loop */
for ( int i = 0; i < 10; i++ ) { myInts[i] = i * 100; }
/* or better : use Array.length */
for ( int i = 0; i < myInts.length; i++ ) { myInts[i] = (i + 1) * 50; }
/* Method overloading, argument type determines which method is called */
public static void printArray(int[] array)
  { for (int i = 0; i < array.length; i++) { /* ... */ } } // end method
public static void printArray(String[] array)
  { for (int i = 0; i < array.length; i++) { /* ... */ } } // end method
// Alternative loop if you just want to use the values, not change them
// Called For Each loop, but 'Each' not used
public static void printArray(String[] array)
  { for ( String s : array ) { /* ... */ } } // end method
```

An array can be initialized by using an array initializer block `{ , , }`, also known as an anonymous array.

```
/* Initialize array and assign multiple values in 1 go,
   only allowed when initializing, length will be nr of elements provided */
String[] myStrings = new String[] { "A", "B", "C", "D", "E" };
```

Aside : when dividing 2 integers, make sure at least one of the actors is cast to `double`. Which one does not matter, both is also fine.

```
public static double averageArray(int[] array) {
  return sumArray(array) / (double) array.length ;  } // end method
```

3 methods to clone an array , all as fast as the other. `clone()` needs no argument, but always does full copy :

```
int[] copy = array.clone();
int[] copy = Array.copyOf(array);
int[] copy = System.arrayCopy(array);
```

Reference Types vs Value types (Lecture 79)

Primitive data types are Value Types, they hold a value.

Array and String are Reference Types, the reference to a place in memory.

```
int a = 0; int b = a; a = 1; // -> a == 1, b == 0
```

Though Strings are objects, they behave the same as primitive data types

```
String a = "a"; String b = a; a = "z"; // a == "z", b == "a"
```

Arrays are Reference Type objects, assigning an array the reference of another array, they both point to the same place in memory

```
int[] myInts1 = new int[] {1,2}; int myInts2[] = myInts1; myInts1[0] = 9;
  // ints1[0] == 9, ints2[0] == 9
```

`new int[]`, `array.clone()`, `Arrays.copyOf(array)`, `System.arrayCopy(array)` create new references.

`Arrays.toString( array )` concatenates content of array into a `String`, separated by a comma and space.

When an array is passed into a method, a new temporary reference is made in memory. Reinitializing the array in the method will have no effect on the original array.

```
int[] myInts1 = new int[2]; // myInts1[0] == 0
add1To0(myInts);            // myInts1[0] == 1
public static void add1To0(int[] array) {
  /* All references are updated */
  array[0]++;
  /* Temporary argument-reference is reinitialized to new reference,
      no change to original */
  array = new int[] {5,5,5,5};  } // end method
```

List and ArrayList (Lecture 82-88)

Resize an array

```
private static int[] myInts = new int[5];
resizeArray(10)
private static void resizeArray(int newLength) {
  int[] orgInts = myInts;
  myInts = new int[length];
  for (int i = 0; i < orgInts; i++) { myInts[i] = myOrgs[i] }  }
```

Quite a silly example, as you can not pass the array, so array has to be called `myInts`. This is better :

```
private static int[] myInts = new int[5]; // -> [0,0,0,0,0]
myInts = resizeArray(myInts, 10);         // -> [0,0,0,0,0,0,0,0,0,0]
private static int[] resizeArray(int[] array, int newLength) {
  int[] orgInts = array;
  array = new int[newLength];
  for (int i = 0; i < orgInts.length; i++) { array[i] = myOrgs[i]; }
  return array;  } // end for
```

Lists, another way of looking at arrays as an array is a list, a sequence of values/references.

> Oracle : https://docs.oracle.com/javase/8/docs/api/java/util/List.html
>
> public interface `List<E>` (E - the type of elements in this list)
>
> extends `Collection<E>`
>
> An ordered collection (also known as a *sequence*). The user of the List interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Very much like arrays.

`Class ArrayList<ElementType>` is a resizable array. `ElementType` is the data type you want to store, can be a Java *class* or own *class*, not a *primitive datatype*!

It stores *objects*! You can not store *primitive types*!

Wrapper classes can be coded, but Java already provides these for each *primitive datatype*, e.g. `int` -> `Integer`.

`ArrayList<String> myList = new ArrayList<String>();` to initialize a List interface of type String. The `()` calls the constructor of the `class ArrayList`. (The last `<String>` can be omitted these days)

Other way of initializing, *double brace initialization*. Apparently not advisable, creates anonymous classes, memory leaks, world wars. It uses an instance initializer block inside an anonymous class. Or something like that.

> https://memorynotfound.com/initialization-arraylist-one-line-example/
>
> Method 4: Create and initialize an arraylist using anonymous inner class
>
> Using an anonymous inner class with an instance initializer (also known as an "double brace initialization").
>
> ```
> ArrrayList<String> planets = new ArrayList<String>() {{
>  add("Earth");
>  add("Mars");
>  add("Venus");
> }};
> ```

The *class* takes care of the sizing etc all by it self.

- `myList.add( <ObjectType> <object> );` adds an element to the list with given value
- `myList.add( <index>, <ObjectType> <object> );` inserts an element to the list in given position with given value
- `myList.addAll( <otherListOfSameType> )` to copy all elements from one list into another
- `myList.get( <index> )` to retrieve the value
- `myList.set( <index>, <newValue> )` to modify a value
- `myList.remove( <index> )` to remove an element
- `myList.size()` for length
- `myList.isEmpty()`, for ... :-)
- `myList.indexOf( <object> )` to return `<index>` of `<object>` in myList, returns `-1` if not found
- and more ...

`ArrayList<String> myList = new ArrayList<String>( <otherListOfSameType> );` will initialize a List interface of type string and initialize the contents with the contents of `<otherListOfSameType>`

List to array :

```
// Assuming class for myList object has a getter getList()
String[] myArray = new String[myList.size()];
myArray = myList.getList().toArray(myArray);
```

When you compare values between objects, always use `<object>.equals()` / `!<object>.equals()` in stead of `==` / `!=`

Overloading again is very useful.

```
public Contact findContact(Contact contact) { /* Can use .indexOf() */ }
public Contact findContact(String name) { /* More elaborate search loop, .equals() */ }
```

Once the element type is established, no need to mention it again when initializing it

```
ArrayList<Contact> contactList, contactList2;
contactList  = new ArrayList<Contact>(); /* Contact is not needed */
contactList2 = new ArrayList<>();
```

Alternative array loop

```
ArrayList<Contact> contactList = new ArrayList<>();
// .. add some contacts
for ( Contact contact : contactList) { /* do something with contact */ }
```

I did find that these loops sometimes can not be used if you change something in the List being processing. I am sure that is fully logical, somehow :-) Probably immutability, set up loop and then changing contents is not a smart thing?

Autoboxing and Unboxing (Lecture 89-92)

Classes like

- *Autoboxing*, casting *primitive datatype* to corresponding *datatype class*
- *Unboxing*, casting *datatype class* to corresponding *primitive datatype*
- Java can do some of the Autoboxing and Unboxing for us at compile time for proprietary *primitives* and *datatype classes* at compile time, Java adds appropriate code.
- This way code can be much concise and readable.

