Contents

Definitions

Java Bean (serializable):

It is a standard. Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

- All properties private (use getters/setters)
- A public no-argument constructor
- Implements Serializable.

For ex: If a library wants to stream any object you pass into it, it knows it can because your object is serializable (assuming the lib requires your objects be proper JavaBeans).

Statement:

Is a complete unit of execution that ends with a semicolon (;).

Creating Executable Java App from Console

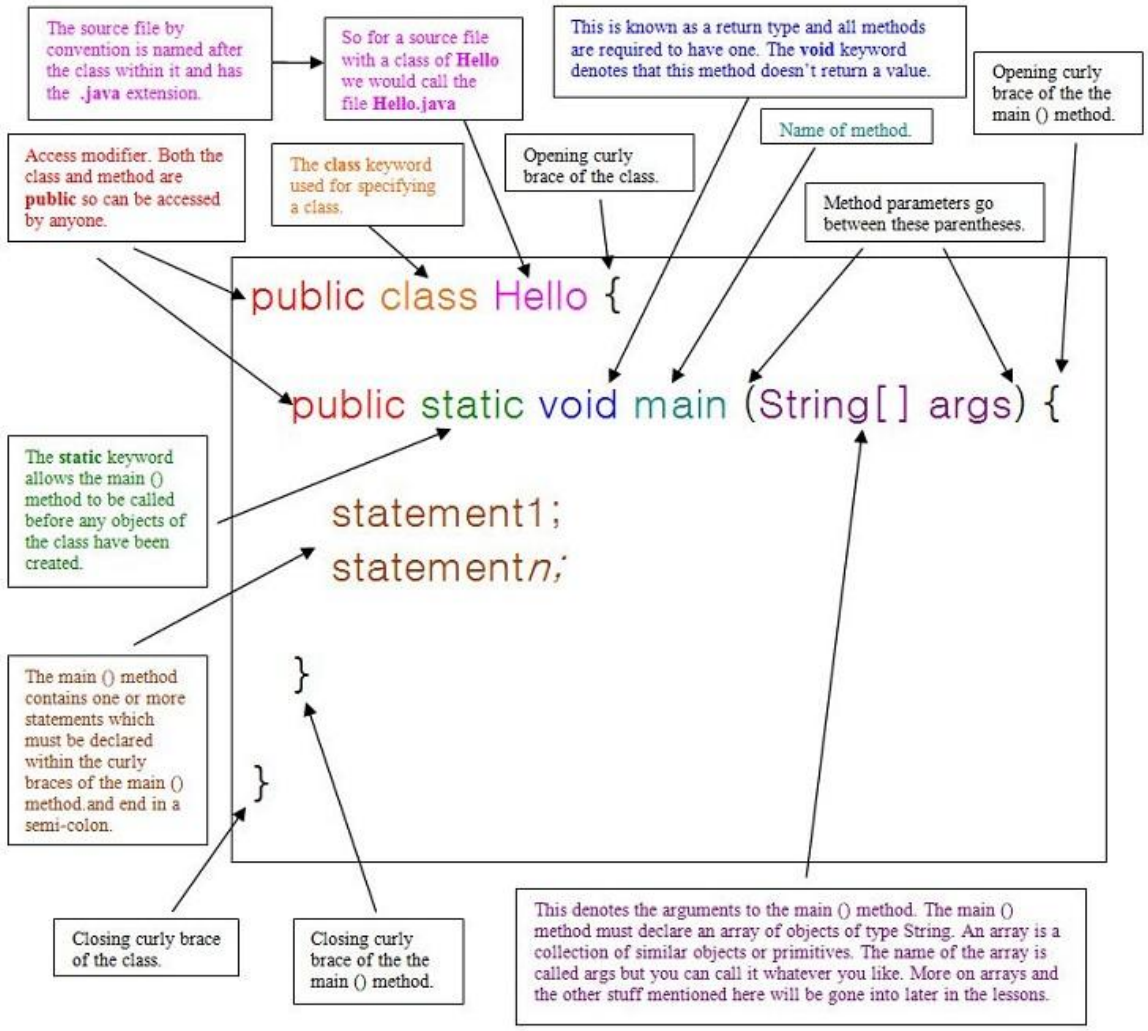
After defining the top level folder path;

```
C:\Users\Turo\Desktop>javac trg.java

C:\Users\Turo\Desktop>java createJavaApp
```

1 Java OO Concepts

1.a Java Class Structure



1.a.1 Access Modifiers (public, private, protected, default)

Public : It is open to all Java Universe (Can be used through importing the package).

