# Programming Paradigms Introduction

## Introduction

I’ve probably talked about my issue about the title of this course. You see, it’s a misnomer, it’s not named correctly. Most of this course will actually focus on objective oriented programing, you can see it in the syllabus. That being the case, it should have been named object oriented programming instead. But we ARE going to spend some weeks to talk about programming paradigms at the start of this course.

## Learning Outcomes

At the end of this discussion you should be able to:

1. Explain what a programming paradigm is.
2. Identify the four main programming paradigms
3. Explain why programming languages are shifting to multi-paradigmness

## The title of this course

Let’s start by talking about the notorious title of the course name because it does sound like some forced alliteration buzz phrase that you’ll probably hear out of the mouth of some CS student. You will start saying this phrase soon so let’s get the definition out of the way.

“Programming paradigms”.

You’re probably familiar what half of this phrase means, I mean, I hope you are, otherwise I don’t know what to do.

## Paradigm

Let’s focus first on the non-obvious part, the word paradigm. This word comes up often in academia. You probably heard of the term paradigm shift some where, it describes some form of fundamental change in the way we think within scientific disciplines often characterizing a scientific revolution, one notable example of a paradigm is the shift from Ptolemaic or Geocentric cosmology to Copernican or Heliocentric cosmology. Based on this context you can kind of formulate what the word paradigm means. You don’t have to take out your dictionaries or whatever, because I’m going to read to you a dictionary definition I found. This one is connotes a similar meaning in the context of programming:

A paradigm is “a worldview underlying the theories and methodology of a particular subject” [[1]](#footnote-25). It is a set of ideas and concepts that describe some way of thinking.

If we go back to the geocentric vs heliocentric paradigms in astronomy, you can’t really definitively say that the heliocentric model is the only correct model of the solar system, you can still reconcile the geocentric model’s perspective of putting the earth in the center by imagining heavenly bodies with strange orbits containing epicycles and other mechanics. After all whether or not the earth or the sun is the center is a matter of perspective. It just so happens that placing the sun in the center provided science with a more natural way of describing planetary movement. The heliocentric paradigm ended up uprooting geocentric paradigm as the dominant worldview, providing science with ideas that we still accept as truth until now, the earth is not the center. The earth is just one of the 9 planets, is not that special, gravity and inertia works in way which causes planets to move and etc.

A paradigm shift like this is actually happening in programming language design, well talk about that some time later.

## Programming

Now that you understand what paradigm means, let’s talk about the first word, programming. It’s a strange question to ask in a second year course but I want you all to think about what the definition of programming is. The way you answer this question may actually tell you which perspective or programming PARADIGM you follow. When you say you are “programming” some kind of mechanism or behavior what are you actually doing? How do you define what a program is and what is its relationship to a computer?

I would have loved to hear your answers on this but since we cant do that. I’ll tell you instead to seriously think about that question before I proceed.

Since I don’t know how you responded to that I’ll instead turn to the internet and look for an answer that probably looks like your answer.

Here’s one:

“Computer programming is the process that professionals use to write code that instructs how a computer, application or software program performs. At its most basic, computer programming is a set of instructions to facilitate specific actions.[[2]](#footnote-27)”

That is correct. Let me simplify that definition to this.

Programming is when you tell a computer what to do. When you write programs you’re writing instructions for your computer.

Step 1. Ask the user for a number

Step 2. Store that number to a variable called x.

Step 3. While x is greater than 7 do step 4 otherwise proceed

Step 4. Subtract 7 to x and store the difference to x

Step 5. Show the user the value of x

That is a good definition of programming. It gives you an understanding on how you write programs that work. All you need to do is to write correct instructions that the computer understands and you’ll have a perfectly working program. A programming language is a medium that describes how to write instructions to communicate to your computer. If you learn to do that then you can go ahead and program away.

It is a correct definition, but is it the only correct definition? It defines programming under the paradigm imperative programming. I will not begrudge you if this is the only definition you know since there is a huge likelihood that the only paradigm you’ve been exposed to has been imperative programming.

## Taxonomy of Programming Paradigms

For someone who has been exposed to C, C++ and nothing else, you might feel that the natural way to code is the *imperative way* when in fact there are alternatives.

![Programming Paradigms](data:text/html; charset=utf-8;base64,)

Programming Paradigms

The diagram[[3]](#footnote-30) here represents the alternative schools of thoughts describing how to program. This diagram taxonomizes programming languages by identifying which paradigms they are under. Most of these paradigms are either not pragmatic, not popular enough or not unique enough to be studied in this course. Instead we will be focusing on four major programming paradigms:

graph TD;  
 Imperative-->Procedural;  
 Imperative-->ObjectOriented;  
 Declarative-->Functional;  
 Declarative-->Logic;

Under the imperative family, procedural programming and objective oriented programming.

Under the declarative family, functional programming and logic programming.

This course will give you an overview on these programming paradigms. Each of these are built upon the foundation of some mathematical formalism. We will explore the advantages and disadvantages of each paradigm while we take a tour through these four. Studying the disciplines upheld by these paradigms will also teach us good programming practices for designing elegant programs that transcends any programming paradigm.

## Multi-paradigm programming languages

The way it used to be was that a programming language would be written with features adhering to the concepts of a particular paradigm. Sometimes, a language is written with fresh features that follow a different mathematical formalism that births its own programming paradigm. Back then paradigms worked like programming language classifications. The programming language C for example is a strong follower of procedural programming. Therefore, you can think of C as classified under procedural programming.

But as time passed by classifying a newer programming language under one paradigm became harder and harder. A programming language like python for example is mostly procedural, object-oriented, but sometimes functional.

Modern programming languages evolved to become multi-paradigm. This inevitably happened because, as programming languages age and grow, more features are added to it. These features are sometimes borrowed from other paradigms to solve a problem in a better way. This is the reason why established and mainstream programming languages like Java, C++, or Python tend to be multi-paradigm.

The multi-paradigmness of programming languages tend to be the reason why some programming language designers have abandoned the notion of building based on a strict paradigm. Instead a language designer would choose specific **features** that they want to be supported on their programming language and implement it, regardless of its paradigm origins.

## Optional Readings

Van Roy, Peter. (2012). [Programming Paradigms for Dummies](https://www.researchgate.net/publication/241111987_Programming_Paradigms_for_Dummies_What_Every_Programmer_Should_Know): What Every Programmer Should Know.

# Imperative Programming

## Introduction

Imperative programming has turned out to be the natural paradigm of programming languages. The members of the imperative programming family has been dominating the market share of programming languages throughout the years with titans like BASIC, Pascal, C, Java and many more.

## Learning Outcomes

At the end of this discussion you should be able to

1. Explain how imperative programming became the natural paradigm
2. Explain the concept of state in the context of imperative programming
3. Explain how the assignment statement enables the progression of states
4. Create structure programs to represent algorithms
5. Differentiate the subparadigms procedural programming and object-oriented programming

**Quick Note on Imperative Programming and Procedural Programming**

People usually use the terms Imperative programming and procedural programming interchangeably. Procedural programming is a subparadigm of imperative programming family but some people refer to procedural programming as imperative programming. That’s because other imperative paradigms like object-oriented programming is derived from procedural programming. You can think procedural programming as the ancestor if other imperative paradigms.

In this lecture I refer to Imperative paradigm as a whole but I will focus on the main ideas that are common between other imperative paradigms. Ideas from object-oriented programming paradigm can be found on a separate lecture.

Imperative programming has turned out to be the natural paradigm of programming languages. The members of the imperative programming family has been dominating the market share of programming languages throughout the years with titans like BASIC, Pascal, C, Java and many more.

If you think about it, this is not really surprising. This paradigms dominance could be attributed to most computer scientists’ preference towards pragmatic and efficient programming languages. Especially since the most straightforward way of communicating to computer hardware is through the explicit manipulation of CPU memory and registers.

If you want the computer to do something for you, then you communicate to the computer that you want this and that to be done. And if you manage to give the computer correct and comprehensive instructions then you’ll end up getting what you want.

If we rewind back to the dawn of programming languages you’ll see that early programming languages were built to communicate to computer hardware. As a result of this, programming languages naturally adopted syntax with imperative moods.

Assembly programs for example was mostly built from sequences of executable instructions which was patterned from imperative statements from natural language.

INC ITER  
MOV AH,7  
ADD AH, AL

For example the assembly instruction, INC ITER, tells the computer to increment the memory variable called ITER. The instruction MOV AH, 7, tells the computer to move the value 7 to the AH register. The instruction ADD AH, AL tells the computer to add the contents of the AH register to the AL register.

As time went by, newer higher level programming languages emerged (higher level meaning farther from hardware and closer to human language) like Basic, Pascal, and C. The syntax of these programming languages were written as abstractions of hardware code. Although programs written in these languages became more human readable compared to its predecessors, these newer languages retained their imperative tones and mechanisms. This progression meant that higher level programming languages built atop of imperative languages naturally adopted the imperative paradigm as well. Java, Python, and C++ for example which were all written in C, followed this progression, thus establishing the imperative family as the dominant paradigm in programming language design.

### The STATE

The existence of an explicit state is the foundation of imperative programming. The **state** of a program or a process on a given instance is the snapshot of its immediate relevant environment and context. The state of your CPU on a given instance for example will refer to the values found in the registers and relevant memory. On a specific process the state will refer to the values inside the memory addresses it resides in.

On a computer program the state can refer the conceptual set of variable values related to the program’s runtime on some given instance. Lets use this program as an example:

int x = 3  
int y = 4  
x = x + y

At the start of runtime, the state of this program would be (*for all intents and purposes*) empty, since there are no relevant variables declared at this point. After executing the first line of code, the state of the program would look something like this:

|  |  |
| --- | --- |
| variable | value |
| x | 3 |

One integer variable named x with the value 3. After the next line of code a new variable is is introduced and immediately assigned with the value 4 so the state of the program at this instance will look like this:

|  |  |
| --- | --- |
| variable | value |
| x | 3 |
| y | 4 |

And at the last line, the value of x is updated by adding the value of y so the final state of this program will look like this:

|  |  |
| --- | --- |
| variable | value |
| x | 7 |
| y | 4 |

## Assignment Statement

Another important construct of the imperative programming paradigm is the assignment statement. Assignment statements and the concept of state are very related to each other.

Assignment statements allow your program to MUTATE the values of your variables. Mutation in the context of programming is a fancy term that basically means change. And as we learned earlier, changes to the context of a program, which includes variables, creates states.

Therefore every assignment statement, corresponds to new states of a for the program.

Assignment statements are usually executed through the use of the “=” operator (some languages like Pascal use “:=” instead). Although it borrows the equality operator from math, assignment operators behave very differently from an equality statement. Instead of communicating some kind proposition, the assignment statement has an **imperative mood**. An equality a=b in math **declares** that some a is b, while an assignment operator a=b **commands** that a’s value is now the same as b. Mutation is introduced once you perform an assignment to a again, signifying a **change** in the value of a.

By the way, the closest corresponding mathematical construct to an assignment statement is the let statement. A statement in math such as “let x be equal to 3”, has an imperative mood. But unlike an assignment statement which can change the value of a variable any number of times, a let statement can only set the value of a variable once.

For every assignment statement you feed to the computer, something meaningful happens. That particular “something” that happens is characterized by changes to your programs context. The context of a program changes for every individual mutation of a variable. And you can compare the difference between the before and after of a specific assignment by comparing the before-assignment state and the after-assignment state. The progression from one state to another characterizes the effect of an assignment.

This is the important take away that you need to remember. Imperative programming is characterized by imperative statements. Statements that tell the computer what to do. The most important type of these statements is the assignment statement. An assignment statements effect to your computer is characterized by the progression from one state to another. Assignment statements make states, and if you combine many of these assignment statements arranged in a particular manner, you can create a meaningful program that does something for you.

## Structured Program Theorem

Creating meaningful programs in imperative programming is done by applying the Bohm Jacopini Theorem. This theorem was one of the theoretical frameworks proposed to characterize imperative programming.

The theorem describes a formalism of a class called control flow graphs which are capable of representing any computable function. These control flow graphs are actually something you are intimately familar of. It is known to you as the trusty old flow chart. Any control flow graph can be created by combining subprograms in three specific ways. A subprogram is a recursive unit of control flow graphs. A subprogram can be a single statement or it can be a combination of more than one subprogram. Here are the three ways to combine subprograms:

1. Executing one subprogram, and then another subprogram (sequence)
2. Executing one of two subprograms according to the value of a boolean (selection)
3. Repeatedly executing a subprogram as long as a boolean expression is true (iteration)

Structured programming enjoyed a universal popularity in computer science. The constructs described by this formalism became the natural architecture for programming language designers. This is the reason why CS students like you are introduced to programming using control flow graphs or flow charts. This is also the reason why programming languages like Pascal, C, Java and their derivatives are designed the way they are.

## Subparadigms under the Imperative family

### Procedural programming

Programming languages like Fortran, ALGOL, BASIC, and C fall under the procedural paradigm. Languages under this paradigm simplify a complex system by subdividing a program into different **procedures** or functions.

### Object oriented programming

Object oriented programming focuses on modelling a system based on the real world ontology of objects. It uses an expressive type system to program the interactions within a system.

## Optional Readings

Rapaport W. (2004) [Great Idea III: The Boehm-Jacopini Theorem and Structured Programming](https://cse.buffalo.edu/~rapaport/111F04/greatidea3.html). [CSE 111, Fall 2004](http://www.cse.buffalo.edu/~rapaport/111F04.html) Accessed August 31, 2020 # Functional Programming Paradigm

## Introduction

During the 1930’s a mathematician investigating the foundation of mathematics, named Alonzo Church, introduced a formal system of expressing computational logic. The system he created was called **Lambda Calculus**. It was until the 1960’s when the system found its way through different disciplines. It became something more than a a mathematical formalism and became an important concept in linguistics and **computer science**[[4]](#footnote-49).

## Learning Outcomes

After this discussion you will be able to:

1. Reduce lambda calculus expressions (Optional)
2. Create higher order functions
3. Identify pure functions
4. Explain how the advantages of statelessness in functional programming paradigm
5. Explain the disadvantages of statelessness functional programming paradigm

## Lambda Calculus

Before we dive into functional programming let’s introduce ourselves to the formalism that inspired it, Lambda Calculus. These concepts may seem strange at first since it imagines a mathematical foundation beyond numbers, sets, and logic. You can skip this video and it won’t really affect your understanding of functional programming concepts. But I think understanding the concepts of this formalism will give us a better appreciation for functional programming.

### Expressions in the Lambda Calculus Formalism

Let be the set of expressions under the Lambda calculus formalism

1. **Variables**. If x is a variable, then
2. **Abstractions**. If is a variable and , then .
3. **Applications**. If , then .

Take a look at these important precedence conventions. You might get confused if you read some lambda calculus expressions. Some people often omit parentheses or single-parametrizations to write shorter expressions:

1. Application is left associative
2. Consecutive abstractions can be uncurried
3. The body of an abstraction extends to the right

### Reductions

Reductions are a ways to simplify and evaluate lambda expressions. You’ll learn later that these reductions are basically concepts that are eventually adapted to functional programming concepts.

#### equivalence:

equivalence states that any bound variable, has no inherent meaning and can be replaced by another variable:

Given a lambda calculus abstraction , this abstraction’s bound variable is . The bound variable may appear somewhere in , the body of the abstraction. An alpha equivalence basically shows that the name of the variable has no inherent meaning. Therefore, you can replace it with any other variable name.

#### Reductions

reductions state how to simplify abstractions. This process is similar to applying a function in the context of programming. For example we use the identity function () and apply it to some free variable .

When you beta reduce some application , what you’re doing is replacing all instances of the bound variable in with . When, Here’s a another example,

#### reductions

reductions describe equivalencies that arise because of free variables. If is a variable and does not appear free in then:

The lambda expression here is just some redundant abstraction. These reductions characterize higher level simplifications that are not always as obvious as the other reductions.

#### Reduction example

For example to reduce the following lambda expression, we must first understand what it means.

In the outermost level, the expression is the application of to itself. It follows the second type of lambda calculus expression discussed earlier, where and . In this context and also .

When you start evaluating this expression, you might be tempted to automatically apply a reduction by itself:

But this reduction is actually incorrect because the although and appear on both lambda expressions, these variables don’t have the same meaning . The and variables inside the left lambda expression are **bound** inside this lambda expression. The and variables outside the left lambda expression (inside the right lambda expression) are **free** in its context, therefore, even though they look the same, it is incorrect to interchange the two variables.

Two avoid confusion with similarly named variables, it is advisable to apply an equivalency, to give them different names. This can be done by replacing the right abstractions bound variables with and . Again, this alpha reduction doesn’t change the meaning of the abstraction, it merely renames the bound variables.

The correct reduction now is as follows. Still a reduction but without the ambiguity of similar variable names.

## The Paradigm

### Reimagined functions

Lambda calculus evolved from a system of logic foundation with deep roots to computation theory into something that became a basis for programming language design. Language designers started to consider the unconventional representation of lambda calculus expression as a valid and pragmatic way of representing data. Around 1950’s programming languages patterned around the framework of lambda calculus started to emerge. One of the earliest and most important of these languages was **Lisp** which evolved to become a large family of programming languages[[5]](#footnote-60). Soon, more programming languages started to implement the same formalisms described by lambda calculus. This opened a new paradigm of programming languages called **functional programming paradigm**. To explore this paradigm this section will introduce the programming language **Haskell**. This language has become one of the most important functional programming language, setting the standards for other languages paradigm.

#### How functions are treated Differently

One of the biggest difference between your classic imperative programming languages like C and Java and a functional programming language, is how it treat its function. You might have probably guessed that since the paradigm has “function” in its name. To explore this contrast lets start by introducing a simple function, written in C. This function basically accepts an integer and returns a the same integer but squared:

int square(int x){  
 return x\*x;  
}

Functions like these are patterned from mathematical functions. It has a **name** to invoke it later called square, it has **specifications** on which type of data it accepts (int x) and produces (int), and finally it has a instructions on what must be done when it is invoked (return x\*x.) Functional programming functions behave in more or less the same way.

square :: Int -> Int  
square x = x \* x

It looks different but all of the parts you can find in a C function can also be found on this Haskell function.

In terms of invocation, they are also used similarly and of course the behave similarly:

square(5);

square 5

Although functions in non-functional programming behave and look similar to functions in functional programming language, they have a huge difference in the way the programming language treats it.

A function in C is treated differently from other types of data. In fact C programmers will rarely call a function a value. What this means is that canonical value types like integers, characters, and arrays (even compound value like structs and objects) can be passed on functions and can be returned as functions.

int\* add\_to\_array(int arr\*,int x,int size){  
 for(int i=0;i<size;i++){  
 arr[i]+x;  
 }  
 return arr;  
}

C discriminates function from these canonical value types. Therefore, during runtime, non-functional programming languages interpret the expression square(5) as “the number squared” while the expression square is just some disembodied function name (*square of which number?*). Imperative programming functions during runtime are meaningless unless they are directly invoked.

#### Higher Order Functions

##### Passing functions

Functional programming languages treat functions the same way it treats values, you can pass them in other functions and you can return them as well.

s:: Int -> Int  
func x = x + 1  
  
p::Int -> Int  
func x = x - 1  
  
applytwice:: (Int-> Int) -> Int -> Int  
applytwice f x = f (f x)

Note these arrow looking characters are actually just dash followed by a greater than character “(->)”. My markdown editor formats it to look like a neat arrow inside code fences.

The code above shows two function definitions (with some type signature annotations for readability). The first is the function s::Int -> Int which is applied to an integer and produces an integer. What it does is it simply adds one to x. The second function is similar but what it does is subtracting one from x.

Type signature annotations are not required here, that’s why they’re called annotations. Adding these annotations will restrict the type the functions can be applied to. Type signature annotation syntax are understood like this

f :: Paramtype -> AnotherParamtype -> ... -> OuputType

The last type in the -> series is the type the function produces (like it’s return type) and everything else before it are parameter types.

The third function is what we call a higher order function. A higher order function is a function that either accepts a function as a parameter or returns a function parameter or both. The function applytwice as described by it’s type signature, is applied to a function f and an integer xand produces an integer. What it does is it applies the function f twice to x, something like ().

By defining a function like this we can do something like this during runtime:

ghci> applytwice s 3  
5  
ghci> applytwice p 3  
1

To a programmer with no experience with functional programming, this feature can be surprising especially since mathematical functions in algebra or calculus don’t even explore this capability. But if you remember this is not just some arbitrary added feature added for novelty. This feature isdirectly patterned from lambda calculus:

In lambda calculus an abstraction and an application does not restrict anyone from the type of expressions bound to variables. In the spirit of implementing lambda calculus, any functional programming language will allow you to do this as well.

##### Returning functions

On the other side of the coin, a function, in functional programming will also let you return functions the same way you return any other kind of data.

To explore this, suppose we have different functions that when applied to an integer, produces that integer plus a certain integer.

addTwo::Int -> Int  
addTwo x = x + 2  
  
addThree::Int -> Int  
addThree x = x + 3  
  
addFour:: Int -> Int  
addFour x = x + 4

We can generalize these functions into a *function-maker* function, that when applied to an arbitrary integer x, will produce an addx which is a function (not a number) that adds x to your integer.

addMaker::Int -> (Int->Int)  
addMaker x = (\y -> x + y)

We are introducing new syntax here, but this new syntax is a representation of an expression we already know from lambda calculus. The definition for your addMaker (\y -> x + y) is basically an implementation of the lambda expression below. y is the bound variable, and the operator -> separates the inputs and the output, x + y.

In fact, the reason why Haskell syntax uses the \ character to represent lambda expressions is because this is the your keyboard’s best physical approximation of the Greek letter . Extra note: “” does not exist in the universe of lambda calculus so instead what’s used here is a reference to a lambda calculus abstraction called “”. The definition of this can be found in the supplemental topic [Lambda Calculus Encodings](www.something.com)

What this expression means then is that addMaker produces a lambda expression, which essentially behaves exactly like a function. This allows you to create functions during runtime

ghci> addSix = addMaker 6

Simply writing the expression addSix on your terminal will yield you an error, because printing addSix doesn’t really have a meaning outside the world of lambda calculus. It is a lambda expression which is *basically* a function. *How do you represent a function as a string?*

But since addSix is a lambda that behaves exactly like a function, you can apply addSix to an integer and it will give you a meaningful answer.

ghci> addSix 3  
9

In fact you can even omit the part where you bind the value returned by addMaker to a name, and instead use it directly. Here, (addMaker 7) is a lambda expression and therefore it can be applied to an integer.

ghci> (addMaker 7) 4  
11

This nifty trick right here is the reason why lambda expressions are also called **anonymous functions** since these expressions on their own don’t have a name. Lambda expressions generally appear in functional programing languages and even non-strictly functional programming languages. Lambdas can be useful if you want to create a function that will be used only once:

ghci> applytwice (\x -> x + 2) 3  
7  
ghci> (\x -> x \* x) 4  
16

Lambdas, just like any other canonical value type can be bound to identifiers. Doing this will **name** the lambda thus allowing it to behave just like any other named function.

By the way, bindings are haskell’s representation of a mathematical let statement. When you see an = operator like the following:

x = 3

This is not an assignment statement, but instead a let binding. **3** is bound to the identifier x. Similar to what happens when you say in math.

###### Closure

A higher order function like addMaker above, is not only producing the lambda inside its definition. What is actually being produced is a construct called a **closure** which is the function definition described by the lambda and the environment of the function call. The extra data, called environment, is the reason why the lambda (\y -> x + y) makes sense outside the context of addMaker. Without passing the environment the variable x would be a free variable which will yield you a compilation error.

Inside your addmaker when you evaluate addmaker 6 , the parameter 6 is bound to the variable . So the resulting lambda that is produced will behave exactly like the lambda (\y -> 6 + y).

Closures are still a direct consequence from lambda calculus’ variable binding rules. Specifically how the variables and are bound in innermost body of the lambda calculus abstraction .

###### Multiple parameters and partial application

If we look back to lambda calculus you’ll notice how abstractions are defined to be:

Here we can see that abstractions are defined to have exactly one parameter. One can argue that this is different from the how functional programming represents its own functions and lambdas since functions with multiple parameters are allowed in these languages. Actually these functions are just disguised to have multiple parameters. These functions are just several single parameter functions combined to simulate multiple parameter functions. As an example: a function add that adds two numbers may look like multiple parameter functions:

plus x y = x + y

But internally this function is equivalent to two lambda calculus abstractions, nested together to simulate multiparameterness.

plus = \x -> (\y -> x + y)

Here plus is a higher level function that accepts a single argument x and produces the lambda expression (to be perfectly correct it is returning a closure) (\y -> x + y). This expression is a direct implementation of the following lambda calculus abstraction:

Just like lambda calculus Haskell’s -> operator is right associative so you can write the same plus function as:

plus = \x -> \y -> x + y

You can even omit the first -> and the \ near y and it will mean the same lambda expression:

plus = \x y -> x + y

Which looks almost exactly similar to a relaxed lambda calculus expression with multiple parameters:

Now when you want to apply this function we write:

ghci> (plus 3) 4  
7

Which means that first we are evaluating (plus 3) which will give us a lambda expression. The lambda expression is then applied to 4 which completes the evaluation to 7. This is also a direct implementation of a lambda calculus application:

And just like lambda calculus, function applications in Haskell are also left associative so you can omit the parentheses:

ghci> plus 3 4  
7

All multiple parameter functions and lambdas in Haskell are nested single parameter abstractions in disguise, so for all intents and purposes these two definitions for plus behave in the exact same way:

plus x y = x + y

plus = \x -> (\y -> x + y)

This means that the expression(plus 3) will have the same meaning regardless of the way you define plus. The expression (plus 3) has a special name, it is called a **partial application**. When you apply a function that is supposed to accept parameters to values (where ) (what this means is that you’re supplying the function less parameters than it is expecting), instead of getting the value, you get a partial application of that function which will evaluate to a lambda expression.

