DZone

[New Guide] Download the 2018 Guide to IoT: Harnessing Device Data

Download Guide▸

JλVλ 8 (A Comprehensive Look)

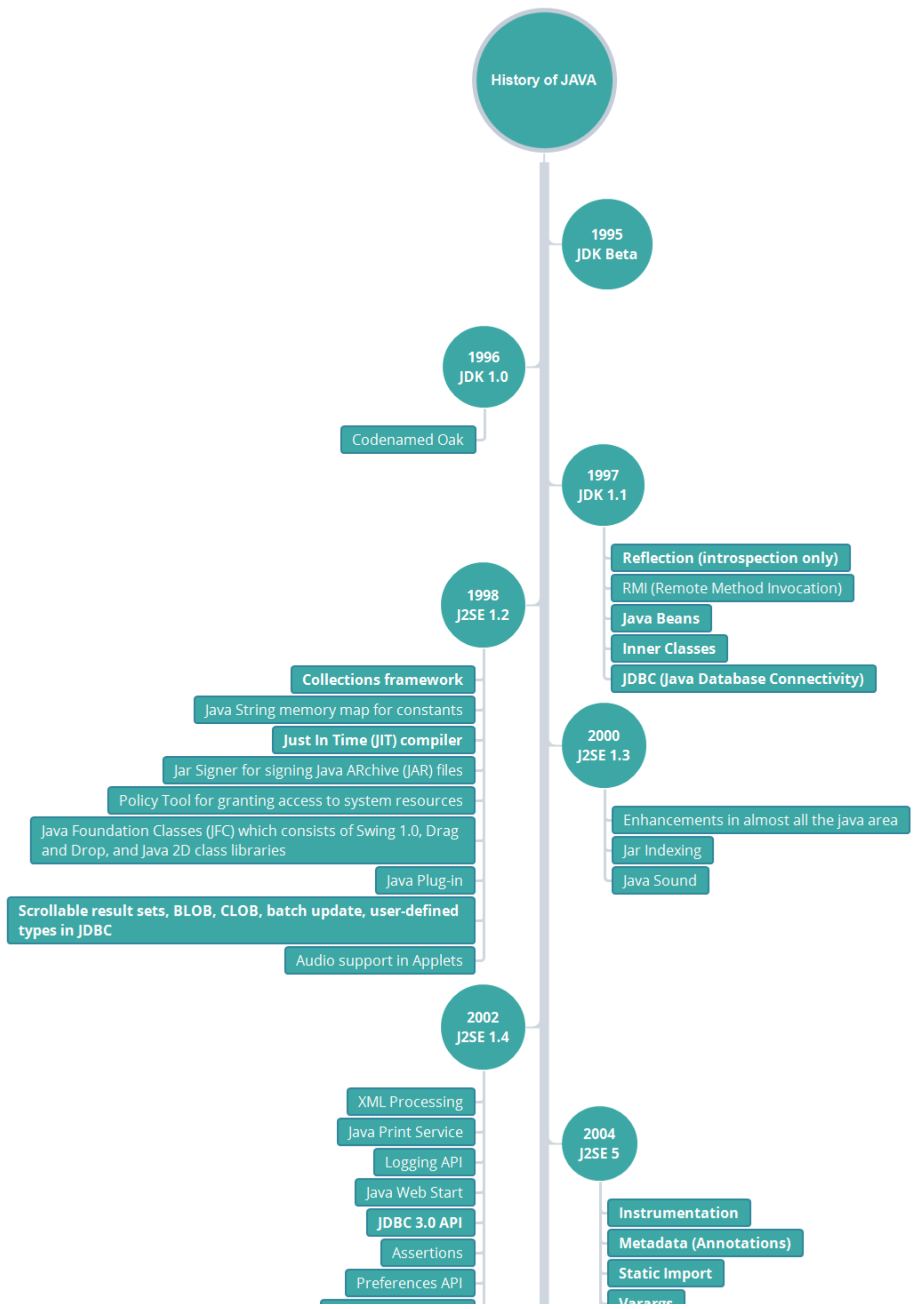
by Robin Rizvi · Apr. 16, 18 · Java Zone · Tutorial

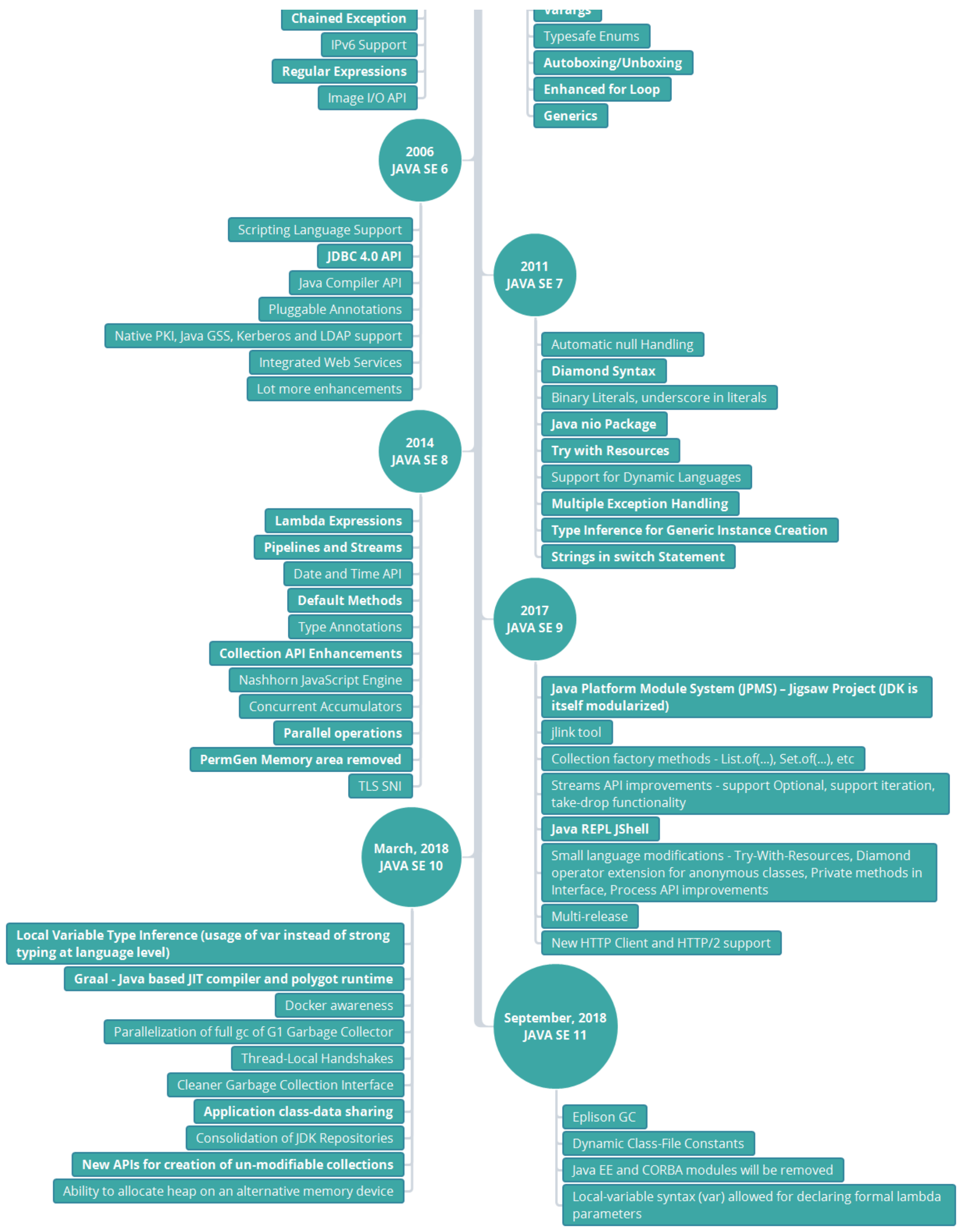
Get the Edge with a Professional Java IDE. 30-day free trial.

