## Introductory programming - guldnoter

1

[**Lecture 1, Chapter 1 - Objects and classes**](#_ka1mhj2at9u6)2

[Chapter 1 - Objects and classes](#_cg6gj4mfii0m) 2

[Concepts chapter 1 - Objects and classes](#_rxv5o9q2rlbd) 4

[**Lecture 2, Chapter 2 - Objectifying in java**](#_hg00l74ustgj)4

[Concepts chapter 2 - Understanding Class Definitions](#_d4d5f6spinoe) 6

[**Lecture 3, Chapter 3 - Object interaction**](#_67ciht2f3v43)7

[Concepts chapter 3 - Object Interaction](#_rs7dp334e16g) 9

[**Lecture 4, Chapter 4.1 - 4.8: Collections and loops**](#_g0fwjfmuyoj5)10

[Concepts chapter 4.1 - 4.8: grouping objects](#_bbaiq3p04buw) 14

[**Lecture 5, chapter 7 - Arrays and for-loops**](#_4b1ltevryt49)14

[Concepts chapter 7](#_kf6s6xd51lf6) 16

[**Lecture 6, chapter 4.8 - 4.15 - Iterators**](#_3j2oazsfetg4)16

[Concepts chapter 4.8 - 4.15](#_d0smoh83o2p7) 17

[**Lecture 7, chapter 6 - Libraries, Sets and Maps**](#_ygkdaelds3y6)18

[Concepts chapter 6 - more sophisticated behaviour](#_c586rlct9nnr) 20

[**Lecture 8: Version Control and Build Systems**](#_3p0moe2qvu4s)21

[**Lecture 9, chapter 8 - Class design**](#_p8978yv1dklr)22

[Concepts chapter 8 - designing classes](#_ehrdn0jvc64o) 23

[**Lecture 10, chapter 9 - Software testing**](#_8s29tbdvlzr)24

[Different types of errors](#_abtjnbusqh1f) 24

[Concepts chapter 9 - well behaved objects (testing)](#_xtccupjfshdv) 25

[**Lecture 11, chapter 10 - Inheritance**](#_qpu9v7d5m5rq)25

[Concepts chapter 10 - improving structure with inheritance](#_t0k3gf9mj2me) 28

[**Lecture 12, chapter 11: More inheritance / Abstract methods and classes**](#_y3b5pb3da3wx)29

[Concepts chapter 11 - more about inheritance](#_grcico7kw770) 30

[**Lecture 13 - Chapter 12: Interfaces and Abstract Classes**](#_u6iomomuq7hk)30

[Concepts chapter 12 - Further abstraction techniques](#_3khhtjjnvl53) 31

[**Lecture 14 - Chapter 14: Exceptions and File I/O**](#_rc30kilswcxp)32

[Defensive programming](#_2q1ndq1srm3p) 32

[Concepts chapter 14 - Handling errors](#_nbj8x8xfvzea) 33

[**Concept list**](#_wtmrbjfa06jn)34

[Midterm forberedelse](#_vstqv022od3l) 40

## Lecture 1, Chapter 1 - Objects and classes

**Object**: Represents a “thing” from the real world (or the problem domain). An object is an instantiation of a class.

**Class**: A class represents all objects of a particular type. A class defines objects *possible* characteristics and abilities. A class is an abstraction of an object.

**Plato’s allegory of the cave.** The prisoners in the cave are only able to see the shadows ‘objects’ reflected on the cave wall. They are trapped to only see these instantiations, while they cannot see the ‘Classes’ (or higher levels of reality like natural sciences, math, logic) that define these shadow’s characteristics.

**Objectives of the program**

Learn to - write, reat, test and document - programs, from a given problem, in Java.

* Analyse a problem description, with the intent of implementing a solution.
* Design a system based on an analysis.
* Implement a system based on a design.
* Test a system and evaluate to what degree the tests cover the system
* Apply basic programming constructs in Java
* Explain basic programming constructs in Java

### Chapter 1 - Objects and classes

**Objects:** Objects are specified by classes. Classes represent the general concept of things, while objects represent concrete instances of a class. There can be many different objects under one class. Objects a written with lower case letters - ex: **car1**

**Classes:** A class represents the general framework of an object. It’s an abstraction of objects. You can instantiate a class into an object. It, for example, specifies the type of object - ex: **Car**. Objects represent individual instances of a class.

A class consists of fields, constructors and methods. In class (Circle.java) we define the different parameters that an object *circle* can have. We create objects - a specific circle - in the main file. We can decide in the main file, that a specific circle we create can be blue, but it can only be blue, because we have defined it can change colour in the Circle.java file (which defines the attributes of that class).

**Methods:** We communicate with objects using *methods*. We can make objects do something if we *invoke* methods on them - ex: **moveDown**, **makeVisible**, **moveHorizontal** or **changeSize**. It is good practise to make all methods public.

**Parameters:** Methods have parameters to specify additional instructions for a task. For each parameter, it defines a *type* and a *name* - ex:

* car1.moveDown(***datatype* *name***)
* car1.moveDown(**int** **distance**)
* car1.moveDown(**50**)

**Data types:** A parameter has types. The type defines what kind of value a parameter can take. Ex:

* **Int** = Integral values = Whole numbers = 123... Ex: Int distance.
* **String** contains text.
* **Boolean** contain **true, false** statements.
* **Double** contains decimal point numbers:

**Signature:** The method name and the parameter types of a method are called its signature. Together, they provide the information needed to invoke a method.

**Multiple instances:** Many similar objects can be created from a single class.

**State:** The set of values of all attributes defining an object (ex: x- position, y-position, color, visibility) is called an objects state. Common terminology.

**Calling/invoking methods:** When you call or invoke a method on an object, you’re asking it to do something (like when you hit ‘run code’ in a programming software). Objects can communicate by calling each other’s methods.

**Variables:** We store information in variables, named locations in memory. Naming a piece of information allows us to use that name later, accessing the information we stored. A variable can store just one value, so any previous value is overwritten.

**Source code:** The underlying code defining the details of a class. The source code defines what fields and methods a class have and also what happens when a method is invoked.

**Compiler/compile:** A program/function that translates Java source code (which is easy to read for humans, but difficult to read for computers) into machine code, that a computer can read and run. *Every time we change the source code, we must first run the compiler before we can use the class again to create an object. Otherwise, the machine code the computer needs will not exist.*

**Scope:** Curly braces mark the scope of a class and method.

**Statement**: One line of code that performs a single task. Semicolons are used to mark the end of a statement, one line of code that performs a single task. Remember, no semicolons for classes or methods!

**Result / return value:** Methods may return information about an object via a return value or it may not. This is defined by using the word **void** (don’t return value) or **string** (return value) in front of the method. Ex:

* **String getName()**
* **void changeName(String replacementName)**

**Compiling files**: we can compile a .java file from the terminal with the command: javac filename.java

If the file compiles successfully, this command produces an executable class: FileName.class. Executable means we can run this program from the terminal.

We run the file by entering this in the terminal: java filename

### Concepts chapter 1 - Objects and classes

* **Object:** Java objects model objects from a problem domain.
* **Class:** Objects are created from classes. The class describes the kind of object. The object represents individual instances of the class.
* **Method:** We can communicate with objects by invoking methods on them. Objects usually do something when a method has been invoked.
* **Parameter:** Methods can have parameters to provide additional information for a task.
* **Signature:** The method name and the parameter types are called the methods signature. They provide the information needed to invoke that method.
* **Type:** Parameters have types. The type defines what kind of value a parameter can take.
* **Multiple instances:** Many similar objects can be created from a single class.
* **State:** Objects have state. The state is represented by storing values in fields.
* **Method calling:** Objects can communicate by calling each other's methods.
* **Source code:** The source code determines the structure and behavior (the fields and methods) of each of the objects of that class.
* **Result:** Methods may return information about an object via a return value.