The process of converting a multi parameter function or lambda to a nested single parameter lambda is called **currying**. This term is named after the mathematician Haskell Brooks Curry, which is the same Haskell, the programming language is named after.

### Functional purity and the absence of states

One of the most distinguishing aspects of functional programming and the declarative family of languages is its philosophy of statelessness. A programmer primarily exposed to mutable imperative programming languages will find that the concept of state is natural and maybe even inevitable. Functional paradigm challenges this concept and offers a much safer and mathematically intuitive philosophy. In the perspective of functional programming, there is no state, and everything is immutable.

#### Pure functions

To understand the functional programming perspective, we have to take a step away from the imperative programming definition of a function. Let’s go back to the definition of a function in mathematics.

Across several, mathematical disciplines a function means the same thing. Consider two sets and . we can define a function as the mapping between the elements of and . The elements of and can be anything, they can be numbers, which shows us how a function can be represented by a formula or a graph. The elements of and can be matrices and vectors, which defines a function as a transformation between two vector spaces. On the higher level perspective of category theory, functions are morphisms between objects of a given category. At its core a function basically defines arrows between things.

If you remember functions from discrete math, functions at its most basic form looks like the one above.

Functions in functional programming languages like Haskell are (arguably) the closest computer representation of a mathematical function. We call these functions, **pure functions**.

They differ from your standard C function because the definition inside pure functions are only instructions on how to produce a result based on the parameters. To fully understand this concept here are some examples of impure functions

int square(int x){  
 addToExternalLogger("calculating square");  
 return x\*x;  
}

The impurity in this function is the line where the function writes to some external logger, addToExternalLogger("calculating square");. A function can only be pure if the result of the function can be fully determined by its parameter. The only parameter here is x. Invoking this square function with the same parameter value does not do the same thing. The effect of changing the logger is dependent on the previous state of the external logger. The effect on the logger is what we call a **side-effect** of the square function. It is a side effect since this line of code modifies values outside boundaries of the function.

int\* increaseArray(int \*a, int size){  
 for(int i = 0; i < size; i++)  
 a[i]+1;  
}

A pass by address/reference function which changes the value of a parameter will automatically be an impure function since changing the value of a is a **mutation**, which is a side-effect.

String headsortails(){  
 if(randInt()%2==0)  
 return "heads";  
 else  
 return "tails";  
}

This function is also impure because the return value is not dependent on the parameters alone. The return value will be dependent on the randomization seed which is something outside the parameters of the function.

A pure function must satisfy these two:

1. A pure function has no side-effects
2. A pure functions output must be dependent on the inputs alone

In fact if and where is not a function at all

A good way to test if a function is pure is if you can can (theoretically) create an infinitely long lookup table such that looking up the value for a specific input is perfectly identical to calling the function with the same input. And if you think about it this is the essence of a function. It is a list of associations between the domain and the range.

#### The Absence of mutation

One of the hallmarks that make imperative programming imperative is the assignment statement. It enables the program to advance to a new state. Purely functional programming languages like Haskell, lacks the mechanism to mutate anything. Using the “=” operator (which signals an assignment statement in imperative languages) in functional languages binds the value on the right hand side to the left hand side. This mechanism is conceptually different from an assignment operation in C. For example:

int x = 0;  
x = 1;

This code in C starts with a combined declaration and assignment: int x = 0 . The second line then, **reassigns** the same variable x to the new value 1. These lines of code corresponds to a mutation on the variable x, (from 0 to 1). The value of the variable x is not definite since it can change within the runtime of the program. Because of the existence of mutation, the values of variables are dependent on the current **state** of the program.

Replicating this C code in Haskell is in fact not allowed:

x = 0  
x = 1

It will give an error message upon compilation:

main.hs:2:1: error:  
 Multiple declarations of ‘x’  
 Declared at: main.hs:1:1  
 main.hs:2:1

In Haskell, any “=” is a declaration of a binding. In fact all declarations in haskell are required to be **bound** to a value. These bindings are final (*in the scope of the identifier*). It is even wrong to call x here a variable since its value does not vary. The correct way to call x is identifier, since it merely identifies the value bound to it.

#### Consequences of Statelessness and Immutability

Although Haskell’s functions are our best representation of a mathematical function, this does not mean that it is pragmatically better than the classic impure functions of C. Restricting a programming language against side-effects seems like an artificial disadvantage introduced only to faithfully implement mathematical functions.

There are several reasons why programming without state is a better way of programming. Eliminating all side-effects is demonstrably safer against accidental errors. Building a library of functions perfectly working without worrying about side-effects makes the system easier to understand and more resilient to changes.

void f(int \*x, int y){  
 \*x = \*x + y;  
 printf("%d\n",\*x);  
 return;  
}  
int main(void) {  
 int x = 0;  
 f(&x, 3);  
 x = x - 2;  
 f(&x, 3);  
}

The exact behavior of the function f depends on where you call it. Even if you use the same parameters, you’re not guaranteed to get the same results.

On small chunks of code, managing the consequences of having states such as global variables, will be trivial since you can reasonably track which variables are global (*or external in general*) and which functions interact with the global variables. (Even this tiny amount of code, the function’s effects and side effects are not very obvious).

As the system grows, using functions and variables without double checking for side effects becomes much harder. As a consequence the whole system becomes a nightmare of impure functions on top of impure functions which may unexpectedly affect other parts of the system. On a corporate setting where multiple people are working on the same system, refactoring becomes unsafe without knowledge of all the side-effects of the functions in use. On systems with shared resources and multi-threading it becomes extra-extra difficult to keep track of things without proper documentation.

Don’t get this wrong, though. Even with all these disadvantages in the state-full mutable paradigm of imperative programming, one can still code robust and harmonious systems. You just have to be extra careful writing your code with discipline, only using globals and side-effects when it is safe and necessary (*this is in fact the reason why writing smaller pure functions is considered good practice in any paradigm*). After all, being able to code with states can be thought of as an extra feature. You can be a C programmer and just treat all your variables as immutable and all of your functions as pure. (*calling variables, immutable is kinda oxymoronic because variables are meant to change, but whatever*)

Functional programming has its own set of disadvantages as well, most of them related to this seemingly artificial crutch of statelessness and immutability. The most obvious one is that creating new values instead of changing an existing variable has extra overhead in both processing and memory, making functional programming slower and less efficient. Programming without state can be difficult to do for certain mechanisms (*Like the external logger for example*). Simulating mechanisms like this may introduce conflict on how you compose your functions. It’s not impossible though, you just have to learn some category theory concepts such as **monads**. Also, for most people who are used imperative programming, recursion, does not feel natural compared to iteration. But in my opinion, after being exposed to functional programming for some time, recursion makes much more sense than iteration.

Nevertheless, these disadvantages are not insurmountable. In fact, there are a plenty of systems written in functional programming languages running without issues. Functional programming has had the reputation of being more conceptual and fancy than the classic imperative programming language. People mocked Haskell for its pristine white tower “elitist” approach to programming. But recently these functional programming languages like Haskell, F# and Scala has enjoyed improvements that has elevated it to be as pragmatic as your classic C, C++ or Java. In fact, functional programming has gained a considerable rise in popularity to an extent that mainstream programming languages with imperative roots like C#, Python, and JavaScript have started to introduced features patterned from pure functional programming languages. (Features such as higher-order functions and lambdas). With the risk of sounding editorial I even argue that learning functional programming concepts has become a necessity for any programmer, regardless of his/her paradigm preferences.

## Optional Readings

Lambda Calculus Encodings

# Lambda Calculus Encoding (Optional Reading)

One of the most interesting things that the Lambda Calculus System demonstrates is it’s use in computability theory. It’s importance in computing has led to the formulation of the Church-Turing Thesis which conjectures that the Lambda Calculus System and the hypothetical Turing machine rules as complete representations of any algorithm. The lambda calculus system is said to be **Turing Complete**, which means that the system can simulate any concievable Turing machine. To demonstrate its turing completeness this topic will show some of the encodings in lambda calculus.

### Boolean Values

A good starting point for encoding is the smallest unit of data, a boolean value. Boolean values, such as true and false can be represented by the following lambda expressions:

* **True:**
* **False:**

The way boolean values are encoded in lambda calculus using an abstraction that when applied to two values, produces first value for true and produces the second value for false. To demostrate the consistency of this encoding we can use this encoding of an if\_then\_else function. An if\_then\_else function in the context of any familiar programming language looks like this:

if(condition)  
 this  
else  
 that

The function consumes three expressions (condition,this, and that) If the condition is true, then the function produces this, otherwise the function produces that. This function can be represented in lambda calculus as the following function:

: condition, : this , : that

Applying this function shows how lambda calculus boolean encodings work. The example below shows what happens when the condition is applied to a true condition

In the first line a sneaky equivalency is done on the True encoding.

The example below in the other hand is the same if\_then\_else function applied to a false condition

In the effort of saving lines reductions with currying are applied in one go.

By representing the if\_then\_else function in lambda calculus we are also able to encode all the logic gates by reusing our if\_then\_else function.

#### not function

The not function can be represented using theif\_then\_else function this way:

if(condition)  
 False  
else  
 True

In lambda calculus:

#### or Function

The or function can be represented using theif\_then\_else function this way:

if(left)  
 True  
else  
 right

In lambda calculus:

#### and Function

The or function can be represented using theif\_then\_else function this way:

if(left)  
 right  
else  
 false

In lambda calculus:

### Natural Numbers

Natural numbers are encoded in lambda calculus similar to how numbers are described using *Peano’s Axioms*. By defining and defining a successor function. This encoding is called Church numerals. The first natural number is defined as the lambda expression:

Which is equivalent to the encoding for boolean false.

All other natural numbers are derived using a special function called the successor function, such that the succesor of any natural number is . The successor function in lambda calculus is represented by:

Using this function we can derive the encoding for the nfireatural number :

To derive this encoding for two we simply find .

Continuing this process and generalizing successions will lead us the following encoding of natural numbers:

|  |  |
| --- | --- |
| Church Numeral | Lambda Calculus Encoding |
|  | $ s. z .z$ |
|  |  |
|  |  |
|  |  |
|  |  |

is a shorthand notation for applications of to or .

Another useful definition of a successor function would be:

#### Addition and Multiplication

The successor function will help us derive the lambda calculus representation of an addition function. In the same sense that addition is just a repetition of increments, we can implement addition by making use of repetitive applications of the successor function.

To test the consistency of this function, we can try adding two arbitrary Church numerals , and .

Multiplication works in a similar manner. To solve for the product of two Church numerals and , we simply add to repetitively, number of times. The function will look similar to a , but with a partial application of instead of the successor function .

# Haskell Cheat Sheet

## Setting up Haskell

To start writing Haskell code, install Haskell through stack. Stack is found in the folder called “Haskell/Stack” inside the provided course pack. Copy the Stack folder and place it in your computer. To be able to use stack anywhere, add your copy of the stack folder in the PATH variable of your computer.

Once stack has been set-up using the steps above, you can run the GHC repl using the command

> stack ghci

The first time you run this code, stack will automatically install the GHC compiler.

After downloading GHC, you will be taken to the Prelude part of the your GHC repl. To test if everything is working properly, try the following Haskell expression:

Prelude> show (1 + 3)

If everything is good to go, the GHC expression will evaluate to:

"4"

To exit GHC run the following GHC command

:quit

To load a Haskell program, enter the GHC repl first

> stack ghci

While inside Prelude, use the command :load <Path to haskell file>. For example

Prelude> :load "Trying Things.hs"

If the path to the haskell file contains spaces, you need to enclose the path in braces

## Type annotations

**Pattern**

The last type in the arrow series is the range type or return type. Every type before it are the domain types or the types of the parameters in order

function\_name :: Type\_of\_Param1 -> Type\_of\_Param1 -> ... -> Type\_of\_Paramn -> ReturnType

**Examples**

Double of an integer,

double :: Int -> Int

Sum of the length of two strings (strings in Haskell are arrays of Char, an array of specific types are written enclosed in square brackets, [Type] represents an array of Types)

len :: [Char] -> [Char] -> Int

Check if some integer is even (boolean values are written capitalized, True and False, )

isEven :: Int -> Bool

Accept two (Int -> Int) functions and produce the composition of those functions (given functions and , )

compose :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)

## Function definitions

**Pattern**

Every identifier placed in between the function name and = are the parameter names. The expression to the right of = is the expression evaluated when the function is called.

function\_name param1 param2 ... paramn = <some expression>

**Example**

Double function

double x = x + x

isEven function

isEven n = n % 1

## If-then-else expression

**Pattern**

One of haskell’s condition expressions are if-then-else expression. Because a haskell expression is required to evaluate to something, unlike C, all if parts must be followed by a then part and else part.

if <bool-exp> then <exp1> else <exp2>

The expression inside the if clause must be an expression that evaluates into a boolean value. If the expression <bool-exp> evaluates to True, then the whole if-then-else expression evaluates to whatever <exp1> evaluates to. If <bool-exp> evaluates to False, then the whole if-then-else expression evaluates to whatever <exp2> evaluates to. The then clause and else clause cannot be empty and they must evaluate to the same type

Haskell boolean literals start with uppercase letters, True and False.

**Examples**

The expression:

if (2 > 1) then 5 else 4

evaluates to 5

The expression:

8 \* (if (3 <= 2) then 2 else 3)

evaluates to 24

If-then-else statements can be nested by writing if statements inside the then part and else part

if (2 > 1) then (if (0 == 1) then 2 else (1 + 2)) else (if (2 == 2) then 5 else (if (3 == 2) then 6 else 7))

evaluates to 3.

If else statements can be written neatly with tabs and newlines like this:

f :: Int -> Int  
f x = if (x > 1)   
 then (if (x == 1)   
 then 2   
 else (1 + 2))   
 else (if (x == 2)   
 then 5   
 else (if (3 == x)   
 then 6   
 else 7))

## Lambdas

**Pattern**

\param1 param2 ... paramn -> <some expression>

All identifiers between \ and -> are the parameters of the lambda. The expression to the right of -> evaluates when the lambda is applied.

**Example**

A lambda that doubles a number

\x -> x + x

A lambda that adds two numbers

\a b -> a + b

The same function but written in its verbose uncurried form

\a -> (\b -> a + b)

## Let Binding

To bind a value to some identifier use the = operator

**Pattern**

identifer = <some expression>

The expression to the left of the = operator is evaluated and then bound to the identifier to the right of the = identifier.

**Example**

the integer 3 bound to x

x = 3

a lambda bound to f

f = \x -> x + x

## Function/Lambda Application

**Pattern**

Two expression separated by a space is a function application. The expression on the left side must evaluate to a function or a lambda. This function/lambda is applied to the right expression as its parameter

<expression1> <expression2>

A series of expressions are curried multiparameter applications. The leftmost expression must evaluate to a function or a lambda. This function/lambda is applied to the right expressions as its parameters.

<expression1> <expression2> <expression3> ... <expressionn>

**Example**

double 3

evaluates to 6

compose addThree double 2

evaluates to 7 # Logic Programming Paradigm

## Introduction

Just as functional programming paradigm is patterned from the formalisms of lambda calculus, logic programming is patterned from predicate calculus. Computer Scientists usually describe families of programming languages under the logic paradigm as a sub-paradigm of declarative programming (*declarative programming being any paradigm that is not imperative*). In terms of application this paradigm is more closely related to knowledge base programming languages like SQL. While SQL uses relations (not tables) to represent knowledge, logic programming uses rules of logic and predicate calculus to represent knowledge.

## Learning Outcomes

1. Create prolog facts, rules, and queries
2. Explain the process of unification
3. Explain how proof search is used to respond to queries
4. Create recursive prolog rules

For this secton we will use the programming language Prolog as the representative of logic paradigm. Other logic programming families are answer set programming, ABYSS and Datalog.

### Facts Rules and Queries

#### Facts

There are three basic constructs in Prolog, facts, rules and queries. A knowledge base is a collection of facts and rules in the same way a c library or a python package is a collection of function definitions. Prolog programs are basically knowledge bases. Here’s an example of a knowledge base:

firetype(charmander).  
firetype(charizard).  
watertype(squirtle).  
flyingtype(charizard).

Each line of code you can in this particular knowledge base is a fact. Just like facts, in logic, facts in Prolog are propositions that are known to be true. This means that your program knows four things. One of those are:

firetype(charmander).

In Prolog firetype(X) represents the mathematical predicate you’ve learned in discrete math, . And just like predicates, this is attached to the meaning X is firetype.

Therefore the fact firetype(charmander) represents the proposition, “charmander is firetype”

So, to summarize, the fact firetype(charmander) is basically a representation of the proposition, $firetype("charmander")$ where is a predicate and $charmander $ is a value assigned to the predicate. Any fact that can be found on the knowledge base are basically true propositions and any proposition that is not in the knowledge base (and cannot be inferred from the knowledge base) are false.

#### Queries

A knowledge base is used by writing queries to prolog. You can probably guess what this prolog construct means just by looking at its name. Queries represent questions you ask prolog. The answer to a question depends on what prolog knows. And what prolog knows is represented by the knowledge base. For example if you load the knowledge base we created earlier. We can ask prolog the following question below. Writing the following will basically ask prolog, “hey, is it true that charmander is firetype?”

?- firetype(charmander).

Based on the knowledge base loaded earlier prolog knows that this proposition is indeed true. Therefore it responds with:

yes

If prolog is asked with the query

?- firetype(squirtle).

Prolog checks its knowledge base again. Realizing that none of the facts match this proposition, prolog responds with:

no

Therefore, if you provide prolog with statements it has never seen before like the following:

?- grasstype(pigeot)

Prolog will interpret this as false statement, responding appropriately with:

no

#### Rules

Aside from facts, you can also define rules in your knowledge base. To illustrate this, lets add rules to our knowledge base.

firetype(charmander).  
firetype(charizard).  
watertype(squirtle).  
flyingtype(charizard).  
  
resistanttofire(squirtle) :- watertype(squirtle).

A rule is basically an implication statement. A rule written as c :- h is equivalent to the implication statement . Prolog rules are written using the “c if h”, the reverse of a conventional “if-then” implication "statement. A lot of people get confused here so just remember, :- is read as if. Therefore, the left half of a rule is the conclusion and the right half is the hypothesis.

Since we’ve written that rule we can ask prolog the following:

?- resistanttofire(squirtle)

Prolog responds with

yes

Although resistanttofire(squirtle) is not written as a fact it can be inferred from the rule resistanttofire(squirtle) :- watertype(squirtle) and the fact watertype(squirtle). Therefore it is true via modus ponens:

#### Variables

Another important thing about Prolog constructs is that you can write them with variables. For example, writing the query:

?- firetype(X)

Basically asks the question, which values when substituted to X in the predicate firetype(X) will yield true statements? This can be interpreted in natural language as “which Pokémon are fire type?” Therefore, this query will yield the response:

X = charmander  
X = charizard

Variables inside facts and rules allows the creation of richer knowledge bases. Instead of the rule resistanttofire(squirtle) :- watertype(squirtle). we can write a more general rule using variables:

firetype(charmander).  
firetype(charizard).  
watertype(squirtle).  
flyingtype(charizard).  
  
isresistantto(X,Y) :- watertype(X),firetype(Y).  
isresistantto(X,Y) :- watertype(X),watertype(Y).

This introduces a more complicated rule isresistanto(X,Y) :- watertype(X),firetype(Y). This rule’s premise is a conjunction of predicates watertype(X) and firetype(Y).

If we imagine that the predicate, means “x is resistant to y”, the whole rule can be interpreted as

for all pairs of X and Y, X is resistant to Y, if X is water type and Y is fire type,

This statement, can be written as the following quantification statement:

By writing this rule, prolog can infer the following facts:

?- isresistantto(squirtle,charmander)

yes

?- isresistantto(squirtle,charizard)

yes

?- isresistantto(squirtle,squirtle)

yes

If you ask prolog a harder question like the following:

?- isresistantto(squirtle,X)

Prolog interprets this as "which values of X make the proposition: squirtle is resistant to X, true? Therefore, Prolog will look for the pokemon, squirtle is resistant to, therefore you with the output:

X = charmander  
X = charizard  
X = squirtle

#### Prolog Syntax

There are three types of prolog terms [[6]](#footnote-96). By composing these terms you can express rich knowledge bases.

1. Constants. These can either be atoms (known to us as strings such as squirtle ) or numbers (such as 24).
2. Variables. (Those that start with an underscore or any uppercase letter such as as X, Z3,\_4310, and List.)
3. Complex terms. These have the form: functor(term\_1,...,term\_n). We’ve seen examples of these in predicates and queries such as firetype(charmander) and isresistantto(X,Y)

### Unification

The way prolog is able to respond to complex queries such as:

?- firetype(X)  
X = charmander  
X = charizard

is through the use of the logical concept known as **unification**. Unification algorithmically identify logical substitutions in symbolic expressions such as prolog facts, queries and rules. Unification is defined in prolog as the following:

Two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.

This definition gives us the unification of trivial cases such as the unification of constants squirtle and squirtle. Prolog also unifies the complex terms watertype(squirtle) and watertype(squirtle) and the variables X and X. The complex terms watertype(squirtle) and watertype(blastoise) will not unite.

Prolog also unifies the variable X with the constant squirtle. Although they are not the same, the variable X can be uniformly instantiated to squirtle (i.e. X = squirtle). What this specific unification case means is that, you can find some binding of the constant squirtle to the variable X without breaking other unification rules. This successful binding means that these values indeed unify. By the same intuition, watertype(X) and watertype(squirtle) will also unify buy instantiating (X=squirtle).

On the other hand the complex terms isresistantto(X,charmander) and isresistantto(squirtle,X) does not unify since you cannot find an instantiation of X that makes them equal. The instantiation X=squirtle evaluates to the terms isresistantto(squirtle,charmander) and isresistantto(squirtle,squirtle). On the other hand, the instantiation X=charmander makes the terms isresistantto(charmander,charmander) and isresistantto(squirtle,charmander). Both of these scenarios break because it forces the incorrect unification of the constant squirtle and charmander.

The process of unification can be summarized by the following[[7]](#footnote-98):

Two terms and unify if and only if

1. and are constants and they are the same number or atom
2. is a variable and is any type of term (in this case is instantiated to ) or is a variable and is any type of term (in this case is instantiated to ). This rule automatically unify any pair of variables
3. and are complex terms and:
   1. They have the same functors and the same number of arguments
   2. all their corresponding arguments unify
   3. the variable instatiations are uniform or compatible (you cannot instantiate to some constant when unifying a pair and instantiate to another constant when unifying another pair of arguments)

You can demonstrate unification in the prolog terminal using the predicate =/2 (this means the = functor with two arguments).

?- =(squirtle,squirtle)  
yes

?- =(squirtle,charmander)  
no

?- =(squirtle,X)  
X=squirtle  
yes

?- =(X,Y)  
X=\_5071  
Y=\_5071  
yes

The instantiations X=\_5071 and Y=\_5071 asserts that both X and Y share the same variables in this case. This is also known as X and Y being aliased, meaning that they share each others instantiations

?- =(watertype(X),watertype(squirtle))  
X=squirtle  
yes

?- =(f(g(X),X),f(Y,a))  
X=a  
Y=g(X)  
yes

#### Programming with unification

Unification is crucial with how one can write interesting logic programs. By creating knowledge bases that take advantage of unification, you can generalize structures based on the facts and rules of its characteristics. For example, the following is a knowledge base describing the characteristics of vertical and horizontal lines:

vertical(line(point(X,Y),point(X,Z))).  
horizontal(line(point(X,Y),point(Z,Y))).

Instances of line that unify with these predicates, are also instances of vertical and horizontal line. Therefore asking the query:

?- vertical(line(point(1,2),point(1,3)))

Will yield the response:

yes

It is indeed a vertical line. And the knowledge base makes sense, since any line that has the same x coordinate is vertical and any line that has the same y coordinate is horizontal.

We can even ask more general queries to haskell such as:

?- horizontal(line(point(2,3),point(Y,4)))

Which basically asks prolog for horizontal lines starting at and ends at a point with as the -coordinate. Since prolog can’t unify this query with any value for (horizontal lines must have the same -coordinate), prolog responds:

no

### Proof Search

Here we will discuss the process called proof search. This is the algorithm that prolog uses to check for unifications and answer queries. Let’s start with an example:

f(a).  
f(b).  
  
g(a).  
g(b).  
  
h(b).  
  
k(X) :- f(X), g(X), h(X).

Asking prolog the query,

?- k(Y)

This gives prolog a **goal**, unifying k(Y) with all possible known facts or inferable facts in the knowledge base.

Whenever prolog unifies queries containing variables, it creates a new internal variable to alias it with Y. So in the perspective of prolog, the new goal is now:

?- k(\_G34)

This query has the exact same meaning as k(Y), since the variable names have no inherent meanings anyway. Prolog does this to create a goal containing variables guaranteed to be unused anywhere else. This removes any ambiguity with other variables named Y (no other variable is named Y but prolog still does this anyway).

Prolog goes through the whole knowledge base from top to bottom and from left to right, attempting to unify the current goal, k(\_G34) to a fact or a head of a rule. In this case since there are no facts k(\_G34) can unify with, it unifies with the head (or the conclusion) of a rule, k(X) :- f(X), g(X), h(X).

Since there are no other facts or rule heads that can be unified with the goal k(\_G34) prolog’s proof search doesn’t branch out.

Since this is a rule, we can prove that k(\_G34) is true by proving the premises, f(\_G34), g(\_G34), h(\_G34). This gives prolog three new goals, proving the conjunction of the predicates, f(\_G34), g(\_G34), h(\_G34)

By unifying k(\_G34) and k(X), the variables \_G34 and X are now aliased therefore they now share the same instantiations

Prolog unifies the three goals one by one, left to right. So, starting with the goal f(\_G34), prolog searches the whole knowledge base again, attempting to unify f(\_G34) with facts or rule heads. Since the knowledge base contains both facts f(a) and f(b) there will now be two paths in this search, instantiating \_G34 to a and instantiating \_G34 to b.