```
// No Autoboxing, Unboxing
// deprecated since J9 : Integer myInteger = new Integer(54);
Integer myInteger = 54;  int myPrimitiveInt = myInteger.intValue();
// deprecated since J9 : Double  myDouble  = new Double(3.14);
Double myDouble = 3.14;  double myPrimitiveDouble = myDouble.doubleValue();
ArrayList<Integer> integerAList = new ArrayList<Integer>();
for (int i = 0; i < 10; i++) {  integerAList.add( Integer.valueOf(i) );  }
for (int i = 0; i < integerAList.size(); i++) {
  System.out.println( i + " => " + integerAList.get( i ).intValue() ); }
```

```
// Autoboxing, Unboxing
Integer myInteger = 54;  int myInt = myInteger;
Double myDouble = 3.14;  double myPrimitiveDouble = myDouble;
ArrayList<Integer> integerAList = new ArrayList<Integer>();
for (int i = 0; i < 10; i++) {  integerAList.add( i );  }
for (int i = 0; i < integerAList.size(); i++) {
  System.out.println( i + " => " + integerAList.get( i ) );  } // end for
```

Java compiles declarations at compile time if it can, e.g. if an `int` is assigned to an `Integer` :

```
Integer myInteger = 56;  // Java compiles it to : Integer myInt = Integer.valueOf(56);
Integer myInteger = 5.5; // Can not compile, incompatible types Integer <-> double
```

Very elaborate challenge : banks, branches, customers, transactions. Hardly any autoboxing/unboxing going on. Took me all day :

```
  /JavaPrograms/Section-08/S08-08-AutoboxUnboxChallenge
```

LinkedList (Lecture 93-99)

`Array[]` and `ArrayList<E>` have value/object 1 on position 0, value/object 2 on position 1, etc (*zero-indexed*).

`<LinkedList>` is another type of list, it adds an *address* to the list.

Index	Address	Value
0	100	34
1	104	18
2	108	91
3	112	453

Internally Java calculates this *Address* by `Address = baseAddress + ( Index * 4 )` where `baseAddress = 100`. So for *Index 3* that would evaluate to 112.

It has to do with memory allocation/memory address or something, not so sure whether that is useful info right now, but let's see.

A technical talk about internals, how references point to memory addresses, and how this different for variable length values (Strings, Objects) and how Java handles that internally, and more things we don't have to worry about.

```
Customer frank = new Customer("Frank"), simon = frank;
simon.setName("Simon"); //-> frank.name : "Simon", simon.name : " Simon"
// simon and frank reference the same object
```

```
ArrayList<Integer> iList = new ArrayList<>;
iList.add(1); iList.add(3); iList.add(4);
iList.add(1, 2); // Inserts 2 at position 1
```

When inserting a value in an `ArrayList<E>`, for all elements following from that position downwards Java has to reassign a memory address. For large arrays that can be time/resource consuming.

`LinkedList<>` is an alternative that accommodates this. One element points to the next through links. In stead of inserting a value, it changes the links of the elements, so the element after which it is inserted links to the new element, and the new element links to the element that held that position before it was inserted.

- list = [a, c, d], links = [a->c, c->d]
- insert **b** at (zero-based) position 1
- list = [a, c, d, **b**], links = [a->**b**, c->d, **b**->c], 1 link is updated, 1 link is added
- remove **c** at (zero-based) position 3
- list = [a, d, b], links = [a->b, b->d], 1 link is updated, 1 link is removed.

- so in the actual lists, it is just added at the end, the links define what position they hold.
- I wonder what the costs in time/resource consumption are removing an element in the list would be

To be able to iterate through a LinkedList<> we need an ... Iterator :-) It uses `.hasnext()`, `.next()`

```
private static void printLLString(LinkedList<String> linkedList) {
    Iterator<String> i = linkedList.iterator();
    while (i.hasNext()) {
        System.out.println("Value " + i.next());
    }
    System.out.println("------------");  } // end method
```

Each Iterator has `.hasNext()`, `.next()`, `remove()` `(, .forEachRemaining( ...)`).

When using `.remove()`, issue a `.next()` / `.previous()` afterward to keep iterator on track.

`ListIterator<E>` also has `.hasPrevious()`, `.previous()`, `.nextIndex()`, `.previousIndex()` and more.

`.hasNext()` and `.hasPrevious()` tell us whether there is another element, `.next()` and `.previous()` go there.

It uses the `LinkedList<E>.listIterator()` , not `LinkedList<E>.iterator()`. Example of use :

```
// Add string in alphabetical order in LinkedList<String> linkedList
private static boolean addInOrderLLString(
      LinkedList<String> linkedList, String newString, boolean allowDoubles) {
  ListIterator<String> listIterator = linkedList.listIterator();
  while (listIterator.hasNext()) {
    int comparison = listIterator.next().compareTo(newString);
    if ( comparison == 0 ) {
      // Already exists
      if ( !allowDoubles ) {
        // No doubles
        return false;
      }
      // Double allowed, insert here
      listIterator.add( newString );
      return true;
    }
    if ( comparison > 0  {
      // Should be inserted before this one, hasNext already has us on this element,
      //  so we have to go back one before inserting it.
      listIterator.previous();
      listIterator.add( newString );
      return true;
    }
  }
  // Not added yet, add it now to the end
  listIterator.add( newString );
  return true;
} // end method
```

Java has implemented the LinkedList as a *double link list*, links have pointers to both *next* and *previous* item.

[**a** -> b , a <- **b** -> c, b <- **c** -> d , c <- **d**]

Because of the nature of `LinkedList`s there is no real pointer to where it is (or something, having to do with avoiding infinite loops), anyway, to make `.next()` and `.previous()` work like you would expect, there is a little more work to determine which way we are going. Check `boolean goingforward` and extra `.next()/.previous()` in *Section-08/S08-09-LinkedLists*.

Something like this (check well for `true`, `false` and `!`) :

```
boolean quit = false, goingForward = true;
ListIterator<String> listIterator = collection.listIterator();
while (!quit) {
  String direction = getDirection());
  switch(action) {
    case "quit":
      quit = true;
      break;
    case "next":
      if (!goingForward) {
        if (listIterator.hasNext())     { listIterator.next(); }
        goingForward = true;
      }
      if (listIterator.hasNext())       { /*OK use .next() */ }
        else                            { /* End */ goingForward = false; }
      break;
    case "previous":
      if (goingForward) {
        if (listIterator.hasPrevious()) { listIterator.previous(); }
        goingForward = false;
      }
      if (listIterator.hasPrevious())   { /* OK, use .previous() */ }
        else                            { /*Begin */ goingForward = true; }
      break;
    default:
      break;
} } // end while
```

Alternative loop