Private: Can only be reached inside its own class.

Protected: Variables, methods, and constructors, which are declared protected in a super class can be accessed only by the subclasses in other package or any class within the same package.

Default: Is the case that no modifiers are specified. In this case, just the different packages does not have access. *Note: In difference on Protected, on another package it is accessible if used by a subclass.*

1.a.2. Java General Components and Benefits

- Encapsulation

Done with access modifiers.

- Inheritance

Using the structure and behavior of a super class in a subclass.

- Polymorphism

Changing the behavior of a super class in the subclass.

- Multi-Threading Support
- Cross Platform Support

Run once, use everywhere. Sandbox, virtual java environment, JVM .

- Strongly Typed

Java is a statically typed language, that requires to define the variable types before use. This provides a more solid development for big teams and structures.

- Built in Security

Java runs in JVM Sandbox, so it is secure to the platform that it is being run.

- Automatic Garbage Collection
- Distributed System Support
- Strong Library

Java has many tested, secured helpful libraries.

1.a.3. Importing Packages

The package imports are to save keystrokes, because the classes can be reached through packageName.className already. The important points are as above:

- java.lang.* is added by default, and does not need to be added.
- The "*" is used to include all classes under the related package. But it will not include the sub-packages or directories.
- Ambiguity because of that same class is declared in two different packages "*package*" and "*otherpackage*" :
    - *package.className* **dominates** *otherpackage.** .
    - *package.** with *otherpackage.** will cause **ambiguity** .

- "import static ..." is used to make static imports.

1.1. Nested Classes

A nested class is the one that defined inside another class.

1.1.1. Static Nested Class

Static defined nested classes are able to be used without instantiating the outer class, cleanly.

- Not able to reach the other elements of outer class.
- Can be accessed as :
  - *OuterClass.StaticNestedClass*
- Can be instantiated as :
  - OuterClass.StaticNestedClass nestedObject
    = new OuterClass.StaticNestedClass()

1.1.2. Inner Class

This class is able to reach its outer class. But cannot be used without instantiating the outer class.

- Reachable from the other elements of outer class.
- Can be instantiated as :
  - **OuterClass.InnerClass innerObject**
    = outerClassObject. new InnerClass()
    So it depends to the formerly instantiated outerClassObject.

1.1.2.1. Local Class

Local classes are the ones that defined inside a block. Same rules valid with Inner Class except; having modifiers.

1.1.2.2. Anonymous Class

An anonymous class is an expression that placed with the class declaration by extending a class or implementing an interface.

- Can access to the members of enclosing class.
- Can access the *final* defined local variables (method) of enclosing scope.
- Cannot access the non-final definitions of local variables (method).
- Can also have static members that defined constant.
- Cannot have constructor.

1.1.2.3. Lambda Expressions (Comparisons of Approaches)

If there are many different blocks of codes that are same in code at all, but had to be varied just for a part, than you can use the lambdas to make a generic block that the different part of the code is given from outside as a *lambda expression*.

***A simple method that gives a simple result, may be needed to vary due to the further requirements. If the similar methods will appear, the need to create new classes, new interfaces, depending on situation, creating local classes and anonymous classes may be necessary. This part will be progressed as telling the use cases of inner class types and their possible further effects.*

***Ref: http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html*

1- If this approach will be in traditional way, there may be an *Interface* that provide to vary the similar methods under one generalized ruler. So the new variations of similar methods, can be created by implementing this common interface, again and again.
2- On the traditional way, the code size will be bigger. There will be an *Interface* that have to be handled and classes that implementing it. Instead this approach, the Anonymous class expression can be used. This will provide to directly put the required variation of method; inside the function call. So there will be no need to create an interface and provide to do all similar works in same method, instead new methods.