Other articles in this series:

- JλVλ 8 (A Comprehensive Look): Part 1.1 - Lambdas Under the Hood
- JλVλ 8 (A Comprehensive Look): Part 1.2 - Insight Into Streams

Let's start from the start and have a look at the journey of Java.

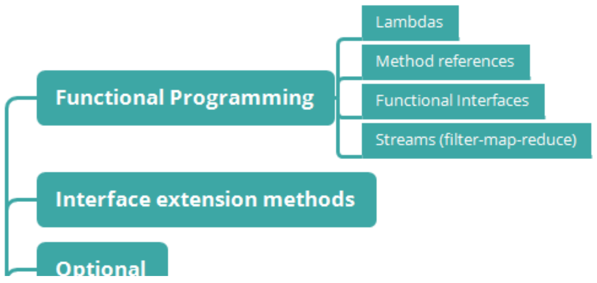

History of JAVA
1995 JDK Beta
1996 JDK 1.0
Codenamed Oak
1997 JDK 1.1
Reflection (introspection only)
RMI (Remote Method Invocation)
Java Beans
Inner Classes
JDBC (Java Database Connectivity)
1998 J2SE 1.2
Collections framework
Java String memory map for constants
Just In Time (JIT) compiler
Jar Signer for signing Java ARchive (JAR) files
Policy Tool for granting access to system resources
Java Foundation Classes (JFC) which consists of Swing 1.0, Drag and Drop, and Java 2D class libraries
Java Plug-in
Scrollable result sets, BLOB, CLOB, batch update, user-defined types in JDBC
Audio support in Applets
2000 J2SE 1.3
Enhancements in almost all the java area
Jar Indexing
Java Sound
2002 J2SE 1.4
XML Processing
Java Print Service
Logging API
Java Web Start
JDBC 3.0 API
Assertions
Preferences API
2004 J2SE 5
Instrumentation
Metadata (Annotations)
Static Import

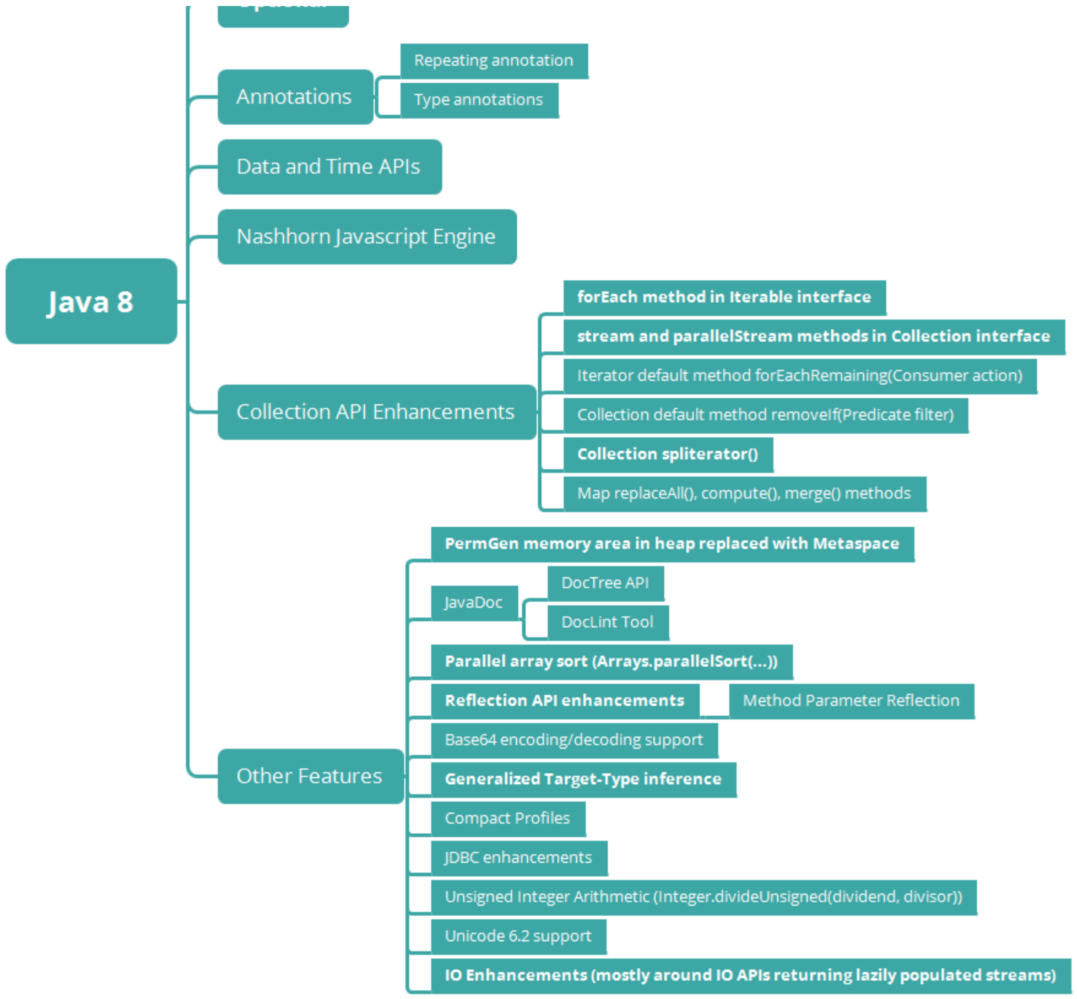

Chained Exception
IPv6 Support
Regular Expressions
Image I/O API
Typesafe Enums
Autoboxing/Unboxing
Enhanced for Loop
Generics
2006 JAVA SE 6
Scripting Language Support
JDBC 4.0 API
Java Compiler API
Pluggable Annotations
Native PKI, Java GSS, Kerberos and LDAP support
Integrated Web Services
Lot more enhancements
2011 JAVA SE 7
Automatic null Handling
Diamond Syntax
Binary Literals, underscore in literals
Java nio Package
Try with Resources
Support for Dynamic Languages
Multiple Exception Handling
Type Inference for Generic Instance Creation
Strings in switch Statement
2014 JAVA SE 8
Lambda Expressions
Pipelines and Streams
Date and Time API
Default Methods
Type Annotations
Collection API Enhancements
Nashhorn JavaScript Engine
Concurrent Accumulators
Parallel operations
PermGen Memory area removed
TLS SNI
2017 JAVA SE 9
Java Platform Module System (JPMS) – Jigsaw Project (JDK is itself modularized)
jlink tool
Collection factory methods - List.of(...), Set.of(...), etc
Streams API improvements - support Optional, support iteration, take-drop functionality
Java REPL JShell
Small language modifications - Try-With-Resources, Diamond operator extension for anonymous classes, Private methods in Interface, Process API improvements
Multi-release
New HTTP Client and HTTP/2 support
March, 2018 JAVA SE 10
Local Variable Type Inference (usage of var instead of strong typing at language level)
Graal - Java based JIT compiler and polygot runtime
Docker awareness
Parallelization of full gc of G1 Garbage Collector
Thread-Local Handshakes
Cleaner Garbage Collection Interface
Application class-data sharing
Consolidation of JDK Repositories
New APIs for creation of un-modifiable collections
Ability to allocate heap on an alternative memory device
September, 2018 JAVA SE 11
Eplison GC
Dynamic Class-File Constants
Java EE and CORBA modules will be removed
Local-variable syntax (var) allowed for declaring formal lambda parameters

