# Java

## Why Java?

- write once, run anywhere (WORA)
- scalable, maintainable, readable, modular/reusable
- Object-oriented (rather than functional, procedural, etc.)
- High-Level (readable, closer to English than machine code)
- Backed by Oracle
- libraries and frameworks
- manages memory for you (garbage collection)
- staticly/strongly typed
- pass by value (NOT pass by reference)

## JDK/JRE/JVM

- JDK: Java Development Kit
    - necessary for developing in Java
    - includes compiler
- JRE: Java Runtime Environment
    - only *runs* Java applications - can't develop with just the JRE
    - includes the core classes and libraries of Java
- JVM: Java Virtual Machine
    - allows Java to be executed
        - different OS have different implementations (this is what allows WORA)
    - includes the class loader, garbage collection

Source Code (.java) -> Compiled (javac) -> Java Byte Code (.class) -> JVM

## Object Class

- this is parent class of ALL objects in Java
- methods such as hashcode, equals, toString that are typically overriden

Implicit: `public class Cat`
Explicit: `public class Cat extends Object`

## Variable Scopes

Reminder: these are NOT access modifiers

- the visibility (or lifetime) of a variable
- Class/Static Scope

    - variables live throughout the applications lifetime
    - begin when the classes are loaded (before the main method executes)
    - end when the application ends
- Instance/Object Scope
    - variables live as long as a particular instance of a class (object)
    - they begin when the object is instantiated
    - they end when the object is "garbage collected" (after the object's memory llocation is no longer referenced)
- Method
    - variables live as long as the method is executing
    - this might be the parameters or variables that are declared during the method's execution
- Block Scope (loop/local)
    - variables live as long as the current iteration of a loop, or as long as the execution of a block
    - this might be a variable declared in a loop or a conditional statement
    - this might be a variable declared in a static or instance block
- Static / Instance Block
    - static block runs when the class is loaded (before the main method, before any objects are instantiated)
        - it's associated with the class itself, so it only runs once during the application
        - `static {}`
    - instance block runs when an object of the class is instantiated (before the constructor)
        - they are associated with that particular object/instance
        - they are used for anything that needs to occur regardless of which constructor is called.

## Strings, String API

- A String is an Object in Java `"This is a String!"`
- Strings are immutable character arrays
    - this means you can't the value that is at the memory location
    - if you change the string, you are creating a new object at a different location in memory
- If we want to have *mutable* strings, we can use an alternative:
    - StringBuilder: mutable, fast, not thread-safe
    - StringBuffer: mutable, slow, thread-safe

```
String newString = "Hello"; // String Literal -> String Pool
System.out.println(newString); // "Hello"
```

```
newString = "Hi"; // Create "Hi" in the string pool and now newString is pointing to "Hi"
System.out.println(newString);
```

# OOP

**abstraction**

- when an entity provides an interface to be used without providing implementation details
- interfaces:
    - all fields are implicitly public, static, final
    - all methods are implicitly abstract
    - you can have implemented methods by using the default modifier (default implementation)
    - cannot be instantiated
    - can extend other interfaces
    - can be implemented by abstract classes and concrete (regular) classes
    - a class can implement multiple interfaces
- abstract classes:
    - fields are normal, methods are normal
    - can have abstract methods (using the abstract modifier)
    - cannot be instantiated
    - can be extended by other abstract classes and concrete (regular) classes

**inheritance**

- when an entity receives state and/or behavior from a parent entity
- includes implements and extends relationships - anything that is a sub/super class relationship

**encapsulation**

- when an entity restricts access to parts of itself to only specific entities
- a fully encapsulated class or Java Bean has all private fields and uses getter and setter methods to access them, and only has a no-args constructor (no overloaded) **polymorphism**
- when an entity acts as another entity seamlessly
- ex. a kitten is a kitten, but it is ALSO a cat
    - might handle some things differently than a cat but can act as a cat

# More

**Collections**

- **Collection**: interface in Java

- **collection**: object that stores an object
- **Collections**: class with utility methods for classes that implement Collection
- **the Collection API**: group of Java entities for storing objects
    - implement Iterable and Collection
    - List: allows duplicates, retains order of insertion
    - Set: doesn't allow duplicates, doesn't necessarily retain order of insertion
    - Queue: first in, first out (FIFO)
- List
    - ArrayList: backed by an array, better performance for retrieving elements
    - LinkedList: uses nodes that reference previous and next element, better performance for adding elements near the beginning or end
- Set
    - HashSet: backed by a hash table, can't be sorted (have their own sorting)
    - TreeSet: backed by a tree, can be sorted
- Queue
    - Deque: "queue that goes both ways"
    - LinkedList