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

1.2. Interfaces, Abstract Classes and Inheritance

The table shows extend and implement abilities of interfaces and classes.

	Interface	Abstract Class	Class
Extend	N Interfaces	1 (Abstract) Class	1 (Abstract) Class
Implement		N Interfaces	N Interfaces

Table : Extend & Implement Capabilities

1.2.1. Interfaces

As default (and does not need to be defined as);

ALL the interface methods are :

public abstract **IF THEY ARE NOT default or static**

ALL The interface variables are :

public static final

- Interfaces can extend N interface
- Interfaces cannot implement
- **Object** is a class, and **interfaces can** not extend classes.
  - Interface is-not-a Object.
- Interfaces can have concrete methods (Java 8)
  - Can be marked as default or static
  - **They still are public**
  - Default method can be used without or with overriding method.
  - Default methods can be used as : InterfaceName.super.method() or method().
  - Default methods can only be inside an interface.
  - Static method cannot be overridden.
  - Static method ONLY can be used with the interfaceName.method() .
  - Static methods HAS TO HAVE the METHOD BODY .

1.2.4. Abstract Classes

- An abstract method inside abstract class have to be marked as abstract.
- Abstract classes cannot be initialized.
- Can have both abstract and non-abstract methods.
- Can extend a class and can import N interface.
- They do not have to implement the inherited methods, but the first concrete class do.
- Abstract methods CANNOT be **final, static, private** .

1.2.3. Inheritance

1.2.1.1. Overriding

- **When a method in sub-class matching function name and parameters, that means that method have to be satisfying the overriding rules.**
- Name and parameters have to be same, return type can be same or a sub-type.
- Overriding method can have less restrictive access modifier.
- Overriding method can throw a narrower or fewer checked exception.
- Final, Static or Private methods cannot be overridden. If static and private methods are written in the subclass, they will be out of polymorphism, as individual methods (hiding method). To intent to write the **final** method in subclass, this will cause a **compilation error**.
- The rest of the modifiers are legal to be used on the overriding method.
- Super's method can be reached with super(). But cannot use : super.super
- If the overriding method does not handling the exception that the super-type method has, in this case while calling this method from the super-type reference, it acts as throwing the exception. It has to be handled.
- In case that there is an overridden method of a super type, and this sub-type is defined by a super type reference, calling this method from this reference will call the overridden sub-type's method by JVM.

1.3. Constructors and Initialization Blocks

The Compiling Order:

- Static Initialization Blocks (Appearance Order)
  - With Super class, Sub class order, all of classes.
- From Top to Down of Inheritance Tree :
  - Instance Initialization Blocks (Appearance Order) & Constructor
    - This couple is compiled for each class in inheritance tree.

1.3.1. Constructors

- New Object creation will invoke all the constructors of the inheritance tree
- Constructors can use all access modifiers.
- Constructors cannot have a return type.
- Interfaces does not have constructors, Abstract classes has.
- If there is an overloaded constructor is created, the no argument default constructor will not be active anymore, it has to be created by hand.
- Constructors can be called just from another constructor with this() method.
  - Only from their first line.
- Default Constructor:
  - Is created when no constructors are written.
  - Has the same access level with the class.
  - Has No arguments.

    - Includes a super() call.
- Sub class constructor has to satisfy the super class constructor parameters.
- Constructors cannot be overridden.
- this() keyword is used to invoke a self constructor.
- The first line of the constructor has to be this() or super(). The compiler decides it. If the first line has this() call, than compiler does not place the "super()" to that constructor.
- In case that super() is not called yet, the constructor can call just the static methods. This case occurs just when a method called inside this(method()).

1.3.2. Initialization Blocks

1.3.2.1 Static Initialization Blocks

- Is run when the class is first loaded, one time (that the class or it's son is instantiated). All the inheritance tree is triggered.
- Runs in order that appear in the code.
- Static - Class , Non Static - Instance
- Ps.: Static variables has the same initial values with the instance variables.
- If there is a same static method in the sub class, that method does not acts in polymorphic rules, is just another independent method from the super class version.

