Frameworks

With frameworks we have the Inversion of control: the program flow of control is dictated by the framework instead of by the caller. The framework provide a default behaviour, the programmer just have to override functions and handle events.

Java Virtual Machine (JVM)

is an example of an abstract machine. It is a virtual machine that provides a runtime environment to execute Java bytecode, which is the compiled form of Java source code. The JVM provides a platform-independent execution environment, as Java bytecode can be executed on any system that has a JVM installed, regardless of the underlying hardware and operating system.

responsibility of JVM

The JVM is responsible for many tasks, including garbage collection, memory management, security, and performance optimization. It is designed to be platform-independent, which means that it can run the same bytecode on any system that has a compatible JVM installed

Pure interpretation

is a technique for executing programs where each instruction is executed directly by an interpreter, without being translated into machine code beforehand. In pure interpretation, the interpreter reads each instruction from the source code and executes it immediately. This contrasts with compiled code, where the source code is first translated into machine code before being executed. like python

Pure compilation

the code is converted from a high-level language to low-level machine code before execution. Like C

JAVA

uses a combination of interpretation and compilation to execute Java programs.

1-compiled, it is translated into bytecode(low-level code)

2-This bytecode can then be interpreted by the Java Virtual Machine (JVM) at runtime

3-(optional)it can be compiled again by a Just-In-Time (JIT) compiler to produce native machine code for better performance.

JVM exploits both compilation and interpretation: the Java code is compiled into bytecode, interpreted by the Java Virtual Machine. Jvm is dynamic link.

Dynamic Linking: the linking is executed when needed at runtime instead of at compile time. is a technique that allows the loading of libraries or modules into memory only when they are needed,

Runtime systems a collection of software components that are responsible for executing and managing programs written in a particular programming language. It provides a runtime

environment for the program, including memory management, input/output operations, and other low-level tasks that are necessary for the program to run correctly.

The runtime support is needed for:
• memory management: push/pop activation records on stack, allocation and garbage collection on heap
• I/O management
• runtime environment interaction
• parallel execution threads/tasks/processes
• dynamic type checking and bounding
• dynamic linking
• debugging, JIT compilation, etc.

Java Runtime Environment (JRE)

Includes all what is needed to run compiled Java programs (bytecode).
It is made by the JVM and the JCL (Java Class Library, aka Java API).
The Java Runtime Environment (JRE) includes the Java Virtual Machine (JVM), which is responsible for executing compiled Java code (bytecode), and the Java Class Library (JCL), which provides a set of reusable classes and interfaces for developers to use in their programs. The JCL is also known as the Java API

Stack frames

Are often allocated on the heap and contains:
• Local variable array: contains all the variables used during a method execution
• Operand stack: general-purpose registers
• Reference to constant pool of current class
• Return address
Stack frames, also known as activation records or call frames, are data structures used by the JVM to keep track of the state of a method or function during its execution. Stack frames are often allocated on the stack, but can also be allocated on the heap in some cases.

Reference types:

array: an ordered collection of elements of the same type, identified by a reference to the array object
class: a template for creating objects, which defines the object's properties and methods
interface: a collection of abstract methods that can be implemented by a class
object representation: the internal representation of objects in memory, which includes the concrete value of null, pointers to instance and class data, mutex locks, and garbage collection flags.

The JVM runtime data areas can be divided into two main categories: heap and non-heap.
The heap is a region of memory where objects and arrays are allocated. The heap is dynamically sized, meaning that its size can grow or shrink during the lifetime of a JVM instance. The heap is

shared by all threads running in the JVM, and it is garbage-collected to reclaim memory that is no longer needed by the program.
Non-heap memory includes areas of memory that are not used to store objects and arrays, but rather are used for internal JVM purposes. The non-heap memory can be further divided into the following areas:
Permanent Generation (PermGen): The PermGen is used to store metadata about classes, methods, and other JVM internal structures. This area was removed in Java 8 and replaced with the Metaspace.
Code cache: The code cache is a region of memory used to store code that has been compiled by the JVM's Just-In-Time (JIT) compiler. The code cache is used to improve the performance of the JVM by caching compiled code that can be reused later.
//Stack: The stack is a region of memory used to store the runtime stack for each thread. Each thread has its own stack, which is used to store method call frames, local variables, and operand stacks. The size of the stack is fixed when the thread is created and cannot be changed during the thread's lifetime.

Threads

Multiple threads are possible starting from a single main. Moreover, there are several background threads for garbage collection, signal dispatching, JIT compilation, etc.Threads have shared access to heap and persistent memory. The latter includes the method area, interned strings, and code cache for JIT compilation.Each thread has its separate stack area

Disassembling Java files
- compile: javac Class.java
- run: java Class.class
- decompile: javap -c -v Class.class

The linking consists in:

- verification: correctness, overflow and underflow, variables and types validity, data-flow analysis
- preparation: allocation of storage
- resolution (optional): loading of the referred classes and interfaces (can be postponed to first use)

The JVM supports three addressing modes for operands:
1 Intermediate, with a constant as part of instruction
2.Indexed: The operand is an index into an array of local variables or parameters.
3.Stack: The operand is the value at the top of the operand stack.