##### Path 1: \_G34 = a

From this instantiation, prolog is now left with the new goals g(a), h(a). Starting with g(a), prolog searches the knowledge base again and unifies g(a) to g(a), reducing the goal toh(a). Since h(a) cannot be unified with any fact or rule head, this path ends up unprovable.

##### Path 2: \_G34 = b

From this instantiation, prolog no has the new goals g(b), h(b). Starting with g(b), prolog searches the knowledge base from the top again and finds the unification g(b) to g(b), reducing the goal to h(b). It then finds the unification, h(b) thus completing the goal and proving the query for the instantiation\_G34 = b.

By completing all possible paths prolog responds to the query:

Y = b  
yes

Since there are no other instantiations that prove the goals, Y = b is the only possible answer.

### Recursive Definitions

Similar to functional programming, logic programming represents repetition using recursion. While functional programming makes heavy use of recursive functions to implement complex behavior, logic programming languages like prolog uses recursive rules to model complex structures. For example: consider the following knowledge base:

is\_digesting(X,Y) :- just\_ate(X,Y).  
is\_digesting(X,Y) :-  
 just\_ate(X,Z),  
 is\_digesting(Z,Y).  
  
just\_ate(mosquito,blood(john)).  
just\_ate(frog,mosquito).  
just\_ate(stork,frog).

You’ll notice that the rule is\_digesting is special since one of its goals is itself. You can interpret this rule as:

is digesting if just ate or ate some that is digesting .

The or part of this implications hypothesis is represented in the knowledge base by giving the conclusion is\_digesting(X,Y) two separate hypotheses to satisfy.

Posing the query:

?- is\_digesting(stork,mosquito)

Following the process of proof search, the query is\_digesting(stork,mosquito) is unified with the line 2, giving it a new goal just\_ate(X,Z), is\_digesting(Z,Y). The goaljust\_ate(X,Z) will then match to just\_ate(stork, frog) and the 2nd goal, is\_digesting(X,Y) is then inferred from just\_ate(frog,mosquito).

#### Representing numbers using logic

Since logic calculus is a formalism for the foundation of mathematics, how do numbers emerge from predicates and propositions?

This is also another concept shared between, logic calculus and lambda calculus. You can represent numerals (specifically natural numbers) using Peano’s axioms:

0 is a numeral

the successor of 0, denoted by s(0) is also a numeral

You can represent these axioms as a knowledge base:

numeral(0).  
numeral(s(X)) :- numeral(X).

This knowledge base will then define all of the possible natural numbers out there, demonstrated by the query:

numeral(X)

X = 0  
X = s(0)  
X = s(s(0))  
X = s(s(s(0)))  
...

Using this representation, you can then define arithmetic operations such as addition and multiplication (also based on Peano’s axioms)

numeral(0).  
numeral(s(X)) :- numeral(X).  
  
add(A,0,A).  
add(A,s(B),s(C)) :- add(A,B,C).  
  
mult(\_,0,0).  
mult(A,s(B),C) :- mult(A,B,D), add(A,D,C).

### Advantages and Disadvantages of Logic Programming

Logic programming shares a lot of similarities with functional programming. It also shares its advantages and disadvantages as well. Both paradigms offer a safer and more consistent framework since they are both patterned form mathematical formalisms. Functional programming has lambda calculus and logic programming has predicate calculus.

Being non-imperative also gives them an edge of automatically being immune to the perils of state and at the same time being prone to the perils of its absence.

Logic programming’s way of expressing knowledge gives it a lot of niche uses. Most of the distributions prolog are admittedly meant for a limited use case only. The straight forward way of listing facts and rules makes it suitable for representing complex information that can be usually found in the domains of AI, NLP, and expert systems. The beauty of unification and proof search shines on these domains as they often require, complex representation involving nested rules and recursive structures.

Logic programming’s disadvantages are indeed similar to functional programming, but much worse. The obvious inefficiency due to the absence of state is much more evident in logic programming because of the thorough approach of backtracking in proof search. The strangeness of logic programming as compared to the imperative way of thinking is also much worse than functional programming (at least functional and imperative share the concept of functions Prolog only has predicates, Haskell is strange but Prolog is way stranger). Because of these logic programming is relegated to solving niche problems in various domains. Just like functional programming though, the spirit of logic programming can be found in other paradigms through the existence of unification libraries. Although logic paradigm is admittedly less relevant than other paradigms, its strange features are definitely useful and worth studying.

# Prolog Cheat Sheet

## Setting up prolog

Install the prolog compiler using the installer included in the course pack. To able to use swi-prolog anywhere, add the path to the directory of swipl executable in your PATH environment variable. It is generally found in the bin folder of the installation.

Once prolog is has been set-up you can run the swipl command to start swi-prolog

> swipl

to exit, use the halt. command (swipl will wait for a . for every query/command)

?- halt.

To load prolog knowledge bases (\*.pl), use the swipl command again but this time include the path to the prolog file as an argument.

> swipl knowledegebase.pl

## Writing Prolog knowledge bases

Knowledge bases are made of facts and rules.

### Prolog facts

Every prolog fact ends with a period. A prolog fact can be a constant, or a complex term.

#### Constant

Constants start with a lowercase letter

truth.

#### Complex Term

Complex terms follow the pattern functorName(arg1,arg2,...,arg3). Arguments can be a constants, variables, or complex terms. Here are examples

isresistantto(squirtle,X).

vertical(line(point(X,Y),point(X,Z)))

Functors are prologs representations for predicates. Whenever you see functorName/2 mentioned, it means it is a functor called “functor” that accepts two parameters.

### Rules

Rules follow the following pattern head :- tail1,tail2,...tail3. head is the conclusion of the implication statement, it can be any fact. The tails represent the hypothesis of the implication. Writing more than one tail means that the hypothesis is a conjunction of the tails. Examples:

isresistantto(X,Y) :- watertype(X),watertype(Y).

is\_digesting(X,Y) :- just\_ate(X,Y).

## Writing queries

To ask the knowledge base some questions, you load the knowledge base file using swipl and write queries in the ?- dialog inside swipl. Remember to always end the query with a period. Queries are written with the same pattern as prolog facts. For example.

?- truth.

true.

If the query has multiple lines of answer, prolog will wait for you to either press semicolon/space to show the next answer or the enter key to stop showing more answers. # Object Oriented Programming Paradigm

## Introduction

As procedural programming became more and more mainstream, computer scientists started to notice the issues behind states and side effects. Some languages offered a complete paradigm shift, completely abandoning the notion of state. This exodus formed the alternative paradigm family, declarative paradigm. Other languages however, went on a different direction. These languages offered a solution to fixing state by introducing richer features and an intuitive design. From this approach the paradigm object oriented programming was born.

## Learning Outcomes

After this discussion you should be able to

1. Explain how programmer’s started to focus on writing maintainable code
2. Differentiate between the object and the class
3. Explain OOP’s philosophy, the surface vs. the volume
4. Explain what abstraction means
5. Describe the fundamental concepts encapsulation, inheritance, and polymorphism

As time went by and as technology evolved, computer scientists noticed new issues on the code they were writing. The goal of building the most optimized machine code became less and less of a priority. Programmers had, under their disposal, faster and better hardware. As the demand for complex systems grew, the priority shifted from *code that run*s, to *code that was intuitive*. As programmers started to build bigger and bigger systems, they started to notice the issues in terms of maintainability.

This shift in focus can be seen from the extremes of procedural paradigm such as the programming language Assembly. Assembly code was not really built to be intuitive. It was built to reliably work. Assembly contained mechanisms to move data around and to control program flow. These features were enough for that time since the objective of programming was to write efficient code that bulky slow CPUs understand. Assembly programmers did not have time to worry about code elegance or intuitiveness.

The landscape started to change when hardware started becoming better and software started becoming more complex. Speed and memory wasn’t that much of an issue anymore so computer scientists’ focus shifted towards the issue of maintainability. Software became bigger and more complex and understanding other programmers’ code became more and more difficult. In fact, understanding your own code was also becoming difficult. Because of this writing code that a computer understands isn’t enough anymore, a good programmer must write code that humans understand as well. Code shifted from computer centered to human centered.

But instead of redesigning the concept of imperative paradigm to solve maintainability, object oriented programming sought to build on top of the features of imperative programming. State despite its known issues, still exists but OOP gave imperative programmers extra tools to protect code from being carelessly mutated. Because of this OOP stayed under the imperative family since the programmer is still focused on telling the computer what to do using assignment statements and mutation.

## Fundamental Concepts of OOP

Object oriented programming is usually defined using its three core design principles:

* Encapsulation
* Inheritance
* Polymorphism

It is said that any programming language that contains these mechanisms is an object oriented programming language. But calling the entirety of a programming language, OOP can be problematic since you are free to write code written in an “OOP language” without actually using these design principles. At the end of the day OOP is not merely a language classification, but a paradigm. Some programming languages are indeed intentionally written to be used for OOP design but these classifications are less important than judging if the code itself written by the programmer adheres to OOP’s design principles.

### The Object

Lets start with the star and the building blocks of object oriented programming,. What exactly is an object?

An object is a living organism in your code. To grasp the capabilities and limitations of an object we will anthropomorphize objects. Treating an object as an organism will guide you on how you use objects effectively.

Just like any creature an object has both form (attributes/fields) and behavior (methods) (this is one of the reasons why C’s struct is not an object since structs only have form). Objects are written to be representations of real world nouns such as a person or an employee or a file. The best way to design objects is to simulate the real world form and behavior of what these objects represent. Think of objects as the **representatives** of things in your code. Since you cannot shove an actual employee in your system, you instead create a representative of that employee using an employee object. If you start thinking about objects like representatives instead of mere data holders you’ll be able to design a safe and elegant object.

### The Class

Often discussed alongside an object is the class. There’s usually a lot of confusion when identifying the difference between a class and an object and it is probably because in the universe of your code, the class and the object will have the same name.

A class is the specifications for the creation of objects. If you want to give a class a more proactive role, you can think of the class as the factory that builds objects. There are a lot of analogies out there to illustrate the relationship of classes and objects. For example, a star shaped cookie cutter (class) that makes star shaped cookies (objects). If you want to make star shaped cookies you use the star shaped cookie cutters. You can also call objects as instances of a class. In fact that’s what I call objects most of the time. For example, the Cash class, $2.75 is an instance of a Cash object. Or a cat named Garfield is an instance of an class called Mammal.

If an object is a representation of a tangible real world object, then the class is the conceptual type/category of that real world object.

The following class diagrams are the representations of a book and an Employee. The attributes of the book are title, author, publishDate, and pages. These attributes also simulate the form of a real world book. This object also has methods called ISBN() and numPages(). The attributes of an employee is name, string, and salary and it has a method called reassignJob().

Class diagram of a book class

Class diagram of a book class

The following are representations of objects:

book1{  
title: "Corpus Hermeticum"  
author: "Hermes Trismegistus"  
publishDate: Dec 12, 2008  
pages: ......  
}  
  
book2{  
title: "Behold a Pale Horse"  
author: "Milton William Cooper"  
publishDate: Dec 1, 1991  
pages: ......  
}  
  
employee1{  
name: "Rubelito Abella"  
jobTitle: "Instructor 1"  
salary: [REDACTED]  
}

### The Surface and the Volume

#### Data Hiding and the Interface

Before we play around with some code, lets dive deep into the philosophy of object oriented paradigm. The base premise of OOP is the concept called **data hiding**. And this formal OOP term describes what I mean when I say that, what OOP did for imperative programming was to allow the programmer to create artificial boundaries between irrelevant data and behavior. In the eyes of an OOP design, procedural code is a mix of data and functions arbitrarily tossed in a spaghetti of mutations and side-effects.

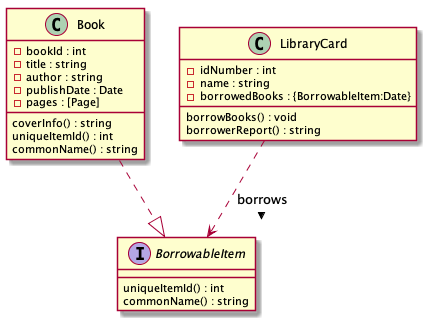
OOP’s mechanism to create boundaries in the form of objects brought structure to imperative programming. All the attributes of a Book object doesn’t need to be accessed by the methods of a LibraryCard object. That’s because in the real world, a library card doesn’t need to know what’s written in page 69 of some book. Some of the data in Book should be **hidden** from a client object LibraryCard.

Object oriented programming’s obsession to simulate real world objects is not caused by it’s dream to become an exact representation of the real world. Object oriented programming obsesses over structure and simulation because it is a necessity for human comprehension and therefore, maintainability. Systems need good structure not because well structured code is pleasant and elegant to look at, but because our feeble minds can’t process poor structure efficiently. In fact the reason why we call a well structured piece code, “elegant” is because our tiny limited minds can process it well.

Imagine code, so poorly structured, that your mind breaks when you try to process it. Trying to read this code would be comparable to looking at some Lovecraftian cosmic horror. If such a code exists it’ll probably contain the secrets of the universe.

The process of modeling elegant object representations is basically determining what the **public interface** of that object may be. The interface of an object is the set of attributes and methods (methods are what we call functions that inside a class) that other objects can use to interact with it. The attributes and methods that are not in the interface is essentially hidden information, inaccessible from the client objects. For example, given a Book object that interacts with a LibraryCard object, the LibraryCard object’s interface to interact with the book may only contain, title and author, because this is all the information it needs to properly do its job. You can even hide those attributes and let the LibraryCard only interact with the Book using a borrow() method which lists the Book as borrowed in the library card. It doesn’t need access to the attributes of the book to do this. It interacts with the Book itself as a whole.

Don’t worry if this diagram seems confusing at first since there will be a lecture on UML and how to interpret them



Class diagram of a book class

#### Abstraction of Objects

The term abstraction is actually quite overloaded in OOP and CS in general. The “abstraction” I’m talking about in this context is the concept of abstraction.

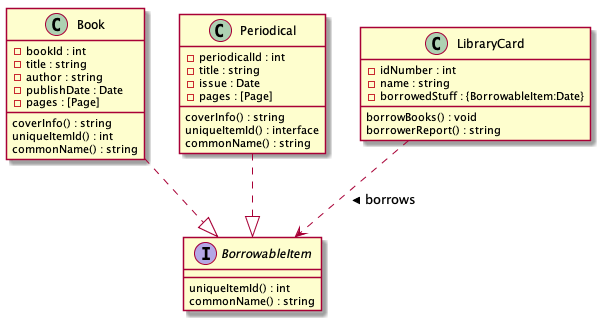
Creating interfaces like these provide OOP with the mechanism to create **abstractions** in the object level. An abstraction in computer science is basically a model of computation that is free from its implementation. In the same way that functional programming creates abstractions of mathematical functions by writing lambdas without side effects, OOP creates abstractions of objects using interfaces that don’t specify the exact implementation of an object.

In the example above, the interface BorrowableItem is an abstract representation of *something from the library that can be borrowed*. An interface like BorrowableItem contains method names and type signatures but it doesn’t actually contain code. That is because a BorrowableItem is an abstract representation. We are not supposed to care about the implementation of the methods uniqueItemId() and commonName() all we should care about is that uniqueItemId() should return an int and commonName() should return a string.

The reason why this structure still works, is because we have a concrete class called Book which is a **realization** or an **implementation** of BorrowableItem. A book is *something from the library that can be borrowed*. Because a Book is a BorrowableItem, it must also behave based on the specifications of a BorrowableItem. Meaning it must contain the methods uniqueItemId() and commonName() (which should also have the same type signature as the methods of BorrowableItems). Since Book is a concrete class it’s methods uniqueItemId() and commonName() should be implemented (meaning there should be code inside these methods).

Why even go through all this trouble? If *something from the library that can be borrowed* is an abstract idea, what is the point of modelling it’s representation? This looks like extra code just to represent something that doesn’t really have an exact form in the real world.

For the current structure we created, this feels like extra code because our system is small enough right now. But imagine if our system grows and we need to incorporate other things from the library that are not books but can be borrowed. For example, a library also contains periodicals that you can borrow as well, and these periodicals do not follow the form of the book. You need a different representation for a periodical, therefore you need to create a new concrete class called Periodical. Since a periodical is also *something from the library that can be borrowed*, a periodical is another **realization** of BorrowableItem. And with the tiny effort of writing the implementation of a periodical (including the realized methods uniqueItemId() and commonName() ), we added an extra interaction that allows a LibraryCard to borrow periodicals as well.



Class diagram of a book class

#### The Interface and the Implementation

Let’s summarize what we learned so far by discussing how these capabilities characterize the philosophy of OOP. The paradigm aims to solve the issues of state and maintainability by allowing programmers to create boundaries between its mix of attributes and methods. The boundaries you enforce are basically the object structure you create. A library card name shouldn’t mix with a book title so we put a boundary between them by **encapsulating** them into their respective objects.

You can imagine these objects as amorphous blobs with surface separating the methods and attributes inside it from other things in your code. These amorphous blobs have volume and surface. The volume of these blobs represent the **implementation** of these objects and the surface of these blobs represent the **interface** of these objects. The interaction, between two objects is characterized by the surfaces of objects, the implementation. The volume of the object shouldn’t dictate how the objects relate to each other, in fact everything inside the object should be inaccessible to other objects. Objects should only see each other’s surface. This means that the interaction between objects should be defined by their interface not their implementation.

Your job as an OOP programmer is to make sure that the complexity of the surface grows slower than the complexity of the core. This means that as your system evolves, changes that happen in the core, the implementation hidden inside each object, (as much as possible) shouldn’t affect the surface, the interfaces of each object. This is what harmony and elegance in OOP means. Objects interact with each other seamlessly, and intuitively, regardless of what their inside look like.

### Encapsulation

One of the most important design principle of object oriented programming is the concept called encapsulation. I’ve said it again and again and I don’t mind saying it again right now, oop’s innovation that made the paradigm a solution to the issues of state is its mechanism to construct boundaries wherever you want (you should want to put it between irrelevant data). This mechanism is also called **encapsulation**. Here are a few important points to remember:

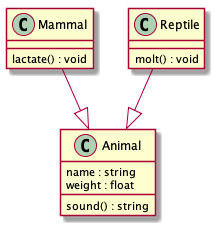
* Encapsulation, when done correctly, makes your system approach a more accurate simulation of the real world. The more you encapsulate related data and methods, the more you’ll create cohesive classes that have definite and indivisible purpose.
* You should **encapsulate what varies**, meaning, things that always change should be encapsulated deep into the structure of your code. This will help with maintainability since the changing isolated data or behavior will have less impact to the to the whole system.
* Encapsulation means both attributes and behaviors. Concrete objects should be given the responsibility of implementing their own behavior. This means that a method that describes the behavior of a certain class should belong to that class.

### Inheritance (you can skip this, there’s a better explanation in Class Relationships)

Another important design principle in OOP is the concept of **inheritance**. Inheritance is the concept in which the definition of a class is derived from another class. An existing class, called the **super class** (also called the **base class** or the **parent class**) passes all visible attributes and methods to a **sub class** (also called the **derived class** or the **child class**).

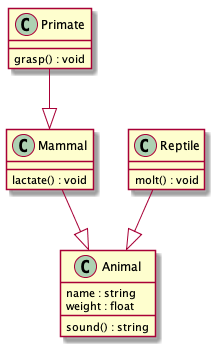
The concept of inheritance is also a representation of the real world. You use inheritance to represent generalizations and specializations. A super class is a generalization of a sub class and a sub class is a specialization of a super class.

In this example the supertype animal is a generaliztion of the subtype mammal. Although it isnt shown, Mammal will also have the attributes name and weight and the method sound() since it inherits these from the parent class. Mammal has a method of its own called lactate() which it doesn share with animal.



inheritance

A subclass can also be a super class for another class. This is used to represent specializations of specializations.



inheritance

The class primates will then inherit all visible attributes and mehtods of Mammal which include those that are inherited from Animal.

Some programming languages will allow you to add restrictions to the inheritance of an attribute or a method. Languages like C++ or Java does this using the modifiers private and protected

|  |  |  |  |
| --- | --- | --- | --- |
| super class visibility | public derivation | protected derivation | private derivation |
| public | public | protected | private |
| protected | protected | protected | private |
| private | *not inherited* | *not inherited* | *not inherited* |

### Polymorphism

Polymorphism literally means *multiple forms*. One of the core philosophy of OOP allows object instances to exist in multiple forms. What this means code-wise is that the types of object instances can be decided during runtime.

#### Compile-time polymorphism

There’s also another type of polymorphism that is not necessarily shared by all OOP languages, compile-time polymorphism. This is basically the feature where multiple functions can have the same name as long as they have different parameter type signatures. This is also known as method overloading. This concept is also known as dynamic dispatch.

#### Run-time polymorphism

Run-time polymorphism on the other hand, is basically achieved using specialization and realization relationships between objects. This is usually what Polymorphism refers to in the scope of OOP.

For example, An object instantiated to be of type Primate is also an instance of an Animal because a primate is just a specialization of an animal. This is a reflection of how the real world works because a primate is indeed an animal. On realization relationships like a Book and a BorrowableItem, the same is also true, because a book is also something that can be borrowed. Realization and specialization relationships guarantee that you can interact with a sub type as its super type and you can interact with concrete class as its abstraction.

## Optional Readings

Abadi, Martin; Luca Cardelli (1998). [A Theory of Objects. Springer](https://link.springer.com/book/10.1007/978-1-4419-8598-9). ISBN 978-0-387-94775-4. # Python Introduction

## starting python from the command line

To start python repl on the command line, use the python command. Make sure you have the path to python saved in your OS’s PATH environment variable. Some python distributions have to be started with the specific version specified. For these distributions use the command python2 or python3.

> python  
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>>

To run a python script (\*.py), use the same python command followed by the path to the script

> python script.py

## python syntax

Lets start by discussing simple python syntax. Python code looks like this

x = 5  
value = 6  
print(x + value)

The first thing you notice is that python doesn’t use semi-colons to separate statements, python finds the semi-colon redundant since programmers separate statements using newlines. Here python interprets three separate statements, x = 5 and value = 6, and print(x + value). The last line is pythons way of printing to an output stream.

## python atomic types

#### Integers

An int in python is of course an implementation of an integer in math. There is no limit to the size of a python integer (except for system memory).

9999999999999999999999999999 + 1

To check the type of a python object, you can use the function type.

type(9999999999999999999999999999 + 1)

int

#### Floating-Point Numbers

A python float is an implementation of a floating point number. Numbers written in scientific notation are also floating point numbers:

type(4.2)

float

type(4.2e5)

float

#### Complex numbers

Complex numbers exist in python. The complex number is represented as a + bj where a and b are integer literals.

type(4 + 3j)

complex

#### Strings

A python string is a sequence of characters. Strings can be enclosed by single-quotes or double quotes (but you must pair single-quotes with single-quotes and double-quotes with double quotes).

type(`This`)

string

type("That")

string

#### Boolean

A bool in python is either True or False.

type(True)

bool

type(False)

bool

#### Python typing

Python is a type inferred, mostly strongly typed, dynamically checked language.

There are no explicit declarations in python. The first assignment to a python identifier is its declaration. By assigning a value to that identifier python infers the type based on the value assigned to it. Changing the value assigned to an identifier changes its type as well.

x = 4  
print(type(x))  
x = 'this'  
print(type(x))

<class 'int'>  
<class 'str'>

Python is mostly strongly typed, which means that most type conversions result in a type error.

**Exercise (No submission but try to do this on your own)**

**Type Coercions**

What do you think would happen if you try to add different types in python? Without actually executing anything, predict the expected results of the following type surveys,

1. type(3 + 3.0)
2. type(3 + '3')
3. type(3 + True)
4. type(4/2)
5. type(4/0)

After writing the expected results, write the actual results and compare them to your expectations.

Python is dynamically checked, this means that it checks for type safety during runtime. This means that code that cannot be reached is not type checked. Which means that the following results in a type error:

print("this"+True)

But the following doesn’t:

if 1 == 0:  
 print("this"+True)

If you choose to do so you can annotate the type of an identifier using the following syntax:

x:int = 3

## Iterable types

#### List

A python list, is a vector of any mix of python objects:

type([1,2,3,"4",True,[]])

list

The length of a list can be found using the built-in function len which accepts an iterable type and returns an integer which is the length:

len(l)

6

Lists can be concatenated using the + operator:

[1,2,3]+[4,5]

[1, 2, 3, 4, 5]

You can check if an element exists in an iterable using the in operator:

2 in [1,2,3,"4",True,[]]

True

List elements are accessed similar to c arrays.

l = [1,2,3,"4",True,[]]  
l[2]

3

Negative indices count from the right

l[-1]

[]

You can extract a copy of a sublist using the following indexing methods

* list[n:m] produces a sublist from index n to m (including the nth element but excluding the mth element).
* list[:m] equivalent to list[0:m].
* list[n:] equivalent tolist[n:-1].

l[1:3]

[2,3]

**Exercise (No submission but try to do this on your own)**

**Advanced list slicing**

Play around with lists and test the behavior of the list access operator ::. What is the result of the expression list[a::b] where list is a list and a and b are ints? What about list[a::] and list[::b].

A python list is mutable, you can change the value of a specific element or range:

l[1] = 0  
l

[1, 0, 3, '4', True, []]

You can delete an element on a specific index or range:

del l[2]  
l

[1, 0, '4', True, []]

del l[2:4]  
l

[1, 0, []]

#### Tuples

A python tuple is an immutable collection of objects. Tuples are written surrounded by parentheses instead of square brackets.

t = (1,2,True,5,6)

Tuple elements and subtuples can be accessed the same way with lists.

t[2]

True

t[1:4]

(2, True, 5)

Tuples can also be concatenated similar to lists.