1.3.2.2 Instance Initialization Blocks

- Runs each time an instance is created.
- Runs right after constructor call to super() ends, before the constructor code.
    - This means that the super class first.
- Runs in the order that appear in the code.
- Often used as a place to put code that all class constructor versions should share.

1.4. Assignments

1.4.1. Stack and Heap

- Instance variables and objects lives on heap.
- Local variables and methods lives on stack.

1.4.2. Primitives Data Types and Literals

Byte - 8 bit (-128..127)
Short - 16 bit (-32768..32768)
Integer - 32 bit
-Decimal Literals (Base 10): Common integers

-Octal Literals(21)(Base 8) : Representing octal form by placing "0" first
-Hexadecimal Literals(16)(Base 16): With prefix "0x"
"0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F"
-Binary Literals (Base 2) : With prefix "0B" or "0b"

Float - 32 bit There has to be "f" or "F" in the end of expression.
Double - 64 bit May or may not have "D" in the end
Long - 64 bit All four integer literals (binary, octal, decimal, and hexadecimal) are defined as int by default, but they may also be specified as long by placing a suffix of "L" or "l" after the number
Character - 16 bit (0..65535) Can be initialized as "\u004E" 4 digits after '\u'
Boolean True or False (Cannot be 0 or 1)

1.4.2.1 Rules for Primitives (and Casting)

- For an object, it is legal to cast just from a same sub-type derived super-type to that sub-type.
- Underscores cannot be in the beginning or end of the literal.
- While assigning from a wider type to a narrower type it needs casting.
- While assigning from a narrower type to a wider type it casts implicitly(auto).
- Result of an arithmetic operation between two primitive :
  - Different types : Narrower type is converted to the Wider
  - Short - Byte - Char always converted to int
- Component assignment operators, += , -= , *= ...etc. all have an implicit cast.
- Every kind of primitive has it's own wrapper class : *Boolean, Byte, Character, Double, Float, Short, Long, Integer* that has their own functions.

1.4.3. Variable Assignment

- Assigning a primitive to another primitive "int a = b" does not make them point same objects. They just have the same values, on different parts of the memory.
- Passing a primitive value, makes just the value of it to pass.
- Assigning an object reference to another object reference makes them point the same object in the memory.
- Passing an object reference , the passed reference variable will point the same object in the memory. The changes to this object also will affect the object that passed. But the reference changes will not change the original object reference. If a change is applied, the reference variable will act independent from the passed reference variable.
- String Objects are exceptional (String Pool). For any changes that occurs in the passed function, so the original reference will not get affected.

1.4.4. Initialization

- Local variables (Object or Primitive) have to be initialized, or compile error will occur while trying to use them.
- The uninitialized **local** object variable does not equal to null. It is compile error to try to use the uninitialized local variable.
- Instance variables does not have to be initialized, they will have the default values in case not initialized.
- The uninitialized **instance** object **is null**.

1.4.5. Garbage Collection

- The heap is that part of memory where Java objects live, and this part is the only place where the JVMs garbage collector will work.
- The exact moment that an Object will be collected, is unknown. We can just know when they are eligible to be collected.
- An Object is eligible for garbage collection when no live thread can access .
- The out of scope objects and pointed to "null" are expected to be eligible if they are not pointed by another outer reference variable.
- If there is an object created without reference variable, this object is eligible for garbage collection just after created (new Animal().go()) .
- If the object reference, is referred by some outer resource, that object is not eligible for GC even if reference is "null" or code scope ends.
- Finalize() is the method that is called before the object is collected, and runs once.

1.5. Operators

1.5.1. Equality

1.5.1.1. Object.equals()

- For primitives, it is value check.
- For objects, it checks whether they are pointing to same object or not.
- (x=o) results o, (x = true) results true, as the value on the left side of the operand.
- The Object.equals() for **String** , **Enum** , **Wrappers** and **ArrayList** is to check if their values are same, not the referred objects.
- For String , the values compared case sensitive.
- StringBuilder does not override Object's .equals() function.