Modules
are a feature of specific programming languages, while components are a more general concept of software development. Modules provide a way to organize code within a language, while components provide a way to create reusable units of code across languages and systems.

Java beans

Java Beans

“A Java Bean is a **reusable software component** that can be **manipulated visually** in a **builder tool**.”

Import java.beans
Create obj of propertychangeSupport
Implement remove and Add properly Changes listener method
In set method ,firePropertyChange
Create interface for propertyChangeListener
Override property change method with evt arguments
In main Create obj and call Add PropertychangeListener

Reflection in Java

The ability of a program to manipulate itself as data.

- Introspection: can observe its internal state (read-only)
- Intercession: can modify itself

This abilities requires reification, a mechanism to encode the execution state as data.

- Structural reflection: complete reification of both the program in execution and its abstract data types.
- Behavioral reflection: complete reification of semantic and implementation of the program and of the runtime system.

Example:

- A is a superclass;
- classes B and C extends A implements the same method doSomething with different parameter type;
- a method m accept objects of type A, that actually are object of type B or C;
- using reflection m can invoke doSomething on both B and C objects with the correct parameter type.

Annotations

are a form of metadata in Java that can be added to classes, methods, fields, and other program elements to provide additional information about them. Annotations are used to provide information that can be processed by tools at compile time or runtime to generate code, perform validation, or enhance functionality.
Annotations are denoted by an @ symbol followed by the annotation type name. For example, @Override is an annotation that indicates that a method is intended to override a superclass method.

@Target is used to specify the program elements to which an annotation can be applied. It takes an array of ElementType values that include options such as METHOD, FIELD, PARAMETER, TYPE, and others.
@Retention is used to specify the retention policy of an annotation, or how long it should be retained. It takes an enum of RetentionPolicy values that include SOURCE, CLASS, and RUNTIME. SOURCE means the annotation is discarded by the compiler, CLASS means it's included in the class file but not available at runtime, and RUNTIME means it's available at runtime and can be accessed through reflection.
@Inherited is a marker annotation that indicates that an annotation should be inherited by subclasses. This means that if a superclass is annotated with the @Inherited annotation, its subclasses will automatically inherit the same annotation.

IOC

The framework has control over the execution flow, calling app code for app-specific behavior.
In traditional programming, components directly depend on each other and are tightly coupled. With IoC, the flow of control is inverted, and the control is transferred to a framework or container. This framework or container manages the components and their dependencies. In other words, the framework decides which components to create and how to connect them.

Loosely coupled systems are designed in such a way that the components or modules are not tightly dependent on each other. This means that changes in one module do not have a significant impact on the other modules. Loosely coupled systems provides many advantages, and in particular improve the extensibility, testability and reusability of the software. Overall, loose coupling is a fundamental principle of good software design, as it leads to more modular, flexible, and scalable systems.

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in a superclass, but defers some steps to the subclasses. The main idea is to define the overall structure of the algorithm while allowing subclasses to customize some of its steps without changing its overall structure.
Concrete operations are the methods that provide default behavior in the template method. These methods are implemented in the base class and cannot be overridden by subclasses.
Primitive operations are the methods that must be implemented by subclasses to provide specific behavior. These methods are declared as abstract in the base class and are implemented by subclasses.
Hook operations are the methods that may be optionally overridden by subclasses to provide additional behavior. These methods are also declared as abstract in the base class, but have an empty default implementation. Subclasses can choose to override them if they need to customize the behavior

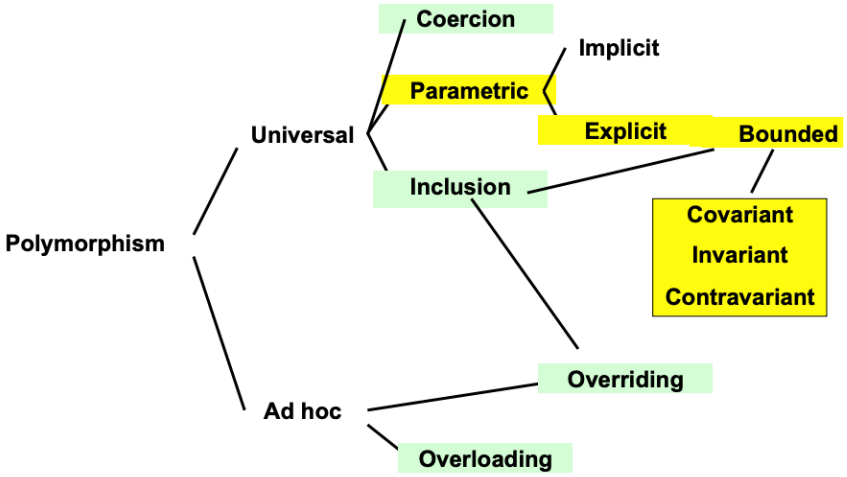
Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as if they were of the same class. It refers to the ability of an object to take on multiple forms or perform multiple functions depending on the context in which it is used.
Inheritance and interfaces are two common ways to achieve polymorphism in object-oriented programming. With inheritance, a subclass can inherit methods and properties from its superclass, and it can also override or add new functionality to those inherited methods. With interfaces, different classes can implement the same interface, which defines a common set of methods that each class must provide.
Polymorphism allows for more flexible and reusable code, as objects can be used interchangeably with other objects of the same class or interface, regardless of their specific implementation. It also enables the creation of generic code that can work with multiple types of objects, which can reduce the amount of code that needs to be written and maintained.
Binding time
The binding of the function name with the code can be:
• at compile time (early, static binding: better, for debugging reasons)
function call is resolved and associated with the corresponding code before the program is executed. This type of binding is better for debugging purposes as it can help identify errors in the code early in the development process.
• at linking time

