Java Primer

In this document, you will jump into the fundamentals and theoretical knowledge of Java software programming. It includes a section for: Java, SQL and Hibernate. Separately, you can follow Revature brand slides for the Spring Framework.

Core Java

Introduction

Introduction to Java: Why Java?

Strictly typed, pass by value, object oriented programming language.

- Portable (**Multiplatform** – If the running system has a **JRE/JVM**).
- Simple syntax, based on C.
- Rich API.
- Java is FREE.
- Support from Oracle Corporation.
- Automatic Memory Management.
  - No pointers.
  - Garbage Collection.
- Is Java 100% **OOP**? (Debate topic).

JDK (Java Development Kit)

What does it include?

- Compiler (Just in Time).
- **JRE** (Java Runtime Environment).
  - **JVM** (Java Virtual Machine).
  - Core **classes** and Supporting **files**.
- **GC** (Garbage Collector).

Basically, the **JRE** is used to **run** Java programs, and the **JDK** is used to **develop and run** Java programs.

OOP Concepts

- **Inheritance**: ability of classes to have **parent's** state and behavior as theirs.
  - A class **extends** from another.
  - An interface **extends** from another.
  - A class **implements** an interface.
  - A class **extends** an abstract class.
  - Multiple inheritance is not allowed in Java.
- **Encapsulation**: control random access of members of a class.
  - Access modifiers.
- **Abstraction**: suppress code complexity and provides reusability.
  - Follows the **black box** principle.
  - *Caring more on **what** something does and not **how**.*
  - Interfaces and Abstract classes.
- **Polymorphism**: ability that classes and methods have **to be** or **to behave** differently at different points in a Java application.
  - Overriding (runtime) and Overloading (compile time).
  - **IS-A** rule (covariance).

Class and Object

Class

- Blue print of objects.
  - They have a **state** (attributes).
  - They have a **behavior** (methods).
  - All classes implicitly inherit from **Object**.

Object

- **Instance** of a **class**.
- They occupy a specific spot in memory.
- Barebones instantiation: **new Class();**

Naming Conventions

- Classes are written in camel case (**Class**).
- Variables, attributes or methods are written in semi camel case (**theVariable**).
- Static and final attributes or variables are written in all caps with underscores (**THE_FINAL_STATIC**).

Basic Concepts

Control Statements

Flow controllers of Java applications.

- **Conditionals**
  - if, if-else, else if, switch.
- **Loops**
  - for, while, do-while.
- **Loops and method control**
  - return, continue, break.
- **Exception Handling**
  - try, catch, finally.

Primitive Data Types

Single value, reserved words, non-object variable types.

- **boolean** (true-false), **byte** (1B), **short** (2B), **char** (2B) unsigned [*integer on the background*], **int** (4B – 32bits), **long** (8B – 64bits), **float** (4B), **double** (8B).
- Primitives are stored in the **Stack**.

Wrapper Classes

Object **representation** of all existing primitives.

- Primitives get **auto-boxed** into wrapper classes automatically.
- Wrappers get **unboxed** into primitives.
- Not compatible wrapper/primitives need to be strictly **casted**.
  - Integer i = 2.0 is **not** possible.
  - Integer i = (int) 2.0 is the possible way.

Operators

Used in flow control statements or arithmetic – String operations.

- **Arithmetic/String**: + , -, *, /, %, ++, --, +=, -=, *=, /=, %=.
  - ++ and -- can be *pre-increment* and/or *post-increment*.
- **Relational**: ==, !=, >, <, >=, <=.
- **Logical**: &, |; &&, || (short-circuit); !.
- **Ternary**: int m = k > 6 **?** 1 : 0;

Constructors

Special methods used to **instantiate** objects using the **class** blue print.

- Methods with no return type.
- Method signature is the **exact** name of the **class**.
- Can be **overloaded** or **encapsulated**.
- **super()** or **this()** can be used to summon parent constructor or another constructor within the same class respectively.
- They implicitly call **super()**.
    - If **this()** is called, then **super()** within that constructor doesn't get called (because it gets called in the other summoned constructor).

Methods
Definers of the **behavior** of classes within Java.

- They contain an **access modifier**.
    - If they don't, it's considered as a **default** access modifier.
- They may or may not be **static**.
- They **must** provide a return type (**void** is considered a return type).
- They possess a **name**.
    - If another method is using it, it needs to be properly **overloaded**.
- They may or may not have **parameters** as **arguments**.
- They may or may not have a **throws** clause to define if the method throws certain types of **exceptions.**

Scopes of a Variable
Locations within code from where a certain variable can be accessed.

- Local (Block).
- Method (Parameters).
- Class (Static).
- Instance (Non-static).

Static Members
Static members are an important concept of the Java framework. Understanding the order of execution of members when a class loads or instantiates, even more.

Order of execution:

1. **Static Attributes**: will be initialized before static blocks execution *when the class loads***,** (makes sense because static blocks might want to access these).

2. **Static Blocks**: a block that executes *when the class loads*, after static attributes are initialized.
3. **Non-Static Attributes**: will be initialized before non-static blocks execution *when the class instantiates*.
4. **Non-Static Blocks**: a block with only curly braces and no static word that will execute after non-static attributes are initialized *when the class instantiates*.

First two steps are going to execute when the application first runs and classes get loaded into memory only once (into the **Stack**), the following two steps will execute every time the constructor for that specific class gets called.

Heap, Stack

Heap

Java Heap memory is where all object instances reside. It also contains the Garbage Collector, which we will talk about in a later topic.

Stack

Java Stack memory holds **primitive** values references and reference to **objects** stored in the heap, used in a specific method execution.

Every method that's going to be executed in a Java program will be pushed into the stack (that's why we are able to see a stack trace when an exception gets thrown).

*Both Stack and Heap are held by the **JVM**. When a class loads, both stack and heap get used.*

First Java Program

Contains:

- Classes, Operators, Primitives, Wrappers, Constructors, Methods, Scopes, Flow Control, Static Members.

Intermediate Concepts

Arrays

Group of primitives or objects settled in a **fixed** size.

- Can be **single** or **multi** dimensional.
- To increase the size of dimensions you need to create new arrays.
- You can access values in the group directly with an **existing** index.

- You can go through all the values using **loop** constructs.

Ways to **declare** an array:

- Type[] type = …
- Type type[] = …
- = {value1, value2, …, valuen}
- new Type[size]
- new Type[]{value1, value2, …, valuen}

Varargs (…)

Convenience strategy that allows a method to receive many parameters as an array of a certain type.

- Values can be accessed as a regular **array** of that type.
- It allows in some cases to avoid **overloading** of methods.
- You can't receive any parameters **before** the Varargs but **after** it's allowed.
- You can pass an **array** of the Varargs type as a parameter.

Access Modifiers

Encapsulate class members with different access modifiers that are conducted in the specific level order (less visible -> more visible):

- private
    - Accessible only within the class.
- default (no access modifier)
    - Accessible only from class and same package.
- protected
    - Accessible only from class, same package or through inheritance.
- public
    - Accessible from **anywhere**.

Final

Three types:

- Final classes: can't be extended (Best example of final class is **String**).
- Final methods: can't be overridden.
- Final attributes or local variables: immutable values once initialized (**Strings** value array).

    - If an attribute of a class is final it must be initialized inline or in all constructors of the class.
        - Default initialization won't be provided.

Packages and Imports

Packages

Organize your code in a modular way with many different levels.

Imports

Import elements from different packages to be used in a class.

- **import** declarations always come after the **package** declaration.

There are different types of imports:

- **Regular Imports**: import classes to your code.
    - **import** com.revature.Class
    - **import** com.revature.* (bad practice).
        - Importing everything from different packages can come to naming conflicts (two classes with the same name, it doesn't even compile).
        - You imported the wrong one and are using it without knowing.
        - Other developers can't tell what you are using at a glance.
- **Static Imports**: import static members present in a certain class.
    - **import static** com.revature.Class.STATIC_VARIABLE
    - **import static** com.revature.Class.staticMethod
    - **import static** com.revature.Class.*

Reflection

A powerful intermediate concept for the Java framework. The Reflection API allows you to access every detail of class information at runtime, like class name, attributes, methods and many others.

For example, you can check if there is any method in a class that receives two integers as parameters, if there is, you can invoke it without even knowing the name or what it does.

Normally, we use the Class.forName() method, that will return a Class object containing all the information of the class name given in String format: "com.revature.model.User".

Use Case

Usage of Reflection depends on your imagination, however, frameworks like Hibernate, Spring and many others use Reflection for many operations.

