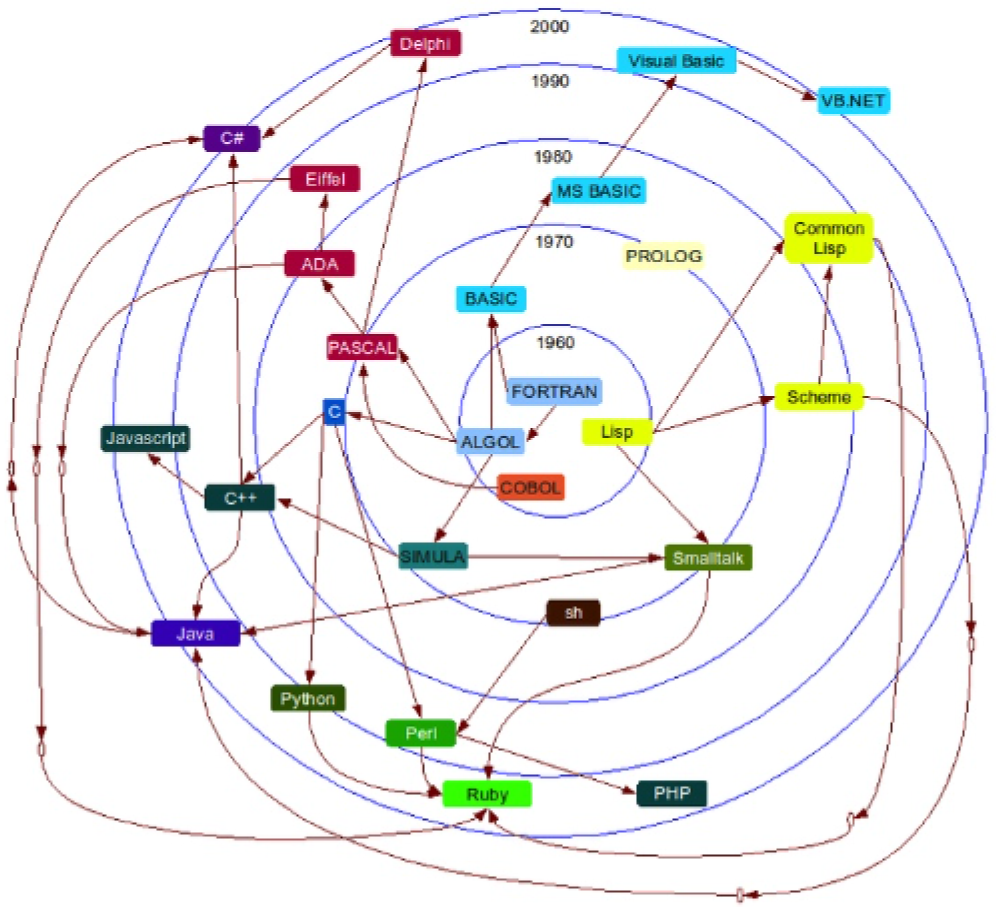
Python to Java

Like the hitchhiker's guide to the galaxy… but the Python developer's guide to Java basics. Enjoy.

Language Relationships

Python is based on C, and Java is based on C++ (which is based on C). Java looks more like C than Python does, though.



Code syntax and format

- Code blocks
    - Python
        - In Python, code blocks are defined by indentation

        - Example:

```python
if b > a:
	print("b is greater than a")
```

    - Java
        - In Java, code blocks are defined by braces ({}). Braces are opened and closed at the beginning and end respectively of blocks of code that need to be grouped together, including the bodies of:
            - Class declarations
            - Function/method declarations
            - Control structures (e.g. conditionals, loops)
        - Note: indentation doesn't have any semantic significance in Java whatsoever, however is important for human readability
        - Example:

```java
if (b > a) {
    System.out.println("b is greater than a");
}
```

- Statement ends
    - In Python, no special syntax is required at the end of statements
    - In Java, every statement must end with a semicolon
- Comments
    - Single-line comment:
        - In Python, single-line comments are written with a preceding '#'
        - In java, single-line comments are written with a preceding ('//')
    - Multi-line comments
        - In Python, multi-line comments are written with a preceding '#' at the start of each line
        - In java, a multiline comment is written with '/*...*/', with the body of the comment replacing the ellipsis
    - JavaDoc comments
        - JavaDoc is a documentation type specific to Java. It is a documentation generator that uses specially formatted comments known as JavaDoc comments in the source code to generate API documentation in HTML format.

        - JavaDoc comments are placed directly above the definition of a class, interface, method, or field that they are documenting. A javadoc comment is written with ‘/**...*/’, with the body of the comment replacing the ellipsis
        - Example:

```Java
/**
 * Description of the method being documented.
 *
 * @param param1     param description
 * @param param2     param description
 * @return what is returned by the method
 *
 */
method signature...
```

Type System

- Python
    - Python is dynamically typed.
    - In dynamically typed languages, type checking occurs at runtime. In practice this means that you don’t have to specify a variable’s type when you declare it. The type of a variable in Python is inferred, meaning it is automatically deduced during compile time from a context where it’s not explicitly declared.
    - This also means that you can change the type of a variable at any point in your code. For example, you can initially set a variable *x* to an integer value and later in your code assign the same variable to a string value.
    - Example:

```Python
x = 7        # x is an integer with value 7
x = "hello"  # x is now a string with value "hello"
```

- Java
    - Java, on the other hand, is strictly statically typed.
    - In statically typed languages, the type of each variable must be known at compile time, before the program is actually run. In practice this means that you must specify the type of each variable when you declare it.
    - If you try to assign the wrong type to a variable (e.g. assigning a string to an integer variable) the compiler will throw an error and the code won’t compile.

    - The benefit of this is the enablement of Java compile-time type analysis, which leads to greater code safety and fewer exceptions at runtime, albeit at the cost of moderately lesser flexibility.
    - Example:

```java
int x = 7;    // x is an integer with value 7
x = "hello";  // an exception is thrown
```

Classes, Variables, and Methods

- Access control / access modifiers
    - Overview
        - Access modifiers, sometimes called access specifiers, are features of object-oriented languages that allow you to set the visibility of classes, interfaces, variables, functions, and other class members. They define the scope of such member's accessibility, meaning from where it can be accessed within your code.
    - Python access control
        - In Python, access control is not enforced as strictly as in some other object-oriented languages like Java. There are no explicit keywords access modifiers used to enforce access control.
        - Python instead uses certain naming conventions to emulate the behavior of access modifiers:
            - In Python, all attributes and methods are public by default. They can be accessed from within the class, from classes derived from the class, and from instances of the class.
            - If an attribute or method name starts with a single underscore `_`, it's treated as non-public / protected. It's still accessible just like public attributes but programmers treat it as a non-public part of the API. It's considered good practice not to access such attributes from outside the class.
            - Python has a notion of name mangling to prevent accidental access to "private" attributes. If an attribute or method starts with double underscores `__`, Python will "mangle" the name so it's not easily visible or directly accessible from outside the class. However, it's still technically accessible and not truly private.
    - Java access control
        - Java uses explicit keyword access modifiers for its access control. These include:
            - Private

                - A member declared as private is only accessible within the class in which it’s defined.For example, a field or method declared as private cannot be called from outside the class to which they belong.
                - The most restrictive and least accessible level of access
            - Package private (default)
                - If no access modifier is specified then it’s considered to be package private, as is the default. Package private access has all the same visibility as private and additionally grants visibility to all classes within the same package.
            - Protected
                - A member declared as protected is accessible within the same package and also from subclasses of the class, even if they’re in different packages.
            - Public
                - A member declared as public is accessible from any class in the program.
                - The least restrictive and most accessible level of access
    - Comparison Table

Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

    - Usage
        - It is best practice to assign an access modifier to every class, interface, field, method, and constructor
        - Typically best practice to grant the minimum level of access necessary for the element to be utilized properly
            - All class fields should be private by default - you should never expose a class member as public with the exception of ‘public final static’ constants (notes below on these keywords) meant for external consumption
        - We use the Lombok third party library to cut down on verbosity/boilerplate and automatically apply access modifiers across a class

- Object declaration and initialization
    - Python
        - In Python, new objects are created simply by calling one of the class’ constructors
        - Example:

```Python
newObj = MyClass()
```

