Lec1

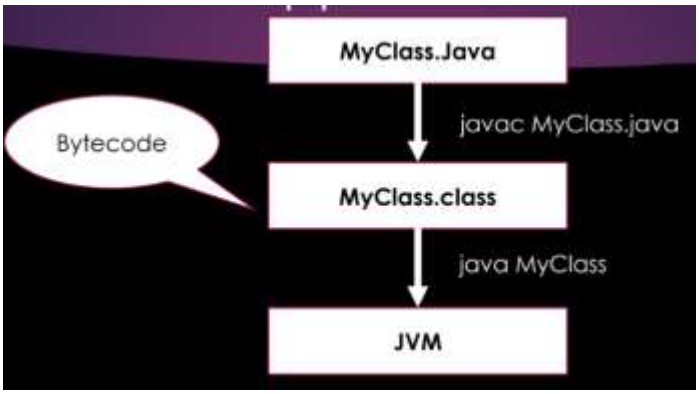
JVM:

- Runs java bytecode.
- Doesn't understand java [*.java], understand inly bytecode [*.class].

JRE:

- Doesn't contain tools and utilities such as compiler and debuggers.

Properties of Java:



- Object-oriented.
- Interpreted.
- Portable.
- Secure and robust.
- Multi-threaded.
- Garbage-collected.
- Has no support of multiple inheritance.

> If you're using command line to run your java program
  - Compile it using: javac MyClass.java
  - Run it using: java MyClass

MyClass >> class name

Garbage collection:

Deallocation >> removing the objects that are no longer referenced from memory

- Memory is dynamically allocated in java.
- Deallocation is the garbage collector's responsibility in java/C# which theoretically make no memory leaks
  Unlike in C/C++ deallocation is the programmer's responsibility which leads to memory leaks.
- When is the object is eligible for garbage collection?
  - A reference to it is set to NULL.
  - The reference to the object is made to refer to another object.
- How to force garbage collector to work?
  - System.gc

OOP principles:

- **Encapsulation:**
  - The process of combining data and methods into a single unite called class.
  - Keeps data safe from outside interference, allowing to expose only necessary data.
  - Can be controlled via access modifiers.
- **Abstraction:**
  - Is to hide the complexity of a program by hiding unnecessary detail from the user.
- **Inheritance:**
  - Allows code reusability.
  - Allows creating hierarchy of related classes.
- **Polymorphism:**
  - Means many forms
  - same concept can have different meanings in different contexts.
  - Has two forms, overloading and overriding.

Overloading > within the same class.

Overriding > in the child classes [overriding a method exists in the parent class].(related to inheritance)

Lec2

Branching: switch:

- switch statement tests expressions based on discrete values, i.e., integer, enum, or String.

- It cannot be used for testing expressions based on ranges, e.g., float values between 1.00 and 2.00
- Each case expression should be terminated by a break, otherwise the control flow across several cases

Break after the default section is not mandatory

> It's preferred to limit the number of break/continue statement to 1 per loop.

Loops: foreach:

- is used to traverse an array or a collection in Java
- has no loop variable.

```
int arr[] = {1,2,3,4,5};
for(int value: arr){
    //this means that "value" will contain a different member
    //in the array "arr" every iteration
    //first iteration -> value = 1
    //second iteration -> value = 2 and so on.
    System.out.println(value);
}
```

```
"C:\Progr
1
2
3
4
5
```

الحلو في ال foreach ان هي هت-iteriate على ال container بتاعي من غير ما

اعرف ال size بتاعه كام عادي, ف بتشيل مشكلة اني اعمل access ل حاجة

برا ال boundaries بتاعة ال collection.

Arrays:

- Has fixed size.
- Can hold any type including simple and complex data types.
- Only holds references -> doesn't holds the actual objects.

- If any position is not initialized, it is NULL.

simple زي ال int-float-long-boolean...etc

complex زي ال student -> user-defined objects

```
int []arr;
//can't print "arr" since it's not pointing at anything
//arr itself is NULL
System.out.println(arr);
//it's a compilation error

int []arr2 = new int[5];
//gonna print the reference that arr2 is pointing at
System.out.println(arr2);
//gonna print 0 since i didn't initialize the values in the array.
System.out.println(arr2[2]);
```

Classes and objects:

Constitutes > يشكل

- A class constitutes the blueprint of a specific type.
- A class defines various levels of hiding to protect its own fields and method.
- A class may contain inner classes.
- An object is an instance of a specific class.
- An **object reserve memory** in the system.
- Can be instantiated using the keyword new, **if not instantiated it will be NULL**

```
User user;
//the "user" object hasn't been instantiated ->
//it's not pointing at anything in the memory
System.out.println(user);
//this is compilation error
```

اي حاجة مش مدياها القيمة بتاعتها ايه و انا بعملها declaration

;int x = 2

او مكنش في جمبها new

;int x =new int

ف دا معناه ان هي اصلا مش موجوده و ال compiler مش عارف ايه دي

و مش واخده مكان من ال memory.

Constructors:

- Each class MUST have at least one constructor.
- Can be many constructors in the same class -> overloading
- Have NO return type.
- MUST have the same class name.
- Constructors are not inherited.
- If none is defined -> compiler creates one with no parameters called default constructor.
  - Default constructor:
    - Initialize fields with their default values -> zero for numeric types, and false for booleans , and null for object references
- Calls the constructor of the parent class implicitly.

- We use “super();” to call the constructor of the parent class in the child classes.