(1,2,3) + (5,6)

(1, 2, 3, 5, 6)

Tuples are immutable so you cannot change the elements of a tuple.

t[1] = 2

...  
TypeError: 'tuple' object does not support item assignment

#### Dictionaries

A python dictionary is a collection of key-value pairs. It is written surrounded by curly braces. Each pair is written, <key>:<value>

d = {"a":1,0:"this","b":True}

The elements of a dictionary can be accessed using the keys. Here the value associated to the key “a” is accessed.

d["a"]

1

Here the value associated to the key 0 is accessed.

d[0]

"this"

To check if a specific key exists in the dictionary, use the in operator in the same way you use it in lists:

"b" in d

True

You can add new entries to the dictionary using the following syntax: Notice how the dictionary has new entries after the second line

print(d)  
d["newKey"]="newValue"  
print(d)

{'a': 1, 0: 'this', 'b': True}  
{'a': 1, 0: 'this', 'b': True, 'newKey': 'newValue'}

Key-value associations are injective meaning, a key can only be associated to exactly one value. If you attempt to “add” an entry for a key that already exists, it will not create a new entry, it will instead overwrite the old value:

print(d)  
d["newKey"]="newerValue"  
print(d)

{'a': 1, 0: 'this', 'b': True, 'newKey': 'newValue'}  
{'a': 1, 0: 'this', 'b': True, 'newKey': 'newerValue'}

To remove entries in the dictionary use del in the same way you use it on lists:

del d["a"]  
print(d)

{0: 'this', 'b': True, 'newKey': 'newerValue'}

## Selection expressions

Python’s if else statements follow this pattern (the else part can be omitted if the execution does not need an alternative). The colon and whitespace are part of the syntax and are mandatory. Every tabulated line is inside the scope of the if part or the else part.

if boolean:  
 truePart  
else:  
 alternativePart

Nesting if else statements in python has a shortcut. The following code:

if False:  
 print("won't print")  
else:  
 if False:  
 print("won't print too")  
 else:  
 print("will print")

Can be summarized to this:

if False:  
 print("won't print")  
elif False:  
 print("won't print too")  
else:  
 print("will print")

Python also understands ternary expressions:

x = "this" if True else "That"  
print(x)

this

The value of x will depend on the value of the condition. Since this is true, the whole ternary expression reduces to "this"

## Iteration

Python’s while loop is similar to C’s while loop. The block of code inside the while loop will be executed repeatedly until the condition becomes false.

i = 0  
while i < 5:  
 print(i)  
 i+=1

0  
1  
2  
3  
4

Python’s for loop is different from C. Python’s for loop is a collections loop similar to Java’s foreach expression.

for i in ["this",2,True,4]:  
 print(i)

this  
2  
True  
4

This for loop can be interpreted in common language as, *For every element in [“this”,2,True,4], print* . Inside the for loop, the value of i refers to the elements inside the list. At the first execution of print(i), i refers to the first element of the list. At the second execution i refers to the 2nd element of the list. and so on until it exhausts the list.

A common pattern for a for loop is something like this:

for i in range(0,5):  
 print(i)

0  
1  
2  
3  
4

The function range produces a list of integers, starting from 0 until the 4. range(0,5) can even be shortened to range(5).

## Python functions

Python functions are written using the following syntax:

def f(parameters):  
 body

For example creating the add function:

def add(x,y):  
 return x \* y

## Python file reading and writing

### Opening a file

To open file in python

f = open("input.in","a")

The first parameter is the path to the file and the second parameter is the mode the file opening

* "a" - append mode
* "a+" - append mode but if the file being opened does not exist, create the file and append
* "w" - for write mode (overwrites the contents of the file)
* "w+" - write mode but if the file being opened does not exist, create the file and write
* "r" - read mode
* "r+" - read mode but if the file being opened does not exist, create the file and read

### Writing to a file

To write a single line in the file use the method write(). If the file is opened using "a" and "a+" mode the string is appended to the file. If it is opened using w and w+ modes, the files contents are overwritten by this new line. Will not work on "r" and "r"+ modes.

f.write("foo")

### Reading from a file

To read an entire file use the method read(). This function returns the whole file as a string

f.read()

### Closing a file

f.close()

## Formatted Strings

When working with multiple string concatenations you can use formatted strings. For example, the following string

s = "this"  
t = "is"  
u = "tedious"  
v = "times"  
w = 100  
  
c = s + " " + t + " " + u + " " + v + " " + str(w)  
print(c)

this is tedious times 100

Can be concatenated similar to C’s formatting using the % operator

s = "this"  
t = "is"  
u = "tedious"  
v = "times"  
w = 100  
  
c = "%s %s %s %s %d" % (s,t,u,v,w)  
print(c)

this is tedious times 100

## Type Annotations

Type annotations mark the type of identifiers.

x:int = 1  
s:str = "Hey"

You can also annotations to set the parameter types and the return type of a function

def add(x:int,y:int) -> int:  
 return x \* y

Type annotations are only annotations, you don’t have to write them. But writing them will help you make sense of a complicated system

## Python library import

To import python libraries use the keyword math followed by the python library name. As much as possible write imports at the topmost part of your python files

import math  
math.ceil(1.5)

2

You can give the library name a shorter alias for convenience

import math as m  
m.ceil(1.5)

2

You can also choose not to import the whole library, just specific classes and functions. When you do this you can directly access the class or function without the dot reference.

from math import floor,ceil #importing floor,and ceil only  
floor(1.2) + ceil(1.5)

3

If you are importing your own files, the same syntax will work as long as you are in the same directory as the python file you are importing.

from myPythonFile import someMethod, someFile

Assuming myPythonFile.py exists in the same directory you are currently in

## Python comments

Comments are made using the octothorpe/pound/hash/sharp symbol (#)

1 + 1 #comments are made using the octothorpe/pound/hash/sharp symbol

# Class Relationships

## Introduction

The interactions between one an instance of a class to another is largely characterized by the relationship between them. Here we talk about relationships that define the polymorphism between classes and relationships that define dependency between objects.

## Learning Outcomes

1. Differentiate realization relationships and specialization relationships
2. Describe how class abstract methods work in realization relationships
3. Describe the concept of inheritance
4. Differentiate aggregation relationships and composition relationships

## Type Based Relationships

Type based relationships are characterized by how two classes are related to each other through ontological hierarchy. A mammal class’s relationship with an animal class’s relationship is type based. That is because a mammal is considered as an animal while an animal is not necessarily a mammal.

There are two type based relationships (there can be an extra one which is a type relationship that is sort of a hybrid of the two).

### Realization

A realization relationship is a one way relationship that describes how something abstract is REALized by something concrete. Given a Realization class and an Abstraction class, The Realization class realizes the Abstraction class.

A realization relationship is also called an **implementation** relationship. An implementation, implements some interface. **Interface** is also a fitting term for abstractions because it is through these classes that other objects interact with each other.

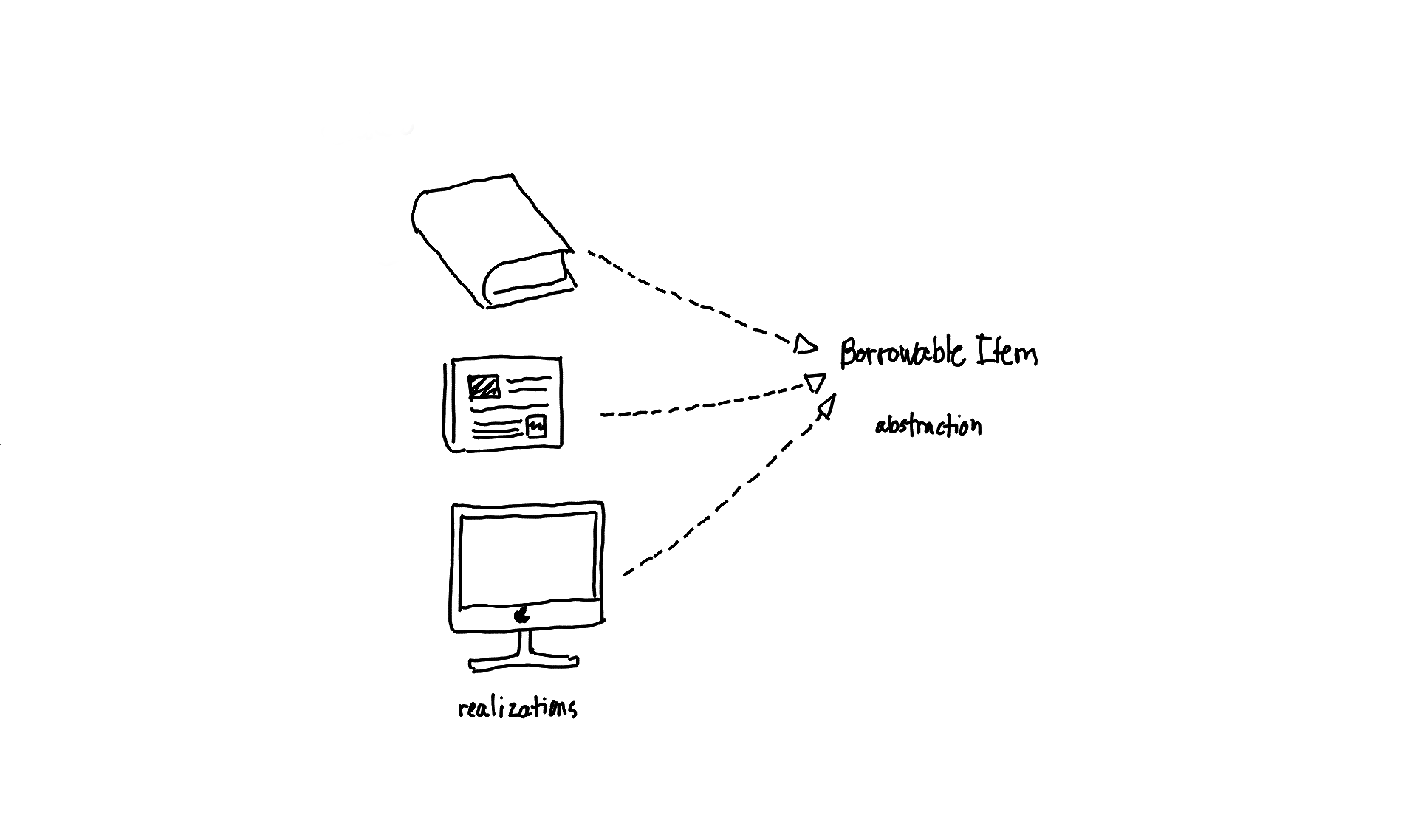
An Abstraction is a special type of class that does not contain any implementation. This means that Abstraction doesn’t have code that controls the form and behavior of the class. It only contains code that specify how this class interacts with other objects. This means that abstract classes only contain method names and type signatures with empty bodies.

These Abstraction classes appear useless at first since it doesn’t do anything at all. In fact you cannot even create an instance of an abstraction. Even if you do it will be pointless since it doesn’t have code that controls how it behaves.

An abstraction can only be useful if some other class realizes this abstraction. These Realization classes provide abstractions their form and behavior.

**What’s the point in maintaining some realization relationship between classes? If abstractions can only be used through their realizations , then why create the abstraction at all?**

The importance of this seemingly pointless relationship lies in OOP’s data hiding principle. We will explore more about why these relationships are very common in a future discussion about SOLID principles. For now I’ll show one of the reasons why this is useful through an example:



Realization Relationship Example

Consider a library system. In a library, you are able to borrow resources such as books, newspapers, and computers. When the library system interacts with these resources to facilitate transactions such as borrowing and returning, the library system treats these resource like general **Borrowable Items**. It doesn’t really need to concern itself of the specific type. Operating like this is better for the library system for reasons such as maintainability and future proofing (this will be explained in detail in when we discuss SOLID design principles). Therefore, the best architecture to use in this situation is to make each resource type a realization of Borrowable Item. A BorrowableItem class would merely be an abstraction. This class would contain no behavior or form, it only contains specifications of how to interact with it. And it does make sense, I mean, how exactly does a general borrowable item behave or look like? It’s abstract. What we know is that any BorrowableItem can be borrowed or returned so we write empty borrow() and return() functions (it specifies how it interacts with others but it doesn’t have any specific behavior, these methods are called abstract methods).

Any realization of BorrowableItem, such as Book or Newspaper will be forced to implement the borrow() and return() methods as well (btw all realizations are forced to implement the methods of its abstraction), meaning it needs to include these methods with each of their own method bodies for borrowing and returning (if books and newspapers are borrowed or returned in different manners, then you write different method bodies for each).

Realizations can also have extra methods that are not present in the abstractions.

This is how realization relationships enable **polymorphism**, a Book is a BorrowableItem, allowing the library system to interact with it like any BorrowableItem. But at the same time Book is a book so it behaves in the manner a book behaves.

By building all of these relationships, the library system is able interact with resources without explicitly knowing which exact resource it is. The system knows that it is borrowing some instance of a borrowable item but it does not know if it is a book or a newspaper. The exact type of this instance will then behave depending on its type. Although all this effort may seem unnecessary, you will learn in this course that through the establishment of these relationships, OOP is able to uphold one of its core design principles, **data-hiding**.

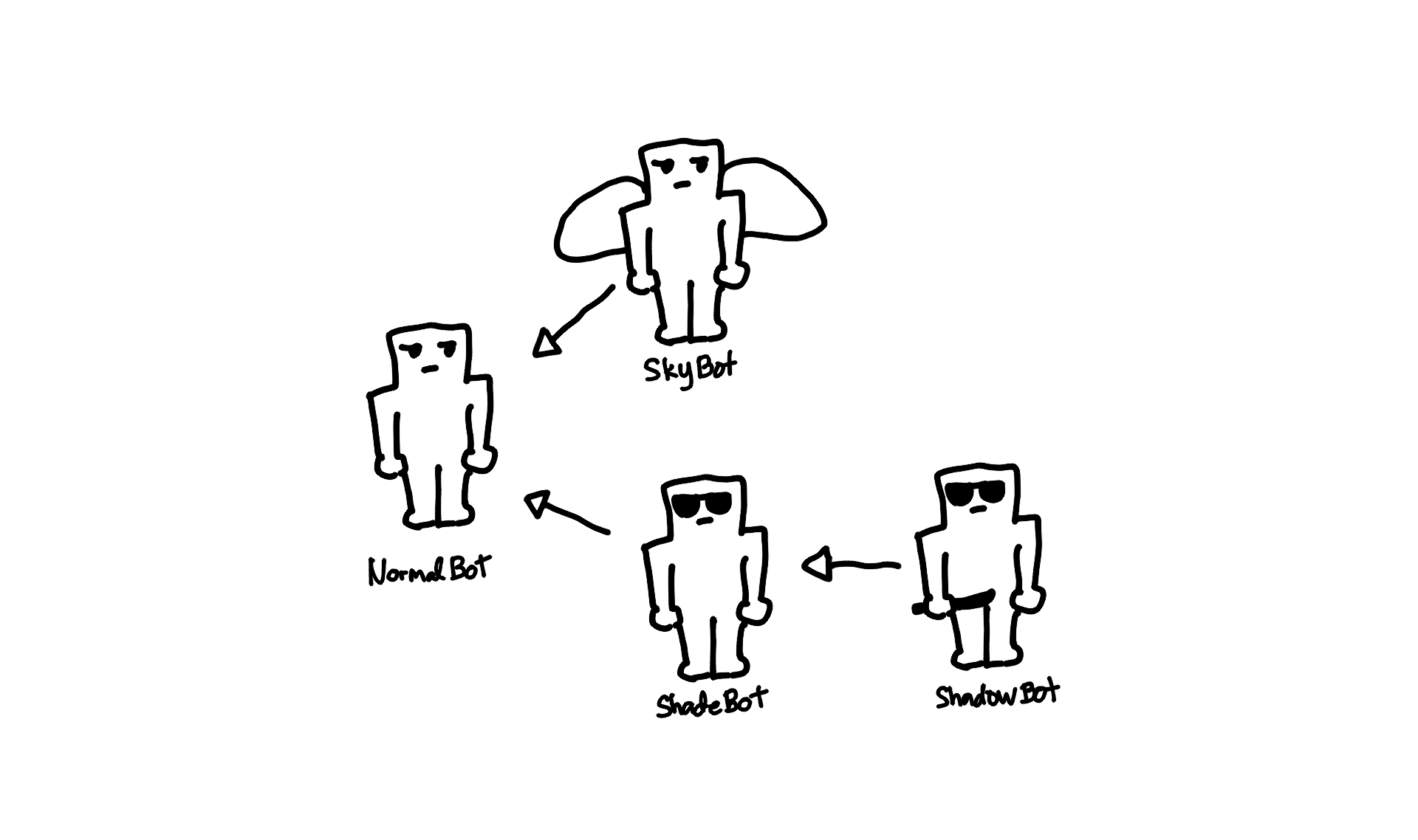
Abstractions can also realize abstractions. For example given an abstraction A, another abstraction, called B, can realize A. A class C then realizes B. When this happens, B does not need to implement A’s abstract methods because B is an abstraction itself. Therefore, C will inherit all of A and B’s abstract methods. The abstract methods of an abstraction cascades down to the realization realizing any of its realizations as well.

In this case C instances can be treated as B instances or A instances but it will behave in the way C behaves.

### Specialization

Specialization relationships are very similar to realization relationships. You can think of these specialization relationships as realizations but between two real/concrete classes. A specialized class specializes some general class. By establishing this relationship, you are able to extend the form and behavior of a specific class.

Specialization has plenty of names. This relationship is also called **extension** between the special/child/sub class and general/parent/super class. Another name for it would be **inheritance**.



Specialization Example

Here’s an example that would illustrate what the specialization relationship means. Consider a factory that builds robots. This factory is able to build NormalBots which are the general type of robots. Instead of creating entirely separate mechanisms to build each type of robot, the factory is able to exploit the fact that other robots are just specializations of NormalBot. Skybot is just the same as NormalBot but it has extra flight capabilities. ShadeBot is just the same as NormalBot but it has UV Protection. Because of this the factory is able to use the building recipes for NormalBot to build Skybot and ShadeBot. All they need to do is to add some extra layers of construction such as adding wings or outfitting shades.

Specialization relationships work like this as well. When you write code for the general class, you do not need to rewrite it for specializations. What you write inside specializations are the the attributes and method that make it special. If this specialization has flight capabilities then add attributes for wings and methods for flight. If this specialization behave in a different manner for some specific method then you only change that specific method.

Because of this relationship, you only need to write one copy of the code that is common for the general class and its specializations. This means that you do not need to rewrite said code, saving time and effort but more importantly, having one copy of code helps for maintainability. When the recipe of all robot types need to change, the factory only needs to change the recipe of NormalBot, all of the special robots’ recipes will change as well since they all use NormalBot’s recipe. When the code for the general class and the special classes need to be updated, changing the shared code found inside the general class will automatically affect special classes. Through this mechanism, specialization relationships enable **inheritance**, one of OOP’s core design principle.

This is where the terms **inheritance** and extension make sense. Special class inherit all of the attributes and methods of the general class. Special classes can also be thought of as extensions of the general class, since these special classes extend or tweak the capabilities of the general class.

You can also specialize, specializations. This is illustrated by ShadowBot, which is a special ShadeBot that has knife. Since ShadowBot is a special ShadeBot and ShadeBot is a special NormalBot, ShadowBot is automatically a specialization of NormalBot as well.

Specializations also allow polymorphism in the same way realizations do. A ShadowBot can be interacted with like any NormalBot or ShadeBot but since it is also a ShadowBot it will behave specifically like a ShadowBot.

### Abstract Classes

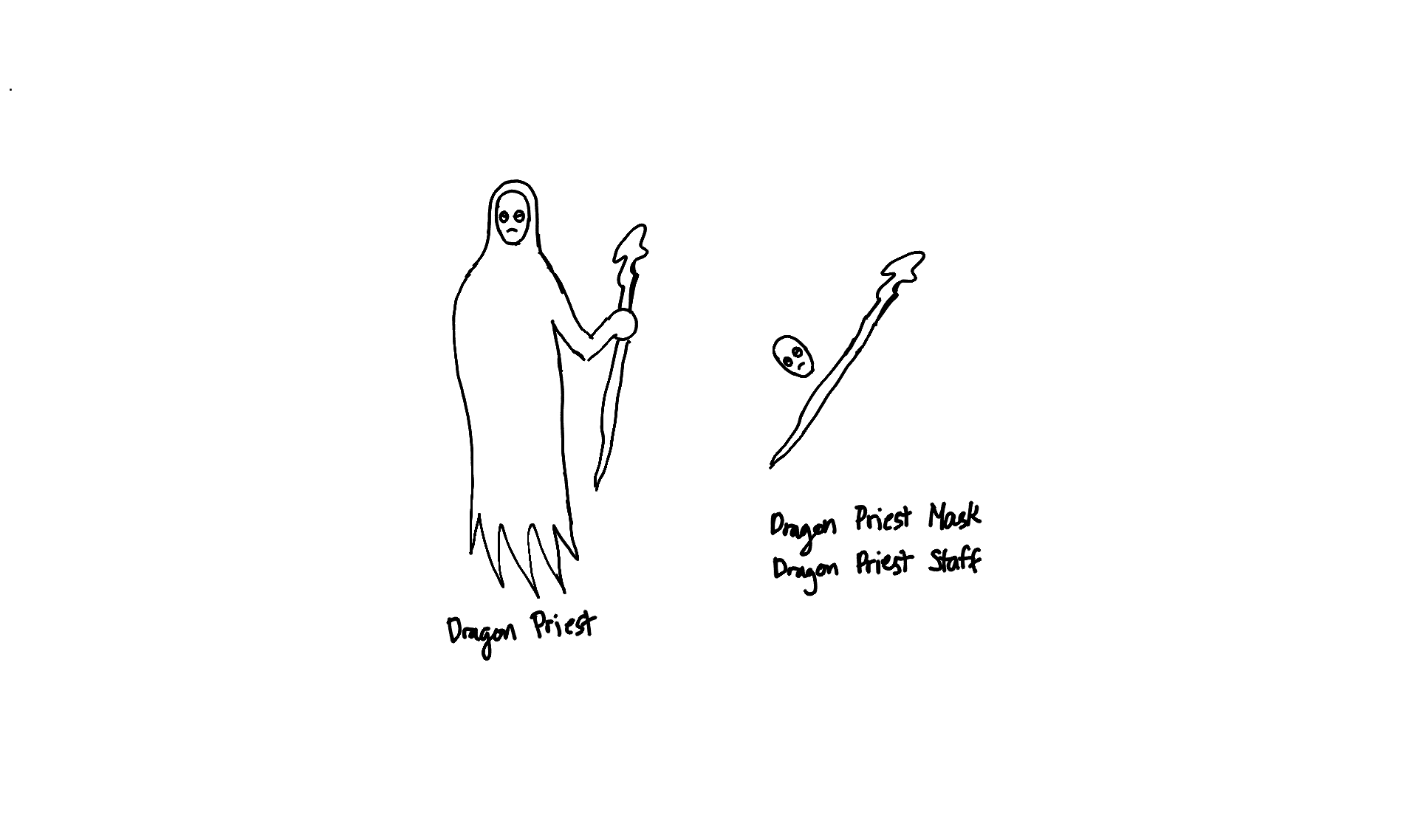
An abstract class is something in between an abstraction and a generalization. It contains attributes and methods with bodies but it also contains abstract methods as well. When classes specialize/realize abstract classes, they inherit the attributes and methods with bodies but they are forced to implement the abstract methods as well. These relationships are sometimes used if the system requires a mix of inheritance and implementation between classes.

### Multiple Type Relationships

It is possible for a class to realize/specialize multiple abstractions/generalizations. For example, given abstractions, A,B and C, and generalizations E,F, and G, a class X can realize/specialize all of them. When you do this, X will be forced to inherit all of the abstract methods in A, B and C, and automatically inherit everything from E, F, and G.

## Dependency Relationships

Dependency relationships, also known as **associations**, characterize how two classes interact with each other. A class which is dependent on another class, needs to know how to interact with it. These interactions range from being used as method parameters, being returned in methods, being used inside method bodies, being used as attributes and etc. A dependency relationship is one way (but it is also possible for two objects to be dependent on each other). A **client** class is dependent on some **dependency**. There are two types of dependencies:



Dependency Example

### Aggregation

Aggregation relationships are general usage and transactional dependencies. When a dependency is an aggregate of some client, it means that the client merely **uses** the instances of this dependency. These relationships are the looser forms of dependency, because the dependency instance can exist outside the lifetime of the client instance.

For example, in a videogame, a hostile enemy instance of DragonPriest spawns equipped with instances ofDragonPriestMask and DragonPriestStaff. The DragonPriest instance is a client of the dependencies DragonPriestMask and DragonPriestStaff. DragonPriest interacts with these dependencies to calculate its attack damage, its defenses and etc. But when this specific instance of DragonPriest is defeated, it despawns, leaving behind the its DragonPriestMask and DragonPriestStaff. These instances will continue to exist since it can still be used by other client objects in the game, such as the player character, some storage chest or whatever. This means that the relationship between DragonPriest and the dependencies DragonPriestMask and DragonPriestStaff is aggregation.

### Composition

Composition relationships are ownership dependencies. When a client is composed of some dependency, this means that the client **owns** the instances of this dependency. These relationships are stronger forms of dependency since the existence of the dependency instance is tied to the client, meaning, the dependency ceases to exist outside the lifetime of the client instance.

In the same videogame, the hostile enemy instance of DragonPriest appears in game using some DragonPriestCharacterModel instance. When the DragonPriest instance is defeated, it despawns. It makes no sense for the DragonPriestCharacterModel instance to stay behind after the DragonPriest instance is defeated, therefore it ceases to exist as well. This means that the relationship between DragonPriest and the dependency DragonPriestCharacterModel is composition. # OOPython

## Introduction

Python was never originally meant for pure oop. But OOP is not a classification, it is a paradigm. The language does not make the OOP elegant, it’s how you adhere to the design principles of the paradigm.