## Lecture 2, Chapter 2 - Objectifying in java

**Class:** The framework of the specific object.

public class Car {}

**Fields:** A Java field is a variable inside a class. For instance, in a class representing an employee, the Employee class might contain the following fields: name. Position. Values are stored in fields (and other variables).

private String car;

private int modelYear;

private String modelName;

**Constructors:** A constructor initializes an object. A constructor in Java is a special method that is used to declare the default state of an object. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. A constructor has the same name as the class and doesn’t declare a return value.

public Car(String car, int year, String name) {

this.car = car;

modelYear = year;

modelName = name; }

Close association with the fields:

* Initial values are typically assigned in the constructor
* Parameters of constructor often contain these values

**Parameters:** A method has parameters. Used to pass data such as string, int and booleans using methods objects through its defined parameters.

public void CarMove(int speed, String direction)

**Accessor methods:** And accessor method is a get method. It gets information about the state of an object and returns the value of a field or a simple calculation. If it doesn’t change the state of the object it is an accessor.

* give information about the state of an object
* returns the value of a field or a simple calculation
* does not have parameters
* name: getSomeOrOther()

public int getPrice() {

return price; }

**Mutator methods:** Is used for changing the state of an object and changes the value of a field. Also called a setter method. If it changes the state of the object it is a mutator.

* is used for changing the state of an object
* changes the value of a field
* takes parameters, e.g., the new value of a field
* doesn’t return a value. Is always void.

public void insert Money(int amount) {

balance = balance + amount; }

**Field variable:** Stores a value for the lifetime of the object. Can be read and written from all methods of that class. Exists for each object.

private int balance;

**Local variable:** A local variable: Declared inside a method or constructor. Disappears when execution exits the block where it is declared, therefore, cannot be accessed by any other method.

int amountToRefund = 0;

String emptyString = ””;

**Expressions:** Expressions can take many forms example of:

**Arithmetic operations** on smaller expressions: **+ - \* / \%**

**Logical operators** on boolean expressions: **! \&& ||**

**Comparison** of values of expressions: **== != < > <= >=**

**Ternary conditional operators:** A boolean statement used to rewrite if statements: **? :**

**String concatenation:** Combining a string with variables and calculations.

return "Tree(age = " + age + ", height = " + height + ")";

**Conditional statement / If else statements**

If the condition is true, do one thing, otherwise do another thing. Often formulated with if else syntax.

**File and class:** Each file has one primary class named after the file. So when we name a Java file *Main.java* or *Cupcake.java* the primary class of that file will also be named *public class Main* or *public class Cupcake*. A Java program must have one class whose name is the same as the program filename. In the example, the Person class must be declared in a program file named Person.java.

**The main method**

Inside the class - at least in the main file - we have our main method. *The main method lists and runs our program's tasks.*

public static void main(String[] args) {}

**The integer class:** The Integer class wraps a value of the primitive type int in an object. The parseInt() function parses a string and returns an integer. An object of type Integer contains a single field whose type is int. Eg:

int C = Integer.parseInt(A);

### Concepts chapter 2 - Understanding Class Definitions

* **Object creation:** Some objects cannot be constructed unless additional information is provided specified in the parameters of the constructor.
* **Field:** Fields store data for an object to use. Fields are also known as instance variables.
* **Comment:** Comments are inserted into the source code of the class to provide information to humans.
* **Constructor:** Constructors allow each object to be set up properly when it is first created.
* **Scope:** The scope of the variable defines the section of source code from which the variable can be accessed.
* **Lifetime:** The lifetime of the variable describes how long a variable continues to exist before it is destroyed.
* **Assignment statement:** Store the value represented by the right- hand side of the statement in the variable named on the left.
* **Accessor method:** Accessor methods return information about the state of the object. - *Return -*
* **Mutator method:** Mutator methods change the state of the object. *- Void -*
* **System.out.Println() or print:** Prints its parameters as text to the terminal.
* **Conditional statement:** Takes on or two possible actions based on the result of a test.
* **Boolean expression:** Has only two possible values - true or false. They are commonly found controlling the choice of action of a conditional statement.
* **Local variable:** A local variable is a variable declared and used within a single method. Its scope and lifetime is limited to that method.

## Lecture 3, Chapter 3 - Object interaction

**Abstraction:** Ignore details and focus attention on the high level view of the problem.

**Modularization:** Divide a big problem into smaller parts.

* Examine and solve each sub-problem separately.
* Combine the solutions for the parts into a solution for the big problem
* Divide-and-Conquer

**Modulo operator:** The mod operator **(%)** computes the remainder of a division. Especially useful for replacing words or ints on a fixed list: 5 % 12 == 2

else if ((i % div1) == 0){//Method body}

**Object diagrams and class diagrams:** The class diagram shows the classes of an application and the relationship between them. It gives information about the source code and presents the static views of a program.

The object diagram shows the objects and their relationships at one moment in time during the execution of an application. It gives information about objects at runtime and presents the dynamic view of a program.

**Primitive types** - lowercase type name. Variables with a primitive type contain a value. Ex:

private int limit;

**Int** 32 bit integer

**long** 64 bit integer

**short** 16 bit integer

**byte** 8 bit integer

**char** characters 'a', 'A', ...

**Double** 64 bit floating point number

**float** 32 bit floating point number

**boolean** Truth value: false or true

**String** Text string (from Java)

**Object types -** uppercase type name. A variable with an object type contains a reference to an object. Ex: private NumberDisplay hours;

**String** Text string (from Java)

**All classes** All predefined classes for ex:

**Scanner** Class that takes an input

**ArrayList** Creates a list of elements that can be regularly modified

**Array[]**

**HashMap** Stores a value and connects it to a key

**List**

**Method calls – external:** Methods in another object can be called by writing:

object.method();

FizzBuzzMethod Fizzi = new FizzBuzzMethod();

Fizzi.fizzBuzz(N, X, Y);

**Method calls – internal:** Internal helper methods are typically private. Make a method private, if it doesn’t make sense to call it from the outside. Helper methods are great for avoiding having the same code repeated.

**Objects creating other objects:**

Objects ‘numberDisplay’ can create other objects ‘hours’ using the new operator.

public ClockDisplay() {

hours = new NumberDisplay(24); }

public NumberDisplay(int rollOverLimit) {

limit = rollOverLimit; }

public NumberDisplay(int limit) {

this.limit = limit;

**Debugging;** Powerful tool that let you go through your code one step at a time and see how variables update, so you can find out where potential bugs are to be fixed.

### Concepts chapter 3 - Object Interaction

* **Abstraction:** The ability to ignore subparts of a problem to focus on the attention to a higher level of the problem.
* **Modularisation:** The process of dividing a whole of problems into well defined parts that can be built and examined separately and that interact in well defined ways.
* **Classes define types:** A class name can be used as the type for a variable. Variables that have a class as their type can store objects of that class.
* **Class diagrams:** The class diagram shows the classes of an application and the relationship between them. It gives information about the source code and presents the static views of a program.
* **Object diagram:** The object diagram shows the objects and their relationships at one moment in time during the execution of an application. It gives information about objects at runtime and presents the dynamic view of a program.
* **Object references:** Variables of object types store references to objects.
* **Primitive types:** The primitive types of a program are the non-object types. Primitive types a int, char, boolean, double and long and has no methods unlike object types.
* **Object types:** A variable with an object type contains a reference to an object. Object types have methods and consist of example String, ArrayList, HashMap, Scanner and other classes.
* **Object creation:** Objects can create other objects using the new operator.
* **Overloading:** A class may contain more than one constructor, or more than one method of the same name, as long as each has a unique set of parameter types.
* **Internal method call:** When methods call other methods of the same class as part of their implementation.
* **External method call:** When methods call methods of other objects using .dot notation.
* **Debugger:** A debugger is a software tool that helps examining how an application executes. It helps to find bugs.