Note: The Java 11 features mentioned above are not the finalized ones to be included in the Java 11 release. This list just mentions the features that most probably will be included or removed. Also, note that Graal — the experimental JIT compiler listed under Java 10 — was already added in Java 9, but it could not be enabled through JVM arguments back then. For more information on Graal, have a look at Graal, Sulong - LLVM bitcode interpreter, JVMCI - JVM Compiler Interface, Truffle, SubstrateVM - AOT compilation

Key Features of Java 8

Moving ahead, let's have a look at the keys features that came in Java 8:




Java 8
Annotations
Repeating annotation
Type annotations
Data and Time APIs
Nashhorn Javascript Engine
Collection API Enhancements
forEach method in Iterable interface
stream and parallelStream methods in Collection interface
Iterator default method forEachRemaining(Consumer action)
Collection default method removeIf(Predicate filter)
Collection spliterator()
Map replaceAll(), compute(), merge() methods
Other Features
PermGen memory area in heap replaced with Metaspace
JavaDoc
DocTree API
DocLint Tool
Parallel array sort (Arrays.parallelSort(...))
Reflection API enhancements
Method Parameter Reflection
Base64 encoding/decoding support
Generalized Target-Type inference
Compact Profiles
JDBC enhancements
Unsigned Integer Arithmetic (Integer.divideUnsigned(dividend, divisor))
Unicode 6.2 support
IO Enhancements (mostly around IO APIs returning lazily populated streams)

Functional Programming

The major portion of Java 8 was targeted toward supporting functional programming. Let's explore and understand what functional programming means and how it is useful and is applied in Java.

Functional programming is a programming paradigm that dictates a different algorithmic way to think about problems and program solutions for them. T contrast it with object-oriented programming, in OOP, the primary abstraction is Class/Object, whereas in functional programming the primary abstraction is a function. As in OOP, objects form the building blocks for computation. Similarly, in functional programming, functions form the building blocks for computation.

In one line, "**In functional programming, all computation is carried out through the execution of (mathematical) functions**." These (mathematical) functions have the characteristic property of not changing any state and only operating on inputs, i.e there is no side-effect on their execution.

Functional programming imposes a declarative programming style, i.e we program through expressions or declarations (computation is expressed in terms of functions/expressions — black boxes). The declarative programming style is all about defining what to do and not how to do it, contrasting it with imperative programming, which is all about defining what to do and also how to do it. In simpler terms, in declarative programming, it's all about having a layer of abstraction that results in expressive code. Programming in a functional style tends to be declarative, and thinking and coding declaratively tends to lead to better, readable, and expressive code. An example of declarative vs. imperative styles is:

```
public class StringFinder {
  public static boolean findImperative(List<String> strings, String string) {
    boolean found = false;

    for (String str : strings) {
      if (str.equals(string)) {
        found = true;
        break;
      }
    }

    return found;
  }

  public static boolean findDeclarative(List<String> strings, String string) {
    return strings.contains(string);
  }
}
```

Lambda Calculus

Functional programming **has its roots in mathematics** and draws most of the concepts from its mathematical counterpart: **Lambda Calculus**. All functional programming constructs used in programming languages are, in a sense, **implementations/elaborations of lambda calculus**. Let's turn to mathematics for a bit.

(From Wikipedia) **Lambda calculus (λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.** The λ-calculus provides a simple semantics for computation, enabling properties of computation to be studied formally. The **λ-calculus incorporates two simplifications** — The first simplification is that the λ-calculus treats functions "anonymously," without giving them explicit names and the second is that the λ-calculus only uses functions of a single input. An ordinary function that requires two or more inputs would require currying.

Simplified Lambda expression syntax: `λvar.expr|var|(expr)` . Here, `λvar` is the lambda declaration with am input variable `var` , `.expr|var` is the function definition/body, and `(expr)` is the function application (or function call in simpler terms).

Eg: `λx.x2(7)` would result in 49. If this was expressed as lambda in Java or other languages, it would be `x→x^2`

In lambda calculus, **functions are 'first-class' citizens**, i.e functions can be used as inputs or be returned as outputs from other functions. First-class functions are discussed in more detail later.

Another interesting thing for me was that Church also defined a Church encoding scheme, which could represent data and operators in lambda calculus. Natural numbers could be represented through lambda calculus under the Church encoding scheme, and those representations are termed as **Church numerals**. The Church numeral *n* is a higher-order function that takes a function *f* as an argument and returns the *n*th composition of *f*, i.e. the function *f* composed with itself *n* times. Eg: Number `3 := λf.λx.f (f (f x))`

Why Do We Need Functional Programming in Java?

- One of the biggest advantages of functional programming is that since computation/programs are expressed through (mathematical) functions, which do not change state or have any side effects and have the same output for each input, **this fits naturally with concurrency and parallelism applicability**. And as we are moving towards more cores and distributed/parallel computing, FP proves to be a more natural fit for such requirements. This is one of the reasons why functional languages or functional constructs are used to code AI/ML or big data solutions. Eg: Apache Spark is a big data platform coded in Scala, which is a (not purely) functional language. Another example is R, the most popular language amongst data scientists, which is functional.
- Following functional style **tends to result in declarative code** that is more concise and readable.
- Also since the code is to be written as mere functions **without considering state, synchronization, and other concerns, one can focus more on the problem** rather than code and its structuring itself.

What's the Catch?

On the other hand, having no side effects or not being able to change/mutate state isn't all good, **it is a serious limitation that restricts the ability to easily and efficiently code systems that perform IO**. Functional languages like Haskell tend to work around this problem by differentiating between pure functions (mathematical functions) and impure functions (that don't follow the rules of mathematical functions, i.e they may change state, have a different output for a specific input, print output, etc). Most of the IO actions that are present in any other language are present, but they are kept separated from the functional core module of the language.

On the other hand, we need to evaluate Java and ask ourselves that why would we even turn to functional programming. What are the gaps that limit us?

Let's turn our heads toward OOP for a bit and see what constraints it imposes.

As we know, in an object-oriented programming, the primary unit is an object (defined by a class), which encapsulates the state and behavior of the object. Since everything is an object, to model a system or a solution, one has to always think and provide implementations based on objects. **If there are cases where we need to implement only some behavior/operation and no state, OOP puts a constraint on wrapping that behavior in a Class to be able to execute that.** This leads to too much boilerplate and verbose code where the computation just requires the execution of a function. Such scenarios are fairly simple and natural to handle in functional programming.

As an example, some such cases would be implementing comparison logic (non-reusable) and executing it, writing some code that should be executed in a separate thread once, etc. There can be a plethora of such cases where, whenever we want to execute a block of code, we need to unnecessarily wrap that up in a class.