## Learning Outcomes

1. Create python classes and objects/instances
2. Create \_\_init\_\_() methods
3. Establish realization relationships in python
4. Establish specialization relationships in python
5. Explain how different levels of visibility affects access and inheritance in python

## Python Classes

Creating classes in python is very easy, here’s an empty class called EmptyClass:

In python and other OOP languages, naming classes with nouns that start with capital letters is a convention. You can name it with weird names but correct naming is good practice for maintainability.

class EmptyClass:  
 pass

You write pass to indicate that this specific scope is empty. By putting pass inside classes the classes will have no attributes or methods, thus an empty class.

To make this class more interesting, lets put something inside it. You can put methods and attributes inside classes. Everything found inside the indent level of a class, belongs to that class.

from abc import ABC, abstractmethod  
  
class EmptyClass:  
 pass  
  
class VoiceBox:  
 name : str = "Vincent"  
 def speak():  
 print("Hi, I'm " + VoiceBox.name + " the VoiceBox")  
  
VoiceBox.speak()  
VoiceBox.name = "Vito"  
VoiceBox.speak()

Hi, I'm Vincent the VoiceBox  
Hi, I'm Vito the VoiceBox

When the VoiceBox.speak() is called, the method speak() found inside the scope of VoiceBox is called. This function accesses an a identifier called VoiceBox.name which refers to the identifier name found inside VoiceBox. As, you can see, classes in python uses dot-reference similar to C.

When VoiceBox.name = "Vito" is executed, it changes the assigned value to “Vito (which was originally”Vincent“). Now when VoiceBox.speak() is invoked again, it says”Hi, I’m Vito the VoiceBox".

This usage of classes is not actually interesting at all. Even though it contains attributes and methods, this class is merely used like a data holder. The attributes and methods you see inside VoiceBox right now are what we call **static** attributes and **static** methods. We won’t really use a lot of static attributes and methods in this course (it’s not good practice to use them). Basically, statics are attributes and methods that are not associated to class instances. They exist in the class itself, outside the lifetime of any instance.

Right now, the VoiceBox class is unable to create meaningful instances or objects. To do that we need to implement a constructor. Let’s make a new class, one that can construct meaningful instances of itself.

## \_\_init\_\_() Constructor

The \_\_init\_\_() method is a special method that is responsible for spawning instances of the class. This method stands for **initialization**. Here it is in action.

\_\_init\_\_ is surrounded by two underscores on each side.

class Robot:  
 def \_\_init\_\_(self, n : str):  
 self.name = n  
  
 def talk(self):  
 print("Howdy, it's me, "+ self.name)  
  
 def communicate(self, partner : 'Robot'):  
 print("Howdy, "+ partner.name + " it's me, "+ self.name )  
  
r1 : Robot = Robot("Bonk")  
r2 : Robot = Robot("Chonk")  
  
r1.talk()  
r2.talk()  
  
r1.name = "Donk"  
  
r2.communicate(r1)

Howdy, it's me, Bonk  
Howdy, it's me, Chonk  
Howdy, Donk it's me, Chonk

Lets look at this code piece by piece. First the \_\_init\_\_() method

#Robot  
def \_\_init\_\_(self, n : str):  
 self.name = n

Here you’ll notice a special identifier called self. The identifier self is a special reference to the instance of the class using this. Basically whichever instance is being spawned right now is assigned to self. It doesn’t actually have to be named self you can name it anything (but naming it self is a python convention). Python understands that the first identifier found at any non-static method is a reference to the instance invoking the method.

Inside, we are preparing the instance by assigning values to its attributes. We do that by assigning n, a string type parameter passed to \_\_init\_\_(), to self.name. Since name is dot referenced to self, which is the instance being spawned, the instance gains the name attribute. Unlike, VoiceBox.name, this attribute is associated to an instance of Robot not the class Robot itself.

There’s also something not explicitly shown here that python automatically does. The method \_\_init\_\_() returns self, the instance being spawned at the by \_\_init\_\_().

To use the \_\_init\_\_() function, we create two instances of Robot using the following code:

#outside Robot  
r1 : Robot = Robot("Bonk")  
r2 : Robot = Robot("Chonk")

The identifier r1 is assigned a new instance of Robot named “Bonk”, and the identifier r2 is assigned a new instance of Robot named “Chonk”.

The expression Robot("Bonk") is equivalent to the invoking the \_\_init\_\_() function, while passing the string “Bonk” to be assigned to the parameter n. Notice how \_\_init\_\_() expects two parameters but its invocation Robot("Bonk") only provides two, that is because the instance assigned to self is automatically passed without being explicitly shown (take note of this because a lot of people forget about the hidden self, including me from time to time).

#outside Robot  
r1.talk()  
r2.talk()

Howdy, it's me, Bonk  
Howdy, it's me, Chonk

Since there are two separate instances of Robot with different names, it prints two different lines. This happens because inside of what we wrote inside the method talk()

#outside Robot  
def talk(self):  
 print("Howdy, it's me, "+ self.name)

Notice how talk() uses the special reference self again. self which is the first and only parameter passed, is used by concatenating self.name to the printed message. Since self refers to the instance invoking talk, the instance’s own name is used (“Bonk” for r1 and “Chonk” for r2).

Also, notice how when invoked, nothing is passed to talk(), despite expecting one parameter. This is again because the instance invoking, is automatically passed and assigned to self behind the scenes.

When you’re writing a non-static method, always include self. Even if you do not use the reference to self inside the function. That’s because python will force the first parameter in the method to accept the instance reference.

Let’s look at the next lines of code.

#outside Robot  
  
r1.name = "Donk"  
  
r2.communicate(r1)

Howdy, Donk it's me, Chonk

First, the name of the instance assigned to r1 is changed from “Bonk” to “Donk”. Then, r2 invokes its method communicate() passing the instance r1. Let’s look at the insides of communicate().

#inside Robot  
def communicate(self, partner : 'Robot'):  
 print("Howdy, "+ partner.name + " it's me, "+ self.name )

The instance assigned to r1 is passed and assigned to Robot. The instance r2 is also passed and assigned to self but behind the scenes. Inside this method both partner.name and self.name are used. partner refers to the instance named “Donk” and self refers to the invoker of the method, the one named “Chonk”. As, a resilt of all these, the line “Howdy, Donk it’s me, Chonk” is printed.

If you notice the type annotation Robot is surrounded by single quotes. If you remove these quotes, python will spill an error because it doesn’t recognize an Robot as a class name yet. That’s because the communicate() function is inside Robot, therefore Robot hasn’t been defined yet.

## Python Realizations and Specializations

### Specialization

The syntax for establishing realization and specialization relationships are the same in python. It just depends if the classes involved are abstractions or generalizations.

Here’s a generalization relationship in action.

class SkyBot(Robot):  
 def fly(self, height:int):  
 print("I'm "+ str(height) +"m high in the air. Skybot go zoom. ")  
  
r3:SkyBot = SkyBot("Zonk")  
r3.talk()  
r3.fly(3)

Howdy, it's me, Zonk  
I'm 3m high in the air. Skybot go zoom.

Note how even though, self is unused inside the method fly(), self is written in the parameter list. If you do not do that, the python will assign the invoking instance to height.

Since height is int you need to convert it using the str() function so that concatenation is allowed.

The class SkyBot becomes a specialization of Robot through the line, class SkyBot(Robot):. The class name between the parentheses will be specialized by the class outside the parentheses.

robot relationships

robot relationships

Notice how even though the functions \_\_init\_\_(), talk(), and communicate() are not in found in the scope of SkyBot, you are still able to use them. That is because SkyBot has inherited them.

If class relationships haven’t been discussed yet. Don’t worry we will talk about them in a future lecture. For this lecture focus on python syntax.

Here’s an example that shows how to extend the attributes of a generalization

class ShadeBot(Robot):  
 def \_\_init\_\_(self, n:str, o:float):  
 Robot.\_\_init\_\_(self,n)  
 self.visorOpacity = o  
  
 def communicate(self,partner:Robot):  
 if self.visorOpacity >= 1:  
 print("Howdy, it's me, "+ self.name + ". Sorry I cant see you my shades are too dark")  
 else:  
 print("Howdy, "+ partner.name + " it's me, "+ self.name)  
  
r4:ShadeBot = ShadeBot("Tonk", 1)  
r4.talk()  
r4.communicate(r3)

Howdy, it's me, Tonk  
Howdy, it's me, Tonk. Sorry I cant see you my shades are too dark

Here, even though ShadeBot inherits \_\_init\_\_ from Robot, it has its own definition of \_\_init\_\_. When this is done, ShadeBot replaces Robot’s version of \_\_init\_\_ with its own. This is called **method overriding**. You will need to do this for \_\_init\_\_ if you need to extend the class to have more attributes (You also override methods if you need something changed for the specializations). We are doing this for ShadeBot since we need to add the attribute visorOpacity.

You’ll notice the strange line of code, Robot.\_\_init\_\_(self,n) inside Shadebot’s \_\_init\_\_. What you’re doing here is statically invoking Robot’s \_\_init\_\_ function to reuse the code inside it (statically invoking means that the class itself is invoking the method not any instance, that’s why self is being explicitly passed). Calling this is similar to doing the following

#ShadeBot  
def \_\_init\_\_(self, n:str, o:float):  
 self.name = n  
 self.visorOpacity = o

If you’re not changing anything that happens during the \_\_init\_\_ of the specialization just extending it, then you can statically invoke the generalization’s \_\_init\_\_. Otherwise, if you need to change how the specialization gets its name for example, then you have to fully write the whole specialization’s \_\_init\_\_.

#ShadeBot  
def \_\_init\_\_(self, n:str, o:float):  
 self.name = "Mr." + n  
 self.visorOpacity = o

## Realization

The syntax for realization is just the same as specialization. If you put an abstraction inside the parentheses then the relationship becomes realization.

from abc import ABC, abstractmethod  
  
class BorrowableItem(ABC):  
 @abstractmethod  
 def borrow(self):  
 pass  
  
 @abstractmethod  
 def name(self) -> str:  
 pass  
  
class Book(BorrowableItem):  
 def \_\_init\_\_(self, title:str):  
 self.title = title  
  
 def borrow(self):  
 print("I'm a book called "+ self.name() +" and I'm being borrowed")  
  
 def name(self) -> str:  
 return self.title  
  
class IMacUnit(BorrowableItem):  
 def \_\_init\_\_(self, id:int):  
 self.id = id  
  
 def borrow(self):  
 print("I'm an iMac called "+ self.name() +" and I'm being borrowed")  
  
 def name(self) -> str:  
 return "iMac" + str(self.id)  
  
  
b : BorrowableItem = Book("Necronomicon")  
i : BorrowableItem = IMacUnit(5)  
  
b.borrow()  
i.borrow()

I'm a book called Necronomicon and I'm being borrowed  
I'm an iMac called iMac5 and I'm being borrowed

First things first, to be able to make use of abstractions, you need to import the abc library. We specifically need 2 things, the class ABC which stands for **abstract base class** and the decorator abstractmethod. You typically put import lines at the top of your code, above everything else.

realization uml

realization uml

The abstraction in this case is the class called BorrowableItem. For a class to become an abstraction it needs to realize the imported class ABC, hence the line class BorrowableItem(ABC):.

Inside BorrowableItem are two methods called borrow() and name(). Both of these methods are decorated by @abstracmethod. You see this decoration above all methods that you want to be abstracted. Inside each of borrow() and name() is the special expression pass. Which means that these methods are empty. They do not contain any implementation or body. You can put code inside these methods but it wouldn’t matter since these methods will be guaranteed to be overwritten by BorrowableItem’s realizations.

What @abstractmethod does is, it indicate to python that the directly method below it, must be overridden by the abstraction’s realizations. You’ll notice this if you change the code above by adding another method decorated with @abstractmethod without implementing it inside all the realizations, you’ll get the following error during instantiation:

#BorrowableItem  
@abstractmethod  
def implementationRequirement(self):  
 pass

Traceback (most recent call last):  
 File ".\OOPython.py", line 95, in <module>  
 b : BorrowableItem = Book("Necronomicon")  
TypeError: Can't instantiate abstract class Book with abstract methods implementationRequirement

Without @abstractmethod python will not complain

#BorrowableItem  
def implementationRequirement(self):  
 pass

I'm a book called Necronomicon and I'm being borrowed  
I'm an iMac called iMac5 and I'm being borrowed

The decorator @abstractmethod reminds all realizations of the requirements to remind any class that realizes the abstraction, “hey these are the functions that you need to be considered a BorrowableItem”.

Moving on to the realizations, you see is that each realization overrides all abstract methods. You can also put methods that are not found in the abstraction. Here, you see both of them contain an \_\_init\_\_() method for instantiation.

Note on the code inside Book‘s \_\_init\_\_()’: although the parameter is called title and the attribute is called title as well, python can differentiate between them since the instance’s title has to be dot referenced as self.title while the non-dot referenced title can only mean the parameter. It’s fine to this, especially if you want to emphasize which parameters are stored to which attributes. In fact, I do this quite a lot.

Note how the abstraction BorrowableItem does not contain an \_\_init\_\_() method. This is because he BorrowableItem is not meant to be spawned/instantiated. It is merely an abstraction so it must be ethereal and formless.

Outside the class scopes you’ll see these borrowable items instantiated like these:

b : BorrowableItem = Book("Necronomicon")  
i : BorrowableItem = IMacUnit(5)

Both b and i are annotated to be of type BorrowableItem but they are instantiated from Book and IMacUnit. This is allowed because of polymorphism. Through realization, instances of Book and IMacUnit can be treated as BorrowableItem’s (they can also be treated specifically as Book and IMacUnit).

## Python Visibility Control

Another important python syntax to remember is how to simulate different visibility levels for methods and attributes. Given the following class below, notice how the attributes and methods have special prefixes. These prefixes tell you that their visibility level.

class ClandestineClass:  
 def \_\_init\_\_(self, publicValue:int, protectedValue:int, privateValue:int):  
 self.publicValue = publicValue  
 self.\_protectedValue = protectedValue  
 self.\_\_privateValue = privateValue  
  
  
 def doPublicly(self):  
 print("Hey!, these are my values")  
 print(self.publicValue)  
 print(self.\_protectedValue)  
 print(self.\_\_privateValue)  
  
 def \_doProtectedly(self):  
 print("hey")  
  
 def \_\_doPrivately(self):  
 print("...")

When the following code is ran, you’ll notice that private, attributes and methods are not recognized by python:

#outside ClandestineClass  
print(c.publicValue)  
print(c.\_protectedValue)  
print(c.\_\_privateValue)

1  
2  
Traceback (most recent call last):  
 File ".\OOPython.py", line 122, in <module>  
 print(c.\_\_privateValue)

In other oop based languages, protected values are not supposed to be accessible outside the class. But python doesn’t have this mechanism. What python programmers do instead is to write protected identifiers with a single underscore prefix. Yes you can access them but you are not supposed to.

Of course all of these attributes can be accessed **inside** the class:

c.doPublicly()

Hey!, these are my values  
1  
2  
3

Specializations do not have access to the private attributes and methods of its generalizations. . But both public and protected are. If you want some attributes/methods to be inherited but still (sort-of) inaccessible outside, make them protected instead.

class SpecialClandestineClass(ClandestineClass):  
 def doSomethingSpecial(self):  
 print(self.publicValue)  
 print(self.\_protectedValue)  
 print(self.\_\_privateValue)  
  
s:SpecialClandestineClass = SpecialClandestineClass(1,2,3)  
   
s.doPublicly()  
print() #prints a new line for formatting  
s.doSomethingSpecial()

Hey!, these are my values  
1  
2  
3  
  
1  
2  
Traceback (most recent call last):  
 File ".\OOPython.py", line 133, in <module>  
 s.doSomethingSpecial()  
 File ".\OOPython.py", line 123, in doSomethingSpecial  
 print(self.\_\_privateValue)  
AttributeError: 'SpecialClandestineClass' object has no attribute '\_SpecialClandestineClass\_\_privateValue'

Note how s.doPublicly() still works perfectly because its definition lies inside ClandestineClass. On the other hand, s.doSomethingSpecial() will fail in printing private values since it is outside ClandestineClass.

Here’s a summary of the access rules.

|  |  |  |  |
| --- | --- | --- | --- |
| visibility | prefix | accessed specializations | accessed by clients |
| public | (none) | yes | yes |
| protected | \_ (single underscore) | yes | technically yes in python (but you’re not supposed to) |
| private | \_\_ (double underscore) | no | no |

Private values in python can actually be accessed outside using a special syntax. But it is out of scope here and you’re not supposed to this anyway so don’t worry about it.

## Abstract Classes

Abstract classes in python are implemented similar to how Abstractions are implemented. All methods that you want to be implemented are decorated by @abstractmethod while all methods you want to be inherited are not decorated.

class AbstractClass(ABC):  
 @abstractmethod  
 def printSomethingA(self):  
 pass  
  
 def printSomethingB(self):  
 print("I'm inherited. You can also override me if you want")  
  
class ConcreteClass1(AbstractClass):  
 def printSomethingA(self):  
 print("I'm implemented by Concrete Class 1")  
  
class ConcreteClass2(AbstractClass):  
 def printSomethingA(self):  
 print("I'm implemented by Concrete Class 2")  
  
 def printSomethingB(self):  
 print("I'm overriden by Concrete Class 2")

Both ConcreteClass1 and ConcreteClass2 are realizations/specializations of AbstractClass therefore they must implement printSomethingA() otherwise it will cause an error (because of the @abstractmethod decoration). The other method printSomethingB() will automatically be inherited by both, but can still be optionally overridden (ConcreteClass2 does this).

#outside the classes  
c1.printSomethingA()  
c1.printSomethingB()  
print()  
c2.printSomethingA()  
c2.printSomethingB()

I'm implemented by Concrete Class 1  
I'm inherited. You can also override me if you want  
  
I'm implemented by Concrete Class 2  
I'm overriden by Concrete Class 2

# Unified Modelling Language for Class Diagrams

## Introduction

The modelling language we are going to use to represent architecture would be UML or **Unified Modelling Language**. We use this to show the relevant classes in the system including their attributes, methods and relationships with other classes. UML differ a little depending on the source. The syntax we’ll be following in this class would be based on PlantUML.

## Learning Outcomes

1. Create class diagrams to represent OOP architecture in UML
   1. Create classes with attributes and methods in UML
   2. Establish class relationships in UML

## Modelling Classes

The example below shows three classes. One is concrete class called ExampleClass (notice the “C” in the title that denotes it is a concrete class). Another is an abstraction called ExampleAbstraction (denoted by “I” which stands for interface). The last one is ExampleAbstractClass which is an abstract class (denoted by “A”).

Class Diagrams

Class Diagrams

Attributes are placed in the top portion and methods are placed in the bottom portion. The shape to the left of each attribute or method indicates its visibility. Filled shapes indicate visibilities for methods while unfilled shapes indicate visibilities for attributes.

When writing UML for python code, I usually omit writing the self special identifier inside method specifications. It is implied that self is passed always for all non-static methods in python.

|  |  |  |
| --- | --- | --- |
| attribute | method | visibility |
| img | img | private |
| img | img | protected |
| img | img | public |

The names for abstract methods, abstractions, and abstract classes are italicized.

If possible, write the expected type of attributes, parameters and function returns. This is written on the right side of their names, to the right of “:”.

**Disclaimer**: sometimes you’ll find some mistakes in my uml diagrams, sometimes I write the incorrect visibility marker or neglect abstract italicizations. Rest assured the code will contain the correct visibility markers, abstract modifiers and etc.

## Modelling Class Relationships

Here’s a reference arrows that indicate the relationship between two classes:

class relationships

class relationships

Sometimes, instead of writing the specific kind of dependency arrow (aggregation or composition), I just write the general dependency arrow instead

## Optional Readings

PlantUML Class Diagrams. https://plantuml.com/class-diagram Accessed August 31, 2020 # Exceptions

## Introduction

One of the features added to imperative programming was the elegant handling of errors. Exception handling provided OOP a mechanism to control what exactly the system does when parts of the fail.

## Learning Outcomes

1. Create methods that raise errors in python
2. Create a try-except clause to properly react to errors in python
3. Design systems that correctly assign error handling responsibilties

Let’s explore this with an example. In the snippet below, the last line is not reached because the system fails during print(quotientUnsafe(3,0)).

def quotientUnsafe(a:float, b:float) -> float:  
 return a/b  
  
print(quotientUnsafe(3,2))  
print(quotientUnsafe(3,0))  
print("some more behavior")

1.5  
---------------------------------------------------------------------------  
ZeroDivisionError Traceback (most recent call last)  
 in   
 3   
 4 print(quotientUnsafe(3,2))  
----> 5 print(quotientUnsafe(3,0))  
 6 print("some more behavior")  
  
 in quotientUnsafe(a, b)  
 1 def quotientUnsafe(a:float, b:float) -> float:  
----> 2 return a/b  
 3   
 4 print(quotientUnsafe(3,2))  
 5 print(quotientUnsafe(3,0))  
  
ZeroDivisionError: division by zero

Python has its own error raised when it encounter division by zero but well create our own for the sake of learning

Problematic functions and methods like quotient don’t always return a float. The problem for this quotient is that there is a possibility you’ll end up dividing with zero. This introduces the concept of exception. Where the quotient function works **except** when the denominator is zero.

To implement this kind of behavior. You create an if-else check (or any control statement) to make sure the denominator is not zero. If you do encounter a zero denominator you **raise** an error. Here you are raising a user defined error object called DivisionByZeroError.

class DivisionByZeroError(Exception):  
 def \_\_init\_\_(self,numerator:float, denominator:float):  
 self.numerator = numerator  
 self.denominator = denominator  
  
def quotient(a:float, b:float) -> float: #maybe a float  
 if b == 0:  
 raise DivisionByZeroError(a,b)  
 else:  
 return a/b

When quotient is called where the denominator passed is zero, python will inform you that the function call has resulted in a DivisionByZeroError

try:  
 quotient(3,2)  
 quotient(3,0)  
except DivisionByZeroError:  
 print("there was dividing by zero somewhere but it's okay I'm still fine")  
   
print("some more behavior")

there was dividing by zero somewhere but it's okay I'm still fine  
some more behavior

By enclosing the lines of code that could potentially raise errors, you create a safety net for the system. The system **tries** to execute these lines, and if there are no errors raised inside the try block then the system runs normally. Nothing abnormal would occur **except** when the problematic lines of code raises an error. In this specific case, the system is catching, a specific type of error called DivisionByZeroError. If a specific type of error is not specified in the exception block, the system will catch any general error.

If a method with potential to raise errors like quotient(), is invoked inside another method, the caller method becomes a method with potential to raise errors as well.

def mixedFraction(a:float, b: float, c:float) -> float: #maybe a float  
 return a + quotient(b,c)  
  
mixedFraction(1,1,0)

---------------------------------------------------------------------------  
DivisionByZeroError Traceback (most recent call last)  
 in ()  
 2 return a + quotient(b,c)  
 3   
----> 4 mixedFraction(1,1,0)  
  
 in mixedFraction(a, b, c)  
 1 def mixedFraction(a:float, b: float, c:float) -> float: #maybe a float  
----> 2 return a + quotient(b,c)  
 3   
 4 mixedFraction(1,1,0)  
  
 in quotient(a, b)  
 7 def quotient(a:float, b:float) -> float: #maybe a float  
 8 if b == 0:  
----> 9 raise DivisionByZeroError(a,b)  
 10 else:  
 11 return a/b  
  
DivisionByZeroError: (1, 0)

Here, the quotient()’s caller,mixedFraction(), does not enclose the problematic line with a try-catch block. This means that mixedFraction is basically the ignoring any potential error, shifting the responsibility of dealing with the error to wherever mixedFraction() is called.

On the example below, quotient() is invoked inside quotientString(), but instead of ignoring the error, quotientString() deals with it using a try-catch block. When quotient fails, the function returns “undefined number” instead of a fraction string.

def quotientString(a:float,b:float) -> str: #always a string  
 try:  
 wholePart = math.floor(quotient(a,b))  
 decimalPart = quotient(a,b) - wholePart  
 return str(wholePart) + " and " + str(decimalPart)  
 except Exception:  
 return "undefined number"  
print(quotientString(17,7))  
print(quotientString(1,0))  
print(quotientString(1,3))

2 and 0.4285714285714284  
undefined number  
0 and 0.3333333333333333

By dealing with potential errors, quotientString() becomes a safe function that has no potential of breaking the system.

You can catch multiple kinds of exceptions if you want to handle different exceptions differently. The exception IndexError is raised when an iterable type like list accesses a non-existent member. Here the function quotientList wants to append math.inf if you’re dividing by zero and not append anything if you’re dividing with non-existent list members.

def quotientList(l:[float],m:[float]) -> [float]: #always a list of float  
 r = []  
 for i in range(0,len(l)):  
 try:  
 r.append(quotient(l[i],m[i]))  
 except DivisionByZeroError:  
 r.append(math.inf)  
 except IndexError:  
 pass  
 return r  
quotientList([1,2,3,4,5,6],[3,0,2,2,1])

quotientList([1,2,3,4,5,6],[3,0,2,2,1])

### To ignore errors or to deal with errors?

So you’ve seen two ways to react with potential errors, to ignore them like mixedFraction() or to deal with them like quotientString(). Which is the proper way? You might think that dealing with errors is better since it’s this way doesn’t break the system. But actually the choice to ignore or to deal with errors depends on who has the correct responsibility in fixing the error.