## Lecture 4, Chapter 4.1 - 4.8: Collections and loops

**Collections:** A java collection is a single unit containing multiple objects. A collection in Java is a term describing the framework that provides the architecture to store and manipulate a group of objects. Collections can perform several data managing operations such as searching, sorting, retrieving, inserting, manipulating, and deletion. The classes that uses Java’s collection interface and it therefor a Java collection comes in the form of:

* ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet.

**Class libraries:** The java.util class library contains useful classes we can use in our program without having to code the classes ourselves. Most classes you can use to work with collections can be found in [java.util](https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html).

**Defining a collection:** First we:

* We specify the type of collection eg.: ArrayList
* The type of object our collection will contain eg.: <String>

import java.util.ArrayList;

public class Test{ //Naming the class

private ArrayList<String> example; //Defining the ArrayList in fields

*private List<Tree> example; //Can also be done this way and declared as ArrayList in the constructor like below*

public Test(){ //Creating a new ArrayList in the constructor

example = new ArrayList<>();

}

public void addFile(String filename){ //Defining methods

example.add(filename);

}

public int size(){ //Defining methods

return example.size();

}

public static void main(String[] args) { //Main method

Test nameOfNewList = new Test(); //Creating an instance named nameOfNewList

nameOfNewList.addFile("test"); //Adding elements to nameOfNewList

System.out.println(nameOfNewList.size()); //Printing size of list

You can also define the list in the fields, and then later specify it as ArrayList in the constructor:

protected List<String> toppings; ... //Fields

public Pizza() {

super("Pizza", 45);

toppings = new ArrayList<>(); //Constructor

}

**Array:** *Not ArrayList.* Storing words in an Array with fixed length with split method (separated by spaces) where profanityString is the scanner input:

String[] profanityWords = profanityString.split(" ");

**Features of the class ArrayList:**

* It increases its capacity as necessary
* It keeps a private count:
  + size() accessor method
  + It keeps the objects in the order they were added
* Details of how all this is done are hidden
* We cannot store primitive datatypes in ArrayList, it can only store objects or ‘non-primitive data types’ such as Strings and Classes.

**Methods of ArrayList:** add, get, remove and size.

You can add elements to the list by using the add. keyword:

**Directly**

nameOfNewList.add("example"); //Adding element to nameOfNewList

**As a method**

public void addFile(String filename){ //Defining methods

ArrayListExample.add(filename); }

You can retrieve the size of the ArrayList using the get. keyword:

**As a method**

public int getNumberOfFiles() {

return files.size(); }

**Index numbering of ArrayList:** ArrayLists use integers to index collections. The ArrayLists are containing their elements in an index, which can be used to retrieving the elements of the list eg.:

public void listFile(int index) {

if(index >= 0 && index < files.size()) {

String filename = files.get(index);

System.out.println(filename);

} else {

Sysout.println(“This is not a valid index”); }}

**Review collections / ArrayLists**

* Collections allow an arbitrary number of objects to be stored
* Class libraries usually contain tried-and-tested collection classes
* We have used the ArrayList class from the java.util package
* Items may be added and removed
* Each item has an index
* Index values may change if items are removed (or further items added)
* The main ArrayList methods are add, get ,remove and size
* ArrayList is a parameterized or generic type

**Loops and iterations**

**Iterations and loops:** We often want to perform some actions an arbitrary number of times in a list. For example printing some file names in a music organizer or finding out how many are stored. For this we can use loop statements: d

**The for each-loop**

public String toString(){

String ForestStart = "Forest(";

String ForestMiddle = ""; //My empty string that I build on in my for each loop

String ForestEnd = ")";

for (Tree tree : trees){ //I name the elements in my collection 'trees' as 'tree'

ForestMiddle += tree;

//I initialise my new String element for each tree to the empty string variable

}

return ForestStart + ForestMiddle + ForestEnd;

}

* The loop body gets executed once for each element in the collection.
* The element variable takes the value of each element of the collection in turn.
* Easy to get right; no indexing errors.
* Iterates over all elements
* Works on all collections; even those that are not index based

**For each loop example 2**

@Override

public void display(){

String noComma = "";

super.display();

System.out.print(" { ");

for (String topping : toppings){

System.out.print(noComma);

System.out.print(topping);

noComma = ", ";

}

System.out.print(" } ");

} //output: 679 kr pepsi { pepperoni, salsa, meatballs }

**For each loop example 3 - using display method of another class**

public void display(){

for (Food orders : ordered){

orders.display();

System.out.println();

}} //Output: 9432 kr Laks

679 kr Speciale { pepperoni, salsa, meatballs }

**The while-loop**

int number = 0;

while (number <= 30) {

System.out.println(number);

number = number + 2;

}

* A while -loop doesn’t need a collection. Only a loop condition and a loop body. Example above to: print the numbers 0, 2, 4, . . . , 30
* We say: “While the condition is true, do...” More precisely:
  + 1. Check condition
  + 2. If false, stop; if true, execute the loop body, and continue from 1 again. Notice: After the loop, the condition is false.
* Much more general (works for other tasks than looping a collection)
* Can skip elements
* Too easy to create an infinite loop

**Null:** null is a special value for object type variables that mean “points at nothing”

### Concepts chapter 4.1 - 4.8: grouping objects

* **Collection:** A collection object can store an arbitrary number of other objects.
* **Loop:** A loop can be used to execute a block of statements repeatedly without having to write them multiple times.
* **Iterator:** Is an object that provides the functionality to iterate over all elements of a collection.
* **Null:** The Java reserved word null is used to mean “no object” when an object variable is not currently referring to a particular object. A field that has not explicitly been initialized will contain the value null by default.

## Lecture 5, chapter 7 - Arrays and for-loops

**Arrays** are special objects that can store other objects of the same type. They have a fixed length (number of elements), specified upon creation

* Arrays do not have useful method themselves (just like primitive types)
* Methods are provided by other classes; e.g., java.util.Arrays

Example:

public class ArrayOfTenInts {

int e0, e1, e2, e3, e4, e5, e6, e7, e8, e9; }

**Primitive types:** Includes byte, short, int, long, float, double, boolean and char

**Non-primitive data types:** String, Arrays and Classes

**Difference between ArrayLists and Arrays:** Array is a fixed length data structure whereas ArrayList is a variable length Collection class. We cannot change the length of an array once created in Java but ArrayList can be changed. We cannot store primitives in ArrayList, it can only store objects. But Array can contain both primitives and objects in Java.

|  |  |
| --- | --- |
| **ArrayList** | **Array** |
| * An arbitrary number of elements the same type. * A variable length Collection class. * Example type: ArrayList<String> * Example creation: new ArrayList<String>(). * Can be changed after creation. * Can only store objects. | * A fixed number of elements of the same type. * A fixed length data structure: * Example type: String[] * Example creation: new String[12] * Cannot be changed after creation. * Can store primitive data types and objects. |

**Arrays**

**Creating an Array:** We don’t specify the size of the Array-collection when defining it.

public class LogAnalyzer {

private int[] hourCounts;

private LogfileReader reader }

We only specify the size of the creation when we are creating the object. In this case, 24.

public LogAnalyzer() {

hourCounts = new int[24];

reader = new LogfileReader();

}

**Declaration**:

int[] hourCounts;

String[] names;

**Creation:**

hourCounts = new int[24];

numbers = new int[]{ 25, 42, 3 };

int[] results = { a + b, a \* b, Math.max(a, b) };

**Use:**

hourCounts[i] = 0;

hourCounts[i]++;

System.out.println(hourCounts[i]);

**Methods of the Array:**

* copyOf for creating a new array with contents from an old
* fill for copying the same value to all entries of an array
* sort for sorting an array
* binarySearch for searching in a sorted array
* toString for converting the array to a printable string

**Recap - Arrays**

* Put bluntly: Software Developers don’t use arrays; Programmers do.
* Arrays are efficient, but too low level for most uses
* Arrays only really show up inside data structures or old libraries
* There are typically more suitable and flexible collections to use instead (B&K ch 6)
* You will need them in the spring in Algorithms and Data Structures
* In some cases, they can increase readability, since a[i][j] is easier on the eyes

than a.get(i).get(j)

**The for-loop**

**Creation:** Gives us the option of looping over an index. It is an iterative control structure that is often used when an index variable is required to select consecutive elements from a collection, such as ArrayList or array.

for (int i = 0 ; i < files.length ; i++) {

System.out.println(files[i]);

}

**It means**

for (initialization ; condition ; post-body action) {

//body

}

### Concepts chapter 7

* **Array:** An array is a special type of collection that can store a fixed (10) number of elements such as objects and primitive data types.
* **ArrayList:** A variable length collection class that can store objects.
* **For loop:** An iterative control structure that is often used when an index variable is required to select consecutive elements from a collection, such as ArrayList or array.

