Programmeertechnieken

# Introduction

# Polymorphism & dynamic binding

## Polymorphism

In object oriented programming languages, polymorphism refers to the concept that an object/variable can have many forms.

The most common use of polymorphism in OOP occurs when a parent class reference is used to a child class object. The only possible way to access an object is through a reference variable which can be of only one type. That means that once a reference variable is declared, the type cannot be changed.

The reference variable can be reassigned to other objects if that variable is not declared final. Also note that a reference variable can refer to any object and of any subtyp of its declared type. A reference variable can be declared as a class or interface type.

Examples of polymorphism in OOP are polymorphic assignment and polymorphic binding (also known as late binding or run-time binding). Barbara Liskov’s substitutability principle is an example of polymorphic assignment.

* Polymorphic assignment: a supertype variable that can hold subtype object.
* Polymorphic binding: a dynamic type is used to bind method implementation with a method call

Programming constructs such as method overloading, inheritance, and interface are used to support polymorphism.

**Method overloading** :

With method overloading, multiple methods can have the same name with different parameters (input and output).

* Example: two methods that basically do the same but have a different output type. Instead of defining two methods that should do the same thing, it’s better to overload one. You should have the same amount of input parameters for both method.

Barbara Liskov’s substitutability principle

**Method overriding**

Superclass and subclass define methods with the same signature. Each class has access to the fields of its class and the protected fields of the super classes.

A superclass will satisfy a static type check.

When a subclass method is called at runtime, it overrides the superclass version. This is where you would use the @Override annotation in order to let the compiler know that the subclass method will override the superclass method. It is a compiler directive but it is not obligatory. The goal of using this directive with this is to have a check at compile-time and improves the readability of your code.

A call to a overridden super class method needs to be done using “super” as otherwise the override sub class method will be used.

If a super class method is not overridden, the Object class implementation is used, so based on inheritance behavior.

Here are some useful methods in class Object:

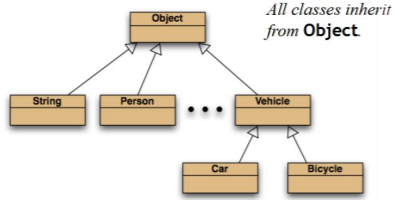
* toString()
  + commonly is overridden to return a String representation of an object
  + this method is implicitly called when a String object is needed
  + the default implementation (“classname@hashCode()”) is not particular useful.
* equals() & hashCode()
  + useful in collection implementations, check for existence, hashtables, etc.
* clone()
  + we will see much more about this method later
  + basically, it’s purpose is to create a deep or shallow copy of an object

## Inheritance

The pre-requisites for an OOP language are:

* Classes and objects
* Association/aggregation/composition
* Inheritance and polymorphism

Subclasses inherit the properties and behavior of their superclass. This makes that the superclass is more generic than a subclass. Other names for super classes are base classes or parent classes. Synonyms for subclasses are derived classes and child classes.

* Single rooted hierarchy : The class Object is the (implicit) superclass of all classes in the project
* Single inheritance: an object can inherit from one superclass but not from multiple one.

**Inheritance and constructors**

A subclass object has superclass fields and methods. The subclass constructor calls (implicitly?) the superclass constructor. The superclass constructor call must be the first statement in the subclass constructor. For example we have a superclass Post and a subclass MessagePost:

public class Post

{

private String username;

private long timestamp;

private int likes;

private ArrayList<String> comments;

public Post(String author){

username = author;

timestamp = System.currentTimeMillis();

likes = 0;

comments = new ArrayList<>();

}

…

}

public class MessagePost extends Post

{

private String message;

public MessagePost(String author, String text){

**super(author);**

message = text;

}

…

}

You can have different types of constructors:

* Implicit constructor: the constructor has no arguments
* User-defined constructor

Advantages of inheritance

* Polymorphism
* Avoiding code duplication
* Code reusage
* Easier maintenance
* Extendibility: it’s easy to add subclasses

## Interface

# Data structures

# Functional programming

# Programming language features

# Design patterns, refactoring & clean code

# Introduction to concurrent programming