binding between a function name and its code is established during the linking phase of the program(after compile time). This occurs when separate modules or libraries are combined to create an executable program.

• at execution time (late, dynamic binding, runtime)

function call is resolved and associated with the corresponding code at runtime. This type of binding is often used in object-oriented programming languages that support polymorphism and inheritance, as it allows for greater flexibility in calling methods on objects. However, late binding can also introduce some performance overhead due to the need for dynamic resolution of function calls.

Slower than early , less efficient



Ad hoc polymorphism, also known as overloading or function overloading, refers to the use of the same function or operator name to denote different implementations that are determined by the actual types of the arguments or operands.

Universal polymorphism, also known as parametric polymorphism or generic programming, refers to the use of a single implementation that is suitable for different types of objects.

Overloading (ad hoc)

Function overloading is a type of ad hoc polymorphism where the same function or operator name is used to represent different implementations that vary in their argument types, number, or order.

In statically typed languages, such as C++ and Java, the binding of overloaded functions is performed at compile time, which means that the correct function is selected based on the type and number of arguments passed to the function. This is known as early binding, as the binding occurs before the program is executed.

In dynamically typed languages, such as Python and Ruby, the binding of overloaded functions is performed at runtime, which means that the correct function is selected based on the actual types of the arguments passed to the function. This is known as late binding, as the binding occurs during program execution.

Overriding is a mechanism in object-oriented programming that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. When a method is

called on an object of the subclass, the overridden method in the subclass is called instead of the original method in the superclass.

Coercion (universal)

Coercion in universal polymorphism refers to the automatic conversion of an object from one type to another, without requiring explicit casting or conversion functions. This is done by the programming language runtime environment or the compiler.

Inclusion (or subtyping) is a key feature of object-oriented programming languages that allows a subtype to be treated as a supertype. This means that an object of a subtype can be used in any context where an object of the supertype is expected

Parametric polymorphism is a type of polymorphism where code can be written in a generic way to work with different types.

Invariance, Covariance and contravariance

- Invariant: don't depend on type hierarchy.
- Covariant: preserve the type hierarchy (safe if the type is read-only).
- Contravariant: invert the type hierarchy (safe if the type is write-only).

Invariance refers to the situation where the type of a variable or method parameter is fixed and does not depend on the type hierarchy. In this case, the variable or method parameter can only be of the exact type specified, and no subtypes or supertypes are allowed. For example, in Java, an ArrayList<String> cannot be assigned to an ArrayList<Object>, even though String is a subtype of Object.

Covariance refers to the situation where the type of a variable or method parameter can vary in the direction of the type hierarchy. In this case, a subtype can be used in place of a supertype, and the program is still type-safe. For example, in Java, an array of String[] can be assigned to an array of Object[], because String is a subtype of Object.

Contravariance refers to the situation where the type of a variable or method parameter can vary in the opposite direction of the type hierarchy. In this case, a supertype can be used in place of a subtype, and the program is still type-safe. For example, in Java, a Comparator<Object> can be used to sort a List<String>, because a Comparator<Object> can compare any two objects, including Strings.

Templates vs Macros in C++

templates are a powerful feature in C++ that allow you to define generic algorithms and data structures that can work with different types of data. A template is a blueprint for creating functions or classes that can work with different types of data, and the actual type is determined at compile-time. Templates are type-safe

```
template<typename T>
T add(T a, T b) {
   return a + b;
```

```
}

int main() {
   int x = add(2, 3); // returns 5
   double y = add(3.14, 2.71); // returns 5.85
   return 0;
}
```

In C++, macros are preprocessor directives that allow you to define symbolic constants, functions, or code blocks that are replaced by the preprocessor with actual code before compilation. Macros are defined using the #define directive, and can take arguments like functions. Macros are not type-safe, because they operate purely on textual substitution.

```
#define PI 3.14159265359

double circleArea(double radius) {
   return PI * radius * radius;
}
```

Java Generics

Java Generics is a feature that was introduced in Java 5 to provide compile-time type safety and code reusability for collections and related APIs. Generics enable you to parameterize types so that a class or method can operate on objects of different types without the need for casting.