## Lecture 6, chapter 4.8 - 4.15 - Iterators

**Arrays:** With Arrays, we can skip the constructor and declared the Array directly as fields:

**Fields**

public class ArrayListOfStrings {

private String[] data;

private int size;

**Declaration**

public ArrayListOfStrings {

data = new String[10];

size = 0;

**Itterator:** An iterator is a non-tangible object that only once iterates over a collection. To use the iterator, import it from: import java.util.Iterator; An iterator is an object that keeps track of where and how far we have gotten in what elements we have returned.

An iterator works well with a for loop, because you yourself define an iterator in the loop: *i++;*

This works well with the for loop because it loops through a collection and can do a given task at each iteration.

An iterator is also an interface: An interface is a list of methods that a class can claim it implements.

**Itterator methods:**

Collections have an iterator() method, which returns an Iterator object.

To loop through a collection, use the *hasNext()* and *next()* methods of the Iterator: Eg.:

while(it.hasNext()) {

System.out.println(it.next());

}

**The scanner:** The scanner is an iterator<String> class. It can take an input of words and return another String of words that has gone through your program. Eg:

Scanner scanner = new Scanner("dog cat bird");

String animal1 = scanner.next(); // "dog"

String animal2 = scanner.next(); // "cat"

String animal3 = scanner.next(); // "bird"

boolean more = scanner.hasNext(); // false

We can delete and remove elements in a collection using an iterator and a for-loop. Eg:

**Food loop**

for (int i = 0 ; i < list.length() ; i++) {

String elm = list.get(i);

if (isBad(elm)) {

list.remove(i);

I--; }

}

### Concepts chapter 4.8 - 4.15

* **Collection:** A collection object can store an arbitrary number of other objects.
* **Loop:** A loop can be used to execute a block of statements repeatedly without having to write them multiple times.
* **Iterator:** Is an object that provides the functionality to iterate over all elements of a collection.
* **Null:** The Java reserved word null is used to mean “no object” when an object variable is not currently referring to a particular object. A field that has not explicitly been initialized will contain the value null by default.

## Lecture 7, chapter 6 - Libraries, Sets and Maps

**Java library:** A java library contains information on all standard classes that can be used in Java. You just import the classes in the top of your program; Most of what we are using come from the [java.util](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-summary.html) package: <https://docs.oracle.com/en/java/javase/11/docs/api/>

import java.util.\*; //Imports all classes

import java.util.Scanner;

import java.util.Arrays;

import java.util.HashMap;

import java.util.ArrayList;

**HashMaps**

**Maps:** HashMap stores items in "key/value" pairs, and you can access them by an index of another type (e.g. a String). So for example, we can use the Map<String, Integer> to store the key “Bob” with the value 23. Principle is; put(K key, V value).

**Creation**

HashMap<String, Object> hash = new HashMap<>();

**Use**

put("Alice", 24) pairs the String as the key with the value 24.

hash.put("Alice", 24);

if (hash.containsKey("Alice")){

System.out.println(hash);

}

**Use with a for each loop**

for (String swearWord : profanityWords) {

wordsMap.put(swearWord, translation(swearWord)); }

**HashMap<String,String>**

Map<K,V> : mapping or table of Keys mapped to Values

* The keys have type K, and the values of type V
* The method put(k, v) associates value v with key k
* The method get(k) retrieves the value associated with the key k (or null , if no value is associated)
* This data structure is the reason I said that Software Developers don’t need arrays
* Developers of data structures (like HashMap ) still needs arrays
* An array is analogous to a Map that can only handle int keys, and has fixed size.

public String generateResponse(HashSet<String> words) {

for (String word : words) {

String response = responseMap.get(word);

if (response != null) {

return response; }

}

return pickDefaultResponse();

}

**HashMaps methods:** .hashCode() and .equals(.)

* All standard library classes have sensible .equals and .hashCode
* If two objects are .equals , then they must have the same .hashCode()
* HashMap uses .hashCode() and .equals(...) to compare keys
* In your classes .equals and .hashCode by default behave like ==
* If you change .equals , you must also change .hashCode

**Sets:** The set interface, for example a **HashSet**, present in the java.util package and extends the Collection interface is an unordered collection of unique objects. It is like a set of objects like a set of numbers in math. It has a feature that restricts the insertion of the duplicate elements.

* Set means there are no duplicates
* A HashSet<V> has values of type V
* The method add(v) add the value v, if it isn’t already there
* The method contains(v) returns true, if v is in the set, and false otherwise

**Methods:**  The method call split(" ") divides the string into words, separated by spaces

The result is a String -array. Eg.: "

**Methodcall**

“your software is too slow".split(" ")

**Returns the set**

{ "your", "software", "is", "too", "slow" }

String[] profanityWords = profanityString.split(" ");

**Difference between List, Set and Map**

|  |  |  |
| --- | --- | --- |
| **List** | **Set** | **Map** |
| List: Indexed, duplicates allowed.  ArrayList<E> (Insertion order) | Set: Unnumbered (means not indexed), duplicates not allowed.  HashSet<E> (Unordered)  TreeSet<E> (Ordered) | Map: associating values to keys. Duplicates not allowed among keys  HashMap<K, V> (Unordered)  TreeMap<K, V> (Ordered by key) |

**Static keyword:** Methods and fields can have the static keyword. Static means that it belongs to the class and not to each object. Static methods cannot access non-static fields directly. We have been using static variables already, without knowing it:

* System.out is a public static variable of the type PrintStream
* Math.sqrt(n) is a static method in the class Math

**Wrapper class:** Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects. This is useful when working with collections sucs as ArrayList, which cannot store primitive types, but only the objects of them.

Primitive data type // **int** = **Integer** // Wrapper class

byte = Byte.

char = Character...

**Reference type:** Reference data types in java are those which contain reference/address of dynamically created objects. These are not predefined like primitive data types. Eg: class types − This reference type points to an object of a class. array types − This reference type points to an array. interface types − points to an object of a class which implements an interface.

### Concepts chapter 6 - more sophisticated behaviour

* **Java library:** The java standard class library contains many useful classes part of the Java standard library classes. It is important to know how to use the library.
* **Library documentation:** Show details of all classes in the library and related methods.
* **Interface:** The interface of a class shows what the class does and how it can be used without showing the implementation.
* **Immutable:** An object is immutable if its contents or state cannot be changed after creation. Strings are examples of immutable objects.
* **Map:** A map is a collection that stores key/value pairs as entries. Values can be looked up by providing the key.
* **Set:** A set is a collection that stores each individual element at most once time - duplicates not allowed. It is an unordered collection.
* **Wrapper class:** Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
* **Reference type:** Reference data types in java are those which contain reference/address of dynamically created objects.
* **Autoboxing:** Performed automatically when a primitive type value is used in a context requiring a wrapper type.
* **Documentation:** The documentation of a class should be detailed enough for a programmer to use the class without having to read the source code.
* **Access modifier:** Defines the visibility of a field, constructor, or method. ***Public*** elements are accessible from inside the same class and from other classes; ***private*** elements are accessible only from within the same class. **Protected** are only visible for extended classes of a super class.
* **Information hiding:** Is a principle that states that internal details of a class’s implementation should be hidden from other classes. It ensures better modularization of an application.
* **Class variable, static variable:** Classes can have fields. These are known as class variables or static variables. Exactly one copy of a class variable exists of a class variable at all times, independent of the number of created instances.