Second Java Program

Contains:

- Arrays, Varargs, Loops, Imports, Access Modifiers, Final, Reflection.

String

Class that represents an **array** of characters behind the scenes.

- They are **immutable** (see String class final value[]).
- They can be instantiated as **literals**.
    - Literals are stored in the **String pool** (String pool is stored in the **heap**).
- They can be instantiated with the **new** clause.
    - These are stored in the **heap**.
    - If a **String literal** is used to call the constructor and it doesn't exist in the **String pool**, it will be added there, plus of course the actual **new String** on the **heap**.

String Important Methods

*All String methods that return a String are a brand **new instance***.

- length(): return the length of the String.
- trim(): remove blank spaces at the beginning or the end of the String.
- replace(stringToReplace, replacement): **replaces** a specific part of the String with another.
    - If the replacement of the String gives as a result the **same** String (no modifications), the method returns the same object.
- charAt(index): returns the character at the specific **index**.
- toLowerCase(), toUpperCase(): returns a **new version** of the String all in lower case or all in upper case.
- **Explore** other methods.
- Wide variety of **constructors**.

StringBuilder, StringBuffer

Builder and Buffer allow to have something like a **mutable** String.

- They increase or decrease their size **dynamically** when adding or removing characters of the String.
- They are stored in the **heap**.

StringBuffer is **synchronized**, StringBuilder is **not**.

Methods

- append(): appends a character or a set of characters to the **end** of the String.
- indexOf(String): returns index of the first occurrence of the given String.
- length(): returns the length of the String.
- reverse(): returns a reverse version of the object.
- ensureCapacity(minimumCapacity): ensures that the minimum size of the object is the specified.
- **Explore** other methods.
- It only provides **four** constructors.

Synchronization

A Synchronized class it is also known as thread-safe. All of it's method use the reserved word: **synchronized**.

A synchronized method states that it cannot be executed by many threads at the same time. The method will be locked when a specific thread is executing it, that means the other thread that needs it will wait.

We will talk more about threads when we reach that topic.

Equals, ==

If a class **doesn't** specify an overridden equals() method, they **both** are exactly the same. They both return true if the object reference (position in memory) is the same.

In the case of **Strings**, the String API possesses an overridden version of the equals() method that compares character by character.

hashCode()

The hashCode of an object (**by default**) specifies their position in memory as an **integer** value. However, hashCode() method can be overridden by any class to provide a different type of hashCode.

In the case of **String** class**,** has an overridden version of the hashCode() method that compares depending on the actual value of the String and not the position in memory.

This can be used to optimized memory management, since if there is no overridden hashCode method: two objects that are actually the same, **will have different hash codes.**

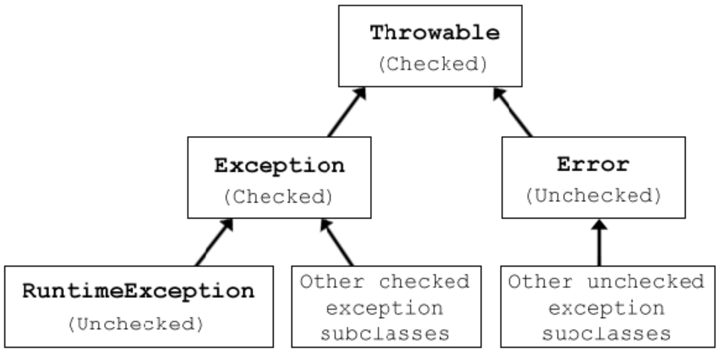
Best example is to compare two different instances of the Object class's hash code against comparing the hash codes of literal string and its object dopelganger.

Exception Handling

Exception

Java special kind of objects that arise based on a specific type of event, normally representing errors.

The Exception hierarchy goes as follows:



Throwables are objects that might by catched during program execution.

- **Exceptions** are checked exceptions.
    - Which means that at **compile** time there will be an error saying that exception needs to be catched (with a try catch block) or ducked (with a throws clause).
- **Errors** and **Runtime Exceptions** are unchecked exceptions.
    - Which means that they can be catched or ducked at **runtime** but won't be forced at **compile time**.
        - Runtime Exceptions **sometimes** makes sense to handle.
        - Errors normally **never** make sense to recover from because normally this are fatal states of a program like (OutOfMemory, StackOverflow).

To throw and exception use: **throw**.

REVATURE

- throw null, is **allowed** (OCA question).

To duck an exception, add the **throws** clause at the end of a method signature.

- Overriding ducking methods follow the same concept as catch clause (broad has to be at the beginning).

Try types

- Regular try.
- Try with resources (closes resources for you. E.G.: JDBC connection).

Catch rules

- There can be as many catch blocks as needed.
- More broad exceptions need to be specified **last** (unreachable code if not).

Finally

- Block of code that **ALWAYS** runs, regardless of unsuccessful try or catch block (it even runs if an **error** gets thrown).
- Finally **doesn't** need a catch block.
- The only case it will not execute is when the system crashes, or System.exit(0) gets called.

Garbage Collection

Automatic process within the **JVM** that deletes instances of objects from memory once there are no references in a program for them.

- Barebones object instantiation (new Object()) can explain this better.
- Garbage collection can **never** be **forced**, only **suggested** (System.gc()).
- Object method **finalize**() gets called by the garbage collector once an object doesn't have a reference.
    - It can be overridden to provide additional clean up.

After final is done, remind them of Final vs Finally vs Finalize.

Polymorphism, Inheritance & Abstraction in-depth

Overloading

- Method overloading it's also known as compile-time polymorphism.
- Method overloading specifies that methods with the same name can exist within the same or an inheriting class as long as they follow a specific set of rules.
  - Parameters type and/or count and/or order is different.
  - If the return type or the throws declaration is different, the method is still not properly overloaded if it contains the same parameters.
    - Constructors can be overloaded following the same rules, but they don't have a return type (?).
- Overloading is the **only** type of polymorphism that doesn't necessarily require Inheritance.

Overriding

- Method overriding it's also known as run-time polymorphism.
- An overridden method doesn't strictly require the @Override annotation.
  - The annotation is used to confirm that a method it's properly overridden and for best practices purposes.
- Method overriding specifies that a method behavior can be changed in a hierarchy, as long as it follows a specific set of rules:
  - Parameter type and count have to be exactly the same.
  - Return type has the same or a child class.
  - If throws clause exists on the parent method, child method can be written without it, have the same type of exception or less broad exception.
  - Access modifier can't be less broad.
    - If the parent method is default, the child can be protected or public.
    - If the parent method is private, it can't be overridden.
  - Synchronized word is not required on the child class.
    - And that means child version is not synchronized.
- Static methods can't be overridden, they can only be overloaded or shadowed.
  - They are part of the class.
  - Shadowing is a different concept than Overriding.
    - A method can exist in a child class like you were overriding, but if you use the @Override annotation, a compile time error is shown.

    - Final and static methods can’t be shadowed (however, the compile time error shown says “it can’t be overridden”, but we just said static methods can’t be overridden, only shadowed).

IS-A

- The IS-A rule, the most powerful kind of polymorphism.
    - **Conceptually uses every single pillar**.
- It states that a class can become or be another class as long as it follow a specific rule.
    - A class that extends or implements another class, can become anything in the hierarchy as long as it follows the IS-A rule.
    - A Student IS A Human -> A Human, is not necessarily a Student.

Casting

Objects and primitive values can be up-casted and/or down-casted.

For Objects:

- The instance of an object can become a child class if it is explicitly casted (down-casting).
    - If the class specified in the cast doesn’t follow the IS-A rule, a runtime ClassCastException will be thrown.
- The instance of an object can become a super class without being explicitly casted (up-casting).
    - If the IS-A rule is not followed while doing this, a compile error is shown.
- Whether it is up-casting or down-casting, the existing count of objects in the **heap** is always **one**.

For primitive values:

- A primitive value can become another primitive value with less capacity if explicitly casted (down-casting).
- A primitive value can become another primitive value with more capacity without explicitly casting (up-casting).
- If up-casting or down-casting happens, a **new version** of the primitive value is created on the **stack**.

Shadowing

Class members can be shadowed. We talked about shadowing static methods through inheritance, well, variables can be shadowed as well.

- If the parent class contains a variable called *public int i*, the child can have it declared as well.
    - Let's say the parent class is Human, and the child class is Student.
        - If Human human = new Student(); is used, and human.i gets called, the *i* from human gets shown.
        - If then we do Student student = (Student) human; and student.i gets called, the *i* from student gets shown.