    - Java
        - In java, the ‘new’ keyword is used to create new objects (instances of a class). The ‘new’ keyword is always followed by a call to a constructor of the class, which initializes the new object.
        - In Java, therefore, each object declaration requires the type of the object to be specified and the use of the *new* keyword before making a call to the class’ constructor
        - Example:

```Java
MyClass newObj = new MyClass();
```

- Method declaration
    - Python
        - In Python, method declaration includes the following: the *def* keyword to specify that you’re declaring a method followed by the name of the method, the parameter names (notably without need for parameter types) enclosed within parentheses, and then the method body following a colon.
        - Example:

```Python
def add(a, b):
   return a + b
```

    - Java
        - In Java, method declaration includes the following: the optional specification of an access modifier, a return type, a function name, parameters, notably including parameter types as well as names, and lastly the body of the method enclosed within braces
        - Notably, method declaration in Java does not require any equivalent of the Python ‘def’ keyword.
        - Example:

```java
public int add (int a, int b) {
	return a + b;
}
```

- Method overloading & input parameter flexibility
    - Python
        - Python’s mechanism for input parameter flexibility is via the ‘*args’ and ‘**kwargs’ arguments.
        - ‘*args’ allows you to pass a variable number of positional arguments to a function. Inside the function, these arguments are received as a tuple.
        - ‘**kwargs’ allows you to pass a variable number of keyword arguments to a function. Inside the function, these arguments are received as a dictionary.
        - Example:

```python
def func_with_args(*args):
	for arg in args:
		print(arg)

func_with_args(1, 2, 3, 4)


def func_with_kwargs(**kwargs):
	for key, value in kwargs.items():
		print(f"{key} = {value}")

func_with_kwargs(name="Alice", age=25, country="Wonderland")
```

        - The inclusion of either or both of these arguments in a method signature enables you to input a variable number and type of arguments into the corresponding method. They’re particularly useful when you’re unsure about the number of arguments, or when you want to pass a variable-length list or dictionary of arguments.
        - Unlike in Java, you can’t have multiple methods in a Python class with the same name but different signatures. If you were to define multiple methods with the same name but different parameters in one class, the last one defined in the class will override the others.
    - Java

- Java does not have the same ‘*args’ and ‘**kwargs’ features as in Python. In order to achieve parameter flexibility in Java you can instead use method overloading.
- In Java, you can define multiple methods with the same name in the same class as long as they have different parameter lists (different numbers and/or types of parameters). The correct method to call will be determined by the number and type of parameters. This process is known as method overloading. Methods with the same name but different parameter lists are known as overloaded methods.
- This is a very useful way to cleanly create flexibility in the amount and type of parameters needed to be inputted into a method in Java, thus cutting out the need to input extraneous parameters in cases where they are not needed. This is one way of creating the appearance of optional parameters in Java. However, it is not quite as flexible as in other languages.
- Example:

```java
public class MathOperations {

	public static int add(int a, int b) {
		return a + b;
	}

	public static int add(int a, int b, int c) {
		return a + b + c;
	}
	public static int add(int a, int b, String c) {
		return a + b + integer.parseInt(c);
	}


	public static int add(int a, int b, int c, int d) {
		return a + b + c + d;
	}
}
```

  - Re the above example: when you invoke the add method, the Java compiler will automatically determine which version of the above add methods to use based on the method signature (name of the method + the number and type of its parameters)
- In general we want to avoid overloading a single method too much because it can clutter code and make it difficult to understand.

- Returning from methods
    - Python
        - In Python, the ‘return’ keyword is used to exit from a method and optionally return a value
        - A method will return ‘None’ by default if no return is specified
    - Java
        - Java also uses the ‘return’ keyword to exit from a method and optionally return a value, as in Python
        - Unlike Python, a method with a non-void return type must explicitly return a value in Java. Failure to do so will result in a compile time error.
        - If the method is declared to have a void return type then the ‘return’ keyword is not strictly necessary and the method will return automatically when it reaches the end of the method.
- Constructors
    - Overview
        - In both Python and Java, constructors are used for instantiating an object
    - Python
        - In Python, the `__init__` method is automatically called when an object of the class is created. It is analogous to a constructor.
        - It can take any number of parameters, but the first parameter is always ‘self’.
        - Example:

Python
```python
class MyClass:
      def __init__(self, param1="hello world", param2=7):
            self.param1 = param1
            self.param2 = param2
```

            - In the above example, if no parameters are passed during the object creation, default values “hello world” for param1 and 7 for param2 are used.
        - Unlike in Java, you can’t have multiple `__init__` methods in a Python class with different signatures. If you were to define multiple `__init__` methods with different parameters in one class, the last one defined in the class will override the others. However, you can achieve a similar effect by accepting variable-length arguments using the *args and **kwargs syntax.
        - Example:

```python
class Person:
	def __init__(self, *args, **kwargs):
		# Assuming *args contains first_name and last_name in that order
		if args:
			self.first_name = args[0] if len(args) > 0 else None
			self.last_name = args[1] if len(args) > 1 else None

		# **kwargs can be used for any other named attributes
		self.age = kwargs.get("age", None)
		self.country = kwargs.get("country", None)
		self.occupation = kwargs.get("occupation", None)
```

- Java
  - In Java, a constructor is similarly a block of code that initializes a newly created object.
  - A constructor in Java resembles an instance method but without a return type.
  - If no constructors are defined in the class, the Java compiler automatically provides a default no-args constructor:
    - In this case, the line `Class var = new Class()` would utilize the default no-args constructor Java uses a default no-args constructor.
    - This constructor takes no arguments and is often used to provide default values for the object's attributes.
    - If any other constructor is defined, the default no-args constructor must be explicitly declared if it's needed.
  - You can also define as many constructors as you like with different input parameters and parameter types, a process known as constructor overloading.
  - Example:

```java
public class TestClass {
	String param_1;
	int param_2;

	// this is the main constructor
	TestClass(String param_1, int param_2){
		this.param_1 = param_1;
		this.param_2 = param_2;
	}
```

```
        // no-args constructor
        TestClass(){
                this.param_1 = "hello world";
                this.param_2 = 7;
        }
}

// Note: see following notes on 'Current instance referencing' for an
explanation of the 'this' keyword
```

        - In the above example, if the TestClass is instantiated with no arguments, the no-arg constructor is used, which assigns "hello world" to param1 and 7 to param2. If two arguments are passed then the main constructor is used.

- Current instance / object referencing
    - Python
        - The 'self' keyword in Python is used to reference the current object, i.e. the objects/instance whose method or constructor is being called
        - Unlike 'this' in Java, 'self' is not a keyword enforced by the language but rather is a naming convention. The name can technically be changed, although this is considered non-idiomatic.
        - It's used in various scenarios:
            - To access members (variables and methods) of the current object;
            - In constructors to refer to the object being created;
            - In non-static methods to refer to the object that the method was called on.
        - The 'self' keyword is required to be the first parameter in any and all instance methods in Python, including the constructor.
        - Example:

```python
class Student:
        def __init__(self, name):
                self.name = name # 'self' refers to the current instance

        def get_name(self):
                return self.name # 'self' refers to the current instance

student = Student("Alice")
print(student.get_name()) # prints: Alice
```

- Java
    - The 'this' keyword is the same function in Java, i.e. it is a reference to the current object.
    - In Java, ‘this’ is a keyword enforced by the language
    - The 'this' keyword can be omitted when there is no ambiguity with local variables. As such, unlike in Python, it does not need to be included as the first parameter in instance methods.
    - Java's 'this' can be used to call other constructors within the same class, which doesn't have a direct equivalent in Python.
    - Example:

```java
public class Student {
	private String name;

	public Student(String name) {
		this.name = name; // 'this' refers to the current object
	}

	public String getName() {
		return this.name; // 'this' refers to the current object
	}

	// Using 'this' to call another constructor
	public Student() {
		this.name = "Unknown";
	}
}

public class Main {
	public static void main(String[] args) {
		Student student = new Student("Alice");
		System.out.println(student.getName()); // prints: Alice
	}
}
```

- Static & class members
    - Overview
        - A static/class variable in object-oriented programming is defined as a variable that belongs to the class rather than to any instance of the class. The variable itself is accessible from any instance of the class and the value is shared across all instances of the class.

