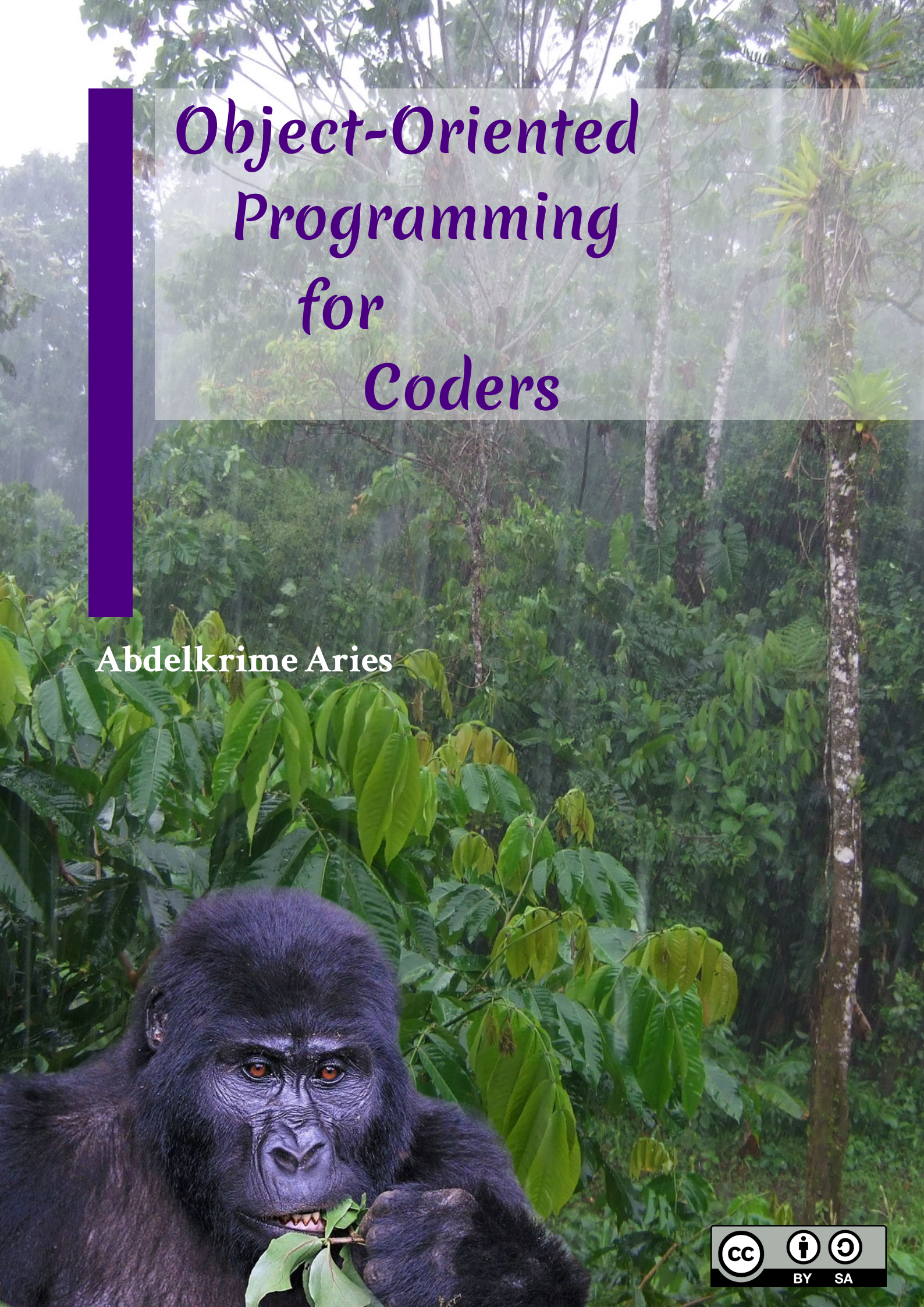
Object-Oriented Programming for Coders

Abdelkrime Aries



COPYRIGHT

Object-oriented programming for coders

Project URL: https://github.com/kariminf/oop4coders

THANKS

Information retrieval:

- Google search engine
- Wikipedia

Logos:

- C++ Logo: Jeremy Kratz, https://commons.wikimedia.org/wiki/File:ISO_C%2B%2B_Logo.svg
- Java Duke Logo: sbmehta converted, https://commons.wikimedia.org/wiki/File:Duke_(Java_mascot)_waving.svg
- Javascript unofficial Logo: Chris Williams, https://commons.wikimedia.org/wiki/File:Unofficial_JavaScript_logo_2.svg
- Lua project Logo: Lua.org, https://commons.wikimedia.org/wiki/File:Lua-logo-nolabel.svg
- Perl Logo: cannot use both since there are restrictions on commercial use, while this book allows commercial use.
- PHP Logo: Colin Viebrock, https://commons.wikimedia.org/wiki/File:PHP-logo.svg
- Python Logo: Python community, https://www.python.org/community/logos/
- Ruby Logo: Yukihiro Matsumoto, https://commons.wikimedia.org/wiki/File:Ruby_logo.svg

Cover:

- Gimp: Image Manipulation Program
- Mountain gorilla (Gorilla beringei beringei) eating, by Charles J Sharp, License: CC-By-SA-4.0, Link: https://commons.wikimedia.org/wiki/File:Mountain_gorilla_(Gorilla_beringei_beringei)_eating.jpg
- Rain in Tena, by Lion Hirth, License: ppublic domain, Link: https://commons.wikimedia.org/wiki/File:Rain_in_Tena.jpg.

Fonts:

- Google fonts

Editing:

- LaTeX: a document preparation system
- Atom: Source code editor.
- TeXstudio: Text editor for Latex
- KDE Neon: Linux system based on Ubuntu

Hosting and version control:

- Git: a version-control system
- Github: a web-based hosting service for version control using Git.

And, of course, all the readers.

VERSIONS

- August 1st, 2018: Draft Edition 1 (version 0.1)
- August 15th, 2018: Draft Edition 2 (version 0.2)
- September 9th, 2018: Draft Edition 3 (version 0.3)
- October 13th, 2018: First Edition (version 1.0)

PREFACE

Object-oriented programming (OOP) is one of the most popular programming paradigms. It is originated from the theory of concepts, and models of human interaction with real world phenomena (Nørmark, 2013). The four main concepts forming OOP are: abstraction, encapsulation, inheritance and polymorphism. There exists a lot of references talking about OOP and explaining its concepts. So if you want to go deeper in this paradigm, this is not the book you are looking for. But, still, you can grasp some ideas about these concepts by observing how programming languages are dealing with them.

In this book, the focus will be OOP from programmers' point of view. That is, the concepts will be presented as concrete programs with different OOP languages. Some programming languages are pure object-oriented such as Ruby, others allow primitive types such as Java, then there are those which skip some concepts such as Python with encapsulation, and those which have other ways to support OOP such as Javascript. This book serves as a comparison between different implementations of OOP concepts afforded by some programming languages. If a language lake support of a given concept, some solutions and hacks can be afforded. The explanations will be as short and informative as possible allowing more space to show all the beauty of codes.

So, "Less talking, more coding"

Why should you be interested?

Back in 2016, I presented the idea of this book on social media seeking some insights, suggestions and of course contributors. Well, the idea was not a success: no one cared to respond. One of my friends (Adnan) has finally asked a good question: "What will be the difference between your book and the huge number of already written books on this matter?". Maybe I did not present the idea well enough that time; but any book has to outline why its future readers should be interested. So, here are some outlines of what this book is about, and what difference it makes from others:

1. It is intended to be FREE (as gratis); no one will loose money for it.
2. It is open source; anyone can help enhancing this book. You can translate, correct, add a programming language, make your own copy from it as long as you mention the contributors and license it under CC-BY-SA 4.0.
3. It is intended to be a manual or a guide through different programming languages. So, no one will loose their time reading it; they will read it because they want to explore how a language handles a certain OOP concept. For instance, if someone wants to know if Javascript has private members, they will go directly to encapsulation chapter, private members section.
4. To my knowledge, this is the first attempt to gather different implementations of OOP concepts by as many languages. The available books and sources interested in OOP, mostly, use one programming language to illustrate OOP's four concepts. Others are books interested in a certain language and presenting OOP as a chapter. You can find the different implementations, but I doubt you will be able to find them on the same support.
5. It is helpful when someone wants to compare the differences between the object oriented languages. If they want to know what are the limits and workarounds of a programming language about some OOP concept, this is what this book is about.

Abdelkrime Aries, March 27, 2016
Revised: September 7, 2018

CONTENTS

LIST OF TABLES

LIST OF ABBREVIATIONS

API Application Programming Interface
CGI Common Gateway Interface
CLI Command-Line Interface
ECMA European Computer Manufacturers Association
ES5 ECMAScript 5
ES6 ECMAScript 2015
OO Object-oriented
OOP Object-oriented programming
SFINAE Substitution Failure Is Not An Error

CHAPTER 1

INTRODUCTION TO OOP

This chapter begins by a little history showing OOP's origins and meaning. And, of course, the four concepts will be presented using concise definitions so no one gets bored. Then, some benefits of OOP can be listed to show why this paradigm gets its fame. Some may argue that these benefits are just thin air and this paradigm has failed what it promised. It is not the purpose of this book to show if this paradigm has failed or succeed. But some opinions, either opposing or promoting it, may be helpful to the reader before diving into this book.

1 History

Simula is considered as the first object-oriented object programming language (Huang, 2004). Simula 67 introduced classes, objects, sub-classes, virtual procedures and dynamic binding (Black, 2013). It was created when ***Kristen Nygaard*** saw the need for a better way to represent heterogeneous elements and operations of a system while working on computer simulation programs. He was joined by ***Ole-Johan Dahl*** to create SIMULA I from Norwegian Computing Center in 1962.

By the time, while being at the University of Utah, ***Alan Kay*** liked the idea behind Simula language (Huang, 2004). He has a vision of a personal computer which provides graphics-oriented applications. He sold his vision to ***Xerox Parc*** and headed a team to create a personal computer called ***Dynabook*** which is programmed using ***Smalltalk***. The term "object-oriented programming" was coined by Alan Kay referring to objects as the foundation for computation. Alan Kay defines OOP in term of smalltalk as follows (Kay, 1993; Cunningham, 2014):

- Everything is an object
- Objects communicate by sending and receiving messages
- Objects have their own memory
- Every object is an instance of a class (which must be an object)
- The class holds the shared behavior for its instances (in the form of objects in a program list)
- To eval a program list, control is passed to the first object and the remainder is treated as its message

When asked by ***Stefan Ram*** in 2003 on definition of OOP, ***Alan Kay*** responded[1]:

- I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages.
- I wanted to get rid of data.
- My math background made me realize that each object could have several algebras associated with it, and there could be families of these, and that these would be very very useful. The term "polymorphism" was imposed much later (I think by ***Peter Wegner***) and it isn't quite valid, since it really

[1] http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

comes from the nomenclature of functions, and I wanted quite a bit more than functions. I made up a term "genericity" for dealing with generic behaviors in a quasi-algebraic form.
- I didn't like the way Simula I or Simula 67 did inheritance (though I thought Nygaard and Dahl were just tremendous thinkers and designers). So I decided to leave out inheritance as a built-in feature until I understood it better.

> Alan Kay (2003)
>
> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

2 Concepts

OOP core concepts are 4: abstraction, encapsulation, inheritance and polymorphism. Mostly, people get confused between the definitions of abstraction and encapsulation; also between the definitions of inheritance and polymorphism. This confusion can be justified by the fact that each pair are too similar. So, lets try to simplify each definition for you.

2.1 Abstraction

Abstraction is not a concept specific to OOP; it is more wide. In computer science, abstraction is a fundamental process which thought to be a requirement for computer scientists and students to develop good software (Saitta and Zucker, 2013). In OOP, abstraction is the mechanism of perceiving an entity in the system and some context, then taking out unnecessary details and keep those useful to that context (Rahman, 2014). You can consider it as a generalization of objects. For example, a person can be represented as an object with some characteristics: first name, last name, birthday, gender, color of eyes, etc. with some behavior: walk, talk, eat, die, etc. If we use this person as a student in the context of university, we will no more be needing some aspects: color of eyes, walk, talk, etc. Also, we need some other aspects such as: inscription year, grades, etc.

It is, also, the process to hide internal complex implementation and show only necessary details to the user. This will help the user to implement more complex logic using the afforded abstraction without the need to understand how it was implemented (Janssen, 2017a). For example, suppose we have an object called **car** with some behavior: start, stop, etc. Users does not need to know how in detail how the car start moving when they ask it to start; they just need to know what parameters they have to afford.

In term of programming, the different objects sharing same aspects can be represented by a class or a prototype as their abstraction. This abstraction will be used to create new objects with the same behavior but with different characteristics. Two major styles has raised in OOP world: class-based programming and prototype-based programming.

Class-based

In class-based programming, each object is an instance of a class. A class may be seen as a template which describes the structure and behavior of its instances. The object will have: a state (data), a behavior (methods) and identity (unique existence among other objects).

Prototype-based

In a prototype-based language, there is no distinction between class and object; there are just objects. An object which has its state, behavior and identity can be used as a template to create new objects by cloning and mutation rather than instantiating. This style was proposed to solve Smalltalk's class-based model's problems (Borning, 1986). The first prototype-based language is ***self*** programming language (Ungar and Smith, 1991).

2.2 Encapsulation

Encapsulation is the mechanism of binding data and methods operating on these data into a single object while restricting access to them (Janssen, 2017b). By hiding variables from the outside of this object, you can prevent misuse of these variables. So, encapsulation ensures basic integrity to the data by protecting it from the outside world. For example, a person has a first name and a last name besides other attributes. A person can not exist without these two, so they are assigned upon instantiation (creation of the object). Once created, the instance of person must not let the user modify these two attributes, otherwise it will affect some other tasks like statistics of clients in a market, etc.

2.3 Inheritance

Inheritance is the mechanism of basing a derived class from a parent class to reuse its existing attributes and methods (TechDifferences, 2016). It is used to create specialized classes from more generic ones by retaining the original class members and defining new ones. For example, lets say we have a class **Person** and want to create two new classes: **Student** and **Professor**. The idea is to not rewrite the same code for both these new classes. Instead, we create an inheritance relation between the two new classes (which will be called derived classes or subclasses) and the class **Person** (which will be called: parent class or superclass).

A class can share the same members of more than one class; this is called: multiple inheritance. For instance, a PhD student while being a student can teach and therefore has the same characteristics of a professor. Not all programming languages support multiple inheritance due to increasing complexity due to:

- If the two parents have the same method signature, so which one the subclass will inherit?
- The diamond problem: from the previous point, we have this problem if these two parents have a shared parent having a method with the same signature.
- The order of initialization when creating a new object of the subclass

It is important to point out that inheritance is different from sub-typing(Cook u. a., 1990). Sub-typing will create a relation **is-a** between the type and subtype, while inheritance only reuses implementation. Some languages, such as: C++, Java and C#, fuse these two concepts which leads to confusion. Go programming language, on the other hand, separates the two concepts.

2.4 Polymorphism

In general, polymorphism is the ability to assign a different meaning or usage to something in many contexts. In OOP, it is the ability to process objects differently based on their class or type (TechDifferences, 2016). Polymorphism can be applied to objects, when the language affords sub-typing along with inher-

itance considering an object of a derived class as an object of the superclass. Also, some languages allow the definition of multiple methods with the same name and different signature; this is called overloading. When a class inherit from another, it can override some of its methods which allows it to assign a different behavior than its parent's. An example of this, is the class Animal with a function talk(). Each class derived from Animal must redefine this method because each type of animals has its own way to talk.

2.5 Associations

An association is a "using" ("has-a") relationship between two objects which causes an object to perform an action on behalf of another. For example, a class "Book" has some elements of the class "Chapter"; in this case, the first class uses the last one. Aggregation is a type of association where the child can exist independently from the parent. For example, a **Car** has many parts such as "Wheel"; if we delete the car, the wheels still exist and may be related to another class. Composition, on the other hand, implies a relation of ownership; a child class cannot exist without the parent one. For instance, chapters can not exist without being in a book.

3 benefits

OOP is designed to simplify design, reuse code and enhance maintainability (Half, 2017; Popyack u. a., 2015). Simplicity is one of the benefits of OOP by modeling a problem using real world objects which makes the program more clear. It is important to mention, not every problem can be solved by OOP. Solving a simple problem using OOP can result in defining many components which leads to more confusion and complexity. For a big complex problem (especially enterprise problems), OOP can improve software development productivity by forcing designers to go through an extensive planning phase. Also, by defining different classes, it provides separation of duties; that is, the program can be divided on many developers where each can work independently from others.

Reuse is the biggest gain you can have from using OOP. Reusing code can decrease your software's size. It enables faster development by reusing libraries and codes developed in previous projects. This will lead to more focus into project analysis and design lowering the overall cost of development.

Maintainability can be enhanced using OOP. Since each class has its components (data and behavior), you can identify errors easily (if your program is well designed). Also, modifications inside a class do not affect other parts of the program. Moreover, adding new features is easier by extending the current project with new classes.

4 Limits

OOP can solve many complex problems and it excels at graphical user interfaces, but it cannot solve every problem. Personally, I think seeing OOP as a unique solution is one of its downfalls. Not every program developed using OOP is good and superior. So, one limit is that its users may seen it as the only paradigm out there, and other paradigms are just inferior. Some other limits are: size, speed and effort (Popyack u. a., 2015).

The size of OOP programs tend to be larger since it involves more code lines. This may cause a slower execution speed. These two limits are not problems nowadays since memory spaces are bigger and programming languages are more optimized.

In term of effort, OOP requires a lot of planning to move to coding phase. It meant to reduce complexity, but instead it deals with the application architecture rather than just the logic and algorithms. Learning OOP can be difficult for some people and it takes time to get used to it.

5 Some opinions about OOP

In this section, some thoughts about OOP are presented. Since this is a book about OOP, it is a good thing to present quotes opposing it (Bugayenko, 2016) so you can have an idea what is going wrong with it. There are some concerns about programming languages implementing the OO concepts. And then, there are other quotes defending and promoting it.

5.1 Oppose OOP

> Edsger W. Dijkstra (1989), TUG LINES Issue 32, August 1989
>
> Object-oriented programming is an exceptionally bad idea which could only have originated in California.

> Paul Graham (2003), The Hundred-Year Language
>
> Object-oriented programming offers a sustainable way to write spaghetti code.

> Richard Mansfield (2005), Has OOP Failed?
>
> With OOP-inflected programming languages, computer software becomes more verbose, less readable, less descriptive, and harder to modify and maintain.

> Eric Raymond (2005), The Art of UNIX Programming
>
> The OO design concept initially proved valuable in the design of graphics systems, graphical user interfaces, and certain kinds of simulation. To the surprise and gradual disillusionment of many, it has proven difficult to demonstrate significant benefits of OO outside those areas.

> Jeff Atwood (2007), Your Code: OOP or POO?
>
> OO seems to bring at least as many problems to the table as it solves.

> Linus Torvalds (2007)
>
> C++ is a horrible language. ... C++ leads to really, really bad design choices. ... In other words, the only way to do good, efficient, and system-level and portable C++ ends up to limit yourself to all the things that are basically available in C. And limiting your project to C means that people don't screw that up, and also means that you get a lot of programmers that do actually understand low-level issues and don't screw things up with any idiotic "object model" crap.

Joe Armstrong, Coders at Work: Reflections on the Craft of Programming

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Oscar Nierstrasz (2010), Ten Things I Hate About Object-Oriented Programming

OOP is about taming complexity through modeling, but we have not mastered this yet, possibly because we have difficulty distinguishing real and accidental complexity.

Rich Hickey (2010), SE Radio, Episode 158

I think that large objected-oriented programs struggle with increasing complexity as you build this large object graph of mutable objects. You know, trying to understand and keep in your mind what will happen when you call a method and what will the side effects be.

Eric Allman (2011), Programming Isn't Fun Any More

I used to be enamored of object-oriented programming. I'm now finding myself leaning toward believing that it is a plot designed to destroy joy. The methodology looks clean and elegant at first, but when you actually get into real programs they rapidly turn into horrid messes.

Joe Armstrong (2011), Why OO Sucks

Objects bind functions and data structures together in indivisible units. I think this is a fundamental error since functions and data structures belong in totally different worlds.

Rob Pike (2012)

Object-oriented programming, whose essence is nothing more than programming using data with associated behaviors, is a powerful idea. It truly is. But it's not always the best idea. ... Sometimes data is just data and functions are just functions.

John Barker (2013), All evidence points to OOP being bullshit

What OOP introduces are abstractions that attempt to improve code sharing and security. In many ways, it is still essentially procedural code.

Lawrence Krubner (2014)

Object Oriented Programming is an expensive disaster which must end. We now know that OOP is an experiment that failed. It is time to move on. It is time that we, as a community, admit that this idea has failed us, and we must give up on it.

Asaf Shelly (2015), Flaws of Object Oriented Modeling

Reading an object oriented code you can't see the big picture and it is often impossible to review all the small functions that call the one function that you modified.

5.2 Oppose implementations

Alan Kay (1997) The Computer Revolution hasn't happened yet

I invented the term object-oriented, and I can tell you I did not have C++ in mind.

Alan Kay (1997) The Computer Revolution hasn't happened yet

Java and C++ make you think that the new ideas are like the old ones. Java is the most distressing thing to happen to computing since MS-DOS.

5.3 Support

Grady Booch (1986) Software Engineering with Ada p. 220.

Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world.

Steve Steinberg, "Hype List", Wired, Vol. 1, No. 1, Mar/Apr 1993

Anyone even peripherally involved with computers agrees that object-oriented programming (OOP) is the wave of the future. Maybe one in 50 of them has actually tried to use OOP – which has a lot to do with its popularity.

Allen Wirfs-Brock, in response to the claims that OOP has failed

Have you ever look at the programs we were building in the early 1980s? At how limited their functionality and UIs were? OOP has been an incredible success. It enabled us to manage complexity as we grew from 100KB applications to today's 100MB applications.

Discussion

There are many programming paradigms, each has its differences, benefits and use cases. There are two computation models that you probably have encountered in your life: imperative and declarative programming. Imperative model defines programs as statements (sequence of steps) that change the state of the computer. Two main imperative paradigms are: procedural and object-oriented programming. On the other hand, declarative model defines what a program needs to accomplish instead of instructing how to achieve it. Two main declarative paradigms are: functional and logic programming. There is no such thing as a unique solution for every problem; this is why we have multiple paradigms. We cannot

conclude that a paradigm is worse or better than other just because it affords more or less features; "*More is not better (or worse), just different*"(Van Roy and Haridi, 2004).

OOP takes its roots from the theory of concepts, and models of human interaction with real world phenomena. It seeks to enhance simplicity, reuse and maintainability of programs by defining some concepts: abstraction, encapsulation, inheritance and polymorphism. Abstraction can enhance simplicity by hiding the complex structure of an entity and afford just interfaces to communicate with it. Then, you can use someone's code without knowing HOW it is implemented; all you have to know is what it does and how to use it. Hence, abstraction supports code reuse by not implementing the same thing implemented by others. Along with encapsulation, they can improve maintainability by putting together same functionalities and preventing direct access to data. Inheritance and polymorphism promote code reuse by defining new classes out of the existing ones, and applying methods to the same kind of objects. Inheritance is the most ill-used concept of all of them, and the one having most critics (bad ones). It is most often stated that using composition over inheritance is more appropriate for code reuse, this is a pattern design known as ***Composite reuse principle*** (Knoernschild, 2002). One drawback of this is that the methods have to be redefined in the derived class to forward those afforded by its components.

CHAPTER 2

HELLO WORLD

Introducing the different OOP languages used in this book is something which should be done before starting our journey. First of all, installing instructions must be presented to help anyone with zero knowledge about these languages get staring. Then, some coding features will be presented from a simple "Hello world" passing through functions and entry point to exception handling. It is important to point out that this chapter introduces the basic syntax without diving in different OOP concepts. Unless, of course, in case of "exceptions" which are based on classes in some languages such as Java.

1 Getting started

To be able to execute the examples afforded in this book, it is strongly recommended to install the languages compilers or interpreters. Here some instructions on how to get them ("**Gotta Catch 'Em All**").

1.1 C++

C++ is a general-purpose compiled programming language which affords object-oriented paradigm among others. It was developed by ***Bjarne Stroustrup*** at Bell Labs since 1979 (Stroustrup, 1993). To use C++ you have to install its compiler. The most used one is GNU g++[1] which is distributed freely and available for most systems.

Linux

Simply install g++ on your system; on Ubuntu you, simply, type:

```
$ sudo apt-get update
$ sudo apt-get install g++
```

Windows

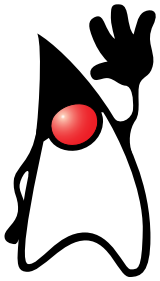
If you want to use g++ on Windows, you can install:

- MinGW: http://www.mingw.org
- CygWin: http://www.cygwin.com if you want to use Linux on Windows

You can use Microsoft Visual studio: https://www.visualstudio.com/vs/visual-studio-express/ Also, you can install Turbo C++ on https://turboc.codeplex.com

[1] GNU g++: gcc.gnu.org/

1.2 Java



Java is a general-purpose compiled programming language which is concurrent, class-based, object-oriented. It is intended to generate platform-independent programs. That is, the same compiled program called bytecodes will be able to execute in different machines and operating systems using a virtual machine. Developed by ***James Gosling*** at Sun Microsystems (acquired by Oracle Corporation) in 1991 under the name of "Oak", in 1995 its name was changed to Java, then its initial release was in 1996 (Panigrahy, 2014).

This is a detailed description on how to install Java https://www.java.com/en/download/help/download_options.xml. Simply, to install Oracle JDK, download it from this link: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

As for OpenJDK, you can install it on Linux distributions following these instructions: http://openjdk.java.net/install/. On some distributions such as **Linux Mint**, it is installed by default. It can be found on repositories, for example in Ubuntu you type:

```
$ sudo apt-get update
$ sudo apt-get install openjdk-8-jdk
```

It is recommended to use an IDE:

- Eclipse: which is used in our case. http://www.eclipse.org
- IntelliJ IDEA: https://www.jetbrains.com/idea/
- NetBeans: https://netbeans.org

In the examples afforded with this book, you have to compile and execute from **src** folder, if you use a command line. This is because the examples use packages to organize different codes of every chapter. So, if you want to compile **ObjMeth.java** in **codes/java/src/clsobj/**, you will have to do it as:

```
cd codes/java/src/
javac clsobj/ObjMath.java
```

1.3 Javascript

JavaScript is an interpreted programming language for the web, which supports object-oriented paradigm. It is meant to be used as client-side programming language executed on different browser, but it can also execute on server side using NodeJs for example. It was developed by Netscape Communications Corporation, and designed by ***Brendan Eich*** in 1995, the it was standardized by European Computer Manufacturers Association (ECMA) (Rubens, 2018).

Javascript can be used directly on any browser. Suppose we have a file called "**func.js**" containing a function "**fact(n)**" which calculates the factorial of a number. To use it, all you have to do is linking the file and calling the function in an HTML file:

```
1 <html>
2   <head>
3     <title>Functions</title>
4       <script type="text/javascript" src="func.js">
5       </script>
6       <script type="text/javascript">
7           function exec(){
```

```
            var n = document.getElementById("n").value;
            var res = "Fact(" + n + ")= ";
            res += fact(n);
            document.getElementById("result").innerHTML = res;
        }
    </script>
  </head>
    <body>
        Please enter an integer value:
        <input type="number" id="n" />
        <button id="func" onclick="exec()">Factorial</button>
        <div id="result"></div>
    </body>
</html>
```

Or you can use ***NodeJs*** to execute it directly on the shell which is the case in our late examples. You can download it here: https://nodejs.org/en/download/. In case of some Linux distributions, you can install it from their repositories:

```
$ sudo apt-get update
$ sudo apt-get install nodejs
```

1.4 Lua

Lua is an interpreted programming language, which supports object-oriented paradigm. It was designed primarily for embedded systems. Designed by ***Roberto Ierusalimschy***, ***Luiz Henrique de Figueiredo*** and ***Waldemar Celes*** in 1993. It is mostly used as a scripting language for games development (Lua.org, 2018).

