# Object Orientation

*Introduction*

Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

# Agenda

# Object Orientation

- Object orientation is a *state of mind* used for software development.
  - Used in most phases, from analysis to implementation and on.
- It is a way to see and understand the inner architecture of the system.
- To *model* it, that is visualise, *UML* can be used.
  - Unified Modelling Language – a notation syntax for OO.
- This lecture will give an introduktion to object orientation and how it should be used.
- Coming lectures will try to deepen that understanding.

# A little history

- In 1968 it was evident that the industry was in what was called the “software crisis”.
  - The term was coined at a NATO conference in Garmisch, Germany.
- More software was needed, but the number of developers was low.
- Projects were
  - Over budget
  - Not according to the requirements
  - Of low quality
- Less than 20% of all started projects did according to plan.

# New was of thinking

- New ways of thinking and working on were needed to combat the crisis.
- From the software crisis came *Software Engineering*.
  - To work in a structured and methodically way with program development.
  - Often with a formal or mathematical foundation.
  - But it can also be a way of working.
- Several development models were developed, with different success.
  - This work is still ongoing and most prominent is the agile way of working today.

# Object oriented thinking

- The object oriented way of thinking has its roots around 1950 in the programming language LISP.
  - Not really object oriented, but the concepts where visible.
- The first real object oriented language was Simula 67.
  - A norwegian language by Ole-Johan Dahl and Kristen Nygaard.
  - Was a development from the non-object oriented Simula 1.
- The first language to fully build on object orientation was Smalltalk.
  - Developed at Xerox by Alan Kay and others during 1970s.

# Anthropomorphism

- Object orientation has its roots in programming, but has also risen from that.
- A foundation in object orientation is *anthropomorphism*.
  - The attribution of human properties to non-human objects.
  - From the Greek *antropos* (human) and *morfos* (form).
- In object orientation it is important to talk about *objects* as real entities with properties and behaviour.
- A **Person** can have properties like blue eyes, shoe size 42 and the name Charles-Clemens.
- Everything in an object oriented system should be seen as *parts* that *communicate* rather than functionality.

# Foundations of Object Orientation

# Definition

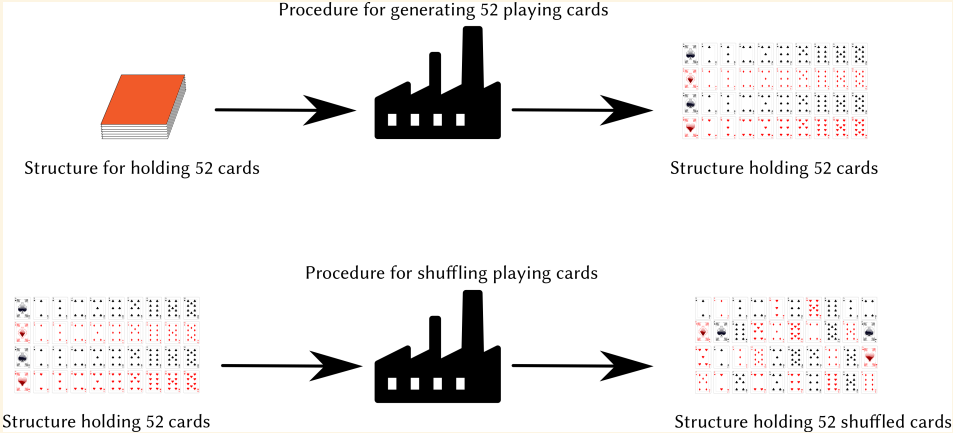
- The basis in object orientation is the *class* from which *objects* are created (and from where the name comes from).
- The definition of an object is:

  *A* discrete *entity with a well-defined boundary that encapsulates* state *and* behaviour*; an* instance *of a* class
- Covers the most important parts of object orientation:
  - Encapsulation
  - State
  - Behaviour
  - Instances

# Difference to traditional programming

- As stated, object orientation is “old”, but there are still many systems based on the traditional ways.
  - With “traditional” we mean *procedural* (or *imperative*) programming.
- Many languages like Python, C++ and JavaScript still allow for traditional programming.
- In those languages it is possible to create procedures (functions) that work like “black boxes” of functionality.
- Data is stored separately and sent to the procedures.
- The data used is set *globally* which means that all procedures have access to it.
  - Can give side effects on data as it can be modified by several procedures.