مش لازم اعمل call لل default constructor بتاع ال parent بس, عادي ممكن اعمل call ل اي واحد.

انا لو عندي class و عملت فيه parameterized constructor ف كدا ال default constructor ال كان بيعمله ال compiler اتغلى

ولو عايزه استخدمه ف لازم هعمل واحد ب ايدي معندوش parameters.

و بنفس المنطق انا لو عندي default constructor عملاه ب ايدي في ال parent class هيتعمله call تلقائي في ال child constructors.

معناه ان اي حاجة موجوده في ال default constructor ال انا عملاه دا هتتنفذ قبل اي حاجة في ال child constructor.

UML -> lab 3

https://drive.google.com/file/d/1h1Bg_p3gg0AVyA6KrCdPwE6hGW53eAQd/view

Unified Modeling Language (UML):

- Object-oriented modeling language.
- It’s not bound to a specific language i.e. not necessary Java.

Represented in UML by a rectangle, usually divided into three sections

1. Class name
2. Attributes (fields)
3. Operations (methods)

Visibility	Symbol
Public	+
Private	-
Protected	#
Default	~

- **UML relations:**
  - A relationship is represented as lines with arrows.
  - Different arrowheads have different meanings.
  - Example relationship types are <u>inheritance</u> and <u>association.</u>
  - Inheritance represents a hierarchy between classes
  - Association represents relationships between object.

Packages:

- Is a container of related classes.
- The package name normally looks like -> domain.subdomain.subsubdomain.....

Access modifiers:

- **For classes:**
  - Allowed public or none -> no private or protected [except for inner classes].
- **For fields:**
  - Field declaration has the form:

    Access-modifier <static><final> datatype fieldname;

  - static means that it is for the entire class and not for a specific object.
  - final means that its value CAN NEVER BE CHANGED (constant)

Most Restrictive ← → Least Restrictive

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Same rules apply for inner classes too, they are also treated as outer class properties

---

لما نعمل static field دا معناه انها هتكون نفس ال value بالنسبة ل كل ال objects ال عندي من الtype دا, ف بدل ما اعمل variable ل كل obj و هو كدا كدا هيشيل نفس القيمة و ياخد مساحة من ال memory كل مره ا-create obj فيها ف بنعمل ال static field و ياخد مساحة مره واحده بس و اقدر اعمله access ب اسم اي obj او حتى ب اسم ال class نفسه.

Lec3

Methods:

- Method declaration has **two parts**:
  - **Method header (signature)**
    - Defines method name, return type and parameter list.
    - Has the following structure:

      <Modifier> ReturnType MethodName(ParamList)

    - Type of modifiers:
      - Static -> makes the method for the whole class and not for a specific object.
      - **Static method can access ONLY static fields.**
      - **Implementation of static methods must be provided when defined.**
      - Main method MUST be declared static.

```
1 usage
static int staticVariable = 10;
1 usage
int nonStaticVariable = 10;
no usages
static void staticMethod(){
    //static method can access only static fields.
    System.out.println(staticVariable);
    //can't access non static field -> compilation error
    System.out.println(nonStaticVariable);
}
```

We can call static methods via class name same as static fields.

      - Final -> makes the method not able to be overridden in child classes (prevents inheritance).
      - Abstract -> **abstract methods MUST be overridden**.
      - **Abstract method has NO body** in parent class.

ال abstract methods لاوم يتعملها override في ال child class و لو متعملهاش ف دا معناه ان ال child class دا اصلا abstract class

  -

- **Method body**
  - Defines the actual body of the method that defines its logic.

Method overloading:

- More than one method with the same name but different in parameter types or numbers.
- **Return type CANNOT be used to overload a method**

```
no usages
void overloading (int a, int b){};
no usages
void overloading (int a, float b){};
no usages
void overloading (int a){};
no usages
int overloading(int a, int b){return 1;};
//there's already a method with the same name and parameters
//we can't overload based on return type -> compilation error.
```

ال parameters بتبقى pass by value بمعنى اني باخد نسخة من القيمة بتاعة الargument و ابعتها لل method

ف مهما اتغيرت القيمة دي في ال method ف مش هتأثر عندي ب حاجة.

علشان انا لما بغير فيه بكون بغير في reference تانيخالص غير الاساسي ال انا بعتاه.

الا لو انا بعت object ف في الحلة دي لو غيرت حاجة في ال obbject دا جوا ال method هتتغير عندي.

علشان ال reference بيكون هو هو ال بعتاه نفسه ال بغير فيه ف بيتغير عادي.

primitive type >> pass by value

objects >> pass by reference

<u>this</u> keyword:

- Refers to the object from which is called.

- Can be used in case of member are shadowed by methods/constructor parameters.

```
String name ;
1 usage
int id;
no usages
User (String name, int id){
    //we use this keyword to differentiate between the class fields
    //and the parameters of methods
    this.name = name;
    this.id = id;
    //this.id -> the class field id that's related to the objc
    //id -> the parameter that's sent tot the method.
};
```

- Can be used to call another constructor in the same class.

```
public class Student {
  String name;
  float marks;

  public Student(String name, float marks){
    this.name = name;
    this.marks = marks;
  }
  public Student(String name){
    this(name, 0.0F);
  }
```

main Method

- Main method should be:
  `public static void main(String[] args)`
- Why `public`? → Can be accessed from outside
- Why `static`? → No instantiation required
- Why `String[] args`? → Pass parameters to the program

Main method:

- Can be used to pass parameter to the program.
- All parameters must be strings.

```
Public class Greetings {
public static void main(String[] args){
      String name = args[0];
      System.out.println("Hello " + name);
   }
}
```

Strings:

- String are immutable -> meaning that its value cannot be changed.