        - Non-static/class variables are known as instance variables
    - Python
        - Variables
            - Python features class variables for this purpose.
            - Class variables in Python, unlike instance variables (variables that belong to an instance of the class), are defined within the class but outside of any method.
            - Python class variables are, by default, accessible both by instances of the class and by the class itself.
        - Methods
            - Python also features both class and static methods, as are distinct.
            - Class methods:
                - These are methods that take a reference to the class itself as their first parameter (commonly named ‘cls’) and operate on the class (class-level attributes) rather than the instance.
                - They're marked with the `@classmethod` decorator.
                - Python class methods are, by default, accessible both by instances of the class and by the class itself.
            - Static methods:
                - These methods don't take any special first parameter, neither a reference to the instance (‘self’) nor the class (‘cls).
                - They're marked with the `@staticmethod` decorator.
                - Don’t rely on class or instance attributes and behave like plain functions except for their association with the class.
        - Example:

```python
class MyClass:
	class_var = 0 # This is a class variable (shared across all instances)

	def __init__(self, value):
		self.instance_var = value # This is an instance variable (specific to this instance)

	def increment_class_var(self):
		MyClass.class_var += 1
		print("Class variable value:", MyClass.class_var)

	def increment_instance_var(self):
		self.instance_var += 1
```

```python
            print("Instance variable value:", self.instance_var)

        @classmethod
        def show_class_var(cls):
            print("Class variable (from class method):", cls.class_var)

        @staticmethod
        def add_numbers(a, b):
            return a + b

obj1 = MyClass(10)
obj1.increment_class_var() # Increments the class variable and prints it
obj1.increment_instance_var() # Increments the instance variable and prints it
obj1.show_class_var() # Prints the class variable using a class method
result = obj1.add_numbers(5, 3) # Calls the static method
print("Sum from static method:", result) # Prints: Sum from static method: 8
```

- Java
    - Variables
        - Java features static variables for this purpose.
        - Static fields in Java also belong to the class itself.
        - Java static variables are also, by default, accessible both by instances of the class and by the class itself.
        - Static variables in Java are preceded by the keyword *static*
    - Methods
        - Java also features static methods.
        - Static methods in Java also belong to the class itself and are accessible only via the class itself, not via an instance of the class.
        - Static methods are similarly preceded by the keyword *static*
        - Static methods can call other static methods or access static data within the same class or other classes, and they can call non-static methods and access non-static data via an instance of a class.
        - Java does not have a direct equivalent to Python's class methods. While static methods in Java are similar to static methods in Python, there's no special method type in Java that takes the class itself as a parameter
        - <u>Note: we like to use static methods at Ethic because it enforces an immutable style of programming in which we're not relying on class instance objects to do the work but rather on class components that take some input, do some work, and spit out a result</u>

            - Example:

```
Java
public class Car {
	private static int totalCars = 0; // Static field
	private String color; // Non-static field

	public Car(String color) {
		this.color = color;
		totalCars++; // Each time a Car is created, increment totalCars
	}

	public static int getTotalCars() { // Static method
		return totalCars;
	}

	public String getColor() { // Non-static method
		return color;
	}
}

public class Main {
	public static void main(String[] args) {
		Car car1 = new Car("Red");
		Car car2 = new Car("Blue");

		System.out.println(Car.getTotalCars()); // prints: 2
		System.out.println(car1.getColor()); // prints: Red
		System.out.println(car2.getColor()); // prints: Blue
	}
}
```

- Generics and type parameters
    - Overview
        - Generics and type parameters allow the definition of classes, interfaces, and methods with type parameters, enabling code that can operate on various types while maintaining type safety in statically typed languages like Java.
    - Python
        - Python's type system is dynamically typed, meaning variable types are determined at runtime.
        - This flexibility allows functions and classes to operate on various types without needing specific declarations, unlike in statically typed languages like Java.

        - Type hints
            - Python’s typing module allows for some level of type guidance through type hints. While not the same as generics, type hints are often used in Python to assist in code readability and development tools in Python.
            - Example:

```python
Python
from typing import List

def print_elements(elements: List[int]) -> None:
	for element in elements:
		print(element)
```

        - Other than type hints, which are similar, there is no direct equivalent to Java's generics, as the dynamic typing system in Python provides the needed flexibility with types by default. Type hints are completely optional and not enforced at runtime.
    - Java
        - Unlike Python, Java’s type system is strictly static, requiring the type of every variable to be known at compile time.
        - Generics in Java provide a way to create data structures and methods that can hold or operate on objects of various types, maintaining type safety.
        - Java necessitates the use of generics to achieve similar dynamic typing flexibility to Python given its strictly static type system.
        - Generic classes:
            - A generic class accepts and operates on a placeholder type, thus allowing instances of this class to be created with any type.
            - Example:

```java
Java
public class Collection<T> {
	private T t;

	public void set(T t) {
		this.t = t;
	}

	public T get() {
		return t;
```

```
        }
}
```

            - In the above example, `Collection<T>` is a generic class that can be instantiated for any type T and contains a single field of type T. For example, you could instantiate a Collection of type String (`Collection<String> = new Collection<>();`) or a Collection of type Integer (`Collection<Integer> = new Collection<>();`)
        - Generics methods:
            - Generics methods can be called with arguments of different types.
            - Example:

```java
public static <T> void print(T element) {
        System.out.println(element);
}
```

        - Each instance of a generic type is specific to one type, e.g. a `LinkedList<String>` can only contain Strings
        - Note: Generics can be quite a complicated topic in Java. However, they are fairly easy to use in everyday coding and are generally just going to be important when dealing with Collections classes, e.g. to instantiate a collection such as `LinkedList<String>`
- Program entry points: Java main method and Python top-level code
    - Overview
        - In both Java and Python, there is a specific entry point where the execution of a standalone application begins. This entry point is essential to initialize the application's execution.
    - Python
        - In Python, the code at the top level of a file is executed when the script is run.
        - Often, a block of code under the `if __name__ == "__main__":` condition serves as the entry point, allowing a script to act as either a reusable module or a standalone program.
    - Java
        - The main method in Java serves as the entry point for any standalone Java application. It is a specific function that gets run when the standalone application is executed.

        - The signature for the method is always as follows:

```java
public static void main(String[] args){
        //body
}
```

        - A brief rundown of the structure of this main method:
            - `public`: The main method must be public because the Java Runtime Environment (JRE) or Java Virtual Machine (JVM) (notes on the JRE and JVM further below) invokes it when the program starts, possibly from outside the class.
            - `static`: The main method must be static since it is invoked by the JVM before any objects are created, and therefore must not rely on an instance of the class.
            - `void`: The main method's purpose is to start the application, not to compute and return a result. Therefore it has a void return type
            - `String[] args`: This parameter, an array of String objects, represents the command-line arguments passed to the application when started from the command line. For example, if you start an application with `java MyApp arg1 arg2` then args would be an array containing "arg1" and "arg2". If no command-line arguments are provided then `args` will be empty. This feature is somewhat antiquated at this point.
- The Java 'final' keyword
    - Lastly, a Java feature that has no equivalent in Python:
    - In Java, 'final' is a keyword used to create constant variables that cannot be modified*, methods that cannot be overridden, and classes that cannot be inherited from
        - *When used with a variable of a primitive data type, the value of the variable cannot be changed. When used with a variable of a non-primitive data type, the object being referred to cannot be changed (e.g. reassigned), though members of the referred object can be changed, unless such members are themselves immutable
    - At Ethic we consider it good practice to make most variables 'final' in Java so as to enforce a functional style of programming.
    - Examples:
        - String variable: `public final String employeeId = 74; //this value of this String field cannot be modified`
        - Map variable: `public final Map<String, String> employeesMap = new HashMap<>(Map.of("1", "Jim", "2", "Mary")); //the`

```
employeesMap map cannot be reassigned, though the values
within the employeesMap can be changed
```

Basic Data Types & Libraries

- Primitives
  - Python
    - All values in Python are objects, including basic data types such as `int` and `str`.
  - Java
    - In Java, basic data types such as these are distinguished from regular Objects. They are known as primitives and have slightly different behaviors and capabilities to regular Java objects. Primitives in Java have characteristics such as:
      - 1) they cannot have methods performed on them
      - 2) they store data of only one type
      - 3) they cannot be null
    - Primitives are elements of the language itself, not instances of any class (i.e. not Objects).
    - They can be easily identified as they start with a lowercase letter
    - The Java primitives include:
      - byte (stores whole numbers from -128 to 127)
      - short (stores whole numbers from -32,768 to 32,767)
      - int (stores whole numbers from -2,147,483,648 to 2,147,483,647)
      - long (stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
        - Long values have the suffix L (e.g. 100011100001L)
      - float (stores fractional numbers up to 7 decimal places)
        - Float values have the suffix F (e.g. 3.14F)
      - double (stores fractional numbers up to 15 decimal places)
        - Double values have either no suffix or optional 'D'/'d' suffix
        - Decimals without an 'F' suffix are always doubles
      - char (stores a single character)
        - Describes a code unit in the UTF-16 encoding
        - Enclosed in single quotes rather than double quotes
    - Primitive wrapper objects
      - For every primitive type, there is a corresponding wrapper class. These are object representations of a primitive value and contain only a single field of the corresponding primitive type. For example, `Integer` is a wrapper for the `int` primitive data type and an object of type `Integer` contains a single field of type `int`.
      - They have the same name as the primitive type but with an upper case first letter (e.g. `Integer`, `Double`, `Character`).