Download it from here: http://lua-users.org/wiki/LuaBinaries. In Ubuntu, you can install it by using its version number as:

```
$ sudo apt-get update
$ sudo apt-get install lua5.2
```

1.5 Perl

Lua is a general-purpose interpreted programming language, which supports oriented-object paradigm. It was developed by ***Larry Wall*** in 1987 to simplify report processing on Unix (Ashton, 2001). It is used as a Common Gateway Interface (CGI) scripting language for web servers.

To install Perl, please refer to this page: https://www.perl.org/get.html. It is high probable that you have Perl installed on your Linux. As for Windows users, there are two projects:

- Strawberry Perl: http://strawberryperl.com
- ActiveState Perl: http://www.activestate.com/activeperl/downloads

1.6 PHP

PHP “PHP: Hypertext Preprocessor” is a general-purpose server-side scripting language which supports oriented-object programming since version 3.0 and improved in version 5.0. It was originally created by ***Rasmus Lerdorf*** in 1994 as a simple CGI (Cowburn, 2018).

For a detailed description on how to install the server, please refer to this page: http://php.net/manual/en/install.php. On Ubuntu, we just need the language:

```
$ sudo apt-get update
$ sudo apt-get install php
```

It will install a Command-Line Interface (CLI) called "php"; used to execute a php file directly from the shell.

1.7 Python

Python is an interpreted general-purpose programming language, support multiple paradigms including OOP. It was created by ***Guido van Rossum*** and released in the early 1990s Python Software Foundation (2018). It supports code readability as a design philosophy by using indentations as a mean of defining code blocks.

Refer to this page: https://www.python.org/downloads/. For Ubuntu, you can type:

```
$ sudo apt-get update
$ sudo apt-get install python
```

Or, you can refer to this page for further information: http://docs.python-guide.org/en/latest/starting/install3/linux/. You have to verify if it is already installed (Linux Mint has it by default).

1.8 Ruby

Ruby is an interpreted, object-oriented, general-purpose programming language. It was designed and developed by ***Yukihiro Matsumoto*** and released to the public in 1995 (Thomas and Hunt, 2001).

To install it, please refer to this page: http://www.ruby-lang.org/en/documentation/installation/. On Windows, you can use RubyInstaller: https://rubyinstaller.org. On Linux systems, you can find it on their repositories, such as Ubuntu:

```
$ sudo apt-get update
$ sudo apt-get install ruby
```

You have to verify if it is already installed (Linux Mint has it by default).

2 Hello world

Let's start with a Hello World introductory set of codes.

2.1 C++

A C++ program always has a main function, which represents the interface with the operating system. When you call a program (from shell for example), you actually calling this function. The main function is defined directly in the file and not inside a class or a structure.

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Suppose we use g++ for C++. To compile this code, you can specify the output file's name or not

```
$ g++ helloworld.cpp
$ g++ helloworld.cpp -o exec
```

This will create a file **a.out**, which we can execute as follows

```
$ ./a.out
$ ./exec
```

2.2 Java

In Java, a .java file can contain many classes, but only one with a public keyword. The public class must have the same name as the file. The main function is defined inside the main class.

```java
package hello;

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

To compile this code using command-line

```
$ javac HelloWorld.java
```

This will create a file **HelloWorld.class**, which we can be executed as follows

```
$ java HelloWorld
```

2.3 Javascript

Javascript is a scripting language; the instructions are written directly without a main class or a main function. The program can be used side-by-side HTML, and therefore executed in a browser.

```
    <script type="text/javascript" >
    alert("Hello World!");
    document.write("Hello World!");
    </script>
```

It can, also, be executed using nodeJs.

```
console.log("Hello World!");
```

To execute it in command-line, you can use either one of these two commands

```
$ node helloworld.js
$ nodejs helloworld.js
```

2.4 Lua

Lua is a scripting language; the instructions are written directly without a main class or a main function.

```
1 print ("Hello world!")
```

To execute it in command-line:

```
$ lua helloworld.lua
```

2.5 Perl

Perl is a scripting language; the instructions are written directly without a main class or a main function.

```
1 #! /usr/bin/perl -w
2 print "Hello world!\n";
```

To execute it in command-line:

```
$ perl helloworld.pl
```

2.6 PHP

PHP is a scripting language; the instructions are written directly without a main class or a main function. A code in PHP is delimited by *<?php* and *?>*

```
1 <?php
2 echo "<B>Hello world!</B>\n";
3 ?>
```

To execute it in command-line:

```
$ php helloworld.php
```

2.7 Python

Python is a scripting language; the instructions are written directly without a main class or a main function.

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 print ("Hello World")
```

To execute it in command-line:

```
$ python helloworld.py
```

2.8 Ruby

Ruby is a scripting language; the instructions are written directly without a main class or a main function.

```ruby
puts "Hello World!"
```

To execute it in command-line:

```
$ ruby helloworld.rb
```

3 Functions

Let's take the factorial function as an example. We implement the recursive version. The program, first, ask the user to introduce an integer number via the keyboard. Then it will call our function and print the result. We will print the code just for the function and not the entire program. Please check the code to see how to read the keyboard.

3.1 C++

A function, mainly, contains a return type, its name and parameters. The language uses *{}* to define instructions blocks. To return a value, the language uses the keyword ***return***. Statements end with a semicolon (*;*).

```cpp
int fact(int i){
    if (i <=1) return 1;
    return i * fact(i-1);
}
```

3.2 Java

A function contains a modifier, a return type, its name and parameters. It is always included in a class. The language uses *{}* to define instructions blocks. To return a value, the language uses the keyword ***return***. Statements end with a semicolon (*;*).

```java
public class Func {

    public static int fact(int i){
        if (i <=1) return 1;
        return i * fact(i-1);
    }

}
```

3.3 Javascript

A function starts with the keyword ***function***, its name and parameters. The language uses *{}* to define instructions blocks. To return a value, the language uses the keyword ***return***. Statements end with a semicolon (*;*).

```javascript
function fact(i) {
    if (i <= 1) return 1;
    return i * fact(i-1);
}
```

3.4 Lua

A function starts with the keyword ***function***, its name and parameters. All blocks are ended with the keyword ***end***. To return a value, the language uses the keyword ***return***. Each line contains at most one statement, but it can contains many separated by a semicolon.

```
function fact(i)
    if i <= 1 then
        return 1
    end
    return i * fact(i-1)
end
```

3.5 Perl

A function starts with the keyword ***sub*** and its name. The parameters can be recovered by shifting them from an array ***@_***. The language uses ***{}*** to define instructions blocks. To return a value, the language uses the keyword ***return***. Statements end with a semicolon (***;***).

```
sub fact {
    my $i = shift;
    if ($i <= 1){
        return 1;
    }
    return $i * fact($i - 1);
}
```

3.6 PHP

A function starts with the keyword ***function***,its name and parameters. The language uses ***{}*** to define statements blocks. To return a value, the language uses the keyword ***return***. Statements end with a semicolon (***;***).

```
function fact($i) {
    if ($i <= 1)
        return 1;
    return $i * fact($i - 1);
}
```

3.7 Python

A function starts with the keyword ***def***,its name and parameters. The language uses indentation to define statements blocks. To return a value, the language uses the keyword ***return***. Each line contains at most one statement, but simple statements can be separated by a semicolon.

```
def fact(i):
        if i <= 1:
                return 1
        return i * fact(i-1)
```

3.8 Ruby

A function starts with the keyword ***def***,its name and parameters. All blocks are ended with the keyword ***end***. To return a value, you can either use the keyword ***return*** or put the value directly in a line. Each

line contains at most one statement, but simple statements can be separated by a semicolon.

```
def fact(i)
    if i <= 1
        1
    else
        i * fact(i-1)
    end
end
```

4 Entry point

When a program is called from command-line, it searches for a function as entry-point. This is not the case for all languages; for instance, scripting languages execute without needing one. Nevertheless, they need a mechanism to recover the command-line parameters. Also, some languages, such as Python, afford a main function for entry-point lovers.

4.1 C++

In C++, main function is obligatory, it can have parameters or not. The return type is an integer which indicates how the program exited. The normal state is 0; Otherwise, it indicates that there was an error during the process. Contrary to C, in C++ you do not need to return explicitly. In this case, the return value is an implicit 0.

```
int main()
int main(int argc, char * argv[])
```

When we want to recover command-line parameters, we have to use the main with arguments.

- argv (argument vector): is a table of pointers on chars (a table of strings). The first entry is the name of the program, and the others are parameters.
- argc (argument count): is the size of that table

```
#include <iostream>
#include <cstdlib>

int fact(int i);

using namespace std;
int main(int argc, char *argv[])
{

    if (argc < 2) {
        cout << "Please enter an integer value\n";
        return 1;
    }

    int n = atoi(argv[1]);
    cout << "Fact(" << n << ")= " << fact(n) << "\n" ;
}
```

The header of our function **fact** is put ahead of the main, so we can use it. The compiler needs function declarations before the point of use (forward declaration). In our case, the implementation is after the main function.

4.2 Java

The main function is mandatory if we want an executable out of our class. It returns nothing (void) and have a list of string arguments. The main function must be always defined in the ***public*** class (which have the same name as the file). Also, it has to be ***static***, so it can be executed without the need to create an instance of its containing class.

```java
package hello;

public class Entry {
    public static void main(String[] args) {

        if (args.length < 1) {
            System.out.println("Please enter an integer value");
            System.exit(0);
        }

        int n = Integer.parseInt(args[0]);
        System.out.println("Fact(" + n + ")= " + fact(n));

    }

    public static int fact(int i) {
        if (i <=1)      return 1;
        return i * fact(i-1);
    }

}
```

4.3 Javascript

In Javascript, the entire file is a big main, which can contain declarations (variables and functions) and statements. The declarations are processed first, then the statements. This is why a function can be called before it is defined without the need to declare its header first. Being executed inside HTML, Javascript does not need arguments; But in case of Node.js, arguments are needed sometimes. The arguments passed by Node.js are stored in an array of strings ***process.argv*** which has **node** as its first elements, the name of the script as the second, and the rest are the command-line arguments.

```javascript
var args = process.argv;

if (args.length < 3) {
    console.log("Please enter an integer value");
    process.exit();
}

var n = args[2];
console.log("Fact(" + n + ")= " + fact(n) );
```

4.4 Lua

Lua does not have a main function, since all the code can be considered as one. Also, functions must be defined before being called, because functions in lua are just values stored in a variable. To recover command-line arguments, ***arg*** array is used.

```lua
fact = function (i)
    if i <= 0 then return 1 end
    return i * fact(i-1)
end
```

```lua

if #arg < 1 then
    print ("Please enter an integer value")
    os.exit()
end

n = tonumber(arg[1])
print ("Fact(" .. n .. ")= " .. fact(n))
```

4.5 Perl

The whole file is a main in Perl, and functions can be called before being defined. The command-line arguments are stored in a vector ***$ARGV***. In this code, we showed two methods to print the results. The first one is the more interesting: it interpolates the function's call and the variable inside a string. The second one is a simple concatenation-based one.

```perl
my $n = $ARGV[0];
die "Please enter an integer value" unless (defined $n);

print "Fact($n)= ${\fact($n)}\n";
print "Fact(" . $n . ")= ". fact($n) . "\n";
```

4.6 PHP

PHP does not have a main function; the first line of code encountered in the file will be executed an on. A function can be called before it was defined. To get command-line arguments, ***$argv*** vector is used; Where the first element is the name of the script.

```php
if (sizeof($argv) < 2) {
    echo "Please enter an integer value\n";
    exit();
}

$n = $argv[1];
echo "Fact($n)= " . fact($n) . "\n";
```

4.7 Python

Python does not need a main function; The statements are executed one by one. But to prevent instructions from executing when the file is exported as a library, there is a main mechanism. A test if a built-in variable ***__name__*** contains the string ***__main__*** which means the module is being run directly from command-line. Otherwise, this variable will contain the current module's name.

A function $f1$ can not be called before it was defined unless it is called inside another function $f2$. Even so, the second function $f2$ must be called after the definition of $f1$ and not before. In our example, if we define the factorial function after the **main** mechanism, we will get an error.

To get the command-line arguments, we must import ***sys*** module and recover the vector ***sys.argv***. The first element contains the name of the script and the rest contains the arguments.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys
```

```
5
6  def main():
7      if len(sys.argv) < 2:
8                  print("Please enter an integer value")
9                  sys.exit()
10
11         n = int(sys.argv[1])
12         print("Fact(" + str(n) + ")= " + str(fact(n)))
13         print sys.argv[0]
14
15 def fact(i):
16         if i <= 0:
17                 return 1
18         return i * fact(i-1)
19
20 if __name__ == "__main__":
21         main()
```

If you want to put the function in the end so badly, there is a solution. Just import the function from the same file under the main mechanism. In the following example, our file is called **entry.py**.

```
1  if __name__ == "__main__":
2          from entry import fact
3          ...
4  def fact(i):
5          ...
```

4.8 Ruby

In Ruby, there is no main function, but there is a mechanism to prevent execution when the file is called as a module. There is a variable called *__FILE__* which contains the file name which is stored in *$0* and *$PROGRAM_NAME*. The command-line arguments are stored in a variable *ARGV*. Functions must be defined before being used, otherwise you will get an error.

```
1  #!/usr/local/bin/ruby -w
2
3  def fact(i)
4      if i <= 1
5          1
6      else
7          i * fact(i-1)
8      end
9  end
10
11 if __FILE__ == $0
12     if ARGV.length < 1
13         print "Please enter an integer value\n"
14         exit 0
15     end
16
17     n = ARGV[0].to_i
18     puts "Fact(" + n.to_s + ")= " + fact(n).to_s
19 end
```

For those who want it so badly to define a function after it is being called. If you import the file using *require* keyword inside the main block, you will get a warning about the method being redefined. Another solution is to use the blocks: *BEGIN* or *END*. The former tells the interpreter to begin with this code, and the latter tells it to finish with it.

```
1  END {
```

```
        if __FILE__ == $0
                ...
        end
}

def fact(i)
        ...
end
```

```
if __FILE__ == $0
...
end

BEGIN {
        def fact(i)
                ...
        end
}
```

5 Exceptions

5.1 C++

In C++, exceptions can be of any type. They are thrown using the keyword ***throw***, and handled using ***try*** and ***catch***. You can throw a string as an exception.

```
int fact(int i){
    if (i < 0) throw "Negative numbers don't have a factorial!";
    if (i > 5) throw "The number is too big!";
```

Then, capture it

```
    try {
        int f = fact(n);
        cout << "Fact(" << n << ")= " << f << "\n" ;
    }
    catch (const char* e) {
        cout << e << "\n";
    }
```

If you want to specify the type of the exception, here comes the OOP solution. You have to define new types by inheritance from ***std::exception***. Then, redefine the function ***what*** which affords the error message.

```
struct NegException : public exception {
    const char * what () const throw () {
        return "Negative numbers don't have a factorial!";
    }
};

struct BigException : public exception {
    const char * what () const throw () {
        return "The number is too big!";
    }
};
```

The exceptions can be thrown by creating a new instance of the defined classes.

```
    if (i < 0) throw NegException();
    if (i > 5) throw BigException();
```

We catch them either by they parent's type **std::exception**, or by catching each type apart.

```
    try {
        int f = fact(n);
        cout << "Fact(" << n << ")= " << f << "\n" ;
    }
    catch (exception& e) {
        cout << e.what() << "\n";
    }
```

5.2 Java

In Java, you can define exceptions by defining some classes which inherit from the class ***Exception***. Or you can throw a new instance of the latter. The message in your new exception classes can be passed to the superclass, by the keyword ***super*** in their constructors. Also, you can redefine the function ***getMessage*** as shown in our example.

```
class NegException extends Exception {
    @Override
    public String getMessage() {
        return "Negative numbers don't have a factorial!";
    }
}

class BigException extends Exception {
    public String getMessage() {
        return "The number is too big!";
    }
}
```

The function which throw new instances of our exceptions, using the keyword ***throw***, must declare that it have the potential to throw them in its header via the keyword ***throws***.

```
    public static int fact(int i) throws NegException, BigException {
        if (i < 0 ) throw new NegException();
        if (i > 5) throw new BigException();
```

To capture these exceptions, you must use ***try*** and ***catch***, or you can send them to an upper caller using **throws**. There is a keyword ***finally*** which is used to execute code in either cases: exception or not. Its benefit is to clean up before exiting; for instance: closing a file.

```
        try {
            int f = fact(n);
            System.out.println("Fact(" + n + ")= " + f);
        }
        catch (NegException | BigException e) {
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("That's all!!");
        }
```

There are some other types of exceptions called ***unchecked exceptions*** which they don't force the programmer to capture them. You can define your own unchecked exceptions by inheriting from ***RuntimeException***. In this case, you will not need to declare that your function is capable of throwing them. There are some built-in unchecked exceptions relative to arrays and number conversion as shown in this example.

```
        int n = 0;
        try {
```

```
            n = Integer.parseInt(args[0]);
        }
        catch (IndexOutOfBoundsException e) {
            System.out.println("Please enter an integer value");
            System.exit(0);
        }
        catch (NumberFormatException e) {
            System.out.println("The argument has to be an integer");
            System.exit(0);
        }
```

5.3 Javascript

Using ***throw*** statement, you can throw an exception of any type: integer, string, object, etc.

```
function fact(i) {
    if (i < 0) throw "Negative numbers don't have a factorial!";
    if (i > 5) throw "The number is too big!";
```

The function's execution will stop and the control will be passed to the first catch block where it is being called. If the error is not captured, the program will terminate. To capture an exception, you need to call the function inside ***try*** and ***catch***. You can add ***finally*** which will execute in both cases: failure or success.

```
try {
    var f = fact(n);
    console.log("Fact(" + n + ")= " + f );
}
catch(error) {
    console.log(error);
}
finally {
    console.log("That's all!");
}
```

You can throw an error using the class ***error***

```
new Error([message[, fileName[, lineNumber]]])
```

5.4 Lua

As mentioned in Lua manual: you do not need error handling for most applications. But in case you want to throw an error, you can use the function ***error***

```
function fact(i)
    if i < 0 then error ("Negative numbers don't have a factorial!") end
    if i > 5 then error "The number is too big!" end
```

Then, if you want to handle this error, you can use the function ***pcall*** to encapsulate your previous code. It will return a success indicator and the returned value from your function. The value, here, is either the result from te function or the raised error.

```
n = tonumber(arg[1])
success, f = pcall(fact, n)
if success then
    print ("Fact(" .. n .. ")= " .. f)
else
    print (f)
end
```

5.5 Perl

Perl does not have a good built-in mechanism for exceptions. This is why there are some perl modules designed specially for exception handling. If you want to throw errors, you can use the keyword ***die***.

```
sub fact {
    my $i = shift;
    die("Negative numbers don't have a factorial!") if ($i < 0);
    die "The number is too big!" if ($i > 5);
```

Using ***eval***, you can capture the error into ***$@***.

```
eval {
    my $f = fact($n);
    print "Fact($n)= $f\n";
};
if ($@) {
    print "The error: $@\n";
};
```

5.6 PHP

In PHP, Exception handling is available in version 5 and greater. To raise an exception, either you create an instance of the class ***Exception*** with a message as parameter, or you create custom exceptions by inheriting from that class.

```
class NegException extends Exception {
    public function errorMessage() {
        return "Negative numbers don't have a factorial!";
    }
}

class BigException extends Exception {
    public function errorMessage() {
        return "The number is too big!";
    }
}
```

To raise the exception, all you have to do is calling the keyword ***throw*** followed by an instance of the custom exception.

```
function fact($i) {
    if ($i < 0) throw new NegException();
    if ($i > 5) throw new BigException();
```

Then, to capture the exception, you can use ***try*** and ***catch***.

```
try {
    $f = fact($n);
    echo "Fact($n)= " . $f . "\n";
}
catch (NegException $e) {
    echo $e->errorMessage() . "\n";
}
catch (BigException $e) {
    echo $e->errorMessage() . "\n";
}
```

From PHP5.5, a block of ***finally*** can be specified to execute code after normal execution or exception handling.

5.7 Python

The exceptions inherit from the class ***Exception***.

```
class NegException (Exception):
        def __init__(self):
                Exception.__init__(self, "Negative numbers don't have a factorial!")

class BigException (Exception):
        def __init__(self):
                Exception.__init__(self, "The number is too big!")
```

To throw an exception, use the keyword ***raise***

```
def fact(i):
        if i < 0:
                raise NegException
        if i > 5:
                raise BigException
```

To capture these exceptions, use ***try*** and ***except***. You have to capture the most specific exception first, then the generic ones. After ***except*** blocks, you can use a clean-up block: ***finally***.

```
try:
        n = int(sys.argv[1])
except ValueError:
        print("The value must be an integer")
        sys.exit()
```

There are some built-in exceptions you can use, such as this example.

```
try:
        f = fact(n)
        print("Fact(" + str(n) + ")= " + str(f))
except Exception as e:
        print(str(e))
```

5.8 Ruby

In Ruby, you can throw an exception affording just a message which will create a new instance of ***RuntimeError***. To throw an exception, you need to use the keyword ***raise***

```
def fact(i)
    case
    when i < 0
        raise "Negative numbers don't have a factorial!"
    when i > 5
        raise "The number is too big!"
```

To handle the exception, you can use the keywords:

- begin: to begin the block
- rescue: to process the exception (you may have many of this)
- else: executes if there was no exception
- ensure: executes always in the end
- end: to close the block

```
begin
    f = fact(n)
    puts "Fact(" + n.to_s + ")= " + f.to_s
rescue Exception => e
    puts e.message + "\n"
else
    puts "No exception\n"
ensure
    puts "That's all!\n"
end
```

You can specify custom classes to handle exceptions which can inherit from ***StandardError***. It is a good practice to regroup these errors all together.

```
module Errors
    class NegException < StandardError
        def initialize()
            super("Negative numbers don't have a factorial!")
        end
    end
    class BigException < StandardError
        def initialize()
            super("The number is too big!")
        end
    end
end
```

To throw them, all you have to do is creating an instance after calling ***raise***.

```
    when i < 0
        raise Errors::NegException.new()
    when i > 5
        raise Errors::BigException.new()
```

You can capture them by their parent exception class ***Exception***, or each by its class.

```
rescue Errors::NegException => e
    puts e.message + "\n"
rescue Errors::BigException => e
    puts e.message + "\n"
```

Discussion

Before introducing oriented-object concepts of each language, how about a little comparison between our programming languages. Table 2.1 represents a general comparison between Object-oriented (OO) languages based on these criteria:

- **Type safety**: is the extent to which a programming language prevents type errors.
- **Type expression**: represents the type declaration. If it is explicit, the programmer have to specify the type of a variable or a method. If it is implicit, the type is inferred based on how the variables are being used.
- **Type checking**: When the code is verified for type constrains, this can be happen in compile-time (static check) or run-time (dynamic check).
- **Implementation**: An implementation can be compiled or interpreted. A language cannot be considered compiled or interpreted, but its implementation can. Since a language can have many implementations, here we present the most used and known implementation for it.

Table 2.1: General comparison

Language	Type safety	Type expression	Type checking	Implementation
C++	Weak	Explicit	Static	Compiled
Java	Strong	Explicit	Static	Compiled (bytecode)
Javascript	Weak	Implicit	Dynamic	Interpreted
Lua	Strong	Implicit	Dynamic	Compiled (bytecode)
Perl		Implicit	Dynamic	Interpreted
PHP		implicit/optional explicit	Dynamic	Interpreted
Python	Strong	implicit/(v3.5+ optional explicit)	Dynamic	Interpreted
Ruby	Strong	implicit	Dynamic	Interpreted

CHAPTER 3

CLASS AND OBJECT

Abstraction is the first pillar of OOP. From a perception of an entity in a system or a context, certain aspects must be retained while others must be ignored. This entity is represented by a class or a prototype, with a state (attributes) and behavior (methods). This chapter will present how a class or prototype is implemented in each language. It will show how each language defines a constructor which is a special method served as a mean to instantiate (copy) a class (prototype). Then, it will take you in a little trip through different class members: fields, properties and methods. Finally, there are special methods which can be implemented for an object, mostly, inherited from a universal base class.

1 Class declaration

Class declaration in the following examples is self explanatory. As for prototype-based implementations, a concise explanation must be afforded.

1.1 C++

```
1 class Person
2 {
3 };
```

1.2 Java

```
1 public class Person {
2 }
```

1.3 Javascript

There are no classes in javascript, a function can be used to simulate a class.

```
1 function Person(name, byear) {
2 }
```

In ECMAScript 2015 (ES6), the ***class*** keyword has been introduced, but it still is a syntactical sugar over the existing prototype-based classes.

```
1 class Person {
2 }
```

1.4 Lua

There is class concept in Lua; but using its meta-programming, objects can be created. There are many ways to do this (Ierusalimschy, 2003), one famous way is to use meta-tables. The *__index* is used as fail-safe mechanism when it fails to lookup something in the table.

```
1 local Person = {}
2 Person.__index = Person
```

1.5 Perl

A Perl class is simply a package which ends with *1;* If you want to create many packages inside one file, you can put their definitions inside curly braces *{}*.

```
1     package Person;
```

1.6 PHP

```
1 class Person {
2 }
```

1.7 Python

```
1 class Person(object):
```

1.8 Ruby

```
1 class Person
2 end
```

2 Constructors and destructors

A constructor is a special function which is responsible for preparing the object to be created. The preparation, mostly, is concerned by initializing variables or calling some functions.

Once an object is deleted or out of scope, it is no longer needed. The destructor is a function which is called before deleting an object. It can be called to free external resources used by the object before it will be terminated.

2.1 C++

The constructor in C++ takes the same name as the class without a return type; It can have parameters or not. If you do not define a constructor, the default one with no parameters will be used. As for the destructor, it has the same name as the class prefixed by a tilde ~.

```
1 class Person
2 {
3
4 public:
5
6     Person(std::string name, int byear){
7         this->name = name;
```

```
        this->byear = byear;
        nbr++;
    };

    ~Person(){
        nbr--;
        std::cout << name << " is out\n";
    };
};
```

An object in C++ can be created as:

- an automatically destroyed object when it goes out of scope, or
- a dynamically created object of which the address is binded to a pointer

```
    Person p = Person("Karim", 1986);
    Person* p2 = new Person("Karim+1", 1987);
```

To free an allocated object's memory, in the second case, you can call ***delete***. If the pointer you are deleting is zero, nothing will happen. Deleting a pointer twice will cause problems; This is why it is recommended to set is to zero right after you delete it (Stranden, 2015).

```
    delete p2; //delete the pointer
```

2.2 Java

The constructor, in Java, is a method having the class name without a return type. There is no destructor in Java, because the objects are heap allocated and garbage collected (ddimitrov, 2008). Finalizers are like destructors in purpose: close files, sockets, etc. But, they are not called immediately after the object is freed. When the garbage collector collects an object with a finalizer, it enqueue it to be finalized. But, there is no guarantee that the finalyzer will ever be called; The application can exit without waiting the object to be finalyzed.