If you immediately deal with the error as quickly as possible, you’ll end up missing the importance of raising errors. The method quotient() for example, is responsible for providing the caller with a quotient, that must be this method’s only responsibility. You should not give quotient the responsibility of fixing division by zeroes. That responsibility lies on its caller, because different caller’s may have different ways to deal with the error. The method quotientString() deals with division by zero with “undefined number” while the method quotientList() deals with division by zero with “inf”. These two callers have different ways of interpreting division by zero so they should be the one’s responsible for dealing with the error.

# SOLID Objects

## Introduction

SOLID is an acronym describing design principles for creating OOP systems. By committing your code to these principles you will naturally build systems that don’t only work perfectly, but work elegantly.

## Learning Outcomes

1. Design methods and classes with exactly one responsibility
2. Design extension to classes to incorporate new behavior
3. Design proper realization and specialization relationships
4. Design purposeful abstractions and abstract methods
5. Design systems that interact through abstractions

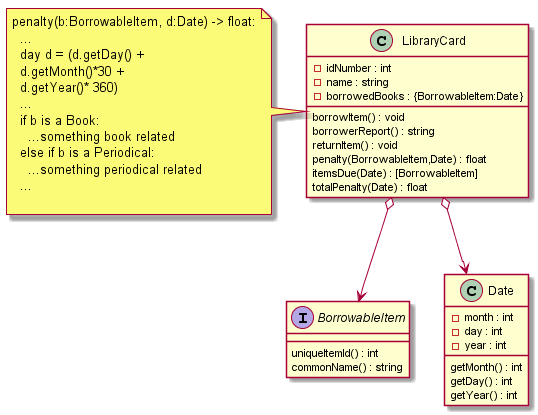
Briefly, these five principles are:

* **Single Responsibility Principle** - Objects should have cohesive and complete responsibilities. It shouldn’t be aware of knowledge it doesn’t need and it shouldn’t perform responsibilities that are irrelevant from it.
* **Open/Closed Principle** - Classes should be open to extension and closed to modification. Instead of changing the form and behavior of an existing class, you should extend the class
* **Liskov-Substitution Principle** - Substituting objects by their subtypes/realizations should always work.
* **Interface Segregation Principle** - A client shouldn’t be forced to implement methods that it doesn’t use
* **Dependency Inversion Principle** - Object relationships should depend on abstractions instead of implementations

### Single Responsibility Principle

#### The GOD Class

One of the canonical examples of violations against SRP is the concept known as the **god class**. A god class is a class that basically contains all the attributes and methods of the whole system. You’ll recognize these god classes as those classes that control the behavior of objects (they contain the implementation of client objects’ behavior). These god classes are also aware of all of the objects secrets (they expose and manipulate private attributes and methods).



god class library card

On the example above LibraryCard is a god class since it exposes the secrets of Date by forcing the creations of *evil* getters (getMonth(), getDay(), getYear()) for otherwise private details. Although it isn’t obvious, it also tampers on the responsibilities of BorrowableItem by deciding by itself how penalty is calculated for each realization.

The moment you ask an object what it’s exact type is, you should consider refactoring your code since introspective checks like isinstance , instanceof typeof and etc., are symptoms of smelly design. You can think of these checks as analogies to racial discrimination since your client object asks the race (type) of the dependency.

#### Assigning the correct responsibilities

When you’re introducing new behavior or information to a system, you should ask first: *who should be responsible of this behavior or information?*

* Who should be responsible of calculating the differences between dates? **Date should be responsible, not LibraryCard**.
* Who should be responsible of calculating the penalties of specific BorrowableItem realizations? **BorrowableItem realizations should be responsible, not LibraryCard**

The best implementation of the system contains multiple examples of SRP. The Date class should be responsible of subtracting dates and adding dates, not the LibraryCard class. Forcing LibraryCard to contain code for Date operations will force you into breaking the boundaries of your classes (e.g. creating getters to expose private attributes). LibraryCard should not be aware of **how** dates are subtracted to be able to subtract dates. The same is true for a BorrowableItem’s due date. The LibraryCard shouldn’t contain the specifications as to how BorrowableItems calculate their due date. In fact LibraryCard shouldn’t even be aware of the exact type of the BorrowableItem (isinstance violates this).

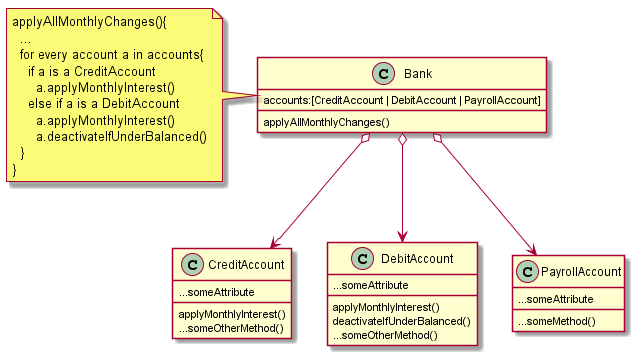
The process of designing these cohesive systems requires not only OOP design techniques but also domain knowledge. The designer of the system, should know how a library system operates so that, he/she can accurately simulate their responsibilities on code.

You can also apply SRP on individual methods inside objects. Methods should be responsible of one thing only. Keeping methods pure like these will help reduce unwanted side effects. The builder/manipulator naming scheme will help you with this. Also a method should not be responsible of handling the problems. The method should delegate that responsibility to the clients of that method. The methods itself should only report the problem. The best way to do this in OOP is by raising an exception.

### Open/Closed Principle

A good class in OOP is both open and closed. It is open for extension but closed for modification.

To understand this principle lets have an example of a system that is closed for extension:

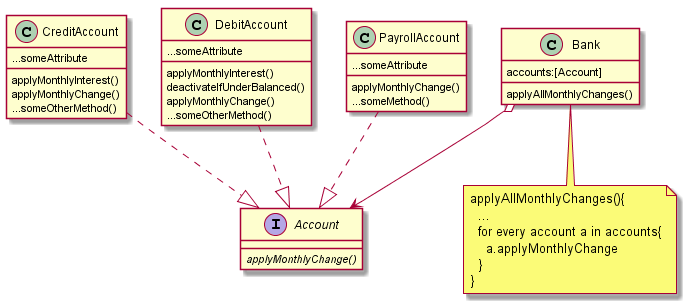


god class library card

Forgive the long Java-like method names, they’re named as descriptive as possible so that I can s skip actually explaining what they do.

This is a system that indeed works perfectly. The bank will be able to apply the appropriate changes to the account types because of the if-else block that segregates the accounts based on its type (another instance of type discrimination so that’s a hint that this is wrong). The problem with this is that whenever there are changes regarding the monthly changes or interest calculations, you would have to tamper with the contents of bank. This is **opening Bank up for modification**. Every time there’s a change related to monthly updates you would have to do some kind of surgical procedure on Bank and rearrange its internal organs so that the change may be supported. This is extra rough on Bank because Bank shouldn’t even be responsible for these behaviors (a violation of SRP). If there are new types of account, then you would have to open up Bank again and to add another else if block. Poor Bank, who knows how many more new types of accounts there are in the future.

Instead of rearranging the organs of your classes to accommodate changes to behavior they are not even responsible for, you should close the classes for modification and open them for extension instead:



open for extension

Since applyMonthlyChange() is an abstract method of account, all its realizations are required to implement it. We extend the functionality of CreditAccount, DebitAccount, and PayrollAccount by adding an extra method.

* inside CreditAccount.applyMonthlyChange() you just call applyMonthlyInterest(),
* inside DebitAccount.applyMonthlyChange() you call both applyMontlyInterest() and deactivateIfUnderBalanced()
* inside PayrollAccount.applyMonthlyChange() you do nothing

Is a method that does nothing inelegant? Not really because this is an accurate representation as to how a PayrollAccount changes every month— it doesn’t. Also, in the future, the inertness of PayrollAccount may change so at least you have the function prepared.

The previous library card example and bank examples are indeed similar . This is because introspective checks like isinstance are again symptoms of inelegant design.

Another example of this is how you need to augment the PayrollAccount class to contain the recieveFund(float) or deposit(float) method so that it can become a recipient for transfer.

Closed for modification does not mean you cannot change literally anything in the class. Of course if there are mistakes in the specific behavior of the class then you have to modify it.

### Liskov-Substitution Principle

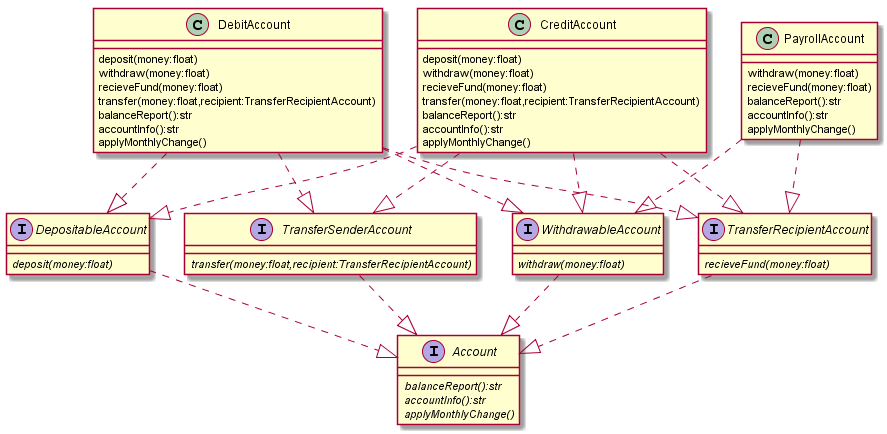
This principle basically dictates when should an object be a subtype or a realization of another object. Should PayrollAccount be a realization of Account? If you can substitute any instance of an Accountwith a PayrollAccount then the answer is yes. The same is true if you want to establish an inheritance relationship between Account and PayrollAccount. The LSP is important because it ensures the polymorphic capabilities of your realizations and subtypes.

### Interface Segregation Principle

Sometimes the subtypes/realizations of a certain object may have diverse functionality. Some subtypes can deposit, some subtypes can’t, all subtypes can be recipient for transfers but not all can be senders. The diversity of functionality supportability may sometimes force the designer to pollute the system with methods that the subtypes don’t actually use. PayrollAccount will not use deposit but since it realizes Account then we reluctantly add it. This is a violation if LSP which states that an object should not be forced to implement methods it doesn’t use.

#### Role Interfaces

The best way to design these diverse systems it by refactoring your architecture to have diverse role interfaces instead.



god class library card

Now instead of cluttering you realizations with useless methods, your systems is now cluttered with role interfaces. This is a benevolent kind of clutter. Because more objects and looser dependencies make for a maintainable and therefore elegant system (in the same way a language with more words have less chances for ambiguity). Because of this clutter you can have rich polymorphism without sacrificing ISP. Although be careful not to over do it though. You wouldn’t want a role interface for every conceivable method out there.

When you have role interfaces, you can refine lines of your code by describing which exact role interface applies. For example, instead if writing transfer to be an process which is Account to Account they are TransferSenderAccount to TransferRecipientAccount.

### Dependency Inversion Principle

The relationships between objects should be defined by surface instead of their interior. This means that How a bank interacts with an account should not be dependent on the implementation. The way Bank prints the balance report should not be by directly concatenating "Your balance is" + account.balance . Bank should interact with an account by printing the return value the abstract builder method, balanceReport(). The class Bank should not dissect an Account to retrieve the balance. It should tell the Account to build a balance report.

## Optional Reading

Bailey D., (2009) [SOLID Development Principles – In Motivational Pictures](https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/). Accessed August 31, 2020 # Extra Stuff

Here are extra things I want to discuss that do not really fall into specific lectures.

## Naming Methods Elegantly

One of the important things to consider about coding in OOP (and in general) is properly naming identifiers, functions, and classes. I’m sure this was talked about in your early programming languages. Creating names is a programming skill in and of itself. Creating concise (unlike Java’s verbose naming schemes), descriptive names help in the maintainability of the code you’re writing. When the name itself tells you what an identifier is for and when the name of the modifier tells you what it does, then you don’t need to explain it in documentation or comments.

Since you’re probably experts in naming identifiers from you C coding experience, lets focus on naming methods.

**Disclaimer**: These naming conventions are according to MY standards of code elegance. I’m not saying that this particular way of naming is the correct way, to be honest these standards are more subjective than objective. While your in my course I hope you’ll at least try incorporate these conventions in your code. I don’t care about how you write names in other course, but in my course, these are the names that I consider to be *beautiful*.

Sometimes, even I forget to adhere to my own conventions, especially when I’m writing code outside OOP. You’ll probably notice some contradictions to these conventions somewhere in this course. Just consider these contradictions as proof that the instructor that made these resources is indeed a human being. Let me offer you a preemptive *whoops* for all of those contradictions.

Methods (and functions in general) can be divided into two types, methods that return something, known as **builders**, and methods that do not return something, known as **manipulators**.

### Builders

An inelegant method name:

def add(addend1:int, addend2:int) -> int:  
 return addend1 + addend2

Now this seems like a descriptive enough name that describes what the method does. What is wrong with it? The issues start to surface once you invoke the function:

solution = 5 - add(3,2)

When you see this line of code, your brain reads this as:

“solution becomes five minus add three and two”

Which can still be understood since we’re used to what the method add() usually means. But consider this alternative way of naming this function:

def sum(addend1:int, addend2:int) -> int:  
 return addend1 + addend2

When called:

solution = 5 - sum(3,2)

Your brain reads this as:

“solution becomes 5 minus sum of 3 and 2”

Which makes way more sense.

One of the most egregious violations to this convention is the naming of **getter**. Like the following

#inside some class  
def getName(self):  
 return self.\_\_name

print("Hello " + p.getName())

Getter functions are usually created to reveal the values of private attributes. First of all getters, are a bit evil and must be avoided as much as possible since they violate the privacy of private attributes. It’s supposed to be a private attribute, other classes are not supposed to access.

But sometimes exposing these attributes are needed. Sometimes, you don’t need to be able to change the value of the private attribute, you just need to know its value. The best way to name a getter is the following.

#inside some class  
def name(self):  
 return self.\_\_name

print("Hello " + p.name())

People might complain, that name() and \_\_name are ambiguous. But for me this ambiguity is fine since they end up meaning the same thing. Plus, they will never be used interchangeably (maybe in functional paradigm they may) because one is a method and the other is an attribute.

So the rule of the thumb that you need to follow when you are naming builders is the following, **A method that returns something (a builder), should be named after a noun that describe what it is returning.**

#### Some more builders named elegantly

This is a factory method, which is responsible for building new instances of a class called person. Therefore this method is called newPerson()

#inside some class  
def newPerson(self) -> Person:  
 return Person()

This method returns the element of a given list at position

def kthElement(list:[float],k:int) -> float:  
 return list[k]

This is a method that returns the concatenation of two strings

def concatenation(u:str, v: str) -> str:  
 return u + v

### Manipulators

This is an example of an inelegantly named manipulator:

#inside some class  
def nameString(self):  
 print(self.\_\_name)

When your brain reads an invocation of this method it appears out of place:

if (value > threshold):  
 nameString()  
else:  
 print("value is invalid")

A better name for this function is the following:

#inside some class  
def printName(self):  
 print(self.\_\_name)

if (value > threshold):  
 p.printName()  
else:  
 print("value is invalid")

The original name nameString() is inelegant because it is named like a builder, when in fact it doesn’t build anything. It returns nothing. This method an example of a **manipulator**. It should be named like a manipulator. This particular method manipulates the output stream of wherever you are printing.

**Setter** methods are actually named correctly, setter methods are methods that set the value of a specific private attribute:

#inside some class  
def setName(self, newValue:str):  
 self.\_\_name = newValue

Setters are also quite evil in the same way getters are evil. These methods change the values of private methods, violating their privacy. Although the usage of setters is unavoidable in some design patterns, avoid them as much as you can. But at least its name follows this convention for naming manipulators.

The rule of thumb for naming manipulators is the following. **A method that doesn’t return anything should be a manipulator and must be named from the verb that describes the its manipulation**.

#### Some more manipulators named elegantly

A method that advances time by 5 seconds:

#inside some class  
def skipForward(self):  
 self.\_\_time = self.\_\_time + 5

A method that removes the first element in the list

#inside some class  
def decapitate(self):  
 self.\_\_list = self.\_\_list[1::]

It’s called decapitate because it removes the head of he list. Sometimes, creative and descriptive method names like this help you remember what they do, just because they are accurately named but still memorable

### Predicates

def isPositive(n:int) -> bool:  
 return n > 0

Predicates are methods that return either true or false. Although these methods are technically builders, they follow a different naming convention. These methods are named like predicates in a sentence. Example:

Checks if a word is capitalized

def isCapitalized(word:str) -> bool:  
 return word[0].isUpper()

Checks is a list is empty

def isEmpty(list:[int]) -> bool:  
 return len(list) == 0

The reason why predicates are named like this is because of how they are used inside if statements. The following is way more readable:

if (isPositive(n)):  
 return n  
else:  
 return -n

“If n is positive then …”

Compared to this

if(positivity(n)):  
 return n  
else:  
 return -n

“If positivity of n then …”

### Single Responsibility Principle and correct naming

The correct naming of methods helps with one of OOP’s design principle, SRP. The name of the function describes the one thing it does. For example the method sum() does exactly one thing, it builds the sum of two numbers. The method decapitate() does exactly one thing, it removes the head of the list. Whenever you encounter a function that gives you trouble in deciding a name because it is both a manipulator and a builder then it means that the method has more than one responsibility. This means that you should break down this method into smaller methods that does exactly one thing.

The following method should be broken down

def decapitateAndHead(self):  
 head = self.\_\_list[0]  
 self.\_\_list = self.\_\_list[1::]  
 return head

Into these two methods:

def decapitate(self):  
 self.\_\_list = self.\_\_list[1::]

def head(self) -> int:  
 return self.\_\_list[0]

## The special function \_\_str\_\_()

You will sometimes encounter the following function in the lab exercises. When you add this function inside your class, it makes the instances of that class *stringable* and *printable*. This means that the class now has a string representation, which means that instances of the class can be easily converted to string and can be printed directly using print().

For example, given a person class:

class Person:  
 def \_\_init\_\_(self,name:str,age:int):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self) -> str:  
 return self.name + " " + str(self.age)

By implementing the \_\_str\_\_() method your class instances can now be treated like a stringable or printable instances:

\_\_str\_\_() should always accept nothing (but the reference to self) as a parameter and return a string.

**Printing the instance itself**

When a printable type is placed inside the print function and nothing else, it automatically converts it as a string and prints it

print(Person("Cheems",29))

Cheems 29

The string above is printed since this is the value that Persons‘s’ \_\_str\_\_() method returns.

**Conversion to string**

A stringable instance can be converted to a string using the builtin str() function:

print("Hi I'm " + str(Person("Cheems",29)) + " years old")

Hi I'm Cheems 29 years old

This will also work for formatted strings if you use the sigil %s:

print("Hi I'm %s years old" % Person("Cheems",29))

Hi I'm Cheems 29 years old

# Design Patterns Introduction

## Introduction

Design patterns, are general, reusable solutions to a commonly occurring problem within a given context in software design. Unlike algorithms, design patterns are not clear instructions that can automatically be transferred to your system. Design patterns are more like templates that describe the general concept to solve the problem. It doesn’t contain implementation details; it contains structural blueprints.

## Learning Outcomes

1. Discuss the origins of design patterns in OOP
2. Explain the advantages of design patterns
3. Explain the disadvantages of design patterns
4. Identify the three classifications of design patterns

## History of Design Patterns

Design patterns are not novel and sophisticated discoveries, they are instead, typical solutions to common problems. The pattern of these solutions become so ubiquitous that it becomes worthwhile to put a name to it. Design patterns in software engineering are just borrowed concepts from architecture/design.

The concept of design patterns is often attributed to Christopher Alexander, from his book, *A Pattern Language: Towns, Buildings, Construction* [[8]](#footnote-231). These patterns may describe how high windows should be, how many levels a building should have, how large green areas in a neighborhood are supposed to be, and so on.

Four software engineers, Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm, used this as an inspiration to publish the famous book, *Design Patterns: Elements of Reusable Object-Oriented Software.* [[9]](#footnote-232) The four became collectively known as the “**Gang of Four”**. And their book became known as the GoF book. It contains a catalog of 23 design patterns solving various problems of OOP design.

## Why Patterns?

The answer to this problem is similar to the reason as to why you don’t “reinvent the wheel”. Design patterns are tried and tested solutions, knowing these patterns give programmers a toolset to solve a variety of problems in software design.

Design patterns also help with communication. A team of software engineers well versed in design patterns wouldn’t need to explain to each other what exactly must be done to use an “Adapter pattern”.

## Why not Patterns?

Design patterns are sometimes used to simulate features that the programming language doesn’t have. If you use a powerful enough language you wouldn’t need the pattern at all. Example of this is how the Strategy pattern can be replaced by lambdas.

Patterns are not end-all be-all solutions to any design problem out there. At the end of the day context matters the most. An inexperienced programmer will implement a problem to the dot, instead of adapting the pattern for the context. Patterns are not end-all be-all solutions to any design problem out there. At the end of the day context matters the most.

Sometimes, you don’t even need a pattern at all. A simple problem solved using a complicated solution is inelegant.

## Classifications of Design Patterns

* **Creational Patterns** provide object creation mechanisms that increase flexibility and reuse of existing code
* **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
* **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

## Optional Reading

Gamma, Vlissides, Johnson, and Helm (1994). Design Patterns: Elements of Reusable Object-Oriented Software # Creational Patterns

## Introduction

Almost all programming languages with object oriented support provides you rich features in creating instances of classes using constructor methods. Inside the constructors you can add business logic to initialize objects and make them ready for use. Some programming languages (like Java or C++) even have the capability to have more than one constructor method so that instances can be shipped with different states depending on the chosen constructor.

Unfortunately native constructors capabilities are not powerful enough for our standards of elegance. Some systems have complicated object production mechanisms that require extra capabilities. Sometimes classes have too many attributes for a simple constructors. Sometimes object creation require polymorphic support and decoupling against the exact subtypes or realizations. Sometimes you need to ensure that certain classes have exactly one instance throughout the lifetime of your system.

## Learning Outcomes

1. Design systems that apply the factory method design pattern
2. Design systems that apply the abstract factory pattern

## Factory Method Pattern

### Problem

The exact type of the dependency (a product) created and used by some client (a factory) is decided by a client of that factory. Somewhere, inside this factory class, a specific product is being instantiated and maybe used (this instantiation happens maybe more than once). But, as it turns out, there are different types of products, (there’s also the possibility of more product types in the future). You can change the code of the factory class to accommodate multiple product types. For every product, you modify the factory and add some if else clause to produce the correct product type.

As you see this process is quite tedious. For every new product type that is added to your system, you perform surgery to the factory class. This process will end up forcing you to create smelly if-else checks to switch to the correct product type.



Factory Method

### Solution

You encapsulate the creation of a class inside a **factory method** that is specified to return an abstraction of the product. If there are other real product types that have to be produced, you create a specialized factory which overrides the factory method.

factory method class diagram

factory method class diagram

Somewhere inside factory you have one or more instances of creating or using the product.

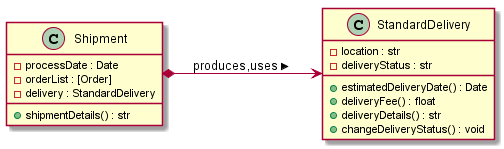
If you choose to build Factory as a concrete superclass, the factory method inside the Factory should return some realization of Product. This is the default product returned by any Factory. If you need to return a different Product realization, you override the factory method to return that particular Product realization.

### Example

#### Online Marketplace Delivery

Consider you’re developing the product delivery side of an online marketplace app (think Amazon/Lazada). Your app is on its early stage so their is only one delivery option, standard nationwide delivery that takes a minimum of 7 days.

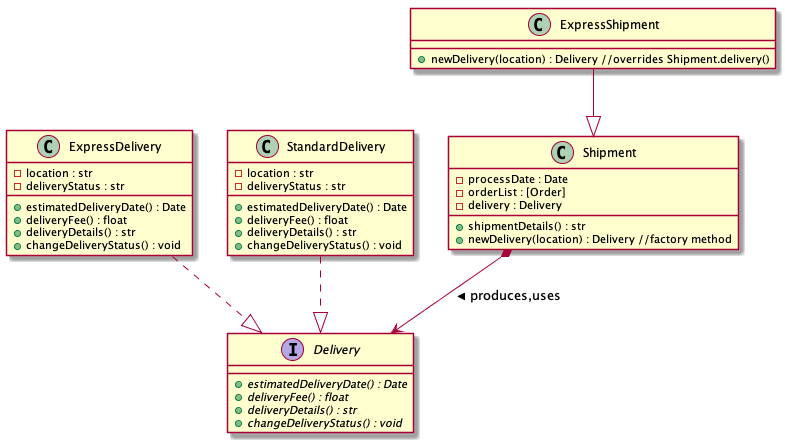
What you have is Shipment class that contains a StandardDelivery class. Inside the shipment class is the shipmentDetails() builder which builds a string representing the details of the shipment, this includes the delivery details (which requires access to the composed StandardDelivery instance). Inside the constructor of Shipment an instance of StandardDelivery is created so that every Shipment is set to be delivered using standard delivery.



online marketplace

This system does work. It works but it is still inelegant. As soon as your app grows, you will incorporate new delivery options like express delivery, or pickups or whatever. Every time you need to add a new delivery method you will need to perform surgery in Shipment since the StandardDelivery instance is created inside the constructor of Shipment. Shipment’s code is too coupled with StandardDelivery.

