**Variable Storage**

Primitive data values are generally stored on the **STACK** (when they are declared inside of a method or a block) **.**If they represent a field of an object( an instance variable of a class), then they are stored on the **HEAP .**

Objects are stored on the **HEAP**.

String str = new String(“myString”).

str is a **reference variable** of type string which holds the memory address of the ‘String’ object created using the new keyword.

* Reference variables are stored on the stack and the objects they reference are stored on the heap
* When the scope of the reference variable ends (when its declared in a method), the stack memory allocated for it is released. An object exists on the heap until there are no longer any reference variables referencing it. Then, it becomes eligible for garbage collection.

**ACCESS MODIFIERS :PPDP**

They restrict access to classes (only public and private), methods, constructors, variables

If we make variables private but make public methods for access (getters and setters) we make sure we won’t encounter unexpected values for those variables.

**Public**: Available everywhere

-> Same Class

-> Same package Subclass

-> Same package non- Subclass

-> Different Package Subclass

-> Different Package non-Subclass

**Private**: Available only inside the class

-> Same class Only

-> Private classes are only defined and used within another class

**Default**: (PACKAGE PRIVATE) Available inside the same package

-> Same class

->Same package Subclass

-> Same package non-Subclass

**Protected**: Available in the same package or in different package, subclass

-> Same class

-> Same package subclass

-> Same package non-Subclass

->Different package, but it has to be a subclass

**NON-ACCESS MODIFIERS**

**­­­­­­­**- static, final, abstract, synchronized, volatile

**STATIC**

**Static Variables:**

-> Variables which exist independently of any instance of the class where they were declared

-> Only one copy of the variable exists

-> INSTANCE VARIABLES declared as static are also known as CLASS VARIABLES

**Static Methods**

**->** Methods which exist independently of any instance of the class where they were declared

-> Methods don’t use instance variables, ONLY CLASS VARIABLES

**Static Classes**

**->** Only available as inner classes (which are classes nested in other classes)

-> Only can access Class variables, not instance variables

**FINAL**

**Final Variables:**

-> To create constant variables

-> It’s value is initialized only once: when it’s declared, inside a static block, or inside an initializer block

**Final Methods:**

**->** To prevent method overriding

**Final Classes:**

**->** To prevent inheritance

**->** Classes like Integer, Double are final classes

-> To create an immutable class: Once an object is created, we cannot change it’s content.

**ABSTRACT**

**Abstract Classes:**

**->** Made only to be extended

-> A class cannot be both final and abstract

-> It can contain both concrete and abstract methods

-> Abstract classes can have fields and constructors, with all the possible access modifiers and they can be accessed from child classes

-> A class can extend only one abstract class

**Abstract Methods:**

**->** Method declared without an implementation, the implementation will come from the subclass

-> A subclass of an abstract class must implement all the abstract methods from the superclass, unless the subclass is also an abstract class

-> If a class contains abstract methods, the class must be declared an abstract class

Sidenote

**INTERFACES**

* An interface is a completely abstract class
* It can’t have any instance variables. Its variables will only be public, static and final at the same time (Basically constants, and we access them statically)
* Interfaces can have default methods ( methods to which we give the implementation)

They were introduced so we no longer have to deal with the diamond problem.

example:

Class child inherits from both interface Father and interface Mother. If both of those interfaces had defined an abstract method with the same name, we would get an error.

**SYNCHRONIZED**

**Synchronized Methods:**

**->** Synchronized blocks inside the run() method or synchronized methods by themselves are used within multi-Threading programs to ensure that only one thread can access the block at one time. All the other threads attempting to enter the synchronized block will be blocked until the thread inside the block exits.

-> This keyword helps prevent the access of one method by multiple threads simultaneously

synchronized(sync\_object)

{

// Access shared variables and other

// shared resources

}

**VOLATILE**

-> When multiple threads are working with the same variable, having that variable declared as volatile, tells all the threads that use it, that once they used it, they must merge their copy of the variable with the super copy in the memory.