```
for (<MyClass> checkedObject: myObjects) {
  if (checkedObject.getField().equals( <someValue> )) { /* do something */ }
    else { /* do domething else */ }  } // end for
```

Section 9, Inner and Abstract Classes & Interfaces

01-02-2019, finished 13-05-2019 Go to Top Go to Bottom

Interfaces (Lecture 100-101)

An *interface* specifies *methods* that a particular *class*, that implements the *interface*, must implement.

The *interface* itself is *abstract*. There is no code for any of the *methods*, you only supply the *method* names and their *arguments*. The code goes into the *class* that implements the *interface*.

The idea is to standardize behavior for all *classes* implementing the same *interface*.

Creating an *interface* is a commitment that the *interface*'s *methods* and their *signatures*, and *constant variables* will not change. This prevents code breaking in other *class*es using this *interface* by changes made to the code.

~~It's good practice to start *Interface* names with a capital I and then the rest also starting with a capital. It makes it clear it is an *interface.*~~ Edit: in Q&A discussions it appears this is an error on the teachers part, probably confused with C#.

In IntelliJ, select *New - Java Class*, in the dialogue *Create New Class* select *Interface*. A new *interface* will be created.

All *methods* with their *signatures* (return type, arguments count/type) are defined, but no code added.

The *access-modifier* keyword is useless here, because the *interface* is implemented in a *class*. In the *class* the *access-modifier* can be specified.

```
package <packagePath>;
public interface MyInterface {
  public void methodOne()); // public is redundant
  String methodTwo(int intValue);
  boolean methodThree(String stringValue));
}
```

In the *class* implementing the *interface*, all of the *methods* from the *interface* have to be implemented and *access-modifiers* can(/must?) be added. Even methods not used or coded for, must be implemented.

```
package <packagePath>;
public class MyClass implements MyInterFace {
  public MyClass () { /* constructor*/ }
  @Override
  public void methodOne()                          { /* not used */ }
  @Override
  public String methodTwo(int intValue)            {
    /* code goes here */
    return intValue + " = " + intValue; }
  @Override
  private boolean methodThree(String stringValue)) {
    /* code goes here */
    return stringValue.equalsIgnoreCase( "masterclass" ); }  } // end class
```

Several ways to *instantiate* an *object*, using a *class* implementing the *interface*, or using the interface *directly*.

Using Interface to declare object also allows using same instance being reused for object of another class implementing the same interface.

```
MyInterface myObject;      // declare myObject as object of type <MyInterface>
myObject = new MyClass();  // create myObject with MyClass implementing <MyInterface>
myObject = new MyClass2(); // create myObject with MyClass2 implementing <MyInterface>
// or ...
MyClass myObject = new MyClass();
```

This is not valid, it does work, but defeats the purpose of *interfaces*.

```
MyInterFace myObject = new MyInterface() {
  public void methodOne()                          { /* ... */ }
  @Override
  public String methodTwo(int intValue)            { return null; }
  @Override
  private boolean methodThree(String stringValue)) { return false; }
}
```

```
MyInterface myObject;
myObject = new MyClass1(); // MyClass1 implements IMyInterface, OK
myObject = new MyClass2(); // MyClass2 implements IMyInterface, OK
myObject = new MyClass3(); // MyClass3 does NOT implement IMyInterface, ERROR!
```

Java libraries make extensive use of *interfaces*, we could change `LinkedList<Class>`, `ArrayList<Class>`, etc into `List<Class>` when defining return type or argument type without any problem, because they implement the `List` *interface*.

Nice find

Just found out you can leave out data type with `ArrayList` and `LinkedList`, and just add any data type to the lists. (`Vector` is another *class* implementing `List` *interface*.)

```
List test = new List() // impossible, List is an Interface
List test = new ArrayList();
test.add( 1 ); test.add( 1L ); test.add( false );
Vector test2 = new Vector();
test2.add( "trueOrFalse" ); test2.add( true );
test.add( test2 );
List<String> test3 = new ArrayList<>(); test3.add( 3 ); // -> Error
```

It can sometimes be hard to decide to implement an *interface* or inherit from a base *class*. The way to decide that generally is to consider the relationship of the final *class* to the *object* it is extending or implementing.

A *class* in Java can only inherit from one super *class*, but you can *implement* from many *interfaces*. Multiple inheritance is only possible by implementing several *interfaces*. An interface can extend multiple other interfaces.

```
public class MyClassB extends MyClassA implements IA, IB { /* ... */ }
public interface IB extends IC, ID { /* ... */ }
```

E.g. a dog (`class Animal`) would implement `interface IWalk`, a bird (`class Animal`) would implement both `interface IWalk` and `interface IFly`. Most fish would implement neither :-)

Interlude

2019-02-01 Took time off course, creating helper classes and creating a game app.

JavaPrograms/Section-09/S09-03-InterfacesChallenge :

Went fully overboard with a challenge, created a fully functioning ConnectFour game, with TDD and fully working interface. Took lot of time, but learned a lot.

Took time off from the course to create some helper Java *classes* (with basic *test-suites*) from snippets of code I've been using in a number of challenges, and to finish a *Connect Flex project* (Tic Tac Toe, Connect Four, Connect Five, Connect Flex (design your own game)), fully API-styled : *classes* return *responses* and don't do any output to the console, that's up to the code using the *classes*.

- If I have my TestMyClass.java next to MyClass.java (`package com.masterclass;`), I can import all of its functionality with `import static com.masterclass.MyClass.*`.
- In my *CMResult.java*, I had to remove the *access-modifier* `class Result`
- If the .java file is called *CMName.java* (`package com.masterclass;`), and the *class* therein is `class Name`, use `import com.masterclass.Name.*`.
- Just found out you can call Java from Javascript! That is something to look into :-)
- https://jxbrowser.support.teamdev.com/support/solutions/articles/9000013062-calling-java-from-javascript

I created several classes and dozens of methods, each class with its own TDD-suite. I kept on improving the classes, game and user interface. The challenge, how it begun, got out of hand and it is now moved to a project I will keep on developing, as the course continues and my knowledge grows.

2019-02-13, time to get back on track with the course.

Interfaces (Lectures 101-103)

It may be a good idea to declare the *object* with the *interface* and instantiate the object with the *class* implementing the *interface*. That way you are more flexible, depending on circumstances, the actual class will be chosen.