## Lecture 8: Version Control and Build Systems

**Git - a Version Control System:** Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

**Gradle - Build system:** Gradle is a build automation tool for multi-language software development. It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing.

Build automation is the process of automating the creation of a software build and the associated processes including: compiling computer source code into binary code, packaging binary code, and running automated tests.

**J-unit - Testing software:** Is a testing environment that benefits big software projects. It is faster testing your own code with the main testing method. JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development.

## Lecture 9, chapter 8 - Class design

Software is corrected, maintained, adapted. The work is done by different people over decades. If software cannot be maintained, it will be thrown away.

**Cohesion**: A unit of code has high cohesion if it is responsible for a single, logical task. We aim for high cohesion. There needs a short and precise answer to the question: Why is this code here? Answer: It performs this single task…

**High cohesion for a unit:** High cohesion gives increased readability, Each unit should have one responsibility.Principles are that each unit should have one responsibility. Responsibilities are:

* + Knowing things: Fields
  + Doing things: Methods
  + Data tracking: Classes
* Each class has the responsibility of keeping track of its own data. So for example, the Room class in the ‘game exercise’ has to keep track of which exits leads to different rooms, and therefore also Rooms responsibility to list them.

**Responsibility driven design:** Code duplication, when you have the same code multiple times, is bad practise, because it creates a number of problems:

* Doubles up on typing and increases size of codebase.
* Makes it harder to fix bugs and modify, as code has to be changed in both places.
* If only one is modified, you will have inconsistent behavior issues.

**Solution:** Declare a helper method for the shared code. (Put the code in its own method). Call the helper method from both places.

**Coupling:** Classes are tightly coupled if they depend closely on the implementation details of each other. We aim for loose coupling.

* With loose coupling the classes don’t know to much about eachother - example by keeping the fields private - and that makes it easier to modify one class without fucking up the other one.

**Refactoring:** Means restructuring code. Including class design.

* It is a way to pay off technical debt. Means, if a code has been written as a quick and dirty solution (lappeløsning) then we need to refactor the code later, so that it follows responsibility driven design.

**Process**

* Class design is updated
* Old functionality is restored (When the code does the same thing as before with the new design, then and only then are you allowed to… ->)
* New functionality is added (off limit til previous steps is achieved)

**Switch statement**

int day = 2;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

} //Prints Tuesday

### Concepts chapter 8 - designing classes

* **Coupling:** The term coupling describes the interconnectedness of classes. We strive for loose coupling in a system—that is, a system where each class is largely independent and communicates with other classes via a small, well-defined interface.
* **Cohesion:** The term cohesion describes how well a unit of code maps to a logical task or entity. In a highly cohesive system, each unit of code (method, class, or module) is responsible for a well-defined task or entity. Good class design exhibits a high degree of cohesion.
* **Code duplication**: Code duplication (having the same segment of code in an application more than once) is a sign of bad design. It should be avoided.
* **Encapsulation:** Proper encapsulation in classes reduces coupling and thus leads to a better design.
* **Responsibility driven design:** Responsibility-driven design is the process of designing classes by assigning well-defined responsibilities to each class. This process can be used to determine which class should implement which part of an application function.
* **Localizing change:** One of the main goals of a good class design is that of localizing change: making changes to one class should have minimal effects on other classes.
* **Method cohesion:** A cohesive method is responsible for one single task.
* **Class cohesion:** A cohesive class represents one, well defined entity.
* **Refactoring** is the activity of restructuring an existing design to maintain a good class design when the application is modified or extended.
* **Switch statement:** You can use the switch statement to select one of many code blocks to be executed. A switch statement selects a sequence of statements for execution from multiple different options.

## Lecture 10, chapter 9 - Software testing

*We test to prevent bugs. A software bug is an error, flaw or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. ... Bugs can trigger errors that may have ripple effects.*

### Different types of errors

* **Syntax errors:** The program may be syntactically ill-formed.
  + Written wrong.
  + Forgetting parenthesis and so on.
* **Semantic errors:** Improper use of program statements. Using the wrong word in the wrong place. Eg:
  + Accessing a non existing element in array (out of bound access)
  + Method calls with wrong type of argument

*The first two types of errors are often caught by the compiler.*

* **Logical errors:** The program is not behaving as expected. Logical errors are mostly caught by testing. They are more difficult to find. Eg.
  + Given two Strings x and y the program is supposed to return xy but it returns yx

**Different types of testing**

* **Unit testing:** Consists of testing individual methods and functions of the classes and components. Quite an easy and low level way of testing - doesn’t require much.
* **Integrations tests:** Testing if different modules and services work together. Usually more expensive and resource consuming.
* **System tests:** Performing tests on a complete, integrated system.
* **Acceptance tests**: Consist of formal test executions to verify if a system satisfies its business requirements
* **Regression tests**: Regression Testing is a type of testing that is done to verify that a code change in the software does not impact the existing functionality of the product.
* **Performance tests**: It checks the behavior of the system when it is under significant load.

**White box testing:** Test involves checks on the internal operations of the system (assumes that the path of the logic in a unit or program is known). Testing is based on coverage of code statements, branches, paths or conditions. It is considered as low-level testing

**Black box testing:** Tester has no information about the details of internal working of the system. The focus in black box testing is on the behavior of the system. It involves testing from an external or end-user perspective

**Unit testing:** “A unit test is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit”.

**Main testing/local testing through the main method**: It is a way to test your code's sub-parts and making sure you understand what is actually going on. Easier for our intended use on small software projects.

* Ask: What do i expect my answer to be.
* Find out what is the answer actually?

### Concepts chapter 9 - well behaved objects (testing)

* **Testing:** Testing is a way to find out if a piece of code (a method, class or program) produces the intended behavior.
* **Debugging:** Debugging is the attempt to to pinpoint and fix the source of an error.
* **Unit testing:** Refers to tests of the individual parts of the application, such as methods and classes.
* **Positive testing:** Is testing of cases where you expect them to succeed.
* **Negative testing:** Is testing of cases where you expect them to fail.
* **Test automation:** Simplifies the process of regression testing.
* **Assertion:** An assertion is a statement of a fact that should be true in normal program execution. If the condition is false, we say that the assertion fails. We can use assertions to state our assumptions explicitly and to detect programming errors more easily.
* **Fixture:** A set of objects in a defined state that serves as a basis for unit tests.
* **Walkthrough**: A walkthrough is an activity of working through a segment of code line by line to observe changes of state and behavior of the application.

## Lecture 11, chapter 10 - Inheritance

To avoid duplicate code in two different classes that share some of the same elements, we can use inheritance and create a superclass / a parent class. It allows for a more maintainable and readable program.

Creating a superclass, we climb the abstraction ladder and define the parent of our subclasses. So for example the subclasses fox and rabbits can both be extended to the superclass Animal.

public class Fox extends Animal{ }

In animals, we describe the shared *fields* and *methods* of the subclasses fox and animal - only the ones they have in common. Subclasses *inherit* these classes from the superclass must implement all methods of its superclass, but the subclasses can have unique methods and fields themselves that separates them from other subclasses.