ازاي immutable مينفعش اغير القيمة ما انا اقدر اخلي ال string يشيل كلمة تانية عادي؟

لما بساوي ال string بحاجة تانية ف انا كدا بروح احجز مكان جديد في الميموري يشيل الكلمة الجديده دي.

بالنسبالنا احنا كدا غيرنا الكلمة اه بس احنا في الحقيقة كأننا عملنا string جديد |

```
String s = "hello";
String s1 = s;
System.out.println(s == s1);
s = "hi";
System.out.println(s == s1);
```

```
"C:\Program F
true
false
```

s == s1 دا كدا انا بشوف هما ليهم نفس ال reference ولا لا

ف لما قولت s=hi كدا انا خليت s تاخد مكان جديد في الميموري ف ليها address مختلف عن ال كانت وخداه الاول

Useful String Methods

- String class has many methods. To name a few:
  - `substring`: returns substring starting at position and ends at anther position
  - `indexOf`: finds a substring if exists and returns its position, and -1 if not found
  - `lastIndexOf`: finds a substring if exists and returns its position but from end, and -1 if not found
  - `replace`: replaces a substring in the string with another substring
  - `split`: splits a string with a specific separator(s)
  - `startsWith`: checks if a string starts with the given substring
  - `endsWith`: checks if a string ends with the given substring

String Comparison

- '==' Operator <u>*CANNOT*</u> be used to compare strings
- It compares object references, i.e. their addresses
- To compare string contents `equals()` method can be used
- <, <=, >, and >= cannot be used to compare strings

- To compare strings `compareTo()` methods can be used
- It returns -1, 0, or 1
  - -1 if s1 < s2
  - 0 if s1 = s2
  - 1 if s1 > s2

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s4 = "b";
System.out.println(s2.compareTo(s1));
System.out.println(s2.compareTo(s3));
System.out.println(s2.compareTo(s4));
}
```

Main

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaa
1
-1
0
```

Lec4

Class reusability:

- Has two form -> composition and inheritance.
- Composition is also called has-a -> placing a reference/object of class in another one.
  For instance, relation between Employee and Department.

```
public class Department {
   Employee[] employees;
}
```

OR

```
public class Employee {
   Department department;
}
```

- Inheritance is also called is-a -> extending a class with another class.
- For instance, relation between Employee and Person.

```
public class Employee extends Person {

}
```

Inheritance:

- Means that a new class (called child class or subclass) inherits from an existing class (called parent class or super class)-> it inherits all its characteristics and non-private members.
- Can add/modify parent class functionality to fit its requirements.
- **A class can have exactly ONE superclass.(no multiple inheritance)**

ما عدا ال constructors مش بيتعملهم inheritance

Subclass constructors:

- Subclass constructor has to have a constructor similar to that of base class.

- Call to super MUST be the first statement in the child constructor.

لازم لما اعمل call لل super constructor وتكون اول جملة, و لازم اعمل call ليه اصلا مش هينفع معملوش.

لو انا مكتبتهاش ب ايدي ف هو كدا كدا ال complier هيعمل call لل default constructor ال موجود في ال parent class

و لو انا عاملة parameterized constructor و معملتش واحد default عند ال parent هيجبلي error.

```
public Child(String name){
    System.out.println(name);
}
```

Even if we didn't call the parent constructor it will get called implicitly anyways.

```
public static void main(String[] args) {

    Child child = new Child( name: "child");
}
```

```
"C:\Program Files\Java\jdk-21\bin
parent default constructor
child
```

Method overriding:

- **Method of subclass has the same signature** as the one in parent class but has different behavior.

لما اجي override ل method ممكن اخلي ال access modifier يكون less restrictive لكن مينفعش العكس

لو هو public مثلا مش هينفع اخليه prtoected/default/private

لكن لو هو كان default اقدر اخليه public عادي

```
no usages   1 override
void test(){
    System.out.println("parent method");
}
```

```
public void test(){
    System.out.println("child method");
}
```

```
public static void main(String[] args) {

    Child child = new Child( name: "child");
    child.test();
}
```

```
"C:\Program Files\Java\jdk-21\
parent default constructor
child
child method
```

Inheritance prevention:

- **Final keyword** can be used to **prevent class inheritance** -> a non-extendable class.

- If we try to extend a final class we get an error.

```
public final class Person {
```

Overriding prevention:

- Any method defined in Java is overridable by default.
- If a method is declared final in the parent class it can't be overridden in the subclasses -> ensures that the behavior of the method doesn't change.
- If we tried to override a final method we get an error.

```
public final void display() {
```

Abstract class:

```
public abstract class Vehicle {
```

- **A class that doesn't have any concrete functionality by itself.**
- It must be inherited to have a meaning.
- **Can NEVER be instantiated** -> **can't create an object of an abstract class**.
- Abstract classes can have both abstract and concrete(non-abstract) methods.
- Abstract classes can have constructors -> but we cannot call it.
- An abstract class variable can be instantiated with a reference to any of its subclasses (if they are not abstract)
- For instance:
- Vehicle v = new Truck(); or Vehicle v = new Car();

عمر ما ينفع يكون عندي class و اخليه final و abstract في نفس الوقت.

final معناه ان مينفعش اعمل inheritance

abstract يبقى لازم نعمل inheritance

Abstract method:

- A method that **has NO body only signature**.
- It has to be overridden in subclasses.
- **If a class has any abstract method then the class itself MUST be abstract.**

Polymorphism:

- Refers to the dynamic binding mechanism that determines which method definition will be called in case of overriding.
- Allows you to define general methods in super classes and leave implantation details for sub classes.
- **Promotes software extensibility** -> at the time of implementation you're not aware of the new classes that will be defined but you're sure that they should implement certain method.

