CORE JAVA

Day-2

REFERENCE TYPES

- These types are references to objects. They point to a memory location where the actual object resides. Reference types include:
  - Classes
  - Interfaces
  - Arrays
- Reference types are passed by reference. When you pass a reference type to a method, you're actually passing a reference to the object, not a copy of the object itself. This means if you change the object within the method, the changes reflect outside the method as well.

REFERENCES TO OBJECTS

- a reference variable contains the memory address of an object. When you create an object using the new keyword, memory is allocated for that object on the heap, and the reference variable holds the memory address where the object resides.
- Reference variables store the memory address of objects, but the exact nature of this address is encapsulated within the Java Virtual Machine (JVM) and not exposed to the programmer.
- The address stored in a reference variable is an opaque handle managed by the JVM. It's not directly accessible or manipulable by the programmer. This encapsulation provides several benefits, including:
  - Abstraction: Programmers don't need to worry about the low-level details of memory management, such as memory addresses and allocation/deallocation.
  - Security: By encapsulating memory addresses, Java provides a level of security and prevents unauthorized access to memory locations.
  - Platform Independence: The JVM abstracts away the underlying hardware details, allowing Java programs to run on different platforms without modification.

PRIMITIVES VS REFERENCES

Feature	Primitive Types	Reference Types
Data Storage	Stores actual data values	Stores memory addresses (references)
Memory Location	Stored on the stack	Objects stored on the heap, references stored on the stack
Direct/Indirect Access	Direct access to data values	Indirect access through references
Default Values	Default values assigned (e.g., 0, false)	Default value is null
Examples	int, double, boolean, char	Classes, interfaces, arrays, strings

TOSTRING()

- In Java, the Object class is the root of the class hierarchy. Every class in Java is a descendant of Object. If a class does not explicitly extend any other class, then by default, it implicitly extends Object.

- The Object class defines several methods that are common to all objects in Java, such as toString(), equals(), hashCode(), getClass(), and finalize(), among others. These methods can be overridden by subclasses to provide specific implementations.

- For example, the toString() method returns a string representation of the object. By default, it returns a string that consists of the class name followed by the "@" symbol and the object's hash code in hexadecimal. However, you can override this method in your own classes to provide a more meaningful string representation.

THIS KEYWORD

- In Java, "this" serves as a reference to the current object instance.
- "this" is a keyword that refers to “this” reference.
- It's often used within a class's method or constructor to refer to the current object. "this" can be particularly useful in situations where there's a need to disambiguate between instance variables and parameters with the same name.

GETTERS AND SETTERS

- Getters and Setters functions are commonly referred to as accessor and mutator methods, respectively.
- These methods are used to access and modify the private fields (variables) of a class while encapsulating the implementation details.
- Getters:
  - Purpose: Getter methods are used to retrieve the value of a private field from an object. They provide read access to the fields.
  - Naming Convention: Typically, getter methods are named with the prefix "get" followed by the name of the field they access.
  - Return Type: Getters return the value of the field they are associated with.

GETTERS AND SETTERS

Getters:

- Purpose: Getter methods are used to retrieve the value of a private field from an object. They provide read access to the fields.
- Naming Convention: Typically, getter methods are named with the prefix "get" followed by the name of the field they access.
- Return Type: Getters return the value of the field they are associated with.
- Ex

```
public int getAge()
 {
   return age;
}
```

GETTERS AND SETTERS

Setters:

Purpose: Setter methods are used to modify the value of a private field in an object. They provide write access to the fields.

Naming Convention: Usually, setter methods are named with the prefix "set" followed by the name of the field they modify.

Parameters: Setters take one parameter, which is the new value to be assigned to the field.

Example:

```
public void setAge(int age)
{
   this.age = age;
}
```

STATIC MEMBERS

- A static member belongs to the class itself rather than to instances of the class.
- This means there is only one copy of a static member, regardless of how many instances of the class are created. All instances share the common value.
- Static members can include variables, methods, and nested classes.
- These are also known as class variables because they belong to the class, not to any particular instance of the class. You declare a static variable using the **static** keyword.
- You can access the static variable using the class name.
- Static variables are initialized only once, when the class is loaded into memory, before any objects of that class are created.
- Initializing static member in constructor is not a good practice and may lead to unexpected behaviour.
- We cannot use the this keyword with static members in Java

STATIC METHODS

- Static methods in Java belong to the class itself rather than to any particular instance of the class.
- They can be invoked without the need for creating an instance of the class.
- They can only access other static members of the class directly.
- They cannot access instance variables or instance methods directly, but they can access them via an object reference if one is provided as a parameter.
- They are not overridden in subclasses, meaning that if a subclass defines a static method with the same signature as a static method in its superclass, it hides the superclass method rather than overriding it.
- They can be accessed even if no objects of the class have been instantiated.
- They are typically used for operations that do not require access to instance-specific data, such as utility methods or factory methods.