```
// definition, list type unknown? You can use an Interface here
List list;
// ...
// instantiation later on, use class implementing interface desired
list = new ArrayList<String>;
```

Piece of interesting code, you need to *cast* the *object* to the *class*, when declaring it with an *interface* and instantiating it with a *class* implementing the *interface*.

```
ISaveable werewolf = new Monster("Werewolf");
System.out.println("Monster is " + ((Monster) werewolf).getName() );
```

> All **variables** declared inside **interface** are implicitly public static final **variables** (constants). All methods declared inside Java **Interfaces** are implicitly public and abstract, even if you don't use public or abstract keyword. **Interface** can extend one or more other **interface**.

Inner classes (Lecture 104-106)

In Java you can nest a *class* inside another *class.*

There are four types of nested classes

- static nested class
  - mainly used to associate a class with its outer class
  - identical to top level class, but packaged in its outer class, not in the package
  - therefor can not access non-static methods or members of its outer class without creating instance of that class
- non-static nested class, that's an inner class
  - instance of inner class has access to all members of outer class, even private ones
  - `OuterClass.this.<member>`
  - instantiate by chaining
  - OuterClass.InnerClass innerObject = outerObject.new InnerClass();
- local class, inner class defined inside of a scope block (usually a method)
  - scope restricted to that block
  - not very often needed / useful
- anonymous class, a nested class without a class name
  - also local class, but has no name
  - has to be declared and instantiated at the same time, because has no name, so there is no reference to it, no way to instantiate later
  - used when a local class is required only once
  - very common for touch-screen event handlers to use in an user interface

Inner (non-static nested) classes are quite useful, because it often does not make sense to refer to a class without its outer class.

Outer class `GearBox`, inner class `Gear`, no sense referring to gear without gearbox.

```
// public (non static) inner class Gear
public class Gearbox {
  private ArrayList<Gear> gears;
  private int currentGear;
  public GearBox() { /* constructor Gearbox */ }
  public class Gear {  private int gearNumber;
    public Gear() { /* constructor Gear */ }   }
}
Gearbox gearBox = new GearBox();
// Syntax : <type> <variable name> <outerclass>.new <innerclas>() !!
Gearbox.Gear neutral = gearBox.new Gear( 0 ); // OK for public non-static
// this does NOT work :
Gearbox.Gear first = new Gearbox.Gear( 1 );
// this does NOT work for public non-static, OK for public static class Gear():
Gearbox.Gear second = new gearBox.Gear( 2 );
```

```
// private inner class Gear
public class Gearbox {
  private ArrayList<Gear> gears;
  private int currentGear;
  public GearBox() { /* constructor Gearbox */ }
  public addGear() {/* adds new Gear( <int> ) to gears */}
  private class Gear {  private int gearNumber;
    public Gear() { /* constructor Gear */ }   }
}
Gearbox gearBox = new GearBox();
gearBox.addGear( 0 ) ; // OK
```

Usually the inner classes will have a `private` access modifier. Instances will be created and manipulated by the outer class' methods.

Interlude 2

Paused Java course to

- do preparational courses for a NodeJS Masterclass I was supposed to do from April on. End of March I was told this Masterclass will not happen :-)
- prepare new house I got :-) Painting, flooring, packing, moving, unpacking.

Restarted 2019-05-08 by going back a few Sections.

There are 2 special cases of inner classes.

- local classes
- anonymous classes

Local classes are declared inside a block (method or if-statement). Their scope is restricted to that block.

Unlike anonymous classes, which are used quite often, local classes are used less often.

Local class :

Can be used for different objects in scope.

```
// Button.java
public class Button {
  private String title;
  private OnClickListener onClickListener;
  public Button(String title) { this.title = title; }
  public String getTitle() { return title; }
  public void setOnClickListener(OnClickListener onClickListener) {
            this.onClickListener = onClickListener;
  }
  public void onClick() { this.onClickListener.onClick( this.title ); }
  public interface onClickListener { public void onClick(String title); }
} // <- Button.java
```

```
// Main.java
private static Button btnPrnt = new Button("Print");
private static Button btnExit = new Button("Exit");
// local class implements interface :
class ClickListener implements Button.OnClickListener {
  public ClickListener() { /* Listener has been attached */ }
  @Override (interfaces onClick method)
  public void onClick(String title) { /* Button has been clicked */ }
}
bntPrnt.setOnClickListener(new ClickListener());
bntExit.setOnClickListener(new ClickListener());
```

> Teacher:
>
> Local class `ClickListener` , how it is actually been defined, is applicable just for that block only.
>
> This particular method using a local class could probably be pretty useful if you want to assign exactly the same object, say to several buttons, if you had several buttons on the screen at the same time, for example. The class is not used anywhere, so making it local in this particular scenario makes sense. Because we are not using it in a shared environment or anywhere else, implementing it as a local class within this method does actually make sense.

Vague, investigate further. Why define an interface and then create a local class implementing it, in stead of having it all done in Button class itself?

Anonymous class :

Also a local class, but has no name. Have to be declared and instantiated at the same time, because has no name. Can be used only once. Useful in e.g. mobile development, each button reacts different to click.

```
// Main.java
private static Button btnPrnt = new Button("Print");
private static Button btnExit = new Button("Exit");
// anonymous class implements interface,
//  used only once for this button
bntPrnt.setOnClickListener(new Button.OnClickListener) {
  @Override
  public void onClick(String title) { /* Print has been clicked, do something */ }
};
bntExit.setOnClickListener(new Button.OnClickListener) {
  @Override
  public void onClick(String title) { /* Exit has been clicked, do something else */ }
};
```

Abstract Classes

Abstraction is when you define the required functionality for something without actually implementing the details. Focus on what needs to be done, not how it is done. *Interfaces* are purely *abstract*.

```
public abstract class Animal {
  private String name;
  public Animal(String name) { this.name = name; }
  public abstract void eat();
  public abstract String breathe();
  public String getName() { return name };
};
```

This forces classes extending the abstract class to implement these methods.

Interfaces are purely abstract, no implementation whatsoever. In abstract classes it can be a mix of non-abstract code and abstract methods.

Subclasses derived from abstract classes can also be abstract.

```
public abstract class Bird extends Animal {
  public Bird( name ) { super( name ); }
  /* implement eat(), breathe()*/
  public abstract void fly(); /* to be implemented by subclasses of Bird */   }
```

```
 public class Pinguin extends Bird {
   public Bird( name ) { super( name ); }
   /* implement fly() even if he can't */   }
```

Difference *abstract class* <-> *interface* : *Is-a*-relationship or *Has-a*-relationship.

A pinguin *Is-a* bird, a car *Has-a* gearbox.

As bats and insects are not birds, but can fly, in the above example it would be better to have a CanFly interface with fly()-method, which both birds and bats and insects can implement.

As all animals eat and breathe, it is better to put these methods in an abstract class for animals.

An abstract class can have member variables that are inherited, this can not be done in an interface.

Interfaces can have variables, but they are all public static final variables, essentially constant values that should never change, with a static scope. They have to be static, because non-static variables require an instance, interfaces can not be instantiated.

Interfaces can not have constructors, abstract classes can. Interface methods are automatically public, abstract class methods can have any visibility. Abstract classes can have defined methods, methods with an implementation, all methods in an interface are abstract.

In the AbstractClassChallenge-solution, in the class definition, variables were initiated of the type of the abstract class itself!! Weird!!