Java team kind of identified this problem early and **added the support of anonymous inner classes in JDK 1.1** so that this problem could be remedied a little with what could be easily supported back then.

So the answer to this “WHY?” is simply that, as discussed, both paradigms have their goods and bads, so Java chose to incorporate the best of both worlds. It chose to follow a **mixed programming paradigm** to support programming in both object-oriented and functional styles and even combining them in ways that dissolve their disadvantages that we discussed before.

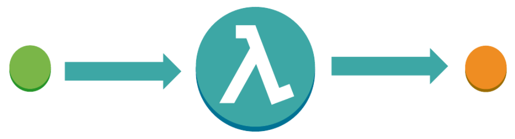
Also, another impetus for this decision was so its developer base did not move or switch to other platforms or languages to leverage functional programming's advantages. That way, the language stayed competitive and productive in comparison to other programming languages and platforms on the market.

The overall objective is to be able to write plain functions/operations (mere computation blocks) in a concise and natural manner without being constrained by the programming language or the programming platform to write more verbose, boilerplate, or imperative code because of the limitations imposed by the programming language.

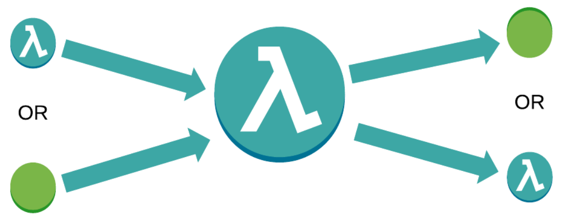
Key Concepts in Functional Programming

Let's have a look at the key concepts in functional programming so that we can relate to the constructs made available in Java 8 to support it.

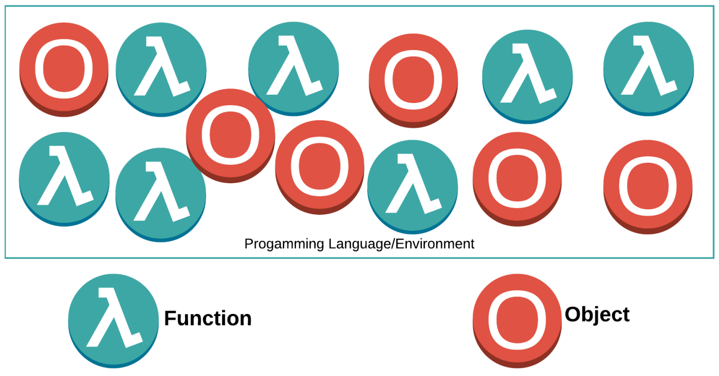
- **Function**: In mathematics, a function is a mapping between input and output. It can be seen as a "black box" that transforms inputs to outputs. There are certain key characteristics of functions that we need to observe:
    - Functions avoid changing state and mutating data. All state that they observe is only the input provided to them
    - Functions don’t change the value of the inputs.
    - For each input, there is a specific output. For example, with f(x) == f(x), the result of function *f* with input *x* is always the same.



- **Higher-order functions**: A function that takes one or more functions as arguments and/or returns a function as its result.

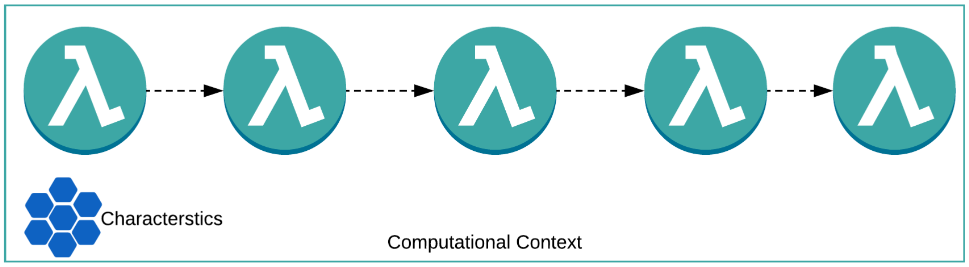


- **First-order functions**: All functions except higher-order functions are termed first-order functions.
- **First-class functions**: A language or system is said to have first-class functions when functions are treated as the primary unit and can exist independently.



- **Monad**: Monads are a structure that represent computations defined as sequences of steps. A type (a class in the Java sense) representing a monad structure defines the linking and sequencing of operations, derivation of characterstics based on the operations and their sequence, and the mechanism to chain that type with other similar types.

mechanism to chain that type with other similar types.



Anonymous Classes and Closure

Anonymous classes in Java are classes with no name. They are just syntactic sugar to avoid having boilerplate code to declare and instantiate a class that is only required for one-time use.

- It is an **inner or local class without a name** and for which only a single object is created.
- **Anonymous classes are expressions**, which means that the class needs to be defined in another statement.
- Anonymous class expression requires **the new operator, the name of an interface to implement or a class to extend, parentheses that contain the arguments to a constructor, and a class declaration body.**
- It cannot have constructors (instance initializers are used instead, as required)
- Since anonymous classes are like local classes, they also support capturing variables (they behave like closures — **a closure is a block of code that can be referenced and passed around with access to the variables of the enclosing scope**).
- **Only local variables that are final or effectively final can be accessed** (this also applies to Lambdas – to be discussed later — and also falls along the principles of functional programming where state could be changed by a function if we assume an anonymous class is just used as a replacement of a block of code without any state).

As an example, check out the class below:

```
public class Closure {
  public void publicMethod() {
    Integer localVariable1 = 10;
    Integer localVariable2 = 10;
    Integer localVariable3 = 10;

    Map<String, Integer> map = new HashMap<String, Integer>() {
      {
        put("a", localVariable1);
        put("b", localVariable2);
        put("c", localVariable3);
      }
    };

    Thread t = new Thread(new Runnable() {
      public void run() {
        System.out.println(localVariable1);
      }
    });

    List<String> list = Arrays.asList("A", "B", "C");

    Collections.sort(list, new Comparator<String>() {
      public int compare(String p1, String p2) {
        return p1.compareTo(p2);
      }
    });
  }
}
```

Let's have a look under the hood and see what this class gets compiled into. After compiling the Anony class mentioned in the example above (javac

Anony.java), 4 class files are generated by the Java compiler. As I mentioned earlier, since anonymous classes are syntactic sugar, it is translated to appropriate artifacts to function correctly. If we have a look at the compiled bytecode, we can easily see how it is compiled into classes and how variable capture/closures work:

`javap -p Anony.class`

```
public class com.java8exploration.Anony {
  public com.java8exploration.Anony();
  public void publicMethod();
}
```

`javap -p Anony$1.class` (If we see the code in Anony.java, we can see that we are referring to 3 local variables. If we see the compiled code, we can see that the **variables are being generated into the Anony$1 class**. Also, its **constructor contains those referenced variables** as arguments and **this is how variable capture/closure behavior works**.

```
class com.java8exploration.Anony$1 extends java.util.HashMap<java.lang.String, java.lang.Integer> {
  final java.lang.Integer val$localVariable1;
  final java.lang.Integer val$localVariable2;
  final java.lang.Integer val$localVariable3;
  final com.java8exploration.Anony this$0;
  com.java8exploration.Anony$1(com.java8exploration.Anony, java.lang.Integer, java.lang.Integer, java.lang.Integer);
}
```