      - The wrapper classes allow you to treat a primitive value as an object, which can be useful in situations where objects are required, such as in collections

- Java Libraries
  - Java has an extremely rich library ecosystem and learning to know and use the libraries is a big part of becoming a good Java developer.
  - JDK Standard Libraries
    - A large set of pre-built Java classes bundled within the JDK (notes on the JDK further below)
    - Includes, for example, Java.lang (most fundamental package including Primitive wrapper classes, Math, String, Thread, and System), Java.util (contains the collections framework and utility classes), Java.math (classes for performing high precision math or math on very large numbers), and Java.io (classes for system input and output and file reading and writing)
  - 3rd-party libraries
    - At Ethic we use a number of 3rd party Java libraries, including Lombok, Apache Commons, Guava, CheckerFramework, Vavr, and others.
    - 3rd party libraries enhance and simplify what we're able to do within our codebase and gaining familiarity with their capabilities over time will help you write cleaner, more efficient, and more powerful code.
    - Ethic Wiki article on 3rd party libraries: https://wiki1.ethic.tech/en/Engineering/Fundamentals/Java-External-Libraries
- Java Collections
  - Overview
    - Java Collections is a framework that provides interfaces and classes to represent and manipulate groups of objects as a single unit. These include various data structures like lists, sets, maps, queues, and others.
    - In comparison, Python also provides collections such as lists, sets, dictionaries (equivalent to Java's maps), and tuples.
    - While Java Collections offer strong type safety due to Java's static typing, Python's collections are more dynamically typed, allowing for more flexibility but potentially less type safety. However, the use of generics (detailed above) in Java collections allows you to declare the type of the Collection upon instantiation, allowing greater flexibility.
    - Both Java and Python's collections are essential tools for organizing and manipulating data.
  - Arrays and Lists
    - Array
      - Overview:
        - Fixed-size data structure. The size needs to be declared upon initialization and cannot be changed later.
        - Can only hold elements of 1 type

        - Can contain both primitive data types as well as objects of a class depending on the definition of the array
        - Can't declare an array of generic objects
        - Can be multidimensional
        - <u>Note: this is antiquated at this point except for lower-level programming. You may need to work with it, especially as certain builtin Java methods return an Array. However, we typically do not initialize Arrays at Ethic - we prefer to use ArrayList.</u>
    - Declaration:
        - Empty array of size *n*: `int[] array = new int[n];`
        - Non-empty array: `String[] strArray = {"John", "Mary"};`
    - Use case:
        - <u>We generally don't use these types of arrays</u>, except in rare cases, e.g. as an intermediary object if and when they are outputted by a JDK standard library function
    - Important methods:
        - Index: `array[i]`
        - Insert: `array[i] = elem`
        - Append/prepend: N/A
        - Size: `array.length`
        - Remove element: N/A
        - Conversion (to List): `Arrays.asList(array)`
        - Copy: `array.clone()`
    - Python closest equivalent: List
        - Note: the Python List is more similar to Java's ArrayList in behavior as they are dynamic
- ArrayList
    - Overview:
        - Variable-sized, mutable data structure.
        - No requirement to declare size upon initialization, although it's good practice to pass in an expected size if you know it
        - Generic class - must declare the type of the object to be contained within the ArrayList
        - Can only hold elements of 1 type (can be of an interface or abstract class type, in which case any subclasses of the interface/abstract class are also valid)
        - Cannot declare an ArrayList to be of a primitive type (e.g. must use Integer instead of int)
            - However, when you add a primitive value, *autoboxing* automatically wraps the primitive value into an instance of its wrapper class and when you

retrieve an element, *unboxing* automatically unwraps the object into a primitive value

- Declaration:
  - Empty array list: `List<String> al = new ArrayList<>();`
  - Non-empty array list declaration: `List<String> al = Arrays.asList("John, Mary");`
- Use case:
  - Good for when you need a resizable array-like structure with random access to elements and the ability to add items dynamically, especially if you know how many items you're expecting to hold
- Important methods:
  - Get/index: `al.get(index)`
  - Insert: `al.add(index, elem)`
  - Append: `al.add(elem)`
  - Replace: `al.set(index, elem)`
  - Add (whole collection): `al.addAll(Collection)`
  - Size: `al.size()`
  - Remove (index): `al.remove(index)`
  - Remove (element): `al.remove(elem)`
  - Remove (whole collection): `al.removeAll(Collection)`
  - Empty check: `al.isEmpty()`
  - Clear: `al.clear()`
  - Contains (single element): `al.contains(elem)`
  - Contains (whole collection): `al.containsAll(Collection)`
  - Index (of element): `al.indexOf(elem)`
  - Sublist: `al.sublist(fromIndex, toIndex)`
  - Conversion (to array): `al.toArray()`
  - Conversion (to String): `al.toString()`
  - Copy: `al.clone()`
- Python closest equivalent: List

- LinkedList
  - Overview:
    - Variable-sized, mutable data structure
    - No need to declare size upon initialization
    - Generic class - must declare the type of the object to be contained within the ArrayList
    - Can only hold elements of 1 type
    - Only supports object entries, not primitive data types
    - Only single-dimensional
  - Declaration:

        - Empty linked list: `List<String> ll = new LinkedList<>();`
        - Non-empty linked list declaration: `List<String> ll = Arrays.asList("John, Mary");`
      - Use case:
        - Perhaps the most efficient and best option for a variable size list-like data structure with dynamic addition and removal
      - Important methods:
        - *See ArrayList important methods.*
      - Python closest equivalent: deque
  - Maps (equivalent of Python dictionaries)
    - HashMap
      - Overview:
        - Key-value pair data structure
        - Variable size, mutable
        - No need to declare size upon initialization
        - Allows one null key and multiple null values
        - Does not maintain any order of elements
      - Declaration:
        - Empty HashMap: `Map<String, Integer> hm = new HashMap<>();`
      - Use case:
        - When you need a map and you're not worried about the order of its entries
      - Important methods:
        - Get: `hm.get(key)`
        - Add: `hm.put(key, value)`
        - Add if key not present: `hm.putIfAbsent(key, value)`
        - Replace value: `hm.replace(key, newValue)`
        - Remove: `hm.remove(key)`
        - Check if key exists: `hm.containsKey(key)`
        - Check if value exists: `hm.containsValue(value)`
        - Get entry set: `Set<Map.Entry<K, V>> entries = hm.entrySet()`
        - Get key set: `Set<K> keys= hm.keySet()`
        - Get value set: `Set<V> values = hm.values()`
      - Python closest equivalent: Dict
    - LinkedHashMap
      - Overview:
        - Key-value pair data structure that maintains insertion order
        - Variable-size, mutable
        - No need to declare size upon initialization
        - Allows one null key and multiple null values