```
public class Person {
    public Person(String name, int byear){
        this.name = name;
        this.byear = byear;
        nbr++;
    }

    @Override
    public void finalize() {
        nbr--;
        System.out.println(name + " is out");
    }
}
```

To create a new object, use the keyword ***new***.

```
        Person p2 = new Person("Karim+1", 1987);
```

To fasten the destruction of an object, you can assign ***null*** to all variables referencing it. But, even doing that, the program can shut down before the finalyzer is called.

```
        p2 = null;
```

2.3 Javascript

In ES6, the constructor is a function defined by the keyword ***constructor***. There is no destructor or finalyzer function.

```
class Person {
    constructor(name, byear) {
        this.name = name;
        this["byear"] = byear;
        Person.nbr++;
    }
}
```

Actually, a function can be used to define a prototype (class) and it is in the same time a constructor. This is what actually happens when you use ES6 class definition. In ECMAScript 5 (ES5), this is how a class is created.

```
function Person(name, byear) {
    this.name = name;
    this.byear = byear;
    Person.nbr++;
}
```

To create a new object, you use the keyword ***new***

```
var p = new Person("Karim", 1986);
```

2.4 Lua

A class can be represented by a meta-table used as a fall-back when an object cannot find a key (method or attribute). The constructor is a method which the name can be anything you want, usually **new** or **create**. This function must return an object (table) to which you have to set a meta-table (our class) using a function called ***setmetatable***.

```
function Person.new(name, byear)     -- The constructor
    Person.nbr = Person.nbr + 1
    return setmetatable({name = name, byear = byear}, Person)
end
```

Then to create a new object, you simply call your constructor as a normal method.

```
local p = Person.new("Karim", 1986)
```

Another fancy way to create a constructor is by using the meta-method ***__call*** which allows you to call a meta-table as a method.

```
setmetatable(Person, {
    __call = function (cls, name, byear)
        Person.nbr = Person.nbr + 1
        return setmetatable({name = name, byear = byear}, Person)
    end,
})
```

Then, the object can be created as follows

```
local p = Person("Karim", 1986)
```

2.5 Perl

In Perl, the constructor is a subroutine called **new**, or any name you want. The parameters are shifted from the global variable *@_* where the first one is the class itself. You have to create a hash and marking it with a package (class) name using ***bless***.

When the last reference to an object goes away, the object is destroyed. In this case, Perl will call the method ***DESTROY*** if it is defined.

```
{
    package Person;

    my $nbr = 0;

    sub new {
        my $class = shift;
        my $self = {
            _name => shift,
            _byear => shift,
        };
        $nbr++;
        bless $self, $class;
        return $self;
    }

    sub DESTROY {
        my ($self) = @_;
        $nbr--;
        print "$self->{_name} is out\n";
    }
}
```

To create a new object, either you use the keyword ***new*** or use the arrow function. If you use the second method, you can call your constructor whatever you want; But, by convention it is ***new***.

```
$p = new Person("Karim", 1986);
$p2 = Person->new("Karim+1", 1987);
```

To delete a reference, you use ***undef*** which will trigger the destructor.

```
undef($p2);
```

2.6 PHP

The constructor is a method called ***__construct*** and the destructor is a method called ***__destruct***.

```
class Person {
    public function __construct( $name, $byear ) {
        $this->name = $name;
        $this->byear = $byear;
        self::$nbr++;
    }

    function __destruct() {
        self::$nbr--;
        print "$this->name is out\n";
    }
}
```

To create an instance of a class, the keyword ***new*** is used.

```php
$p = new Person("Karim", 1986);
```

To delete a reference, the keyword ***unset*** is used.

```php
unset($p2);
```

2.7 Python

The constructor is a method called ***__init__*** and the destructor is a method called ***__del__***.

```python
class Person(object):
        def __init__(self, name, byear):
                self.name = name
                self.byear = byear
                Person.nbr = Person.nbr + 1

        def __del__(self):
                Person.nbr = Person.nbr - 1
                print (self.name + " is out")
```

To create an object, the class is called as if it was a method.

```python
p = Person("Karim", 1986)
```

To delete a reference, the keyword ***del*** is used.

```python
del p2
```

2.8 Ruby

The constructor is a method called ***initialize***. Ruby does not have a destructor, but it has a finalizer. To bind the object with a finalizing function, you have to call ***ObjectSpace.define_finalizer*** in the constructor. The finalizer, like Java, can never be called.

```ruby
class Person

    @@nbr = 0

    def initialize(name, byear)
        @name, @byear = name, byear
        @@nbr += 1
        ObjectSpace.define_finalizer( self, proc {Person.finalize(@name)})
    end

    def self.finalize(n)
        @@nbr -= 1
        puts "#{n} is out"
    end
end
```

To create a new object from a class, ***new*** is called.

```ruby
p = Person.new("Karim", 1986)
```

To free a reference from an object, you can assign it the ***nil*** value.

```ruby
p2 = nil
```

3 Members

There are three types of members:

- Field: They are state variables or attributes. For example: a person has a name, birth year, etc.
- Methods: Procedures which define the behavior of the object. For example: a person can talk, walk, etc.
- Properties: They are something between a field and a method. The read/write syntax is similar to fields' but it is translated into getters/setters methods.

A member can be accessed as:

- Class member: It is shared between all the instances of this class. Also, it can be called from the class itself without instantiating.
- Object member: The member is specific to the instance (object).

For properties, lets say: we have a class **Rectangle** which is composed of **height** and **width**. We want to keep the width, always, bigger or equal to the height. For the sake of the example, suppose when we want to get the height we add 1, and when we want to get the width we add 2. Also, suppose only the width has a setter and the height will receive its value based on the previous width and its setter's value. We assign the width twice: 50 and 20. Then, we use an internal method **info** which shows the state of the two fields. Finally, we use getters over the two fields and print the recovered values. The result in the languages which support properties, or has a way to emulate them, should be as follows:

```
Width: 50, Height: 20
w: 52, h: 21
```

3.1 C++

Fields are defined by type and name inside the class header. Static fields are defined using the keyword ***static***.

```
1     std::string name;
2     int byear;
3     static int nbr;
```

Non constant class (static) fields must be initialized outside the header and not inside it.

```
1 int Person::nbr = 0;
```

Methods can be defined inside the class header as shown in this example. Static methods must be defined using the keyword ***static***.

```
1     void info() {
2         std::cout << "My name: " << name ;
3         std::cout << ", My birth year: " << this->byear << "\n";
4     };
5
6     static int population() {
7         return nbr;
8     };
```

To call static methods, you should use class name. As for the object methods, if the reference is a pointer use arrow annotation.

```
    p.info();
    p2->info();
    int nbr = Person::population();
```

There are no properties in C++, but they can be emulated by overriding assigning operator.

3.2 Java

Fields are defined by access modifier (See Encapsulation chapter), type and name. Class (static) fields are defined using the keyword ***static***. All fields must be defined inside the class block; They can be initialized without needing a constructor. In fact, they are automatically initialized: integers to 0, objects to Null, etc.

```
    private String name;
    private int byear;
    private static int nbr = 0;
```

Methods are defined inside the class block where static ones are distinguished by the keyword ***static***.

```
    public void info(){
        System.out.print("My name is: " + name);
        System.out.println(", My birth year is: " + byear);
    }

    public static int population() {
        return nbr;
    }
```

Static methods are called using the class name; But, also, they can be called from its instances.

```
        p.info();
        p2.info();
        int nbr = Person.population();
```

There are no properties in Java, you have to define your getters/setters and use them as methods. As far as we know, there is no mechanism to emulate their functionality either.

3.3 Javascript

The fields are defined inside the class body (either ES6 class or function) using the keyword ***this***. You can define them using two ways (they are the same), as shown in the example.

```
        this.name = name;
        this["byear"] = byear;
```

As for static fields, you can use the pattern **<class name>.<field>** to access it. The field must be initialized outside the class definition.

```
Person.nbr = 0;
```

Methods are defined inside the class definition in ES6. Also, static methods are preceded by the keyword ***static***.

```
class Person {
    info(){
```

```
        console.log("My name: " + this.name + ", My birth year: " + this.byear);
    }

    static population() {
        return Person.nbr;
    }
}
```

If the class is defined using a function (which is the real definition in Javascript), you can define functions inside the constructor using the keyword ***this***. Also, you can define them outside the constructor by assigning a function to the class's prototype.

```
Person.prototype.info = function(){
    console.log("My name: " + this.name + ", My birth year: " + this.byear);
}

Person.population = function(){
    return Person.nbr;
}
```

Properties can be defined using ***Object.defineProperties*** and the keywords ***get*** and ***set***.

```
Object.defineProperties(Rectangle.prototype, {
    w: {
        get: function () {
        },
        set: function (value) {
        }
    },
    h: {
        get: function () {
        }
    }
});

var r = new Rectangle();
r.w = 50;
r.w = 20;
r.info();
console.log("w: ", r.w, ", h: ", r.h);
```

In Javascript, you can create members inside the constructor or inside the prototype. When a member is created inside the constructor, each time you create an object, it will have its **own** member. In case of fields, it is so natural that each object has its own and do not have to lookup its class for a field. But in case of methods, you will end up with duplicates while you can use just one. When a member is created inside the prototype, an object will not have its own while created from a class's constructor; it will lookup that method in its class.

```
function Person(){
    this.talk = function(){}
}

function Machine(){}
Machine.prototype.talk = function(){}

var p1 = new Person(),
    p2 = new Person();

var m1 = new Machine(),
    m2 = new Machine();

console.log("p1.talk == p2.talk", p1.talk == p2.talk);
```

```
15 console.log("m1.talk == m2.talk", m1.talk == m2.talk);
```

3.4 Lua

The fields are defined inside the table which will be our object. In our example, a table is created inside the constructor then the class table is set to be its meta-table. To define a static field, it can be assigned to the meta-table (our class).

```
1 function Person.new(name, byear)     -- The constructor
2     Person.nbr = Person.nbr + 1
3     return setmetatable({name = name, byear = byear}, Person)
4 end
5
6 Person.nbr = 0
```

To define dynamic methods, there are two ways: the dot notation where you have to specify your current object, and the colon notation (*:*) which will add **self** implicitly.

```
1 function ClassName.method1(self)
2 end
3 function ClassName:method2()
4 end
```

As for static methods, they are defined using the dot notation as *method1* without self parameter.

```
1 function Person:info()
2     print("My name: " .. self.name .. ", My birth year: " .. self.byear)
3 end
4
5 function Person.population()
6     return Person.nbr
7 end
```

To call dynamic methods, there are two ways, again: dot notation or colon notation.

```
1 p:info()
2 p2.info(p2)
3 local nbr = Person.population()
```

Properties can be defined using the meta-methods ***__index*** and ***__newindex***. When the interpreter can not find a field in the current table, it calls the ***__index*** meta-method. This meta-method has **self** and a key as parameters, and can be used to define getters in our case. You have to verify if the key already exists and return the member, otherwise your methods will not be known by the interpreter. ***__newindex*** meta-method is used for table update and can be used as a setter.

```
1  local Rectangle = {}
2  Rectangle.__index = function(self, key)
3      if key == "w" then
4      elseif key == "h" then
5      end
6  end
7
8  Rectangle.__newindex = function(self, key, value)
9      if key == "w" then
10     end
11 end
12
13 local r = Rectangle.new()
14 r.w = 50
```

```
r.w = 20
r:info()
print("w: " .. r.w .. ", h: " .. r.h)
```

3.5 Perl

Object fields are defined in the hash which will be returned by the **new** subroutine as the new object.

```
        my $self = {
            _name => shift,
            _byear => shift,
        };
```

Class (static) fields are defined outside the new subroutine as global variables scoped at the file level using ***my*** keyword. Or, they can be defined as package level variables using the keyword ***our***.

```
    my $nbr = 0;
```

There is no difference between dynamic and static methods. If the method is called from an object, the object will be passed as the first argument. If it is called from a class, the class name will be passed as the first argument. The implementation of the method will decide its access type.

```
    sub info {
        my( $self ) = @_;
        print "My name: $self->{_name}, My birth year: $self->{_byear}\n";
    }

    sub population {
        return $nbr;
    }
```

As said, the difference between dynamic and static methods is in how they are called.

```
$p->info();
$p2->info();
$nbr = Person::population();
```

Properties in Perl do not follow the definition given in this section; Sure we can define setters/getters, but they do not behave as attributes. A subroutine which has two arguments (the object and the value) is a setter. If there is just one, then it is a getter.

```
package Rectangle;

sub w {
    my $self = shift;
    if (my $value = shift) { # setter
    } else { # getter
        return $self->{_width} + 2;
    }
}

sub h {
    my $self = shift;
    return $self->{_height} + 1;
}
1;

$r = new Rectangle();
$r->w(50);
```

```
$r->w(20);
$r->info();
print "w: " . $r->w . ", h: " . $r->h . "\n";
```

3.6 PHP

The fields are declared by defining access mode and name. In PHP's terminology, fields are called properties; But, we rather call them fields to differentiate them from the properties we defined earlier. A static field is defined using the keyword ***static***.

```
class Person {

    private $name;
    private $byear;
    private static $nbr = 0;
}
```

There are no difference between static and dynamic functions in header syntax. The difference is in implementation: the static one can access just static fields by using the keyword ***self*** instead of ***this***.

```
class Person {
    public function info() {
        echo "My name: $this->name, My birth year: $this->byear \n";
    }

    public function population() {
        return self::$nbr;
    }
}
```

There is a difference between calling a dynamic method and a static one: dynamic methods use arrow function and static methods use a double colon.

```
$p->info();
$p2->info();
$nbr = Person::population();
```

Properties getters/setters can be defined using PHP's magic methods ***__get*** and ***__set***. It is not a good idea though (Sauer, 2013), and the alternative is to define getters and setters as methods.

```
class Rectangle {
    function __set($property, $value) {
        if ($property == 'w') {
        }
    }

    function __get($property) {
        switch ($property) {
            case 'w': return $this->width + 2;
            case 'h': return $this->height + 1;
        }
    }

}

$r = new Rectangle();
$r->w = 50;
$r->w = 20;
$r->info();
echo "w: $r->w, h: $r->h\n";
```

3.7 Python

Static fields are defined inside the class block, while dynamic ones are defined inside the constructor using the parameter **self** (or any reference to the current object). In pure Python, the dynamic fields of an object (instance) are stored in a dictionary ***self.__dict__*** (Python Software Foundation, 2018). A static field is accessed by the class name, not by **self**. Also, static fields are stored in other dictionary bounded to the class, not the instance.

```
1  class Person(object):
2
3          nbr = 0
4
5          def __init__(self, name, byear):
6                  self.name = name
7                  self.byear = byear
8                  Person.nbr = Person.nbr + 1
```

Methods are defined using the keyword ***def*** inside the class block. There is no difference between static and dynamic methods in term of method's signature. But in Python 2.6+, you can use a decorator ***staticmethod*** to mark a static method.

```
1  class Person(object):
2          def info(self):
3                  print("My name: " + self.name + ", My birth year: " + str(self.byear))
4
5          @staticmethod
6          def population():
7                  return Person.nbr
```

Dynamic methods are accessed via the instance while static ones are accessed using the class name.

```
1  p.info()
2  p2.info()
3  nbr = Person.population()
```

Properties can be defined using the new style classes (object as superclass) for Python 2.2+. Starting from Python 2.6, there are decorators to define setters and getters. In the example, the implementation is omitted.

```
1  class Rectangle(object):
2          @property
3          def w(self):
4
5          @property
6          def h(self):
7
8          @w.setter
9          def w(self, value):
10
11 r = Rectangle()
12 r.w = 50
13 r.w = 20
14 r.info()
15 print("w: " + str(r.w) + ", h: " + str(r.h))
```

3.8 Ruby

Static fields are defined inside the class block using @@, while dynamic fields are defined inside the constructor or the class block using the keyword @.

```ruby
class Person

    @@nbr = 0

    def initialize(name, byear)
        @name, @byear = name, byear
        @@nbr += 1
    end
end
```

Methods are defined using the keyword ***def*** and delimited by the keyword **end**. Static methods always begin with the keyword ***self***. Also, methods without parameters do not need round brackets ***()***.

```ruby
class Person
    def info
        puts "My name: #@name, My birth year: #@byear"
    end

    def self.population
        @@nbr
    end
end
```

We call dynamic methods using the instance name and static ones using the class name. Round brackets can be omitted if no argument is expected.

```ruby
p.info()
p2.info
nbr = Person.population
```

Properties can be defined by a field using a no-parameter method as a getter and another with the template **<prperty name> = (value)** as a setter.

```ruby
class Rectangle
    def w
        @width + 2
    end

    def h
        @height + 1
    end

    def w=(value)
    end
end

r = Rectangle.new
r.w = 50
r.w = 20
r.info
puts "w: #{r.w}, h: #{r.h}"
```

Also, there are automatically implemented getters/setters in Ruby using read*write access.

```ruby
class Rectangle
        attr_accessor :width # read/write
        attr_reader :height # read only
        attr_writer :bar # write only
end
```

4 Object methods

These are methods inherited from the Object class, or as called by some languages: magic methods. Most programming languages agree on the following methods:

- String representation: When an object is concatenated with a string, it can act like one by calling a method.
- Object copy: A method which enables an object to create a clone of itself.
- Value equality: A method which helps to decide if an object equals another based on some criteria (their fields).
- Object comparison: A method which is used to decide if an object is greater, equal or lesser than another.
- Hash code: A method which returns a hash code for the object.

In our example, we will implement our **Person** class, and the previously presented methods as follows:

- String representation: The method must return a string containing the name and the year of birth.
- Object copy: The method must return a new object with the same name but with plus one year to the current birth year.
- Value equality: The method returns true if the objects have the same value for the field called **name**.
- Object comparison: The comparison is made on birth years; If they are equal, then on the name.
- Hash code: The hash code will be based on the name (we can add the year as well, but it will be a long code).

4.1 C++

In C++, there are no methods inherited from the object class because there is no universal base class. But, C++ supports operation overloading which can replace base-class methods. To overload an operation for a class based on its fields, you have to mark these operations as friends to the class. This will allow them to access class's private members.

```
1 class Person
2 {
3 private:
4     friend std::ostream& operator<<(std::ostream &strm, const Person &p);
5     friend bool operator==(const Person &p1, const Person &p2);
6     friend bool operator>(const Person &p1, const Person &p2);
7 };
```

String representation

There is no ***toString*** in C++, but operators such as << can be overloaded to handle an object of a class as a string.

```
1 std::ostream& operator<<(std::ostream &strm, const Person &p) {
2     return strm << "(" << p.name << ": " << p.byear << ")";
3 }
```

Then, you can call it as follows:

```
1     Person p = Person("Karim", 1986);
2     std::cout << "As string: " << p << "\n";
```

Object copy

You can just define a cloning function for the class and use it.

Value equality

You can overload the operator `==` to test the equality between two objects of the same class.

```
bool operator==(const Person &p1, const Person &p2) {
    return p1.name == p2.name;
}
```

Object comparison

You can overload the operators `>`, `<`, `>=`, `<=` and `!=` to compare two objects of the same class. *std::string* class has its own *compare* method.

```
bool operator>(const Person &p1, const Person &p2) {
    if (p1.byear == p2.byear) return (p1.name.compare(p2.name) > 0);
    return (p1.byear > p2.byear);
}
```

Hash code

You can define a hashcode function for your current class.

4.2 Java

In Java, every class inherits from the universal class *Object*. This class defines some methods by default: string representation, object copy, value equality and hashcode. To personalize them, you can override these methods in your class.

String representation

You can override *toString* which comes from the class *Object*.

```
    class Person2 implements Cloneable, Comparable<Person2> {
        @Override
        public String toString() {
            return "(" + name + ": " + byear + ")";
        }
    }
```

When the object is concatenated with strings, this method will be called automatically to represent this object as a string.

```
        System.out.println("As string: " + p);
```

Object copy

The universal class *Object* implements a method *clone* which performs a shallow copy. That is, if a field is an object (not primary type), it will copy the reference and not create a new instance of it. To perform

deep copy, you have to override this function. But before that, you have to implement the class ***Clonable*** otherwise your function will throw an exception ***CloneNotSupportedException***.

```
1    class Person2 implements Cloneable, Comparable<Person2> {
2        public Person2 clone() throws CloneNotSupportedException {
3            Person2 c = (Person2) super.clone();
4            c.byear++;
5            return c;
6        }
7    }
```

Value equality

You can override the method ***equals*** which takes a variable of type ***Object*** as argument. This is the other object to which we want to test our current object, and it can be of any other effective type. So, basically, you can test your current object against any other type if you wish to. In our example, if the other object has not the same type as the current's, the method returns false.

```
1    class Person2 implements Cloneable, Comparable<Person2> {
2        @Override
3        public boolean equals(Object obj) {
4            if (getClass() != obj.getClass()) return false;
5            return name.equals(((Person2) obj).name);
6        }
7    }
```

When testing equality with non primitive variables, you have to call this method. If you use ***==*** instead, you will have a reference equality test; that is, the variables will be tested if they have the same reference or not.

```
1            System.out.println("p == p3? " + (p==p3));
2            System.out.println("p.equals(p3)? " + (p.equals(p3)));
```

Object comparison

Objects, by default, do not implement a comparing method. If you want to compare an object with another, you can implement the interface ***Comparable***. It is very useful since a lot of other methods are based on it such as ***sort*** of collections. Then, you have to implement one method which is ***compareTo***. It takes one argument of type ***Object*** which is the other object to compare to, unless you use generics to specify that object's type. This function returns a negative integer, zero, or a positive integer if the current object (specified by: this) is less than, equal to, or greater than the other object.

```
1    class Person2 implements Cloneable, Comparable<Person2> {
2        @Override
3        public int compareTo(Person2 p2) {
4            int cmp = byear - p2.byear;
5            if (cmp == 0) cmp = name.compareTo(p2.name);
6            return cmp;
7        }
8
9    }
```

Hash code

You can override ***hashCode*** method to define specialized hash code calculation to a class.

```
        @Override
        public int hashCode() {
            return name.hashCode();
        }
```

4.3 Javascript

All objects in JavaScript are descendants of ***Object***; all objects inherit methods and properties from ***Object.prototype***.

String representation

Every object has a ***toString*** method inherited from the universal class, and it can be overridden.

```
function Person(name, byear) {
    this.name = name;
    this.byear = byear;
}

Person.prototype.toString = function(){
    return "(" + this.name + ": " + this.byear + ")";
}

var p = new Person("Karim", 1986);

console.log("As string: " + p);
```

Object copy

There is no method permitting an object to deep clone itself. You can use ES6 ***Object.assign*** to fuse object's members to an empty one, but it will afford shallow copying.

```
let obj2 = Object.assign({}, obj1);
```

To deep copy an object, you can use libraries such as ***lodash*** via its method ***clonedeep***

Value equality

There is no value equality method; you can define your own.

Object comparison

There is no comparison method; you can define your own.

Hash code

There is no hash code method; you can define your own.

4.4 Lua

Lua is prototype-based OOP language where an object is a table. There exists some meta-methods defined for these tables and can be overridden.

String representation

By defining the meta-method *__tostring*, the object can be used as a string.

```lua
function Person:__tostring()
    return "(" .. self.name .. ": " .. self.byear .. ")"
end
```

Then, the object can be used as as string

```lua
function Person:__tostring()
    return "(" .. self.name .. ": " .. self.byear .. ")"
end
```

Object copy

To deep copy an object, you have to implement it yourself since there is no meta-method for this.

Value equality

The equality of two objects of a class can be defined using the meta-method *__eq* which has two objects as parameters.

```lua
function Person.__eq (p1, p2)
    return p1.name == p2.name
end
```

Object comparison

Besides equality, there are two more meta-methods: less than *__lt* and less than or equals *__le*. The other relations can be inferred by negating these three. Strings can be compared by default.

```lua
function Person.__lt (p1, p2)
    return (p1.byear < p2.byear) or (p1.byear == p2.byear and p1.name < p2.name)
end

function Person.__le (p1, p2)
    return (p1.byear < p2.byear) or (p1.byear == p2.byear and p1.name <= p2.name)
end
```

To call them:

```lua
print("As string: " .. tostring(p))
print("p == p3? " .. tostring(p==p3))
print("p ~= p3? " .. tostring(p~=p3))
print("p <= p2? " .. tostring(p <= p2))
print("p > p2? " .. tostring(p > p2))
```

Hash code

You have to implement your own hash code method.

4.5 Perl

In Perl, all classes inherit from a base class named ***UNIVERSAL***. This class affords some methods other than the ones we are discussing here, the most important ones:

- **$obj->isa(TYPE)** which verifies if this object is an instance of a certain type.
- **$obj->can(METHOD)** which verifies if this object affords a certain method.

By overloading some operations, our special methods can be emulated.

String representation

The string representation of an object can be done by overloading the stringification operator.

```
use overload
'""' => sub {
    my ($self) = @_;
    return "($self->{_name}: $self->{_byear})"
},
```

Stringification is not the only object transformation, there are other operations that can be overloaded to treat the object in different contexts:

- Boolification by overloading ***bool***, the object can be used as a boolean **if ($obj) {...}**
- Numification by overloading ***0+***, the object can be used as a number **say $obj + 1;**
- Regexification by overloading ***qr***, the object can be used as a regex **if ($str =~ /$obj/)**
- Scalarification by overloading ***${}***, the object can be used as a scalar ref **say $$obj;**
- Arrayification by overloading ***@{}***, the object can be used as an array ref **say for @$obj;**
- Hashification by overloading ***%{}***, the object can be used as a hash ref **say for keys %$obj;**
- Codeification by overloading ***&{}***, the object can be used as a code ref **say $obj->(1, 2, 3);**
- Globification by overloading ****{}***, the object can be used as a glob ref **say *$obj;**

Object copy

You can write your own cloning method since there is no default one.

Value equality

Perl uses operations such as **==** for numeric equality and ***eq*** for strings equality. Like other operations, they can be overloaded. We can overload numeric equality for that matter, or we can define a numification of the object that gives us numeric values compatible to what we want to achieve.

```
use overload
'==' => sub {
    my ($self, $other) = @_;
    return ($self->{_name} eq $other->{_name})
}
```

Object comparison

Likewise, comparing operations: <=> for numerals and ***cmp*** can be overloaded. They return -1, 0 or 1 if the first element is less, equal or greater than the second. Do not use the same operation inside its overloaded implementation, otherwise you will have an error: Use of uninitialized value in numeric comparison. Other logic operations can be overloaded such as >, <, etc.

Hash code

There is no hash code method in Perl.

4.6 PHP

There is no universal base class in PHP. But there exists some reserved keywords for methods names known as magic methods. String representation and object cloning fall into this category of methods.

String representation

The ***__toString()*** method allows a class to decide how it will react when it is treated like a string.

```
1 class Person {
2     public function __toString() {
3         return "($this->name: $this->byear)";
4     }
5 }
```

Object copy

An object copy is created using the ***clone*** keyword (which calls the object's ***__clone*** method if possible). An object's ***__clone*** method cannot be called directly.

```
1 class Person {
2     public function __clone() {
3         $this->byear ++;
4     }
5
6 }
```

Value equality

There is no magic method for equality, and no operation overload. Using ==, two objects are equal if they have the same attributes and values, and they are instances of the same class. Using ===, two object variables are equal if they point to the same object (reference equality).