With generics, you can define a class or method with one or more type parameters, which can be any valid Java type, including classes, interfaces, and other type parameters. For example, the following code shows a simple generic class:

```
public class MyList<T> {
   private T[] data;
   private int size;

   public MyList() {
      data = (T[]) new Object[10];
      size = 0;
   }
```

```
    public void add(T element) {
        data[size++] = element;
    }

    public T get(int index) {
        return data[index];
    }
}
```

To use the MyList class, you can provide a type argument for the T parameter, such as:

```
MyList<String> myList = new MyList<>();
myList.add("foo");
String s = myList.get(0);
```

Type Bounds

<TypeVar extends SuperType> hadeaksar

– *upper bound*; SuperType and any of its subtype are ok.

<TypeVar extends ClassA & InterfaceB & InterfaceC & …>

– *Mul4ple* upper bounds

<TypeVar super SubType> hadeaghal

– *lower bound*; SubType and any of its supertype are ok

wildcard

In Java generics, a wildcard is a special type argument that can be used with a generic class, interface, or method to represent an unknown type. The wildcard is represented by the ? character and can be used in two forms: the upper bounded wildcard and the unbounded wildcard.

The upper bounded wildcard is used to represent a type that is a subclass of a given type or implements a given interface. The syntax for an upper bounded wildcard is:

<? extends UpperBound>

Here, UpperBound is the upper bound for the wildcard. Any type that extends UpperBound can be used with this wildcard.

```
public static void printList(List<?> list) {
   for (Object obj : list) {
      System.out.println(obj.toString());
   }
}
```

In this example, the list parameter is declared using an unbounded wildcard <?>, which means that the method can accept a list of any type of object.

Type erasure is a feature of Java generics that ensures that generic type information is only available at compile time and is erased (removed) at runtime. This means that at runtime, a generic class or method looks like it is working with non-generic (raw) types. This is done for backward compatibility with pre-generics code and to ensure that generics do not cause any performance issues.

```
public class MyList<T> {
   private T[] data;
   private int size;

   public MyList() {
      data = (T[]) new Object[10];
      size = 0;
   }

   public void add(T item) {
      data[size++] = item;
   }

   public T get(int index) {
      return data[index];
   }
}
```

At compile time, the type parameter T is replaced with its upper bound Object. The resulting code looks like this:

```
public class MyList {
   private Object[] data;
   private int size;

   public MyList() {
      data = new Object[10];
      size = 0;
   }

   public void add(Object item) {
```

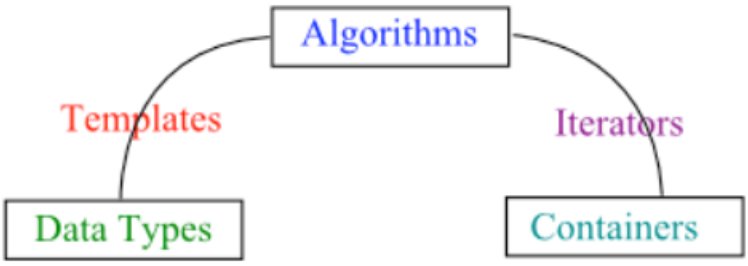
```
        data[size++] = item;
    }

    public Object get(int index) {
        return data[index];
    }
}
```

As you can see, the type parameter **T** has been replaced with **Object** in the compiled code. This means that at runtime, the generic type information is not available, and the class works with non-generic (raw) types.

The Standard Template Library (STL)

is a library of generic algorithms and data structures implemented in C++. It was designed to provide a standard set of tools for working with data structures and algorithms, so that developers can focus on solving specific problems rather than worrying about the low-level details of managing memory and writing complex algorithms.



Insert/delete complexity in containers:

Container	Beginning	Middle	End
vector	linear	linear	constant
list	constant	constant	constant
deque	constant	linear	constant

Iterator types

Forward iterators support the dereference operator (*) and both pre- and post-increment operators (++operator and operator++).

Input and output iterators are a type of iterator used in C++ to iterate over input and output streams.

Bidirectional iterators are a type of iterator that extends the functionality of forward iterators by adding the ability to move backward in the container. They have all the operations that forward iterators have, plus the pre/post decrement operators (--operator/operator--).
Random access iterators allow for constant-time jumps to any element in the sequence. They are the most versatile type of iterator.

Container	Iterator
vector	random access
list	bidirectional
deque	random access

Memory management

Memory model

STL abstract the memory model with allocators. Allocators are classes that encapsulate the information about the memory model. Each container is parametrized by such an allocator to let the implementation unchanged when switching memory models.