        - Declaration:
            - Empty LinkedHashMap: `Map<String, Integer> lhm = new LinkedHashMap<>();`
        - Use case:
            - When you need a map that maintains insertion order
            - Good for caches
        - Important methods:
            - *See HashMap important methods.*
        - Python closest equivalent: OrderedDict
    - TreeMap
        - Overview:
            - Key-value pair data structure that maintains the natural order of its keys (sorted) OR can be ordered by a comparator provided at map creation time.
            - Variable-size, mutable
            - No need to declare size upon initialization
            - Cannot have a null key. Values can be null.
        - Declaration:
            - Empty TreeMap with natural order: `Map<String, Integer> tm = new TreeMap<>();`
            - TreeMap with custom reverse natural order comparator, for example: `Map<String, Integer> tm = new TreeMap<>(Comparator.reverseOrder());`
        - Use case:
            - When you need a map and you want to keep your entries sorted by keys.
            - Useful in scenarios when you require operations like “give me the smallest key larger than this key”.
        - Important methods:
            - *See HashMap important methods.*
        - Python closest equivalent: No direct Python built-in equivalent, but could use libraries like SortedDict to achieve similar functionality
- Sets
    - HashSet
        - Overview:
            - Stores unique elements only (no duplicate values)
            - Variable-size, mutable
            - No need to declare size upon initialization
            - Allows one null value
            - Does not maintain any order of elements
        - Declaration:
            - Empty HashSet: `Set<String> hs = new HashSet<>();`
            - Non-empty HashSet: `Set<String> hs = new HashSet<>(Arrays.asList(“John”, “Mary”));`

        - Use case:
            - When you need a collection of items where every item is unique and you don't care about their order
        - Important methods:
            - Add: `hs.add(elem)`
            - Add (whole collection): `hs.addAll(Collection)`
            - Remove: `hs.remove(elem)`
            - Contains: `hs.contains(elem)`
            - Size: `hs.size()`
            - Empty Check: `hs.isEmpty()`
            - Clear: `hs.clear()`
        - Python closest equivalent: Set
    - LinkedHashSet
        - Overview:
            - Similar to HashSet but maintains insertion order
            - Variable-size, mutable
            - No need to declare size upon initialization
            - Allows one null value
        - Declaration:
            - Empty LinkedHashSet: `Set<String> lhs = new LinkedHashSet<>();`
            - Non-empty LinkedHashSet: `Set<String> lhs = new LinkedHashSet<>(Arrays.asList("John", "Mary"));`
        - Use case:
            - When you need a collection of unique items and their insertion order matters
        - Important methods:
            - Add: `lhs.add(elem)`
            - Add all elements from a collection: `lhs.addAll(Collection)`
            - Remove: `lhs.remove(elem)`
            - Contains: `lhs.contains(elem)`
            - Size: `lhs.size()`
            - Empty Check: `lhs.isEmpty()`
            - Clear: `lhs.clear()`
        - Python closest equivalent: OrderedDict with each value set to None
- Queues & Heaps
    - PriorityQueue
        - Overview:
            - Queue data structure where elements are ordered according to a comparator or their natural ordering
            - Variable-size, mutable

            - No need to declare size upon initialization
            - Does not permit null values
            - Not syncrhonized
        - Declaration:
            - Empty PriorityQueue: `PriorityQueue<Integer> pq = new PriorityQueue<>();`
            - Non-empty PriorityQueue: `PriorityQueue<Integer> pq = new PriorityQueue<>(Arrays.asList(1, 2, 3));`
        - Use case:
            - When you need to access elements according to their natural priority (e.g. numeric items are accessed in ascending order)
        - Important methods:
            - Add: `pq.add(elem)`
            - Remove: `pq.remove(elem)`
            - Contains: `pq.contains(elem)`
            - Get (don't remove) head: `pq.peek()`
            - Get (and remove) head: `pq.poll()`
            - Clear: `pq.clear()`
            - Empty check: `pq.isEmpty()`
        - Python closest equivalent: heapq module
    - ArrayDeque
        - Overview:
            - A resizable array, typically used as a stack (last in first out) or queue (first in first out)
            - Variable-size, mutable
            - No need to declare size upon initialization
            - Does not permit null values
        - Declaration:
            - Empty ArrayDeque: `ArrayDeque<Integer> dq = new ArrayDeque<>();`
            - Non-empty ArrayDeque: `ArrayDeque<Integer> dq = new ArrayDeque<>(Arrays.asList(1, 2, 3));`
        - Use case:
            - Efficient for adding or removing elements at both ends, such as in stacks and queues
        - Important methods:
            - Append: `dq.add(elem)`
            - Prepend: `dq.push(elem)`
            - Remove: `dq.remove(elem)`
            - Contains: `dq.contains(elem)`
            - Get (don't remove) first: `dq.getFirst()`
            - Get (and remove) first: `dq.removeFirst()`

        - Get (don't remove) last: `dq.getLast()`
        - Get (and remove) last: `dq.removeLast()`
        - Size: `dq.size()`
        - Clear: `dq.clear()`
        - Empty check: `dq.isEmpty()`
      - Python closest equivalent: deque
- Other
  - *Void*: used as a return type for functions or methods that do not return anything
  - Object: the root of the Java class hierarchy / the ultimate superclass of every Java class
    - Every class in Java is directly or indirectly derived from the Object class
    - Can be used as a type to specify that a collection can contain any type of object, for example `ArrayList<Object> s = new ArrayList<>(Arrays.asList("John", 1));`

Basic Operations

- Basic Common Methods
  - Printing: `System.out.println(elementToPrint);`
  - Equality checks (see below section on Equality Checking):
    - Two objects: `Objects.equals(object1, object2);`
    - Two Arrays: `Arrays.equals(object1, object2);`
  - Null check: `Objects.isNull(object);`
  - Conversion to String: `object.toString();`
  - Parsing and converting strings:
    - To Integer: `Integer.parseInt("123");`
    - To Double: `Double.parseDouble("123.45");`
  - Collection sorting: `Collections.sort(Collection);`
  - Exiting the program: `System.exit(exitCode);`
- Conditionals
  - Java conditionals work similarly to Python's with just slightly varying syntax
  - If/else:
    - Python:

```python
Python
if condition:
        #...
elif conditoin:
        #...
else:
        #...
```

        - Java:

```java
if (condition) {
        //...
} else if (condition) {
        //...
} else {
        //...
}
```

            - In java, parentheses are required around the condition and braces around the body of the if or else
            - Java uses the ‘else if’ keyword as opposed to the Python ‘elif’
    - Ternary operator
        - Python: `result = value_if_true if condition else value_if_false`
        - Java:
            - In java, parentheses are required around the condition, the ‘if’ and ‘else’ are replaced by ‘?’ and ‘:’, and the formatting is slightly different
            - Example: `int result = (condition) ? value_if_true : value_if_false`
            - Best practice not to nest ternary operators
- Looping
    - For loop:
        - Python:
            - Python has several styles of for-loop, including the standard for loop, the enumerate loop, and the range loop.
            - Standard for-loop:
                - This is used to iterate over elements in an iterable like a list, tuple, or string.

```python
for value in iterable:
        # loop steps through each element in the collection
```

            - Enumerate loop:
                - This is used if you need to access the index as well as the value.

```
Python
for index, value in enumerate(iterable):
	# loop steps through each element in the collection and increments index
```

        - Range loop:
            - This is used to loop over a sequence of numbers, whether to use as indexes of an iterable or separately.

```
Python
for i in range(10):
	# value of i increments with each loop
```

    - Java:
        - Java has three primary styles of for-loop (that we use here at Ethic): a traditional, C-style for-loop, the enhanced for-each loop, and the Stream API for-loop.
        - Traditional for-loop:
            - The traditional, C-style for-loop has three parts: initialization, condition, and increment/decrement.
            - It requires braces around the body and parentheses around the condition.
            - This is similar in functionality to Python's Range loop.

```
Python
for (int i = 0; i < 10; i++) {
	//value of i increments with each loop
}
```

        - Enhanced for-each loop:
            - This loop is similar in functionality to Python's Standard for-loop.
            - It is useful for iterating through elements in a Collection rather than through numbers in a range.

```
Python
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

for (Integer number : numbers) {
	// loop steps through each element in the collection
}
```

        - Stream API for-loop:
            - The Stream API for-loop can be used to iterate over collections in a more functional style.
            - Functionally this loop is also most similar to Python's Standard for-loop.
            - The syntax of this loop involves a lambda expression within the forEach body, which is covered in a later section.
            - Notes on Java's Stream API appear later in this document

```
Python
array.forEach(value -> {
       // loop steps through each element in the collection
});
```

        - At Ethic we generally prefer the use of the Stream API for-loop
- While loop:
    - Python
        - In Python, while loops are often used to repeatedly execute a block of code as long as a condition is `True`.
        - Basic while loop:

```
Python
while condition:
       #code to execute while the condition is true
```

        - Break and continue:
            - You can control the loop's flow using the `break` and `continue` keywords. `break` allows you to exit the loop whilst `continue` allows you to skip to the next iteration.

```
Python
while condition:
       if some_other_condition:
              break #while loop ends

       #code to execute

       if yet_another_condition:
              continue #while loop skips to the next iteration
```