Interface

Part of the abstraction pillar, interfaces define a contract for whichever class implements it. These are some characteristics of an interface:

- All contract methods are implicitly **public** and **abstract**.
    - Keywords can be used, but they are already there.
- Interfaces can have variables, they are implicitly public, static and final.
- Interfaces are normally use to the define the top of a hierarchy.
- In Java 8, interface can provide behavior for some of the methods, using the ***default*** keyword.
    - Abstract classes still have an advantage in access modifiers (because interfaces only have public methods).
    - Personally, I don't recommend it, only use this when you want to provide some **default behavior** at **the top of a hierarchy**.
        - That means you can't provide half way implementation (an interface cannot extend or implement an abstract class, of course).

For the inheritance part:

- Interface use the ***extends*** keyword between each other.
- Classes or Abstract classes use the ***implements*** keyword to provide behavior.
    - This is one of the main reasons you would use an interface over an abstract class, a class can implement multiple interfaces (you can't extend multiple classes).

Abstract Class

Part of the abstraction pillar like interfaces, abstract classes are a specific type of class that share similarities between interfaces and regular classes.

- They can provide abstract methods like interfaces.
  - With the difference on access modifiers: it can also use **protected** and **no access modifier** besides **public**.
- They can provide concrete methods (behavior).
  - *If Java 8 interfaces can provide something like these, then, why abstract classes? (Discuss after finishing the section).*
- They can have static and non-static variables.
- They can have constructors.
  - Utilized by child classes when they call super().

Abstract classes are normally used to provide half-way implementation and define more generic/general parts to a class hierarchy (you want to implement part of the contract in a more general way, but not all of it).

A class hierarchy should always start with interfaces, then it can have more interfaces, then abstract classes, then concrete classes.

Collection API

Generics

They are used to **define** a class type or types (They can be used also to define method types but these are more complex).

They are used popularly with **collections**.

- They use the **diamond** syntax (<>).
- Java doesn't accept **primitives** as a type (use **Wrappers** instead).
  - They accept arrays of primitives (because they are technically objects).
- They can define many types (<,>).
- Collections accept creation without generics, however, this gives a warning and is not best practice at all (compile time safety).

Iterable, Iterator

Iterable<T>

Enables use of enhanced for loops over collections.

Iterator<T>

Used to iterate through collection Objects in a different way.

- Every Class that implements Iterable<T> possess an iterator.
- collection.iterator().
    - while(iterator().hasNext()).
    - iterator().next().

Collection

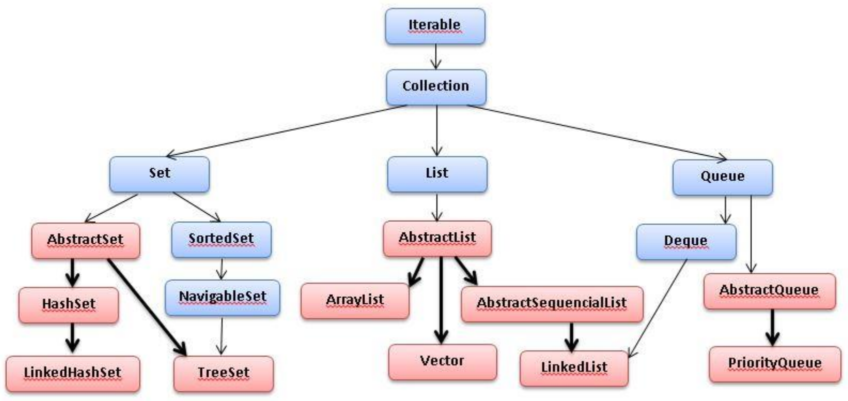
A collection represents a **group** of objects, known as its **elements**.

- Is the root interface in the collections hierarchy.
- More specific collection interfaces are defined in a sub level: List, Set, Queue.

There are three ways to iterate over Collections:

- Enhanced for loop (this is possible because of the **Iterable** interface).
    - Remind them that enhanced for loops can also be used for arrays but they are not iterable.
- Obtaining the iterator from the collection.
    - while (iterator.hasNext())
        - iterator.next() [Would be the object].
- Classic for loop from 0 to it's size – 1.

The hierarchy of collections is described as follows:



For **associates**, the most important part to know about the hierarchy is the three main interfaces (Set, List and Queue), and the most used Collections (TreeSet, ArrayList and LinkedList).

Set

- They don't accept duplicate values (only one null value is accepted).

    - o1.equals(o2) **x.**
- User can insert objects freely but it cannot control in which index this object is going to be (since normally we use SortedSet and when sorting happen, objects move).

HashSet

- Uses a HashMap behind the scenes.
- That means it doesn't guarantee order of elements.

SortedSet

- Automatically sorted set if group type defined in the collections implements Comparable or a Comparator is provided in concrete SortedSet constructor parameter.
- Doesn't accept duplicates depending which attribute compareTo() or compare() from **Comparable** or **Comparator** compare respectively.
    - If the POJO compares by id, two objects with the same id won't be accepted in the Set.

TreeSet

- Most used SortedSet in the Collections hierarchy.
- Automatically sorts objects each time one is inserted, handling a B-Tree behind the scenes.

List

- They accept duplicate values.
- Also known as sequences.
- User has control over where objects are inserted (specific index).

ArrayList

- Uses a regular array behind the scenes.
- If an object is inserted and the size of the array is not enough, instead of increasing size by 1, it's increased by a 150%.
- **Faster** for object retrieval, **Slower** for insertion.

LinkedList

- Uses a double linked list behind the scenes.

- **Faster** for object insertion, **Slower** for retrieval.

Vector

- **Synchronized** version of ArrayList.

Queue

- There are LIFOs, priority queues and FIFOs (stacks).

Deque

- Short of “double ended” queue.
- Allows insertion and removal at both ends.

Collections Utility

Contains utility methods for Collections. E.G.: Collections.sort(Collection<T>).

Comparable, Comparator

For collection **sorting**, objects in the collection **must** implement the Comparable<T> interface.

If the objects don’t implement Comparable<T>, an instance of a Class that implements Comparator<T> **must** be given in the SortedSet constructor.

If non of these are given, a runtime exception gets thrown.

Comparable Implementation

- compareTo(Object)
  - return this.compareTo(Object).
  - If **this** is used first the order of sorting will be **ascending**.

Comparator Implementation

- compare(Object1, Object2)
  - return Object1.compareTo(Object2).
  - If Object1 is used first the order of sorting will be **ascending**.

Map

Object that maps **keys** with **values**.

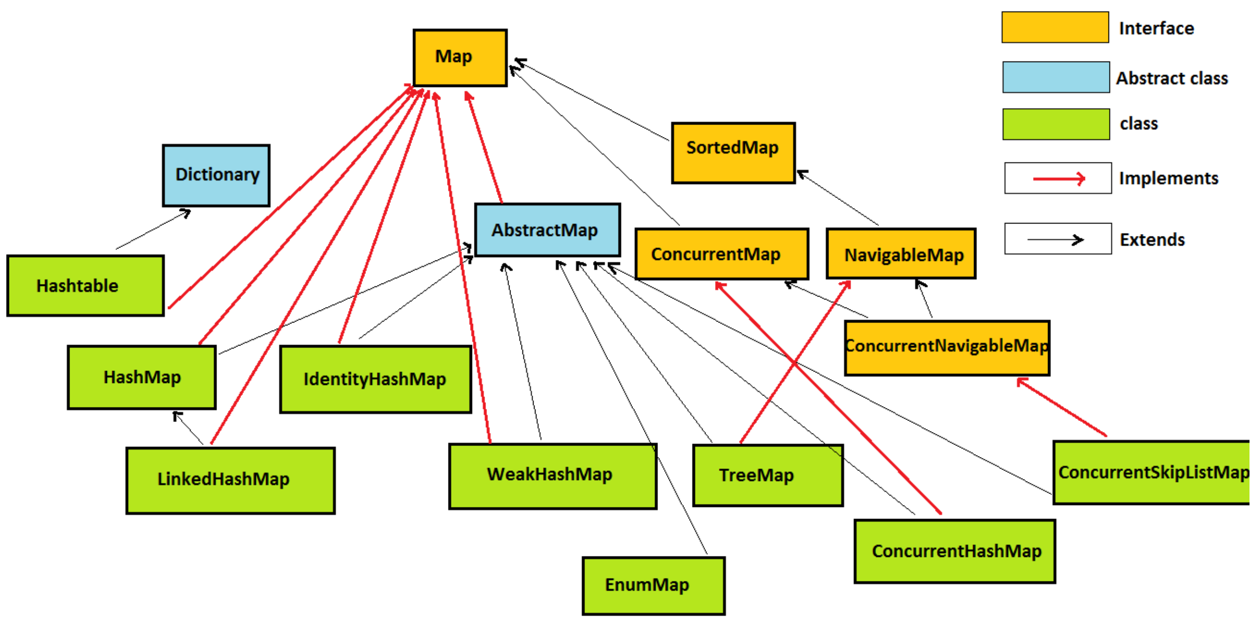
- Keys are represented with a **Set**.