`javap -p Anony$2.class`

```
class com.java8exploration.Anony$2 implements java.lang.Runnable {
  final java.lang.Integer val$localVariable1;
  final com.java8exploration.Anony this$0;
  com.java8exploration.Anony$2(com.java8exploration.Anony, java.lang.Integer);
  public void run();
}
```

`javap -p Anony$3.class`

```
class com.java8exploration.Anony$3 implements java.util.Comparator<java.lang.String> {
  final com.java8exploration.Anony this$0;
  com.java8exploration.Anony$3(com.java8exploration.Anony);
  public int compare(java.lang.String, java.lang.String);
  public int compare(java.lang.Object, java.lang.Object);
}
```

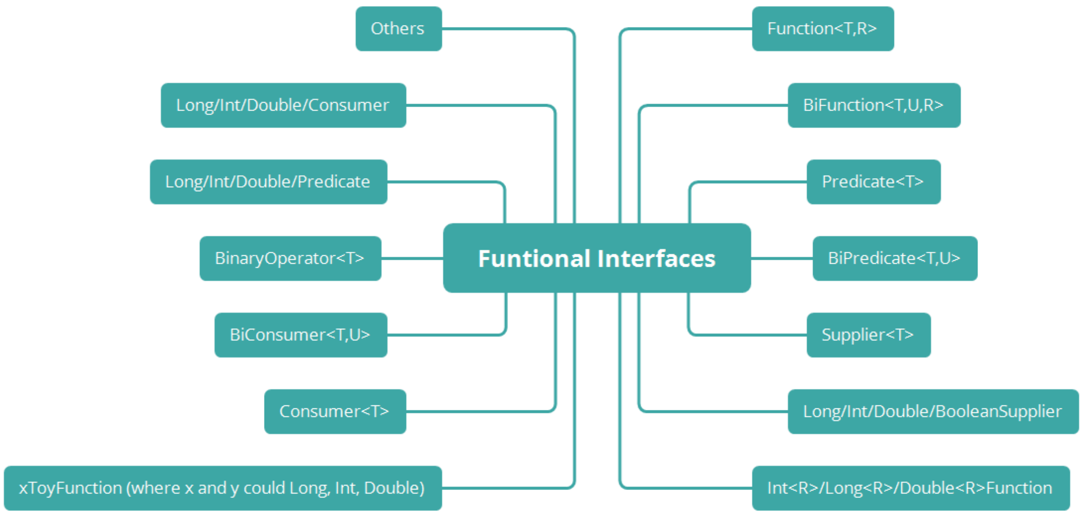
Functional Interfaces

Let us now have a look at Funtional Interface in Java 8.

- A Functional Interface is an **interface in Java that has only one abstract method**
- These interfaces are sometimes referred to as **SAM** (Single-Abstract-Method) types
- **Lambda expressions only work with functional interfaces**. So that means that anywhere a function interface is expected, we can use a lambda expression (also method references can be used).
- Functional Interfaces are a **key construct that allow Java, being an object-oriented platform, to support functional programming** (by serving as a type wrapper for a function) and to **deal with functions**.
- There are already many interfaces in Java libraries (and other libraries_ that would have only one method, but since Java needs to incorporate functional programming concepts into its object-oriented core, it needs this **abstraction of a functional interface, which is a wrapper of only one method. That, in effect, represents an executable function in a functional programming language**. So Java 8 just formalizes this idea via this construct.
- **The @FunctionalInterface** annotation is also provided to tag such interfaces so that the compiler can give warnings/errors if there is any discrepancy in defining a functional interface.
- Since functional interfaces are a way to represent functions, Java provides numerous interfaces for representing different function types, which are hosted under **java.util.function package**. Note that functional interfaces can take generic parameters, which supports defining any function's input/output types

- Since a primitive type can't be a generic type argument, there are **versions of the Functional Interfaces for the most-used primitive types** like double, int, long, and their combinations of argument and return types.

There are many functional interfaces available in java.util.function package that we can use readily to represent our type of function, but if you don't find a Functional Interface to represent your custom function, you can always create your own Functional Interface.



Lambda Expressions

Here comes the heart of functional programming in Java. Lambda expressions are the primary construct in Java to provide a new syntax in the language to allow us to express functions in a natural way.

- **Lambda expressions are the realization of functional programming principles** in an object-oriented platform.
- **Lambda expressions are used to express a function directly in code without the need of object-oriented wrappers** to support them (from the language syntax point of view). In terms of methods, it can be viewed as an anonymous method.
- Since lambdas follow the semantics of a function, as in a functional programming language, **they get all the parallelism and concurrency advantages** that could be drawn from immmutable, consistent functions.
- **Lambda expressions can be used in code wherever a functional interface is expected**, which actually means that, internally, lambdas are implementations of functional interfaces and thus are first-class citizens in the language. They can be assigned/stored, passed as an argument, etc.
- **Lambdas can replace anonymous classes if** the anonymous class is an implementation of an interface that contains only one method (in such cases, the intent really is that you are trying to use a function as an argument), then the anonymous class can be replaced with a lambda, which would result in clean and less verbose code.

Let's now **compare lambda expressions in Java to constructs in lambda calculus** or functional programming:

- **Functions are anonymous – lambdas expressions are anonymous** as well
- **Functions do not change state – lambda expressions follow the same closure rules** as anonymous/local classes, i.e they can access only final or effectively final variables
- Functions may be used as inputs or be returned as outputs from other functions
- Lambda calculus' syntax, λx.x2, matches that of a lambda expression x→x^2. **λx becomes the argument part, dot becomes the arrow, and expr body becomes the lambda expression's body here.**

Lambda Expression Syntax

Let us have a look at the language syntax that is used to express different types of functions in code.

- Specify **function parameters**, parameter type is optional, parenthesis can be omitted if there is only one parameter
- The **arrow operator** to separate the parameters and body
- **Function body**, parenthesis and return can be omitted if there is only one statement

Syntax examples:

- No parameter: `() →System.out.println("Nothing")`
- One parameter: `x →x+2`
- Two parameters: `(x,y) →x+y`
- With parameter types: `(Integer x, Integer y) →x+y`

- Multiple statements:

```
(x,y) → {
    System.out.println(x);
    System.out.println(y);
    return x+y;
}
```

Let's compare code with Lambda expressions against traditional code to see the differences and advantages of using lambda expressions in code:

Traditional Code	**Java 8 Code (with lambda expressions)**

`Thread t = new Thread(new Runnable() {`
`  @Override`
`  public void run() {`
`    System.out.println("From a thread");`
`  }`
`});`
	`Thread t = new Thread(() -> System.out.println("From a thre`

`Collections.sort(Arrays.asList(3, 2, 1), new Comparator<Integer>(`
`  @Override`
`  public int compare(Integer o1, Integer o2) {`
`    return o1.compareTo(o2);`
`  }`
`});`
	`Collections.sort(Arrays.asList(3, 2, 1), (o1, o2) -> o1.com`

`List<String> strings = Arrays.asList("R", "O", "B");`
`for (String string : strings) {`
`  System.out.println(string);`
`}`
	`List<String> strings = Arrays.asList("R", "O", "B");`
	`strings.forEach(s -> System.out.println(s));`
	
	`// or use method reference if there is a direct execution o`
	`strings.forEach(System.out::println);`

More details on Lambda Expressions and its "under the hood" implementation in Java will be discussed in Part 1.1 of this article.