        - While-else loop:

        - In a Python while-else loop, the ‘else’ block is executed if the loop is terminated because the condition becomes false and is skipped if the loop is exited via a `break` statement.

```python
while condition:
       # code to execute while the condition is true
else:
       # code to execute when condition is False
```

    - Java
        - In Java, while loops are very similar in structure and function. It differs only really in syntax.
        - Note: we tend not to use these very often at Ethic, and only really for lower-level code
        - Basic while loop:
            - This is very similar to Python’s Basic while loop.

```java
while (condition) {
       //code to execute while the condition is true
}
```

        - Do-while loop:
            - This is unique to Java (and some other C-like languages) where the loop body is guaranteed to execute at least once, even if the condition is initially false.

```java
do {
       // code to execute
} while (condition);
```

        - Java’s `break` and `continue` keywords function in the same way as in Python
        - Java does not have an equivalent to the Python while-else loop. The code immediately following a Java while loop is executed as soon as the condition is false, however will also be executed if the loop is exited via a `break` statement, unlike the code in the ‘else’ block of a Python while-else loop.
- Switch/match & case blocks

- Python
  - Python's match/case block takes an expression (the `match` expression) and compares it to several patterns using `case` clauses. The first `case` pattern that matches the `match` expression will have its corresponding block of code executed.
  - Once a pattern is matched, the corresponding code block is executed and then the match statement is executed.
  - Example:

```python
match expression:
	case pattern1:
		# code block for pattern1
	case pattern2:
		# code block for pattern2
```

  - Patterns:
    - Patterns can be a wide variety of types and structures
      - Literal patterns: matching exact values.
      - Variable patterns: matching any value and binding it to a variable.
      - Sequence patterns: matching lists or tuples.
      - Mapping patterns: matching dictionaries.
      - Class patterns: matching instances of classes.
      - Wildcard pattern: matching anything.
    - Combining patterns: you can use the '|' operator to combine patterns.
    - Guards: guards are additional conditions that can be added to a pattern using the `if` keyword.
    - Examples:

```python
match expression:
	case 42: #literal pattern
	case y: #variable pattern
	case [0, 0]: #sequence pattern
	case {"name": "Alice", "age": age}: #mapping pattern
	case Circle(radius=r): #class pattern
	case 1 | 0: #combining literal patterns
	case y if y > 10: #guards on a variable pattern
	case _: #wildcard pattern
```

- Java
    - In Java, the switch/case block provides a similar multi-way branch, allowing you to check for equality between a variable or expression (`switch` statement) and a series of values (`case`s) and execute different possible blocks of code depending on which `case`s match the `switch` expression.
    - The `switch` expression must evaluate to a char/Character, byte/Byte, short/Short, int/Integer, String, or an enum value.
    - `default` block:
        - The code in the 'default' block is executed if the value of the variable/expression being checked does not match any of the cases.
        - Serves the same functionality as the Python wildcard pattern.
        - <u>Note: it is best practice to ALWAYS add a default block, often just to throw an exception if none of the cases match</u>
    - Fall-through
        - In the Java switch/case block, unlike in the Python match/case block, once a pattern is matched, the execution of the switch/case block does not exit.
        - In the Java switch/case block, unlike in the Python match/case block, once a case is matched, the execution of the switch/case block does not exit automatically.
        - Rather, the control will 'fall through' from one case statement to the next, executing the code under each case until a `break` statement is encountered or the end of the block is reached. This process is called "fall-through".
    - `break` statements:
        - The break statement in a switch/case block, similar to the effect of the same keyword in a while loop, exits the block.
        - This is useful and necessary because of fall-through: you may want to exit execution of the block after a `case` is matched against the switch expression rather than continuing on to further `case` statements.
    - Example:

```java
switch (variable) {
        case value1:
                //code to execute if the switch expression matches value1
                break; //execution of the block exits
        case value3:
```

```
        //code to execute if the switch expression matches value3
    case value4:
        //code to execute if the switch expression matches value4
        //because of the lack of break statement in the previous case, if
    the switch expression matches both value3 and value4 then both the 'case
    value3' and 'case value4' code blocks will be executed)
    ...
    default:
        //code to execute only if no cases are matched against the switch
    expression
}
```

- Casting variables
  - Python
    - In Python, casting between types is often done using constructor functions like `int()`, `float()`, and `str()`.
    - Python is dynamically typed so variables don't have a fixed type, but you may need to convert between types in different situations.
    - Examples:

```
Python
#Casting a float to an integer
x = 1.2
y = int(x) # y is 1

#Casting an integer to a string
x = 42
y = str(x) # y is '42'

#Casting a string to a float
x = "3.14"
y = float(x) # y is 3.14
```

  - Java
    - In Java, casting is more rigid due to the statically typed nature of the language.
    - There are two types of casting:
      - Primitive type casting
        - Implicit (widening) casting:

            - This involves converting from a smaller primitive type to a larger type that can represent more values (e.g. `int` to `float`).
            - This is done automatically by Java.

```java
int myInt = 9;
double myDouble = myInt; // Automatically casts the int to a double
```

        - Explicit (narrowing) casting:
            - This involves converting from a larger primitive type to a smaller type, during which conversion information may be lost.
            - This must be done manually by placing the type being cast to in parentheses before the name of the new variable.

```java
double myDouble = 3.14;
int myInt = (int) myDouble; // myInt is 3
```

    - Reference type casting
        - Casting between Objects (non-primitives) of different types. This must similarly be done manually by placing the type being cast to in parentheses before the name of the new variable.
        - The cast must be between classes that have an inheritance relationship.
        - If you try to cast an object to a subclass that it's not an instance of, it can lead to a ClassCastException at runtime.

```java
Object myObject = new String("Hello");
String myString = (String) myObject; // Valid cast as String is an Object
```

- Autoboxing/unboxing
    - Java allows for automatic conversion between primitive types and their corresponding wrapper classes (e.g. from `int` to `Integer`). This is called autoboxing/unboxing.

- Lambdas
    - Overview
        - A lambda expression is a convenient syntax available in many programming languages for writing short functions.
        - Lambda expressions do for functions what object-oriented programing does for objects: it makes a function something you can assign to a variable.
    - Python
        - In Python, lambda functions are defined using the `lambda` keyword followed by a list of arguments, a colon, and an expression to execute with the arguments.
        - Example:

```python
add_one = lambda x: x + 1
result = add_one(1) # result is now 2
```

        - Lambdas in Python are often used with higher-order functions like map, filter, and reduce, for example:

```python
numbers = [1, 2, 3, 4]
squares = map(lambda x: x**2, numbers) # squares is [1, 4, 9, 16]
```

    - Java
        - Java has lambda functions similar to those in Python.
        - The syntax of lambda functions in Java is slightly different, involving no need for a keyword. They simply include the arguments listed within parentheses, a '->' symbol, and the expression to be executed.
            - If a lambda expression requires only a single line of body code then it follows this syntax.
            - If the lambda expression requires multiple lines of body code then the body must be contained within brackets ('{}').
        - Java lambdas can be executed using builtin methods on the type of the lambda (see below for Java lambda types). They can also be passed in as function arguments.
        - Example:

```java
Function<Integer, Integer> addOne = (x) -> x + 1;
```

```java
int result1 = addOne.apply(1); // result1 is 2

//Multi-line lambda expression
Function<Integer, Integer> doMultipleOperations = (x) -> {
       y = x + 1;
       return x * y;
}
int result2 = doMultipleOperations.apply(2); // result is 6
```

- Java lambda types:
    - When instantiated as objects, Java lambdas can take one of several types. These types are each Functional Interfaces, a special type of interface with a single abstract method. Though the concept of Functional Interfaces is not quite necessary for the purposes of this documentation, know that the types of lambdas correspond to Functional Interfaces. These Functional Interfaces include (but are not limited to):
        - Function<T, R> - a function that accepts one argument and produces a result
        - BiFunction<T, U, R> - a function that accepts two arguments and produces a result
        - BinaryOperator<T> - an operation upon two operands of the same type, producing a result of the same type as the operands
        - Predicate<T> - an operation that takes in one argument and returns a Boolean
        - BiPredicate<T, U> - an operation that takes in two arguments and returns a Boolean
        - Consumer<T> - an operation that accepts a single input argument and returns no result
            - Useful when no value return is needed as they are expected to operate via side-effects
        - BiConsumer<T, U> - an operation that accepts two input arguments and returns no result
            - Similarly useful when no value return is needed as they are expected to operate via side-effects
    - Any programmer can define their own Lambda types by creating a custom Functional Interface, so we're not at all limited to what's available out of the box
- Type inference
    - Java's compiler often infers the types of the parameters in the lambda, so they can typically be omitted. For example:

```java
Function<String, String> comp = (s) -> s + "_modified";
```