- Values are represented with a **Collection**.
- Entry **sets** can be obtained as well with a Key, Value set.
- Maps use the **put**() method instead of **add**().

There are three ways to iterate in a Map:

- Enhanced for loop over the key set (map.keySet()).
  - Then access values with each key (map.get(key)).
- Enhanced for loop over the values collection (map.values()).
- Enhanced for loop over map.entrySet() (**most efficient because no search is needed**).
  - for (Map.Entry<K,V> entry : map.entrySet()).

The simple version of hierarchy of maps goes as follows:



Hashtable vs HashMap

- Hashtable is **synchronized**, HashMap is not.
- Hashtable doesn't allow null keys or values. HashMap allows **one** null key and any number of null values.

Sorting

- **TreeMap** is the TreeSet version of a map, that means that we can actually sort by Comparable or Comparator.

REVATURE

- Elements will be sorted by the key.

Multi-Threading

Runnable

Parent interface of **Threads**.

- Classes can implement the Runnable interface and implement the run method.
    - Override run method to perform a routine.
    - To run the thread: new Thread(new Class()).start().

Thread

Independent line of execution. Implements Runnable.

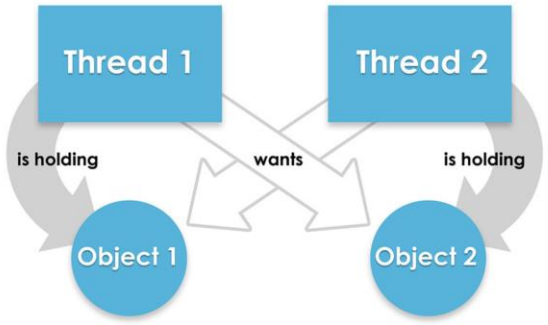
- Classes can extend Thread.
    - Override the run method to perform a routine.
    - To run the thread: new MyThread().start().
- Use thread.isAlive() to check if thread is still running.
- Use thread.join() to make current thread wait until thread is dead.
- Implement Runnable preferred over extending Thread because it allows to implement (and extend other meaningful classes).
    - Also, Runnable is an interface, that means the run() method overriding it's forced.
        - The Thread class contains an implementation for the run() method (it does **<u>nothing</u>**).

States of a Thread

NRBWTT

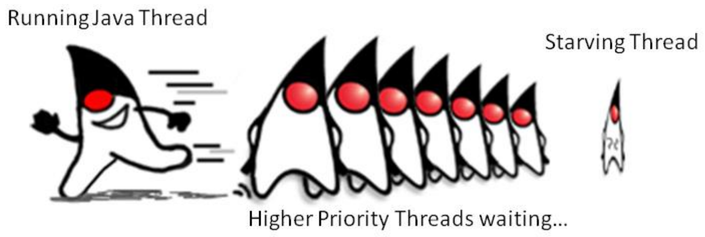
- **NEW**: A thread that has not yet started is in this state.
- **RUNNABLE**: A thread executing in the Java virtual machine is in this state.
- **BLOCKED**: A thread that is blocked waiting for a monitor lock is in this state.
- **WAITING**: A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- **TIMED_WAITING**: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- **TERMINATED**: A thread that has exited is in this state.

Deadlock



Starvation

A thread is waiting **forever** to have CPU time allocated.



Producer, Consumer

I/O

Scanner

A text scanner which can parse primitive values and strings. A scanner breaks inputs by white spaces by default, but it can be set to a different delimiter (all this if it's done in the same line, nextLine() will delimit by the enter character).

We can receive user input through the CLI if we send *System.in* to the constructor of Scanner.

It is normally used to receive user input anyways, however, we can send any kind of InputStream to the Scanner and parse (it can be a plain file, an XML, etc.).

File I/O

The Java API offers many different utilities for file manipulation (input/output -> read/write).

Byte Streams

Writing or reading byte by byte.

Different types of data can be read with these type of streams (text files, audio files, string buffers, etc.).

- All reading classes inherit from **InputStream**.
- All writing classes inherit from **OutputStream**.
- Use true or false in the constructors to define **appending** or not.

The most commonly used are:

- **FileInputStream**.
- **FileOutputStream**.

If we read bytes, we somehow must transform it to something readable.

- For example, we could use the String class constructor that receives a byte array.

Declare it:

- *new FileInputStream("/filepath/file.txt")*

Extra: Since these are bytes, everything bigger than a byte will show as the wrong character.

Character Streams

Writing or reading character by character.

- **FileReader.**
- **FileWriter.**

Declare it:

- *new FileReader("/filepath/file.txt")*

Line-Reading Streams

Writing or reading line by line or **bulk optimized**.

- **BufferedReader**
- **BufferedWriter**

Wrap FileReader and/or FileWriter into this classes.

- *new BufferedReader(new FileReader("/filepath/file.txt"));*

One or the Other?

The style of writing or reading of these classes are very important. By testing, the order of speed is: Buffered > Char Reading > Byte Reading; going from faster to slower.

The Buffered version is the optimized version and **faster file I/O** available.

Serialization

Is the process of transforming an object into a byte stream, or vice-versa. This process can be used to either write an object directly to a file (persist it) or transmit it through the network (most commonly used for this reason).

In order for an object to be able to be transformed into a byte stream, it must implement the **Serializable** marker interface. A **marker interface** doesn't contain any method at all (it's empty). It tells the JVM that this object is able to do something at run-time.

We can decide to make one of the variables to not be serialized (not written in a file, not shown, not transmitted), for that, we use the keyword **transient**.

- A good example of this would be any of kind of sensitive field that we don't want to transmit in some cases (passwords, etc).

Writing or reading an object.

- **ObjectInputStream**.
- **ObjectOutputStream**

Wrap a stream with these classes.

- *new ObjectInputStream(new FileInputStream("/filepath/file.txt"));*

Remember: Byte stream != Byte code.

Creational Design Patterns

Singleton

The singleton design pattern states that only **one** instance of a specific object can exist. It is normally used for optimization purposes when many instances of a specific object are not required (business objects are not a good use case of a singleton: data access, services or controllers are best case scenarios).

To implement a Singleton:

- Make the constructor **private**.
- Define a **private static** attribute which will contain the instance of the object.
- Provide a **static method** that returns the instance of the object.
    - If the static attribute is null, return a **new** instance.
    - Else, return the static attribute.

Singletons are used in a wide variety of frameworks by default, such as **Spring** or **Angular**.

Factory

The factory design pattern provides control over the creation of many types of objects, that are somehow related. It can be seen as a server in a restaurant, which brings only the food or drinks that you asked.

To implement a factory:

- Provide a general super abstract class or interface.
- Have some concrete classes implement the abstract methods.
- Create a factory class which will have a method that returns the type of the super class and receives specific information about the object that needs to be created.
- Have a switch statement in the method that checks for the parameter received and returns a specific type of objected depending on that.

Factories are used widely in many frameworks as well, and they are normally used when some specific information is available (a string containing the type, some properties of the object, etc.), but the class that's needed for that information is not known.

Java 8

Lambda

Lambda expressions are a very important concept introduced in Java 8. They are normally used to implement ***functional interfaces*** on the go.

Functional Interface

Functional interfaces are not more than interfaces with only **one** method declared as part of the contract. Good examples of this interfaces are *Runnable* and *Comparator*.

Rules

- Once the functional interface exists, a lambda expression can be performed:
    - *MyInterface i = (parameters) -> { //Code };*
    - Then the name of the method can be executed from the variable.
    - The left side of the lambda has to be a reference, or the lambda itself needs to be put as a parameter of something that expects the type of the functional interface (more clear with Runnable example).

Use Case

Lambdas are widely used in many technologies such as Big Data and Artificial Intelligence. Sometimes you don't necessarily need to provide a reusable concrete implementation, but you need something to be done with less code and on the fly.

Lambdas are widely used in JavaScript as well, but they are slightly different from the Java ones.

Stream

Utility class that supports functional-style operations on **streams** of elements.