**Source code for the new subclasses:** You must call the constructor of the super class as the first thing in the subclasses, otherwise it will automatically be inserted with a call to the empty constructor. If the called constructor doesn’t exist, you get an error. You must insert the same type and parameter as the superclass in the sub-class constructor as well as the local parameter only applying for the individual subclass. Then you call the super and construct the value of the unique parameter below: Eg.

public Fox(String name, Boolean randomAge, Field field, Location location){

super(field, location); // the inherited constructor parameters

this.name = name; // the local/unique parameters

if(randomAge) { // the local/unique parameters

age = rand.nextInt(MAX\_AGE);

foodLevel = rand.nextInt(RABBIT\_FOOD\_VALUE);

}

else {

age = 0;

foodLevel = RABBIT\_FOOD\_VALUE;

} }

|  |  |
| --- | --- |
| Advantages of inheritance | Disadvantages of inheritance |
| * Avoid duplicate code (and its related problems) * Code reuse for future extensions. * Easier to maintain (only change one piece of code) * Easy to extends (e.g., adding a field language to Post, and all subclasses get it) | * It requires ”abstraction” * It requires planning * A bad class hierarchy is worse than no class hierarchy |

**Set name in a cubclas through a super class:**

public void setName(String name){ //Setting name through super class name field

super.name = name;

}

**Creating extended constructor in subclasses**

public Pizza(){

super("Pizza", 45);

toppings = new ArrayList<>();

}

**Creating extended methods in sub-classes**

@Override

public void display(){

String noComma = "";

super.display(); //here

System.out.print(" { ");

for (String topping : toppings){

System.out.print(noComma);

System.out.print(topping);

noComma = ", ";

}

System.out.print(" } ");

}

**Final inheritance / not covered in the book**: It means a method can avoid being overridden. The book only talks about final as a modifier for fields but:

* A method can prevent being overridden by being declared final :

class Cat {

public final void kill() { ... }

}

class HouseCat extends Cat {

public void kill() { sleepAndPurr(); } //cannot override kill

}

**Final class:** A class can prevent having subclasses, by being declared final:

public final class String {...

**Final field:** A field can prevent from changing by being declared final:

final private int limit;

**Substitution principle:** “An object of a subclass must always be usable anywhere an object of the superclass is expected.” Barbara Liskov, 1987. It means plainly: a subclass must honor the contract set by the superclass. It comes into play when we subtype; *You can use subtype objects wherever objects of its supertype are expected. This is known as substitution.*

**Subtyping:** Means to honor the class hierarchy. A variable ‘V’ of the Superclass Vehicle, can be equal to the object ‘C’ of the Subclass Car.

Vehicle v;

Car c;

Bicycle b;

c = new Car();

v = c; // ok (substitution principle)!

b = (Bicycle) c; // compile-time error!

b = (Bicycle) v; // runtime error!

**Methods of subtyping:** Using instance of, you can ask whether the type of an object is a subtype of something.

if (v instanceof Bicycle) { do something }

### Concepts chapter 10 - improving structure with inheritance

* **Inheritance:** Inheritance allows us to define one class as an extension of another.
* **Superclass:** A class that is extended by another class.
* **Subclass:** A class that inherits fields and methods from another superclass.
* **Inheritance hierarchy:** Classes that are linked through inheritance form an inheritance hierarchy.
* **Superclass constructor:** The constructor of a subclass must always invoke the constructor of its superclass as its first statement. If the source code does not include such a call, Java will attempt to insert a call automatically.
* **Reuse:** Inheritance allows us to reuse previously written classes in a new context.
* **Subtype**: As an analog to the class hierarchy, types form a type hierarchy. The type defined by a subclass definition is a subtype of the type of its superclass.
* **Variables and subtypes:** Variables may hold objects of their declared type of any other subtype of the same type.
* **Substitution:** You can use subtype objects wherever objects of a supertype are expected. This is known as substitution.
* **Object:** All classes with no explicit superclass have Object as their superclass.

## Lecture 12, chapter 11: More inheritance / Abstract methods and classes

**Static vs. dynamic type:**

|  |  |
| --- | --- |
| **Static type** | **Dynamic type** |
| * Types checked before run-time * The static type of a variable v is the type as declared in the source code in the variable declaration statement. * Static Type (aka compile-time type) can cause compile type errors * The declared type of a variable * Static type is often a superclass, as it stays the same. * The compiler checks the static types, and can cause a compile error if there are subtyping issues. | * Types checked on the fly, during execution * The dynamic type of a variable v is the type of the object that is currently stored in v. * Dynamic Type: (aka runtime type) can cause run-time errors * The actual class of the object at runtime * Dynamic type is often a subclass, as it can change. |

**Access modifiers:**

private < restricted < protected < public

**Private:** All fields should be private if at all possible, so each class only has access to it’s own fields.

**Restricted:** Is used in *abstract* classes to define that a class cannot be used to create objects (to access it, it must be inherited from another class).

**Protected:** If fields become protected, they are visible in all subclasses of a superclass

**Public:** Fields can be viewed by all classes.

**Overriding:** In any object-oriented programming language, overriding allows a subclass or to provide a specific implementation of a method that is already provided by one of its superclasses. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

* So you define display() in the superclass
* And then Override display() in the subclass. Eg:

@Override //How to override

public void display(){

String string = "";

String comma = "";

super.display(); //Calling the superclass method as first thing

System.out.print(" { "); //Start of call to subclass’ unique addition the method

for (int i = 0; i < toppings.size(); i++){

System.out.print(comma);

string = toppings.get(i);

comma = ", ";

System.out.print(string);

}

System.out.print(" }");

### Concepts chapter 11 - more about inheritance

**Static type:** The static type of a variable v is the type as declared in the source code in the variable declaration statement.

**Dynamic type:** The dynamic type of a variable v is the type of the object that is currently stored in v.

**Overriding:** A subclass can override a method implementation. To do this, the subclass declares a method with the same name as the superclass, but with a different method body. The overriding method takes precedence for method calls on subclass objects.

**Method polymorphism:** Method calls in java are polymorphic. The same method call may at different times invoke different methods, depending on the dynamic type of the variable used to make the call.

**toString:** Every object in Java has a toString method that can be used to return a string representation of itself. Typically, to make it useful, a class should override this method.

**Protected:** Declaring a field or a method with the *protected* keyword allows direct access to it from (direct or indirect) subclasses.

## Lecture 13 - Chapter 12: Interfaces and Abstract Classes

**Simulation:** Programs regularly used to simulate real-world

Activities, such as weather forecasting, atomic bomb effects, economic models, traffic predictions and so on…

Simulations always depends on modelling of the domain

* The real world is too complex to simulate exactly
* Models depend on abstractions of reality
* The result of the simulation only reflects reality to the extent that the model does. Like in the fox and rabbit example, simplifications has been made, as foxes and rabbits has the same sex.

Choices of the simplification and what to leave out is made by the domain expert - example a biologist. So the more parameters taken into account, the more real is the simulation. But often is a simpler simulation better or just as good for the intended purpose, as more perimeters can also obscure the simulation.

**Abstract:** The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class**: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method**: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods. You can for example declare the class Animal to be abstract and the method act() to be abstract eg.:

abstract class Animal {

public abstract void act(){

//no method body for abstract methods }

}

* An abstract class cannot be instantiated directly, only through subclasses
* Abstract methods must be overridden by subclasses

**Multiple inheritance:** Having a class inherit directly from multiple ancestors.

Each language has its own rules, Java for example forbids it for classes but allows it for interfaces.