Dynamic binding:

- When a method is overridden in a subclass and you define an object of base type (parent type) -> method of subclass is still called.

  For instance >> Person p = new Employee();

  This means that p internally refers to an Employee, however you can only reference methods defined in Person (at compile time).

- **Via inheritance, a variable of superclass can point to an object of the class itself or any of its subclasses**. [parent = new child]
  But, you **CANNOT directly make a variable o0f a subclass type and point to object of its superclass**. [child = new parent]
- The actual type of the instance AT RUNTIME determines which method will get invoked.

لو ال method معمولها override في ال childclass ف هي دي ال هيتعملها call اما لو مش معمولها ف

هيتعمل call لل موجوده في ال parent class`

Upcasting and downcasting:

- Upcasting is converting an object of a subclass to it superclass -> **done implicitly**

- Downcasting is converting an object of a superclass to one of its subclasses -> **Must be done explicitly.**

```
Parent parent = new Parent();
Child child = new Child();
//upcasting
parent = child;
//downcasting
//can't make a subclass point at an object of a superclass -> compilation error
child = parent;
//i have to type cast it explicitly
child = (Child) parent;
```

Instance of operator:

- instanceof operator can be used to check whether an object is a certain class type or not.

```
Parent parent = new Parent();
Child child = new Child();
Parent childUpcasted = new Child();
if(parent instanceof Parent)
    System.out.println("parent is instance of parent class")
if(parent instanceof Child)
    System.out.println("parent is instance of child class");
if(child instanceof Parent)
    System.out.println("child is instance of parent class");
if(child instanceof Child)
    System.out.println("child is instance of child class");
if(childUpcasted instanceof Child)
    System.out.println("childUpcasted is instance of child class");
if(childUpcasted instanceof Parent)
    System.out.println("childUpcasted is instance of parent class");
```

```
parent is instance of parent class
child is instance of parent class
child is instance of child class
childUpcasted is instance of child class
childUpcasted is instance of parent class
```

Lec5

Exceptions:

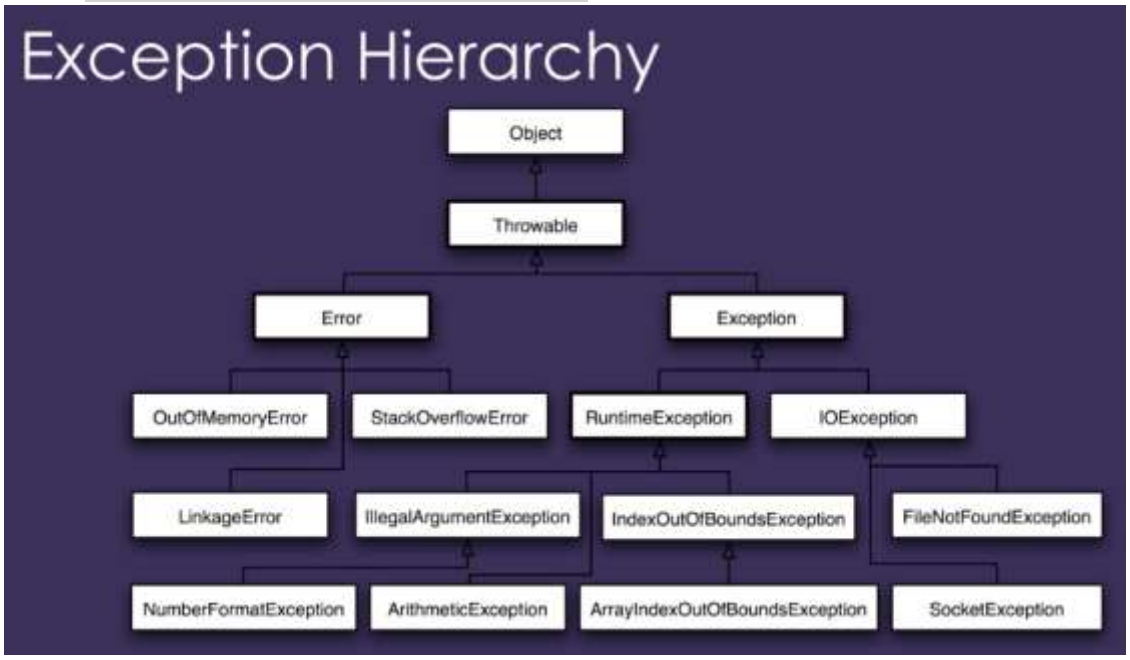
- is a problem/error that occurs **during the normal flow** of a program.
- It causes the program to terminates abnormally, unless there's an exception handling block to help handling it gracefully.
- You have to handle them using try-catch or try-catch-finally.
- Sometimes a code block may throw several types of exceptions → multiple catch blocks are required. [try-catch()-catch()….]
- **There's two types of exceptions:**
  - Checked exceptions.
  - Unchecked exceptions.