```
1 $p = new Person("Karim", 1986);
2 $p3 = clone $p;
3 $b = ($p==$p3)? "true": "false";
4 print "p == p3? " . $b . "\n";
```

Object comparison

No comparison, you have to define your own.

Hash code

There is no internal hash code method for classes.

4.7 Python

In new Python syntax, all classes inherit from the class ***object***. There exists many magic methods in Python, which are called upon using an object as argument to a built-in function. For example, the magic method ***__str__*** of a class is called when one of its instances is used to create an object of type ***str***. Besides the methods we are presenting here, there exists many magic methods such those for arithmetic operations, for built-in unary methods, etc.

String representation

To use an object as a string, you have to define the magic method ***__str__***.

```
1 class Person(object):
2         def __str__(self):
3                 return "(" + self.name + ": " + str(self.byear) + ")"
```

Then, you can use it by creating an ***str*** object out of another existing object

```
1 p = Person("Karim", 1986)
2 print ("As string: " + str(p))
```

Object copy

To clone an object, you need to use the module ***copy*** which defines two methods: ***copy*** for shallow copy and ***deepcopy*** for deep copy.

```
1 p = Person("Karim", 1986)
2 p3 = copy.copy(p)
```

To tell those methods how to copy an object, you have to define the magic methods: ***__copy__*** and ***__deepcopy__***.

```
1 class Person(object):
2         def __copy__(self):
3                 return Person(self.name, self.byear + 2)
```

Value equality

The equality can be defined using the magic method ***__eq__***. It is called when you use the equality test ***==***.

```
1 class Person(object):
2         def __eq__(self, other):
3                 return (self.name == other.name)
4 p = Person("Karim", 1986)
5 p3 = copy.copy(p)
6 print("p == p3? " + str(p==p3))
```

Object comparison

Comparing objects can be done using magic method ***__cmp__*** which returns 0 if they are equal, a negative number if the caller is less than the other object, a positive number otherwise. It is no longer available for Python3 due to redundancy with other magic methods. The comparison methods replacing it are: equal ***__eq__***, not equal ***__ne__***, less than ***__lt__***, greater than ***__gt__***, less than or equals to ***__le__*** and greater than or equals to ***__ge__***.

```
class Person(object):
        def __lt__(self, other):
                return (self.byear < other.byear)
p = Person("Karim", 1986)
p3 = copy.copy(p)
print("p < p3? " + str(p<p3))
```

Hash code

Magic method ***__hash__*** is used to define how an object generates its hash code. It is called when we use the object as argument of the method ***hash***.

```
class Person(object):
        def __hash__(self):
                return hash(self.name)
p = Person("Karim", 1986)
print ("HashCode: " + str(hash(p)))
```

4.8 Ruby

Object is the default root of all Ruby objects. Besides our methods, this class defines other ones.

String representation

By overriding the method ***to_s***, you can use the object as a string.

```
class Person
        def to_s
                "(#{@name}: #{@byear})"
        end
end

p = Person.new("Karim", 1986)
puts "As string: #{p}"
```

Object copy

To create an object copy, there are two methods: ***clone*** and ***dup***. ***dup*** creates a new copy of the object, but its original members are shared. Which means, if you copy an object then modify the attributes in the copy, this will affect the original. Also, it does not copy methods from the original object. In addition, it does not preserve the frozen state.

The two methods: ***initialize_dup*** and ***initialize_clone*** must be overridden to ensure a specialized copy. A shared method between them is ***initialize_copy*** which is called when they are done.

```
class Person
```

```
        def initialize_clone(source)
                c = super
                c.byear += 2
                c
        end
end

p = Person.new("Karim", 1986)
p3 = p.clone
```

Value equality

There are three methods for equality:

- `==` It is used to test reference equality. Typically, this method is overridden in descendant classes to provide class-specific meaning.
- *equal?* Same as `==`, but it is used to determine object identity and should never be overridden.
- *eql?* It is used by *Hash* collection to test members equality. It tests if the two objects refer to the same hash key.

```
class Person
        def eql?(other)
                name == other.name
        end
end

p = Person.new("Karim", 1986)
p3 = p.clone
puts "p.eql? p3? #{p.eql? p3}"
```

Object comparison

To compare two objects, the method `<=>` is used. When overridden, it should return: -1 when self is smaller than other, 0 when self is equal to other and 1 when self is bigger than other. Nil means the two values could not be compared.

When you define <=>, you can include Comparable to gain the methods <=, <, >=, > and between?.

```
class Person
        include Comparable
        def <=>(other)
                cmp = byear <=> other.byear
                if cmp == 0
                        name <=> other.name
                else
                        cmp
                end
        end
end

p = Person.new("Karim", 1986)
p2 = Person.new("Karim+1", 1986)
p3 = p.clone
puts "p <=> p3? #{p <=> p3}"
puts "p <=> p2? #{p <=> p2}"
puts "p < p2? #{p < p2}"
```

Hash code

To generate a hash value for an object, the method ***hash*** is used. It must follow the condition: a.eql?(b) implies a.hash == b.hash, because this function is used with ***eql?*** by the class ***Hash*** to determine if two objects reference the same hash key.

The hash value for an object may not be identical across invocations or implementations of Ruby (strings case). If you need a stable identifier across ruby invocations and implementations you will need to generate one with a custom method.

```
1 class Person
2         def hash
3                 name.hash
4         end
5
6 end
7
8 p = Person.new("Karim", 1986)
9 puts "HashCode: #{p.hash}"
```

Discussion

OO languages can be considered as class-based or prototype-based, based on their class style. Prototype-based languages use a data structure or a closure (which is a special data structure) to act like a class. For instance, Javascript uses closures (even ES6 with its new syntax) and Lua uses meta-tables or closures. All of these are scripting languages; but not every scripting language is prototype-based. Perl considers packages as classes; as for PHP, Python and Ruby, they have standalone mechanisms for creating classes.

A class can be considered as an object in many OO languages. In prototype-based languages, it is clear that a prototype is, in fact, an object. Javascript defines a class using closures which are functions; as for Lua, a class can be defined using a metatable. Perl is a class-based OO language which does not have a specific notation for classes; instead it uses packages. Unlike C++, a class in Java, Python and Ruby has its own type: java.lang.Class, type and Class respectively.

Some OO languages have a universal superclass (base class) for their instantiated objects. In prototype-based languages, you can consider the type of the prototype(class) as a superclass. C++ and PHP do not have a universal superclass for their objects. Other languages define a universal superclass: Java (java.lang.Object), Perl (UNIVERSAL), Python (object) and Perl (Object).

A class or a prototype is meant to be a template used to create new objects providing their attributes and methods. To access these members from inside the object, a keyword is used to refer to the current object: usually ***this*** or ***self***. In C++ and Java, this keyword is optional, unless there is a confusion with a method's parameter. In other languages, it is mandatory As for shared members between objects of the same class, they are called class members (static members). Languages such as C++ and Java afford keywords to define static members. Some define static attributes outside the constructor: Python and Ruby. Others can emulate static attributes by defining global variables such as the case of Perl.

Properties are something between fields and methods: when you read or write it, the syntax is like fields; but these read/write operations are translated to methods called getters/setters. C++ and Java do not afford such members; as for python, it uses annotations. Perl's setters act like normal methods rather than triggered upon variable assigning.

Table 3.1 represents a comparison between our OO languages based on their abstraction properties. An OOP language can be one of the two types: class-based or prototype-based. A class or prototype can have a type or not, which can be the type of its objects as well. This is the case of prototype-based languages, others have a different type as base class for objects. Current object can access its dynamic members differently from language to another. As for static members, some languages afford standalone mechanisms to define them, while others are not designed primarily to have them. Some languages afford a third member type: properties, which are something between fields and methods.

Table 3.1: Abstraction comparison

Language	Style	Class type	Definition	Universal superclass	Current object	Static members	Property
C++	class based	/	*class* MyClass {}	/	*this* (**optional**)	*static*	/
Java	class based	Class	*class* MyClass { }	Object	*this* (**optional**)	*static*	/
Javascript	prototype based	function which is an object	*function* MyClass(){}; or *class* MyClass {}	Object	*this* (**mandatory**)	Not really; *static*: for functions in ES6	Object. defineProperties
Lua	prototype based	metatable	*local* MyClass = {} MyClass.__index = MyClass	Not really	*self*, with the colon syntax; any variable with dot syntax (**mandatory**)	Not really	Using *__index* and *__newindex*
Perl	class based	package	*package* MyClass; 1;	UNIVERSAL	The first variable shifted from the context array (**mandatory**)	Not really	Not really
PHP	class based	?	*class* MyClass {}	/	*$this* (**mandatory**)	*static*: for fields	Magic methods: *__get* and *__set*
Python	class based	classobj: old style; type: new style	*class* MyClass: *pass* in new style *class* MyClass(*object*): *pass*	object (new style classes)	Any first parameter, *self* is convention (**mandatory**)	Annotation	Annotation
Ruby	class based	Class	*class* MyClass *end*	Object (inherits from BasicObject)	@field	@@field; self.method	*def* getter; *def* setter=(value)

CHAPTER 4

ENCAPSULATION

Encapsulation is an important concept of OOP. It is the mechanism of restricting direct access to some members of a class. It helps managing complexity when the source code is debugged. If a field is set to be accessed everywhere in a source code, it will be difficult to find errors related to it. Also, if a field of a class (A) is used by a class (B) then it is changed (either its name or how it gets assigned), all the code where it is appears in (B) have to be changed as well. This chapter has one purpose which is showing different visibility modes and how they are implemented in each language. In term of restricting access to some members, there are two views: "Many programmers are irresponsible, dim-witted, or both." and "Programmers are responsible adults and capable of good judgment". So, a programming language may fall into one or another.

1 Public members

Public visibility mode is used to access a member of a class everywhere. Public methods of a class provide an interface that allows them to be called so the object can afford some behavior. As for fields, public visibility mode is not preferable since it breaks encapsulation, and it has limited uses:

- Constant fields can be exposed to the outside classes.
- Classes which represent just data, and unlikely to be changed in the future.

1.1 C++

Public members are defined inside the class header using the modifier ***public***. Every member coming after that modifier is considered as public. Using ***struct*** instead of ***class***, all members will be public by default.

```
1 class Person
2 {
3 public:
4     int luckyNumber;
5     Person(std::string name);
6     void info();
7 };
```

These members can be accessed anywhere

```
1 int main()
2 {
3     Person p = Person("Karim_p");
4     p.luckyNumber = 5;
5     p.info();
6     s.info();
7     return 0;
8 }
```

An important remark: when you compile multiple files using g++, you have to include all cpp files

```
$ g++ app.cpp person.cpp student.cpp frnd.h
```

If you include a header twice or more, you may have an error telling you that you defined the class more than once. To correct this, you have to use ***#include guard***.

```cpp
#ifndef PERSON_H
#define PERSON_H
class Person
{
};
#endif /* PERSON_H */
```

1.2 Java

Every public member in Java must be preceded by the keyword ***public***.

```java
package encapsulation.core;

public class Person {

    public int luckyNumber;
    public Person(String name) {
    }

    public void info() {
    }
}
```

These members can be accessed anywhere

```java
package encapsulation.main;

import encapsulation.core.*;

public class App {

    public static void main(String[] args) {
        Person p = new Person("Karim_p");
        p.luckyNumber = 5;
        p.info();
    }

}
```

1.3 Javascript

All fields with the keyword ***this*** are public. All methods defined using the same keyword inside the constructor or using the prototype are public.

```javascript
function Person(name) {
    this.luckyNumber = 0;
}
Person.prototype.info = function(){
}
```

Which can be accessed anywhere

```
var Person = require("./person.js");
var p = new Person("Karim_p");
console.log(p.luckyNumber);
p.info();
```

Actually, objects in Javascript can be updated by assigning them new fields and methods dynamically. This gives you the power to upgrade existing prototypes (classes) and decorate objects without the need to create more classes. Also, as ***uncle Ben*** of ***Spider-man*** once said: "*With great power comes great responsibility*".

```
let Person = function(name) {
    this.name = name;
}

function create(name, members) {
    let p = new Person(name);
    p = Object.assign(p, members);
    return p;
}

let student = create ("Karim_s", {
    year: 2
});

let teacher = create ("Karim_t", {
    courses: ["C1", "C2"]
});
```

In the following code, two objects with different fields are created. Then, a function which prints the different fields of a Person is defined by looping over an object's members. Just a note: you have to verify that the member is not a function, since functions in Javascript are first class objects. Which means: they are a type (***function***), they are instances of ***Object*** and they can be treated as any variable.

```
function info (person) {
    console.log(person.name);
    for(let member in person){
        if (typeof member != "function") {
            console.log(member, " = ", person[member]);
        }
    }
}

info(student);
info(teacher);
```

1.4 Lua

Using tables to create objects, all members are accessible outside the class, hence public.

```
local Person = {}
Person.__index = Person

function Person.new(name)     -- The constructor
    local self = {}
    self.luckyNumber = 1;
    return setmetatable(self, Person)
end
function Person:info()
end

return Person
```

You can access them anywhere

```lua
local Person = require "person"

local p = Person.new("Karim_p")
p:info()

p.luckyNumber = 6
```

1.5 Perl

> Larry Wall
>
> Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.

You get the idea: all members are public; but, there is always a hack to limit visibility. Lets see the normal case where every field and method are public.

```perl
package Person;

sub new {
    my $class = shift;
    my $self = {
        name => shift,
    };
    bless $self, $class;
    return $self;
}

sub info {
    my( $self ) = @_;
    print "My name: ", $self->{name},"\n";
    print "----------------------------\n";
}

1;
```

They can be accessed outside the class

```perl
$p = new Person("Karim_p");
$p->info();
$p->{name} = "other name";
```

1.6 PHP

The default visibility mode in PHP is public. A method defined without a visibility modifier or a field defined using ***var*** are public by default. It is good practice to prefix class members with a visibility modifier, since not doing that is just a mean to keep compatibility with versions before 5.1.3.

```php
class Person {

    public $luckyNumber = 0;
    var $luckyNumber2 = 0;//public by default
    public function __construct( $name) {
    }
    public function info() {
    }
```

```
}
```

The member with public visibility mode can be accessed anywhere.

```
require_once "person.php";
$p = new Person("Karim_p");

$p->info();
$p->info();
```

1.7 Python

All class members are public by default; it is the only visibility mode in Python.

```
class Person(object):
        def __init__(self, name):
                self.luckyNumber = 0
        def info(self):
                print("My lucky number is: " + str(self.luckyNumber))
```

The members can be accessed anywhere

```
from person import Person
from student import Student

p = Person("Karim_p")
p.luckyNumber = 5
```

1.8 Ruby

By default methods are public except for ***initialize*** method and the global methods defined under the object class which are always private. If desired, you can use the keyword ***public*** in a line and all methods defined after it will be public. Fields are not public unless they are defined as constants. To access them like public in other languages, you have to define accessor methods (the fields have to be properties). In Ruby, public getters and setters are called ***attribute readers*** and ***attribute writers***.

```
class Person
    attr_accessor :luckyNumber #attr_reader, attr_writer
    def initialize(name)
        @luckyNumber = 0
    end
    def info
    end
    public
    def copy (other)
    end
end
```

The field can be accessed for read and write. Actually, the field is accessed via a getter and a setter which are public by default (because they are methods).

```
p = Person.new("Karim_p")
s = Student.new("Karim_s")

p.luckyNumber = 5
p.info
p.copy(s)
```

2 Protected members

Protected visibility mode is used to access a member of a class from the class itself and its subclasses. It is used to allow a class accessing its superclass's members directly. If a member is defined as protected, keep in mind that changing it later may break subclasses.

2.1 C++

Protected members are defined inside the class header using the modifier ***protected***. Every member coming after that modifier is considered as protected.

```
1 class Person
2 {
3 protected:
4     std::string t;
5 };
```

These members can be accessed only from the class itself or its subclasses

```
1 #include "student.h"
2
3 Student::Student(std::string name): Person(name)
4 {
5     this->t = "student";//protected member
6 }
```

They cannot be accessed elsewhere

```
1 int main()
2 {
3     Person p = Person("Karim_p");
4     //p.t = "admin"; //error: protected
5     return 0;
6 }
```

2.2 Java

Protected members are prefixed each by the keyword ***protected***. A protected member is visible inside the class and its subclasses, and also to other classes in the same package. The protected methods can be defined the same as protected fields.

```
1 package encapsulation.core;
2
3 public class Person {
4     protected String t;
5 }
```

The field can be accessed from a subclass (which is not necessarily in the same package)

```
1 package encapsulation.core;
2
3 public class Student extends Person {
4
5     public Student(String name) {
6         super(name);
7         t = "student";
8     }
```

```
9
10 }
```

It can be accessed from classes of the same package

```
1  package encapsulation.core;
2
3  public class App2 {
4
5      public static void main(String[] args) {
6          Person p = new Person("karim_p");
7          p.t = "admin";
8      }
9
10 }
```

And cannot be accessed from a class outside the package and not inheriting from their class.

```
1  package encapsulation.main;
2
3  import encapsulation.core.*;
4
5  public class App {
6
7      public static void main(String[] args) {
8          Person p = new Person("Karim_p");
9          //p.t = "admin"; //error: protected
10     }
11
12 }
```

It is also important to point out that protected members can be accessed from another object of the same class.

```
1  public class Person {
2      protected String t;
3      public void copy(Person other) {
4          t = other.t;
5      }
6
7  }
```

2.3 Javascript

There is no protected members in Javascript. You can find some hacks on the web, but they can be expensive in term of memory and processing power. To emulate such mechanism, you have to know the caller object. If you insist on having protected and private members, some libraries can be helpful such as ***mozart***[1].

2.4 Lua

There is no protected members in Lua, but you can use a naming convention to specify them.

[1] mozart: https://github.com/philipwalton/mozart

2.5 Perl

Perl can capture the caller (the location or package where a method was called) using the keyword ***caller***. Also, the base class ***UNIVERSAL*** provides a method ***isa*** which returns true if the object is blessed to the given type or inherits from it. Adding to it the keyword ***__PACKAGE__*** which captures the current package and exception management ***die***, Bingo! we can limit access. We cannot limit visibility: the method is still visible, but we raise an exception if the access does not serve our purpose (rir of PerlMonks, 2006).

```
1 package Person;
2 # ...
3 sub protected_fct {
4         die "protected_fct is protected!" unless caller->isa(__PACKAGE__);
5         # ... The rest of the code
6 }
7 1;
```

As for fields, you can hide them entirely (private) then use getters and setters to access them. Since these getters/setters are methods, you can limit their access.

2.6 PHP

Protected members are prefixed, each, by the keyword ***protected***.

```
1 class Person {
2     protected $t;
3 }
```

They can be accessed from the class itself or its subclasses

```
1 class Student extends Person {
2 
3     public function __construct($name) {
4         parent::__construct($name);
5         $this->t = "student";
6     }
7 
8 }
```

They cannot be accessed elsewhere

```
1 require_once "person.php";
2 $p = new Person("Karim_p");
3 //$p->t = "admin"; //error: protected
```

An object can access a protected member of another of the same class

```
1 class Person {
2     public function copy($other) {
3         $this->t = $other->t;
4     }
5 
6 }
```

2.7 Python

In Python, all class members are publicly visible. But, Python community use coding conversions to tell developers about a field's visibility. Some developers use underscore _ before the class member to signal that it is protected. However, looking to this convention's definition in Python's docs: *"a name prefixed with an underscore should be treated as a non-public part of the API. It should be considered an implementation detail and subject to change without notice."*, this suggests that it is a private member convention. But, since there is another convention for private members (double underscore), you can consider one underscore as protected.

```
class Person(object):
        def __init__(self, name):
                self._t = "person"
```

It can be accessed from subclasses

```
from person import Person

class Student(Person):

        def __init__(self, name):
                Person.__init__(self, name)
                self._t = "student"
```

Or, from any class, module or piece of code

```
from person import Person
from student import Student

p = Person("Karim_p")
p._t = "admin"
```

The idea is to tell developers that this class member is protected and it can be changed in the future, not to enforce the visibility on them. If they want to access it directly, it is their choice.

2.8 Ruby

All fields are protected by default and there is no other visibility mode for them. Methods are public by default, but can be defined as protected using either ***private*** or ***protected*** keywords.

```
class Person
    attr_reader :t, :name
    protected :name
    def initialize(name)
        @name = name
        @num = @@nbr
        @t = "person"
    end
    private
    def info1
        puts "My number: #@num"
    end

    protected
    def info2
        puts "I am a:  #@t"
    end
end
```

Methods defined using ***private*** or ***protected*** can both be accessed from sub-classes.

```
class Student < Person
    def info_student
        puts "info student"
        info1
        info2
    end

end
```

The difference is that those with ***private*** cannot be accessed from another object even if it is of the same class. In the contrary, those with ***protected*** can be accessed from the an object of the same class.

```
class Person
    def other_info (other)
        puts "other info"
        #other.info1 #private
        other.info2
    end
end
```

The fields and the protected methods cannot be accessed outside the class and its sub-classes

```
p = Person.new("Karim_p")
#p.num = 6 #can't access the attribute
#puts "p.name = #{p.name}" #private
# p.info1 #private
# p.info2 #protected
```

3 Private members

Fields are set to be private so they can not be changed directly. Instead, setters and getters can be used to access them because they are more appropriate for that. Private methods are a good way to break tasks into smaller pieces of code. Also, they can prevent duplication in case many public functions of the same class have a shared behavior.

3.1 C++

Private members are defined inside the class header using the modifier ***private***. Every member coming after that modifier is considered as private.

```
class Person
{
private:
    std::string name;
    int num;
    static int nbr;
};
```

These members can be accessed only from the class itself and not its subclasses

```
Student::Student(std::string name): Person(name)
{
    //this->name = "other name";//private member: error
}
```

Also, an object can access another object's members if they are of the same class whatever their visibility is.

```
void Person::copy(Person other)
{
    name = other.name;
    t = other.t;
}
```

3.2 Java

Private members are prefixed each by the keyword ***private***. A private member (field or method) is visible only inside the class.

```
package encapsulation.core;

public class Person {
    private int num;
}
```

It cannot be accessed from a subclass

```
public class Student extends Person {
    public Student(String name) {
        super(name);
        //this.num = 80;//cannot be accessed: private
    }

}
```

Also, it cannot be accessed from another class

```
public class App {

    public static void main(String[] args) {
        Person p = new Person("Karim_p");
        //p.num = 6;//error: private
    }

}
```

It is also important to point out that private members can be accessed from another object of the same class

```
public class Person {
    protected String t;
    public void copy(Person other) {
        num = other.num;
    }

}
```

3.3 Javascript

There is a convention to use underscore _ as a mean to signal a class member as private. But the member stays accessible as public. Another way is to use closures (*the combination of a function and the lexical environment within which that function was declared*), by defining a field inside the constructor using the keyword ***var*** or ***let*** (Crockford, 2001). But, any method using this variable must be defined inside the

constructor. This means each time you create a new instance (object), a new method will be created for this object.

```
module.exports = Person;

function Person(name) {
    this._name = name;
    var num = Person.nbr;
    Person.nbr++;

    this.info1 = function(){
        console.log("My number is: ", num);
    }
}

Person.nbr = 0;
```

The private member defined by convention can be accessed anywhere. In contrary, the member defined with closures cannot be accessed.

```
var Person = require("./person.js");
var p = new Person("Karim_p");
console.log(p._name);
console.log(p.num);//undefined
```

3.4 Lua

Using closures, you can create internal fields inside the constructor of an object (Lua users, 2011). The functions which access this field must be defined inside the constructor.

```
local Person = {}
Person.__index = Person

function Person.new(name)     -- The constructor
    local self = {}
    local _name = name;
    self.info1 = function()
        print("My name: " .. _name)
    end
    return setmetatable(self, Person)
end
return Person
```

You cannot access it outside the class. But, you can create a new public field dynamically which will not replace the internal one.

```
local Person = require "person"

local p = Person.new("Karim_p")
-- print("name outside= " .. p.name) -- error: nil value
p.name = "other name"
```

3.5 Perl

There is a convention to start private members with underscore _, but they still are accessible. For methods, you can use ***caller***, ***eq***, ***__PACKAGE__*** and ***die*** to limit access though they still are visible (radiantmatrix of PerlMonks, 2006). If the method is not called from inside the package (which is the class in our case), it will raise an exception and stop executing.

```perl
package Person;
# ...
sub private_fct {
die "private_fct is private!" unless caller eq __PACKAGE__;
# ... The rest of the code
}
1;
```

As for fields, you can hide them entirely (private) using many ways.

Using scope and object reference

This is a way I thought of in order to hide internal fields of an object. Using a container hash, each object's reference is considered as a key and the value is another hash containing private fields names and their values. The hash must be defined using ***my*** and not ***our*** so it will not be accessible outside the package. You can put your package inside a code block so the container hash will not be accessible to the rest of code in the same file. You can consider this hash as a private static member. To get an object's reference in memory, you can use ***refaddr*** function.

```perl
{
    package Person;
    use Scalar::Util 'refaddr';
    my %private;

    sub new {
        my $class = shift;
        my $self = {};#no public members
        bless $self, $class;

        $private{refaddr $self} = {
            name => shift
        };
        return $self;
    }
    1;
}
```

Then, you can access any field of an object using the instruction **$private{refaddr $obj}->{name}**. You can, also, create a private getter/setter if you do not want to repeat that instruction every time you want to access a private field.

```perl
{
    package Person;
    use Scalar::Util 'refaddr';
    # ...
    sub name {
        die "name is private!" unless caller eq __PACKAGE__;
        my $self = shift;
        if (my $value = shift) { # setter
            $private{refaddr $self}->{name} = $value;
        }
        return $private{refaddr $self}->{name}; #getter
    }
    sub copy {
        my( $self, $other ) = @_;
        $self->name($other->name);
        $self->t($other->t);
    }
    1;
}
```

If you want a protected access, just modify the setter/getter to do so. In addition, if you want to prevent modifying the content of a protected field from a subclass, just delete the setter code block.

Using closures

This solution is what you will find on the web (btrott of PerlMonks, 2000). Instead of using hashes as objects, you may implement them as closures. A closure is a subroutine reference that has access to the lexical variables that were in scope when it was created. The idea is to define a subroutine that will act as an setter/getter method and bless it into the class instead of the hash. The solution presented here is a simplified one, with some restrictions on closure. Here I forbid access outside the class to make all fields private, then you can create second hand setters/getters for those you want to grant access to.

```
1  package Person;
2
3  sub new {
4      my $class = shift;
5      my $self = {
6          name => shift,
7          t => "person"
8      };
9
10     my $closure = sub {
11         die "cannot access fields outside the class" unless caller eq __PACKAGE__;
12         my ($field, $value) = @_;
13         die "no field called: $field" unless exists $self->{$field};
14         if ($value) {#setter
15             $self->{$field} = $value;
16         }
17         return $self->{$field};#getter
18     };
19
20     bless $closure, $class;
21     return $closure;
22 }
23
24 sub t {
25     die "t is protected!" unless caller->isa(__PACKAGE__);
26     my($self, $value )= @_;
27     return $self->("t", $value);
28 }
29 1;
```

3.6 PHP

Private members are prefixed, each, with the keyword ***private***.

```
1  class Person {
2      private $name;
3  }
```

They can be accessed only from the class itself and not its subclasses

```
1  class Student extends Person {
2
3      public function __construct($name) {
4          parent::__construct($name);
5          //$this->name = "other name";//cannot be accessed: private
6      }
7
8  }
```

They cannot be accessed elsewhere

```php
require_once "person.php";
$p = new Person("Karim_p");
//$p->num = 6;//error: private
```

An object can access a protected member of another of the same class

```php
class Person {
    public function copy($other) {
        $this->name =$other->name;
    }

}
```

3.7 Python

As said before: all members are public (A remainder so you don't think otherwise). But, there is a mechanism used to avoid name clashes of names with names defined by subclasses, called ***name mangling***. When a field is prefixed by a double underscore __, its name will be changed to **_classname__membername**.