```
 public abstract class ListItem {
     protected ListItem rightLink = null, leftLink = null;
     protected Object value; // Object -> can be any type when implemented
 }
```

We also built a binary search tree, it uses recursion. Java has a maximum recursion depth of 63.

First node added will be the root, other nodes added to the tree according to value. Traversing the tree uses recursion. Removing a node is very complicated.

Section 10, Java Generics

13-05-2019, finished 14-05-2019 Go to Top Go to Bottom

Generics

Generics allow us to pass *types* to classes, *type parameters*. We have used them before.

```
// Pass the type String to class ArrayList
ArrayList<String> items = new ArrayList<>();
```

Using a class like ArrayList without passing a *type* results in an object of the *raw type*.

```
ArrayList items = new ArrayList<>();
```

Raw types used to be the only thing available, before generics entered the scene (Java 1.5). Some ancient code may still have them.

Generics were introduced to help find bugs at compile time. It tells the compiler which type is expected, and helps it find code that does not meet that expectation.

```
ArrayList<Integer> items = new ArrayList<>();
items.add(0);
items.add("0"); // Compile time error
```

```
public abstract class Player {/**/}
public class Team<T> {
  public static void addPlayer(T player) {
    /* ((Player) player).getName(), casting needed now */   }   }
public class SoccerPlayer extends Player {/**/}
SoccerPlayer matthijs = new SoccerPlayer("Matthijs de Ligt");
DartsPlayer raymond   = new DartsPlayer("Raymond van Barneveld");
Team<SoccerPlayer> ajax = new Team<>(); // using generics
ajax.addPlayer(matthijs); // OK
ajax.addPlayer(raymond);  // compile time error, wrong type of player
```

Any type can now be passed to Team, this is unwanted.

Enter *bounded type parameters*, restrict types allowed to be passed into class.

```
public class Team<T extends Player> {
  public static void addPlayer(T player) {
    /* player.getName(), casting not needed any more */   }   }
```

Types passed as parameter can be either a class or an interface.

You can pass 1 class and multiple interfaces as types. Always pass the class first.

```
public class Team<T extends Player & Coach & Manager> {/**/}
```

Implement *compareTo()* method from *Comparable* interface.

```
public class Team<T extends Player> implements Comparable<Team<T>> {
  public static int compareTo(Team<T> otherTeam) {/**/}
}
ArrayList<Team> teams; /* ... */
Collections.sort(teams); // uses compareTo() method defined in Team class.
```

Section 11, Naming Conventions and Packages, 'static' and 'final' Keywords

14-05-2019, finished ..-..-.... Go to Top Go to Bottom

Naming Conventions

Makes code easier for others to read, and for yourself if you read it back way later.

Things programmer names :

- Packages
  - always lower case
  - should be unique
  - use your internet domain name, reversed, as a prefix for the package name (??)
    - invalid characters in a domain name (i.e. '-') should be replaced by underscore
    - domain name components starting with a number should instead start with an underscore
    - domain name components starting with a Java keyword should instead start with an underscore
  - Oracle : https://docs.oracle.com/javase/specs/jls/se6/html/packages.html#7
- Classes
  - CamelCase, start with capital
  - name should be a noun (they represent things)
  - each word in the name should also start with a capital
- Interfaces
  - CamelCase
  - consider what objects implementing the interface will become, or what they will be able to do (e.g. List, Comparable, Serializable)
- Methods
  - mixedCase, start with lowercase
  - often verbs
  - reflect the function performed or the result returned
  - e.g. size(), getName(), addPlayer()
- Constants

    - UPPERCASE
    - separate words with an underscore
    - declared using the *final* keyword
    - e.g. static final int MAX_INT; static final double PI = 3.141592653;
- Variables
    - mixedCase, start with lowercase
    - meaningful and indicative
    - no underscores
    - e.g. i, league, boxLength
- Type Parameters
    - single character, capital letter
        - E - Element (used extensively by the Java Collections Framework)
        - K - Key
        - N - Number
        - T - Type
        - V - Value
        - S,U,V etc. - 2nd, 3rd, 4th types
    - https://docs.oracle.com/javase/tutorial/java/generics/types.html

Packages

Over 9 million Java developers worldwide, therefor Class/Interface naming conflicts are inevitable.

Mechanism is needed to fully specify class. Allow use of classes with the same name in the same project (or, even, the same class).

```
import javafx.scene.Node;
public class Main {
  public static void main(String[] args) {
    Node node = null;   /* using the import */}   }
```

```
// When using classes with same name from different packages,
//   names have to be specified when declaring
public class Main {
  public static void main(String[] args) {
    javafx.scene.Node nodeFx  = null;
    org.w3c.dom.Node  nodeW3C = null;  }   }
```

or if most of your nodes are from 1 class, and one or a few from another class :

```
import javafx.scene.Node;
public class Main {
  public static void main(String[] args) {
    Node nodeFx = null;              /* JavaFX */
    org.w3c.dom.Node nodeW3C = null; /* W3C */   }   }
```

Reasons to use packages :

- programmers can easily determine that the classes are related
- it is easy to know where to find the classes and interfaces that can provide the functions provided by the package
- because package creates a new namespace, class and interface name conflichts are avoided
- classes within the package can have unrestricted access to one another while still restricting access for classes outside the package

When importing `java.awt.*` like this :

```
import java.awt.*;          // Java Abstract Window Tootkit
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

it does not import `java.awt.event.Windowadapter` because `java.awt.event` is a separate package.

IntelliJ : keep `<Ctrl>` pressed and hover with cursor over a class name (e.g. `myFrame extends Frame`, hover over `Frame`), it will show you the package it is from, it's version and the class declaration for the class (e.g. `[<11>] java.awt, public class Frame extends Window implements MenuContainer`)

Create your own packages, with Java package names starting with `com.example` or `org.example` is reserved for packages that you are never going to distribute, you can also use your own `com.yourname.yourpackage`.

IntelliJ: when creating a new class, you can specify the package name, e.g. `org.example.game.Player`. It will create a new subdir of `game` for dir `org.example` and create the class `Player` t here.

Packages are published in a .JAR-file (Java ARchive).

> A **JAR** (Java ARchive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution. **JAR** files are archive files that include a Java-specific manifest file.