-> Used when we desire visibility without mutual exclusion

-> Syncronized assures both visibility and mutual exclusion

**OOP CONCEPTS IN JAVA : AEIP**

The four concepts are:

**Abstraction, Encapsulation, Inheritance, Polymorphism**

**Abstraction:**

**->** We use simple things to represent complexity

**Encapsulation:**

**->** Wrapping up data and methods into a single unit : class.

-> Keeping data safe, while providing access to the data using public methods, like setter and getters

-> We can reuse data or code without allowing access to the data system wide

**Inheritance:**

**->** Lets us create new classes that share some attributes of existing classes

-> It lets us build on previous work without rewriting everything

-> A subclass inherits all the non-private methods and attributes of the superclass (**NOT CONSTRUCTORS THO)**

-> Multiple inheritance is possible with implementing interfaces

-> Declaring a field in a subclass that also exists in the superclass with the same name results in hiding the inherited field from the superclass (**This should be avoided)**

**->** Declaring a new static method in the subclass with the same signature as one in the superclass hides the inherited one

**Polymorphism:**

**->** Let’s us create methods or variables which behave differently in a different context

**-> Compile Time Polymorphism and Run Time Polymorphism**

**Compile-Time Polymorphism:**

**->** Method Overloading: methods can vary in number of arguments, order of arguments, argument types, method types. **NOT IN RETURN TYPE**

**Run Time Polymorphism:**

**->** We usually have a parent class and a child class. We have a method in the parent class which is overridden in the child class. We can instantiate a parent class referencing a child class. And if created such an object and then we would call the common method, at run time, it would be decided which method is actually called.

**SOLID DESIGN PRINCIPLES IN JAVA**

Single Responsibility:

-> A class should have only one responsibility and only one reason to change

-> A class should be responsible to only one actor or stakeholder

-> It makes testing easier and makes the code more organized and easier to read

-> It helps us maintain the separation of concerns

-> Not respecting this principle could results in having something called a GOD Object which is an anti-pattern

Open-Closed principle:

-> A class should be open for extension, closed for modification

-> We should not modify our existing working code, since we might get errors in doing so. We should extend the class and add functionalities as we please, while keeping our old code working as it was originally intended

-> We should extend by composition over inheritance

Liskov Substitution:

-> If class A is a subclass of class B, we should be able to replace class B with class A without modifying the way our program works

-> From what I understand of it, it’s a way of deciding when we should use inheritance and when we shouldn’t. Let’s say we have an Animal Class and a Cat Class which extends the Animal Class and animals speak. Therefore, cats speak too. But if we would have another subclass Snail, we would break the Liskov Substitution Principle because even though snails are still animals, snails don’t speak.

-> Whenever we use the supertype, we should be able to swap that instance for an instance of the subtype without affecting the logic

-> A child shouldn’t remove things that we know are true about the parent !!!

Professor Bertrand Meyer talked about **Design by Contract** stating that a class defines a contract. The preconditions and postconditions are assertions which should be valid, before the execution of an instruction and after respectively.

The Liskov Substitution Principle basically states a subclass should respect the contract of its superclass. => A subclass should not ask more than the superclass (referring to preconditions, no stronger preconditions) and should not provide less (referring to postconditions, no weaker postconditions).

Precondition = condition which should be true for the method/code to work

Postcondition = condition which must be true after running the method.

Interface Segregation

-> Client should not be forced to depend on interfaces they do not use

-> Big interfaces should be split into many more smaller interface

-> That way, we can design new objects without adding entirely new classes

-> This way, we shouldn’t make clients depend on big interfaces but rather let each client only depend on the interface he needs

-> It favors composition over inheritance (microservices like)

An example would be the printer example. Let’s say we have an interface called Machine, which has 3 methods : print, fax and scan.

A user might come and create a class called AdvancedPrintingMachine which implements this Machine Interface and everything would work fine.

But if another user comes with a class called OldPrintingMachine, they would try to implement this interface but it wouldn’t benefit them because their OldPrintingMachine can only realistically use the print method from the interface ( the methods fax and scan would have no purpose because the OldPrintingMachine cannot satisfy these functions)

To solve this, we could implement 2 interfaces called OldMachine and NewMachine. OR we could even create 3 separate interfaces called Printer, Scanner and Faxer, each containing their respective method. ( Because, as we know, a class may implement many interfaces, so the required methods could be taken from each interface by implementing it)

Dependency Inversion:

-> High level modules should not depend on low level modules, both should depend on abstractions

-> If we have a class, which is constructed with other classes, we have tightly coupled these classes together, it also makes the high level module hard to test.

