Slides available at the end of the videos

**Folder for java:** /usr/lib/jvm/java-11-amazon-corretto

Is case sensitive, works with keywords

**class definition:**

*public class ClassName{*

*//class body is between curly braces*

*}*

*Public* is a access modifier, declaring what has access to the class defined. ClassName is to be unique.

Print text and variable:

System.out.println(“Lol = “ + number\_value)

Java programs are ran through the *main method,* with other methods also possible being defined.

Since Java is an OOP, we have the file named ClassName, the class ClassName defined inside this file and the main method defined inside *ClassName*.

**Method definition:**

*accessModifier static/dynamic returnType methodName(arguments){*

*// Body*

*}*

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

*}*

**Variables:**

All variables has a *type/data type* indicating what kind of data the variable should hold.

datatype variableName = valueoftypeDataType

int myNumber = 5; or int mynumber;

The **primitive** data types are **boolean, byte, char, short, int, long, float and double.**

***Int****: E*xists lower and upper ranges for the type. Accessed via integet.MIN\_VALUE/Integer.MAX\_VALUE. Will **overflow/underflow.** About -4billion to 4 billion, 32 bits

**Byte:** -128 to 127, 8 bits of memory

**Short: -**32768 to 32767, 16 bits

**long:** Has a width of 64, created by long myLongVal = 100L;. Long values without L are treated as integers

**Casting:**

Done via (newType) (variable)

**Float, double.** Double has double the precision. Extensive on a relative basis compared to whole numbers. Overflow still exists. Double are the default, have to explicitly state that it’s a float with an f at the end. Praxis is to use f and not cast, cleaner. **Using double** is what is used and is actually faster.

**Division:** Integers does whole number division, whereas the float/double holds enough precision for decimals

Does exist **BigDecimal** class where precise calculations are needed

**Char:** Holds one character, only ‘’ allowed, allows UNICODE by ‘\uXYZW’

**String:** String varName = “Hey there fucker!”; + is used for appending strings. They are immutable, so if they are altered with, copies, assignments and discardments are done. Not very efficient!! **Not a primitive type,** a class.

**Boolean:** true or false, boolean

Adding integers and strings results in string concatenation between the two.

Can use result++; <==> result = result +1;. Works with all operators

result+= 2; also works, same with all operators.

Does allow one liners in if statement

**Logical operators:**

**==, !=, >, <, >=, <=,** &&, ||, | is bitwise or, & is bitwise and, ! Is not

**Ternary operator, shorthand if statement**

varType varName = condition ? ValueTrue : ValueFalse;

Uses if(condition){

}else if (condition){

} else{

}

Variables created inside code blocks are “local” and can not be accessed outside the code block!

**Methods:**

Can not be created inside another method, create inside the class where they are supposed to be used. Can be defined below usage in code.

**Access flexible return name ( arguments)**

public static void methodName(argType argName){

body

}

Parameters are what is specified in the function definition, whereas arguments are the values you put in via usage. Always need to return data if void is not used as return

A method returning void is usually called a procedure.

**Diffmerge**

Used to check differences between files and file versions, also works for folders

sourcegear.com/diffmerge

**Method overloading**

Methods sharing **name** but taking different number arguments or combinations

Need unique method signatures. Return types does **not** change signatures

**Arbitrary number of arguments**

public static void main(int… args)

Where you can do args.length, for (in x: args) etc

Constants can be declared via, often with upper case letters for its name,

private static **final** type NAME = 123;. Errors when it is changed

**Control flow- while, for, do while, switch**

**switch:** handy where we need to test same variable for several numbers. BREAKS NEEDED

Is case sensitive!! Use string.toLowerCase()

switch(valueToTest){

case1 val1:

do stuff

break;

…

default:

do stuff

break;

can write case 3: case 4: case 5:

do stuff. In order to save space and code writing

When printing calculations involving doubles, it can be of use to do String.format(“%.2f”, number) to force simply 2 decimal points.

**for loop:**

for(**init**;**termination**;**increment**)

init is done once, termination is checked before entering loop every time, and increment is done once after each loop block is executed. Variable does not exist outside loop

**Does exist for each,** usage in arrays

**while and do while:**

while(condition){

do stuff that might alter the condition or break on a if statement etc

}

while loops **does not** guarantee that things are evaluated, since the expression might evaluate to false directly. In comes the **do while!**

do {

stuff that is executed at least once

} while(condition);

the **continue** keyword can be used in order to skip the rest of the code block in the loop, without exiting the loop. Starts over at the start of the loop

**Parse string as number**

Often values obtained might be interpreted as strings and need to be converted to be processed, in this case we might do int number = Integer.parseInt(numberAsAString);. Throws an error if the parsing is not successful, consider catching the exception. Same functionality exists for doubles

**Reading user input**

Scanner class, impport java.util.Scanner;

Scanner scanner = new Scanner(System.in); // Defines the scanner

String name = scanner.nextLine(); //Read whole lineli

Does exist nextInt() etc as well, converts input to int automatically

If trying to parse input as an int, and characters are passed, an error is thrown

Use boolean varName = scanner.HasNextInt(); which is true for valid value, false if value would have generated an error. Does not store the int, read it by scanner.nextInt();

scanner.close(); // to release reseources used

When reading numbers from user, we want to make sure the next read does not capture the next line character as a input, hence we do a empty scanner.nextLine(); on the row after

**Classes**

Consists of methods and fields attached to it. Can be seen as a *blueprint*!

Can be seen as a new primitive datatype on **STEROIDS**

Need a file name with ClassName.java with a class declaration inside with

public class ClassName{

accessModifier dataType fieldName;

//private often used, can be public, protected

public: unrestricted

private: no access from outside

protected: classes in the same package can access the variables

};

In main file, we can then do Car variableName = new Car(); //Default constructor?