### Concepts chapter 12 - Further abstraction techniques

* **Abstract method:** An abstract method definition consists of a method header without a method body. It is marked with the keyword abstract.
* **Abstract class:** An abstract class is a class that is not intended for creating instances. It’s purpose is to serve as a superclass for other classes. Abstract classes may contain abstract methods.
* **Abstract subclass:** For a subclass of an abstract class to become concrete, it must provide implementation for all inherited abstract methods. Otherwise it will itself be abstract.
* **Super class method calls:** Calls to non-private instance methods from within a superclass are always evaluated in the wider context of the object's dynamic type.
* **Multiple inheritance:** When a class inherits from more than one superclass it is called multiple inheritance.
* **Interface:** A java interface is a specification of a type (in the form of type name and set of methods). It often does not provide an implementation for most of its methods.

## Lecture 14 - Chapter 14: Exceptions and File I/O

### Defensive programming

**Client-server architecture**

One or more clients use the same server.

* A server maintains an internal state.
* **Server**: *Passive* (responsive). A server is a computer that provides data to other computers. In computing, a server is a piece of computer hardware or software that provides functionality for other programs or devices, called "clients". This architecture is called the client–server model. A server can be a web-server.
* **Clients**: *Active* (requester)
  + Requests the server to act. For example, show me this website.
  + Often asks server to change its state
  + Can ask server about data based on its state
  + My browser is the clients. Ex my laptop.

A server can also be a client, if it requests to fetch data from a database. Then the server is a client of the database, and the database is then a server.



**Exceptions:** An exception is an object representing details of program failure. An exception is thrown to indicate that a failure has occurred.

We can create exception objects to avoid different kinds of exceptions. For example Illegal argument exceptions. Out of bound exception, and so on… We can ‘catch’ exceptions, and tell the program to return something else in case of an exception happening. Eg.:

First create an Exception object. Then throw the objectt:

Exception E = newIllegalArgumentException("Null passed to...");

throw E;

**Final block:** The finally block is a way to end a list of exceptions. It is always executed!

**Assertions:** Assertions are explicit checks for implicit assumptions:

* If the asserted expression is false, an AssertionError is thrown!
* Assertions are only checked during development
* If an assertion fails, we know there is a bug in the code
* Think of these as inline tests

**File I/O:** Java IO is an API that comes with Java which is targeted at reading and writing data (input and output). Most applications need to process some input and produce some output based on that input. For instance, read data from a file or over network, and write to a file or write a response back over the network.

import java.io.\*;

### Concepts chapter 14 - Handling errors

* **Exception:** An exception is an object representing details of a program failure. An exception is thrown to indicate that a failure has occurred.
* **Unchecked exception:** Is a type of exception whose use will not require checks from the compiler.
* **Checked exception:** A checked exception is a type of exception whose use will require extra checks from the compiler. In particular, checked exceptions in Java require the use of throw clauses and try statements.
* **Exception handler:** Program code that protects statements in which an exception might be thrown is called an exception handler. It provides reporting and/or recovery code should one arise.
* **Assertion:** An assertion is a statement of a fact that should be true in normal program execution. If the condition is false, we say that the assertion fails. We can use assertions to state our assumptions explicitly and to detect programming errors more easily.
* **Serialisation:** Serialization allows whole objects, and object hierarchies, to be read and written in a single operation. Every object involved must be from a class that implements the Serializable interface.

## Complete concept list

***abstract class*** An abstract class is a class that is not intended for creating instances. Its pur- pose is to serve as a superclass for other classes. Abstract classes may contain abstract methods.

***abstract method*** An abstract method definition consists of a method header without a method body. It is marked with the keyword abstract.

***abstract subclass*** For a subclass of an abstract class to become concrete, it must provide implementations for all inherited abstract methods. Otherwise, it will itself be abstract.

***abstraction*** Abstraction is the ability to ignore details of parts, to focus attention on a higher level of a problem.

***access modifier*** Access modifiers define the visibility of a field, constructor, or method. Public elements are accessible from inside the same class and from other classes; private elements are accessible only from within the same class.

***accessor method*** Accessor methods return information about the state of an object. ***anonymous***

***inner classes*** Anonymous inner classes are a useful construct for implementing event listeners that are not functional interfaces.

***array*** An array is a special type of collection that can store a fixed number of elements.

***assertion*** An assertion is a statement of a fact that should be true in normal program execution. If the condition is false, we say that the assertion fails. We can use assertions to state our assumptions explicitly and to detect programming errors more easily.

***assignment statement*** Assignment statements store the value represented by the right- hand side of the statement in the variable named on the left.

***autoboxing*** Autoboxing is performed automatically when a primitive-type value is used in a context requiring a wrapper type.

***boolean expression*** Boolean expressions have only two possible values: true and false. They are commonly found controlling the choice between the two paths through a condi- tional statement.

***checked exception*** A checked exception is a type of exception whose use will require extra checks from the compiler. In particular, checked exceptions in Java require the use of throws clauses and try statements.

***class*** Objects are created from classes. The class describes the kind of object; the objects represent individual instances of the class. A class name can be used as the type for a variable. Variables that have a class as their type can store objects of that class.

***class cohesion*** A cohesive class represents one well-defined entity.

***class diagram*** The class diagram shows the classes of an application and the relation- ships between them. It gives information about the source code and presents the static view of a program.

***class variable, static variable*** Classes can have fields. These are known as class variables or static variables. Exactly one copy exists of a class variable at all times, independent of the number of created instances.

***code duplication*** Code duplication (having the same segment of code in an application more than once) is a sign of bad design. It should be avoided.

***cohesion*** The term cohesion describes how well a unit of code maps to a logical task or entity. In a highly cohesive system, each unit of code (method, class, or module) is respon- sible for a well-defined task or entity. Good class design exhibits a high degree of cohesion.

***collection*** A collection object can store an arbitrary number of other objects

***comment*** Comments are inserted into the source code of a class to provide explanations to human readers. They have no effect on the functionality of the class.

***components*** A GUI is built by arranging components on screen. Components are repre- sented by objects.

***conditional statement*** A conditional statement takes one of two possible actions based upon the result of a test.

***constructor*** Constructors allow each object to be set up properly when it is first created.

***content pane, menu bar*** Components are placed in a frame by adding them to the frame’s menu bar, or content pane.

***coupling*** The term coupling describes the interconnectedness of classes. We strive for loose coupling in a system—that is, a system where each class is largely independent and communicates with other classes via a small, well-defined interface.

***debugger*** A debugger is a software tool that helps in examining how an application executes. It can be used to find bugs.

***debugging*** Debugging is the attempt to pinpoint and fix the source of an error.

***design pattern*** A design pattern is a description of a common computing problem and a description of a small set of classes and their interaction structure that helps to solve that problem.

***documentation*** The documentation of a class should be detailed enough for other pro- grammers to use the class without the need to read the implementation.

***dynamic type*** *The dynamic type of a variable v is the type of the object that is currently* stored in v.

***Encapsulation*** *Proper encapsulation in classes reduces coupling and thus leads to a better* design.

***event listener*** An object can listen to component events by implementing an event- listener interface.

***exception*** An exception is an object representing details of a program failure. An exception is thrown to indicate that a failure has occurred

***exception handler*** Program code that protects statements in which an exception might be thrown is called an exception handler. It provides reporting and/or recovery code should one arise.

***external method call*** Methods can call methods of other objects using dot notation. This is called an external method call.

***Field*** *Fields store data for an object to use. Fields are also known as instance variables.*

***Filter*** *We can filter a stream to select only specific elements.*

***Fixture*** *A fixture is a set of objects in a defined state that serves as a basis for unit tests.*

***For loop*** *an iterative control structure that is often used when an index variable is required to select consecutive elements from a collection, such as an ArrayList or an array.*

***functional style*** In the functional style of collection processing, we do not retrieve each element to operate on it. Instead, we pass a code segment to the collection to be applied to each element.

***image format*** Images can be stored in different formats. The differences primarily affect file size and the quality of the image.

***immutable*** An object is said to be immutable if its contents or state cannot be changed once it has been created. Strings are an example of immutable objects.