Functional Programming

Functional programming is a programming paradigm that emphasizes the use of functions to perform computations. The key idea of functional programming is to do everything by composing functions, without using mutable state or causing side effects. In functional programming, functions are treated as first-class citizens, meaning they can be passed as arguments to other functions, returned as values from functions, and stored in data structures.

Main Concepts

• 1st class and high-order functions: functions can be denoted, passed as argument to other functions and returned as result
• Recursion instead of iteration
• Powerful list facilities

• Polimorphism: universal parametric implicit
• Data structures cannot be modified, them must be recreated

ML family

The ML family of programming languages, which includes Standard ML, Caml, OCaml, and F#, has several key features, including:

Features:

• Type safe, with type inference and formal semantics
• Both compiled and interactive use
• Expression-oriented
• Higher order functions
• Anonymous functions: lambda
• Abstract data types
• Garbage collector
• Module system
• Exceptions
• Impure: allows side-effects

Haskell

Features:

Type checking and type inference: Haskell has a strong static type system that checks for type errors at compile time. Additionally, Haskell's type system can infer the types of expressions without the need for explicit type annotations in many cases.
Polymorphism: Haskell supports two types of polymorphism: parametric polymorphism (similar to ML) and ad hoc polymorphism (similar to overloading in object-oriented languages). Lazy evaluation: Haskell uses lazy evaluation, which means that expressions are only evaluated when they are needed. This can lead to more efficient code and enables the use of infinite data structures.
Tail recursion and continuations: Haskell supports tail call optimization, which allows certain recursive functions to be optimized so that they use constant stack space. Additionally, Haskell provides support for continuations, which enable advanced control flow features. Purely functional: Haskell is a purely functional language, which means that functions do not have side effects and always return the same output for the same input. This makes it easier to reason about the behavior of programs and enables more advanced optimization techniques.
Variables are bound to expressions without evaluating them: In Haskell, variables are not assigned values in the traditional sense. Instead, they are bound to expressions, which are not evaluated until they are needed. This supports lazy evaluation and enables more efficient code.

Core Haskell

Tuple: an ordered collection of values of different types. List: a sequence of values of the same type.
Record: a collection of named fields.
Patterns: used to match and destructure data structures.

Declarations: used to define functions, types, and other values.
Functions: defined using the syntax "name arguments = body", where arguments can be pattern-matched and the body is an expression or series of expressions.
Polymorphism: Haskell supports two forms of polymorphism:
Implicit parametric polymorphism: functions can be defined to work with any type, and the type is inferred at compile time.
Ad-hoc polymorphism: also known as overloading, allows functions to have different implementations based on the types of their arguments. Type declarations: used to define new types or type synonyms. Type classes: a form of ad-hoc polymorphism that defines a set of functions that must be implemented for a given type. Monads: used to encapsulate side effects and control the order of evaluation.
Exception: Haskell has a built-in exception mechanism that can be used to handle errors or unexpected conditions.

Lazyness Functions and data constructors don't evaluate their arguments until they need them.
In several languages there are forms of lazy evaluation (if-then-else, shortcutting && and ||)
///
In lazy evaluation, expressions are not evaluated until their results are actually needed. This can be useful in avoiding unnecessary computations, particularly in cases where evaluating an expression would be expensive.
In Haskell, this is the default evaluation strategy, and it applies to both functions and data constructors. For example, if you have a function that takes two arguments and only uses one of them, the unused argument will not be evaluated. Similarly, if you have a list with a large number of elements, only the elements that are actually used will be evaluated.
Other languages, such as Python and Ruby, also support some forms of lazy evaluation. For example, in Python, the and and or operators use short-circuit evaluation, which means that the second operand is only evaluated if necessary.

Parameter passing modes

– In
– In/out
– Out

In: This mode is used when the parameter is passed to the function for read-only access. In other words, the function cannot modify the parameter, and any changes made to the parameter within the function do not affect the value of the parameter in the calling program.
In/out: This mode is used when the parameter is passed to the function for both input and output. In this mode, the function can read and modify the value of the parameter, and any changes made to the parameter within the function are reflected in the calling program.
Out: This mode is used when the parameter is passed to the function for output only. In this mode, the function can modify the value of the parameter, but the

initial value of the parameter is not used by the function. Any changes made to the parameter within the function are reflected in the calling program

Parameter passing mechanisms

– Call by need (in)
– Call by sharing (in/out)
– Call by name (in+out)

Call by need (in): The argument is evaluated only when it is first used inside the function. The evaluation result is then cached and reused for subsequent uses of the argument within the function.
Call by sharing (in/out): A hybrid of call by value and call by reference. The function receives a copy of the argument, but the copy is a reference to the original object. Changes made to the object within the function are reflected in the original argument.
"Call by name" is a method of parameter passing used in programming languages. In this method, instead of passing the value of a parameter to a function, the expression corresponding to the parameter is passed. The expression is then evaluated every time the parameter is used within the function.