**maven**

- project management tool, build automation tool
- manages the lifecycle of your application
- manages dependencies using the maven repository
- project structure:
    - src/main/java: where your package/java classes go
    - src/main/resources: where resources like properties or xml files go
    - src/test/java: where test classes go (like JUnit)
    - src/test/resources: resources (like properties) for tests
    - pom.xml: project object model (metadata like settings, dependencies, parent projects)

**unit testing**

- testing is the process of checking to see whether your application's behavior is as expected
    - positive testing is making sure your application handles correct inputs
    - negative testing is making sure your application handles incorrect inputs
- unit testing is the bottom of the "testing pyramid" as it is the most abundant type of testing; it is testing individuals units of functionality
- usually in the context of java applications, this refers to testing methods or small groups of methods

**test driven development**

- this is the process of writing your tests first to establish what functionality is desired for the application, then writing your code to pass the tests

    - usually this requires writing interfaces or empty methods first so that your tests have unimplemented methods to reference when you're writing them, then those methods are implemented to be able to pass the tests
- this is beneficial primarily because it saves time:
    - you already have established desired behavior so less changes need to occur during the development process
    - you can write less code because you can easily find where problems arise right away in the process of making them pass tests (rather than writing all of your code and being unsure where exactly unexpected behavior is coming from)
    - establishing behavior first means that your methods actually have to do the right thing, whereas writing the method first then making a test to pass it won't necessarily guarantee that the method actually does what it's supposed to

**junit**

- JUnit is a unit testing framework that was created around 2000 and standardized test driven development (TDD)
- it includes many annotations and classes to write automated unit tests
- annotations:
    - @BeforeAll : over a method that runs ONCE before all the tests execute
    - @BeforeEach : over a method that runs before each test
    - @AfterEach : over a method that runs after each test
    - @AfterAll : over a method that runs ONCE after all tests have executed
    - @Test : over a method that is a test
    - @DisplayName : allows you to give the test a display name for semantic purposes
    - @TestMethodOrder() : goes over the class and allows you to set an order for the tests to run (they are not consistently ordered by default)
    - @Order(1) : goes with the above annotation if the parameter was OrderAnnotation.class
- assertions
    - import static org.junit.jupiter.api.Assertions.*; // in JUnit 5
    - in JUnit 4, the class is Assert
    - common methods
        - assertEquals(expected, actual)
        - assertTrue(actual)
        - assertFalse(actual)
        - assertArrayEquals(expected, actual)
        - assertThrows(expectedException, executable)
            - the executable is a functional interface so you will need to write a lambda with the code that you are expecting to throw an exception
    - these methods represent what you are actually testing; they will throw an exception if the assertion is not met (therefore, a JUnit test fails when it throws an exception UNLESS the exception is one that is expected through assertThrows, etc. or it is caught in a try-catch)

**generics**

- the ability to have a type that is set at compile time
    - compiler scrubs the generic types and replaces them with the specified types
- public class GenericClass { public void takeEntity(T t) { } public T getEntity() { }
- GenericClass catClass = new GenericClass();
- List list1 = new ArrayList(); list1.add(5);
- List list2 = new ArrayList(); list2.add(new Cat());
- "T" or whatever you choose to name your generic is a placeholder
- good for having more reusable code
    - when you are writing the same methods where the only difference is the type, generics allow you to avoid rewriting the code
- restrict the types like so: public class Calculator { }
    - can use the "instanceof" keyword in method implementations if the type changes the implementation
    - if (t instanceof Integer) { } else if (t instanceof Cat) { }
- specifying multiple generics in the same class (like a map with key-value pairs) Map <K, V> Map<Integer, String> map = new HashMap<Integer, String>();
- see generic dao example

**maps**

- a collection that stores key-value pairs
- does NOT implement iterable, so you can't use an enhanced for loop with maps
- you can get a set of the keys and use that to iterate with an enhanced for loop (mapName.keySet())

**scanner**

- Scanner allows you to read an input stream, usually the console input
- you should only have one Scanner open, and it should always be closed at the end of the application
    - scanner.close()
- Scanner scan = new Scanner(System.in);
- there are several methods for reading the information
    - .next(), .nextLine(), .nextInt(), etc.
    - you should typically use nextLine and parse from there if necessary, as it prevents extra characters like newline characters from being left in the buffer and causing unexpected behavior