# Procedural Thinking



# What are objects

- Objects are a fundamental part of object orientation.
- An object is something concrete – something real.
- A cohesive package of *data* and *functionality*.
- Data is *encapsulated* and only reachable via operations.

# What objects have

- Each object has:
  - Identity
  - State
  - Behaviour
- The identity is *unique* for the object.
  - It does not have to be a specific *property* – each object *is* unique.
- State and behaviour that an object has, other objects are also able to have.
- An object is unique with a certain set of state and behaviour *at a specific time*.
- Varje enskilt objekt består av två delar:
  - Attribut values – data part, state
  - Operations – behaviour

# Additional example

- An Atari 1040 STf
- 8 MHz Motorola 68000 processor
- Manufactured between 1985 and 1993

- Questions:
  - What is the object?
  - How can we be sure of the identity of the object?
  - What state does it have?
  - What behaviour does it have?

# Identity

- What difference is there between different – but of the same kind – objects?

- Just as for physical objects, each object is unique.
- Artificial identifiers are possible, but not necessary!
  - Each computer has a unique serial number, but it does *not* have to be an attribute.

# State

- What states are possible for a computer?
  - On
  - Hibernating
  - Reading from disk
  - Running a program
- Things that changes the information of the object.
  - Information $\rightarrow$ data part $\rightarrow$ attributes of the object.
  - The thing that changes the state is called operations (more later).
- Each object's state is also defined by its relations.
  - What other objects does it need?
  - How many of them?

# State and relations of objects

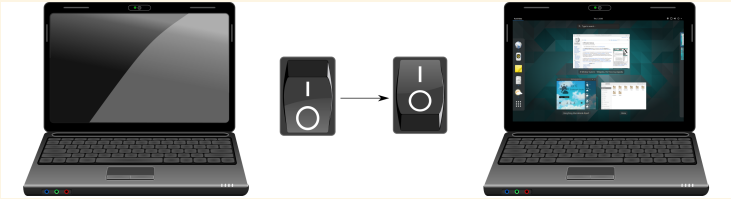
- An object's state is decided by the values of the attributes as well as the existing relations.
- Every object changes its state several times during its life time.
- This also means that the relations change during the life time.

# What can an object do?

- What an object is able to *do* is called its behaviour.
- Examples of behaviour for a computer could be:
  - turnOn()
  - turnOff()
  - runProgram()
- By using parenthesis we say that it is a behaviour.
- The behaviour defines the operations of the object.

# Operations change attributes

- State is changed for an object when the operations are called (executed).
- This means that when a behaviour is performed the state of the object is often changed.
- In Java the operations become *methods* as discussed in previous lectures.

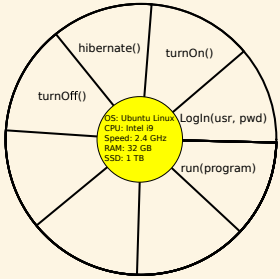


# Encapsulation

- All data is protected *inside* the object.
  - Data inside an object can only be reached via operations.
  - The object decides *how* data can be reached.
  - The object decides *what* data can be reached.
- The object is in *authority* of itself.
  - Called *encapsulation* or *data hiding*.

# Soccer diagrams

- A way to visualise encapsulation is to show a *soccer diagram*.
- In such a diagram the data is put in the centre and *around* it operations are put to shield it from the outside.