```python
class Person(object):

        nbr = 0

        def __init__(self, name):
                self.__num = Person.nbr
```

It can be accessed from subclasses using the new mangled name

```python
from person import Person

class Student(Person):

        def __init__(self, name):
                Person.__init__(self, name)
                self.__num = 7 #won't change the num
                self._Person__num = 8
```

Also, it can be accessed using the new name from any class, module or piece of code. Trying to read the field as it is defined in the class will raise an error since it does not exist with that name anymore. But trying to assign a value to the field will work, because you are actually creating a new field of the object dynamically.

```python
from person import Person
from student import Student

p = Person("Karim_p")
#n = p.__num #error: no attribute __num
p.__num = 6 #no error but no modification either
p._Person__num = 10
```

3.8 Ruby

There are no private members in Ruby, which are visible just inside their class. Even if you try to limit access of a field to read-only, it can be accessed from sub-classes. This is because the field itself is protected and ***attr_reader*** creates a getter for this field.

```ruby
class Person
    attr_reader :t, :name
    protected :name
    def initialize(name)
        @name = name
    end
end
```

```ruby
class Student < Person
    def initialize(name)
        super(name)
        @name = "#@name 2"
end
```

4 Other visibility modes

4.1 C++

public, protected and private are the only visibility modes applied to class members. But, a class can allow an external function of another class to access its members whatever their visibility mode is. This is known as ***friend function*** and ***friend class*** respectively. They can be declared inside the class header using the keyword ***friend***.

```cpp
class Person
{

    friend class Frnd;
    friend void info_fct(Person p);
protected:
    std::string t;
private:
    std::string name;
};
```

The new class can access all fields even the private ones.

```cpp
class Frnd
{
public:
    static void info(Person p) {
        std::cout << "Friend class- name: " << p.name << "\n";
        std::cout << "Friend class- job: " << p.t << "\n";
    }
};
```

The same thing for friend functions.

```cpp
void info_fct(Person p) {
    std::cout << "Friend function- name: " << p.name << "\n";
    std::cout << "Friend function- job: " << p.t << "\n";
}
```

There are some notes about friendship:

- It is not transitive: a friend of a friend is not a friend.
- It can not be inherited: subclasses of a friend class are not friends.

4.2 Java

Another visibility mode is ***package visibility*** where the members of a class are only accessed from classes of the same package. If you don't prefix the member by any of the three visibility modes, it means it is a package member.

```
1  package encapsulation.core;
2
3  public class Person {
4      int luckyNumber2;
5  }
```

It can be accessed from classes of the same package

```
1  package encapsulation.core;
2
3  public class App2 {
4
5      public static void main(String[] args) {
6          Person p = new Person("karim_p");
7          p.luckyNumber2 = 8;
8      }
9
10 }
```

But cannot be accessed outside the package

```
1  package encapsulation.main;
2
3  import encapsulation.core.*;
4
5  public class App {
6
7      public static void main(String[] args) {
8          Person p = new Person("Karim_p");
9          //p.luckyNumber2 = 6;//error: package visibility
10     }
11
12 }
```

Discussion

Members visibility can have three main modes: public, protected and private. Some languages give full access to class members; in this case, the visibility mode can be indicated via the API's documentation. In this type of languages (such as Javascript, Lua, Perl and Python), it is common to use coding conventions such are the underscore to indicate different visibility modes. There are languages (such as C++, Java and PHP) which afford the ability to choose between these three visibility modes. There are others (such as Ruby) which limit access to some members (mostly fields as protected).

Table 4.1 represents a comparison between OO languages based on their visibility modes: public, protected and private. In this table, the word **member** stands for the entire definition of a method or a field. In Perl and Lua, the object is created inside the constructor which is referred to as **myobject**.

Table 4.1: Encapsulation comparison

Language	Private	Protected	Public
C++	private: members	protected: members	public: members
Java	private member	protected member	public member
Javascript	None (can use closures scope)	None	this.member MyClass.prototype. member
Lua	None (can use closures scope)	None	myobject.attribute MyClass:method
Perl	None	None	$myobject->{attribute} $myobject = {attribute => value} sub method()
PHP	private member	protect member	public member
Python	None	None	self.attribute def method()
Ruby	None	Attributes are always protected private :method protected :method	Methods are public by de-fault public :method

CHAPTER 5

INHERITANCE

One of the most controversial core concepts of OOP is inheritance. It enables the definition of a new class called subclass based on an existing one called superclass. This allows reusing visible members of the superclass and add additional specific ones. Many languages create a relationship "isa" between a subclass and its superclass (which is called subtyping). When a class inherits from many superclasses at once, that is called multiple inheritance. This is not possible in all languages, since it leads to many problems such as diamond problem. In this chapter, the journey begins by a little walk through single harmless inheritance. Then, you will have an incursion into the jungle of abstract and final classes and methods. Finally, you will meet the monster: multiple inheritance.

1 Single inheritance

In this section, simple inheritance will be presented by trying to answer some questions for each programming language:

- Do you have to define a constructor for the new subclass? i.e. can the constructor be inherited?
- Can we change members visibility?
- How to access superclass's and grandparent class's members?

1.1 C++

If the superclass has the default constructor (without parameters), you do not have to define a constructor explicitly unless you want to include some other initializations. If the superclass defines just constructors with parameters, you have to define the constructor of your new subclass.

```
1  class Person
2  {
3  public:
4      Person(std::string name);
5      void info();
6      int luckyNumber;
7
8  private:
9      std::string name;
10 };
```

In C++11, you can inherit the constructor using the keyword ***using***;

```
1  class Professor: private Person
2  {
3  public:
4      using Person::Person;
5  };
```

In C++, to extend a class, you have to define a visibility mode: public, protected or private. Public mode preserves members access as they are defined in the superclass. As for Protected mode, public members of the superclass will be considered as protected in the new class. While Private mode considers every member of the superclass as private.

```
class Student: public Person
{
public:
    Student(std::string name, int grade);
    void info();
    int luckyNumber;

private:
    int grade;
};
```

Since C++ accepts multiple inheritance, there is no keyword for superclass. Instead, you can call a method of the parent or grandparent using their names followed by *::* then the methods name.

```
Student::Student(std::string name, int grade):
    Person(name)
{
    this->grade = grade;
}

void Student::info()
{
    Person::info();
    std::cout << "My grade: " << this->grade << "\n";
    std::cout << "My luckyNumber2: " << this->luckyNumber << "\n";
}
```

1.2 Java

When extending a class which has the default constructor (without parameters), you do not have to define a constructor explicitly unless you want to include some other initializations. But, if the superclass defines just constructors with parameters, you have to define the constructor of your new subclass.

```
public class Person {

    public int luckyNumber;
    private String name;

    public Person(String name) {
    }

    public void info() {
    }

    private void info2() {
    }

}
```

In this example, if we want to extend the class, we have to define a constructor explicitly. This constructor has to call the superclass's explicitly using the keyword ***super***. This call must be the first thing to do before any other instruction. Also, we can override its methods and reuse those of the superclass using the keyword ***super***.

```
public class Student extends Person {
```

```
2
3       private int grade;
4
5       public Student(String name, int grade) {
6           super(name);
7           this.grade = grade;
8       }
9
10      public void info(){
11          super.info();
12          System.out.println("My grade is: " + grade);
13      }
14
15  }
```

You cannot assign weaker access privileges when overriding a method, but the inverse is permitted. As for fields, you cannot override a field. If you define a field with the same name as in the superclass, you will still access that of the superclass inside the subclass using the keyword ***super***; this is called ***variable hiding***.

```
1   public class Professor extends Person {
2       static int num = 0;
3       private int luckyNumber = 9;
4       public Professor() {
5           super("Professor" + num);
6           num++;
7       }
8
9       public void info2() {
10          System.out.println("Professor info2, luckyNumber: " + luckyNumber);
11          System.out.println("Professor info2, super.luckyNumber: " + super.luckyNumber);
12      }
13
14  }
```

If the field of the superclass is public and you define a private field with the same name in its subclass, you will no longer be able to access it outside the subclass.

```
1       public static void main(String[] args) {
2           Person pe = new Person("Person1");
3           Student st = new Student("Student1", 15);
4           Professor pr = new Professor();
5           pe.luckyNumber = 10;
6           st.luckyNumber = 20;
7           //pr.luckyNumber = 30;
8       }
```

You cannot access grandparent's members directly using ***super.super***, you can just access the parent's using ***super***.

```
1   public class GradStudent extends Student {
2
3       public GradStudent(String name, int grade) {
4           super(name, grade);
5           //super.super.info();//error: <identifier> expected
6           //super.super.luckyNumber = 10;//error: <identifier> expected
7       }
8
9   }
```

1.3 Javascript

In Javascript, all class members are public. So, visibility modes stay as they are when extending a class.

ECMAScript 5 (ES5)

Lets define a class by using a function as a constructor (as usual: ES5).

```
1 function Person(name) {
2     this.luckyNumber = 0;
3     this._name = name;
4 }
5
6 Person.prototype.info = function(){
7     console.log("My name: ", this._name);
8     console.log("My lucky number is: ", this.luckyNumber);
9 }
```

To call the superclass's constructor, we use the function ***call*** which takes the context of the current object ***this*** as its first argument. This will allow the superclass to assign and initialize new fields to the current object (this). To inherit a class's methods, the subclass's prototype must be an instance of this class's one. So, a new object based on the superclass's prototype must be created and assigned to the new class's prototype. Be aware not to assign the superclass's prototype directly (the two prototypes will refer to the same object). If you do that, any method specific to the subclass will be defined to the superclass too. After assigning the prototype, you have to set the constructor to the current class because the cloned prototype's constructor will be referring the superclass's constructor.

```
1 function Student(name, grade) {
2     Person.call(this, name);
3     this._grade = grade;
4 }
5
6 Student.prototype = Object.create(Person.prototype);
7 Student.prototype.constructor = Student;
```

To call a method from the superclass's prototype, you can use the function ***call*** by passing the context as first argument. In fact, you can call methods of any class even if it is not an ancestor and apply it to the current context, in condition that it uses compatible fields.

```
1 Student.prototype.info = function(){
2     Person.prototype.info.call(this)
3     console.log("My grade: ", this._grade);
4 }
```

The constructor of the superclass can be called anywhere inside the subclass's constructor. You have to be cautious when doing that, because the fields values can be overridden. Also, not calling the superclass's constructor will not result in an error, but some fields and initializations may be found missing.

```
1 function Professor() {
2     //Person.call(this, "Professor");
3 }
4
5 Professor.prototype = Object.create(Person.prototype);
6 Professor.prototype.constructor = Professor;
```

ECMAScript 2015 (ES6)

Classes in new Javascript are primarily syntactical sugar over its existing prototype-based inheritance (MDN web docs, 2018a). The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

```
class Person {
    constructor(name) {
        this.luckyNumber = 0;
        this._name = name;
    }

    info() {
        console.log("My name: ", this._name);
        console.log("My lucky number is: ", this.luckyNumber);
    }
}
```

To call the superclass's constructor, we use the keyword ***super***. This keyword allows us to access the superclass's methods as well.

```
class Student extends Person {

    constructor(name, grade) {
        super(name);
        this._grade = grade;
    }

    info(){
        super.info();
        console.log("My grade: ", this._grade);
    }

}
```

Like old Javascript, the constructor of the superclass can be omitted without errors. But, some fields which must be defined in the superclass can be absent from the object resulting in malfunctioning of the methods using them.

```
class Professor extends Person {}
```

1.4 Lua

In lua, the constructor is a method which creates a new object (table) and assign the class as its meta-table. In our case, the new object will have a field named **luckyNumber** which is the same as its meta-table (a reference to it). If you want this object to have its own, you have to set the field using the same name.

```
local Person = {luckyNumber = 0}
Person.__index = Person

function Person.new(name)    -- The constructor
    local instance = {name = name}
    setmetatable(instance, Person)
    return instance
end

function Person:info()
    print("My name: " .. self.name);
    print("My lucky number: " .. self.luckyNumber)
end
```

To extend a class, all you have to do is creating the table representing it, then set the superclass as its meta-table. In this case, if you call a member (field or method) on this new class and it is not found, it will lookup its meta-table. You have to create the object from the superclass, so it will contain the same fields, and add specific fields to it. Then, set the current class as its meta-table; otherwise, the new object will lookup its methods in the superclass and not in the current class.

As for calling superclass's methods, you can use the dot call (.) instead of the colon (:) and pass the context (current object, ***self***) as first parameter.

```lua
local Student = {}
setmetatable(Student, Person) -- Student extends Person
Student.__index = Student

function Student.new(name, grade)     -- The constructor
    local instance = Person.new(name) --get the superclass's object
    instance.grade = grade
    setmetatable(instance, Student) -- object extends Student
    return instance
end

function Student:info()
    Person.info(self)
    print("My grade: " .. self.grade)
end
```

A fancy method to instantiate a class is to pass an object containing some predefined fields to its constructor. But, if we do not pass a field needed by one of its methods, this will result in an error when calling that method. One way to fix this is by verifying if the field has been passed or not; if it was not, create a default one.

```lua
local Professor = {}
setmetatable(Professor, Person) -- Professor extends Person
Professor.__index = Professor

function Professor.new(instance)     -- The constructor
    instance = instance or {name = "professor"}
    setmetatable(instance, Professor)
    return instance
end
```

As for accessing other classes' methods, you can call their methods and pass the object as context.

1.5 Perl

A class (package) can extend another using a special variable ***@ISA***. In the extended class's constructor, you can create a new object using the superclass's constructor. This will allow it to have the same fields as its superclass and add specific fields to it. You can call superclass's methods, including the constructor, using the keyword ***SUPER***.

```perl
package Student;
use Person;
our @ISA = qw(Person);

sub new {
    my ($class, $name, $grade) = @_;
    $self = $class->SUPER::new($name);
    $self->{grade} = $grade;
    bless $self, $class;
    return $self;
```

```
}

sub info {
    my( $self ) = @_;
    $self->SUPER::info();
    print "My grade: $self->{grade}\n";
}

1;
```

Like any method, the constructor can be inherited if not overridden. The new class will be created similarly to its superclass.

```
package Professor;
use Person;
our @ISA = qw(Person);

1;
```

1.6 PHP

Lets define a class with some private/public fields and methods.

```
class Person {

    public $luckyNumber = 0;
    private $name;

    public function __construct( $name) {
        $this->name = $name;
    }

    public function info() {
        echo "My name: $this->name\n";
        echo "My lucky number is: $this->luckyNumber\n";
    }

    private function info2() {
        echo "Person.info2\n";
    }

}
```

To extend this class, you have to use the keyword ***extends***. To call the superclass's methods, you need to use a special keyword ***parent***.

```
class Student extends Person {

    private $grade;

    public function __construct($name, $grade) {
        parent::__construct($name);
        $this->grade = $grade;
    }

    public function info(){
        parent::info();
        echo "My grade is: $this->grade\n";
    }

}
```

If you do not define a constructor for the subclass, it will inherit the constructor of its superclass. As for methods visibility, you can pass from more restricted access to less restricted.

```php
class Professor extends Person {

    public function info2(){
        echo "Professor info2\n";
    }

}
```

If you define a field with the same name as one of the superclass, it will override it and not hide it as in Java; which means the parent's methods will use the new one instead (Deschenes, 2014). Using the keyword ***parent***, you can access just the superclass's methods and not those of grandparent class.

```php
class GradStudent extends Student {

    private $grade;

    public function __construct($name, $grade) {
        parent::__construct($name, $grade);
        $this->grade = $grade + 2;
        //parent::parent::info(); //cannot do this
        parent::info();
        //echo parent::luckyNumber;
    }

}
```

1.7 Python

You can call a superclass method (including the constructor) using two ways:

- Using the name of the superclass to call the method and pass the context as first argument
- Using the function ***super*** which takes two arguments: the type and the context. It returns a proxy object that delegates method calls to a parent or sibling class of type.

```python
class Student(Person):

    def __init__(self, name, grade):
        Person.__init__(self, name)
        #super(Student, self).__init__(name)#the same
        self.grade = grade

    def info(self):
        super(Student, self).info()
        #Person.info(self) #the same
        print("My grade: " + str(self.grade))
```

A constructor can be inherited; if you do not define one in the subclass, the superclass's will be used. The ***pass*** statement does nothing; it is required when there is no code to afford.

```python
class Professor(Person):
    pass
```

1.8 Ruby

You can call a superclass's method (including the constructor) from a method of the same name using the keyword ***super***.

```ruby
class Student < Person

    def initialize(name, grade)
        super(name)
        @grade = grade
    end

    def info
        super
        puts "My grade: #@grade"
    end

end
```

If the constructor is not defined, the class will inherit its superclass's. Also, when you override a method, you can change its visibility mode.

```ruby
class Professor < Person

    def info1
        puts "Professor info1"
    end

end
```

To access a superclass method which does not have the same name, there are two ways (Meyer, 2014):

- either by aliasing it before overriding, using ***alias_method***.
- or by binding the current context to a class's method. Using this, you call call even grandparents methods.

```ruby
class GradStudent < Student

    alias_method :parent_info, :info

    def initialize(name, grade)
        super(name, grade)

        puts "======================"
        parent_info # calling info of the parent class
        info # calling info of the current class
        Person.instance_method(:info).bind(self).call
    end

    def info
        puts "GradStudent info"
    end

end
```

2 Abstract class

An abstract class is a class which cannot be instantiated, but can be extended. An abstract method is a method which has to be overridden before it can be used.

2.1 C++

A class is abstract when it has at least one pure virtual method. Hence, you cannot have an abstract class with only fields.

```
1 class Person
2 {
3 public:
4     void info();
5     virtual void info2() = 0;
6 };
```

The pure methods must be overridden in a concrete subclasses (do not forget to override them in the headers).

```
1 #include "person.h"
2
3 class Professor: public Person
4 {
5 public:
6     void info2();
7 };
```

If a class does not override all abstract methods of its superclass, it will be implicitly considered as abstract.

```
1 class Student: public Person
2 {
3 };
```

2.2 Java

An abstract class is defined by the keyword ***abstract***. It does not have to contain an abstract method, which means you can have an abstract class with just fields. If the class has at least one abstract method, you have to define it as abstract explicitly or you will have an error.

```
1  public abstract class Person {
2
3      public void info() {
4          System.out.println("====INFO===");
5          info2();
6      }
7
8      public abstract void info2();
9
10 }
```

The abstract methods must be overridden in the concrete subclasses

```
1  public class Professor extends Person {
2
3      public void info2() {
4          System.out.println("Professor info2");
5      }
6
7  }
```

If a class does not override all abstract methods of its superclass, it has to be defined explicitly as abstract.

```
1 public abstract class Student extends Person {
2 }
```

You can instantiate just the non-abstract classes

```
        //Person pe = new Person();//is abstract; cannot be instantiated
        //Student st = new Student();//is abstract; cannot be instantiated
        Professor pr = new Professor();
        GradStudent gs = new GradStudent();
```

2.3 Javascript

There is no abstract classes or methods either in ECMAScript 5 or ECMAScript 2015. But, you can prevent instantiating a class by verifying if the new object's constructor is the class's (Kazarian, 2016). If so, you throw an exception. As for methods, you throw the exception anyways.

```
function Person() {
    if (this.constructor == Person) {
        throw new Error("Cannot instanciate abstract class");
    }
}

Person.prototype.info = function(){
    console.log("====INFO===");
    this.info2();
}

Person.prototype.info2 = function(){
    throw new Error("Cannot call abstract method");
}
```

If you want a subclass to be abstract too, you have to do the same thing with its constructor. Otherwise, it can be instantiated even if you did not override the abstract method. In this case, you cannot call the abstract method or any other method using it.

```
function Student() {
    Person.call(this);
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;
```

```
function GradStudent() {
    Student.call(this);
}

GradStudent.prototype = Object.create(Student.prototype);
GradStudent.prototype.constructor = GradStudent;

GradStudent.prototype.info2 = function(){
    console.log("GradStudent info2");
}
```

The same analogy goes with ECMAScript 2015 (codes are afforded).

2.4 Lua

There are no abstract classes or methods in Lua. You can define a method which always throws an error when called, and consider it as abstract. As for abstract classes, I could not find or think of a mechanism to prevent calling the constructor of a class and creating a new object (table) without preventing it to be

called inside a subclass constructor. You can prevent the new object from calling the class's methods (all of them) by not setting the class as its meta-table.

```lua
local Person = {}
Person.__index = Person

function Person.new()    -- The constructor
    local instance = {}
    --setmetatable(instance, Person)
    return instance
end

function Person:info()
    print("====INFO====");
    self:info2()
end

function Person:info2()
    error("Cannot call abstract method")
end
```

2.5 Perl

You cannot prevent calling the constructor definitely because it has to be called from its subclasses. But, you can define the constructor as protected: cannot be accessed unless the caller is a subclass. As for methods, you can throw an exception if it is not overridden (glasswalk3r of PerlMonks, 2000).

```perl
package Person;

sub new {
    die "cannot be instantiated" unless caller->isa(__PACKAGE__);
    my $class = shift;
    my $self = {}; #fields
    bless $self, $class;
    return $self;
}

sub info {
    my( $self ) = @_;
    print "====INFO====\n";
    $self->info2();
}

sub info2 {
    die "abstract method"
}

1;
```

The abstract method must be overridden; if not, there will be errors when calling them.

```perl
package Student;
use Person;
our @ISA = qw(Person);

sub new {
    my ($class, @args) = @_;
    $self = $class->SUPER::new(@args);
    bless $self, $class;
    return $self;
}

```

```perl
sub info2 {
    print "Student info2\n";
}

1;
```

2.6 PHP

Classes defined as abstract may not be instantiated. To define an abstract method or class, you have to use the keyword ***abstract***.

```php
abstract class Person {

    public function info() {
        echo "===INFO====\n";
        $this->info2();
    }

    abstract protected function info2();

}
```

Any class that contains at least one abstract method, or does not override one, must also be abstract.

```php
abstract class Student extends Person {}
```

To override an abstract method, you just delete the keyword abstract and afford an implementation.

```php
class Professor extends Person {

    protected function info2(){
        echo "Professor info2\n";
    }

}
```

2.7 Python

To create abstract classes and methods, there exists a module called ***abc*** (abstract base classes). A class's meta-class can be set to be ***ABCMeta***, but the class can be instantiated unless there is at least one abstract method. This means, you cannot have an abstract class with just fields. An abstract method is marked by the decorator ***@abstractmethod***. The abstract methods, contrarily to the usual definition of the word, can have implementations which can be called from the subclasses methods.

```python
from abc import ABCMeta, abstractmethod

class Person(object):
    __metaclass__ = ABCMeta

    def info(self):
        print("====INFO====")
        self.info2()

    @abstractmethod
    def info2(self):
        pass
```

If all the abstract methods are overridden, the class can be instantiated

```
class Professor(Person):

    def info2(self):
        print("Professor info2")
```

If not all abstract methods are overridden, the new class cannot be instantiated

```
class Student(Person):
    pass
```

2.8 Ruby

There are no abstract classes or methods in Ruby. If you want to prohibit instantiating a class, you can raise an exception whenever its constructor is called. But, the constructor will not be useful to other subclasses (Rajhans, 2009). To handle this, you have to verify if the object's class is the current class. As for methods, you raise an exception preventing them from being called.

```
class Person

    def initialize()
        if self.class == Person then
            raise "Abstract class"
        end
    end

    def info
        puts "====INFO===="
        info2
    end

    def info2
        raise "abstract method"
    end

end
```

Even if you don not override the constructor, the subclass will be accessible because the object's class no longer will be the superclass.

```
class Professor < Person

    def info2
        puts "Professor info2"
    end

end
```

If you do not override the abstract method, the subclass can be instantiated but the abstract method and the methods using it will raise an exception.

```
class Student < Person

    def initialize()
        super
    end

end
```

3 Final class and method

A final class is a class which cannot be extended. Likewise, a final method is a method which cannot be overridden. Having final methods does not imply the class has to be final.

3.1 C++

Final class

In early versions, there were no final classes. But, there is a hack to prevent subclasses from instantiating; it does not prevent inheriting directly. You can define a class with a private constructor and a friend type, which means: you cannot instantiate this class but the friend class (type) can (sbi of stackoverflow, 2009).

```
template <typename T>
class MakeFinal
{
private:
   MakeFinal() {}
   friend T;
};
```

Then, you extend that class to make the subclass final: it can access the constructor of its superclass which means you can instantiate it. You can extend that class with no problem, but your new class will be useless since it cannot be instantiated (friend property is not inherited).

```
class Person: virtual MakeFinal<Person> {};

class Student: public Person {};
```

In C++11, life is more easy: just use the keyword ***final*** after the class's name. In this case, the compiler will complain when you extend a final class.

```
class Person final {};

//class Student: public Person {}; //cannot derive
```

Final method

In early C++, a method which is not defined as ***virtual*** should not be overridden (Török, 2010). If you try to override it, you are actually creating a new method with the same name and hiding the super-method. This is not how final methods should act, but the new method will not be accessible from a reference on the superclass (it is not polymorphic).

C++11 comes to the rescue with its new ***final*** keyword, which must be positioned in the end of the method's header. To be final, a method has to be virtual.

```
#include <iostream>

class Person {
public:
    void virtual finalMethod() final {
        std::cout << "Person final method\n";
    };
};

```

```
class Student: public Person {
public:
    //void virtual finalMethod() {};//cannot override
};
```

3.2 Java

A final class is marked using the keyword ***final***. It cannot be extended, and if you try to extend it you will throw a compile error.

```
final class Person {}

//class Student extends Person {}//cannot inherit
```

A final method is also marked using the keyword ***final***. It cannot be overridden, otherwise you will have a compile error.

```
class Person {
    public final void finalMethod(){
        System.out.println("Person final method");
    }
}

class Student extends Person {
    //public final void finalMethod(){}//cannot override
}
```

3.3 Javascript

There are no final classes (objects) in Javascript. But, you can limit instantiating to only the current class by verifying if the new objects constructor is the class itself or not. This is the inverse of what we did in abstract class.

```
class Person {
    constructor() {
        if (this.constructor !== Person) {
            throw new Error("Cannot extend final class");
        }
    }
}

class Student extends Person {}

var pe = new Person();
//var st = new Student();//Error: final class
```

As for final functions, good luck finding a mechanism to do that!

3.4 Lua

No final classes and methods in Lua. As far as I know, there is no way to emulate them either.

3.5 Perl

In Perl, you can prevent a class from being instantiated. This can be done using the class name in the constructor and the package name. If they are equal, then it is OK; otherwise, this will raise an exception.

```
1  package Person;
2
3  sub new {
4      my $class = shift;
5      die "final class" unless $class eq __PACKAGE__;
6      my $self = {}; #fields
7      bless $self, $class;
8      return $self;
9  }
10
11 1;
```

You can create a subclass out of our previous class. But, when you try to instantiate it, this will raise an exception. Unless you do not call the parent constructor inside its constructor.

```
1  package Student;
2  our @ISA = qw(Person);
3
4  sub new {
5      my ($class, @args) = @_;
6      $self = $class->SUPER::new(@args);
7      bless $self, $class;
8      return $self;
9  }
10
11 1;
12
13 $pe = new Person();
14 $st = new Student();# final class
```

There is no easy way to define a final method.

3.6 PHP

A final class can be defined using the keyword ***final***.