To solve this you need to implement the factory method pattern. Right now shipment is a factory since it constructs its own instance of StandardDelivery. To refactor this into elegant code, you need to so create an abstraction calledDelivery first to support polymorphism. Inside Shipment instead of creating instances of Delivery’s using a constructor, you invoke a factory method that encapsulates the instantiation of Delivery. In this case we name this method newDelivery(). All it does is return an instance of StandardDelivery using its constructor.



online marketplace

In this new architecture, whenever there are new delivery methods a shipment could have, all you have to do is to create a realization of that delivery method. In this case the new delivery method is ExpressDelivery which delivers for two days but is twice as expensive. And instead of changing Shipment (violates Open/Closed Principle), you make an extension to Shipment. This extension is the specialization to shipment called ExpressShipment (a shipment that uses express delivery). In this specialization, you only need to override the factory method delivery, so that every instance of delivery construction creates ExpressDelivery. The difference between ExpressDelivery and Delivery is that ExpressDelivery has a delivery fee of 1000 and the estimated delivery date is 1 day after the processing date.

### Why this is elegant

* **Single Responsibility Principle** - the extra level of encapsulation on the construction of the product (factory method), allows the factory to be responsible of creating the exact product type it needs.
* **Open/Closed Principle** - instead of modifying the factory to incorporate the creation of different product realizations, you instead create an extension of the factory. No need for introspective checks since the factory method supports polymorphism of the product it creates.
* *Encapsulate what varies* - This pattern upholds one of OOP paradigms most important principles. Since the construction of product varies from product type to product type, it is encapsulated into the factory method.
* This avoids tight coupling between the factory and the product

Coupled classes are classes which are very dependent on each other. Changing the code of one will most likely affect the other

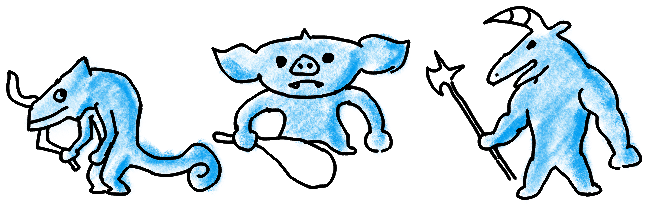
### How to implement it:

1. Create an abstraction for all product types (Product).
2. Inside the base factory create the the factory method function newProduct():Product. Make sure it is specified to return abstraction Product.
3. For every new product type that is added to the system, create 2 new classes: the new product type as a realization of Product and, and the factory for the new product type as a realization of Factory.
4. Inside each factory specialization override newProduct() to return the correct realization of Product.
5. Replace every instance of constructor calls inside Factory with a call to the factory method newProduct().

## Abstract Factory

### Problem

Your system consists of a family of related products. These products also have different variants. You need a way to create these products so that the products match the the same variant. The exact variants of the family of products are decided during runtime, somewhere else in the code (similar to product creation in a factory method)



Abstract Factory

### Solution

You create different kinds of factories that realize under the same abstract factory. The exact type of factory will decide the variant of the family of products that are created. To do this you need to create different factory methods for each product. These factory methods must be abstract methods in the abstract factory so that every factory realization can create all members of the product family.

abstract factory

abstract factory

The family of products, are ProductA and ProductB, These products come in two variants, variant 1 and 2. FactoryVariant1 is a realization of Factory which creates all of the product in variant 1 while FactoryVariant2 creates all the products in variant 2.

If it makes sense for the system you can make an abstract Product class for all the types of products.

When the client of an abstract factory produces its products, it doesn’t need to know what kind of factory is producing the products. This means that the concrete type of a product (its variant) is not decided during compile time but instead it depends on the concrete type of the factory that is creating it.

### Example

#### Bootleg Text-based Zelda Game

You’re creating the dungeon encounter mechanics of some bootleg text-based zelda game. In this game,every time you enter a dungeon, you encounter 0-8 monsters (the exact number is randomly determined). There are 3 types of monsters, bokoblins, moblins, and lizalflos (different types have different moves). The exact type of monster is randomly decided as well.

Right now the game works like this:

As soon as you enter the dungeon, all the enemies are announced:

5 monsters appeared  
A lizalflos appeared  
A lizalflos appeared  
A lizalflos appeared  
A moblin appeared  
A moblin appeared

After this, each enemy in the encounter attacks. They randomly pick an attack from their moveset.

Lizalflos thorws its lizal boomerang at you for 2 damage  
Lizalflos thorws its lizal boomerang at you for 2 damage  
Lizalflos camouflages itself  
Moblin stabs you with a spear for 3 damage  
Moblin stabs you with a spear for 3 damage

The encounter ends with Link dying since you haven’t coded anything past this part.

You decide to make things exciting for your game by adding harder dungeons, medium dungeon and hard dungeon.

##### Medium dungeon

Instead of encountering, normal monsters you encounter stronger versions of the monsters, these monsters are blue colored:

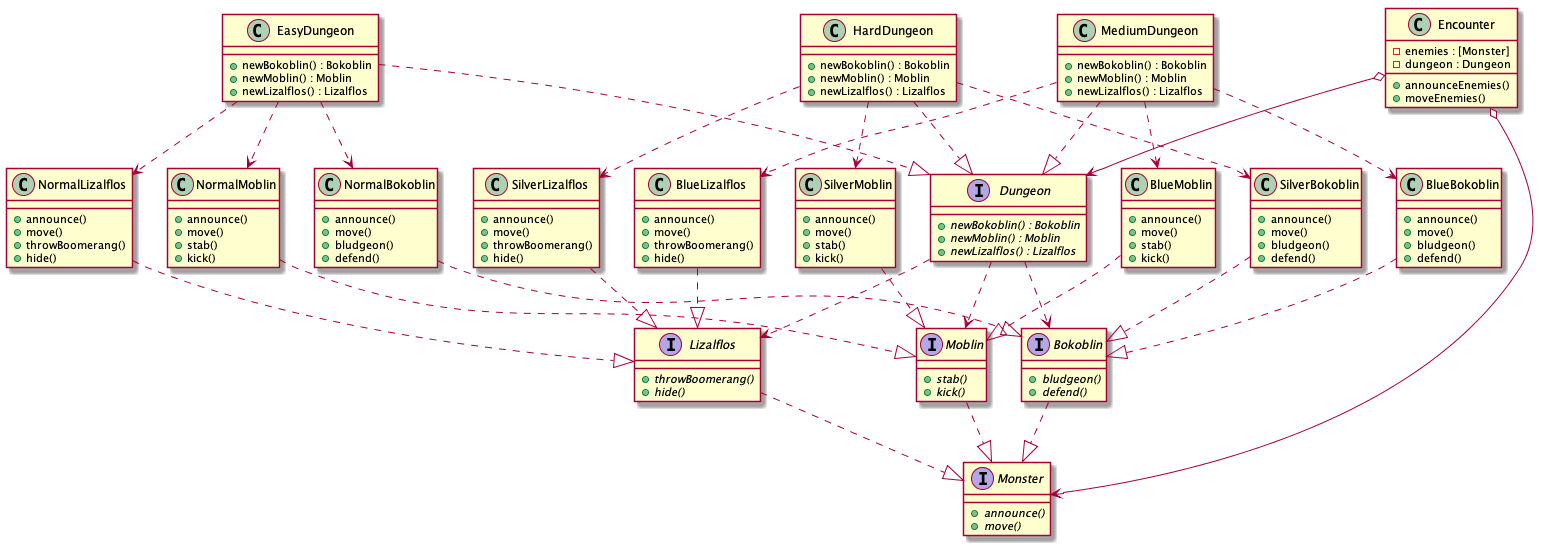
* **Blue Bokoblin**
  + equipped with a spiked boko club and a spiked boko shield
  + bludgeon deals 2 damage
* **Blue Moblin**
  + equipped with rusty halberd
  + stab deals 5 damage
  + kick deals 2 damage
* **Blue Lizalflos**
  + equipped with a forked boomerang
  + throw boomerang deals 3 damage

##### Hard dungeon

These monsters are silver colored extra stronger versions of the monsters

* **Silver Bokoblin**
  + equipped with a dragonbone boko club and a dragonbone boko shield
  + bludgeon deals 5 damage
* **Silver Moblin**
  + equipped with knight’s halberd
  + stab deals 10 damage
  + kick deals 3 damage
* **Silver Lizalflos**
  + equipped with a tri-boomerang
  + throw boomerang deals 7 damage

To seamlessly incorporate these harder monsters in your system, you need to create an abstract factory for each dungeon difficulty. There are now three variants for each monster. For every variant, there is a factory that spawns new instances of each monster.



abstract factory example

### Why this is elegant

* **Open/Closed Principle** - This solution is easier to maintain since you can add more variants of Product without touching any existing code. All you have to do is to add new realization for Product and a new realization AbstractFactory
* Changing the form and behavior of specific variants are isolated since its creation is abstracted.
* You can easily switch between variants by swapping out the factory.

### How to implement it:

* For every product in the family of products, create an abstraction of it (ProductA, ProductB).
* For every variant of the products, create a factory, (FactoryVariant1, FactoryVariant2). These factories must realize under an abstract Factory. The factory should contain abstract factory methods for each product,
* Inside every factory implement all factory methods.

## Singleton (Optional Read)

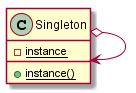
### Problem

Sometimes it wouldn’t make sense for a class to have more than one instance in the lifetime of the application. These things are called singletons.

### Solution

Inside the singleton. Create a static attribute that represents the singleton. Since it is static all instances of the class will share this value. Create a builder to lazily instantiate the value of the singleton and expose the value of the instance.

Disallow the usage of the normal constructor as much as possible. To access the shared static instance, use the builder.



singleton

Disallowing the creation of a singleton depends on the language you use, you can set the constructor to private, or you can raise an error if you try to use the constructor outside the instance builder.

You can also choose to not disallow the use of the constructor, as long as you trust the users to always use the instance builder instead.

### Example

#### A Catalog of Globals

It doesn’t make sense for you to keep multiple copies of global variables in your application, so you decide to place them in a singleton class.

#### Why this is NOT is elegant

A singleton pattern is actually hated by most developers. Yes you can ensure that there is exactly one instance of a class, but its advantages come with a lot of drawbacks.

* You can ensure single instance classes, just by being vigilant.
* Singletons require the use of static attributes and methods. Statics are anti-pattern because they are global variables and manipulators that can have invisible changes to state.
* Singletons are usually symptoms of bad design.

## Optional Reading

Shvets A. (2018) [Creational Patterns](https://sourcemaking.com/design_patterns/creational_patterns) Accessed August 31, 2020 # Behavioral Patterns

## Introduction

Some systems require complex and extremely decoupled relationships. Behavioral patterns are used on these tightly interconnected systems so that they are easier to maintain. These patterns separate behavioral responsibilities among the classes in your system in such a way that volatile behaviors are encapsulated deep into your object structure.

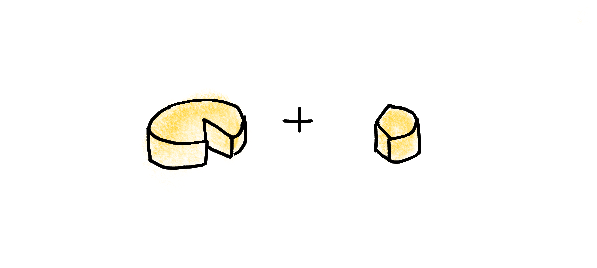
## Learning Outcomes

1. Design systems that apply the strategy pattern
2. Design systems that apply the state pattern
3. Design systems that apply the command pattern
4. Design systems that apply the observer pattern
5. Design systems that apply the template pattern
6. Design systems that apply the iterator pattern

## Strategy pattern

### Problem

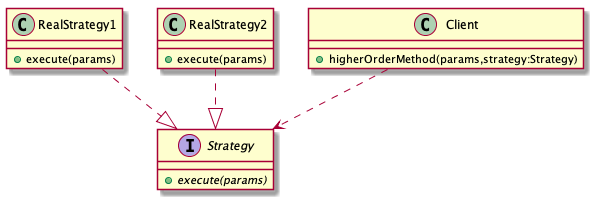
Some systems require behavior that have to be parametrized for other behavior. This is easily done in a functional programming environment since higher order functions are used to represent these. In programming languages that don’t support these features, the strategy pattern is used.



strategy

### Solution

Functions that are not first class citizens are encapsulated inside a Strategy class. A strategy class simply contains the method execute(params), which represents the behavior that should be passed into a higher order function. Any method that can be passed into the higher order function should realize Strategy.



Strategy pattern

The object params represent the data that you need to pass into the correct class. In this pattern you pass the the whole Strategy realization so that strategy.execute(params) perform the desired behavior. You can add other methods in the Strategy abstraction, if it makes sense for the system.

### Example

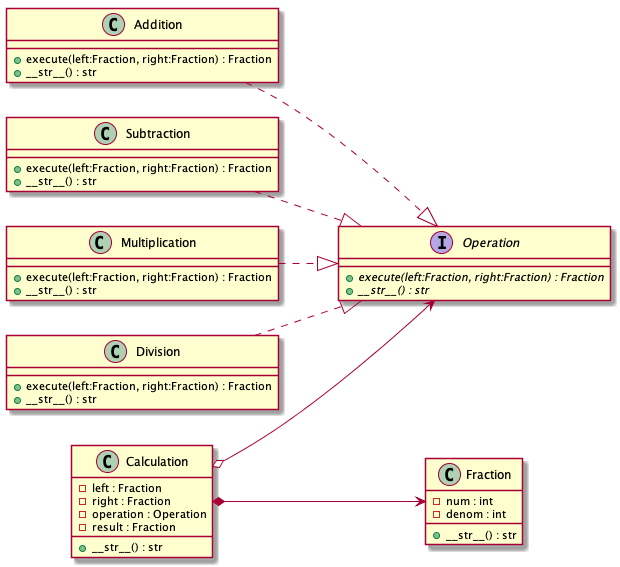
#### Fraction Calculations

You’re creating a less sophisticated version of a fraction calculator. This calculator only has arithmetic operations inside it, addition, subtraction, division, and multiplication. Inside this calculator, a calculation is represented in a Calculation instance. Every calculation has four parts:

* \_\_left - represents the left operand fraction
* \_\_right - represents the right operand fraction
* \_\_operation - represents the operation (,,,)
* \_\_answer - represents the solution of the operation

Python does indeed support higher order functions but your boss is anti-functional programming so he forbids the use these features. Because of this you decide to implement the strategy pattern.

To do this, you need to create an abstraction called Operation to represent the different operations. For each operation, you create a class that realizes Operation.



strategy pattern example

execute() should have been named like a builder method (something like solution()), I’m keeping the name execute() since this is how Strategy patterns usually names this particular method.

### Why this is elegant

* **Open/Closed Principle** - If you want to add new strategies, you wouldn’t need to touch any existing code.
* The implementation of a strategy is deeply tucked inside multiple layers of encapsulation. Changing these implementations is very easy.
* You can swap strategies during runtime in the same way you do in functional programming.

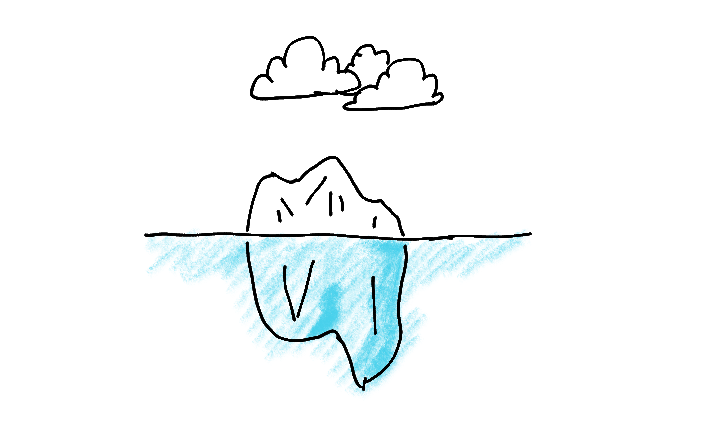
### How to implement it

1. Create an abstract Strategy that contains an abstract method called execute. This method should be specified to accept all the necessary parameters needed by your parameterized function.
2. For every strategy, the higher order method can accept, you create a realization for Strategy and implement the correct behavior in execute.
3. The higher order function should now be specified to accept a strategy of type Strategy.
4. Inside the higher order function, whenever it wants to perform the strategies embedded behavior, call strategy.execute(...).

## State Pattern

### Problem

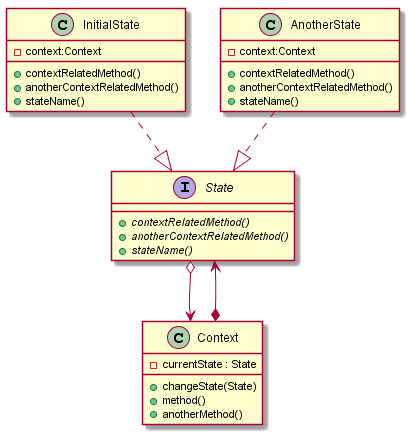
Some objects can change into many different states. If different objects behavior is dependent on its current state, it would require, bulky and annoying if-else blocks to handle its dynamic behavior.



State

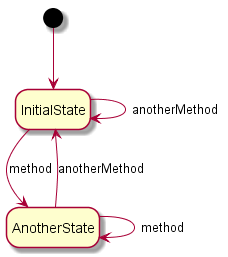
### Solution

An object that can have many states should contain an attribute representing its state. Instead of performing, state dependent behavior directly inside the object, you delegate this responsibility to its embedded state instead. In this way the object will behave according to its current state.



state pattern

The state may be required to contain backreference to the context object that owns it. This is only required if state methods requires to access/control the context that owns it.



state diagram

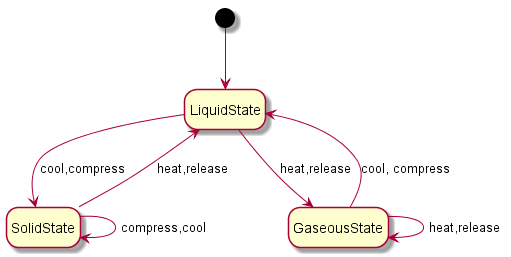
### Example

#### States of matter

The state of any given matter is dependent on the pressure and temperature of its environment. If you heat up some liquid enough it will turn to gas, if you compress it enough it will become solid.

You are to build a less sophisticated version of this model in code. Matter comes in three states, solid, liquid, and gas. The state of the matter may change if you put/remove pressure on it or heat/cool it.

The state diagram would look something like this:

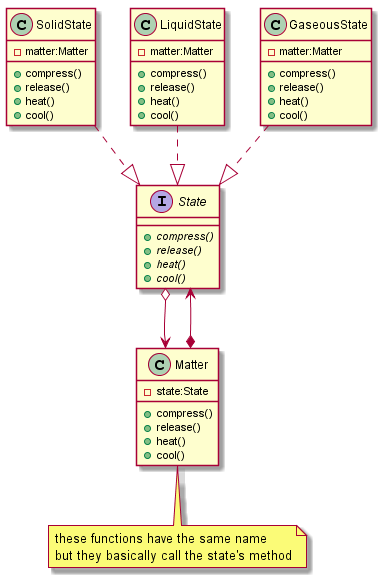


state diagram example

To implement something like this, you would need to create matter which owns an attribute called state which represents the matter’s current state. Since there are three states, you create three realizations to a common abstraction to state.

When you compress/release/cool/heat the matter, you delegate the appropriate behavior and state change inside state’s version of that. Each State realization will need a backreference to the Matter that owns it so that it can change it’s state.

Delegating behavior to the composed state means that, when the Matter instances invoke, compress(), relaease() heat(), and cool(), the composed State owned by the matter calls its own version of compress(), relaease() heat(), and cool().



state example

Matter owns an instance of State, and that instance has an attribute called matter. The attribute matter is the reference to the instance of Matter that owns it. The State instance needs this reference so that it can change the matter’s state when it is compressed, released, heated, or cooled.

### Why this is elegant

* **Single Responsibility Principle** - behavior related to state is delegated to the state itself.
* **Open/Closed Principle** - You can incorporate new states to the system without touching any existing client code
* Implementing this pattern will remove bulky and annoying state conditionals

### How to implement it

1. Create an abstract State that contains abstract methods for all state dependent behavior (context related behaviors that are dependent on context’s state).
2. For every state the Context can have, create a realization of State.
3. Context owns an attribute that represents the current state (currentState) that it owns.
4. If the state needs to control the Context instance that owns it, add a backreference to Context inside state.
5. Whenever a Context instance performs state dependent methods, it calls currentState.contextRelatedMethod() instead so that its behavior is dependent on its current state.

## Command Pattern

### Problem

Sometimes, object behavior contain complicated constraints. Sometimes, the system require the behavior to be invoked by an object but performed by another (this is common in presentation layer/domain layer separation in MVC enterprise systems). Sometimes, the system requires behavior to be undone. Sometimes the system requires a history of the behaviors that were being performed.



Command (What does this strange picture mean?)

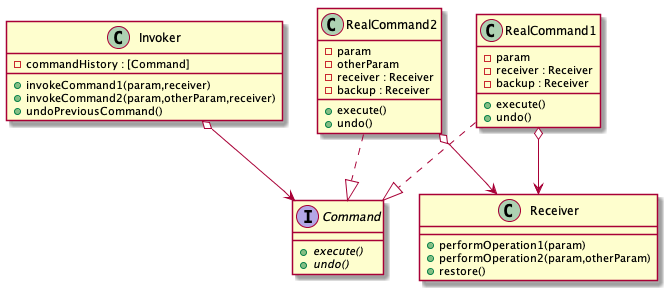
### Solution

These problems have a common solution, the **command pattern**. A Command is a more powerful version of a Strategy. While both of them encapsulates behavior, a Strategy is just that, a function wrapper. A Command on the other hand contains which object performs the behavior, which parameters are needed to perform the behavior, and how to undo the command (if needed).

Creating Commands, allow for more flexible behavior responsibility assignments. A separate Invoker object triggers the behavior by creating a command. This Invoker prepares the command with the appropriate Reciever (the object performing the command), and the appropriate parameters. The invoker then executes the prepared command. The command doesn’t actually do anything, it just tells the Reciever to call the appropriate method.

This separation of responsibility allows for the creation of extra features that may be required for your system:

* If you want to keep a history of the performed commands, the Invoker may keep a list of Commands. This way the data stored in the list history, is a perfect representation of the previous commands.
* If you want the Commands to be undoable, you can store a backup of the receiver (and other affected objects) by the Command inside each instance of Command . Undoing a command will be as simple as restoring the receiver to its backup.



command pattern

Instead of passing the receiver in the Invoker methods, you can create an attribute called receiver inside Invoker. But doing this will make it so there is one Receiver instance for every Invoker instance.

The commands should only affect the receiver. If the behavior that is performed changes a lot of objects, then make a Receiver class that encapsulates all of the affected objects. Doing this will make the implementation of undo easier since the backup inside of the command will simply be an older version of Receiver.

Different command realizations are not necessarily of the sameStrategy. That’s why the parameters of the behavior are stored as attributes of the command, not passed in the execute() function. This is so that no matter what the command is, all execute() functions will have the same type signatures.

### Example

#### Zooming through a maze

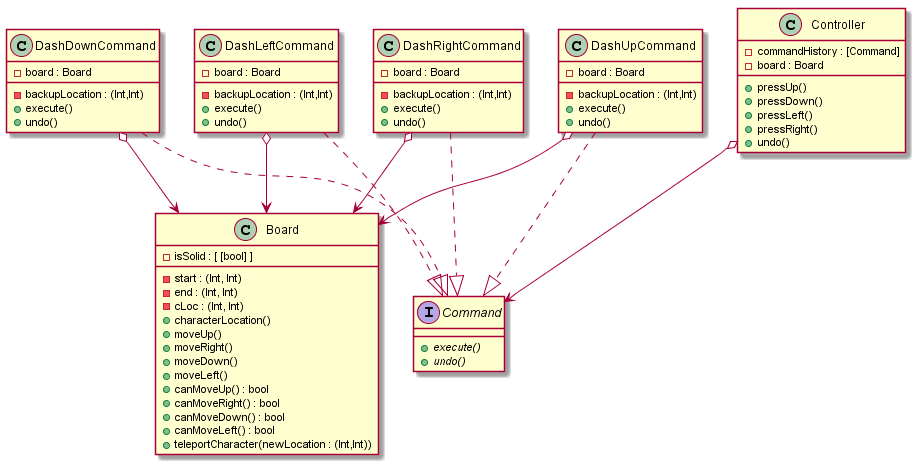
You’re creating a maze navigation game thing. This is what the application currently has right now:

* Board - this represents the layout of the maze. The layout is loaded from a file. It has these attributes:
  + \_\_isSolid - this is a 2 dimensional grid encoded as a nested list of booleans which represents the solid boundaries of the maze. For example if \_\_isSolid[row][col] is true then it means that that cell on (row,col) is a boundary
  + \_\_start - a tuple of two integers that represent where the character starts
  + \_\_end - tuple of two integers that represent the position of the end of the maze
  + \_\_cLoc - tuple of two integers that represents the current location of the character
  + moveUp(), moveDown(), moveLeft(), moveRight() - moves the character one space, in the respective direction. The character cannot move to a boundary cell, it will raise an error instead.
  + canMoveUp(), canMoveDown(), canMoveLeft(), canMoveRight() - returns true if the cell in the respective direction is not solid.
  + \_\_str()\_\_ string representation of the board. It shows which are the boundaries and the character location

What’s missing right now is controller support. This is how a player controls the character on the maze:

* dpad\_up(), dpad\_down(), dpad\_left(), dpad\_right() - The character dashes through the maze in the specified direction until it hits a boundary.
* a\_button() - The character undoes the previous action it did.