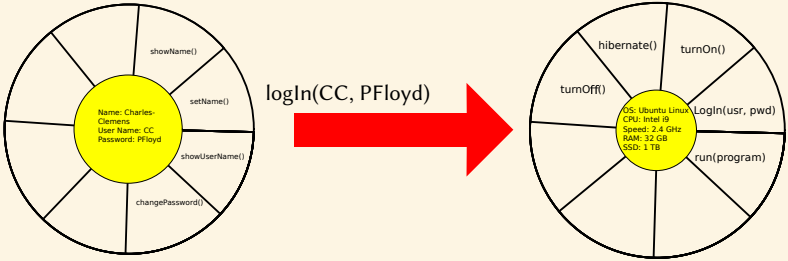


# Communication

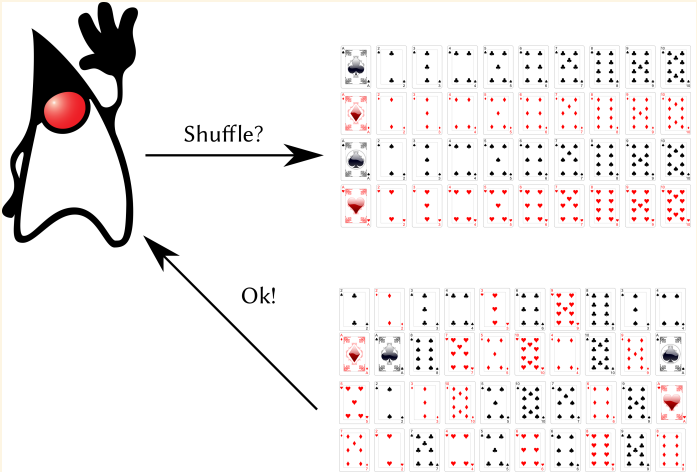
- Objects collaborate by sending *messages* to each other.
  - A message is sent to an operation from *one* object to *another* object.
  - Objects descided what to accept and how to react to messages.
- If the object *Charles-Clemens* gets the message wake up by the object *Police Pelle* then *Charles-Clemens* decides how to handle it.
  - For example “ignore”, “wake up”, “run” or similar.
  - This changes the attribute “activityState” for the object.

# Example of message

- The receiving object (computer) decides whether the user name and password are correct and whether to let the user in or not.



# Example with cards



# Classes

- The other important concept in object orientation is the *class*.
- Each object is an *instance* of a class – the class describes what “type” the object has.
- Classes allow us to model a set of objects with the same properties.
  - The class is a *template* for the objects.
- It’s the task of the class to define structure and properties for *all* objects belonging to the class.

# More on classes

- All objects of a class *must have* the same set of attributes and operations.
  - But they can have different attribute values, that is state.
- This way of classify things is an important part of how humans abstract the world.

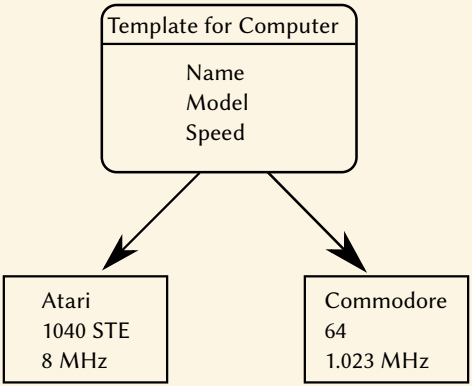
# Examples of classes and objects

- Previously in Java we have written:

```
Scanner scan = new Scanner(System.in);
```

- In this case `Scanner` is the class.
- The object is `scan` and it is connected to the standard input.
- The object is *unique*, there is only *one* such object.
- The class can be used to create many more objects.
  - In this case perhaps for reading from a file or the network.

# One class, many objects



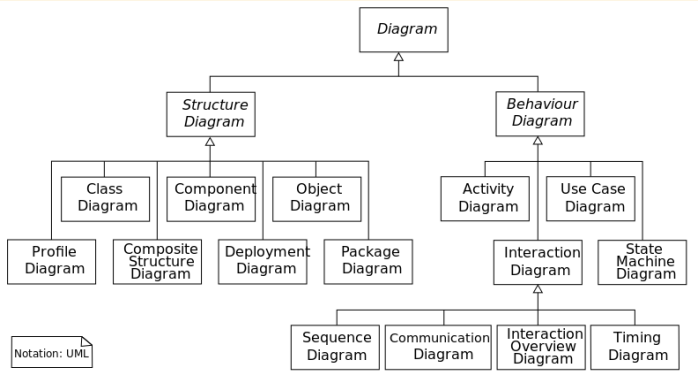
# UML

# UML

- UML is a *notation* for visualising object orientation.
- Before UML there were a number of techniques doing the same (but differently).
  - OMT – Object Modelling Technique
  - Booch
  - OOSE – Object-Oriented Software Engineering
- The company Rational (today part of IBM) hired Rumbaugh (OMT) and Booch to create a unified notation.
- Later the Swedish company Objectory AB with Ivar Jacobson was also bought by Rational.
- The so called “three amigos” Rambaugh, Booch and Jacobson developed the first version of UML.