```
1  final class Person {}
2
3  //class Student extends Person {}//error: final class
```

Likewise, the same keyword is used for final methods

```
1  class Person {
2      final public function finalMethod() {
3          echo "Person finalMethod\n";
4      }
5  }
6
7  class Student extends Person {
8      //final public function finalMethod() {}//error: final function
9  }
```

3.7 Python

Finals are not Pythonic; there is no standard way to do it. But, you can define a meta-class which prevents extending a specific class (Byers, 2010).

```
class Final(type):
    def __new__(cls, name, bases, classdict):
        for b in bases:
            if isinstance(b, Final):
                raise TypeError("'{0}' is final class".format(b.__name__))
        return type.__new__(cls, name, bases, dict(classdict))
```

When applying this meta to a class, the latter can no longer be extended

```
# class Person(metaclass=Final): pass #Python 3
class Person(object):
    __metaclass__ = Final

#class Student(Person): pass # error: final
```

As for final methods, there a hack using meta-class but too much (Alinkaya, 2008). The code is afforded with this book.

3.8 Ruby

In Ruby, there is no final classes and methods. However, there is a hack to prevent a method from being overridden by a subclass (but not recommended) (MVP of stackoverflow, 2016).

4 Multiple inheritance

4.1 C++

C++ supports multiple inheritance: the constructors of inherited classes are called in the same order in which they are inherited (GeeksforGeeks, 2012).

```
class Person {
public:
    Person(){
        std::cout << "I am a person" << std::endl;
    }
};

class Machine {
public:
    Machine(){
        std::cout << "I am a machine" << std::endl;
    }
};

class Android: public Machine, public Person {
public:
    Android(){
        std::cout << "So, I am an android" << std::endl;
    }
};
```

If you have two classes (**Person** and **Machine**) with the same method signature (**void info()**) and a third one (**Android**) extending both of them, calling this method from an object of the third class (*Android*) will generate a compile error indicating ambiguity. A solution is to define a method with the same signature in the third class.

```
class Person {
public:
    void info() {
    }
};

class Machine {
public:
    void info() {
    }
};

class Android: public Machine, public Person {
};
int main()
{
    Android a = Android(); //or Android a;
    //a.info(); //request for member 'info' is ambiguous
    return 0;
}
```

Also, if your parent classes each has a field with the same name, the subclass cannot access it directly by its name since there is a conflict. Instead, it must indicate the field of which parent.

```
class Person {
protected:
    int serial;
};

class Machine {
protected:
    int serial;
};
class Cyborg: public Person, public Machine {
public:
    void info() {
        //reference to 'serial' is ambiguous
        //std::cout << "Cyborg Serial: " << serial << std::endl;
        std::cout << "Cyborg.Person Serial: " << Person::serial << std::endl;
        std::cout << "Cyborg.Machine Serial: " << Machine::serial << std::endl;
    }
};
```

One of multiple inheritance's problems is the diamond problem. Suppose we have a class called **Person** with a field **name** and the constructor to initialize it. This class has two subclasses: **Student** and **Professor** which will inherit its fields and methods. Then, a class **PhdStudent** which inherits from **Student** and **Professor**. When this class's constructor is called, it will call the constructors of its superclasses, which in their turn each will call the constructor of **Person**. This will create two copies of all the members of **Person** causing ambiguities. To fix this problem, the two classes **Student** and **Professor** have to be virtual using the keyword ***virtual***. In this case, when the constructor of **PhdStudent** calls its parents', it has to call its grandparent's as well.

```
class Person {
public:
    Person(std::string name){
        this->name = name;
    }
```

```
protected:
    std::string name;
};

class Student: virtual public Person {
public:
    Student(std::string name, double mark): Person(name){
    }
protected:
    double mark;
};

class Professor: virtual public Person {
public:
    Professor(std::string name, int hours): Person(name){
    }
protected:
    int hours;
};

class PhdStudent: public Student, public Professor {
public:
    PhdStudent(std::string name, int hours, double mark):
        Student(name, mark), Professor(name, hours), Person(name){
    }

    void info() {
        std::cout << "name: " << name << std::endl;
    }
};
```

4.2 Java

Prior to version 8, Java did not support multiple inheritance as we know it. But, it supports multiple inheritance of type using interfaces; a class implementing many interfaces is considered as having all these types at once. Version 8 introduces ***default*** methods to the interfaces, which means a class implementing an interface can inherit some behavior. It is called: multiple inheritance of implementation; as for multiple inheritance of state, it is not supported and thus you can extend one class (oracle, n.d.).

If a class inherits the same method signature more than twice, then it has to override this same function in order to avoid ambiguity. In case of method ambiguity between its parent types (class and/or interfaces), it can access the method of its superclass by simply using ***super***, and the defaults of its parent interfaces by using the name of the interface followed by ***.super.*** followed by the name of the method.

```
interface HumanBehaviour {
    default void info() {
        System.out.println("I am a person");
    }
}

interface MachineBehaviour {
    default void info() {
        System.out.println("I am a machine");
    }
}

class Person implements HumanBehaviour {
    public Person(){
        info();
    }
}
```

```
18
19 class Machine implements MachineBehaviour {
20     public Machine(){
21         info();
22     }
23 }
24
25 class Android extends Machine implements HumanBehaviour {
26     public void info() {
27         super.info();
28         HumanBehaviour.super.info();
29         System.out.println("So, I am an android");
30     }
31 }
```

As for diamond problem: if we have an interface **HumanBehavior** with a default method **void info()** and two interfaces (**StudentBehaviour** and **ProfessorBehavior**) extending it, then a class **PhdStudent** implementing these two, we will have these cases (Naftalin, 2012):

- The two interfaces do not override their parent's method: in this case, the class will inherit its grandparent's implementation.
- One of the two interfaces overrides its parent's method: in this case, the class will inherit the more specific implementation which is that of its parent.
- The two interfaces override their parent's method: in this case, the class must override it as well.

```
1  interface HumanBehavior {
2      default void info(){
3          System.out.println("Person");
4      }
5  }
6
7  interface ProfessorBehavior extends HumanBehavior {}
8
9  interface StudentBehavior extends HumanBehavior {
10     default void info(){
11         System.out.println("Student");
12     }
13 }
14
15 class PhdStudent implements StudentBehavior, ProfessorBehavior {}
```

4.3 Javascript

In EcmaScript5, a constructor of a class can call any other constructor by passing itself (the context) to that constructor. If there are any fields initialization, they will be assigned to the context. If you call more than one constructor and they assign the same field, there will be one unique field with the last assigned value. To inherit methods, a class assigns to its prototype a new instance of its parent's using ***Object.create***. To inherit more than one prototype, you can use the method ***Object.assign*** (MDN web docs, 2018b). If there is a conflict between parents methods, the subclass will inherit the last assigned one since its definition will overwrite the first ones.

```
1  function Person() {
2      console.log("I am a person");
3  }
4
5  Person.prototype.info = function() {
6      console.log("Person");
7  }
8
```

```
function Machine() {
    console.log("I am a machine");
}

Machine.prototype.info = function() {
    console.log("Machine");
}

function Android() {
    Person.call(this);
    Machine.call(this);
}

Android.prototype = Object.assign(
                        Object.create(Person.prototype),
                        Object.create(Machine.prototype)
                    );
Android.prototype.constructor = Android;
```

As for diamond problem, a certain context has only one copy of a field, which will take the last value assigned to it. Suppose we have a class **Person** which creates a field **name** and two other classes (**Student** and **Professor**) which inherits from it. If a class **PhdStudent** extends these two classes, the field **name** is created once and updated each time it is assigned. In our example, it will take the value assigned to **Student**'s constructor.

```
function PhdStudent(name, hours, mark) {
    Professor.call(this, name, hours);
    Student.call(this, name + "_student", mark);
    console.log("PhdStudent called");
}

PhdStudent.prototype = Object.assign(
                        Object.create(Professor.prototype),
                        Object.create(Student.prototype)
                    );
PhdStudent.prototype.constructor = PhdStudent;
PhdStudent.prototype.info = function() {
    console.log("my name is: ", this.name);
}
```

In EcmaScript 2015, ***extends*** does not accept more than one superclass. So, you have to work around this as in (Dorin, 2016).

4.4 Lua

One way to implement multiple inheritance is to use the meta-method ***__index***, which is called whenever Lua cannot find a key in the current table (Ierusalimschy, 2003). But, an object has to own its fields; this is why we should create an object with all its parents fields. To create a new object in the constructor, we have to create new objects from parents then merge them into a single one. If a key is available for many classes, it will get the last value assigned to it.

Suppose we have two classes (**Person** and **Machine**), each has its own fields and methods. You can create a function which returns an index function based on a list of given parents. In our example, the index will favor the first parents if a key is not available in the current table (object). Also, when it finds a key in a parent class, it will link it to the current object so next time it will not loop again. Do not forget to pass the current class first; otherwise, the object will start searching in its parents and not its class first.

```
-- ...
```

```lua
-- Index function
function get_index(parents)
    return function(self, key)
        for i=1, #parents do
            local member = parents[i][key]
            if member ~= nil then
                rawset(self, key, member) -- next time it will be its own key
                return member
            end
        end
    end
end

-- Android class
local Android = {}
Android.__index = get_index({Android, Machine, Person})

function Android.new()     -- The constructor
    local instance2 = Machine.new() -- first parent
    local instance = Person.new() -- second parent
    for k,v in pairs(instance2) do -- Merge the two objects
        instance[k] = v
    end
    setmetatable(instance, Android)
    print("I am an android");
    return instance
end
```

Using this code, you will have unique fields even if there is a diamond inheritance. As for methods inheritance, the subclass will inherit the first appropriate method encountered in its parents.

4.5 Perl

Perl supports multiple inheritance via its keyword ***@ISA*** by searching for all methods of the first class and its ancestors, then it passes to the next (Schwartz and Phoenix, 2003). So, ambiguity between methods is solved. As for the fields, you can create an object (hash) from each parent and merge them into one object.

```perl
package Person;

sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    print "I am a person\n";
    return $self;
}

1;

package Machine;

sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    print "I am a machine\n";
    return $self;
}

1;

```

```perl
package Android;
our @ISA = qw(Machine Person);

sub new {
    my $class = shift;
    my $self = $class->Person::new();
    my $self2 = $class->Machine::new();
    $self = { %$self, %$self2};
    bless $self, $class;
    print "So, I am an android\n";
    return $self;
}

1;
```

Using this code, you will have unique fields even if there is a diamond inheritance. As for methods inheritance, the subclass will inherit the first appropriate method encountered in its parents.

4.6 PHP

In PHP, there is no multiple inheritance, but using interfaces we can achieve multiple inheritance of type. Innterfaces can define behavior without implementing it; it can even declare the constructor. Prior to PHP 5.3.9, a class could not implement two interfaces that specified a method with the same name, since it would cause ambiguity. More recent versions of PHP allow this as long as the duplicate methods have the same signature (Cowburn, 2018).

Another mechanism is traits: they are used as mean to reuse code, but they do not support polymorphism. If two traits are used and they have conflicting methods, an error is produced. To solve the problem, you can choose which one to use using the operator ***insteadof***; or you can add an alias to one of them using the operator ***as*** (Cowburn, 2018).

We can use both of them to achieve something close to multiple inheritance: interfaces for type and traits for code reuse. In our example, we define two interfaces **iPerson** and **iMachine** to represent the types. Then, two traits **tPerson** and **tMachine** implementing the behavior **info()**. A trait **iConstruct** which implements a constructor calling **info()** of the parent class if it exists, then calling **info()** of the current class. Two concrete classes **Person** and **Machine** implementing the last interfaces respectively using the defined traits. An **Android** is a **Machine** which has behavior similar to a **Person**. A **Cyborg** is a **Person** which has behavior similar to a **Machine**.

```php
interface iPerson { public function info(); }

interface iMachine { public function info(); }

trait tPerson {
    public function info() { echo "I am a person\n"; }
}

trait tMachine {
    public function info() { echo "I am a machine\n";}
}

trait tConstruct {
    public function __construct() {
        if (get_parent_class()) { parent::info(); }
        $this->info();
        $this->moreInfo();
    }
```

```
    public function moreInfo(){}
}

class Person implements iPerson { use tPerson, tConstruct;}

class Machine implements iMachine { use tMachine, tConstruct; }

class Android extends Machine implements iPerson {
    use tPerson, tConstruct;
    public function moreInfo(){ echo "So, I am an android\n";}
}

class Cyborg extends Person implements iMachine {
    use tMachine, tConstruct;
    public function moreInfo(){ echo "So, I am a cyborg\n";}
}
```

As for diamond problem, PHP supports the extension of one class. You can create an interface **iPerson** to represent persons type, and two others extending it: **iStudent** and **iProfessor**. Then, a class **Person** implementing the interface **iPerson** which will serve as a base to other classes.

```
interface iPerson {
    public function info();
}
interface iStudent extends iPerson {}
interface iProfessor extends iPerson {}

class Person implements iPerson {
    protected $name;
    public function __construct($name) {
        $this->name = $name;
    }
    public function info() { echo "my name: $this->name\n"; }
}
```

Then, we can create two traits (**tStudent** and **tProfessor**) to contain members unique to classes **Student** and **Professor** respectively. These two classes extend the class **Person** and add their unique characteristics using their respective traits.

```
trait tStudent {
    protected $mark;
    public function __construct($name, $mark) {
        parent::__construct($name);
        $this->mark = $mark;
    }
    public function info() { echo "my mark: $this->mark\n"; }
}

trait tProfessor {
    protected $hours;
    public function __construct($name, $hours) {
        parent::__construct($name);
        $this->hours = $hours;
    }
    public function info() { echo "my work hours: $this->hours\n"; }
}

class Student extends Person implements iStudent {
    use tStudent {tStudent::info as private sinfo;}
    public function info() {
        parent::info();
        $this->sinfo();
    }
```

```
}

class Professor extends Person implements iProfessor {
    use tProfessor {tProfessor::info as private pinfo;}
    public function info() {
        parent::info();
        $this->pinfo();
    }
}
```

Finally, **PhdStudent** extends **Person** and adds its characteristics using the two previously defined traits.

```
class PhdStudent extends Person implements iStudent, iProfessor {
    use tStudent {
        tStudent::__construct as private __sconstruct;
        tStudent::info as private sinfo;
    }
    use tProfessor {
        tProfessor::__construct as private __pconstruct;
        tProfessor::info as private pinfo;
    }
    public function __construct($name, $hours, $mark) {
        $this->__pconstruct($name, $hours);
        $this->__sconstruct("$name modified", $mark);
    }
    public function info() {
        parent::info();
        $this->sinfo();
        $this->pinfo();
    }
}
```

P.S. I used interfaces just to maintain inheritance of type in case someone uses ***instanceof***, otherwise you can drop them.

4.7 Python

Python supports a limited form of multiple inheritance. For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in the subclass, it is searched in the left superclass, then (recursively) in its ancestors, and so on. For new-style classes, the method resolution order changes dynamically to support cooperative calls to ***super*** (Python Software Foundation, 2018).

```
class Person(object):
    def __init__(self):
        print("I am a person")

class Machine(object):
    def __init__(self):
        print("I am a machine")

class Android(Machine, Person):
    def __init__(self):
        Machine.__init__(self)
        Person.__init__(self)
        print("So, I am an android")
```

Since fields are initialized by pushing them into the context object (***self*** by convention), the attribute will not be defined twice: one in the subclass and one in the superclass of a given object. In this case, when we have multiple inheritance, the last constructor to be called among superclasses' is the one which will

set the value of a shared field.

Lets try to create a beautiful scary diamond by defining a class **Person**, two classes (**Student** and **Professor**) extending it, and a fourth class **PhdStudent**. The name attribute of an instance of **PhdStudent** will have the last value assigned to it via **Student**'s constructor. Also, **PhdStudent** will inherit the method **info()** of **Student**, not because it is the most specific but because it is the left one (when defining inheritance).

```
class Person(object):
    def __init__(self, name):
        self.name = name
        print("Person")

    def info(self):
        print("my name: " + self.name)

class Student(Person):
    def __init__(self, name, mark):
        Person.__init__(self, name)
        self.mark = mark
        print("Student")

    def info(self):
        Person.info(self)
        print("my mark: " + str(self.mark))

class Professor(Person):
    def __init__(self, name, hours):
        Person.__init__(self, name)
        self.hours = hours
        print("Professor")

class PhdStudent(Student, Professor):
    def __init__(self, name, hours, mark):
        Professor.__init__(self, name, mark)
        Student.__init__(self, name + "-1", mark)
        print("PhdStudent")
```

4.8 Ruby

Ruby does not support multiple inheritance; a class can inherit from one class. There is an alternative called ***mixins*** which are some modules included inside a class's definition to enhance code reuse (Thomas and Hunt, 2001).

Lets define behavior using mixins: **PersonMixin** to define methods applied to persons, and **MachineMixin** to define mmethods apllied to machines. Another mixin which defines the initializer: its function is to search for **info()** in the superclass, if found it will be called; then call the current info; finally, call **moreInfo** which is useful in case the specific class wants to add more information. **Person** and **Machine** are classes with **PersonMixin** and **MachineMixin** behaviors respectively; they have both an initializer afforded via **InitMixin**. Then **Android** and **Cyborg** which are **Machine** with **PersonMixin** behavior and **Person** with **MachineMixin** behavior respectively.

```
module PersonMixin
    def info; puts "I am a person"; end
end

module MachineMixin
    def info; puts "I am a machine"; end
end
```

```ruby

module InitMixin
    def initialize()
        info2 = method(:info).super_method
        if info2 != nil then info2.call end
        info
        moreInfo
    end
    def moreInfo; end
end

class Person include PersonMixin, InitMixin end

class Machine include MachineMixin, InitMixin end

class Android < Machine
    include PersonMixin, InitMixin
    def moreInfo; puts "So, I am an android"; end
end

class Cyborg < Person
    include MachineMixin, InitMixin
    def moreInfo; puts "So, I am an cyborg"; end
end
```

Discussion

Inheritance is a mean to reuse code from parent classes. Constructors can be considered as behaviors too, so they must be inherited too. Not every language allows constructor inheritance, thus we can categorize languages into three groupes:

- **Out of box**: These are the languages designed to support constructor inheritance. Most of our languages are among this category: C++11, Perl, PHP, Python and Ruby.
- **Can be done**: These are the languages which have many ways of doing inheritance. Lua belongs to this category.
- **Cannot be done**: These are the languages which forces constructor override. Java and Javascript are examples of this category along with C++ prior to version 11.

When inheriting members from a superclass, a class can change their visibility mode in some OOP languages. Based on this ability, we can divide OOP languages into three categories:

- **Allowed**: A language allows visibility mode change either for methods or fields. C++ can restrict original visibility of all members upon inheritance without selecting each one apart. PHP can exceed original visibility of methods upon overriding, and fields by defining them again (real overriding).
- **Just methods**: A language allows visibility mode change just for methods. Java and Ruby do so upon overriding methods. Java can only exceed original visibility; as for fields it does not change its visibility, but it creates a new field with the same name as its parent's and preserve the latter.
- **Not allowed**: These are languages which do not allow visibility change either for methods or fields. In our case, they are the languages with just public members: Javascript, Lua, Perl and Python.

When you try to override a field in C++ and Java, you actually defining a new field while the parent's still exists. Other languages just override the value of the field if it exists, or create it otherwise.

An abstract class is a class which cannot be instantiated and has to be extended. It can be abstract because

it cannot exist in the current context or because it has some unfinished methods implementations. Based on this criterion, a programming language can be one of these three categories:

- **Class-level abstraction**: In this category, we can define a class as abstract even if all its methods have implementations. Java and PHP belong to this category.
- **Method-level abstraction**: In this category, a class is considered as abstract if it has at least one abstract method. This category contains: C++ and Python (via ***abc*** module).
- **No abstraction**: All the languages which do not afford a built-in mechanism to support abstract classes. Javascript, Lua, Perl and Ruby are examples of this category.

Final classes are those which cannot be extended, and final methods are those which cannot be overridden. Languages such as Java and PHP affords this capacity; C++11 does too.

Multiple inheritance is the capacity to extend many classes at once. We can define three types of multiple inheritance; a given language can afford all of these types.

- **Type multiple inheritance**: A class can inherit many types at once. When an instance of this class is tested against these types, it belongs to all of them. Languages affording this capacity are: C++ (class inheritance), Java (interfaces), Perl (***@ISA***), PHP (interfaces) and Python (class inheritance).
- **Behavior multiple inheritance**: A class can inherit behavior from many sources. In this case, we can consider code reuse via composition mechanisms such as traits and mixins as a form of inheritance. Languages affording this capacity are: C++ (class inheritance), Java (defaults via interfaces), javascript (merging prototypes), Lua (using ***__index***), Perl (***@ISA***), PHP (traits), Python (class inheritance) and Ruby (mixins). It is important to point out that Java and Ruby's behavior multiple inheritance is limited. Java's default methods are limited when it comes to operate on class's state: they can be totally independent from class's fields, or they can be dependent on final fields which have to be initialized in the interface. Ruby's mixins can have calls to superclass's methods; but when including consecutive mixins, their super will refer to the previous included mixin.
- **State multiple inheritance**: A class can inherit fields from multiple classes or sources. These languages afford this capacity: C++ and PPython(class inheritance); Javascript, Lua and Perl (by combining parents objects); and PHP (traits).

Table 5.1 represents a comparison between some OOP languages based on their inheritance capacities:

- **Constructor inheritance**: Can a subclass inherit its parent's constructor? (no need to define it again).
- **Visibility change**: Can a subclass change the visibility modes of its parent's members?
- **Field hiding**: If a subclass defines the same field, will be two fields of the same name: one of the subclass and one of its parent?
- **Abstract**: Can the language afford abstract classes and methods?
- **Final**: Can the language afford final classes and methods?
- **Multiple inheritance**: How far a language can afford multiple inheritance (including code reuse from sources other than classes)?

Table 5.1: Inheritance comparison

Language	Constructor inheritance	Visibility change	Field hide	Abstract	Final	Multiple inheritance
C++	C++11: via keyword ***using***	Yes	Yes	method: virtual myMethod = 0; class: at least one abstract method	method: (C++11) virtual myMethod final class: (C++11) class MyClass final {}	All
Java	No	Methods: after overriding them	Yes	method: abstract myMethod class: abstract class MyClass	method: final myMethod class: final class MyClass	Type (interfaces), Behavior(defaults in interfaces)
Javascript	No way (since the class is its own constructor)	No	No	No	No	State (combine parents new objects) and Behavior: (combine parents prototypes)
Lua	Can be done: using colon notation for new	No	No	No	No	State (combine parents new objects) and Behavior: (using ***__index***)
Perl	Yes	No	No	No	No	Type and Behavior (***@ISA***), State (combine parents new objects)
PHP	Yes	Methods: after overriding them; Fields	No	method: abstract myMethod class: abstract class MyClass	method: final myMethod class: final MyClass	Type (interfaces), State and Behavior (traits)
Python	Yes	No	No	using abc module	No	All
Ruby	Yes	Methods: after overriding	No	No	No	Behavior (via mixins)

CHAPTER 6

POLYMORPHISM

Polymorphism *is another key concept of OOP. In general, the term refers to the ability to assign a different meaning or usage to something in many contexts. Hence, it is not a concept specific to OOP since you can find other forms such as parametric polymorphism (generic programming). In languages with type inheritance, an object of a class can be considered as an object of its superclass; and therefore, the same applications can be executed on it. Some languages afford functionalities to verify if a certain object is instantiated from a class, to get its class or to cast the variable using it to another subclass. Methods can express polymorphism in two ways: overloading (same name with multiple signatures) and overriding (same signature, different implementation). Not every language affords overloading, but methods overriding is a core task of OOP.*

1 Subtype polymorphism

According to the ***Liskov substitution principle*** (Liskov, 1987), a function written to take an object of a certain type **T** must work correctly if passed an object of a type **S** which is a subtype of **T**. C++ and Java are examples of OOP languages with static type checking affording this principle. Some languages do not need an object to be created from a class/prototype extending another to execute certain treatments on it. They use a mechanism called ***Duck typing*** which states that "*If it walks like a duck and talks like a duck, it must be a duck*".

A simple example will be sufficient to show how every language handles such polymorphism. Having a class **Person** with a method **talk** and two subclasses **Student** and **Professor**:

- A function **announce** designed to call **talk** of **Person** is passed objects of **Student** and **Professor** as argument.
- An array intended to contain objects of **Person** is filled with objects of its subtypes.
- If the language supports Duck typing: some unrelated classes and objects having the method **talk** can be used with the two previous propositions.

1.1 C++

Lets create our classes

```
1 class Person {
2 public:
3     void talk(){
4         std::cout << "I am a person" << std::endl;
5     }
6 };
7
8 class Student: public Person {};
```

```
9
10 class Professor: public Person {};
```

Passing objects of these three classes to a function designed for objects of type **Person** will work just fine.

```
1  void announce(std::string msg, Person p){
2      std::cout << msg << ": ";
3      p.talk();
4  }
5
6  int main()
7  {
8      Person pe; Student st; Professor pr;
9
10     announce("Person", pe);
11     announce("Student", st);
12     announce("Professor", pr);
13
14     return 0;
15 }
```

Objects of type **Person** or any other subclass can be assigned to a variable of type **Person**.

```
1  int main()
2  {
3      Person pe; Student st; Professor pr;
4      std::cout << "Table of Person" << std::endl;
5      Person people[] = {pe, st, pr};
6      for (int i =0; i<3; ++i) { people[i].talk(); }
7
8      return 0;
9  }
```

1.2 Java

Lets create our classes

```
1  class Person {
2      public void talk(){
3          System.out.println("I am a person");
4      }
5  }
6  class Student extends Person {}
7
8  class Professor extends Person {}
```

Passing objects of these three classes to a function designed for objects of type **Person** will work just fine.

```
1  public class Subtype {
2      public static void announce(String msg, Person p) {
3          System.out.print(msg + ": ");
4          p.talk();
5      }
6      public static void main(String[] args) {
7          Person pe = new Person();
8          Student st = new Student();
9          Professor pr = new Professor();
10
11         announce("Person", pe);
12         announce("Student", st);
13         announce("Professor", pe);
```

```
    }
}
```

If we use an interface (**Machine**) and a function **announce2** similar to **announce** but taking as parameter **Machine** instead of **Person**, this will work as well because interfaces in Java define new types and support type polymorphism.

```
public class Subtype {
    //...
    public static void main(String[] args) {

        Robot ro = new Robot();
        Cyborg cy = new Cyborg();

        announce2("Robot", ro);
        announce2("Cyborg", cy);
        announce("Cyborg", cy);
    }
}

interface Machine {
    public void talk();
}

class Robot implements Machine {
    public void talk(){
        System.out.println("I am a robot");
    }
}

class Cyborg extends Person implements Machine {}
```

Objects of type **Person** or any other subclass can be assigned to a variable of type **Person**.

```
public class Subtype {
    public static void main(String[] args) {
        Person pe = new Person();
        Student st = new Student();
        Professor pr = new Professor();
        System.out.println("Table of Person");
        Person[] people = {pe, st, pr};
        for (Person p: people) { p.talk(); }
    }
}
```

1.3 Javascript

Lets create our classes (ES6 style, ES5 code is afforded as well): all of them afford a method **talk**.

```
class Person {
    talk() {
        console.log("I am a person");
    }
}

class Student extends Person {}

class Professor extends Person {}

class Robot {
    talk() {
        console.log("I am a robot");
```

```
    }
}
```

A javascript function does not specify the types of its parameters. In this function, it is clear that the object named **talker** must define a method called **talk**. Because we do not know what type the object is, and if it has a function **talk**, we can verify before calling it.