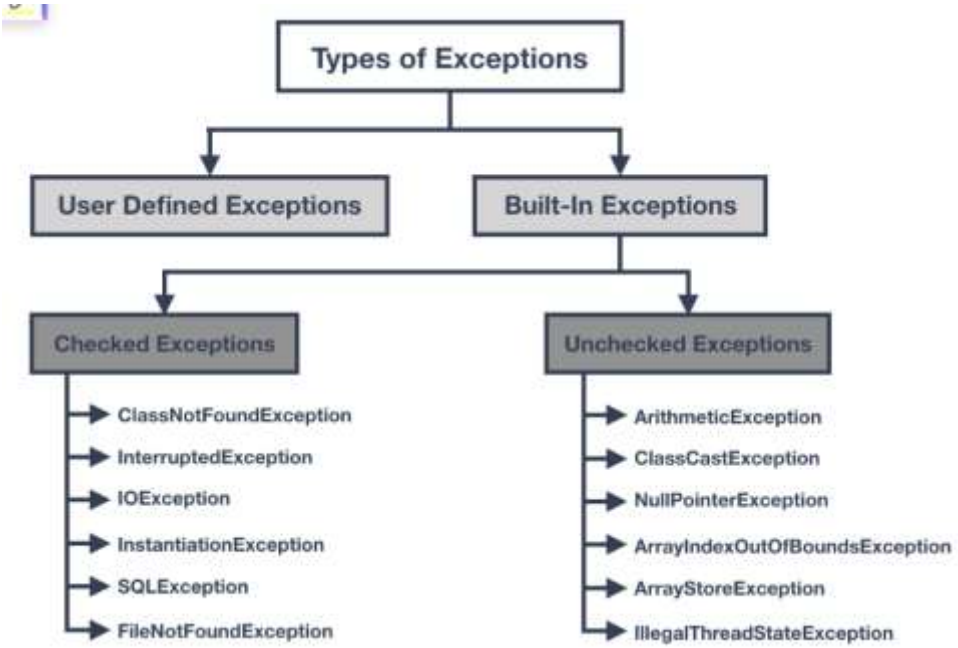
Checked exceptions:

- Are the ones that the program should anticipate and handle.
- They're **checked at compile time** -> you must handle them or you get compilation error.
- MUST extend –Throwable- class either directly or indirectly.

Unchecked exceptions:

- Occurs outside the program, most of the time can't be expected.
- They're **not checked at compile time**, you may or may not handle them -> no compilation error if not handled.
- MUST extend -RuntimeException- class either directly or indirectly.





Exception Example

```
public void writeArrayItemsToFile(int[] arr, int length){
    PrintWriter printWriter = new PrintWriter(new FileWriter("E:/input.txt"));
    for (int i = 0; i < length; i++) {
        printWriter.write(arr[i] + "\t");
    }
    printWriter.close();
}
```

- Two types of exceptions can be thrown here; `IOException` and `ArrayIndexOutOfBoundsException`

- Compiler will prompt you about `IOException` only
- Why??? → IOException is a checked exception while `ArrayIndexOutOfBoundsException` is not
- In order to resolve that error, this code fragment should be enclosed in a `try-catch` block

```
public void writeArrayItemsToFile(int[] arr, int length){
    try{
        PrintWriter printWriter = new PrintWriter(new FileWriter("E:/input.txt"));
        for (int i = 0; i < length; i++) {
            printWriter.write(arr[i] + "\t");
        }
        printWriter.close();
    } catch(IOException exp){
        System.out.println(exp.getMessage());
    }
}
```

Catching All Exceptions:

```
try{
    ....
} catch(IOException exp){
    System.out.println(exp.getMessage());
} catch(ArithmaticException exp){
    System.out.println(exp.getMessage());
} catch(Exception exp){
    System.out.println(exp.getMessage());
}
```

- Since all exceptions extend class –Exception-, if there is a catch block for Exception, then all exceptions will be caught by that catch block.
- The exceptions should be ordered from the very special to the very general -> Exception class MUST BE placed at the very end.

حتى لو الكود هيعمل اكتر من exception ف دا مش معناه ان الكود هيدخل كل ال catch blocks بتاعتهم.

اول حاجة من ال exceptions هتحصل ف هنسيب ال try كلها مش هتكمل و هنمشي على ال catch blocks واحده واحده

لحد ما يلاقي حاجة بت match ال exception ال حصل دا و ي-execute الكود بتاعها و خلاص على كدا.

علما ان لو في finally block هنحش فيه كدا كدا مهما كان في exception ولا لا اصلا, ديما هينفذ.

ال catch blocks كأنها if , else if

طلاما دخلت في block منهم مش هتدخل حاجة تانية.

Throwing an exception:

- If in your method, there is a critical error that you need to notify about -> you can throw an exception (built in or user defined).
- In order to declare that a method throws and exception, you can use -throws- keyword

User-defined exception:

User Defined Exception Example

```
public class NumberRangeException extends Exception {
    public NumberRangeException(int lowerBound, int upperBound){
    super("The number must be between " + lowerBound + " and " + upperBound);
    }
}
```

```
public int add(int num1, int num2) throws NumberRangeException{
    if(num1 < 1 || num1 > 100 || num2 <1 || num2 > 100){
        throw new NumberRangeException(1, 100);
    }
    return num1 + num2;
}
```

We create a class for the user-defined exception and extends it from Exception.

javaDocs:

- JavaDocs are used to document your own code and/or APIs.
- Once JavaDocs are generated, a couple of HTML files containing documentation are generated.
- The **classes are grouped by their respective packages.**
- The documentation has to be enclosed in: /** .... */

JavaDoc Example

```
/**
* This is a simple calculator
*/
public class Calculator {
    /**
    * Adds two numbers and returns their sum
    * @param num1 First number
    * @param num2 Second number
    * @return The sum of the two numbers
    */
    public int add(int num1, int num2){
        return num1 + num2;
    }
}
```

Lec6

Interfaces:

- An interface is a pure abstract class.
- Interfaces cannot be instantiated, like in abstract classes.
- **No constructors can be defined** as the interface CANNOT be instantiated by itself.
- An interface constitutes a contract between a class and the outside world, and it is imposed at compile-time by the compiler thus, **if a class implements an interface**, then it **HAS TO OVERRIDE ALL of the interface methods.(all of the non-default and non-static methods.)**

لازم نعمل override لكل ال non-default methods ال موجوده في ال interface عند ال classes ال عاملة implement ليها

لو سبت و لو واحده معملتهاش override الكود مش هيـرن -> compilation error

- A class can implement AS MANY interfaces as required.
- An interface can inherit (extend) another interface.
- **Interface components:**
  - An interface can contain both fields and methods. However, those members are special.
  - All data members are public static final, and there is no need to define them explicitly as public static final.
  - All methods are public abstract, and there is no need to define them explicitly as public abstract

- Some methods can be declared as default.

Interface Example

```
public interface Calculator {
    public static final int MIN_NUMBER = 0;
    public static final int MAX_NUMBER = 100;

    public abstract int add(int num1, int num2);
    public abstract int subtract(int num1, int num2);

    public default int multiply(int num1, int num2){
        return num1 * num2;
    }
}
```

Interfaces can also have static methods.

public و public abstract و public static final

مش لازم اكتبهم لانهم كدا كدا معروفين طلاما دي interface

كتبتهم او لا عادي مش غلط

For more explanation and examples about interfaces please refer to Lab8

https://drive.google.com/file/d/1cYjgJtZSF3676OqzdjUlFQkCLmV2kUeV/view

Default methods:

- They provide a default implementation for a specific method.
- Unlike other interface methods, the implementing class **may or may not override them.**

```
public interface Vehicle {
    float speedUp();
    default float slowDown(){
        return 0.0F;
    }
}
```

speedUp method MUST be overridden while slowDown method may or may not be overridden (not an obligation)

Interface inheritance details:

- If a **class** implements an interface and it is **declared abstract** it **does not have to implement all interface methods.**
- If a **superclass implements an interface** -> **all its subclasses also implement it.**

Most common interfaces:

- Comparable.
- Comparator.
- Serializable.
- Cloneable.

Interfaces vs Abstract Classes

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

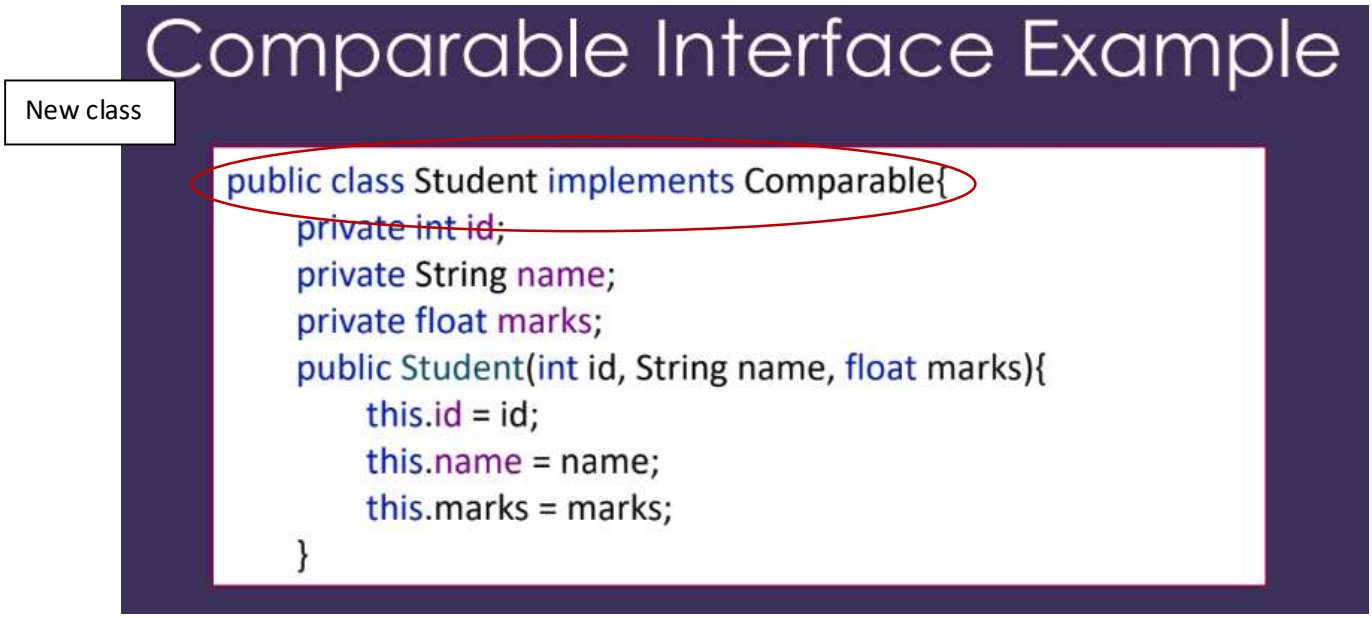
Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Comparable Interface:

- **It has a single method compareTo.**
  It has the form: int compareTo(Object o)
  It returns one of three possible values, -ve, 0, or +ve
- It can be used for comparing two objects of the same type(two students/two vehicle objects).
- Comparing two or more objects helps in sorting them either ascendingly or descendingly.
- It can be used to achieve class's "**natural ordering**.", i.e**., default sorting mechanism.**
- All common types (Intger, Float, String) implement comparable interface -> that's why we can compare and ultimately sort an array of that type.(page 12 Lec 3, compareTo with strings)
- Therefore, we can sort an array of strings using Arrays.sort() or Collections.sort() for lists.



```java
    @Override
    public int compareTo(Object obj) {
        Student otherStudent = (Student) obj;
        if(this.id < otherStudent.id){
            return -1;
        } else if(this.id > otherStudent.id) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

```java
public static void main(String[] args) {
    Student[] arr = new Student[3];

    arr[0] = new Student(7, "Ahmed", 30.0F);
    arr[1] = new Student(1, "Mona", 50.0F);
    arr[2] = new Student(5, "Ashraf", 70.0F);

    Arrays.sort(arr);
}
```

```
"C:\Program Files\Java\jdk-21\bin\ja
object at index = 0  is -> Mona
object at index = 1  is -> Ashraf
object at index = 2  is -> Ahmed
```

Output of the previous code after sorting >>

Comparator Interface:

- Similar to Comparable interface, Comparator can be used for comparing two objects of the same type however, **Comparator can override the natural ordering**, i.e., **deviate from the default sorting mechanism.**
- It has form: int compare(Object o1, Object o2)
- It also returns one of three possible values, -ve, 0, or +ve.
- It has to be passed as a parameter to Arrays.sort() or Collections.sort()

In the last example (page 27) the natural ordering of Students is via -id- member.

What if we want to sort them via -marks- instead?

We SHOULDN'T change the natural ordering.

Rather we use – Compartor- interface to override that natural ordering.

- -ve: if the first object is smaller than the second object
- 0: if both objects are equal
- +ve: if the first object is larger than the second object

```java
public class StudentComparator implements Comparator {
    @Override
    public int compare(Object obj1, Object obj2) {
        Student student1 = (Student) obj1;
        Student student2 = (Student) obj2;
        if(student1.marks < student2.marks){
            return -1;
        } else if(student1.id > student2.id) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

```java
public static void main(String[] args) {
    StudentComparator comparator = new StudentComparator();
    Student[] arr = new Student[3];

    arr[0] = new Student(7, "Ahmed", 30.0F);
    arr[1] = new Student(1, "Mona", 50.0F);
    arr[2] = new Student(5, "Ashraf", 70.0F);

    Arrays.sort(arr, comparator);
}
```

```
"C:\Program Files\Java\jdk-21\bin\ja
object at index = 0  is -> Ahmed
object at index = 1  is -> Mona
object at index = 2  is -> Ashraf
```

Lec7

Lists:

- A -List- Is a standard Java interface that provides a way to **store ordered collection of elements.**
- Is ordered is the same as sorted?? No
  - Ordered means that it **preserves the insertion order of elements**.

ArrayList:

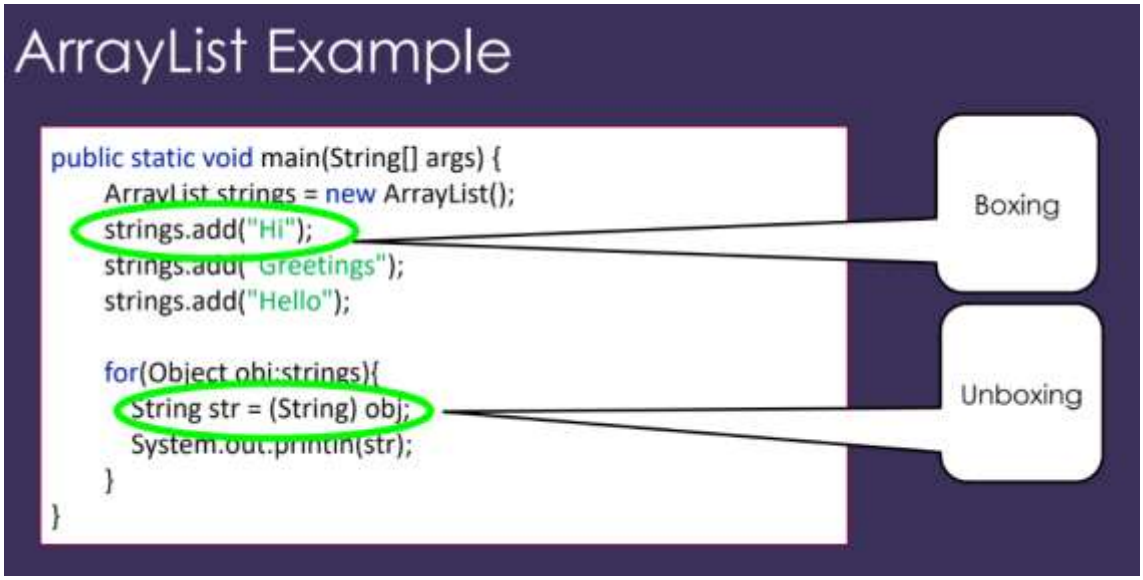
- An -ArrayList- is an auto **resizable** array of elements.
- It implements the -List- interface.
- It can hold **any** type of objects-> **can't hold primitive type.**

```java
ArrayList<String> arrayList = new ArrayList<>();
//to add elements.
arrayList.add("hello 1");
arrayList.add("hello 2");
//to get a specific index.
System.out.println(arrayList.get(1));
System.out.println("-----------");
for (String s: arrayList)
    System.out.println(s);
//to update a specific element
arrayList.set(0,"hello 1 repeated");
System.out.println("-----------");
for (String s: arrayList)
    System.out.println(s);
System.out.println("-----------");
//to search for an element and return its first occurrence
//and return its index, return -1 if it doesn't exist.
System.out.println(arrayList.indexOf("hello 2"));
System.out.println(arrayList.indexOf("hello 1"));
System.out.println("-----------");
//to check if this element exists or not
System.out.println(arrayList.contains("hello 2"));
System.out.println(arrayList.contains("hello 1"));
```

```
"C:\Program Files\J
hello 2
-----------
hello 1
hello 2
-----------
hello 1 repeated
hello 2
-----------
1
-1
-----------
true
false
```