***implementation*** The complete source code that defines a class is called the implementa- tion of that class.

***information hiding*** Information hiding is a principle that states that internal details of a class’s implementation should be hidden from other classes. It ensures better modulariza- tion of an application.

***inheritance hierarchy*** Classes that are linked through inheritance relationships form an inheritance hierarchy.

***inheritance*** Inheritance allows us to define one class as an extension of another.

***interface*** A Java interface is a specification of a type (in the form of a type name and a set of methods). It often does not provide an implementation for most of its methods.

***interface*** The interface of a class describes what a class does and how it can be used without showing the implementation.

***internal method call*** Methods can call other methods of the same class as part of their implementation. This is called an internal method call.

***iterator*** An iterator is an object that provides functionality to iterate over all elements of a collection.

***Java library*** The Java standard class library contains many classes that are very useful. It is important to know how to use the library.

***lambda*** A lambda is a segment of code that can be stored and executed later. ***layout*** Arranging the layout of components is achieved by using layout managers.

***library documentation*** The Java class library documentation shows details about all classes in the library. Using this documentation is essential in order to make good use of library classes.

***lifetime*** The lifetime of a variable describes how long the variable continues to exist before it is destroyed.

***local variable*** A local variable is a variable declared and used within a single method. Its scope and lifetime are limited to that of the method.

***localizing change*** One of the main goals of a good class design is that of localizing change: making changes to one class should have minimal effects on other classes.

***loop*** A loop can be used to execute a block of statements repeatedly without having to write them multiple times

***map*** A map is a collection that stores key/value pairs as entries. Values can be looked up by providing the key.

***map*** We can map a stream to a new stream, where each element of the original stream is replaced with a new element derived from the original

***menu bar, content pane*** Components are placed in a frame by adding them to the frame’s menu bar or content pane.

***method calling*** Objects can communicate by calling each other’s methods.

***method cohesion*** A cohesive method is responsible for one, and only one, well-defined task

***method polymorphism*** Method calls in Java are polymorphic. The same method call may at different times invoke different methods, depending on the dynamic type of the variable used to make that call

***method*** We can communicate with objects by invoking methods on them. ***modularization*** Modularization is the process of dividing a whole into well-defined parts that can be built and examined separately and that interact in well-defined ways.

***multiple inheritance*** A situation in which a class inherits from more than one superclass is called multiple inheritance.

***multiple instances*** Many similar objects can be created from a single class. ***mutator method*** Mutator methods change the state of an object.

***negative testing*** Negative testing is the testing of cases that are expected to fail.

***null*** The Java reserved word null is used to mean “no object” when an object variable is not currently referring to a particular object. A field that has not explicitly been initialized will contain the value null by default.

***object*** All classes with no explicit superclass have Object as their superclass. Java objects model objects from a problem domain.

***object creation*** Objects can create other objects, using the new operator. Some objects cannot be constructed unless extra information is provided.

***object diagram*** The object diagram shows the objects and their relationships at one moment in time during the execution of an application. It gives information about objects at runtime and presents the dynamic view of a program

***object references*** Variables of object types store references to objects.

***overloading*** A class may contain more than one constructor, or more than one method of the same name, as long as each has a distinctive set of parameter types.

***overriding*** A subclass can override a method implementation. To do this, the subclass declares a method with the same signature as the superclass, but with a different method body. The overriding method takes precedence for method calls on subclass objects.

***parameter*** Methods can have parameters to provide additional information for a task. ***pipeline*** A pipeline is the combination of two or more stream functions in a chain, where each function is applied in turns

***positive testing*** Positive testing is the testing of cases that are expected to succeed.

***primitive type*** The primitive types in Java are the non-object types. Types such as int, boolean, char, double, and long are the most common primitive types. Primitive types have no methods.

***println*** The method System.out.println prints its parameter to the text terminal. ***protected*** Declaring a field or a method protected allows direct access to it from (direct or indirect) subclasses.

***prototyping*** Prototyping is the construction of a partially working system in which some functions of the application are simulated. It serves to provide an understanding early in the development process of how the system will work.

***reduce*** We can reduce a stream; reducing means to apply a function that takes a whole stream and delivers a single result.

***refactoring*** Refactoring is the activity of restructuring an existing design to maintain a good class design when the application is modified or extended.

***responsibility-driven design*** Responsibility-driven design is the process of designing classes by assigning well-defined responsibilities to each class. This process can be used to determine which class should implement which part of an application function.

***result*** Methods may return information about an object via a return value. ***reuse*** Inheritance allows us to reuse previously written classes in a new context

***scenarios*** Scenarios (also known as “use cases”) can be used to get an understanding of the interactions in a system.

***scope*** The scope of a variable defines the section of source code from which the variable can be accessed.

***serialization*** Serialization allows whole objects, and object hierarchies, to be read and written in a single operation. Every object involved must be from a class that implements the Serializable interface.

***set*** A set is a collection that stores each individual element at most once. It does not maintain any specific order.

***signature*** The method name and the parameter types of a method are called its signature. They provide the information needed to invoke that method.

***source code*** The source code of a class determines the structure and behavior (the fields and methods) of each of the objects of that class.

**state** Objects have state. The state is represented by storing values in fields

***static type*** The static type of a variable v is the type as declared in the source code in the variable declaration statement.

***static variable, class variable*** Classes can have fields. These are known as class variables or static variables. Exactly one copy exists of a class variable at all times, independent of the number of created instances.

***streams*** Streams unify the processing of elements of a collection and other sets of data. A stream provides useful methods to manipulate these data sets.

***subclass*** A subclass is a class that extends (inherits from) another class. It inherits all fields and methods from its superclass.

***substitution*** Subtype objects may be used wherever objects of a supertype are expected. This is known as substitution.

***subtype*** As an analog to the class hierarchy, types form a type hierarchy. The type defined by a subclass definition is a subtype of the type of its superclass.

***superclass*** A superclass is a class that is extended by another class.

***superclass constructor*** The constructor of a subclass must always invoke the constructor of its superclass as its first statement. If the source code does not include such a call, Java will attempt to insert a call automatically.

***superclass method calls*** Calls to non-private instance methods from within a superclass are always evaluated in the wider context of the object’s dynamic type.

***switch statement*** A switch statement selects a sequence of statements for execution from multiple different options.

***testing*** Testing is the activity of finding out whether a piece of code (a method, class, or program) produces the intended behavior.

***toString*** Every object in Java has a toString method that can be used to return a string representation of itself. Typically, to make it useful, a class should override this method.

***type*** Parameters have types. The type defines what kinds of values a parameter can take. ***unchecked exception*** An unchecked exception is a type of exception whose use will not require checks from the compiler.

***use cases*** Use cases (also known as “scenarios”) can be used to get an understanding of the interactions in a system usually do something if we invoke a method.

***variables and subtypes*** Variables may hold objects of their declared type or of any sub- type of their declared type.

***verb/noun*** Classes in a system roughly correspond to nouns in the system’s description. Methods correspond to verbs.

***walkthrough*** A walkthrough is an activity of working through a segment of code line by line while observing changes of state and other behavior of the application.

### Midterm forberedelse

A lot of multiple choice. Done through LearnIt. Then there will be a practical coding problem, that we will submit through codejudge. Might put a limit on number of times not to overload the system.

Multiple choice question. A statement and four answers… \\

Read the concept boxes and margins of the book and the summaries of each chapter very carefully! You might need to write a loop that print out such and such.

Begins 12:30. 1 hour duration. On sight test. Will try to give us the same facing as january exam - the format.

We can bring books and notes. We can fill our computer with as many notes as we like. We just can’t go online. Except for on learn-it and codejudge.

We solve the coding problem using vs-code.

* Mix of multiple choice
* Mix of coding -> More in the style of the latest;
  + step 1 – do this.
  + Step 2 – do this
  + Step 3 – do this.