Value vs reference
• Value copy the value into the variable (copy of data)
• Reference copy the reference to the value into the variable (shared data)
Reference vs pointer
• Reference to x: address of the cell in which is stored x
• Pointer to x: location containing the address (reference) of x

Value (in): The function receives a copy of the argument value, which is passed by value. Any changes made to the parameter within the function do not affect the original argument.
Reference (in/out): The function receives a reference to the argument, allowing it to modify the original argument.
Result (out): The function does not receive any parameters, but instead returns a value that is the result of its computation.
Value/result (in/out): The function receives a copy of the argument value and returns a value as its result. Any changes made to the parameter within the function do not affect the original argument.

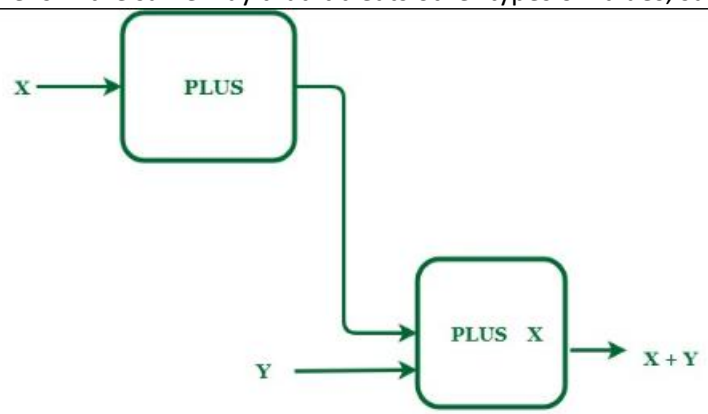
Higher-Order FuncDons

• FuncDons that take other funcDons as arguments or

return a funcDon as a result are **higher-order func4ons**.

*Any curried function with more than one argument is higher-order: applied to one argument it returns a function.

A higher-order function is a function that takes one or more functions as arguments, and/or returns a function as its result. In other words, a higher-order function is a function that treats functions as first-class citizens in the same way that it treats other types of values, such as integers, booleans, and strings.



Curried

add :: Int -> Int -> Int

add x y = x + y

This function takes two Int arguments, x and y, and returns their sum. The function is curried because it's defined to take its arguments one at a time, producing a new function each time.

A curried function with more than one argument is a higher-order function because when partially applied, it returns a new function that can be used later. For example, given a function add with type add :: Int -> Int -> Int, you can partially apply it like this:

addOne = add 1

Now addOne is a new function with type addOne :: Int -> Int, which adds 1 to its argument. You can use addOne later in your code or pass it as an argument to another function.

Map Applies argument function to each element in a collection.

`map (+1) [1,2,3] => [2,3,4]`

map is a higher-order function because it takes a function as an argument. In Haskell, functions are first-class values, which means that they can be passed as arguments to other functions, returned as values from functions, and stored in data structures.

Reduce `foldl`, `foldr`, `foldl'`, `foldl1` etc. `sum' xs = foldl (\acc x -> acc + x) 0 xs` The difference between `foldl` and `foldr` is that the first start from the top, so is applicable also to infinite lists, the second one requires a finite list. The difference between `foldl` and `foldl'` is that the second one uses tail-recursion (see below) and for that reason is more performant.

Recursion

Iteration and recursion are equally powerful in theoretical sense. Anyway, in general a procedure call is much more expensive than a conditional branch, thus recursion is in general less efficient. Tail recursive functions *Non tail-recursive*

```
int rfun()
 { ... return 1+rfun();
}
```

Tail recursive

```
int trfun() {
...
return trfun();
}
```

type classes in functional programming languages like Haskell allow for ad-hoc polymorphism, meaning that a single function can have multiple implementations depending on the types of its arguments.

A type class defines a set of functions (or operations) that can be applied to one or more types. For example, the Eq type class defines the functions == and /= that can be applied to types that can be compared for equality.

Default type classes

- `Eq`: equality
- `Ord`: comparison
- `Num`: numerical operations
- `Show`: convert to string
- `Read`: convert from string
- `Enum`: operations on sequentially ordered types

Declaration

A class declaration in Haskell defines a set of functions, called methods, that must be implemented by any type that is an instance of the class.

```
class Num a where
 (+) :: a -> a -> a
 (*) :: a -> a -> a
 negate :: a -> a ...
```

Instance

An instance declaration for a type Int says how the Num operations are implemented on Int's.

```
instance Num Int where
a + b = intPlus a b
a * b = intTimes a b
negate a = intNeg a
```

Default methods

Type classes can define default methods. Instance declaration can still override it by providing a more specific definition. If an instance declaration doesn't provide a method implementation, the default one is applied.

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

If is not necessary to override the default definition, deriving can be used.