Method References

Lambda expressions are used to express a function's body, but **if you have the function/method already written/defined, you can directly use them as method references wherever lambda expressions could be used**. We can see lambda expressions and methods having the same usage, it's just that the first is anonymous whereas the latter is named. So if we have a method already defined, method references allow us to use that method by name in place of anonymous lambda expressions.

There are **four kinds** of method references (from Oracle docs):

- **Reference to a static method** — ContainingClass::staticMethodName
- **Reference to an instance method of a particular object** — containingObject::instanceMethodName
- **Reference to an instance method of an arbitrary object of a particular type** _ ContainingType::methodName
- **Reference to a constructor** — ClassName::new

Let's see examples of the four kinds of method references:

```
public class MethodRef {
  public void printLowerCase(String s) {
    System.out.println(s.toLowerCase());
  }
  public static void printUpperCase(String s) {
    System.out.println(s.toUpperCase());
  }
  public void publicMethod() {
    List<String> list = Arrays.asList("A", "B", "C");
    list.forEach(this::printLowerCase);
    list.forEach(MethodRef::printUpperCase);
    list.forEach(String::toLowerCase);
    list.forEach(String::new);
  }
}
```

In the above example:

- `list.forEach(this::printLowerCase)` is **an example of a method reference to an instance method of a particular object**. Note that there is one difference if you are using a lambda expression here instead of this method reference: You can only access final or effectively final fields when using a lambda expression whereas with this method reference, you can access any field.
- `list.forEach(MethodRef::printUpperCase)` is **an example of a method reference to a static method.**
- `list.forEach(String::toLowerCase)` is **an example of a method reference to an instance method of an arbitrary object of a particular type**.
- `list.forEach(String::new)` is **an example of a method reference to a constructor**.

Streams

Streams in Java allow for defining a pipeline of operations that can carry out the transformation of input data into the required form. Streams allow for executing operations on any data source that can provide access to its elements as required by the Streams API. Most commonly, it would be any Collection (? extends Collection) in Java. Collection was supercharged in Java 8 to support streams. Other than that, one can use any of the factory methods or generator methods to construct a stream-compatible data source.

Streams provide the capability of **processing data declaratively** instead of doing it imperatively. Streams also allow **applying the filter-map-reduce pattern** (which is available in most of the languages that support functional programming) to collections. Streams allow for easy and seamless **sequential and parallel execution** of operations.

Stream is not a data structure or a collection, it is a declarative expression of operations that would be performed on a data source.

In functional programming terminology, **streams are essentially Monads**, which means that it is a structure that represents computations defined as sequences of steps. A type (class, in a Java sense) representing a monad structure defines the linking and sequencing of operations, derivation of characterstics based on the operations and their sequence, and a mechanism to chain that type with other similar types.

A Stream execution flow can be compared to an Iterator (to have a code analogy) – the values flow by, and when they're consumed (once the stream is executed through a terminal operation), they are no longer usable.

A Stream pipeline is composed of three levels:

(Stream methods reference from Java Streams, Part 1)

- **Source:** Represents the data source that feeds the data items to a process. Common sources are:
  - **Collection.stream()** — Create a stream from the elements of a collection
  - **Stream.of(T...)** — Create a stream from the arguments passed to the factory method
  - **Stream.of(T[])** — Create a stream from the elements of an array
  - Stream.empty() — Create an empty stream
  - IntStream.range(lower, upper) — Create an IntStream consisting of the elements from lower to upper, exclusive
  - IntStream.rangeClosed(lower, upper) — Create an IntStream consisting of the elements from lower to upper, inclusive.
- **Zero or more intermediate operations:** Intermediate operations return a stream so we can chain multiple intermediate operations. **An important characteristic of intermediate operations is laziness. Intermediate operations will only be executed when a terminal operation is present**. Common intermediate operations are:
  - **filter(Predicate<T>)** — The elements of the stream matching the predicate
  - **map(Function<T, U>)** — The result of applying the provided function to the elements of the stream
  - flatMap(Function<T, Stream<U>> — The elements of the streams resulting from applying the provided stream-bearing function to the elements of the stream
  - distinct() — The elements of the stream, with duplicates removed
  - sorted() — The elements of the stream, sorted in natural order
  - Sorted(Comparator<T>) — The elements of the stream, sorted by the provided comparator
  - limit(long) — The elements of the stream, truncated to the provided length
  - skip(long) — The elements of the stream, discarding the first N elements
- **Single terminal operation:** Terminal operations are either **void or return a non-stream result. They cause the stream pipeline to execute** and return the result of the execution of all the applied operations. Common terminal operations are:
  - **forEach(Consumer<T> action)** — Apply the provided action to each element of the stream
  - toArray() — Create an array from the elements of the stream

- toArray() — Create an array from the elements of the stream
- **reduce(...)** — Aggregate the elements of the stream into a summary value
- **collect(...)** — Aggregate the elements of the stream into a summary result container
- min(Comparator<T>) — Return the minimal element of the stream according to the comparator
- max(Comparator<T>) — Return the maximal element of the stream according to the comparator
- count() — Return the size of the stream
- {any,all,none}Match(Predicate<T>) — Return whether any/all/none of the elements of the stream match the provided predicate. **This is a short-circuiting operation**
- findFirst() — Return the first element of the stream, if present. **This is a short-circuiting operation**
- findAny() — Return any element of the stream, if present.

Let's see an example the **three levels of a stream pipeline**:

```
Stream.of("1", "2", "3", "4", "5") // source
.filter(s -> s.startsWith("1")) // intermediate operation
.map(String::toUpperCase) // intermediate operation
.sorted() // intermediate operation
.forEach(System.out::println); // terminal operation
```

Let's see through some examples of **parallel Stream execution**:

```
Arrays.asList("1", "2", "3", "4", "5")
.parallelStream() // parallel processing
.filter(s -> s.startsWith("1"))
.map(String::toUpperCase)
.sorted()
.forEach(System.out::println);
```

```
Stream.of("1", "2", "3", "4", "5")
.parallel() // parallel processing
.filter(s -> s.startsWith("1"))
.map(String::toUpperCase)
.sorted()
.forEach(System.out::println);
```

Let's see some other examples of Stream API usage:

```
Stream.of("1", "2", "3", "4", "5")
.mapToInt(Integer::valueOf)
.sum();
// Output: 15

Stream.of("1", "2", "3", "4", "5")
.mapToInt(Integer::valueOf)
.reduce(0, (x,y) -> x+y);
// Output: 15

Stream.of("R", "O", "B", "I", "N")
.collect(Collectors.joining());
// Output: ROBIN

Stream.of("R", "O", "B", "I", "N")
.reduce("", (a,b)->a+b);
// Output: ROBIN
```

More insights into Streams and its "under the hood" implementation in Java, is discussed in Part 1.2 of this article.

Interface Extension Methods

Interface Extension Methods enable **multiple inheritance of functionality** (but not types) through interfaces. It allows adding **method implementations to interfaces** with backward compatibility. It was **essentially required to support functional programming operations** (streams, etc.) and other operations in the Collections API without requiring modifications to several sub-types.

Java does not allow multiple inheritance to avoid the "**Diamond problem**" where if a common method is implemented in parent classes, it would become ambiguous to decide which one to execute in child classes.