# UML – thirteen diagrams

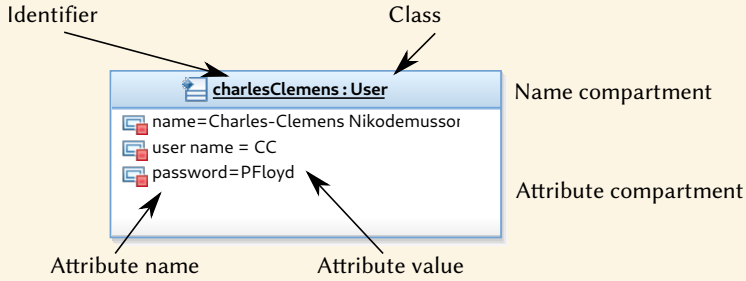
- UML consists of thirteen diagrams in two categories, static and dynamic.




Source: "UML diagrams overview" by Derfel73; Pmerson – This file was derived from:Uml_diagram2.png. Licensed under Public domain via Wikimedia Commons – http://commons.wikimedia.org/wiki/File:UML_diagrams_overview.svg#mediaviewer/File:UML_diagrams_overview.svg

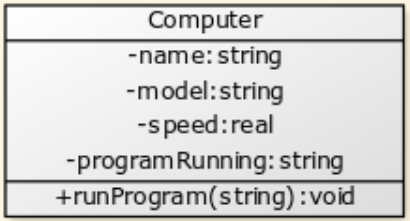
# UML notation for objects

- Below the notation for describing an object is shown.
- You seldom visualise an object...



# UML notation for classes

- Classes is one of the most common things to show with UML.




- For attributes the data type should be displayed.
- Operations have parameters and return type.

# Class diagrams

- The real value of object orientation is shown when you do *class diagrams*.
- These are diagrams showing the different classes interaction with each other.
  - What classes that communicate with which other classes.
- This will be described more in detail in the next lecture.
- At that time we will also look a bit more into the process of finding classes and relations.
- The third lecures will be even more about class diagrams and *object thinking*.

# Object Orientation and Java

# Classes and objects

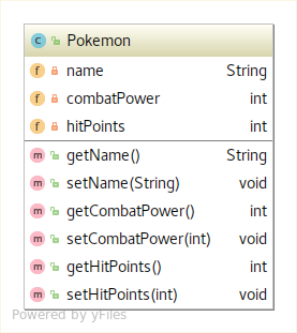
- We have already used several classes and objects.
- Each program has so far been its own class.
  - The name we give it is the name of the class.
- In this class we have added our own methods as well as used `main()`.
- Other classes we have used are `Math` and `Random`:

```
Random rnd = new Random();
```

- First we sasy that `rnd` should be of the class `Random`.
- After that we create a new object with `new Random()`.
- The new object is connected to `rnd` with the assignment.

# Our own class

- We are going to create a class to hold Pokémon.
- The class is a template for what information we are interested in for a Pokémon.
- From that we can create a number of different objects that all are Pokémon.
  - Bulbasaur, Caterpie, Rattata, Gloom and many, many more...



# Create a new class

- A new class is created as previously, `New` → `Java Class`.
- Important that each new class should be a new file.
  - Later we are going to look at *inner classes*, but for now each class should be its own file.
- Instead of adding `public static void main()` we will add our own methods.
- These methods will encapsulate the attributes, or fields as they are also called, as ordinary variables.

# Attributes/instance variables

- The variables we declare inside the curly brackets will become *instance variables*.
- We will mark most of them with the visibility `private`.
  - This will make them *only* visible in the class they are declared.
  - But it also means that they are visible in the *entire* class.
- This is an important part of the object oriented concept of *encapsulation*.
- Besides that they work exactly as variables we have seen before.
- They need a data type (actually often another class) and a name.
- Sometimes they are given a default value, but that is not necessary.
- Attributes should have names with lower case first letter.

# Methods

- For a class to actually *do* something it needs *methods*.
  - It's really something the objects do, but the class describes the behaviour.
- We have previously seen methods and done them ourselves.
- The method looks as it did previously:

```
visibility return_type name(parameters)
```

- An example could be:

```
public string showName()
```

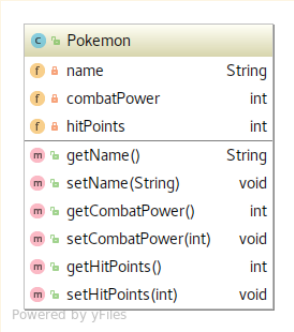
- Methods are often seen as the *interface* towards the class (there is also something else *called* an interface, more in the next course).