1.5.1.2. Operator "=="

- For Primitives, Enums: it is always value check .
- Classes, it is always memory check just :
- Wrappers and Strings :
    - If value is assigned directly with "=" , it is a value check .

1.5.2. Instanceof

- Used as (reference instanceof Object)
- It can be used between the types that are in same inheritance tree:
    - class A {} class B extends A{} class C extends A{}
    - Can be used to see which sub-type is the reference.
- Cannot be used un-related types, or; compilation fails.
- The **uninstantiated** Array references are just the instanceof Object, nothing more.

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

First Operand (Reference Being Tested)	instanceof Operand (Type We're Comparing the Reference Against)	Result
null	Any class or interface type	false
Foo instance	Foo, Bar, Face, Object	true
Bar instance	Bar, Face, Object	true
Bar instance	Foo	false
Foo [ ]	Foo, Bar, Face	compiler error
Foo [ ]	Object	true
Foo [1]	Foo, Bar, Face, Object	true

1.5.3. Logical Operators (&, |) (Non-Short Circuit)

Unlike the short circuit (&&, ||) , regardless of that the first expression, the second expression is also executed.

1.5.4. Short Circuit Logical Operators (&&, ||)

Here first looked to the left side and if the condition is provided, the second expression is not executed.

For && operation, if the first expression results false,

For || operation, if the first expression results true,

The second expression will not be executed.

Note: The operator ^ (XOR) is true only in case one true one false.

Note: integer + string = string , integer + integer + string = totalinteger + string

int b = 2;

String str = "" + b + 3 ; //prints 23 because it is left-to-right.

1.5.5. Operator Precedence

UMARELSA

Unary Operators (++, --, !)

Multiplication, Modulation, Division (* , / , %)

Addition, Subtraction (+ , -)

Relational (> , >= , <= , <)

Equality (== . !=)

Logical Operators (& , |)

Short-Circuit (&& , ||) ***

Assignment (+= , -= , *= , /=)

***** Special Case ! If left of this operators (&& , ||) meets with condition already, JVM will not compile the right side .**

1.6. Flow Controls

1.6.1. If-Else Statement

- If there is an else that did not specified with parentheses, that else is belong to the innermost if .

if (...)
- if (...)
- else

if (...)

1.6.2. Switch Statement

- The parameter that will be used in expression have to be pre-defined.
- A switch expression must be evaluate to a char, byte, short, int, enum, String.

    - Switch does not take BOOLEAN, float, double, long etc..
- The cases of switch have to be in the bounds of the evaluated type.
- The cases have to be unique, cannot have twice with same case.
- After the matching case, if there is no break, all the cases will work without looking the cases until a break, even the default case.

1.6.3. While and Do-While Statement

- The parameter that will be used in expression have to be pre-defined.
    - On do-while it also is valid.
- In Do-While, there has to be a semicolon in the end "while(...);"

1.6.4. For Loop

- Three statements are not obligatory to exist.
- First statement
    - Have to be declaring same type of variables as:
    - (int x=0,y=1) is legal. (int x=0, int y=1) illegal.
    - Can define variable or use the pre-defined variable
- Second statement need to be a Boolean expression.
- Third statement can call function etc. but cannot define new parameter.

1.6.5. Enhanced For Loop

- There are two statements that separated with " : " .
- Iterates over each Type : Type Array[] (for declaration : expression)
- Has to define the parameter inside statement (int x : xArray)

1.6.6. Break and Continue

- Break goes out of the innermost loop
- Continue jumps to the next iteration of the innermost loop

1.6.7. Labeled Break and Continue

- The label has to be just before a loop.
- Only difference between normal ones, is that effects the pointed loop.(Unlike C++ goto!!)
- Break makes that loop to end, and Continue makes that loop to step next iteration.

1.7. Exceptions

1.7.1. The Rules of Exceptions

- It is necessary and sufficient to use try with finally() or catch().
- Try, Catch and Finally have to following immediately
- More specific exceptions have to be catched before more general ones.
- Finally will always run.