But Java does allow multiple inheritance with interfaces because there were no method implementations in there. But now, since interfaces can have method implementations and Java allows for multiple inheritance with interfaces, one can encounter the diamond problem. If so, the compiler will throw an error and force you to override that method.

Having method implementations in the interface also **help to avoid having utility classes**, take for example the Collections utility class. All methods from this could directly be implemented in the Collection interface.

Interfaces allow **adding default and static methods** that do not need to be implemented in the classes that implement the interface. The default keyword is used with the method signature to indicate a default method. Here, default indicates a default implementation that can be overridden by implementing classes. Default methods are also termed as **Defender Methods or Virtual extension methods**. As for static methods, they are part of the interface and cannot be used with implementation class objects.

Common areas in JDK where default methods are used:

- Several default methods like **replaceAll(), putIfAbsent(Key k, Value v) and others, added in the Map interface**
- **The forEach default method added to Iterable** interface
- **The Stream, parallelStream, spliterator methods added in Collection** interface

Let's have a look at the java.lang.Iterable interface:

```
public interface Iterable<T> {
    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

Let's have a look at an example with default and static method implementations in the interface:

```
public interface Extension {
  default void printHello() {
    System.out.println("Hello from default method in interface");
  }

  static void printHello2() {
    System.out.println("Hello from static method in interface");
  }
}

public class ExtensionTest implements Extension {
  public static void main(String[] args) {
    ExtensionTest impl = new ExtensionTest();
    impl.printHello();
    Extension.printHello2();
  }
}
```

Output:

```
Hello from default method in interface
Hello from static method in interface
```

Optional

Optional allows having a **formal typed solution for representing optional values** instead of null references. Optional is part of the java.util package. When using Optional, **null checks and null pointer exceptions at runtime can be avoided**.

Let's try and understand the philosophy behind Optional. Java is a statically typed language that puts focus on strong typing and ensures type safety at compile time so that problems in the code are identified early and not at runtime. So, **as a language, its philosophy has been to fail-fast**. But **having null in the language goes against this philosophy**, and if we have some code that returns null representing empty values, and this is used without proper null checks, then we won't know until runtime if there is some problem with our code.

If we compare it to other programming languages like C#, Kotlin, Groovy, etc, they don't follow the same philosophy of avoiding null altogether and having a formal type to represent optional values (C# has Nullable, but for value types only). Instead, what it does is provide required syntactic sugar in the language to **ease the handling of null values in code and avoid NPEs. They support the safe navigation operator (?.) and elvis operator (?:)/null coalescing operator (??)**.

The safe navigation operator allows you to safely handle null values and avoid NPEs by returning null if any field/method is accessed on a null value. Meanwhile, the elvis or null coalescing operator isa binary operator (can be seen as a variant of the ?: ternary operator) that allows supplying a default value when the left operand is a null value.

On the other hand, **functional languages like Haskell and Scala have the MayBe and Option types** to represent optional values formally. So, Java decided to follow suit and added an Optional type.

Main advantages of Optional:

- Usage of Optional tends to lead to a more **cleaner, readable, and expressive codebase** than one that contains null references and checks.
- If Optional's API is used appropriately, one can **avoid having bloated code with null checks** and stuff and avoid runtime surprises caused by NPEs.
- It serves as a **precautionary indicator to the developer** that the field you are using may optionally contain a value so that they know upfront when seeing an Optional type that some handling is required to properly access that field.
- It allows us to code **better and comprehensible APIs**, wherein the consumers of your API can see the method signature and know right away that an Optional would be returned — and the returned Optional will require them to unwrap it and deal with it appropriately.

One other thing that was not included in Java (but that I would surely love to have) was the safe navigation operator and the null coalescing operator. There is so much code in our projects and other 3rd-party libraries that we use that uses null. Though we could work around this by wrapping those values in an Optional before using it in our code, that's an extra step that has to be performed everywhere we suspect a null value.

Let's **compare code written with Optional against traditional Java code** with an example. The domain model of our example has **three entities — Project, Employee, and Address**, where a project could optionally have a manager assigned of the Employee type, and this manager could optionally have an address specified:

```java
public class Project {
  private String id;
  private String name;
  private Optional<Employee> manager = Optional.empty();
  public Optional<Employee> getManager() {
    return manager;
  }
  public void setManager(Employee manager) {
    this.manager = Optional.of(manager);
  }
  // ...other accessors
}
public class Employee {
  private String id;
  private String name;
  private Optional<Address> address = Optional.empty();
  public Optional<Address> getAddress() {
    return address;
  }
  public void setAddress(Address address) {
    this.address = Optional.of(address);
  }
  // ...other accessors
}
public class Address {
  private String city;
  private String state;
  private String country;
  // ...other accessors and overridden toString
}
```

Also, there are other variants of these classes named ProjectT and EmployeeT to represent traditional domain classes without Optional. Now let's try to **get the address of a manager assigned to a project** using traditional approach and the Optional approach:

Traditional Code:

```
public static void main(String[] args) {
  EmployeeT managerWithNoSpecifiedAddress = new EmployeeT("E1", "Robin");
  ProjectT projectWithAssignedManagerWithNoAddress = new ProjectT("P1",
      "Project 1", managerWithNoSpecifiedAddress);

  String managerAddress = null;
  EmployeeT manager = projectWithAssignedManagerWithNoAddress.getManager();
  if (manager != null) {
    Address addr = manager.getAddress();
    if (addr != null) {
      managerAddress = addr.toString();
    }
  }

  if (managerAddress != null) {
    System.out.println(managerAddress);
  } else {
    System.out.println("No address specified");
  }

  // if we try to access fields directly, we get a NullPointerException
  managerAddress = projectWithAssignedManagerWithNoAddress.getManager()
      .getAddress().toString(); // NullPointerException
  System.out.println(managerAddress);
}
```

Code with Optional:

```
public static void main(String[] args) {
  Employee managerWithNoSpecifiedAddress = new Employee("E1", "Robin");
  Project projectWithAssignedManagerWithNoAddress = new Project("P1", "Project 1",
      managerWithNoSpecifiedAddress);

  // No value is set for the address field of the manager assigned to the project
  managerAddress = projectWithAssignedManagerWithNoAddress.getManager()
      .flatMap(Employee::getAddress).map(Address::toString)
      .orElse("No address specified");
  System.out.println(managerAddress);
}
```

So as we can see above, the code with Optional is shorter and sweeter. We can **chain map, flatMap, and filter methods from the Optional API to avoid null checks in our code altogether**.

Let's **extend this same example above to cover all the functionalities and usages provided by Optional**. Please go through the code below in detail to understand the usage of Optional (code is well commented to indicate the usage of Optional's API methods):

```
public static void main(String[] args) {
    Project projectWithNoAssignedManager = new Project("P1", "Project 1");
    Employee managerWithNoSpecifiedAddress = new Employee("E1", "Robin");
    Project projectWithAssignedManagerWithNoAddress = new Project("P1", "Project 1",
        managerWithNoSpecifiedAddress);
    Address address = new Address("Lucknow", "UP", "India");
    Employee managerWithSpecifiedAddress = new Employee("E1", "Robin", address);
    Project projectWithAssignedManagerWithAddress = new Project("P1", "Project 1",
        managerWithSpecifiedAddress);

    // No value for manager field is set for the project
    String managerAddress = projectWithNoAssignedManager.getManager()
        .flatMap(Employee::getAddress).map(Address::toString)
        .orElse("No address specified");
    System.out.println(managerAddress); // No address specified