To solve this, the lower modules should implement interfaces. And whenever we would want to implement the higher module which would be implemented with lower modules, we would construct the higher module to be implemented with the corresponding interfaces of those lower modules instead.

Text

Description automatically generated

Let’s say we want to create an app that stores information about relationships between people

So we would have a class for people and an enum for the different types of relationships people can have.

Text

Description automatically generated

Now create a class relationships which uses TRIPLETS to store the relationships.

This class has a private list which stores all known relationships and different methods for creating relationships between people.

Text

Description automatically generated

And now create a class called Research which takes in its constructor an object of type Relationships. From the Relationships object we get the relations list.

And then we do whatever we want to do with the relations we have stored.

Text

Description automatically generated

**THIS EXAMPLE BREAKS THE DEPENDENCY INVERSION**

**Because**

The Relationships class is a low-level module. It doesn’t contain any business logic. It’s only job is to store the data and do some modification on to it like add new relations.

The Research class is a high-level module. It is the one which the user has the most interest in. It does everything we want to do with researching through the relations. And the problem is it depends on the low-level module because it has the Relationships class as a dependency.

So how do we fix this? => We should depend on abstractions instead.

So we introduce an interface called RelationshipBrowser.

Text

Description automatically generated

And we now have the Relationship class implement this interface. In the relationship class, we provide the implementation for this findAllChildrenOf method.

Text

Description automatically generated

So now in the high-level module, we will no longer depend on the concrete class, we will depend on this interface abstraction.

Text

Description automatically generated

And this provides additional flexibility because now, let’s say we want to change the list of relationships from the Relationship class to no longer be a list of triplets, to be something else. Well now we can and the implementation we see in the above picture will still work.

**Data Structures**

**SET - INTERFACE**

* Unordered collection of objects in which duplicates cannot exist
* They don’t maintain the insertion order
* There are 2 interfaces which extend the set implementation : SortedSet and NavigableSet
* TreeSet is an implementation of a self balancing tree
* Set is an interface so we cannot create objects of this type. We always need a class which extends this list in order to create an object.

**HASH SETS - CLASS**

* Implements the set interface
* The data structure it represents is a hash table
* Duplicate values are not allowed
* Order of insertion is not maintained. The objects are inserted based on their hash code
* NULL elements are allowed in a hash set
* HashSet also implements Serializable and Clonable

**LINKED HASH SETS – CLASS**

* A hashSet in which the order of insertion is maintained
* It represents a hash table data structure with a doubly linked list between its elements

**SORTED SET – INTERFACE**

* Extends the Set interface
* Stores all the elements in this set in a sorted manner
* Implemented using the TreeSet class

**TREE SET – CLASS**

* Extends the Sorted Set interface
* Stores elements in a sorted manner
* Very useful to iterate over both in ascending and descending order using the .iterator() and .descendingIterator() methods respectively

**LIST ASPECTS**

Adding a list to itself works perfectly if the types of the list’s elements are basic data types.

If the list contains objects, the addAll will not insert the same elements in the list again, but will insert references to the initial elements. So if you have a list of 2 objects and you call the addAll method for the list, with the same list, the third element will be the reference of the first, so it will be modified automatically if you modify the first element.

O imagine care conține text

Descriere generată automat

**THREADS**

* An executing instance of a program is called a process. A thread is a subset of the process
* Threads share the address space of the process that created them while processes have their own address space
* Each thread has its private run-time stack => if 2 threads execute the same method, each thread will have its own copy of the local variables of the method
* All threads see the same dynamic memory (heap) => 2 different threads can act on the same object or the same static fields concurrently
* There are 2 ways to create threads :

-> We subclass (extend) the Thread class and instantiate a new object of that class

-> We implement the Runnable Interface

-> In both cases, the run() method has to be implemented

**Threads are useful because:**

* They maintain the responsiveness of an application during a long time run (A client keep requesting data from an application)
* They enable the cancelation of separate tasks (You collect data from 2 devices at the same time. You want to stop collecting data from the first but keep collecting from the second)
* They help solve problems that can only be solved with parallelism