- Exception, Error, RuntimeException... anything that is extending Throwable is able to be thrown and be catched.
- Throwing an exception that is not catched will cause exit from the method directly.
- Throwing an exception from final block will suppress the rest of the exceptions that attempted to be thrown in try or catch blocks.
- RuntimeException = UnChecked Exception:
- Checked Exceptions:
    - Have to be handled with try catch OR throws definition at including method
    - Common Checked exceptions are :,
        - IOException
        - FileNotFoundException,
        - MyException extends Exception

- The wider Exception will be enough for the derived ones.
- Inheritance rules that "cannot define wider or extra Exception in sub class", is valid for just checked exceptions.

1.7.2. Common Exceptions

1.7.2.1. Runtime Exceptions (java.lang)

ArithmeticException : (JVM) Occurs when dividing to zero.

IllegalArgumentException: (Programmer)

NumberingFormatException(Son of IllegalArgumentException)

ArrayIndexOutOfBoundsException : (JVM)

NullPointerException: (JVM)

ClassCastException(JVM)

1.7.2.2. Checked Exceptions (java.io)

FileNotFoundException

IOExecption

extends Exception

1.7.2.3. Errors (java.lang.Error)

StackOverFlowError:(JVM) Thrown when a method is called itself many times.

ExceptionInInitializerError: (JVM) Occurs when an exception thrown in static block.

1.8. String , StringBuilder, Calendar and Wrappers

1.8.1. Strings

- Strings are Immutable; that a newly created string, will place to the string pool and not ever change.
  - String str = new String("asd") : Creates a separate String object in the memory and not in the String pool.
  - this **new String("asd") comparison with "==" will compare the memory.**
  - String str = "asd" : Creates a string in string pool and points str reference to it
  - this String str comparison with "==" is will compare the value.
  - **A final String's value cannot be changed, but a final StringBuilder can.**
- Immutable : For the new String reference, compiler checks the *string pool* if the value is already exists to refer to it. No change can be done on any strings inside the pool
- Any String method will not change the string reference's value if it is not assigned to itself again.
- Each character in string is a 16-bit Unicode Character.
- "+" Operator works for appending to the end.

1.8.1.1. Important String Methods

String concat() : Adds string to the end.

Char charAt(): Gets the char at given index.

Boolean equalsIgnoreCase(): Compares string non-case sensible.

String trim(): Removes the white spaces in the beginning and in the end.

String toLowerCase() , toUpperCase()

int length(): Gives the length of string.

String replace(): Replaces all existing given character with another.

String substring(): From index , to index-1

**substring(6,5) will give an EXCEPTION on runtime

1.8.2. StringBuilder

- StringBuilder is Mutable. The changes on the value will be changing just that value without creating a new one
- StringBuilder is useful when making many changes on same string to avoid memory waste.

- StringBuilder methods does not have to be assigned to a new variable. It effects directly. Ex: sb.insert("asd")
- Capacity of StringBuilder object :
    - If created default , it is 16 chars : new StringBuilder()
    - Size can be defined : new StringBuilder(30)
    - It will grove with the append() or insert() methods.
- "+" operator **does not** work with StringBuilder object
- append(String s) adds the string to the end
- insert(index,value), inserts the value beginning from index.(String or other literals). Can be starting from exact end of length.
- delete(index,index) removes from index to index-1. (like substring)
- reverse() reverses the string.
- replace(from index, to index, string)
- length()
- substring()

Note: The Chained Methods (result = method1().method2().method3();) will run the ladder method with the former method's return.

1.8.3. Calendar

- Elements are IMMUTABLE . Except period.
- Created with Factory Classes, that creating **without new** keyword, instead, with instantiating methods like LocalDateTime.of(...).
    - ~~LocalDateTime time = new LocalDateTime()~~ **Wont Compile**.

java.time.LocalDateTime : Year, Month, Day, Hour, Minute, Seconds, fractions of sc.

java.time.LocalDate: Year, Month, Day

java.time.LocalTime: Hour, Minute, Seconds, fractions of seconds.

java.time.formatter.DateTimeFormatter:

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("MM. dd. \" YYYY \"");

java.time.period : Period.between(LocalDate, LocalDate)

1.8.4. Wrapper Classes

- Each primitive have its own wrapper class.
- **Assigning a value** :
    - Direct Assignment ... reference = number;
        - If the number is in a correct form of reference type, no problem.
            - Correct form including the characters in the end of num.