Streams are different than collections because:

- They have **no storage**, which means they have **no data structure**. They are a source of information that can be gathered from different sources.
  - Collection API .stream().
  - Arrays.stream(Object[]).
  - And many others.
- **Functional** in nature, all operations in a stream produce a result without modifying the source. A filter operation on a stream produces a new stream without the filtered elements rather than modifying the actual data structure.
- **Laziness-seeking**, many stream operations such as filtering, mapping or duplicate removal can be performed lazily, which opens options for optimization.
- **Unbounded**, collections are finite, streams can be infinite and there are operations to handle these.
- **Consumable**, the elements of a stream are only visited during the lifecycle of the stream, in order to perform a new operation on a stream, a new stream needs to be gathered.

Use Case

In combination of *Lambdas*, streams are great fit to handle massive amounts of data in a very functional way. For sure, one of the biggest use cases is Big Data, since MapReduce-like operations can be performed. However, they can also be used to provide collections manipulation in a more functional manner.

Predicate

Predicates in math are sentences that a establish that a certain condition is met: P -> Q.

In Java, it is a functional interface (Predicate<T>) that can be used to perform true/false like operations on a group or collections of like elements.

A predicate being a functional interface means that we can pass it as a parameter with a lambda expression whenever a Predicate is expected.

- Stream<T> filter (Predicate<T> predicate);

Use Case

More than obvious, it not only combines with *Lambdas*, but with *Streams* as well, to perform true/false like operations on the fly.

Predicates can also be found as Strings with a pattern, normally used in AOP (covered in the framework week).

Date

LocalDate, LocalTime and LocalDateTime (composite one) are classes found in the Java 8 API utilized to perform or gather date and time operations.

This should be used instead of the Date class, because Date is outdated and performing certain operations can be tricky.

Some of the operations that can be performed are:

- .now().
- .of(2012, Month.DECEMBER, 12, 9, 54).
    - Many overloaded versions.
- .parse(“10:15:30”, DateTimeFormatter.ofPattern(“dd-MM-yyyy HH:mm”).

Test & Automation Introduction

Maven

A **build automation tool** used to manage project dependencies. Manual including of jar files into a project is not the recommended way, Maven can handle this for you and it can also easily be integrated with **continuous integration** and **testing** tools.

Maven has two types of repositories:

- **Central Repository (Remote)**.
    - This are maven servers in the cloud which contain a wide variety of jar files (specific versions).
- **Local Repository**.
    - Which is located in the home of the computer in the .m2 folder.
    - Maven will first check if the jar exists in the local repository and only go to the central one if the jar’s not there.

Maven dependencies are located in the **POM** (project object model), which is a well-formed **XML** file. It contains different tags like:

- <dependencies>
    - Group of dependencies.
- <dependency>
    - Group of properties of a dependency.
- <groupId>
    - Structure of the package of that jar (com.revature for example).
- <artifactId>
    - Name of the jar.
- <version>

    - Version of the jar.
- And many others.

Once the changes on the **POM.xml** are saved, the IDE will start downloading the required jar files if no errors are found in the XML.

A Maven **build** is specific amount of steps, known as **goals,** that will execute in a specific order to provide an executable file (a jar or war file in our case). A Maven **goal** will execute all previous ones as well. These goals are also known as the **build** life cycle:

- **validate**.
    - Validate that project information is correct (POM.xml parsing) and all the required information is available (dependencies exist in the central or local repository).
- **compile**.
    - Compile the project source code.
- **test**.
    - Run all test cases of the project.
- **package**.
    - Most widely used.
    - Will provide an executable file in the /target folder of the project.
- **verify**.
    - Verify any quality checks after integration and testing.
- **install**.
    - Install the package into the local repository, so it can be used by other projects.
    - Installing ojdbc7 is the example provided.
- **deploy**.
    - Copies the final package to the remote (central) repository so it can be used by other developers (sharing).

If the package goal is executed: validate, compile and test will happen, as well.

JUnit

A testing framework for Java enterprise applications. It integrates naturally with build automation tools (Maven) and continuous integration tools (Jenkins). It is normally used to provide quality assurance to all project functionality. JUnit provides a variety of annotations that can be used to provide testing functionality.

To specify that a method is a specific **test case**, the @*Test* annotation is used on top of it. Different logic can happen in a test case:

- Verification.
    - Asserts can check that the output of a method is the expected output. The test fails if any assert condition fails.
        - Equals.
        - False.
        - True.
        - Others.
- Performance.
    - @Test(timeout=100), the test fails if the test method takes more than a 100 milliseconds to execute.
- Exceptions.
    - @Test(expected=Exception.class), the test fails if the expected exception is not thrown.

Test cases have a life cycle, which can be specified with certain annotations.

- @BeforeClass.
    - It will happen before any test cases executes.
    - This will only execute only once.
    - Used to prepare global mock data or initialization.
- @Before
    - Executes before every single test case.
    - This will execute as many test cases exist.
- @After.
    - Executes after every single test case.
    - This will execute as many test cases exist.
- @AfterClass
    - It will happen after all test cases execute.
    - This will only execute one.
    - Used to provide clean-up and close resources.

Testing is a very important field of software development, every developer should at least know the basics, and there are specific professionals that only focus on this area.

Log4j

A logging library for Java that is used almost as a standard in any Java enterprise application. Log4j can handle different **types of output** files like: **console**, **text file** and **HTML**. It also provides different **levels** of logging:

- **FATAL**, will show fatal errors.
- **ERROR**, will show any kind of throwable.
- **WARN**, defines a potential error situation.
- **INFO**, for informational messages.

- **DEBUG**, for debugging information (Tomcat, Spring, Hibernate -> Environment setup for example).
- **TRACE**, for verbose of everything that the code is doing (good for testing purposes).
- **ALL**, will show all logging levels.
- **OFF**, will turn off logging.

Similar to Maven goals, if WARN is the logging level, ERROR and FATAL levels are going to be logged as well.

Log4j configuration file should be called **log4j.properties**, and usually it's located under the resources of the project (this might change).

Logging should be part of every project, it's the standard and good practice and provides many advantages:

- Centralized way to control different logging levels.
    - Logging code by itself can be centralized with AOP as well.
- Helps with troubleshooting and making sure everything is running smoothly.
    - It can also provide excellent tracing when used in conjunction with AOP.
- Provides automatic file output, which is very helpful in remote servers where the console will not be visible (*System.out*).

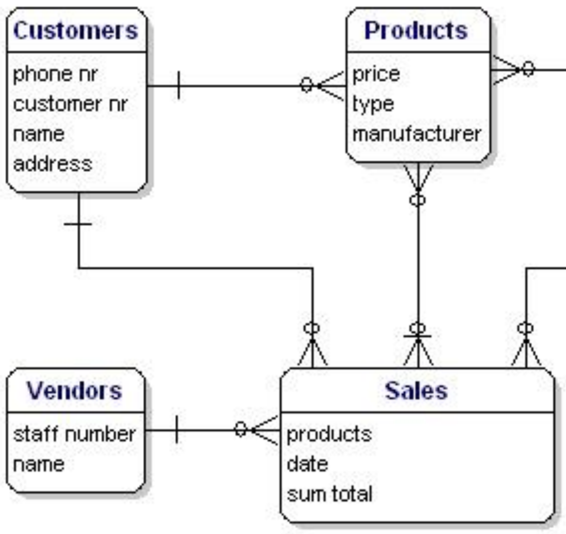
SQL

Introduction to RDBMS

Relational DataBase Management Systems, one of the **key** components of any enterprise application or system.

Data by itself **doesn't mean** a lot, think about it, a ton of diverse data thrown in a bucket, would it mean something? Well, a lot of **analysis** would have to come through. **Related data** it's what gives the business meaning, "an employee is in one or more departments".

To represent the design of a relational database, the James Martin "Crow's Feet" notation is the standard; also known as **ERD** (Entity-Relationship Diagram).



Normally, relational databases are used in an **OLTP** (OnLine Transaction Processing) environment, which means that the idea of having **related data** is preferable in a very transactional system, and that are normally **row-based**.

For non-transactional environments, the counter part is **OLAP** (OnLine Analytic Processing) systems, which are normally **columnar-based**, which is faster for reading but slower for manipulation.

There are also non-relational databases, which are known as **NOSQL**, they utilize Key-Value pairs structures and are becoming very famous in many fields like Big Data and Artificial Intelligence.

SQL
Structured Query Language, is an extensive statement oriented language that is used to communicate and perform any kind of manipulation operations on relational databases.

Sub Languages
All SQL statements are separated into different sub categories, also known as sub languages.