# No longer `static`

- Notice that the methods no longer have `static` in the signature.
- This means that they are no longer on class level but on *object level*.
- This in turn means that they can only be used if the class is *instantiated* into an object.
- All classes created from now on must be instantiated from a main class somewhere.
  - All examples will have at least two files since each class has its own file.
- The previously used methods have been on class level.

# A first class

```
public class Pokemon {
    private String name;
    private int combatPower;
    private int hitPoints;

    public String getName() {
        return name;
    }
    public void setName(String theName) {
        name = theName;
    }

    public int getCombatPower() {
        return combatPower;
    }
    public void setCombatPower(int theCombatPower) {
        combatPower = theCombatPower;
    }

    public int getHitPoints() {
        return hitPoints;
    }
    public void setHitPoints(int theHitPoints) {
        hitPoints = theHitPoints;
    }
}
```



# Creating objects

- To test a class you need another class.
- In this class create a `public static void main()`.
- The class just created can then be instantiated just as other classes.
  - To make it easier see to that both files belong to the same *package*.
- Through *dot notation* the public methods are reachable.
  - Notice that the IDE won't show you any of the attributes as they are private.
- Beroende på om metoden *tar emot* eller *returnerar* ett värde så kan man använda dem.

# Example

```java
package lecture7;

public class MyPokemons {
    public static void main(String[] args) {
        Pokemon pikachu = new Pokemon();

        pikachu.setName("Pikachu");
        pikachu.setCombatPower(98);
        pikachu.setHitPoints(50);

        System.out.println("My Pokémon is called " + pikachu.getName());
    }
}
```

Running:

```
My Pokémon is called Pikachu
```

# Larger example

```
package lecture7;

import java.util.Random;
import java.util.Scanner;

public class ManyPokemons {
    public static void main(String[] args) {
        Pokemon aPokemon = new Pokemon();
        Pokemon anotherPokemon = new Pokemon();
        Scanner scan = new Scanner(System.in);
        Random rnd = new Random();

        System.out.print("What is your first Pokémon? ");
        String name = scan.nextLine();
        aPokemon.setName(name);
        aPokemon.setCombatPower(rnd.nextInt(112) + 1);
        aPokemon.setHitPoints(rnd.nextInt(101) + 1);
```

# Larger example, cont.

```
System.out.print("What is your second Pokémon? ");
        anotherPokemon.setName(scan.nextLine());
        anotherPokemon.setCombatPower(103);
        anotherPokemon.setHitPoints(70);

        System.out.println("The first Pokémon is called " + aPokemon.getName()
                + " and has CP " + aPokemon.getCombatPower()
                + " and HP " + aPokemon.getHitPoints());

        System.out.println("The second Pokémon is called "
                + anotherPokemon.getName()
                + " and has CP " + anotherPokemon.getCombatPower()
                + " and HP " + anotherPokemon.getHitPoints());
    }
}
```

```
What is your first Pokémon? Pikachu
What is your second Pokémon? Rattata
The first Pokémon is called Pikachu and has CP 70 and HP 94
The second Pokémon is called Rattata and has CP 103 and HP 70
```

# The purpose of methods

- So far the methods have served the attributes right out.
  - It is possible to use the attribute without limitations.
- The purpose of methods is, however, to encapsulate the attributes so that they cannot be use *wrongly*.
- Therefore the code in a method should see to that no error is given to the attributes.
- Further down in the course we will learn about *exception handling* to manage this.
- Here we will just use simple error handling.

# Extended class

```java
package lecture7;

public class Pokemon {
    private String name;
    private int combatPower;
    private int hitPoints;

    public String getName() {
        if(name.equals("")){
            return "Unknown Pokémon";
        } else {
            return name;
        }
    }

    public void setName(String theName) {
        if(theName.equals("")) {
            name = "Unknown Pokémon";
        } else {
            name = theName;
        }
    }

    public int getCombatPower() {
        return combatPower;
    }

    public void setCombatPower(int theCombatPower) {
        if(theCombatPower > 0 && theCombatPower < 5000 ) {
            combatPower = theCombatPower;
        } else {
            combatPower = 0;
        }

    }

    public int getHitPoints() {
        return hitPoints;
    }

    public void setHitPoints(int theHitPoints) {
        if(theHitPoints > 0 && theHitPoints < 500) {
            hitPoints = theHitPoints;
        } else {
            hitPoints = 0;
        }
    }
}
```

# Constructors

- Many times you would like to send a value when you create an object.

```
Scanner scan = new Scanner(System.in);
```

- Here we say that `Scanner` should read from the keyboard.
- But, there is no method looking like that.
- It is a *constructor*, a special method used to accept “starting values”.
- A class can have one or many constructors.
- If no constructor is created, like in the example with `Pokemon`, there is a default constructor.
- However, if you create one or more of your own constructors, the default constructor disappears.