Deriving

For `Read`, `Show`, `Bounded`, `Enum`, `Eq` and `Ord` the compiler can generate instance declaration automatically.

```
data Color = Red | Green | Blue
  deriving (Show, Read, Eq, Ord)
```

Under the hood

This

```
square :: Num n => n -> n
square x = x * x
```

is compiled into this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

where the extra value argument d of the function is a value of data type Num n and represents a dictionary of the required operations.

Furthermore, the compiler converts each type class declaration into a dictionary type declaration and a set of selector functions. In other words, this

Compositionally

```
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

becomes

```
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x) (square d y)
```

به کامپایلر میگوید که از اوپریشن های num برای دو عملوند + و square استفاذه کن برای نوع n چون برای num این عملوند ها قبلا تعریف شده اند

Functor

The Functor type class allows mapping a function over a type constructor.
It is defined in
data Box a = Box a
instance Functor Box where
fmap f (Box x) = Box (f x)

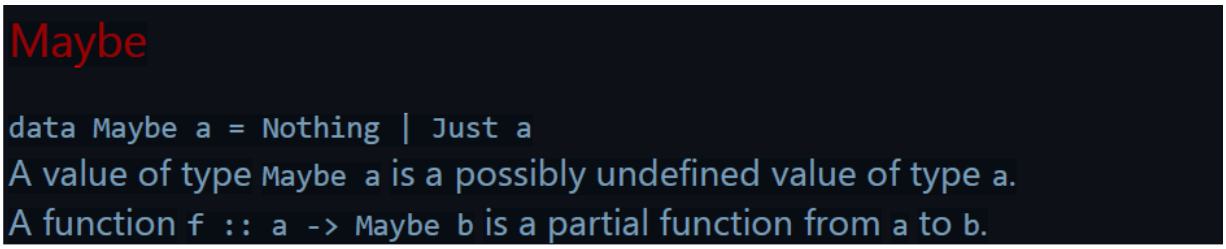
The first rule is that fmap id should be equivalent to the identity function id. In other words, applying fmap to the identity function on a functor should not change the functor in any way:
fmap id == id
The second rule is that fmap should preserve function composition. In other words, if you apply two functions to a functor in sequence, that should be the same as applying their composition to the functor:
fmap (f . g) == fmap f . fmap g

a functor is a typeclass that defines the behavior of mapping over a container type.

The Monad type class provides a way to sequence computations that involve monadic values. For example, suppose we have two computations that return Maybe values:
foo :: Int -> Maybe Int
foo x = if x > 0 then Just (x * 2) else Nothing
bar :: Int -> Maybe Int
bar x = if x < 10 then Just (x + 1) else Nothing
We can use the >>= operator to sequence these computations:
baz :: Int -> Maybe Int
baz x = return x >>= foo >>= bar

Functors are a type constructor that allow functions to be applied to values inside the functor. Monads are a type constructor that extend the concept of functors by introducing additional operations, specifically return and bind (also written as >>=), which allow values to be "boxed" and composed together.
In summary:

- A monad is a design pattern in functional programming that encapsulates a sequence of operations on a data type.
- A monad is a type that defines two operations: return and bind.
- The return operation takes a value of a given type and returns a monad that contains that value.
- The bind operation takes a monad containing a value of type a, and a function that takes a value of type a and returns a monad containing a value of type b. It applies the function to the value inside the monad, and returns a new monad containing the result.
- Monads provide a way to sequence operations that involve side effects, such as I/O or state changes.
- Monads are used extensively in Haskell, particularly in areas such as parsing, I/O, and concurrency.
- Some examples of monads in Haskell include the Maybe monad, the Either monad, and the IO monad.



The IO Monad in Haskell is a way to perform input and output operations in a purely functional language like Haskell. In a purely functional language, it is not possible to perform side effects like input and output in the traditional sense, since functions should not have any observable side effects.
The IO Monad provides a way to sequence IO actions in a way that is guaranteed to be safe and purely functional. The `main` function in a Haskell program is an IO action, and all IO actions are executed within the IO Monad.

Java 8 The eighth version introduces **functional programming** through **Lambda expressions**. Concurrently to lambdas, they releases also the **Stream API**, that helps in using lambdas without recompiling existing binaries.
Java 8 is a version of the Java programming language that was released by Oracle Corporation in March 2014. It is one of the most significant releases of Java, and it introduced several new features and improvements to the language, including:
Lambda expressions: Lambda expressions allow developers to write more concise and functional code by enabling the use of functional interfaces, which represent a single abstract method.
Stream API: The Stream API provides a functional way to process collections of objects, allowing for parallel processing and filtering of large datasets.
Type annotations: Type annotations allow developers to specify annotations on any type, including generic types, enabling better type checking and improved code analysis.

Lambda expression

Benefits of Lambdas

- Enables functional programming in Java
- Allows to pass **behaviors** as well as data to functions
- Allows **lazyness in stream** processing
- Facilitates **parallel programming**
- Provides more generic, reusable and flexible APIs

Examples:

- lambda: `list.forEach(x -> System.out.println(x));`
- method reference: `list.forEach(System.out::println);`
- block/multiline: `list.forEach(x -> { System.out.println(x); });`

1- The code list.forEach(x -> System.out.println(x)); is an example of using a lambda expression to iterate over a list and print its elements to the console

2- 2- Method reference is a feature introduced in Java 8 that allows you to refer to an existing method or constructor by its name. It provides a shorthand way to create a lambda expression that calls an existing method or constructor.

3- 3- A block or multiline lambda expression in Java is a lambda expression that contains more than one statement or requires additional curly braces to group the statements. This allows for more complex behavior to be defined in the lambda expression.