Our class has some default methods to it.

For methods inside a class, ‘this’ keyword is used to reference fields that are attached to the class

setFieldName and getFieldName methods are called setters and getters

There can be several references to the same instance of a class! Variables are just references to memory!

**Constructors**

A method without return type, default constructor is created for you automatically

public ClassName(arguments){

Initialize all values from the arguments passed

}

Can “simulate” default values for all arguments by overriding the default constructor and adding a **this(arg1, arg2, …, argN);** which has to be from the top line in the default constructor, note that this call **only** works from inside constructors

Constructors and setters are 2 different schools, use constructors and save field values directly

**Constructor chaining** is when constructors call eachother and variables are only assigned in one

**Inheritance**

Idea is that several classes might share functionality and characteristics! Create a “base” class that other classes inherit things from. Also called a **super class**

We create the base class as usual, and then in the subclass we write

*public class SubClassName* ***extends*** *BaseClassName {*

*}*

Bear in mind that we need a constructor for the base class, and its arguments need to be parameters to the subclass constructor, and base class constructor is called via **super(arg1, ..);**

Add subclass specific fields to the subclass!!

All fields and methods in base class are available to subclass!

An identical method in the subclass **overrides** the method in the superclass, the method in the superclass is accessible via the super.methodName()

Often good idea to solely use methodName() in case you decide to override it in the future, because if super.methodName() is used, the override does not have any effect

The **“superconstructor”** must be the first thing that is called in the constructor for the subclass

**Overriding**

Only possible for instance methods. The annotation of using @Override is recommended since it will throw errors if the overriding is done incorrectly

Need same name, need broader access modifier.

Can not override final methods, able to access the overridden method via the super.MethodName()

**Covariant return types** allow for different return types in overriding methods in the subclass, lets us return either a parent or subclass as return type

**Static vs instance methods**

Static methods are methods that do not need the instance variables and methods, can not access instance methods and variables directly. **Class need not have been instantiated!!!**

Instance methods needs an instance of a class, can access instance methods/fields as well as static methods and variables directly

If the method you are writing needs to access instance variables and/or methods, it should probable be an instance method, otherwise static!

**Static vs instance variables**

**Static** variables are declared using the keyword *static,* every instance of the class shares the same static variable. Changes made will be seen by all instances of the class! Not used so often

**Instance** variables belong to the instance of a class, and each instance has its own copy

**Composition**

A relationship test of the kind “has a”, A car has wheels, wheels has rubber etc, building classes using other classes, wheels being an instance variable to car etc

Composing a class of other class objects as instance variables

If we only need an object for passing as parameter to another class creation, we can pass new ClassName(arg1, …, argN) as parameter and not store it to a variable

**Instance methods of the instance variables are available via getters**

Composition “has a” relationship whereas Inheritance “is a” relationship

**Can limit access** to the instance variables methods via access modifier. Or, we can hide the functionality to them by writing functions that access instance variables in “base” class

Either, we can let the main program **access the subcomponents via getters** and use the **subcomponents methods directly.** MyRoom.getBed().makeBed();

***ALTERNATIVELY***

We provide the functions we want the user to have access to in the main class, and call the functions from the instance fields directly. myRoom.makeBed() where makeBed(); calls bed.makeBed(); internally

**Encapsulation**

Restriction of access to objects, stopping other classes from accessing inner workings of a given class

We want to control how the class is used and make sure it is not used in a way different than it was supposed to. Also, changes in the class might be hard to cope with if not using encapsulation correctly.

**Polymorphism**

A method returning an object of base class can also return all of the subclasses

Given a class with a method called meth1, and subclasses to this class, some of which implement meth1, some overriding. Then, if we randomly generate objects, we don’t need to know at build time which object will be called since all have meth1 implemented!

Different functionality for different object created

**Arrays**

varType[] variableName = new int[10]; Array with 10 slots for int elements

0-indexed, accessed via variableName[index];

Can be assigned using int[] myIntArray = {1,2,3}; Kan endast göras vid initiering

can do variableName.length

Will raise outOfBoundIndexError if accessed incorrectly

**Value vs reference types**

Primitive types are stored by value, and objects are stored as value of the reference, hence a reference to the object

Assigning objects to one another using primitive types creates independent copies of the original object. Doing the same with objects without the new keyword will assign the same reference to the other object! Dangerous

Parameters can be de-referenced by parameterName = new int[] {val1, …, valn); values optional

Can convert an array to a string for printing, separating by commas, by Arrays.toString(arrayName)

**Arraylist**

If arrays are to be re-sized, one can make a copy of the original array, overwrite it with another size using new int[newSize], and then loop through the original array and assign the old values to the new array

*Arraylist* on the other hand can handle re-sizing for you. Here, the types need not be specified

Defined by

accessModifier ArrayList<typeToStore> name = new ArrayList<typeToStore>();

Need not to specify size at instantiation!

**Adding** elements by arrayListName.add(itemOfType);

Use arrayListName.size() for **length**.

**Access** elements by arrayListName.get(index);

**Update** items by name.set(index, newItem);

**Remove** items by name.remove(index);

**Lookup** by name.contains(item); returns Boolean

**Position finding** by name.indexOf(item); returns -1 if not found, else index

**Copying array** can be done by first creating a new one, and then doing

newArray.addAll(anotherArrayList); via a getter or something

Can also pass the anotherArrayList in creation

new ArrayList<Type>(**here)**

Might be good idea to remove the index reading from the user, and work with item names instead, and find that index. Put the right functionality in the right place

To create an array from an arrayList, we first initiate the array using size method

Then we assign values via arrayName = arrayListName.toArray();