# Creating a constructor

- A constructor is always public as it is used to create objects.
- The constructor always has the same name as the class itself.
- Inside the parenthesis the parameters needed are placed (if any).
  - This part works as an ordinary method.
- Notice that you never write a return type, not even `void`, for a constructor.
  - In reality what is returned is an object of itself, but that isn't stated explicitly.
- If you do several constructors they must differ in the number and type of parameters.

# With constructors

```
package lecture7;

public class Pokemon {
    private String name;
    private int combatPower;
    private int hitPoints;

    public Pokemon() {
    }

    public Pokemon(String theName,
                   int cp,
                   int hp) {
        name = theName;
        setCombatPower(cp);
        setHitPoints(hp);
    }

    public String getName() {
        if(name.equals("")){
            return "Unknown Pokémon";
        } else {
            return name;
        }
    }
```

```
    public void setName(String theName) {
        if(theName.equals("")) {
            name = "Unknown Pokémon";
        } else {
            name = theName;
        }
    }
    public int getCombatPower() {
        return combatPower;
    }
    public void setCombatPower(int theCombatPower) {
        if(theCombatPower > 0 && theCombatPower < 5000 ) {
            combatPower = theCombatPower;
        } else {
            combatPower = 0;
        }

    }
    public int getHitPoints() {
        return hitPoints;
    }
    public void setHitPoints(int theHitPoints) {
        if(theHitPoints > 0 && theHitPoints < 500) {
            hitPoints = theHitPoints;
        } else {
            hitPoints = 0;
        }
    }
```

# Classes and arrays

- Classes (and therefore objects) works like *any other* type with arrays.
- This means that it is possible to create an array of a class.
  - Each element is a unique object.
  - It is important that *every* element must be instantiated with new.
- It is also possible to use arrays as attributes.
  - It is then important to protect the attribute with methods that handle the index ranges.

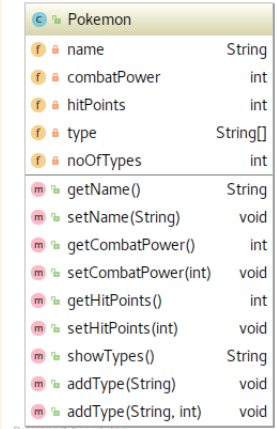
```java
public class MorePokemons {
    public static void main(String[] args) {
        Pokemon[] manyPokemons = new Pokemon[3];
        Scanner scan = new Scanner(System.in);
        Random rnd = new Random();

        System.out.print("What is your first Pokémon? ");
        manyPokemons[0] = new Pokemon(scan.nextLine(),
                rnd.nextInt(397) + 1,
                rnd.nextInt(55)+ 1);

        System.out.print("What is your second Pokémon? ");
        manyPokemons[1] = new Pokemon(scan.nextLine(),
                rnd.nextInt(392) + 1,
                rnd.nextInt(86)+ 1);

        System.out.print("What is your third Pokémonen? ");
        manyPokemons[2] = new Pokemon(scan.nextLine(),
                rnd.nextInt(1777) + 1,
                rnd.nextInt(150)+ 1);

        System.out.println("My Pokémon: ");
        for(int i = 0; i < manyPokemons.length; i++) {
            System.out.println(manyPokemons[i].getName()
                    + ": CP(" + manyPokemons[i].getCombatPower()
                    + ") & HP(" + manyPokemons[i].getHitPoints()
                    + ")");
        }
    }
}
```

# Example execution

```
What is your first Pokémon? Weedle
What is your second Pokémon? Kakuna
What is your third Pokémonen? Beedrill
My Pokémon:
Weedle: CP(228) & HP(23)
Kakuna: CP(363) & HP(42)
Beedrill: CP(1320) & HP(143)
```