In IntelliJ, to publish a package,

- go to (menu) *File-Project structure*
- under *Project Settings*,choose *Artifacts*
- click the *plus-sign*
- choose *JAR*
- choose *From modules with dependencies*
- for an executable package, specify main class
- choose *extract to the target JAR*
- press *OK*
- go to (menu) *Build-Build Project*

The project is now built. To generate the .JAR-file

- go to (menu) *Build-Build Artifacts*
- in popup choose *Build*

A .JAR-file is now created : `./out/artifacts/<projectname>_jar/<projectname>.jar`.

To use the built library in another project, in the other project

- go to (menu) *File-Project Structure*
- under *Project Settings*, choose *Libraries*
- click on *plus-sign*
- navigate to .JAR-file, select it and click *OK*
- in popup *Choose Modules*, click *OK*

Now the package is `import`-able, or package names can be prefixed to class name/interface name.

Scope

Grant or restrict access to objects (visibility of class, member, variable). Easy once you understand it, but confusing until then.

Enclosing block, if Java can not find an object in the current code block, it will look for it in enclosing blocks.

Declaration of an object/variable (`<type> var = ...`) creates a new scope.

```
// accesibility from outside
public class ScopeCheck {
  public int publicVar = 1;          // accessible as <object>.publicVar
  private int privateVar = 2;        // only accessible through public getter()
  private int privateVar2 = 3;       // ditto
  /* privateVar * privateVar2 = 6 */
  public static publicStaticVar = 4; // accessible as ScopeCheck.publicStaticVar
  public void methodOne() {
    int privateVar = 20; /* privateVar now private to methodOne()*/
    /* privateVar * privateVar2 = 60, privateVar2 found in enclosing block */
    for (int privateVar = 1; privateVar < 10; privateVar++) {
      /* privateVar now private to for loop code block */
    }
    for (privateVar = privateVar; privateVar > 0; privateVar--) {
      /* privateVar of methodOne is used (and altered) */
    }
    public class InnerClass {
      public int privateVar = 200; /* public/private, new var is created in this scope */
      public InnerClass() {
        /* privateVar for this scope is now 200 */
      }
      public int getPrivateVar() { return privateVar ; /* returns 200 */}
      public int getPrivateVar2() { return ScopeCheck.this.privateVar ; /* returns 2 */}
    }   }   }
```

Visibility

Visibility is determined by *scope* and *access modifiers*.

(*package-private*, no *access modifier*, `public class C {private v = 1; int getV { return v; }}`)

A *private* var in an inner class is visible to it's outer class, but not from outside the outer class.

A *public* var in an inner class is visible from outside the outer class.

```
OuterClass oc = new OuterClass();
OuterClass.InnerClass ic = oc.new InnerClass();
// ic.varOne not accesible here
// ic.varTwo accesible here
public class OuterClass {
  private InnerClass ic = new InnerClass();
  // ic.varOne and ic.varTwo accesible here
  public class InnerClass {
    private int varOne = 1;
    public  int varTwo = 2;   }   }
```

Scope challenge, crazy solution :-)

```
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {
        new X( new Scanner(System.in) ).x();   }   }
class X {
    private int x;
    public X( Scanner x  {
        System.out.println("Gimme a number : ");
        this.x = x.nextInt();   }
    public void x() {
        for (int x = 1; x <= 12; x++) {
            System.out.println(x + " * " + this.x + " = " + x * this.x);
        }   }   }
```

Access modifiers

If you expose fields of your classes (make them `public`), anyone using the class can manipulate them.

```
class Account {
  public int balance = 0;   }
Account.acc = new Account(); acc.balance = 1000000;
```

Making them `private` solves this. `public` methods can be used to manipulate these fields.

Four access modifiers.

Access control is granted by the top level or at the member level.

At top level, you can make your classes and interfaces `public` or `package-private`. You can not define a `private` class at top level.

Top Level

- only classes, interfaces and *enums* can exist at the top level, everything else must be included within one of these
- *public* (`public`), the object is visible to all classes everywhere, whether they are in the same package or have imported the package containing the public class
- *package-private* (no modifier), the object is only available within its own package and is visible to every class within the same package. Omit the modifier `public` to make a class *package-private*

Member Level (inside a class)

- *public* (`public`), public class members, fields, and methods can be accessed from anywhere
- *package-private* (no modifier), visible to every class within the same package, not to classes from other packages
- *private* (`private`), visible only within the class it is declared in. Also not visible in subclasses of its class
- *protected* (`protected`), visible anywhere in its own package but also in subclasses even if they are in another package

Before Java8 : When defining an interface, all variables declared are implicitly `public static final` (constants) and all methods are implicitly `public abstract`.

Java8: interfaces can have methods defined using `static` (can not be overwritten) or `default` (can be overwritten)

Since Java9, `private` and `private static` methods can be defined for interfaces, enables encapsulation.

Static

Declaring a variable `static` means it belongs to the *class*, not the *object* instantiated from the class. It is also called a *class variable*. This `static` variable is shared by all instances of that class. It holds 1 place in memory, how ever many instances of the class are created.

If you have a method that works on `static` fields only, it makes sense to make that method also `static`. We can access the method without having to use a classes instance.

Static methods and fields belong to the class, not to the instances of the class.

```
public class StaticTest {
  private int numInstances1 = 0;
  private static int numInstances2 = 0;
  public StaticTest() {
    numInstances1++;
    numInstances2++;
  }
  public int getNumInstances1() { return numInstances1; }
  public static int getNumInstances2() { return numInstances2; }
}
StaticTest.getNumInstances2(); // -> 0
StaticTest st1 = new StaticTest(); // numInstances1 = 1, numInstances2 = 1
StaticTest st2 = new StaticTest(); // numInstances1 = 1, numInstances2 = 2
StaticTest.getNumInstances2(); // -> 2
```

```
public class Main {
  public static void main(String[], args) {
    // this must be static to give Java an entry point,
    //  main() can now be called without a class instantiation.     }   }
```

Class *Main* does not have to be called *Main*, but it is convention to do so at the highest level of a project. We can look for the *main* method, entry point to our project, in the *Main* class.

Normal class fields require an instance of the class. You can not reference normal class fields/methods from a static context **in its own class**, because the instance may not have been created yet.

The reverse is not true. When an instance is created the *class variables* and *methods* (`static`) are available.

Final

In general used to define constant values. Strictly speaking, `final` fields are not actually constants, because they can be modified, but only once. Modification must be done before the class constructor finishes. So either when defining the variable (`private final int nr = 1;`) OR assign it a value in the constructor method.

Set variables as `final` if the value should not be changed after initial assignment.