```
function announce(msg, talker) {
    process.stdout.write(msg + ": ");
    if (typeof talker.talk == "function") talker.talk();
    else console.log("Sorry! I do not talk!");
}
```

Then, objects created from class **Person** and its subclasses can be passed to **announce** since they all afford the method **talk**. Also, objects created from a different class (**Robot**) or created directly containing that method can be passed as well, without causing any problem. If an object has no method **talk** (lets say: a number), the function **announce** will print the other message instead of calling our method.

```
let pe = new Person(),
    st = new Student(),
    pr = new Professor();

let ro = new Robot();
let cat = {
    talk: function(){
        console.log("Meow!");
    }
};

announce("Person", pe);
announce("Student", st);
announce("Professor", pr);
announce("Robot", ro);
announce("a cat", cat);
announce("a number", 25);
```

Lets try an array with different elements

```
console.log("A table with different types");
let elements = [pe, st, pr, ro, cat, 25];
for (let i = 0; i < elements.length; i++){
    try {
        elements[i].talk();
    }
    catch(error){
        console.log("The element n° ", i, " does not talk()");
    }
}
```

1.4 Lua

Lets create our classes: all of them afford a method **talk**. In this code, only the superclass affords the function **new** by assigning ***self*** (the class name) as meta-table of the newly created object(table). The benefit of doing this is to not repeat the same **new** method for subclasses.

```
-- Person class
local Person = {}
Person.__index = Person

```

```lua
function Person:new()     -- The constructor
    local instance = {}
    setmetatable(instance, self) --self, here, will refer to the calling class
    return instance
end

function Person:talk()
    print("I am a person")
end

-- Student class
local Student = {}
Student.__index = Student
setmetatable(Student, Person) -- extends

-- Professor class
local Professor = {}
Professor.__index = Professor
setmetatable(Professor, Person) -- extends
```

A lua function does not specify the types of its parameters. In **announce**, it is clear that the object named **talker** must define a method called **talk**. Because we do not know what type the object is, and if it has a function **talk**, we can verify if it is a table and if it has a member of type **function** called **talk**.

```lua
function announce(msg, talker)
    io.write(msg .. ": ")
    if type(talker) == "table" and type(talker.talk) == "function" then
        talker:talk()
    else
        print("Sorry! I do not talk!")
    end
end
```

Then, objects created from class **Person** and its subclasses can be passed to **announce** since they all afford the method **talk**. Also, objects created directly using a table affording the same method can be passed without problem.

```lua
local pe = Person:new()
local st = Student:new()
local pr = Professor:new()

local cat = {
    talk = function(); print("Meow!"); end
}

announce("Person", pe)
announce("Student", st)
announce("Professor", pr)
announce("a cat", cat)
announce("a number", 25)
```

Lets try an array with different elements

```lua
print("A table with different types")
local elements = {pe, st, pr, cat, 25}
for i =1, #elements do
    success, res = pcall(function() elements[i].talk(); end)
    if not success then
        print("The element n°" .. i .. " does not talk()")
    end
end
```

1.5 Perl

Lets create our classes: all of them afford a method **talk** even the one not inheriting from **Person**. The **new** method blesses a hash (our object to be) to the first argument which is the class calling it. When other classes inherit **new**, each will pass its name when calling this method; and therefore, the new object will be blessed to it.

```
1  package Person;
2  sub new { return bless {}, shift; }
3  sub talk { print "I am a person\n"; }
4  1;
5
6  package Student;
7  our @ISA = qw(Person);
8  1;
9
10 package Professor;
11 our @ISA = qw(Person);
12 1;
13
14 package Robot;
15 sub new { return bless {}, Robot; }
16 sub talk { print "I am a robot\n";}
17 1;
```

A perl subroutine does not specify the types of its parameters and either their number. In **announce**, it is clear that the object named **talker** must define a method called **talk**. Because we do not know what type the object is, we can verify if it has a method called **talk** using the UNIVERSAL method ***can***.

```
1  sub announce {
2      my ($msg, $talker) = @_;
3      print "${msg}: ";
4      if ( $talker->can("talk")) {$talker->talk();}
5      else {print "Sorry! I do not talk!\n";}
6  }
```

Any object created from a class affording **talk**, or one of its subclasses can be passed to the function **announce**.

```
1  $pe = Person->new();
2  $st = Student->new();
3  $pr = Professor->new();
4  $ro = Robot->new();
5
6  announce("Person", $pe);
7  announce("Student", $st);
8  announce("Professor", $pr);
9  announce("Robot", $ro);
10 announce("a number", 25);
```

Lets try an array with different elements

```
1  print "A table with different types\n";
2  my @elements = ($pe, $st, $pr, $ro, 25);
3  foreach my $i (0 .. scalar @elements - 1) {
4      eval { $elements[$i]->talk(); };
5      if ($@) { print "The element n°$i does not talk()\n"; }
6  }
```

1.6 PHP

Lets create our classes: all of them afford a method **talk** either directly or by inheritance.

```php
class Person {
    public function talk(){
        echo "I am a person\n";
    }
}

class Student extends Person {}

class Professor extends Person {}

class Robot {
    public function talk(){
        echo "I am a robot\n";
    }
}
```

A PHP function does not specify the types of its parameters. In **announce**, it is clear that the object named **talker** must define a method called **talk**. So, we have to verify that using a function called ***method_exists***.

```php
function announce($msg, $talker){
    echo "$msg: ";
    if (method_exists($talker, "talk")) $talker->talk();
    else echo "Sorry! I do not talk!\n";
}
```

Then, objects created from any class affording the method **talk** can be passed to **announce** without a problem. If an object has no method **talk** (lets say: a number), the function **announce** will print the other message instead of calling our method.

```php
$pe = new Person();
$st = new Student();
$pr = new Professor();
$ro = new Robot();

announce("Person", $pe);
announce("Student", $st);
announce("Professor", $pr);
announce("Robot", $ro);
announce("a number", 25);
```

Lets try an array with different elements

```php
echo "A table with different types\n";
$elements = array($pe, $st, $pr, $ro, 25);
foreach($elements as $i=>$element) {
    try {
        $element->talk();
    }
    catch (Error $e) {
        echo "The element n°$i does not talk()\n";
    }
}
```

1.7 Python

Lets create our classes: all of them afford a method **talk** either directly or by inheritance.

```
class Person(object):
    def talk(self):
        print("I am a person")

class Student(Person): pass

class Professor(Person): pass

class Robot(object):
    def talk(self):
        print("I am a robot")
```

A Python function does not specify the types of its parameters. In **announce**, it is clear that the object named **talker** must define a method called **talk**. So, we have to retrieve that method using a function called *getattr*; if there is no member called **talk**, the function will return *None* as specified in the code. To verify if it is a method and not a field, we use the function *callable*.

```
import sys
def announce(msg, talker):
    sys.stdout.write(msg + ": ")
    talk = getattr(talker, "talk", None)
    if talk != None and callable(talk):
        talk()
    else:
        print("Sorry! I do not talk!")
```

Then, objects created from any class affording the method **talk** can be passed to **announce** without a problem. If an object has no method **talk** (lets say: a number), the function **announce** will print the other message instead of calling our method.

```
pe = Person()
st = Student()
pr = Professor()
ro = Robot()

announce("Person", pe)
announce("Student", st)
announce("Professor", pr)
announce("Robot", ro)
announce("a number", 25)
```

Lets try an array with different elements

```
print ("A table with different types")
elements = [pe, st, pr, ro, 25]
for i, element in enumerate(elements):
    try:
        element.talk()
    except Exception:
        print("The element n°" + str(i) + " does not talk()")
```

1.8 Ruby

Lets create our classes: all of them afford a method **talk** either directly or by inheritance.

```
class Person
    def talk
        puts "I am a person"
    end
end
```

```ruby

class Student < Person; end

class Professor < Person; end

class Robot
    def talk
        puts "I am a robot"
    end
end
```

A Ruby function does not specify the types of its parameters. In **announce**, it is clear that the object named **talker** must define a method called **talk**. So, we have to verify if the object affords this method using a function ***method_defined?*** which is applied to classes (we have to retrieve the object's class first).

```ruby
def announce (msg, talker)
    print "#{msg}: "
    if talker.class.method_defined? :talk then
        talker.talk
    else
        puts "Sorry! I do not talk!"
    end
end
```

Then, objects created from any class affording the method **talk** can be passed to **announce** without a problem. If an object has no method **talk** (lets say: a number), the function **announce** will print the other message instead of calling our method.

```ruby
pe = Person.new
st = Student.new
pr = Professor.new
ro = Robot.new

announce("Person", pe)
announce("Student", st)
announce("Professor", pr)
announce("Robot", ro)
announce("a number", 25)
```

Lets try an array with different elements

```ruby
puts  "A table with different types"
elements = [pe, st, pr, ro, 25]
elements.each_with_index do |element, i|
    begin
        element.talk
    rescue Exception => e
        puts "The element n°#{i} does not talk()"
    end
end
```

2 Type manipulation

There exists some methods afforded by OOP languages to manipulate types. Even languages with dynamic type checking have some of these methods. For each programming language, we will see these points:

- **Get object type**: How to get the type of a given object.

- **Is instance of**: Verify if an object is generated from a class/prototype.
- **Members existence**: Check if a member (field or method) exists in a given object.
- **Type casts**: Downcasting is the act of changing a reference of a base class to one of its derived class. It is useful in case of statically typed languages like C++ and Java.

2.1 C++

Lets create our classes. A class is said to be ***polymorphic*** if it has at least one virtual method (defined in it or inherited). In our example, the class **Person** has one virtual method, while the class **Machine** has none.

```
1  class Person {
2  public:
3      virtual void talk(){ std::cout << "I am a person" << std::endl; }
4  };
5
6  class Student: public Person {
7  public:
8      void learn(){ std::cout << "I am learning" << std::endl; }
9  };
10
11 class Professor: public Person {
12 public:
13     int nbr;
14     Professor(){ nbr = 5; }
15     void teach(){ std::cout << "I am teaching" << std::endl; }
16 };
17
18 class Machine {};
19
20 class Mooc: public Machine, public Professor {};
```

C++ affords a function ***typeid*** from the standard library ***typeinfo***. You can either print the type of a pointer or, dynamically, the type of the object pointed to. A type must be polymorphic so this function can operate dynamically on it.

```
1  int main()
2  {
3      Person pe; Student st; Professor pr; Mooc mooc;
4      Person *pst = &st;
5      Person *ppr = &pr;
6      Person *pmooc = &mooc;
7      Machine *mmooc = &mooc;
8      std::cout << "typeid(pst): " << typeid(pst).name() << std::endl;
9      std::cout << "typeid(*pst): " << typeid(*pst).name() << std::endl;
10     std::cout << "typeid(*pmooc): " << typeid(*pmooc).name() << std::endl;
11     std::cout << "typeid(*mmooc): " << typeid(*mmooc).name() << std::endl;
```

To verify if an object pointed to by a pointer is of a certain type dynamically, you can try to cast it to that type and verify if it succeeded. A function proposed by Panzenböck (2014) which works polymorphic pointers.

```
1  template<typename Class, typename T>
2  bool instanceof(const T *ptr) {
3      return dynamic_cast<const Class*>(ptr) != 0;//nullptr with C++11
4  }
```

Then, a pointer is tested as follows

```
1      std::cout << "instanceof<Person>(pst): " <<  instanceof<Person>(pst) << std::endl;
```

In C++, you can check if a class has a member using a mechanism called: Substitution Failure Is Not An Error (SFINAE). For example, this is a template to verify if classes has a method **learn**.

```
template<typename T>
struct HasLearnMethod
{
    typedef char Yes[1]; typedef char No[2];
    template <typename C> static Yes& test(typeof(&C::learn) ) ;
    template <typename C> static No& test(...);
    static const bool value = (sizeof(test<T>(0)) == sizeof(Yes)) ;
};
```

It can be used as follows

```
    std::cout << "Student.learn: " << HasLearnMethod<Student>::value << std::endl;
```

There are two types of down-casting (among others): ***static_cast*** and ***dynamic_cast***. Using the first one, type check is performed during compilation. The dynamic cast checks types at runtime and return ***NULL*** in case of pointers, or throw an exception in case of references.

```
    Student * st2 = static_cast<Student*> (pst);
    st2->learn();
    ((Student*) pst)->learn();
    Mooc * mooc2 = dynamic_cast<Mooc*>(pmooc);
    mooc2->teach();
```

2.2 Java

Lets define some classes

```
class Person {
    public void talk(){ System.out.println("I am talking"); }
}

class Student extends Person {
    public void learn(){ System.out.println("I am learning"); }
}

interface Worker {
    default public void work(){ System.out.println("I am working"); }
}

class Professor extends Person implements Worker {
    public int nbr = 5;
    public void teach(){ System.out.println("I am teaching"); }
}
```

To find the class of an object, you can use the method ***getClass*** inherited from the universal class ***Object***. A class ***Class*** affords a method ***getName*** which returns its name as a ***String***.

```
public class TypeManip {
    public static void main(String[] args) {
        Person pe = new Person();
        Person st = new Student();
        Person pr = new Professor();
        System.out.println("Type of st: " + st.getClass().getName());
    }
}
```

To verify if an object is an instance of a class, the keyword ***instanceof*** is used.

```
        System.out.println("st instanceof Professor: " + (st instanceof Professor));
```

It is possible to verify if an object has a field or method in Java using reflection. You have to import **java.lang.reflect.*** which contains two classes ***Method*** and ***Field***. A class ***Class*** affords two methods ***getMethod*** which returns a method, and ***getField*** which returns a field. These two methods when they cannot find the member in question, they throw an exception. To execute a ***Method***, the method ***invoke*** is used, given the first parameter which is the object affording this method and a list of objects used as parameters. Likewise, to recover a value ***Field***, the method ***get*** which returns an ***Object***; or you can use other methods which return some other types if you know the type of your field. Reflection is not usually used since your methods, mostly, will have a defined typed parameters.

```
    public static void testMembers(Object obj) {
        try {
            Method learnM = obj.getClass().getMethod("learn");
            learnM.invoke(obj, new Object[]{});
        }
        catch (Throwable t) {
            System.out.println("Method learn does not exist!");
        }

        try {
            Field nbrF = obj.getClass().getField("nbr");
            System.out.println("Field nbr = " + nbrF.get(obj));
        }
        catch (Throwable t) {
            System.out.println("Field nbr does not exist!");
        }
    }
```

A variable with a type of some class referring an object with a type of its subclass cannot call the subclass's methods. If you try to do that, you will have a compiler error telling you that the object does not afford that method. So, you have to tell the compiler that the variable is, in fact, referencing an object of the subclass. This is called: type casting, and more precisely: downcasting.

```
        Student st2 = (Student) st;
        st2.learn();
        Professor pr2 = (Professor) pr;
        pr2.teach();
        pr2.work();
        ((Worker) pr).work();
```

2.3 Javascript

Lets create our classes (ES6 style, ES5 code is afforded as well).

```
class Person {
    talk() { console.log("I am talking"); }
}

class Student extends Person {
    learn() { console.log("I am learning"); }
}

class Professor extends Person {
    constructor(){ super(); this.nbr = 5; }
    teach() { console.log("I am teaching"); }
}

function PhdStudent() {
```

```
    Object.assign(this, new Professor(), new Student());
}
PhdStudent.prototype = Object.assign(
                        Object.create(Professor.prototype),
                        Object.create(Student.prototype)
                    );
PhdStudent.prototype.constructor = PhdStudent;
let pe = new Person(),
    st = new Student(),
    pr = new Professor(),
    phd = new PhdStudent();
```

Using ***typeof***, the type you will get is **object** for our new objects. If you want to know the prototype used to create an object, each object has a property called ***constructor*** which refers to the class/prototype used to create the object.

```
console.log("Typeof st: " + (typeof st) + ", class: " + (st.constructor.name));
```

To verify if an object is an instance of a class, the keyword ***instanceof*** is used. You can realize that a prototype can inherit just one prototype, and even using ***Object.assign*** the first prototype is considered as the superclass and the others are just mixins.

```
console.log("st instanceof Person: " + (st instanceof Person)); //true
```

To verify if an object has a method, simply check if the method is of type "**function**". To verify if an object has a field (property in Javascript lingua), you can use ***hasOwnProperty*** inherited from the universal class ***Object***. Some solutions on the web suggest that you verify if a field's type is not "**undefined**", but a field can exist for an object with "undefined" value.

```
function testMembers(msg, obj) {
    process.stdout.write(msg + ": ");
    if (typeof obj.learn == "function") obj.learn();
    else console.log("Method learn does not exist!");

    process.stdout.write(msg + ": ");
    if (obj.hasOwnProperty("nbr")) console.log("Field nbr = " + obj.nbr);
    else console.log("Field nbr does not exist!");
}
```

There is no cast on Javascript, all happens in execution time. If you called a non existing method of an object, you will have an error.

2.4 Lua

Lets create our classes: classes are simple tables in our case. We add a field called **__NAME** to each class to afford its name.

```
-- Person class
local Person = {__NAME = "Person"}
Person.__index = Person

function Person:new()     -- The constructor
    local instance = {}
    setmetatable(instance, self) --self, here, will refer to the calling class
    return instance
end

function Person:talk()
```

```lua
    print("I am talking")
end

-- Student class
local Student = {__NAME = "Student"}
Student.__index = Student
setmetatable(Student, Person) -- extends

function Student:learn()
    print("I am learning")
end

-- Professor class
local Professor = {__NAME = "Professor"}
Professor.__index = Professor
setmetatable(Professor, Person) -- extends

function Professor:teach()
    print("I am teaching")
end

local pe = Person:new()
local st = Student:new()
local pr = Professor:new()

local cat = {
    __NAME = "Person",
    nbr = 13,
    learn = function(); print("I am a cat; I cannot learn"); end
}
```

Using the function ***type***, Lua can afford the type of a given element: nil, boolean, number, string, userdata, function, thread, and table. Which means, creating a table representing a class and setting it as a metatable to another does not create a new type. There are various methods to find the class (metatable) of an object (Manura, 2013), one of them is to define a field **__NAME** in each class. This field must not be overridden in the copies (objects), otherwise it will not function appropriately. We can define a new function called **typeOf** based on this field. But, if we create an object (table) dynamically with a field **__NAME** equals to one of our classes, you can fool this function. An example of this is the object **cat** which will be considered as a **Person**. You can go beyond the function afforded in our code by verifying if the object (table) has a metatable. If it does not then its type is, lets say, **Class** whatever the **__NAME** is. If it has a metatable, you can check if its name is similar to its metable's. If not, you can consider it as a class; otherwise, the **__NAME** will be its type.

```lua
function typeOf(obj)
    if type(obj) ~= "table" then return type(obj) end
    if type(obj.__NAME) ~= "string" then return "table" end
    return obj.__NAME
end
```

There is no built-in function to verify if an object is instance of a class in Lua, since objects and classes are just tables. But using the solution used to verify the types, we can define a function **instanceof** which compare **__NAME** of an object with those of its metatable chains (metatable and metatable of metatable, etc.).

```lua
function instanceof(obj, cls)
    local mt = getmetatable(obj)
    if type(mt) ~= "table" or type(cls) ~= "table" then return false end
    if type(mt.__NAME) ~= "string" or type(cls.__NAME) ~= "string" then
        return false
    end
```

```
    if mt.__NAME == cls.__NAME then return true end
    return instanceof(mt, cls)
end
```

To verify if an object has a method, first you have to verify if the object is a table then if this method is of type **function**. Likewise, to verify the existence of a field, first you have to verify if the object is a table then if this field is not ***nil*** (also it is not a function).

```
function testMembers(msg, obj)
    io.write(msg .. ": ")
    if type(obj) == "table" and type(obj.talk) == "function" then
        obj:talk()
    else
        print("Method learn does not exist!")
    end

    io.write(msg .. ": ")
    if type(obj) == "table" and obj.nbr ~= nil then
        print("Field nbr = " .. tostring(obj.nbr))
    else
        print("Field nbr does not exist!")
    end
end
```

There is no cast in Lua, all happens in execution time and you will not be needing it.

2.5 Perl

Lets create our classes: Perl's classes are packages and objects are hashes blessed to a package.

```
package Person;
sub new { return bless {}, shift; }
sub talk { print "I am talking\n"; }
1;

package Student;
our @ISA = qw(Person);
sub learn { print "I am learning\n"; }
1;

package Professor;
our @ISA = qw(Person);
sub new { return bless {nbr => 5}, shift; }
sub teach { print "I am teaching\n"; }
1;

$pe = Person->new();
$st = Student->new();
$pr = Professor->new();
```

The function ***ref*** gives the package's name to which a hash is blessed.

```
print "ref(st): " . ref($st)  . "\n";
```

There is a built-in method ***isa*** inherited from the universal class ***UNIVERSAL*** which is used to determine the type of a reference. But before using it, we have to verify if the object is blessed.

```
use Scalar::Util qw(blessed);
sub instanceof {
    my ($obj, $cls) = @_;
```

```
    return (blessed($obj) and $obj->isa($cls));
}
```

To verify if an object has a method, Perl affords a method ***can*** inherited from the universal class ***UNIVERSAL***. To verify the existence of a field, a function ***exists*** can be used.

```
sub testMembers {
    my ($msg, $obj) = @_;
    print "$msg: ";
    if (blessed($obj) and $obj->can("learn")) {$obj->learn();}
    else { print "Method learn does not exist!\n";}

    print "$msg: ";
    if (blessed($obj) and exists($obj->{nbr})) {
        print "Field nbr = $obj->{nbr}\n";
    }
    else { print "Field nbr does not exist!\n";}
}
```

There is no cast in Perl, all happens in execution time and you will not be needing it.

2.6 PHP

Lets create our classes and one interface

```
class Person {
    public function talk(){ echo "I am talking\n"; }
}

class Student extends Person {
    public function learn(){ echo "I am learning\n"; }
}

interface Worker {}

class Professor extends Person implements Worker {
    public function teach(){ echo "I am teaching\n"; }
}

$pe = new Person();
$st = new Student();
$pr = new Professor();
```

The function ***gettype*** returns the type of a variable: the objects are of type **object**. To get the class of an object, the function ***get_class*** is used; it cannot be used with primitive types such as integers.

```
echo "gettype(st): " . gettype($st) . ", get_class(st): " . get_class($st) . "\n";
echo "gettype(25): " . gettype(25) . "\n";
```

To verify if an object is an instance of a class (or its parents), the keyword ***instanceof*** is used. Also, there is a function ***is_a*** which serves the same purpose, but the class's name is passed as a string or a variable containing the name.

```
echo "pr instanceof Professor: " . ($pr instanceof Professor);
echo ", is_a(pr, 'Professor'): " . is_a($pr, 'Professor') . "\n";
echo "pr instanceof Worker: " . ($pr instanceof Worker);
echo ", is_a(pr, 'Worker'): " . is_a($pr, 'Worker') . "\n";
```

To verify if an object has a method, PHP affords a function ***method_exists***. As for fields, it affords a function ***property_exists***. These two functions work on objects; if you pass them another type such as an integer, an error will occur. To test if a variable is an object, the function ***is_object*** can be used.

```
function testMembers($msg, $obj) {
    echo "$msg: ";
    if (is_object($obj) && method_exists($obj, "learn")) {$obj->learn();}
    else { echo "Method learn does not exist!\n";}

    echo "$msg: ";
    if (is_object($obj) && property_exists($obj, "nbr")) {
        echo "Field nbr = $obj->{nbr}\n";
    }
    else { echo "Field nbr does not exist!\n";}
}
```

There is no cast in PHP, all happens in execution time and you will not be needing it.

2.7 Python

Lets create our classes

```
class Person(object):
    def talk(self):
        print("I am a talking")

class Student(Person):
    def learn(self):
        print("I am learning")

class Professor(Person):
    def __init__(self):
        self.nbr = 5
    def teach(self):
        print("I am teaching")

class PhdStudent(Student, Professor): pass

pe = Person()
st = Student()
pr = Professor()
phd = PhdStudent()
```

The function ***type*** returns the type of a variable. Because everything in Python is an object, even built-in types such as **int** accept this function. Another mechanism to get an object's class is using the attribute ***__class__***. To get the name of a class as a string, you can use the attribute ***__name__***.

```
print("type(pe): " + type(pe).__name__)
print("st.__class__: " + st.__class__.__name__)
print("type(25): " + type(25).__name__)
```

To verify if an object is an instance of a class (or its parents), the function ***isinstance*** is used.

```
print ("isinstance(st, Student) " + str(isinstance(st, Student)))
print ("isinstance(phd, Professor) " + str(isinstance(phd, Professor)))
```

To verify if an object has a member, Python affords a function ***hasattr***. Another function ***getattr*** is used to get a member if it exists or returns a defined value otherwise. To verify if a member is a method, the function ***callable*** can be used.

```
import sys
def testMembers(msg, obj):
    sys.stdout.write(msg + ": ")
    learn = getattr(obj, "learn", None)
    if learn != None and callable(learn):
        learn()
    else:
        print("Method learn does not exist!")
    sys.stdout.write(msg + ": ")
    if hasattr(obj, "nbr"):
        print("Field nbr = " + str(obj.nbr))
    else:
        print("Field nbr does not exist!")
```

There is no cast in Python, all happens in execution time and you will not be needing it.

2.8 Ruby

Lets create our classes and a mixin

```
class Person
    def talk; puts "I am talking" end
end

class Student < Person
    def learn; puts "I am learning" end
end

class Professor < Person
    attr_reader :nbr
    def initialize; @nbr = 5 end
    def teach; puts "I am teaching" end
end

module LearnerMixin
    def learn; puts "I am a learner" end
end

class Robot
    include LearnerMixin
end

pe = Person.new
st = Student.new
pr = Professor.new
ro = Robot.new
```

To find the class of an object, you can simply call the method ***class***. Then, the method ***to_s*** is used to get the sting representation. In Ruby, everything is an object and even numbers has a class.

```
puts "st.class: " + st.class.to_s
puts "25.class: " + 25.class.to_s
```

To verify if an object is an instance of a class (or its parents), the functions ***kind_of?*** and ***is_a?*** can be used. The two methods are equivalent; they verify if an object inherit from a mixin as well. There is another method ***instance_of?*** which verify if an object was instantiated from a specific class (it does not verify its parents).

```
puts "st.is_a?(Person): " + st.is_a?(Person).to_s
puts "st.instance_of?(Student): " + st.instance_of?(Student).to_s
```

```
puts "st.is_a?(Professor): " + st.is_a?(Professor).to_s
puts "ro.kind_of?(LearnerMixin): " + ro.kind_of?(LearnerMixin).to_s
```

To verify if an object has a method, a class of an object has a method ***method_defined?***. As for fields, they are encapsulated (protected mode) and cannot be accessed outside the class's hierarchy unless you define accessors (getters and setters). The accessors are methods and can be checked the same way a normal method is checked. In our example, **Professor** class has a field **nbr** and a reader accessor (getter). To verify the setter (not shown in the example), you write **obj.method_defined? :nbr=**.

```
def testMembers (msg, obj)
    print "#{msg}: "
    if obj.class.method_defined? :learn then obj.learn
    else puts "Method learn does not exist!" end

    print "#{msg}: "
    if obj.class.method_defined? :nbr then puts "Field nbr = #{obj.nbr}"
    else puts "Field nbr does not exist!" end
end
```

There is no cast in Ruby because variable type is dynamic.

3 Methods overloading

It is the feature of a class having multiple methods defined by the same identifier and different parameters. Statically-typed programming languages are, usually, the ones affording methods overloading since they enforce type checking during function calls. In this section, we will verify these properties:

- If the programming language supports methods overloading. If not, how to afford something similar.
- If we can overload methods over inheritance

3.1 C++

In C++, overloading is permitted in the same class

```
class Person {
public:
    void read(){ std::cout << "I am reading" << std::endl; }
    void read(std::string text){ std::cout << "A text: " << text << std::endl; }
};
int main()
{
    Person pe;
    pe.read();
    pe.read("I am a person");
    return 0;
}
```

When the same method is overloaded in a subclass, the new definition will hide those of the parent class; if they are called, this will generate a compilation error. To call them, there are two ways: import them using ***using*** declaration, or by calling them explicitly (Bailey, 2009).