# Arrays as attribute

- In this example `type` is an array of strings.
- Easy would be to just set and return the array but...
  - A Pokémon can only have two types, so you need to limit the usage.
- In the example `noOfTypes` is only used internally in the class.
- It is handled all the time by the methods to make sure that only two types are set.



```java
public class Pokemon {
    private String name;
    private int combatPower;
    private int hitPoints;
    private String[] type = new String[2]; // Length is important
    private int noOfTypes;

    public Pokemon() {
    }
    public Pokemon(String theName, int cp, int hp) {
        name = theName;
        setCombatPower(cp);
        setHitPoints(hp);
        noOfTypes = 0;
    }
    public Pokemon(String theName, int cp, int hp, String[] theType) {
        name = theName;
        setCombatPower(cp);
        setHitPoints(hp);
        if(theType.length == 1) {
            type[0] = theType[0];
            noOfTypes = 1;
        } else if(theType.length == 2) {
            type[0] = theType[0];
            type[1] = theType[1];
            noOfTypes = 2;
        }
    }
```

# Nya metoder

```
    public String showTypes() {
        if(noOfTypes == 2) {
            return type[0] + " & " + type[1];
        } else if(noOfTypes == 1) {
            return type[0];
        } else {
            return "Unknown type(s)";
        }
    }

    public void addType(String theType) {
        if(noOfTypes == 0) {
            type[0] = theType;
            noOfTypes++;
        } else if(noOfTypes == 1) {
            type[1] = theType;
            noOfTypes++;
        }
    }

    public void addType(String theType, int pos) {
        if(pos < 2) {
            type[pos] = theType;
        }
    }
}
```

# Huvudprogram, exempel

```java
package lecture7;

public class PokemonWithType {
    public static void main(String[] args) {
        Pokemon aPokemon = new Pokemon("Zubat", 412, 54);
        aPokemon.addType("Poison");
        aPokemon.addType("Flying");

        String[] types = new String[1];
        types[0] = "Fire";
        Pokemon anotherPokemon = new Pokemon("Charmander", 654, 28, types);

        System.out.println(aPokemon.getName() + ": " + aPokemon.showTypes());
        System.out.println(anotherPokemon.getName() + ": " + anotherPokemon.showTypes());
    }
}
```

Utdata:

```
Zubat: Poison & Flying
Charmander: Fire
```

# Important! About encapsulation

- Avoid to return the *entire* array.
- Object orientation is about *protecting* the object against inproper use.
- If the following method existed:

```
public String[] getTypeArray() { // BAD! Do not use!
    return type;
}
```

- Then it would have been possible to write code like this:

```
aPokemon.getTypeArray()[1] = "Grass";
System.out.println(aPokemon.getTypeArray()[1]);
```

- This is code making direct access to the content of the array!

# `this` and `null`

- Sometimes it is necessary to refer to the same object as we are in (ourselves).
- In Java there is a special type to do this called `this`.
- It can also be used to call another constructor in the same class.

```
public Pokemon(String theName, int cp, int hp, String[] theType) {
    name = theName;
    setCombatPower(cp);
    setHitPoints(hp);
    // Removed code
}
```

- Can be changed to:

```
public Pokemon(String theName, int cp, int hp, String[] theType) {
    this(theName, cp, hp);
    // Removed code
}
```

# Automatically generated

- Methods for setting and getting the attributes can be automatically generated by Intellij (and Eclipse).
- When this is done, `this` is used to refer to the actual object.
- This can be done in Intellij by writing the attributes and then right click and select `Generate` → `Getter and Setters`.
- The result is the following for the attribute `level`

```
public int getLevel() {
    return level;
}

public void setLevel(int level) {
    this.level = level;
}
```

# No object

- A common error is `NullPointerException`.
- This means that you have tried to reach an object of a class that has not been instantiated.
- This is especially common if you have objects in arrays.
- A `NullPointerException` happens because an uninitiated object is set to the value `null`.
  - `null` means “nothing” or something similar.
- To remove an object from memory it is also possible to set it to `null`.
  - The garbage collector will then later remove the value from memory.

# Summary

- This was the introduction to what object orientation is in Java.
- It is important to understand how object orientation works in java but equally important to understand *how to thing object oriented*.
- The next lecture will deal with relations between classes.
  - And with that, between objects.
- After the lectures you have a basic knowledge of how OO works, what is missing is inheritance and interfaces, which will be dealt with in the next course.
- To train your object thinking we will continue to look at object orientation.