Real constants are usually declared as `static final` and written in upper case, e.g. `Math.PI` :

```
// no class can extend a final class
public final class Math {
  // empty private constructor, this class can never be instantiated.
  private Math() {}
  public static final double PI = 3.14159265358979323846;   }
```

By marking a class `final`, you can prevent it being subclassed.

```
public class MyMath extends Math { /* ... */ } // impossible, Math is final
Math m = new Math();                           // impossible, private empty constructor
```

To protect methods from being overridden in subclasses, use the `final` keyword.

```
public final void storePassword() { /* ... */ }
```

Static initializers

Static initialization blocks, a static equivalent of a constructor. It is an advanced feature, there's rarely a case to use them.

The constructors we have seen until now are instance constructors, used to initialize an object created from a class, they are not static.

```
public class SomeClass {
  static { /*  code */ }
  public SomeClass() { /* constructor */ }
  static { /*  more code */ }
}
```

All static initialization blocks are run before the constructor is called.

Section 12, Java Collections

21-05-2019, finished 03-06-2019 Go to Top Go to Bottom

Collections

Interface `List` and *classes* `ArrayList` and `LinkedList` form part of the *Java Collections framework*.

`Set`s, `Map`s, `Tree`s and `Queue`s are also included in this framework, at the top level of the collections framework is the `Collections` class (a `Map` is not a true collection).

```
List<Seat> seats = new LinkedList<>(); seats.add(new Seat("A01"));
int foundSeat = Collections.binarySearch( seats, new Seat("A01"), null );
return seats.get(foundSeat).getPrice();
// Making it generic. Has its drawbacks though,
//  lot of casting because of methods class Collection does not have,
//  but interface List does
Collection<Seat> seats = new LinkedList<>(); seats.add(new Seat("A01"));
int foundSeat = Collections.binarySearch( ((List) seats), new Seat("A01"), null );
return ((Seat) ((List) seats).get(foundSeat)).getPrice();
```

```
// implement Comparable interface and compareTo() method to use these methods
public class <E> implements Comparable {
  @Override
  public int compareTo(E e) { /* ... */ }   }
Collections.sort(<List>);
Collections.reverse(<List>);
Collections.shuffle(<List>);
Collections.min(<List>);
Collections.max(<List>);
Collections.binarySearch(<List>) // binary search is very efficient.
```

When *shallow copying* a `List`, the order in the copy may be changed, but the objects in the original and the copy are one, change the object in the copy and it will also be changed in the original.

Comparable and Comparator interfaces

When implementing the `Comparable` interface on a class, you implement the `compareTo()` method in that class, returning an int of 1, 0, or -1. This method is used for searching, sorting, etc.

You can also create one or more Comparators

```
public class Theatre {  /* ... */
  // anonymous inner class implementing the Comparator interface
  // static, available as Theatre.PRICE_ORDER_ASC
  static final Comparator<Seat> PRICE_ORDER_ASC = new Comparator<Seat>() {
    @Override
    public int compare( Seat seat1, Seat seat2 ) {
      /* return positive, 0 or negative int */ }   }   }
/*...*/
Collections.sort( theatre.getSeats(), Theatre.PRICE_ORDER_ASC);
```

A Comparator that does not take all fields of an object into account is considered *inconsistent with* `.equals()`. The field(s) compared may be equal, but the objects may not be.

Map interface and HashMap class

Map interface is part of the *Collections framework*, but not a collection in the true sense of the word.

It is a collection of *key* - *value* pairs. The *keys* are unique.

`.put(<key>, <value>)` returns `null` if the *key* did not yet exist, otherwise the old *value* that got overwritten.

`.putIfAbsent(<key>, <value>)` stores the *key* and *value* if the *key* did not yet exist and returns `null`, otherwise returns the existing *value*.

There is no order in the HashMap, key-value pairs added later may appear earlier in loops.

`.remove(<key>)` if `<key>` is present removes pair and returns old value, otherwise returns `null`.

`.remove(<key>, <value>)` if `<key>` is present and its value equals `<value>` removes pair and returns `true`, otherwise returns `false`.

`.replace(<key>, <value>)` if `<key>` exists replaces value and returns old value, otherwise returns `null` .

`replace(<key>, <oldValue>, <newValue>)` if `<key>` is present and its value equals `<oldValue>` replaces value and returns `true`, otherwise returns `false`.

```
Map<String, String> myMap = new HashMap<>();
myMap.put("key1", "valueA");    // -> null
myMap.put("key1", "valueB");    // -> "valueA"
myMap.get("key1");              // -> "valueB"
myMap.forEach((key, value) -> { /* ... */ });     // not in order of adding pairs
for (String key: languages.keySet()) { /*...*/ }; // not in order of adding pairs
myMap.containsKey("key1");      // -> true
myMap.containsValue("valueX");  // -> false
myMap.put("key2", "valueC");
myMap.remove("key2", "valueD"); // -> false
myMap.remove("key2");           // -> "valueC"
```

Immutable classes

Immutable classes can't be changed once they are created. Valuable if you want instances of your classes to be immutable, but also to increase encapsulation and reduce errors if external code is allowed to modify the class instances (objects).

Lecture is somewhat unclear. Do not allow outside code to make any changes to an instances fields.

```
public class Location {
  private final Map<String, Integer> values;
  public Location(Map<String, Integer> values) {
    // create new object in stead of just this.values = values;
    //  otherwise if variable used in code to instantiate this object
    //   was altered, it would also alter in the instance.
    this.values = new HashMap<String, Integer>(values);   }
  public Map<String, Integer> getValues() {
    // return a new object initialized with instance variable
    return new HashMap<String, Integer>(this.values));   }   }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html for more information.

> 1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
> 2. Make all fields `final` and `private`.
> 3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as `final`. A more sophisticated approach is to make the constructor `private` and construct instances in factory methods.
> 4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
>   - Don't provide methods that modify the mutable objects.
>   - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

If you use class instances as a key in a Map (`Map<Location, Integer>`) you want to make sure your class is immutable.

Java will compile fine, but give a runtime error if a null value for a Map was passed at runtime when null value is passed on into Map constructor (`NullPointerException`).

```
public class Location {
  private final Map<String, Integer> values;
  public Location(Map<String, Integer> values) {        // values = null
    this.values = new HashMap<String, Integer>(values); /* boom */   }   }
```

```
    this.values = values == null
      ? new HashMap<String, Integer>()
      : new HashMap<String, Integer>(values) ; /* no boom */
```

or

```
    values == null ? new HashMap<String, Integer>() : values ;
    this.values = values ; /* no boom */
```

Set interface & HashSet class

Sets are used less often than Lists and Maps, but can be very useful.

- a Set has no defined ordering (Oracle says chaotic :-))
- a Set can not contain duplicates
  - `s.add()`, returns true if added, false if unable to add (duplicate)
  - `s.remove()`, returns true if removed, false if not present
  - `s.clear()`, `s.size()`, `s.isEmpty()`
  - `s.forEach( e -> { /* ...*/ })`
  - `for (<Type> t : s) {/* ... */}`

`HashSet` is the best performing class implementing the `Set` interface. It currently uses `HashMap`, storing the elements as key and a dummy Object as value, thus ensuring uniqueness of the values (`HashMap` can not have duplicate keys).

.equals() and .hashcode()

If you use your own objects as a key in a `Map`, or an element in a `Set`, the class should override the `.equals()` and the `.hashcode()` methods. As a programmer you have to decide what makes your objects unique, in order to maintain uniqueness in `Map`-keys and `Set`-elements (make the class *immutable*).

E.g. for a planet, the name should be unique, but for a person, that would not be enough.

Use the same field(s) computing the hashcode as you do to determine equality.