```
class Student: public Person {
public:
    using Person::read;
```

```
    void read(int nbr){ std::cout << "I read on table n°: " << nbr << std::endl; }
};

int main()
{
    Student st;
    st.read(); // using Person::read; or error
    st.Person::read(); // Without using Person::read;
    st.read("I am a student"); // using Person::read; or error
    st.read(5);

    return 0;
}
```

3.2 Java

In Java, overloading is permitted in the same class or its subclasses as well.

```
public class Overloading {
    public static void main(String[] args) {
        Person pe = new Person();
        Student st = new Student();
        pe.read();
        pe.read("I am a person");
        st.read();
        st.read("I am a student");
        st.read(5);
    }
}

class Person {
    public void read(){ System.out.println("I am reading"); }
    public void read(String  text){ System.out.println("A text: " + text); }
}

class Student extends Person {
    public void read(int nbr){ System.out.println("I read on table n°: " + nbr); }
}
```

3.3 Javascript

Javascript does not support method overloading. A similar thing can be achieved by testing arguments types.

```
class Person {
    read(arg1) {
        switch (typeof arg1) {
            case "undefined":
                console.log("I am reading");
                break;
            case "string":
                console.log("A text: ", arg1);
                break;
            default:
        }
    }
}
let pe = new Person();
pe.read();
pe.read("I am a person");
```

If there are many arguments, you can use the keyword ***arguments*** which is an Array-like object accessible inside functions that contains the values of the arguments passed to that function. Subclasses can define the same method, verify the arguments for new definitions and delegate the control to the super's methods.

```
class Student extends Person {
    read(arg1){
        if (typeof arg1 == "number"){
            console.log("I read on table n°: ", arg1);
        } else {
            super.read(arg1);
        }
    }
    count(){
        console.log("number of arguments: ", arguments.length);
    }
}
let st = new Student();
st.read();
st.read("I am a student");
st.read(5);
st.count(5, "text", 10);
```

3.4 Lua

Lua does not support method overloading. A similar behavior can be achieved by using a ***variadic function*** and testing arguments number and types. You can select a parameter using the function ***select*** which returns the nth parameter and its following parameters (RBerteig of stackoverflow, 2011). The same function can be used to know te number of parameters.

```
local Person = {}
Person.__index = Person

function Person:new()     -- The constructor
    local instance = {}
    setmetatable(instance, self) --self, here, will refer to the calling class
    return instance
end

function Person:read(...)
    local arg1 = select(1, ...)
    if not arg1 then -- select("#", ...) for parameters number
        print("I am reading")
    else
        if type(arg1) == "string" then
            print("A text: " .. arg1)
        end
    end
end
local pe = Person:new()
pe:read()
pe:read("I am a person")
```

Subclasses can define the same method, verify the arguments for new definitions and delegate the control to the super's methods. If you do not want to use ***select***, you can put the arguments in a list and access the parameters as list members.

```
-- Student class
local Student = {}
Student.__index = Student
setmetatable(Student, Person) -- extends
```

```
5
6  function Student:read(...)
7      local args = {...}
8      if type(args[1]) == "number" then
9          print("I read on table n°: " .. args[1])
10     else
11         Person.read(self, ...)
12
13 print("STUDENT");
14 local st = Student:new()
15 st:read()
```

3.5 Perl

Perl does not support method overloading. A subroutine receive its parameters using an array @_; thus, it is possible to verify the number of arguments (the first one is the object). If you pass more than the acceptable amount of arguments, this will not be a problem (as shown in the example). Perl does not make a difference between scalar types (eg. numbers and strings), so you can just verify blessed objects using *ref*.

```
1  package Person;
2  sub new { return bless {}, shift; }
3  sub read {
4      if (@_< 2 ) {
5          print "I am reading\n";
6          return;
7      }
8      my ($self, $arg1) = @_;
9      print "I am reading: $arg1\n";
10 }
11 1;
12 $pe = Person->new();
13 $pe->read();
14 $pe->read("Book");
15 $pe->read("Book", 5);
```

Subclasses can define the same method, verify the arguments for new definitions and delegate the control to the super's methods.

```
1  package Student;
2  our @ISA = qw(Person);
3  sub read {
4      if (@_ == 3) {
5          my ($self, $arg1, $arg2) = @_;
6          print "I am reading: $arg1 for $arg2 times\n";
7      }
8      else {
9          $self = shift;
10         $self->SUPER::read(@_);
11     }
12 }
13 1;
14 $st = Student->new();
15 $st->read();
16 $st->read("Book");
17 $st->read("Book", 5);
```

You can check out *MooseX::MultiMethods*[1] for Multi Method Dispatch based on Moose type constraints.

[1] MooseX::MultiMethods: https://metacpan.org/pod/MooseX::MultiMethods

3.6 PHP

PHP does not support OOP's method overloading; Overloading in PHP is entirely a different concept (check PHP documentation). However, there is a mechanism to achieve a similar thing using a ***variadic function***: the number of arguments is recovered using a function ***func_num_args***, and the arguments are recovered using a function ***func_get_arg*** (Tan, 2011).

```php
class Person {
    public function read(){
        switch (func_num_args()) {
            case 0:
                echo "I am reading\n";
                break;
            case 1:
                $arg1 = func_get_arg(0);
                if (gettype($arg1) == "string"){
                    echo "A text: $arg1\n";
                }
                break;
            default:
        }
    }
}
$pe = new Person();
$pe->read();
$pe->read("I am a person");
```

Subclasses can define the same method, verify the arguments for new definitions and delegate the control to the super's methods. Arguments can, also, be passed as a list as follows.

```php
class Student extends Person {
    public function read(...$args){
        if (count($args) == 1 && gettype($args[0]) == "integer"){
            echo "I read on table n°: ". $args[0] . "\n";
        }
        else { parent::read(...$args); }
    }
}
$st = new Student();
$st->read();
$st->read("I am a student");
$st->read(5);
```

3.7 Python

Python does not support method overloading. Using default values and type checking, you can achieve a similar behavior.

```python
class Person(object):
    def read(self, text=None):
        if text == None:
            print("I am reading")
        elif type(text) is str:
            print("A text: " + text)
pe = Person()
pe.read()
pe.read("I am a person")
```

You can, also, use a ***variadic function*** for unknown amount of parameters. Subclasses can define the same method, verify the arguments for new definitions and delegate the control to the super's methods.

```
class Student(Person):
    def read(self, *args):
        if len(args) == 0:
            Person.read(self)
        elif len(args) == 1:
            Person.read(self, args[0])
        else:
            print("I am reading: " + str(args[0]) + " for " + str(args[1]) + " times")
st = Student()
st.read()
st.read("I am a student")
st.read("a book", 5)
```

You can define an Application Programming Interface (API) for methods overloading such as ***python-langutil***[2].

3.8 Ruby

Ruby does not support method overloading. Using default values and type checking, you can achieve a similar behavior.

```
class Person
    def read(text=nil)
        if text == nil then
            puts "I am reading"
        else
            if text.is_a?(String) then
                puts "A text: #{text}"
            end
        end
    end
end
pe = Person.new
pe.read
pe.read("I am a person")
```

You can, also, use a ***variadic function*** for unknown amount of parameters. Subclasses can define the same method, verify the arguments for new definitions and delegate the control to the super's methods.

```
class Student < Person
    def read(*args)
        case args.length
        when 0
            super
        when 1
            super(args[0])
        when 2
            puts "I am reading: " + args[0].to_s + " for " + args[1].to_s + " times"
        end
    end
end
st = Student.new
st.read
st.read("I am a person")
st.read("a book", 5)
```

[2]pythonlangutil: https://github.com/ehsan-keshavarzian/pythonlangutil

4 Methods overriding

Method overriding allows a subclass to provide a specific implementation of an existing parent's method. The method must have the same signature (name and arguments) and return type (if the language allows it) as the parent's. Some methods can be abstract or final: abstract methods must be overridden and final methods cannot be overridden (see inheritance chapter).

4.1 C++

To override a method, you have to define a subclass with the same method's signature and return type. A method to be overridden can be defined using the keyword ***virtual*** or not. A virtual method is resolved at run-time using a mechanism called ***Late binding***. While a normal method is resolved at compilation-time using a mechanism called ***Early (Static) binding***.

```
1  class Person {
2  public:
3      virtual void talk(){ std::cout << "I am a person" << std::endl; }
4      void type(){ std::cout << "Person" << std::endl; }
5  };
6
7  class Student: public Person {
8  public:
9      void talk(){ std::cout << "I am a student" << std::endl; }
10     void type(){ std::cout << "Student" << std::endl; }
11 };
```

Early binding follows the pointer's type when the method is called; therefore, if the pointer's type is a superclass, the implementation which will be used is that of the superclass are not the object's. In the contrary, a method defined using ***virtual*** (Late binding) will be called based on the object's real type and not that of the pointer.

```
1  int main()
2  {
3      Person *pe = new Person();
4      Student *st = new Student();
5      Person *pst = st;
6
7      pe->talk(); // I am a person
8      st->talk(); // I am a student
9      pst->talk(); //I am a student
10
11     pe->type(); //Person
12     st->type(); //Student
13     pst->type(); //Person
14
15     return 0;
16 }
```

4.2 Java

In Java, you can override a method using the same signature and return type. As mentioned in Encapsulation chapter, the visibility mode can be set to more permissive one (for example, from private to public). To be sure you overrode an existing method, you can use the annotation ***@Override*** which tells the compiler to check if the method actually exists in one of the superclasses.

```
1  class Person {
2      public void talk(){ System.out.println("I am a person"); }
```

```
}

class Student extends Person {
    @Override
    public void talk(){ System.out.println("I am a student"); }
}
```

An object of a class referred by a reference of its superclass will execute its specific implementation and not that of the reference.

```
public class Overriding {
    public static void main(String[] args) {
        Person pe = new Person();
        Person st = new Student();
        pe.talk(); // I am a person
        st.talk(); // I am a student
    }
}
```

4.3 Javascript

In Javascript, you can override a method by just defining it using the same signature. The object of the subclass will use the new defined method.

```
class Person {
    talk(){ console.log("I am a person"); }
}

class Student extends Person {
    talk(){ console.log("I am a student"); }
}

let pe = new Person(),
    st = new Student();
pe.talk(); // I am a person
st.talk(); // I am a student
```

4.4 Lua

In Lua, you can override a method by just defining it using the same signature. The object of the subclass will use the new defined method.

```
-- Person class
local Person = {}
Person.__index = Person

function Person:new()    -- The constructor
    local instance = {}
    setmetatable(instance, self) --self, here, will refer to the calling class
    return instance
end

function Person:talk()
    print("I am a person")
end

-- Student class
local Student = {}
Student.__index = Student
setmetatable(Student, Person) -- extends
```

```lua

function Student:talk()
    print("I am a student")
end

local pe = Person:new()
local st = Student:new()
pe:talk() -- I am a person
st:talk() -- I am a student
```

4.5 Perl

In Perl, you can override a method by just defining it using the same name. The object of the subclass will use the new defined method.

```perl
package Person;
sub new { return bless {}, shift; }
sub talk { print "I am a person\n"; }
1;

package Student;
our @ISA = qw(Person);
sub talk { print "I am a student\n"; }
1;

$pe = Person->new();
$st = Student->new();
$pe->talk(); # I am a person
$st->talk(); # I am a student
```

4.6 PHP

In PHP, you can override a method by just defining it using the same signature. The object of the subclass will use the new defined method.

```php
class Person {
    public function talk(){ echo "I am a person\n"; }
}

class Student extends Person {
    public function talk(){ echo "I am a student\n"; }
}

$pe = new Person();
$st = new Student();
$pe->talk(); // I am a person
$st->talk(); // I am a student
```

4.7 Python

In Python, you can override a method by just defining it using the same signature. The object of the subclass will use the new defined method.

```python
class Person(object):
    def talk(self):
        print("I am a person")

class Student(Person):
```

```
    def talk(self):
        print("I am a student")

pe = Person()
st = Student()
pe.talk() # I am a person
st.talk() # I am a student
```

4.8 Ruby

In Ruby, you can override a method by just defining it using the same signature. The object of the subclass will use the new defined method.

```
class Person
    def talk()
        puts "I am a person"
    end
end

class Student < Person
    def talk()
        puts "I am a student"
    end
end

pe = Person.new
st = Student.new
pe.talk # I am a person
st.talk # I am a student
```

Discussion

Polymorphism allows us to process objects differently depending on their data type or class. Based on type compatibility and equivalence, three type systems can be determined:

- **Nominal type system**: The compatibility and equivalence of data types is is determined by explicit declarations or name. Example: C++, C#, Java.
- **Structural type system**: The compatibility and equivalence of data types is is determined by their structures or definitions: if they have the same definition then they are equivalent. Example: OCaml, Scala, Go.
- **Duck typing**: The compatibility is determined by the presence of certain methods and fields. "*If it walks like a duck and talks like a duck, it must be a duck*". Example: Javascript, Python, Ruby.

Table 6.1 represents a comparison between some OOP languages based on their polymorphism capacities:

- **Type compatibility**: Nominal, structural or duck.
- **Get object type**: How to get the type of a given object.
- **Is instance of**: Verify if an object is generated from a class/prototype.
- **Members existence**: Check if a member (field or method) exists in a given object.
- **Type casts**: Downcasting is the act of changing a reference of a base class to one of its derived class.
- **Overloading**: Having methods overloading or not.

As for overriding, no need to compare since all of our languages afford this property.

Table 6.1: Polymorphism comparison

Language	Type compat-ibility	Object's type	Instance of	members existence	cast	overlo
C++	Nominal	typeid(obj)	Using cast	/	(Cls) obj; static_cast<Cls*> (ptr) dynamic_cast<Cls*> (ptr)	Yes
Java	Nominal	obj.getClass()	obj instanceof Cls	using reflection	(Cls) obj	Yes
Javascript	Duck	Type: typeof obj Class: obj.constructor.name	obj instanceof Cls	obj.hasOwnProperty ("property")	/	/
Lua	Duck	Type: type(obj) Class: /	/	access and verify type	/	/
Perl		ref(obj)	obj->isa(Cls)	obj->can("method")	/	/
PHP	Duck?	Type: gettype(obj) Class: get_class(obj)	obj instanceof Cls is_a(obj, "Cls")	method_exists(obj, "method") property_exists(obj, "field")	/	/
Python	Duck	type(obj) obj.__class__	isinstance(obj, Cls)	hasattr(obj, "property")	/	/
Ruby	Duck	obj.class	obj.is_a?(Cls) instance_of?(Cls) kind_of?(Cls)	obj.class.method_defined? :method	/	/

BIBLIOGRAPHY

[Alinkaya 2008] Alinkaya, M. U.: *Making functions non override-able*. 2008. – URL: https://stackoverflow.com/a/321334. – Access date: Sep. 9, 2018

[Ashton 2001] Ashton, Elaine: *The Timeline of Perl and its Culture*. 2001. – URL: http://history.perl.org/PerlTimeline.html. – Access date: Sep. 8, 2018

[Bailey 2009] Bailey, CB: *Overloading a method in a subclass in C++*. 2009. – URL: https://stackoverflow.com/a/1734905. – Access date: Oct 6, 2009

[Black 2013] Black, Andrew P.: Object-oriented Programming: Some History, and Challenges for the Next Fifty Years. In: *Inf. Comput.* 231 (2013), October, S. 3–20. – URL: http://web.cecs.pdx.edu/~black/publications/O-JDahl.pdf. – ISSN 0890-5401

[Borning 1986] Borning, A. H.: Classes Versus Prototypes in Object-oriented Languages. In: *Proceedings of 1986 ACM Fall Joint Computer Conference*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1986 (ACM '86), S. 36–40. – URL: http://dl.acm.org/citation.cfm?id=324493.324538. – ISBN 0-8186-4743-4

[btrott of PerlMonks 2000] btrott of PerlMonks: *Objects with Private Variables*. 2000. – URL: https://www.perlmonks.org/?node_id=8251. – Access date: Sep. 9, 2018

[Bugayenko 2016] Bugayenko, Yegor: *What's Wrong With Object-Oriented Programming?* 2016. – URL: https://www.yegor256.com/2016/08/15/what-is-wrong-object-oriented-programming.html. – Access date: August 30, 2018

[Byers 2010] Byers, Mark: *In Python, how can I prohibit class inheritance? [duplicate]*. 2010. – URL: https://stackoverflow.com/a/3949004. – Access date: Sep. 9, 2018

[Cook u. a. 1990] Cook, William R. ; Hill, Walter ; Canning, Peter S.: Inheritance is Not Subtyping. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1990 (POPL '90), S. 125–135. – URL: http://doi.acm.org/10.1145/96709.96721. – ISBN 0-89791-343-4

[Cowburn 2018] Cowburn, Peter: *PHP Manual*, 2018. – URL: http://php.net/manual/en/. – Access date: Sep. 5, 2018

[Crockford 2001] Crockford, Douglas: *Private Members in JavaScript*. 2001. – URL: https://crockford.com/javascript/private.html. – Access date: Sep. 9, 2018

[Cunningham 2014] Cunningham, Ward: *Alan Kays Definition Of Object Oriented*. 2014. – URL: http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented. – Access date: August 26, 2018

[ddimitrov 2008] ddimitrov: *Is there a destructor for Java?* 2008. – URL: https://stackoverflow.com/a/171961. – Access date: Sep. 8, 2018

[Deschenes 2014] Deschenes, Francois: *PHP Variable Overriding.* 2014. – URL: https://stackoverflow.com/a/6339221. – Access date: Sep. 9, 2018

[Dorin 2016] Dorin, Poelinca: *ES6 Class Multiple inheritance.* 2016. – URL: https://stackoverflow.com/a/35925061. – Access date: Sep. 5, 2018

[GeeksforGeeks 2012] GeeksforGeeks: *Multiple Inheritance in C++.* 2012. – URL: https://www.geeksforgeeks.org/multiple-inheritance-in-c/. – Access date: Sep. 5, 2018

[glasswalk3r of PerlMonks 2000] glasswalk3r of PerlMonks: *Dealing with abstract methods in Perl 5.8.* 2000. – URL: https://www.perlmonks.org/?node_id=611771. – Access date: Sep. 9, 2018

[Half 2017] Half, Robert: *4 Advantages of Object-Oriented Programming.* 2017. – URL: https://www.roberthalf.com/blog/salaries-and-skills/4-advantages-of-object-oriented-programming. – Access date: Sep. 8, 2018

[Huang 2004] Huang, Jian: *Lecture notes in Fundamental Algorithms (CS302).* 2004. – URL: http://web.eecs.utk.edu/~huangj/CS302S04/lecture_notes.html. – Access date: August 26, 2018

[Ierusalimschy 2003] Ierusalimschy, Roberto: *Programming in Lua.* Lua.org, 2003. – URL: https://www.lua.org/pil/contents.html#P2. – ISBN 8590379817

[Janssen 2017a] Janssen, Thorben: *OOP Concept for Beginners: What is Abstraction?* 2017. – URL: https://stackify.com/oop-concept-abstraction/. – Access date: Sep. 7, 2018

[Janssen 2017b] Janssen, Thorben: *OOP Concept for Beginners: What is Encapsulation.* 2017. – URL: https://stackify.com/oop-concept-abstraction/. – Access date: Sep. 7, 2018

[Kay 1993] Kay, Alan C.: The Early History of Smalltalk. In: *SIGPLAN Not.* 28 (1993), March, Nr. 3, S. 69–95. – URL: http://doi.acm.org/10.1145/155360.155364. – ISSN 0362-1340

[Kazarian 2016] Kazarian, Avétis: *Advanced JavaScript Class: Abstract Class & Method.* 2016. – URL: https://coderwall.com/p/r47jlw/advanced-javascript-class-abstract-class-method. – Access date: Sep. 9, 2018

[Knoernschild 2002] Knoernschild, K.: *Java Design: Objects, UML, and Process.* Addison-Wesley, 2002. – URL: https://books.google.dz/books?id=4pjbgVHzomsC. – ISBN 9780201750447

[Liskov 1987] Liskov, Barbara: Keynote Address - Data Abstraction and Hierarchy. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum).* New York, NY, USA : ACM, 1987 (OOPSLA '87), S. 17–34. – URL: http://doi.acm.org/10.1145/62138.62141. – ISBN 0-89791-266-7

[Lua users 2011] Lua users: *Object Orientation Closure Approach.* 2011. – URL: http://lua-users.org/wiki/ObjectOrientationClosureApproach. – Access date: Sep. 9, 2018

[Lua.org 2018] Lua.org: *Lua: About*. 2018. – URL: https://www.lua.org/about.html. – Access date: Sep. 8, 2018

[Manura 2013] Manura, David: *Type Introspection*. 2013. – URL: http://lua-users.org/wiki/TypeIntrospection. – Access date: Apr 6, 2013

[MDN web docs 2018a] MDN web docs: *JavaScript reference*. 2018. – URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference. – Access date: Sep. 9, 2018

[MDN web docs 2018b] MDN web docs: *Object.create()*. 2018. – URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create. – Access date: Sep. 5, 2018

[Meyer 2014] Meyer, Mark: *Calling method in parent class from subclass methods in Ruby*. 2014. – URL: https://stackoverflow.com/a/18448863. – Access date: Sep. 9, 2018

[MVP of stackoverflow 2016] MVP of stackoverflow: *How to make a base class method non-overridable in ruby?* 2016. – URL: https://stackoverflow.com/a/40247731. – Access date: Sep. 9, 2018

[Naftalin 2012] Naftalin, Maurice: *http://www.lambdafaq.org/what-about-the-diamond-problem/*. 2012. – URL: Whataboutthediamondproblem?. – Access date: Sep. 5, 2018

[Nørmark 2013] Nørmark, Kurt: *Programming Paradigms: Overview of the four main programming paradigms*. 2013. – URL: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html. – Access date: Sep. 7, 2018

[oracle n.d.] oracle: *rail: Learning the Java Language*. n.d.. – URL: https://docs.oracle.com/javase/tutorial/java/index.html. – Access date: Sep. 5, 2018

[Panigrahy 2014] Panigrahy, Nilanchala: *History of Java Programming Language*. 2014. – URL: https://stacktips.com/tutorials/java/history-of-java-programming-language. – Access date: Sep. 8, 2018

[Panzenböck 2014] Panzenböck, Mathias: *C++ equivalent of instanceof*. 2014. – URL: https://stackoverflow.com/a/25231384. – Access date: Aug 10, 2014

[Popyack u. a. 2015] Popyack, Jeffrey L. ; Boady, Mark W. ; Zoski, Paul ; Salvage, Jeff: *Lecture notes: Introduction to Computer Science*. 2015. – URL: https://www.cs.drexel.edu/~introcs/Fa15/courseNotes_index.html. – Access date: Sep. 8, 2018

[Python Software Foundation 2018] Python Software Foundation: *Python 2.7.15 documentation*, 2018. – URL: https://docs.python.org/2/. – Access date: Sep. 6, 2018

[radiantmatrix of PerlMonks 2006] radiantmatrix of PerlMonks: *Private and Protected class methods*. 2006. – URL: https://www.perlmonks.org/?node_id=571509. – Access date: Sep. 9, 2018

[Rahman 2014] Rahman, Maqsood: *Confused about the definition of 'abstraction' in OOP*. 2014. – URL: https://softwareengineering.stackexchange.com/a/230409. – Access date: Sep. 7, 2018

[Rajhans 2009] Rajhans, Chirantan: *How to implement an abstract class in ruby?* 2009. – URL: https://stackoverflow.com/q/512466. – Access date: Sep. 9, 2018

[RBerteig of stackoverflow 2011] RBerteig of stackoverflow: *Variable number of function arguments Lua 5.1.* 2011. – URL: https://stackoverflow.com/a/7630202. – Access date: Oct 3, 2011

[rir of PerlMonks 2006] rir of PerlMonks: *Re: Private and Protected class methods.* 2006. – URL: https://www.perlmonks.org/?node_id=571548. – Access date: Sep. 9, 2018

[Rubens 2018] Rubens, Arden: *The History of JavaScript [INFOGRAPHIC].* 2018. – URL: https://www.checkmarx.com/2018/02/12/history-javascript-infographic/. – Access date: Sep. 8, 2018

[Saitta and Zucker 2013] Saitta, Lorenza ; Zucker, Jean-Daniel: *Abstraction in Different Disciplines.* In: *Abstraction in Artificial Intelligence and Complex Systems*, Springer-Verlag New York, 2013. – ISBN isbn

[Sauer 2013] Sauer, Lorenz L.: *Getter and Setter?* 2013. – URL: https://stackoverflow.com/a/18683330. – Access date: Sep. 8, 2018

[sbi of stackoverflow 2009] sbi of stackoverflow: *final class in c++.* 2009. – URL: https://stackoverflow.com/a/1366482. – Access date: Sep. 9, 2018

[Schwartz and Phoenix 2003] Schwartz, Randal L. ; Phoenix, Tom: *Learning Perl Objects, References, and Modules.* O'Reilly & Associates, 2003. – URL: https://docstore.mik.ua/orelly/perl4/porm/index.htm. – Access date: Sep. 5, 2018. – ISBN 0-596-00478-8

[Stranden 2015] Stranden, Håvard: *Is it good practice to NULL a pointer after deleting it?* 2015. – URL: https://stackoverflow.com/a/1931171. – Access date: Sep. 8, 2018

[Stroustrup 1993] Stroustrup, Bjarne: A History of C++: 1979&Ndash;1991. In: *SIGPLAN Not.* 28 (1993), March, Nr. 3, S. 271–297. – URL: http://doi.acm.org/10.1145/155360.155375. – ISSN 0362-1340

[Tan 2011] Tan, Daniel: *PHP function overloading.* 2011. – URL: https://stackoverflow.com/a/4697712. – Access date: Jan 15, 2011

[TechDifferences 2016] TechDifferences: *Difference Between Inheritance and Polymorphism.* 2016. – URL: https://techdifferences.com/difference-between-inheritance-and-polymorphism.html. – Access date: Sep. 8, 2018

[Thomas and Hunt 2001] Thomas, D. ; Hunt, A.: *Programming Ruby: The Pragmatic Programmer's Guide.* Addison-Wesley, 2001. – URL: http://ruby-doc.com/docs/ProgrammingRuby/. – Access date: Sep. 6, 2018. – ISBN 9780201710892

[Török 2010] Török, Péter: *How to define 'final' member functions for a class.* 2010. – URL: https://stackoverflow.com/a/3625560. – Access date: Sep. 9, 2018

[Ungar and Smith 1991] Ungar, David ; Smith, Randall B.: SELF: The power of simplicity. In: *LISP and Symbolic Computation* 4 (1991), Jul, Nr. 3, S. 187–205. – URL: https://doi.org/10.1007/BF01806105. – ISSN 1573-0557

[Van Roy and Haridi 2004] Van Roy, Peter ; Haridi, Seif: *Concepts, Techniques, and Models of Computer Programming*. 1st. The MIT Press, 2004. – URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.7366&rep=rep1&type=pdf. – ISBN 0262220695, 9780262220699

INDEX

Lua

Perl

PHP

Python

Ruby

That’s all

…

The END