- Incrementing and Decrementing
    - Python
        - In Python, there are no in-built increment or decrement operators.
        - Instead, variables are incremented or decremented using the addition (`+=`) or subtraction (`-=`) assignment operators.
        - Example:

```python
# Increment
x = 1
x += 1 # x is now 2

# Decrement
y = 2
y -= 1 # y is now 1
```

    - Java
        - Java allows for the assignment `+=` and `-=` operator syntax for incrementing and decrementing variables as in Python.
        - It also includes specific in-built increment (`++`) and decrement (`--`) operators, which can be placed preceding or following the variable to be incremented or decremented.
        - Example:

```java
// Increment
int x = 1;
x += 1; // x is now 2
x++; // x is now 3
++x; // x is now 4

// Decrement
int y = 4;
y -= 1; // y is now 3
y--; // y is now 2
--y; // y is now 1
```

- Equality Checking
    - Python

        - In Python, the `==` operator checks for value equality. If two objects have the same content, then `obj1 == obj2` will return `True`.
        - The `is` keyword, on the other hand, checks for reference equality. If two references (variables) point to the same object in memory then it will return `True`.
        - Example:

```Python
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1
print(list1 == list2) # True, because the contents of the lists are the same
print(list1 is list2) # False, because they are two different objects in memory
print (list3 is list1) # True, because they are the same object in memory
```

    - Java
        - Conversely, in Java, the `==` operator checks for reference equality. If two references (variables) point to the same object in memory then it will return `true`. If two strings are identical due to string interning (see notes below), they might be the same in memory, but that's not a guarantee.
        - The .equals() method, on the other hand, checks for value equality. It will return true if two objects are equivalent in content, even if they are different objects in memory. Every class in Java should (though it is not required) have a .equals() method implemented for this purpose.
        - String interning:
            - Java maintains a special pool of strings (the string pool), which is part of the heap memory. When you create a string literal (i.e., a string defined between double quotes), Java checks the string pool to see if an identical string already exists. If it does, Java will simply return a reference to the cached string object. Otherwise, it adds the new string to the pool.
            - This behavior is why sometimes you'll see two string literals with the same content reference the same object in memory, making `str1 == str2` return `true`.
            - This is an optimization that helps save memory since strings are immutable in Java.
            - This behavior is specific to string literals and doesn't apply whens trings are created using the `new` keyword. If you create a new string using the `new` keyword then you'll bypass the string pool and automatically create a new object in memory.
        - Example:

```Java
String str1 = "hello"
String str2 = new String("hello");
String str3 = str1
String str4 = "hello"

System.out.println(str1 == str2); // prints false
System.out.println(str1.equals(str2)); // prints true
System.out.println(str1 == str3 && str1.equals(str3)); // prints true
System.out.println(str1 == str4 && str1.equals(str4)); // prints true
```

Inheritance

- Overview
    - Inheritance is an OOP mechanism that allows one class to inherit the attributes (fields) and behaviors (methods) of another class. This forms a parent-child relationship where the class that is inherited is known as the super/parentclass and the class that does the inheriting is known as the sub/childclass.
    - In general at Ethic we prefer composition, a design concept that models a "has-a" relationship. It enables you to reuse code by containing instances of other classes within your class. This often leads to a more flexible structure since it's easier to change class relationships without affecting other parts of your code.
    - However, inheritance can still be very useful in certain contexts, especially when the parent-child relationship is natural and clear.
- Python inheritance
    - The *abc* (Abstract Base Classes) module of Python enables the definition of interfaces and abstract classes which other related classes can inherit
        - Interfaces and abstract classes are created by defining a class that inherits ABC (see following example)
    - Abstract methods can be created within abstract classes and interfaces by using the @abstractmethod decorator
    - Abstract classes
        - Can be created through the use of abstract base classes containing one or more abstract methods (a method declared without an implementation) and 0 or more non-abstract methods
        - Cannot be instantiated as objects
        - Purpose is to be inherited by other classes
        - Other classes can inherit any number of abstract classes
    - Interfaces
        - Can be mimicked through the use of abstract base classes with entirely abstract methods
    - Inheriting abstract classes and interfaces

        - Subclasses can inherit from abstract classes or implement interfaces in Python by including the abstract base class or interface name in the class definition. This is done by placing the name of the abstract base class or interface in parentheses after the name of the subclass.
    - Example:

```python
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):
	@abstractmethod
	def do_something(self):
		pass

class Subclass(AbstractClassExample):
	def do_something(self):
		print("The subclass is doing something")
```

- Java inheritance
    - Abstract methods
        - Abstract methods are methods that possess a method signature but no method body / implementation (see below example of an abstract class for an example)
        - Abstract methods can be created using the *abstract* keyword together with a method signature and no method implementation
    - Abstract classes
        - Can be declared using the *abstract* keyword at the beginning of the class definition
        - Can include both abstract methods and non-abstract methods
        - Cannot be instantiated as objects
        - Purpose is to be inherited by other classes
        - A class can only inherit 1 abstract class
        - Declared fields can be non-static and non-final, unlike interfaces
            - This allows for different subclasses implementing the abstract class to have these fields with unique values
        - Example:

```java
abstract class Animal {
	abstract void speak(); //abstract method
}
```

- Interfaces
    - In java, an Interface is another special type of class intended for inheritance purposes.
    - Like an abstract class, they cannot be instantiated
    - Can have only abstract, default, and static methods.
    - All declared fields are static and final (have to instantiate them with a value and the same value applies to every object implementing that interface)
    - Purpose is to be inherited by other classes
    - A class can inherit multiple interfaces
    - Example:

```java
interface Movable {
	void move();
}

interface Machine {
	void start();
}
```

- Inheriting abstract classes and interfaces
    - Abstract classes in Java can be inherited by including the ‘extends’ keyword followed by the name of the abstract class being inherited in the class definition. A class can only inherit one abstract class.
    - Interfaces in Java can be inherited by including the ‘implements’ keyword followed by the name or names separated by commas of the interface(s) being inherited in the class definition. A class can inherit multiple interfaces.
    - Examples:

```java
//a java class that inherits a single interface
class Vehicle implements Movable {
	public void move() {
		System.out.println("skrt");
	}
}

//a java class that inherits multiple interfaces
```

```
class Vehicle implements Movable, Machine {
	public void move() {
		System.out.println("skrt");
	}

	public void start() {
		System.out.println("vroom");
	}
}

//a java class that inherits an abstract class
class Dog extends Animal {
	void makeSound() {
		System.out.println("Bark");
	}
}
```

Exceptions

- Throwing errors and exceptions
    - Python
        - In Python, exceptions are thrown using the raise keyword.
        - You can raise a specific exception by creating an instance of the exception class, optionally providing an error message.
        - Example:

```python
if x < 0:
	raise ValueError("x cannot be negative")
```

-
    - Java
        - In Java, you can manually throw an exception using the `throw` keyword.
        - You can similarly throw a specific exception by creating an instance of the exception class, optionally providing an error message.
        - Example:

```java
if (x < 0) {
        throw new IllegalArgumentException("x cannot be negative");
}
```

- Exception handling
    - Python
        - In Python, exception handling is typically done using a try/except block in order to catch exceptions that might be thrown as your program executes.
        - The ‘try’ block includes the code to execute that might cause an exception.
        - The ‘except’ block includes the code to execute if an exception occurs. You can catch specific exception types or all exceptions.
        - An ‘else’ block, as is optional, includes code to execute if no exception occurs. This code is always executed if no exception occurs.
        - A ‘finally’ block, as is also optional, includes code that is ALWAYS executed after the code in the previous blocks, regardless of whether or not an exception occurs. This is typically used for cleanup actions.
        - Example:

```python
try:
        # Code that might throw an exception
except ExceptionType as e:
        # Code to handle the exception, e.g.:
        print(e)
else:
        # Code to run if no exception occurs
finally:
        # Code that will always run, like closing a file
```

        - <u>Note: exception handling in Python is optional and the programmer is not required to handle or declare them. There’s no compile-time checking to ensure that exceptions are handled or declared.</u>
    - Java
        - In Java, the try/catch block serves the same function and has very similar syntax.
        - The ‘try’ block similarly includes the code to execute that might cause an exception.
        - The ‘catch’ block, like the Python ‘except’ block, includes the code to execute if an exception occurs. You can catch specific exception types or all exceptions.