        - If not, it has to be casted to it's primitive.
    - Instantiate Assignment - Constructor()
        - All instead Character; accepts the "string" value.
          Numeric Values:
        - Double, Long, Integer, Float accepts numeric values.
        - Short and Byte needs casting for numeric values.
        - Character does not accept numeric values.
- For equality test:
    - Integer x = 5 , Integer y = 5
        - x.equals(y) -> true
        - x == y -> true
    - Integer x = new Integer (5) , Integer y = new Integer(5)
        - x.equals(y) -> true
        - x == y -> false

2 Concurrency

2.1. Threads and Objects

**See BasicThreadCreation on Eclipse.*

2.1.1. Object.Wait

2.1.2. Object.Notify

2.1.3. Object.NotifyAll

2.1.4. Thread.sleep

2.4.5. Thread.join

Join is used to make current thread to wait the joining thread to finish its job.

2.4.6. CountDownLatch

Functions : *countdown()*, *await()*, *getCount()*

It is a synchronizer. The thread where the .await() is positioned, will be waiting for that Latch to complete the amount that specified when it instantiated like :

CountDownLatch ctl = new CountDownLatch(5);

2.4.7. CyclicBarrier

Functions: *await()*

It is a number watch to make all threads wait in a specific point of code(cyb.await()) until the specified amount of threads come until this point, and then make all continue (in a random priority as JVM acted in other thread queues)

2.2. Synchronization

Synchronization is used to solve data inconsistency problem.

2.2.1 Synchronization At class level

What happens when a static synchronized method is invoked, is; since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class

2.2.2 Synchronization At instance level

Synchronized keyword can be applied to a method or block. It creates a critical section, means that only one thread can access this block at a time, uses, then releases it.

When there are two or more threads are changing the same instance's value, there may be same time processes and this same time process will cause the two threads read same value , but not aware of each other's change.

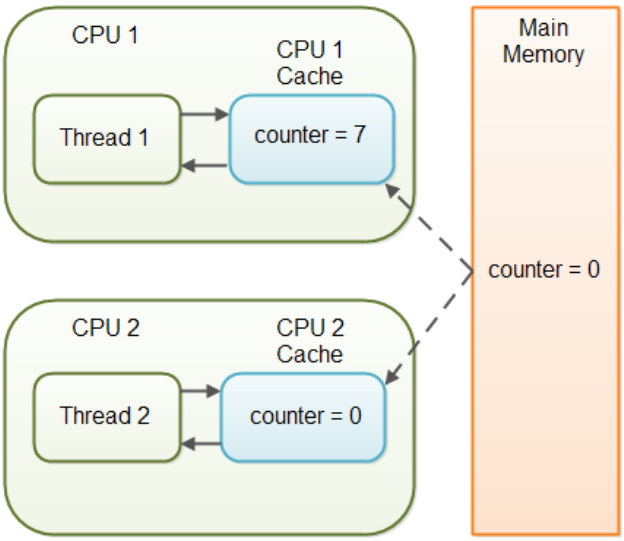
Every object in java has monitor log (mutex) - intrinsic lock. This lock will be able to taken by one thread in a time that it will release the lock after finishes its work.

Volatile is making an object to be seen to all of the threads.

Synchronized, makes this available also.

2.2.3. Volatile

Volatile keyword is able to be applied only to Instance Variables. It avoids the value of a variable to be corrupted by changed with two threads at the same time.



Volatile

2.3. Liveness

2.3.1. Deadlock

2.3.2. Starvation and Livelock

2.4. Guarded Blocks

2.5. Immutable Objects

2.6. High Level Concurrency

4 Collections

4.1. Comparisons of Collections

4.1.2. Differences between Linked List and Array List.

- Array List get (*Recuperar*) data. Because no need to walk through list, getting it directly.
- Linked List add/remove (*Añadir Borrar*) data. No need to do bit shifting since there is two side link.