Things to know:

All the variable used in a lambda must be final (by declaration) or effectively final.
Can also be static field of the class, but for use them we need to be in a static method (e.g. main).

Backward compatibility To maintain backward compatibility is not possible to add new abstract methods to an interface. For that reason, Java 8 allows existent interfaces to include abstract methods, static methods and default methods defined in term of other abstract methods. Hence, the backward compatibility for the Java Collections Framework is granted through lambda espressions and default methods.

As we know interfaces has 1 abstract method. If we wan to add other methos ,java 8 define default method .

Stream API

The `java.util.stream` package provides utilities to support functional-style operation on streams of values.

Stream vs Collection

A stream is not a data structure, hence doesn't store elements. It conveys elements from a source (data structure, generator function, I/O) through a pipeline of operations.
It produces a result without modifying its source.

Stream properties

- Laziness-seeking: operations are divided into intermediate (stream-producing) and terminal (value-producing). All the intermediate function are lazy, and are evaluated only when a result is needed by the following operation.
- Unbounded: while collections are finite, streams are possibly not.
- Consumable: stream's elements are visited only once. Then another stream must be generated from the same source, as for the iterators.

— A **source**, producing (by need) the elements of the stream

— Zero or more **intermediate opera/ons**, producing streams

— A **terminal opera/on**, producing side-effects or nonstream

values Intermediate methods are not performed untile the terminal method is called. At this point the stream is consumed and is not possible to execute other operations on that. Short-circuit intermediate methods can cause the erlier intermediate methods to be processed until is not possible to process the short-circuit method. An example of short-circuit method is a filter, that cause the procession of items until the current one is not a suitable output.

In a pipeline of intermediate operations, such as map, filter, and reduce, each operation transforms the data and passes it to the next operation in the pipeline. When a short-circuiting operation, such as filter, is encountered in the pipeline, it stops processing the remaining items as soon as the result can be determined.

For example, in the case of the filter operation, the intermediate items are processed one by one, and if an item does not satisfy the filtering condition, it is immediately discarded, and the next item is processed. This means that the remaining items in the pipeline are not processed unnecessarily, which can greatly improve performance for large datasets.

Parallel streams

Stream operations can be executed in serial (default) or in parallel using `.parallelStream()`.
The Runtime Support takes care of using multithreading in a transparent way.
Thread safety is guaranteed also on non-thread-safe collections, unless some operations have side-effects.
To process items in parallel the source must be efficiently splittable. Most of Collections are, but I/O is not.
Moreover, intermediate operations should be stateless and not affect the source (non-interfering behaviour).
For parallel streams with side-effects, ensuring thread safety is the programmers' responsibility.

Stream implementation Java 8 Streams are implemented with the Spliterator, that is the parallel analogue of an Iterator.
As an iterator it has methods to sequentially advancing and to apply an action to the next item, but has also the ability of splitting off some portion of the input into another spliterator which can be processed in parallel. In Java 8 and later versions, streams are implemented using the Spliterator interface, which is the parallel equivalent of an Iterator.
A Spliterator provides a way to traverse and partition elements of a collection in a parallel manner. It has two main methods:
tryAdvance(Consumer<? super T> action): This method sequentially advances the Spliterator, applying the specified action to the next element if there is one. If there is no next element, it returns false.
trySplit(): This method splits the elements of the Spliterator into two parts, returning a new Spliterator for the second part. This new Spliterator can then be processed in parallel.
Streams use the Spliterator to divide the input data into smaller chunks that can be processed in parallel by multiple threads. The Spliterator tries to evenly divide the input data, but it may not be able to do so if the input data is not easily divisible.
By splitting the input data and processing it in parallel, streams can take advantage of multiple processor cores, resulting in improved performance and faster processing times. Additionally, streams can be used to process large datasets in a memory-efficient manner, as the data is processed in a lazy and on-demand fashion.

Monads in Java Monads in Java are implemented with Optional value and Streams. An Optional value is an object that can or not have a value, e.g. `Optional<Integer>` can have an integer value. The `isPresent()` method returns true if the object has a value. The `get()` method returns the value if it is present, otherwise throws `NoSuchElementException`.
In a similar way, is possible to obtain stream with one single element, or empty. The `flatMap` method allows to map elements in a Stream on in an Optional with other elements, returning a new Stream or Optional.