**lambdas/functional interfaces**

- functional interfaces are interfaces that only have one abstract method
- some examples that exist in Java are Runnable, Predicate, Supplier, Consumer
- lambdas are implementation of functional interfaces
    - where classes are made for reusable code, lambdas are good for "single-use" code
- a lambda is just a shorthand for a class that implements the one abstract method of the functional interface

    - doesn't require you to instantiate an object of the class separately
    - allows you to avoid cluttering the namespace/code base with a class that only has a single purpose
- FunctionalInterfaceName varName = (parameterNames) -> valueToReturn;
- FunctionalInterfaceName varName = (parameterNames) -> { // things // stuff return something; };
- calling the method on the lambda: varName.methodName(parameter);
- Predicate
    - functional interface whose method is .test(T t);
    - returns a boolean
    - sort of acts like an assertEquals in practice
    - also includes default method implementations .and, .or, .negate
    - using the other methods you can do "Predicate chaining"
        - greaterThanTen.and(lessThanTwenty).test(15);
- Supplier
    - functional interface whose method is .get()
    - returns an object
    - can loosely act like a factory
- Consumer
    - functional interface whose method is .accept(T t)
    - accepts an object
    - used to see "side effects" of accepting an object for something, or passing an object somewhere
- see example on gitlab for the class that is doing the exact same thing as the lambda

**threads**

- sequence of instructions being processed, or code that is currently running
    - multithreading is having multiple sequences of instructions running simultaneously
    - JVM handles threads for us, we have the Java Thread API
    - creating a thread, you can have a class that implements Runnable or that extends Thread
        - when implementing Runnable, you have to instantiate the thread like so:
            - Thread exampleThread = new Thread(new MyRunnableImpl());
            - this is because you need to have the Thread.start() method to run your threads simultaneously
            - your code for the thread goes into an implementation (override) of the .run() method
        - when extending Thread, you have to be careful not to override the .start() method because you can prevent them from running simultaneously
            - MyThread exampleThread = new MyThread();
    - thread.start() calls the .run() method that you implement/override
    - methods in the Thread API
        - .sleep(1000) make the thread pause for the specified time (in ms)
        - .join() makes the thread wait for the other thread before continuing
        - .isAlive() returns a boolean for whether the thread is alive (running)

        - .getState() returns the Thread state
    - Thread states
        - RUNNABLE: active
        - NEW: hasn't started yet
        - TIMED_WAITING: sleeping using the .sleep method or timed .join
        - WAITING: waiting for some condition/resource/other thread
        - BLOCKED: in a synchronized method
        - TERMINATED: the thread has completed running

**singleton design pattern**

- is used to ensure that there is only one instantiation of a class
- enforced with the following conditions in the singleton class:
    - private constructor: we can't instantiate the class/an object using the "new" keyword because that calls the constructor
        - makes it so that the class cannot have children because child constructors call the parent constructor implicitly
    - private static instance of the singleton class: being static enforces that there is only one because it is associated with the class
    - public static synchronized accessor method: this is what allows us to get our private static instance, within the code it makes sure that it is always returning the same one rather than creating new ones, it is synchronized to prevent different threads from accessing it at the same time and potentially creating an extra instance
- good to use for objects that you only want to exist once, such as a Scanner or a database Connection

**factory design pattern**

- is used to return/create an implementation of a specified interface
- used when the implementation doesn't matter or depends on different conditions that may not be available to the calling class
- enforces abstraction because you get an implementation without knowing which one it is; forced to code to the interface

**application setup**

- beans/entities/models
    - our classes defining the types of objects we'll be using in our application
    - examples: Cat, Person, Hero, Enemy
- data access objects (DAOs)
    - our classes that actually interact with the database directly (through JDBC) or data in general (Collections)
    - this is the only layer that interacts with the data directly
    - we use interfaces to define the methods that we will use (CRUD operations)
        - create, read, update, delete

        - implement those interfaces for different types of data access (Collections, Oracle SQL, mySQL, PostgreSQL, etc.)
- services
    - this deals with combining the data access with the actual functionality desired by the user
    - interacts with the DAO layer and the "front" layer
    - might combine things from different DAOs
        - example: adoptCat method, used the PersonDAO to add the Cat to their cats, the CatDAO to update adopted status
- "front"/controller
    - this interacts with the user (in the case of a command line application) or the front end
    - deals more directly with user stories, logIn method, register method, adoptCat method
        - these methods deal both with getting input, sending/retrieving info through the service layer, giving info to the user/front end
- package structure example: com.revature.beans, com.revature.data, com.revature.services, com.revature.controller