- Rendimiento esta major cuando hacienda cambios en una Linked List. Pero es mejor para almacenar y recuperar los datos desde Array List.

4.1.3. Differences between Lists and Sets

- A Set has unique items, but a List can have many same records.

4.1.4. Differences between Queues and Stacks

- Queue – FIFO, Stack – LIFO

4.1.5. How HashMaps / Hashtables work (hashing, collisions, Object#hashcode)

- HashTable is synchronized and if no need to be synchronized, HashMap is a better choice for performance, and null can be used.
- It is better to create the hashmap with an initial size to avoid re hashing in each size exceed for a better performance.

4.2. Array and ArrayList

4.1.1. Arrays

- Has the attribute length.
- An array does not override equals() and so uses object equality.
- The array is another object that is created distinctly from its member objects.
- Size has to be defined on initialization. On definition no size can be given.

  int [] intArray= new int[5]

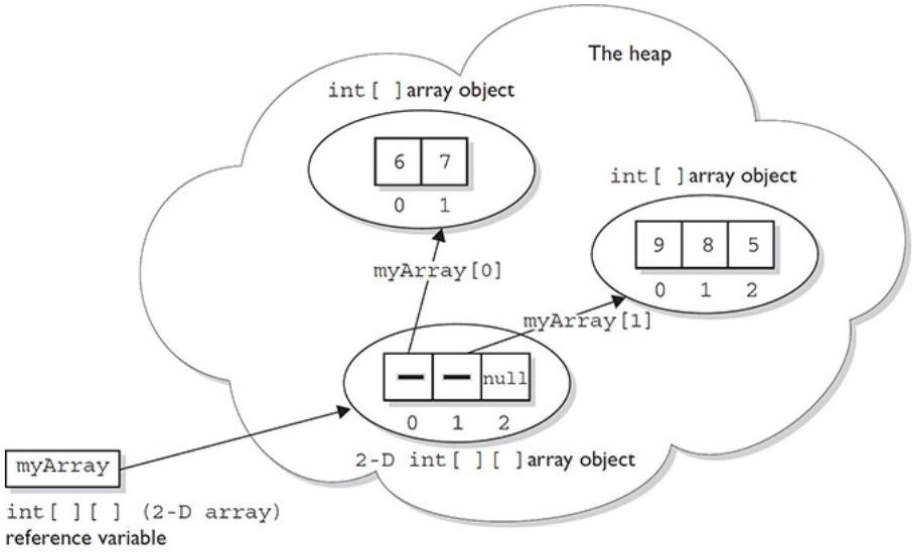
  Now we have an array with default integer values.

- Initiating an array with an Object type, does not mean that the objects are initiated . All will have the initial values.
- Inside of a method, unlike defining normally, the objects or literals that inside an instantiated array are able to be used.
- Can be converted to ArrayList with Arrays.asList(theArrayList)
- What can we put inside the definition of an Array ?
    - Pre-defined objects or the objects that defined inside parenthesis.

      Dog x = new Dog("Chopin")

      Dog [] dogs = {x, new Dog("Beethoven")}

Multi-Dimensional Arrays



Picture demonstrates the result of the following code:

```
int [ ] [ ] myArray = new int[3] [ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

FIGURE 6-5 A two-dimensional array on the heap

4.1.2 ArrayList

- Has the method .size() to get the int size of it.
- Has the method theArrayList.toArray(theArray) to convert to Array.
- Collections.sort(theArrayList) can be used to sort many types as:
  - All wrapper classes
  - Date
  - String
- Can be defined as :
  - ArrayList list = **new** ArrayList();
  - ArrayList<Object> list1 = **new** ArrayList<>();
  - List list3 = **new** ArrayList();
  - List list4 = **new** ArrayList<>();
  - List list5 = **new** ArrayList<String>();
- Can't be defined as :

- ArrayList<> list = new ArrayList<>();

- ArrayList overrides equals() and defines it as the same elements in the same order.