DML
Data Manipulation Language statement are used to perform **CRUD** operations on the actual data. Operations are normally performed by row in a relational database.

- **INSERT**, to insert a new row into a table.
  - ***INSERT INTO*** *TABLE_NAME* ***VALUES*** *(V1, V2, …, VN).*
  - ***INSERT INTO*** *TABLE_NAME (C1, C2, C3)* ***VALUES*** *(V1, V2, V3).*
- **UPDATE**, to update one or more rows column values of a table that match a specific **WHERE** clause.

REVATURE

    - ***UPDATE** TABLE_NAME **SET** C1 = V1, …, CN = VN **WHERE** (…)*
- **DELETE**, to delete one or more rows of a table that match a specific **WHERE** clause.
    - ***DELETE** TABLE_NAME **WHERE** (…)*
- **SELECT**, to obtain one or more rows of a table that match a specific **WHERE** clause.
    - In ORACLE databases this one is considered **DML**.
    - This is how we perform queries in a database.
    - ***SELECT** C1, …, CN **FROM** TABLE_NAME **WHERE** (…) **GROUP BY** (...)*
    - ***HAVING** (…) **ORDER BY** (…)*

DDL

Data Definition Language statements are utilized to define the database schema or skeleton. Is how we implement the design structure of it.

- **CREATE**, to create new objects or tables.
    - ***CREATE TABLE** TABLE_NAME (C_NAME C_TYPE C_SYZE [NULL | NOT NULL], [CONSTRAINT])*
- **ALTER,** to modify existing objects or tables.
    - ***ALTER TABLE** TABLE_NAME [ADD | MODIFY | DROP] C_NAME.*
    - ***ALTER USER IDENTIFIED BY** PASSWORD.*
- **DROP**, to delete existing objects or tables.
    - **DROP TABLE** TABLE_NAME [CASCADE].
- **TRUNCATE**, to delete all the data existing within a table leaving the skeleton of the table only.
    - **TRUNCATE TABLE**.
    - Similar to performing **DELETE TABLE** with no where clause, the key difference is that **TRUNCATE commits** at the end of the operation.

All DDL operations **cannot be rolled back**, which means that any change made by these are **permanent**.

DCL

Data Control Language statements are used to manage the security and control of database systems.

- **GRANT**, to grant any permissions to an existing user.
    - **GRANT** PERMISSION **TO** USERNAME.
- **REVOKE**, to revoke any permissions of an existing user.
    - **REVOKE** PERMISSION **TO** USERNAME.

TCL

Transaction Control Language statements are utilized to manage transactions within a relational database (transactions topic will be covered after).

- **COMMIT**, any DML operations that where executed before the statements will be persisted permanently.
- **ROLLBACK**, any DML operations between two **COMMIT** statements will be completely erased (something like Ctrl + Z that will stop only when it reaches last time you opened the specific file).
  - **Committed** transactions cannot be **rollbacked.**
- **SAVEPOINT**, utilized to **ROLLBACK** to a specific point in time.
  - [Many DML Operations].
  - **SAVEPOINT** A.
  - [Many DML Operations].
  - **ROLLBACK TO** A.

DQL

Data Query Language, not really a sub language within Oracle databases, is the sub language where only the **SELECT** statement exists. Remember, in Oracle the **SELECT** statement is considered as **DML**, but **DQL** could be an interview question.

Data Types

As any database, columns can have specific data types assigned. Here is a list of the most used or relevant ones in an Oracle database.

- **VARCHAR**(size), old version of strings.
  - Max size is 4000 characters.
- **VARCHAR2**(size), the new version of strings and the one we should use.
  - VARCHAR2 has **optimization** as an additional.
  - If the size is 4000 and only 20 are being used, the engine will optimize and not use the other 3980 spaces, they will be somehow reserved.
- **NUMBER**(Precision, Scale).
  - Max value is 999...(38 9's) x10^125, and can be represented with a 38 digit full *mantissa*.
  - For 99999,99 you need NUMBER(7,2)
- **DATE**.
  - Handled as a big integer behind the scenes which represent the amount of milliseconds that have passed since 4800 B.C.
- **TIMESTAMP**.
  - Date's doppelganger, is more precise because of the way it handles fractional seconds (nano seconds included).
- **CLOB**.
  - Maximum capacity is 4GB.
  - It is normally used to store text binaries (word documents, pdf, text files, etc.).
- **BLOB**
  - Maximum capacity is 4GB.

    - It is used to store any kind of binary fil (images, sound, etc.).
- **BFILE**
    - It doesn't store the actual file in the database.
    - It's a pointer to the file in the file system.

Constraints

Rules that are applied to a specific column.

Identity Integrity

- **PRIMARY KEY**.
    - Used to **uniquely identify** each row of a table.
        - UNIQUE and NOT NULL.
    - Can be applied to many columns (composite primary key).
    - They can be Natural.
        - Represent something of the actual entity (e.g.: SSN).
    - They can be Surrogate.
        - The most commonly used, an integer value that increments.
    - Numbers are recommended for these columns, number comparison is faster.
- **UNIQUE**.
    - Values must be unique, and only one null value is accepted.

Domain Integrity

- **NOT NULL**.
    - Mandatory values.
- **CHECK**.
    - Values can only vary from a specific group of values.

Referential Integrity

- **FOREIGN KEY**.
    - They provide **referential integrity**. Which states that this column has to point to an existing row in another table.
    - Used to establish a relationship between two tables.
    - Normally points to a primary key of another table.
    - They can be null (not pointing to anything).

Sometimes Considered

- **DEFAULT**.
    - If no value is specified for the column, the default would be used.

Multiplicity

There are two important concepts which can generally be confused easily: **Cardinality** and **Multiplicity**. They both express the same thing but they are used in different contexts.

With **Cardinality** we say: “A student can have one or many courses”. With **Multiplicity** we are a little bit more specific: “1 Student can have 0..N courses”.

Aggregate Functions

These functions are used to **combine** (aggregate) the values existing in one **column**.

- MAX(COLUMN), which returns the max value on a column.
- MIN(COLUMN), which returns the minimum value on a column.
- AVG(COLUMN), which returns the average value of the column.
- SUM(COLUMN), which returns the sum of the column.
- COUNT(COLUMN), which returns the count of elements in a column.
- And many others.

This functions are used in the **SELECT** clause. They **can’t** be used in the **WHERE** clause.

If there is more than one column being selected in the **SELECT** column section of a query which is not aggregating, a **GROUP BY** clause is required.

In order to perform similar **WHERE** clause Boolean operations with aggregate functions, the **HAVING** clause can be used.

GROUP BY & HAVING

The **GROUP BY** clause will combine **all rows by a column specified** in a query and perform any aggregate functions which are stated.

- **SELECT** NAME, **COUNT**(NAME) **FROM** STUDENT **GROUP BY** (NAME);

The **HAVING** clause will pass another filter similar to the **WHERE** clause after everything has been filtered and grouped.

- **SELECT** NAME, **COUNT**(NAME) **FROM** STUDENT **GROUP BY** (NAME);
    - **HAVING COUNT**(NAME) > 5;

If you try to perform this **HAVING** clause in a **WHERE** clause, a SQL error will be thrown, and it makes sense, the RDBMS doesn’t want you to perform an aggregate function combining all rows, per each row, it’s a performance safety measure.

Scalar Functions

Different than aggregate functions, scalar functions will perform the operation **per row**, and can be used in the **SELECT** or **WHERE** clause.

- TO_CHAR(DATE,'DATE_FORMAT').
- TO_DATE('DATE','DATE_FORMAT).
- UPPER('VALUE').
- LOWER('VALUE').
- NVL(CANBENUL, 'VALUE' OR NUMBER)
- COALESCE(CANBENUL, CANBENUL)
- And many others.

To write them in a query:

- **SELECT UPPER**(NAME) **FROM** STUDENT;
- **SELECT** NAME **FROM** STUDENT **WHERE UPPER**(NAME) **LIKE** 'P%'.

Result Set
What do DML statements return.

Index
- Think of a phone book index, these are created separately pointing to a specific column for faster retrieval.
- Primary Key AND Unique columns are automatically indexed by default.
- Clustered: It's part of the table, it technically makes the rows a SortedSet.
- Non-clustered: It's kept separately, which means the rows are in a regular Set, order is not guaranteed (Oracle).

View
A regular view, is a database object that allows specific users to only see portions of a tables or rows. Queries from the view, will always go to the **base table**.

Materialized View

Stored query for later purposes. The data is stored in a separated physical table.

CREATE VIEW AS SELECT…

Synonym
Another name for a database object, Employee -> is also Worker.