```
@Override
public boolean equals( Object obj ) {
  /* if object == object, return true */
  if ( this == obj ) return true;
  /* given object null or different class, return false */
  if ( obj == null || getClass() != obj.getClass() ) return false;
  /* cast object to this class */
  Planet that = (Planet) obj;
  /* determine if it is equal, in this case String.equals() */
  return name.equals( that.name );   } // equals()
@Override
public int hashCode() {
  return Objects.hash(name);   } // hashCode()
```

Rules for `.equals()` : https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

There is discussion about using `instanceof` or `getClass()` in `equals()`, regarding subclassing and the symmetry rule :

- `obj1.equals(obj2) == obj2.equals(obj1)`
- https://stackoverflow.com/questions/596462/any-reason-to-prefer-getclass-over-instanceof-when-generating-equals

`getClass()` is *inheritance unfriendly*.

When using `instanceof`, it is advisable to make the `equals()` method `final` in the super class, so it can not be overridden in a sub class.

There was talk about *hashcode collision*, something about using hashCode() of a String field of an instance colliding with an actual String.hashCode() for a String with the same value. The teacher added a prime number to the returned hashcode to avoid that.

```
// an instance planet with a field called name with the value "Earth"
planet.getName().hashCode(); // would return an int value
// a String with the value "Earth"
"Earth".hashCode();          // would return the same int value
```

To avoid that :

```
  @Override
  public int hashCode() {
    // To avoid collisions with String.hashCode() for the same value,
    //  add a prime number (not very well explained).
    return this.getName().hashCode() + 57;
  }
```

Not sure if the same applies to `Objects.hash(name)`, this method was used by a generated `equals()/hashCode()` override by IntelliJ (java 7+ template).

Sets - Symmetic & Asymmetric

Bulk operations

- `<Set1>.addAll( <Set2> )`
- `<Set1>.containsAll( <Set2> )`
- `<Set1>.retainAll( <Set2> )`
- `<Set1>.removeAll( <Set2> );`

Bulk operations on sets are destructive, they modify the Set they are used on, except `.containsAll()`, it just returns `true` or `false`.

```
set1.addAll( set2 );   // set1 altered
```

```
Set<Integer> union = new HashSet<>( set1 ); // pass set1 to constructor
union.addAll( set2 );  // set1 unaltered
```

Symmetric and *asymmetric* operations on sets, algebraic terms

- *symmetric* : an operation on Set A using Set B yields the same result as the same operation on set B using set A.
- *asymmetric*: operations yield different results.

For instance, finding duplicates in 2 sets (intersection) is *symmetric*, comparing Set A to Set B, or Set B to Set A, will result in the same set of duplicates.

Unifying the sets and removing the duplicates, will also have the same result set, and is *symmetric*.

Removing duplicates from either set will result in different sets (assuming the sets are different) and is *asymmetric*.

Asymmetric difference : values contained in Set A, but not in Set B, or vice versa.

- *Set A.removeAll(Set B)*
- different from :
- *Set B.removeAll(Set A)*

Symmetric difference : values contained in Set A OR Set B, but not both (union minus intersection).

- *Set C = A.retainAll(Set B)*
- *Set A.addAll(Set B)*
- *Set A.removeAll (Set C)*
- now Set A holds union (Set A + Set B) minus intersection

- I can switch Set A and Set B and the outcome is the same -> *symmetric*

Check whether Set A is a superset of Set B (Set B is a subset of Set A)

- *Set A.containsAll(set B)*
- returns `true` or `false`
- If `( Set A.containsAll( set B ) && Set B.containsAll( Set A ) ) == true`, both sets hold same elements.

In challenge solution *enums* were used :

```
public class HeavenlyBody {
  private final BodyTypes bodyType;
  public enum BodyTypes { STAR, PLANET, MOON /* ...*/}
  public HeavenlyBody( String name, double orbitalPeriod , BodyTypes bodyType ) {
    /* ... */ }   }
/* now these constants can be referred to as e.g. BodyTypes.MOON */
public class Moon extends HeavenlyBody {
    public Moon( String name, double orbitalPeriod ) {
        super( name, orbitalPeriod, BodyTypes.MOON );   }   }
```

In `.equals()` comparing `enum`s it is better to use `==` . Something to do with compiletime versus runtime error detection by Java.

```
return name.equals(that.getName()) && bodyType == that.getBodyType();
```

When outputting the value of a field of the enum type, it will print the constants name. BodyTypes.MOON will print out as MOON.

The constants defined in the enum are objects that have methods that can be overwritten. There is also a final method `name()` for enum constants objects. There is also a final method `name()` for enum constants objects, it always returns the name in uppercase.

```
  public enum BodyTypes {
    STAR,
    PLANET,
    MOON { public String toString() { return "Moon"; }},
    /* ...*/    }
BodyTypes.MOON.toString(); // -> Moon
BodyTypes.MOON.name();     // -> MOON
```

Sorted Collections

HashMap and *HashSet* are in no particular order (*chaotic* according to Oracle).

The *LinkedHashMap* and *LinkedHashSet* maintain *insertion order* (by default).

Get value for key, or default value if not found :

```
.getOrDefault( <key>, <defaultValue>)
```

Create an unmodifiable *view* of a map :

```
javaCollections.unmodifiableMap(<SourceMap>)
```

Loop through *HashMap* / *LinkedHashMap* :

```
for (Map.Entry<String, StockItem> item : stockList.entrySet()) { /* ... */ }
```

Create a *LinkedHashMap*, items ordered in order of insertion.

```
private final Map<String, StockItem> stockList = new LinkedHashMap<>();
```

Create a *TreeMap*, items ordered in order defined by *Comparable* / *.compareTo()* or *Comparator* (*natural ordering*). Calls compare method each time an item is added, so more work for system.

```
private final Map<StockItem, Integer> shoppingBasket = new TreeMap<>();
```

In Java, it is OK for a class to have a variable and a method with the same name.

Section 13, JavaFX

04-06-2019, finished ..-06-2019 Go to Top Go to Bottom

Go to Top

Bottom anchor