        - The ‘finally’ block, as is again optional, also includes code that is ALWAYS executed after the code in the previous blocks, regardless of whether or not an exception occurs. This is typically used for cleanup actions.
        - The Java try/catch block does not include the ‘else’ block of the Python try/except block.
        - Example:

```java
try {
      // Code that might throw an exception
} catch (SpecificException e) {
      // Handle a specific exception, e.g. through:
      e.printStackTrace();
} catch (Exception e) {
      // Handle all other exceptions
      e.printStackTrace();
} finally {
      // Code that will always run, like closing a file
}
```

        - <u>Note: exception handling in Java is not always optional, and certain kinds of exceptions (checked exceptions - see below for notes) are required to be handled or declared by the programmer. There is compile-time checking to ensure that these kinds of exceptions are handled or declared.</u>
- Java checked vs unchecked exceptions
    - Java has two types of exceptions: check and unchecked exceptions.
    - Checked exceptions:
        - Represent invalid conditions in areas outside the immediate control of the program, e.g. IOException and FileNotFoundException.
        - Java forces you to wrap these in a try/catch or declare them in a method signature using the `throws` keyword, thus allowing the method’s caller to handle them (notes on this in the following section).
    - Unchecked exceptions:
        - Typically errors in your code or bugs (e.g. null pointer exception or array index out of bounds), e.g. RuntimeException and IllegalStateException.
            - At Ethic we mostly lean on IllegalStateException
        - These are exceptions you don’t have to explicitly surround with a try/catch in your code or declare in the method signature.
- Java method exception declaration
    - In some cases with checked exceptions in Java, you want to delegate responsibility for handling the exception to the calling method. In this case you declare your method as throwing that type of exception

    - Example:

```java
public void readFromFile(String filename) throws IOException {
	BufferedReader br = new BufferedReader(new FileReader(filename));

	String line;
	while ((line = br.readLine()) != null) {
		System.out.println(line);
	}

	br.close();
}
```

Programming Paradigm Differences

- Object-oriented vs procedural programming
    - Python
        - Python supports both procedural programming and OOP, meaning you can create classes and objects but functions can also be created and invoked independently of classes and objects.
        - Python supports OOP but it is not as strictly enforced as it is in Java.
    - Java
        - Java also supports both procedural programming and OOP, but in a different way.
        - OOP is much more strictly enforced in Java, as every piece of code must belong to a class and every piece of data must be an object (an instance of a class).
        - Procedural programming is still achievable within methods and static contexts. However, all code in Java needs to be within a class structure, so pure procedural programming is not present.
- Data structures & their usage
    - Python
        - Built-in data structures are relatively simple and versatile.
        - For example, the Python dict can be used in a wide variety of use cases and it is not quite as specialized as the variety of Java Map implementations.
        - Specific characteristics of the data structure (e.g. order of insertion) may not be explicitly managed through the data structure itself but instead through how you manipulate it in your code.

        - Operations on data structures, such as checking for a value's existence or altering an item, often require separate instructions and logic and are not necessarily built into the data structure's inherent methods.
        - This structure allows for flexibility in their usage.
        - Fewer specialized built-in methods can result in more verbose code.
    - Java
        - Java enables a more specialized and strategic use of data structures. For example, there is a wider variety of Map implementations (HashMap, LinkedHashMap, TreeMap), each with unique characteristics and best use cases.
        - The use of each type in Java needs to be more strategic and thought out based on the specific requirements of your program.
        - Data structures often include methods to perform more complex and specialized operations directly. For example, the hashMap.putIfAbsent() method allows you to check for the existence of a key and insert a key-value pair if the key does not exist all in one operation without the need for a surrounding conditional (as would be required in Python).
        - This can lead to more efficient code.
        - This may also require a deeper understanding of the structures' properties and the trade-offs between them.

Performance

- Python
    - Python is an interpreted language. This means that the code is executed line-by-line directly by the interpreter at runtime. There's no separate compilation step - the source code is directly executed.
    - This results in generally slower runtimes than Java.
    - This does, however, allow for more flexibility and often easier debugging, as changes can be made to the code and immediately run without the need to recompile. It also makes the development process somewhat simpler and platform-independent as there's no need to compile the code for different platforms.
- Java
    - Java is a compiled language, meaning that the source code is first compiled to bytecode by a compiler before the Java Virtual Machine (JVM) then interprets or further compiles this bytecode and executes it at runtime. This compilation happens before runtime and is known as Just-in-Time compilation.
    - This results in generally faster execution times in Java compared to Python.
    - This design also allows Java to maintain some platform independence (“Write Once, Run Anywhere”) while benefiting from the speed of compiled execution.
    - This does, however, result in generally slower build times. Additionally, changes to the code require recompilation. Also, compiled code may need to be targeted to specific platforms or architectures, which can reduce portability.

JDK

- The JDK is a software development kit (SDK) used for developing Java applications and applets
- Provides the tools you need to write and run Java programs
- Includes/encapsulates the Java Runtime Environment (JRE), Java Virtual Machine (JVM), standard libraries, an interpreter/loader (java), a compiler (javac), an archiver (jar), debugger, and the Java Virtual Machine (JVM) amongst other tools
    - JRE:
        - Provides the libraries, the JVM, and other components necessary to run compiled Java applications
        - Does not include any of the development tools that come with the JDK, such as compilers or debuggers
        - If you're just running Java applications you'd only need the JRE but if you're developing those applications you need the JDK
    - JVM:
        - The JVM is a virtual machine that interprets Java bytecode and executes it as a program
        - Equivalent to the Python interpreter
    - *java*: the Java application launcher which executes java classes
        - Similar to running a Python script in the terminal with the *python* command
    - *javac*: the Java compiler which converts source code into Java Bytecode that can be executed by the JVM
        - Similar to the Python interpreter converting Python scripts to bytecode before executing them
- Python doesn't have separate entities such as JDK and JRE as Python's interpreter can execute the Python scripts directly, so when you install Python you're effectively installing the equivalent of the Java JDK and JRE together

Additional Differences

- Python Slice notation
    - In Python, you can use slice notation to easily and quickly access subsections of sequences (like lists, strings, and tuples) by specifying a range of indices.
        - E.g. `my_list[2:5]` will get elements from the 3rd to the 5th in `my_list`
    - This syntax is not available in Java. Instead you would need to use a loop or builtin methods (e.g. `list.sublist(startIndex, endIndex)`) to achieve the same result.
- Python List comprehension
    - List comprehensions are a powerful Python feature that allow you to create and manipulate lists on-the-fly in a concise and readable way.

    - Java does not have an equivalent feature. Streams (via Java's Stream API) bring some of the list comprehension flavor in terms of the modularity of combining/chaining operations, but not in quite as compact and terse of a style. Similar operations could also be done in explicit loops.
- Java Stream API
    - Java's Stream API, introduced in Java 8, is a powerful feature that allows for functional-style operations on sequences of elements, such as collections.
    - The Stream API brings a new abstraction of data manipulation using a functional approach, and it can greatly simplify many data processing tasks.
    - It enables more concise and readable bulk data operations.
    - It includes a rich set of operations for filtering, mapping, reducing, collecting, and more, and it allows for easy parallelization, leading to potentially more efficient code execution.
    - It is one of the key features in modern Java programming that aligns with functional programming paradigms. We use it extensively here at Ethic.
    - More comprehensive notes are contained in a separate document.
- Python Tuple data type
    - Python has a built-in tuple data type. Python's tuple is an immutable, ordered collection of elements that can contain elements of different types.
        - Example: `my_tuple = (1, "two", 3.0)`
    - Java does not have built-in support for tuples. At Ethic we use third-party libraries that provide support for tuples, particularly the Vavr library. We have also written extensive helper functions around the Vavr tuples.
        - Example declaration: `final Tuple2<Integer, Integer> my_tuple = Tuple.of(1, 2);`