To implement controller support you need to create a Command abstraction which is realized by all controller commands. The Controller (which represents the controller) is the invoker for the commands. Since commands are undoable, this controller needs to keep a command history, represented as a list. Every time a controller button is pressed, it creates the appropriate Command, executes it and appends it to the command history. Every time the a\_button() is pressed to undo, the controller pops the last command from the command history and undoes it.



command example

### Why this is elegant

* **Single Responsibility Principle** - The behavioral responsibilities in the system are thoroughly separated. One invokes the command, and the other performs the behavior associated with the command.
* **Open/Closed Principle** - If there are more commands you want to add, you don’t have to touch any existing code.
* Switching between invokers and receivers is easily done
* You can implement undo (and redo)
* You can defer the execution of behavior
* A command may be made of smaller simpler commands

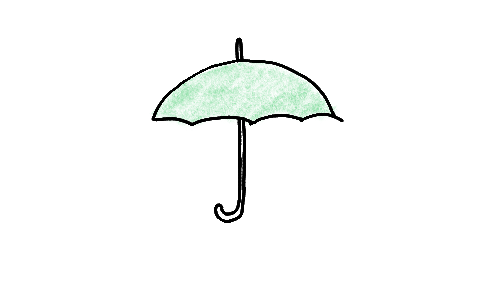
### How to implement it

1. Create an abstraction Command that contains abstract method execute(), and other command related methods like undo().
2. For every command, create a realization to Command. These commands uses a reference to a Reciever instance. This instance represents the instance/s that are affected whenever Command realizations are executed.
3. Create an Invoker class that will be responsible for instantiating, preparing, and executing commands. Inside these class are methods for invoking each commands. When these methods are called, the invoker does the following:
   1. Instantiate an instance of Command called c with the correct realization.
   2. select the receiver of the Command, including the related parameters.
   3. invoke c.execute().
4. If the system supports undoable commands, the Invoker should keep a list of commands called commandHistory and each command instance should keep a reference called backup to enable restoration of Reciever instances.

## Observer Pattern

### Problem

What if you need to inform a lot of objects about the changes to some interesting data? If you globalize the data and let your client objects poll for changes all the time, this will affect the security and safety of your interesting data. Plus, global data is something that should be avoided as much as possible. Also, forcing your objects poll for changes all the time will be inefficient if your interesting data has not changed.



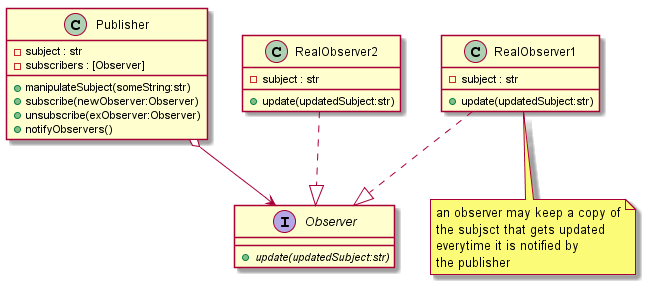
observer

### Solution

The responsibility of sharing information about the changes to interesting data should not be placed in the clients of the data. You should create a notifier class that encapsulates the interesting data. This class should be responsible of notifying interested clients about changes on the data.

To do this you need to encapsulate the interesting data (from now on lets call it the subject), into a Publisher class. An instance of this class will be responsible of notifying the observers for any change in the subject. Whenever there are changes to the subject, the Publisher instance calls notifyObservers() so that all interested, observers will be informed of the change. Any class that is potentially interested in the subject should realize an Observer abstraction, which in the bare minimum contains, the update(updatedSubject) function. Inside Publisher’s notifyObservers() method, every subscriber (an interested observer) is updated (subscriber.update()).

Any instance of an Observer should be subscribed to the change notifications using Publisher’s subscribe() function. They can also be unsubscribed using theunsubscribe() function.



observer

Whenever an observer has updated, the publisher needs to pass all the necessary details in the notification. This is generally done by passing the updated subject in the update(updatedSubject) method.

In some cases, the observer needs to keep a copy of the subject as an attribute. Make sure to change the value of this attribute during updates.

Make sure that changes to the subject are only done using the Publisher class (manipulateSubject()). If you change the subject without using Publisher’s methods, your subscribers won’t be notified.

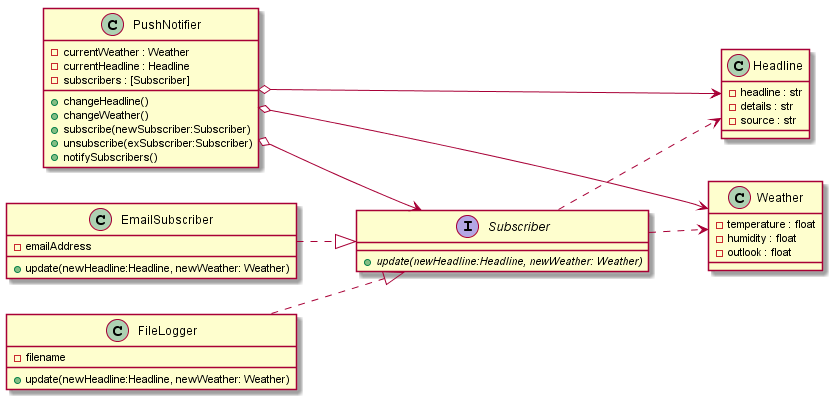
### Example

#### Push Notifier for Weather and Headlines

You are creating a push notification system that works for multiple platforms. You want to distribute information about the current weather and news headlines. This system will be potentially used on many platforms so you have to think about the maintainability issues for adding new platform support.

To implement this, you have to apply the observer pattern. Your subject would be Weather data and Headline data (which are their own classes). These subjects should be encapsulated into a single publisher class (which will be called PushNotifier).

Any platform, that is interested in the changes to the subject should realize a Subscriber abstraction (Observer), which contains the abstract method update().



observer example

### Why this is elegant

* **Open/Closed Principle** - You can add new Observer realizations seamlessly every time there are new objects that are interested in the subject.
* A observer can be subscribed/unsubscribed during runtime

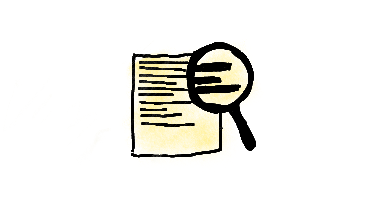
### How to implement it

1. Create an Observer abstraction that represents all classes that can potentially observe changes to the Publisher. Observer will contain the abstract method update().
2. All classes that want to be notified about changes to the subject should realize Observer.
3. Publisher will either own/use an instance of the subject of interest. It will also use an attribute which is stores the list of Observers that are interested in the subject. To attach or detach Observers, Publisher contains the methods subscribe() and unsubscribe().
4. Every time subject is manipulated, it should be done through Publisher , because Publisher needs to notify all Observers in its observer list attribute after every manipulation. This notification is done through notifyObservers() after the end of every subject manipulation.
5. Inside the Publishers notifyObservers method, every Observer in its list of observers invoke their update() method.

## Template Method Pattern

### Problem

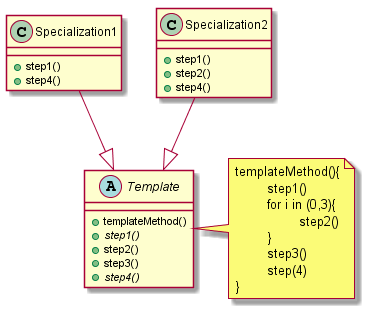
Say you have two or more *almost* identical behaviors from different classes. Rewriting these object behaviors as separate methods for each class duplicates many parts of the code (especially if the behavior has a lot of lines of code).



template

### Solution

To avoid code duplication, you break down your code into individual steps. By doing this you can create a superclass that contains the implementation for all common steps. This superclass will also contain the common implementation for the **template method**, the method that combines all steps into the original object behavior. differences between steps will be resolved under different specializations of this superclass.



template method

If the exact instance of the class is a Specialization1, it performs the template method with special versions of step1() and step4() (since Specialization1 overrides them) but the other parts are inherited from the Template.

The steps in the superclass can be a mix of abstract methods and concrete methods. Make a method abstract if you want to force all specializations to override these steps. You’ll want to do these if some of the steps in your template doesn’t have a default implementation.

### Example

#### Brute Force Recipe

If you write brute force algorithms as search problems, they will have a common recipe. This is the reason why it is called the exhaustive search algorithm. It will traverse all of the elements in the search space, trying to check the validity of each element, until it completes the solution

**Equality Search**

Search for integers equal to the target

#searchSpace = [2,3,1,0,6,2,4]  
#target = 2  
  
i = 0  
solutions = []  
candidate = searchSpace[0]  
while(i<len(searchSpace)):  
 if candidate == target:  
 solutions.append(candidate)  
 candidate = searchSpace[++i]  
   
#solution = [2,2]

**Divisibility Search**

Search for integers divisible by the target

#searchSpace = [2,3,1,0,6,2,4]  
#target = 2  
  
i = 0  
solutions = []  
candidate = searchSpace[0]  
while(i<len(searchSpace)):  
 if candidate % target == 0:  
 solution.append(candidate)  
 candidate = searchSpace[++i]  
   
#solution = [2,0,6,2,4]

**Minimum Search**

No target, searches for the smallest integer

#searchSpace = [2,3,1,0,6,2,4]  
#target = None  
  
i = 1  
solutions = [searchSpace[0]]  
candidate = searchSpace[1]  
while(i<len(searchSpace)):  
 if candidate <= solutions[0]  
 solutions[0] = candidate  
 candidate = searchSpace[++i]  
   
#solution = [0]

**Common Recipe**

i = 0  
solutions = []  
candidate = first()  
while(isStillSearching()):  
 if valid(candidate):  
 updateSolution(candidate)  
 candidate = next()

Because of this we can write a general brute force template method that would return the solution to brute force problems. To do this you create a superclass SearchAlgorithm() that contains the template method for brute force algorithms. If you want to customize this algorithm for special problems, all you have to do is to inherit from SearchAlgorithm and override only the necessary steps.

template example

template example

is isValid() and updateSolution(candidate) is different for each algorithm so it doesn’t have a default implementation. It would be best to make these steps abstract.

### Why this is elegant

* **Open/Closed Principle** - The Template is open for extension but closed for modification
* *Encapsulate what varies* - steps can vary from specialization to specialization, therefore they are encapsulated into step methods.
* Implementing this pattern will remove duplicate code in the common parts of the algorithm.
* Clients may override only certain steps in a large algorithm, making it easier to create specializations

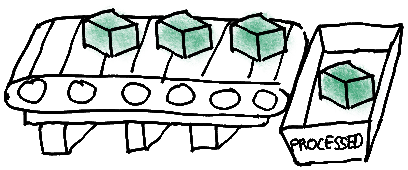
### How to implement it

1. Create an abstract class called Template. It contains the method templateMethod() broken down into separate steps through separate step1(), step2(), … and etc. methods. When invoked templateMethod() will just call these step methods.
2. Each step method inside Template will contain the default implementation of that step. If there is no default implementation, the method should be abstract.
3. For every similar behavior to templateMethod() a specialization of Template is created. These methods will implement all abstract methods and override all step methods that vary for its behavior.

## Iterator

### Problem

One of the most common iteration recipes that you’ll likely implement is the **for-each** loop. This loop traverses a collection, and performing some kind of operation along the way. Most programming languages implement for each loops on built in collections like arrays, sets, and trees. But what about non-built in collections?



iterator

### Solution

For non built-in collections, you can create an iterator that does the traversal for you. On the bare minimum these iterators will realize some Iterator abstraction that contains the methods, next(), and hasNext(). From these methods alone you can easily perform complete traversals without knowing the exact type of the collection:

i = collection.newIterator()  
while i.hasNext():  
 print(i.next())

iterator

iterator

The hasNext() method, returns a boolean value that indicates whether or not there are more elements to be traversed. The next() method, returns the next element in the traversal.

A collection can have more than one Iterators, if it makes sense for the collection to be travsersed in more than one way. Despite this possibility, a collection must have a default iterator which will be the type of the new instance returned in the factory method, newIterator()

### Why this is elegant

* **Single Responsibility Principle** - Traversal algorithms can now be placed into separate classes that interact with an iterator instead of the collection itself.
* **Open/Closed Principle** - You can implement new types of collections and iterators without touching any existing code.
* You can traverse all the elements in a collection, even if you don’t know the exact type of the said collection.
* Two iterators, can iterate over the same collection without problem as long as the iterators are of different instances.

### How to implement it

1. Create an abstraction called Iterator which contains the abstract methods next() and hasNext().
2. Create an abstraction called Collection which contains the abstract method newIterator().
3. For very collection that can be iterated through create a realization to Collection. Inside these Collection realizations, the newIterator() method must be implemented which simply returns a new instance of the default Iterator. (for collections that can be iterated through in more than one way, create different methods for creating other iterators as well but always keep newIterator() as the one that returns a new instance of the default iteration).
4. For every different way of iterating through a Collection realization, a realization to Iterator must be created as well.
5. Iterator realizations should contain an attribute that refers to the collection instance it is iterating through.

## Optional Reading

Shvets A. (2018) [Behavioral Patterns](https://sourcemaking.com/design_patterns/behavioral_patterns) Accessed August 31, 2020

# Structural Patterns

## Introduction

As your system evolves, the structure of your classes could get complicated. As you introduce more features, your classes become bigger and harder to maintain. To alleviate these issues, you can assemble objects into maintainable structural patterns.

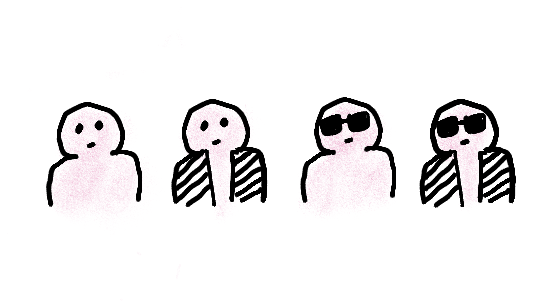
## Learning Objectives

1. Design systems that apply the decorator pattern
2. Design systems that apply the adapter pattern

## Decorator Pattern

### Problem

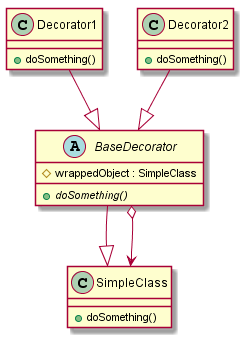
Some of your classes require extra features that can be added and removed during runtime. Sometimes you even need to support a set of extra features that can be arbitrarily combined with each other. You need to do this without breaking how these classes are being used by their clients.



decorator

### Solution

To solve this issue, all you have to do is to apply the open/closed principle. For every feature that can be arbitrarily added to some simple class, you need to create a Decorator that extends the features of classes using inheritance and composition at the same time. The neat thing about this pattern is that the Decorators will have polymorphically the same type as the simple class due to inheritance. Decorators will also be able to control instances of the simple class because of composition.



decorator

To create an instance of a SimpleClass decorated by Decorator1, all you need to do is to wrap the SimpleClass instance with an instance of Decorator1. When this Decorator1 instance, calls doSomething() it calls the wrapped SimpleClass instance’s doSomething() and do some extra behavior.

#Decorator1's implementation of doSomething():  
  
def doSomething(self):  
 self.\_wrappedObject.doSomething()  
 doSomthingExtra()

It would be handy to create a BaseDecorator abstract class that is inherited by all decorators. It’s not required but this class will form a class hierarchy for all decorators. Plus, you can write all of the common behavior and data into this class. It would be better for this class’s doSomething() to be abstract, since it doesn’t make sense for you to create instances of BaseDecorator.

### Example

#### Decorating Sentences

A sentence can be defined as a list of words (words are strings). The string representation of a sentence is the concatenation of all of the words in the list, separated by a space.

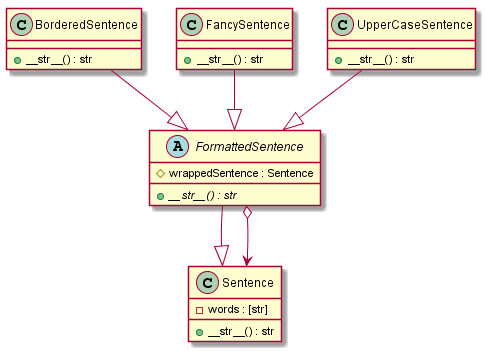
Instances of sentences can be printed with formatting:

* **bordered** - Given the sentence, ["hey","there"] it prints:
* -----------  
  |hey there|  
  -----------
* **fancy** - Given the sentence, ["hey","there"] it prints:
* -+hey there+-
* **uppercase** - Given the sentence, ["hey","there"] it prints:
* HEY THERE

The formatting of a sentence is decided during runtime. These formats should also allow for combinations with other formats:

* **bordered fancy** - Given the sentence, ["hey","there"] it prints:
* ---------------  
  |-+hey there+-|  
  ---------------
* **fancy uppercase** - Given the sentence, ["hey","there"] it prints:
* -+HEY THERE+-

To accomplish these features, you need to implement the decorator pattern. Each formatting will be a decorator for Sentence objects. These formats need to inherit from some abstract FormattedSentence class. This abstract class is specified to compose and inherit from sentence. The behavior that needs to be decorated is the \_\_str\_\_() function since you need to change how sentence is printed for every format.



decorator example

### Why this is elegant

* **Open/Closed Principle** - Decorators extend classes via inheritance. It is easier to add new decorators without touching any exiting code.
* A class which can have many variants because of diverse combinations of behaviors can be cleanly implemented using this pattern.
* You can arbitrarily mix and match decorators without the worry of polymorphic incompatibility during runtime

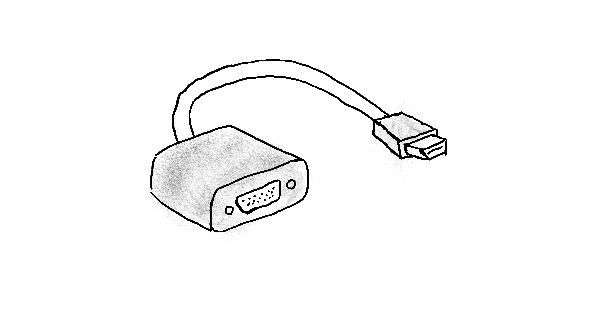
### How to implement it

1. Create an abstract class BaseDecorator that specializes some SimpleClass. This BaseDecorator will also have the attribute wrappedObject which is a reference to an instance of the decorated SimpleClass. This attribute is set as protected so that it may be inherited by BaseDecorators specializations. BaseDecorator also contains the abstract method doSomething(). This method’s behavior, when invoked by BaseDecorators specializations, changes depending on the decorations attached to SimpleClass
2. For every decoration, that can decorate SimpleClass, a specialization for BaseClass is created. These specializations implement doSomething() in a manner that augments/modifies SimpleClass’s own doSomething()

## Adapter Pattern

### Problem

As the system evolves, you’ll likely encounter interfaces of instances that are incompatible with their intended clients. These interfaces do perform the necessary behavior, but maybe the method names are just different. This happens quite a lot since the interface of the dependency may be originally built for different reason. The interface may be an external module imported on existing client code. You can just change the incompatible interface to support the functionality you need but this is not always possible and may introduce code duplication.



adapter

### Solution

In the same way a usb-c interface is usable on a usb 2.0 using an adapter, you can use an incompatible service on a client as a compatible instance using the adapter pattern.

Say you have an instance of AbstractService (it could be any realization of AbstractService), that needs to be used like an instance of RequiredInterface by some client. What you need to do is to create an adapter to AbstractService called ServiceAdapter which realizes RequiredInterface. To adapt the instance of AbstractService, you have to compose it inside the ServiceAdapter. So that serviceMethod1() is adapted to method1().

adapter

adapter

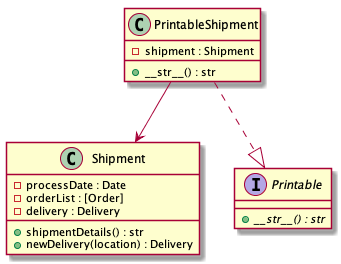
Whenever, a ServiceAdapter calls method1() it instead delegates the behavior to the embedded service, which instead calls serviceMethod1()

### Example

#### Printable Shipments

Looking back at our previous lab exercises, some of the example classes contain string representation but do not implement the \_\_str\_\_() function. An example of this is Shipment back from the factory method example. It does contain a string representation builder called shipmentDetails(), but printing a shipment is quite tedious since you have to print, s.shipmentDetails(). You can replace the name of shipmentDetails() to \_\_str\_\_() but this will potentially affect other clients of shipment. You can add the \_\_str\_\_() function which does exactly the same but this may introduce unwanted code duplication.

The best solution for this problem is to create an adapter for shipment called PrintableShipment. This adapter will realize some Printable abstraction, which only contains the abstract method \_\_str\_\_().



adapter example

### Why this is elegant

* **Open/Closed Principle** - Instead of changing existing incompatible interfaces, you can extend them by creating adapters.
* Instead of cluttering up your code with duplicate functions and unused interface methods, you instead create adapters only when it is needed.

### How to implement it

1. If you want an AbstractService to be used like a RequiredInterface, Create a ServiceAdapter that realizes RequiredInterface and contains an attribute service that is a reference to an instance of AbstractService.
2. Inside ServiceAdapter, the implementations of RequiredInterface’s abstract methods are merely calls to the methods of the reference service.

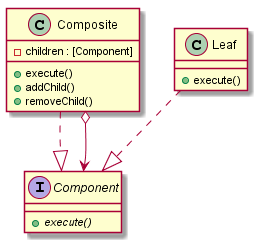
## Composite (Optional Read)

### Problem

When entities in your system needs to be represented like trees, then you represent them like trees.

### Solution

The composite pattern describes a tree structure described polymorphically. A tree node can either be a general tree or a leaf. in the composite pattern, a Component (tree node) can either be Composites (general tree), or a Leaf. Leaves and Composites are realizations of Component.

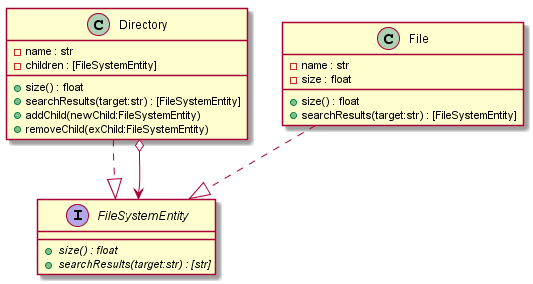


composite

### Example

#### File System

The file system in your computers are described using a tree structure. The entities in your file system are either directories or files.



composite pattern

What you need to do is to implement a simulation of a file system. Each node of the file system should be able to call the following methods:

* **size()** - the size of a File is based on the attribute size which is set during the initialization of the File instance. The size of a Directory is the size of all the FileSystemEntities inside it.
* **searchResults(target)** - if used by a File, if the name of the file matches the target it returns a list containing the File, if not it returns an empty list. If used by a Directory returns a list containing all the of the instances of FileSystemEntity (including itself) that matches the target inside the Directory.

### Why this is elegant

* **Open/Closed Principle** - You can introduce new component realizations in the system without touching any existing code.
* Working with composite trees are easier because of the polymorphism in the pattern

### How to implement it

1. Create an abstraction called Component that contains the abstract methods that are supposed to be executed across all components.
2. The Component has two realizations, Composites and Leafs.
3. Component contains an attribute children which is a list of Composite instance references and the methods addChild() and removeChild() to attach/detach Composites. It also has execute() which an implemented method from Component.
4. Leaf contains the method execute() as well.
5. When a Composite’s execute is invoked, it calls invokes all of its children’sexecute(). When Leaf’s execute is invoked it performs, leaf related behavior.

## Facade (Optional Read)

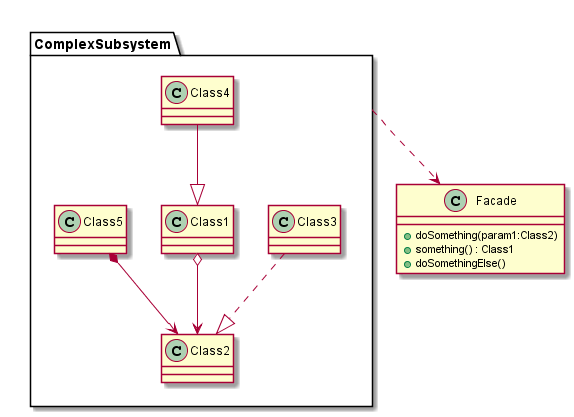
### Problem

When your system becomes large enough, parts of the system which is responsible for a single operation may involve interactions between multiple classes. Creating flexible and maintainable systems tend to look like this.

Looking from the outside, simple functionality (like borrowing a book or depositing money to an account) will appear complex since it involves multiple lines of object interaction.

### Solution

To solve this issue, you create a straightforward interface, that contains methods to encapsulate complicated functionality in your subsystem. Instead of using the internal classes to perform some functionality, you call the facade interface’s method instead.



facade

### Why this is elegant

* Implementing this pattern encapsulates complicated subsystem behavior into simple straightforward functions.

## Optional Readings

Shvets A. (2018) [Structural Patterns](https://sourcemaking.com/design_patterns/structural_patterns) Accessed August 31, 2020

1. Alexander (1977). *A Pattern Language: Towns, Buildings, Construction*. [↑](#footnote-ref-25)
2. Gamma, Vlissides, Johnson, and Helm (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* [↑](#footnote-ref-27)
3. Programming Paradigms according to VanRoy by [MovGP0](https://commons.wikimedia.org/wiki/User:MovGP0) used under [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) from https://en.wikipedia.org/wiki/Programming\_paradigm#/media/File:Programming\_paradigms.svg [↑](#footnote-ref-30)
4. Alexander (1977). *A Pattern Language: Towns, Buildings, Construction*. [↑](#footnote-ref-49)
5. Gamma, Vlissides, Johnson, and Helm (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* [↑](#footnote-ref-60)
6. Alexander (1977). *A Pattern Language: Towns, Buildings, Construction*. [↑](#footnote-ref-96)
7. Alexander (1977). *A Pattern Language: Towns, Buildings, Construction*. [↑](#footnote-ref-98)
8. Alexander (1977). *A Pattern Language: Towns, Buildings, Construction*. [↑](#footnote-ref-231)
9. Gamma, Vlissides, Johnson, and Helm (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* [↑](#footnote-ref-232)