Joins
Used to combine two or more **tables**, is a database technique used in SELECT statements.

Joins are normally performed comparing primary keys to foreign keys, however, they can be performed with any type of column, **as long as the types match**.

They can be performed in many ways:

- **INNER JOIN**.
    - The most commonly used type of join, returns rows **only** if the columns specified in the join clause match.
- **OUTER JOIN**.
    - The OUTER keyword can be used with LEFT, RIGHT or FULL keywords to obtain rows which some of the join columns are NULL.
    - However, in Oracle, this word is **optional**. LEFT, RIGHT or FULL will be automatically OUTER.
- **LEFT [**OUTER**] JOIN**.
    - Returns the matching rows plus the ones that where null in the first table.
- **RIGHT [**OUTER**] JOIN**.
    - Returns the matching rows plus the ones that where null on the second table.
- **FULL [**OUTER**] JOIN**.
    - Returns all rows from both tables specified including the ones which had null values on either side.
- **CROSS JOIN**.
    - Returns the cartesian product two or more tables.
- **SELF JOIN**.
    - An INNER JOIN performed matching two columns existing in the same table.
    - They represent hierarchies.

As you can see, all JOINS are INNER but with added things when specified.

To write a join:

- **SELECT** A.FIRSTNAME, C.NAME **FROM** STUDENT S **INNER JOIN** COURSE C **ON** S.COURSEID **=** C.ID;
- **SELECT** A.FIRSTNAME, C.NAME **FROM** STUDENT S, COURSE C **WHERE** S.COURSEID **=** C.ID;
- **SELECT** A.FIRSTNAME, C.NAME **FROM** STUDENT S, COURSE C **WHERE** S.COURSEID **=** C.ID(+);
    - **RIGHT OUTER JOIN**.

To optimize joins, put the tables to join from **more** data to **less** data, and then perform the joins **in order**.

Sub Query

SELECT statements that can be performed in specific places of any **DML** statement.

- **INSERT**.

    - In the **VALUES** clause as long as the **SELECT** returns only **one row**.
        - To ensure this, add **ROWNUM** = 1 in the **WHERE** clause.
- **UPDATE**.
    - In the **SET** clause as long as the **SELECT** statement returns **one row**.
    - In the **WHERE** clause (same **SELECT WHERE** rules apply).
- **DELETE**.
    - In the **WHERE** clause (same **SELECT WHERE** rules apply).
- **SELECT**.
    - In the **FROM** clause.
    - In the **WHERE** clause.
        - If the **SELECT** returns more than one row, use the keyword **IN** instead of the = operator.

By theory, **JOINS** are faster than **Sub Queries**, however, these last ones come in handy in many cases and there are some cases that they might perform better than a join.

Set Operators

UNION [ALL], INTERSECT, MINUS.

Normalization

Database optimization technique utilized to **reduce data redundancy**. It can be applied in **many levels**, however, up to the 3rd normal form is the best practice in any enterprise application, after that, it becomes very mathematical (more than 30 normal forms).

1NF

Atomic and **Singular**, “The Key”.

- To make a row singular, a primary key column(s) is(are) needed.
- To make it atomic, each column should be as **granular** as possible.
    - E.g.: Name column should be -> First Name, Last Name, Middle Name, Suffix.

2NF

No **partial** dependencies, “The whole key”.

- 1NF needs to be met.
- If there are no composite primary keys, you are automatically in 2NF.
- Columns can not be dependent of one part of the key.

3NF

No **transitive** dependencies, “Nothing but the key”.

- 2NF needs to be met.

- Columns can not be dependent of a non primary key column that is dependent of the actual primary key.

Transaction

Any amount of **DML** statements before a **COMMIT** statement is considered a transaction. After the **COMMIT** is done, the transaction should follow the **ACID** properties.

- **ATOMIC**.
    - "All or nothing", if any statement on the transaction fails, the whole transaction fails.
- **CONSISTENT**.
    - If the DB was in a consistent state before the transaction, it should be after it.
- **ISOLATED**.
    - One transaction shouldn't affect other transactions. It can be applied in different **levels**.
- **DURABLE**.
    - Persisted data should be everlasting.
    - Some RDBMS's have different approaches, even to recover from catastrophes (Oracle has special logs [Bitacora]).

Isolation Levels

These levels are applied in RDBMS to provide consistency and in some cases, to avoid phantom or dirty reads.

- **Serializable**.
    - Allowed in Oracle.
    - Read/Write locks.
    - Applies range locks even in the **WHERE** clauses of a select statement.
        - Phantom reads can't happen because of this.
        - Table that is being read can't be modified until the reading is done (no **INSERTS**, no **UPDATES**, no **DELETES**).
- **Repeatable Reads.**
    - Not very used.
    - Read/Write locks.
    - Doesn't provide range locks, that means phantom reads can happen.
        - Doesn't lock the whole **SELECT** statement, nor **INSERTS**, nor **UPDATES**, nor **DELETS**.
- **Read Committed.**
    - Oracle default.
    - Write only locks.
    - Only data that is committed will be seen by other transactions.

    - Dirty reads can’t happen, but Phantoms can.
        - This is why is recommended to not perform very long transactions.
- **Read Uncommitted.**
    - A disaster.
    - Dirty reads are normal, any transaction can see any uncommitted data.
    - Very inconsistent.

Phantom & Dirty

- **Phantom Read:** reading data that is being added or modified by a running transaction.
- **Dirty Read**: reading data that is uncommitted.

PL/SQL

Procedural Language is a complete programming language, which also allows SQL statements.

Besides the programming itself, Oracle offers an OOP aspect to its databases, providing certain types of objects, which some of them require PL/SQL code.

CREATE [OR REPLACE] and **DROP** statements are allowed for these objects. **ALTER** is not allowed.

Sequence

An object which holds a numeric number that starts from a certain point and it also contains a max. It increments by a specific amount every time **NEXTVAL** is called.

They can be combined with **Triggers** to auto increment primary key columns.

E.g.: **CREATE [**OR REPLACE**] SEQUENCE START WITH** 1 **INCREMENT BY** 1.

Trigger

Block of code that executes when a specific event happens. These events can be **INSERT**, **UPDATE** or **DELETE** statements, and they can happen **AFTER** or **BEFORE**.

E.g.:

- **CREATE [**OR REPLACE**] TRIGGER**
- **BEFORE INSTERT ON** TABLE_NAME
- **FOR EACH ROW**
- **BEGIN** (PL/SQL code) **END**;

Cursor

Pointers to a **<u>result set</u>**. They can be used to loop programmatically on the output of a **SELECT** statement (similar to **iterators** in Java).

Oracle provides **SYS_REFCURSOR**, its own type of **REFCURSOR**. This means you can create your own type of **REFCURSOR**. (TYPE SYS_REFCURSOR IS REF CURSOR)

Stored Procedure

PL/SQL code that can be executed in certain ways and has a certain amount of properties.

- They don't return anything.
- They may or may not contain **IN** (by value) and **OUT** (by reference) parameters.
- They allow any **DML** statements within.
  - These means **transactions** can be created in a stored procedure.
- Stored procedures can call other procedures and functions.
- Can **NOT** use stored procedures in **DML** statements.
  - **EXEC** STORED_PROCEDURE.

Function

Also known as **User Defined Functions**, are like stored procedures but have some other restrictions or abilities.

- They **must** return something.
  - Cursors are allowed.
  - It should be a single value.
- They may or may not contain **IN** parameters (by default).
- Only **SELECT** statements are allowed.
- Functions can only call other functions (no stored procedures).
- They can be used in any **DML** statement.
  - To call a function, you have to use a **DML** statement (you can't **EXEC**).
  - Use **FROM DUAL** if you are not selecting from a specific table.
    - DUAL is a dummy table which returns anything your throw at it.

Types

Similar to **classes**, they can be used as your own datatypes for columns. We are not going to use this, because this is a high level technique which doesn't follow the **normal forms**.

VARRAY

Arrays of any Oracle data type or your own or existing type, they can be used for columns.

Nested Table

Infinite arrays, they can be seen as tables in columns. They have to be of a specific data type.

JDBC

Java Database Connectivity, offers an API (just interfaces), that allows us to connect Java applications to any RDBMS that offers specific implementation to these **JDBC** interfaces.

To connect to a database

The following elements are needed.

1. **DriverManager**, the implementation of the JDBC interfaces (**OJDBC** for Oracle) and where we get the **Connection** object.
2. **Username**.
3. **Password**.
4. **URL**, endpoint to the database.