    // No value is set for the address field of the manager assigned to the project
    managerAddress = projectWithAssignedManagerWithNoAddress.getManager()
        .flatMap(Employee::getAddress).map(Address::toString)
        .orElse("No address specified");
    System.out.println(managerAddress); // No address specified

    // All values are available
    managerAddress = projectWithAssignedManagerWithAddress.getManager()
        .flatMap(Employee::getAddress).map(Address::toString)
        .orElse("No address specified");
    System.out.println(managerAddress); // Lucknow, UP, India

    // Execute some code only if manager field has some value
    // with inline lambda
    projectWithAssignedManagerWithAddress.getManager()
        .ifPresent(m -> System.out.println(m.getName())); // Robin
    // with method reference
    projectWithAssignedManagerWithAddress.getManager()
        .ifPresent(OptionalTryOut::printName); // Robin
    // with plain java code
    if (projectWithAssignedManagerWithAddress.getManager().isPresent()) {
      System.out.println(projectWithAssignedManagerWithAddress.getManager().get().getName()); // Robin
    }

    // Conditionally return the value based on some condition/predicate
    projectWithAssignedManagerWithAddress.getManager().flatMap(Employee::getAddress)
        .filter(a -> a.getCity().equals("Lucknow")).ifPresent(
            m -> System.out.println("Manager from Lucknow city is present")); // Manager from Lucknow city is present

    // If value is not present, get some other value
    Employee manager = projectWithNoAssignedManager.getManager()
        .orElse(new Employee("DEFAULTMGR", "Default manager value"));
    manager = projectWithNoAssignedManager.getManager().orElseGet(() -> {
      // some custom logic here to supply a default value
      return new Employee("DEFAULTMGR", "Default manager value");
    });

    // If value is not present, throw an exception
    projectWithNoAssignedManager.getManager()
        .orElseThrow(IllegalArgumentException::new);
}
```

One **important thing to note** here is that one should use the **Optional.get()** method very responsibly. Using the Optional.get() method directly is not the recommended usage of the Optional API. It should **always be nested with a Optional.isPresent()** check.

```
if (optionalVal.isPresent()) System.out.println(optionalVal.get());
```

If it used directly it might throw a NoSuchElementException, in case of value not being present and thus it suffers from the same problem as the null references. Even with the correct usage of Optional.get nested with Optional.isPresent, it is not recommended to use this approach because its like doing the same thing as the null checks.

Annotations

Java 8 brought support for **repeating annotations**. There are some scenarios where we would want to apply the same annotation with different values. If

Java 8 brought support for **repeating annotations**. There are some scenarios where we would want to apply the same annotation with different values. If you recall from earlier versions, this was not allowed. Before Java 8, if we were to apply a repeated annotation, we had to wrap them in container annotations. Let's see this through an example. So let's say we want to apply a Profile annotation that indicates the DEV, TEST, PROD etc. profiles that a class should be instantiated for.

Traditional code:

```
public @interface Profile {
  String value();
}

@Retention(RUNTIME)
public @interface Profiles {
  Profile[] value();
}

@Profiles({
  @Profile("DEV"),
  @Profile("PROD")
})

public class AnnotationTraditionalTryOut {
  public static void main(String[] args) {
    for (Profile profile : AnnotationTraditionalTryOut.class.getAnnotation(Profiles.class).value()) {
      System.out.println(profile.value());
    }
  }
}
```

Java 8 code:

```
@Repeatable(Profiles.class)
public @interface Profile {
  String value();
}

@Retention(RUNTIME)
public @interface Profiles {
  Profile[] value();
}

@Profile("DEV")
@Profile("PROD")
public class AnnotationTryOut {
  public static void main(String[] args) {
    Stream.of(AnnotationTryOut.class.getAnnotationsByType(Profile.class))
        .forEach(a -> System.out.println(a.value()));
  }
}
```

Java 8 also introduced **new methods in the Reflection API — getAnnotationsByType() and getDeclaredAnnotationsByType()** — to get access to repeating annotations directly. You can see thier usage in the Java 8 code example above.

Type annotations in Java 8: Earlier annotations could only be applied on declarations like on classes, methods, variable declarations, etc. **Starting with Java 8, annotations can be applied wherever a type is used**, such as types in declarations, generics, instance creation statements (new), impements clauses, throws clauses, casts, etc.

```
@Email String email;
void connect(@Open Connection conn) { ... }
List<@NonNull String> ids;
if (id instanceof @NonNull String) { ... }
obj.<@NotEmpty String>getId(...);
strObj = (@NotEmpty String) obj;
```

This feature is **primarily intended to support better type checking and validation and thus helps with improved Java program analysis by IDEs and other 3rd-party compiler plugins**. The Java platform itself does not provide any dedicated type checking framework but allows pluggable compiler plugins that could be used in conjunction with the javac compiler. Please read this article to get a good introduction to building compiler plugins. There are some 3rd-party type checking modules already available, like the Checker Framework, that can be used for stronger type checking and validation. You may explore more on type annotations here: Type Annotations in Java 8 and Java 8 type annotations. Also in context of compile-time annotation processing, you may check the Java Annotation Processing APIs. Here is a good tutorial.

Other Features

- **Date and Time APIs:** Commonly used classes are LocalDate, LocalTime, and LocalDateTime.Methods available are: from, now, of, parse, isAfter, isBefore, isEqual, format, minus, plus, and their overloads and variants. Duration and Period classes are also available.
- **Nashorn Javascript Engine:** Nashorn was the new default JavaScript engine for the JVM as of Java 8. Earlier, Rhino was used. Java 8 includes a command line interpreter called jjs, which can be used to run JavaScript files. The common use of Nashorn in Java is to allow scripting usage in Java applications. Since it is intended to be used for scripting purposes, it does not have DOM functionality.
- **PermGen memory area replaced with Metaspace**. PermGen had a default maximum. That led to those OOM (Out of Memory) errors, but with Metaspace, there is no default maximum and auto-increases to the maximum available system memory
- **Parallel array sorting**: Arrays.parallelSort(...)
- **Method Parameter Reflection**: java.lang.reflect.Executable.getParameters can be used to introspect method parameters (the classes Method and Constructor extend the class Executable and therefore inherit the method Executable.getParameters method). Formal parameter names are not emitted to class files by default. You need to compile the Java source file with the parameters flag (javac -parameters ...) to be able to use the Reflection API to introspect the formal parameter names.
- **Generalized Target-Type inference**
- **Compact Profiles:** A compact profile is a subset of the full Java SE Platform API to enable Java applications to run on resource-constrained devices. There are three profiles: compact1, compact2, and compact3. Each profile includes the APIs of the lower-numbered profiles (compact2 is a superset of compact1). The full SE API is a superset of the compact3 profile (from Oracle docs, for more details have a look at Compact Profiles)
- **IO Enhancements:** Mostly related to IO APIs returning streams.

Like This Article? Read More From DZone

Mixing Java Functions and Groovy Closures

Understanding flatMap

Infinite Streams in Java 8 and 9

Free DZone Refcard
Getting Started With Kotlin

Topics: JAVA , LAMBDA , FUNCTIONAL INTERFACE , CLOSURE , STREAMS , OPTIONAL , JAVA 8 , TUTORIAL