Here I specified that the arraylist will only contain strings.

But I can make an arraylist hold any kind of object but I have to unbox the elements with the right datatypes.



Wrapper datatypes:

Wrapper classes is used to store primitive datatypes. -> Integer datatype is used for wrapping the -int- datatype.

Boxing:

Is a wrapping operation to transform a primitive type into an object (wrapper datatype), once we box a datatype we get an object.

Unboxing:

Is the reverse transformation of boxing.

- ArrayList can accept any type of object -> it can accept mixed data types(here comes the importance of unboxing.)
- You MUST use the correct unboxing when accessing each element -> otherwise an exception will be thrown.

```java
ArrayList arrayList = new ArrayList<>();
//arraylist hold any type of objects
arrayList.add(1);
arrayList.add("hello");
arrayList.add(3.0F);
//right unboxing
System.out.println((int) arrayList.get(0));

//invalid unboxing
System.out.println((String) arrayList.get(0) );
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\
1
Exception in thread "main" java.lang.ClassCastException Create breakpoint : class java.l
    at Main.main(Main.java:19)
```

Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java.lang.String <- previous exception

Generics:

Like what I did in the example on page 30

- Generics allow "a type or method to operate on objects of various types while providing compile-time type safety".
- You can use that feature **to restrict the types a collection accepts**.
- Helps in avoiding the incorrect unboxing -> boxing and unboxing aren't required -> Avoids ClassCastException that might be thrown in case of incorrect unboxing.

```java
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add(3);
```

Required type: String

Provided: int

We've limited the arraylist to take only strings.

Overriding -equals-:

- Similar to -toString-, -equals- is a method in -Object- class.
- You should override equals in order to support equality check, otherwise Java **compares objects based on their respective address** -> **similar to Strings ->** (s1 == s2)
- Its form: public boolean equals(Object obj)

```
public class Student {
    private int id;
    private String name;
    private float marks;
    public Student(int id, String name, float marks){
        this.id = id;
        this.name = name;
        this.marks = marks;
    }
}
```

indexOf without Overriding equals

```
public static void main(String[] args) {
    ArrayList<Student> students = new ArrayList<>();
    students.add(new Student(5, "John Smith", 70.5F));
    students.add(new Student(2, "Jane Doe", 59.5F));

    int index = students.indexOf(new Student(2, "Jane Doe", 59.5F));
    System.out.println(index);
}
```

Output

-1

The output is -1 **as Java compares elements based on their addresses**.

We should override equals to instruct Java to check the equality based on some field.

```java
@Override
public boolean equals(Object obj) {
    Student otherStudent = (Student) obj;
    if(this.id == otherStudent.id){
        return true;
    } else{
        return false;
    }
  }
}
```

Overriding `equals` Example

```java
public static void main(String[] args) {
    ArrayList<Student> students = new ArrayList<>();
    students.add(new Student(5, "John Smith", 70.5F));
    students.add(new Student(2, "Jane Doe", 59.5F));

    int index = students.indexOf(new Student(2, "Jane Doe", 59.5F));
    System.out.println(index);
}
```

Output

1

Stream APIs:

- They allow dealing with collections as a sequence of elements.
- Enable performing operations such as **filtering**, or **reducing**
- There are many stream functions -> We check filter and forEach only.

Without Stream APIs Example

```java
public static void main(String[] args) {
    ArrayList<String> strings = new ArrayList<>();
    strings.add("Hi");
    strings.add("Greetings");
    strings.add("Hello");

    for(String str:strings){
        if(str.startsWith("H")){
            System.out.println(str);
        }
    }
}
```

Iterating over the list and printing only the ones that start with "H"

```java
//with stream API
ArrayList<String> strings = new ArrayList<>();
strings.add("Hi");
strings.add("Greetings");
strings.add("Hello");

strings.stream()
        .filter(str -> str.startsWith("H"))
        .forEach(str -> System.out.println(str));
//it won't affect the arraylist.
System.out.println("----------------");
for(String s:strings)
    System.out.println(s);
```

```
"C:\Program Files\
Hi
Hello
----------------
Hi
Greetings
Hello
```

End of lecture 7 yahooo!!!

And of course all of this is absolutely useless if you don’t write and trace code yourself.

Don’t just count on this please this is only a way to gather lecture notes in one place “)

Best of luck mates ^^