The main interfaces

1. **Connection**, used to execute statements and get database information.
2. **Statement**, regular SQL statement which can be hard to manage because single quotes need to be provided for String values.
    a. **SQL Injection** can happen.
3. **PreparedStatement**, Java precompiled and controlled type of **Statement** that provides more readability, maintainability and security (no SQL injections).
    a. Values are inserted with indexes.
    b. E.g.: "INSERT INTO TABLE_NAME VALUES(?,?,NULL)"
        i. statement.setInt(1, 35);
        ii. statement.setString (2, "Hello");
        iii. Yes, they start **indexing from 1**.
4. **CallableStatement**, used to call stored procedures. (E.g.: {call <procedure-name>[(<arg1>,<arg2>, ...)]}).

Extra

TNSnames.ora

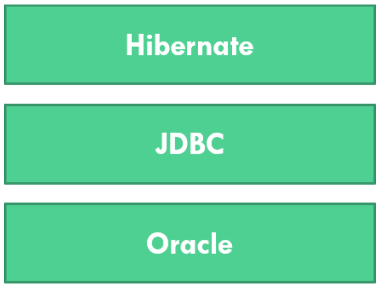
This is where the Services ID are stored (ORCL, XE), it can be changed to point to remote servers. (Local Naming Parameters).

CONNECT <USER_ID>@xe

Hibernate

- An Object Relational Mapping (ORM) framework.
- **ORM** provides an **object-oriented** representation of entities existing in a database (mapping).
- An abstraction of **JDBC**.

- Hibernate was created with the purpose of providing data base manipulation in an **object-oriented** way.



Object States

1. **Transient**, a new object. Not persisted, not in session.
2. **Persistent**, represented in DB, just loaded or saved, associated with session.
3. **Detached**, it has been persistent but its session has been closed. It can be associated with a new session in a later moment.

Annotations

Hibernate uses Java Persistence API (**JPA**) annotations to provide object relational mapping.

- **@Entity**. Tells Hibernate that a class is an entity that needs to be mapped. Hibernate will look for this annotations if the package was specified to be scanned.
- **@Table(name=”**TABLE_NAME**”)**. Used to provide a table name that will be or is being used in the database.
    - If not provided, Hibernate will use the name of the class as the name of the table.
- **@Id**. Used to defined that an attribute is a primary key.
- **@Column(name=**”COLUMN_NAME”, [unique = true/false], [nullable = true/false]**)**. Used to provide a column name that will be or is being used in the database.
    - If not provided, Hibernate will use the attribute name as the name of the column.
- **@GeneratedValue, @SequenceGenerator**. Used in conjunction to provide auto incrementing features with sequences.

Relations

- **@JoinColumn(name=”**COLUMN_NAME**”)**. Used to give a name to the join column that will be or is being represented as a **foreign key**.
- **@OneToOne** – Object object.
    - This one is the only one EAGERLY loaded and it makes sense.
- **@OneToMany** – List or Set.

- **@ManyToMany(mappedBy=**”javaFieldInAnotherEntity”**)**. If mappedBy is not provided in one of the entities and this annotation is being used as two tables, two junction tables will be created instead of one.
- **@ManyToOne**. mappedBy on his @OneToMany doppelganger if it’s used.
    - This one is EAGERLY done as well.

Fetching

In Hibernate, fetching can be done in **LAZY** (default) or **EAGER** form.

- **@OneToOne(**fetch = FetchType.**LAZY**)
    - Whenever the attribute is used, by a constructor, setter, getter, or any logic that uses the field, it will be fetched at that point.
- **@OneToOne(**fetch = FetchType.**EAGER**)
    - Whenever the object is instantiated the fetch will happen with it’s PK.
    - This is not recommended for big data sets.

Cascade

Hibernate supports JPA Cascade Types, which are used for transitive operations that should happen in multiple related tables. They are used in any relational annotation like: **@OneToOne(**cascade = CascadeType.**ALL**).

This operations can happen on **Hibernate sessions**.

- **DETACH** or **REFRESH**. refresh(entity). Re read the state of the given instance from the underlying database.
- **MERGE**. merge(entity). Copy the state of the given object onto the **persistent** object with the same **identifier**.
- **PERSIST**. persist(entity). Make a **transient** instance **persistent**.
- **REMOVE** or **DELETE**. delete(entity). Remove a **persistent** instance from the data store.
- **DELETE_ORPHAN**. Deprecated, use *@OneToOne(orphanRemoval=true)*.
- **SAVE_UPDATE**. saveOrUpdate(entity).
- **LOCK**. buildLockRequest(entity, lockOptions).
- **REPLICATE**. replicate(entity, replicationMode). Persist the state of the given detached instance, reusing the current identifier value.
- **ALL**. For all operations, this is what we normally use.

Hibernate has additional cascading types besides the JPA ones, the most important thing is to know that they all have their own use, however, **ALL** is what you normally should use.

ACID

Reiterate over Transaction properties and Isolation levels.

Queries

There are **three** ways to perform queries in **Hibernate**.

- **Criteria**.
  - createCriteria(Pojo.class)
  - .add(Restrictions.like(“fieldInJava”, value))
  - .add(Restrictions.eq(“fieldInJava”, value))
- **HQL**.
  - hql = “FROM com.revature.model.Customer WHERE id = :id”
  - createQuery(hql).
- **Native SQL**.
  - createSQLQuery(“query”).
  - createSQLQuery(“CALL NAME_PROCEDURE(:paremeter)”);

Hibernate was meant to use **Criteria** over any other type of query, however, **HQL** is still heavily used and in some cases, **NativeSQL** is needed.

Session Factory

The SessionFactory is an object **configured** in the Hibernate Configuration File, which contains the **necessary data to connect to a specific database**.

Once the connection is done, we can leverage the SessionFactory to obtain **session instances**, and perform database operations.

L Caching

- **L1** cache is associated with each **session**, and it’s available by **default**.
- **L2** cache is associated with the **SessionFactory**, and needs to be **configured** (if you want to use it).
  - **ehcache** (palindrome).
  - **Apache Ignite** is one library for this.

Exceptions

Hibernate exeptions are **RuntimeException**. They are meant to be handled by a framework (like Spring) and reduce the amount of code. Some common Hibernate exceptions.

- **LazyInitializationException**.
  - The session it’s closed and you try to access an attribute that is configured to be fetched lazily.
- **AnnotationException**: unknown id generator.
  - The configured sequence generator doesn’t exist in the database (you didn’t provide @SequenceGenerator after @GeneratedValue annotation).
- **QuerySyntaxException**: table is not mapped.

    - This usually happens when you have you don't let Hibernate generate your schema (which is a good thing). For example: the table name of your entity doesn't match the actual table in the database.
- **PersistentObjectException**: detached entity passed to persist.
    - You try to persist an entity with your own defined primary key, but you have it configured to be generated.

Interfaces

- SessionFactory
    - openSession().
- Session
    - beginTransaction().
    - getTransaction().
    - save()
    - saveOrUpdate()
- Criteria
    - add(Restrictions) for where
    - add(Projections) for aggregate
    - list()
- Query
    - setString()
    - setAnyKindOfType()
    - list()
- SQLQuery
    - setAnyKindOfType()
    - list()
- Transaction
    - commit()
    - rollback()

Configuration

Main

```
<session-factory>
	<property name="hibernate.connection.username">HIBERNATE_TEST
	 </property>
	<property name="hibernate.connection.password">p4ssw0rd
	</property>
	<property name="hibernate.connection.url">jdbc:oracle:thin:@RDS_LINK:1521:ORCL
	</property>
	<property
	name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver
	</property>
	<property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect
	</property>
```

```
        <!-- Show SQL Output -->
        <property name="hibernate.show_sql">true</property>

        <!-- Auto generate/update schema -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Validate schema (Used when you don't want hibernate to modify your schema)
        -->
        <!--property name="hibernate.hbm2ddl.auto">validate</property>-->

        <!-- Mapping -->
        <mapping class="com.revature.model.Pokemon"/>
        <mapping class="com.revature.model.PokemonTrainer"/>

        <!-- Configuration Mapping -->
        <mapping resource="pokemon-type.hbm.xml"/>
    </session-factory>
```

Mapping

```
<hibernate-mapping>
      <!-- Pokemon Type -->
      <class name="com.revature.model.PokemonType" table="POKEMON_TYPE">
            <id name="id" type="int" column="PT_ID">
                  <generator class="native" />
            </id>

            <property name="name" column="PT_NAME" type="string" />
      </class>
</hibernate-mapping>
```

Spring

Follow slides separately.