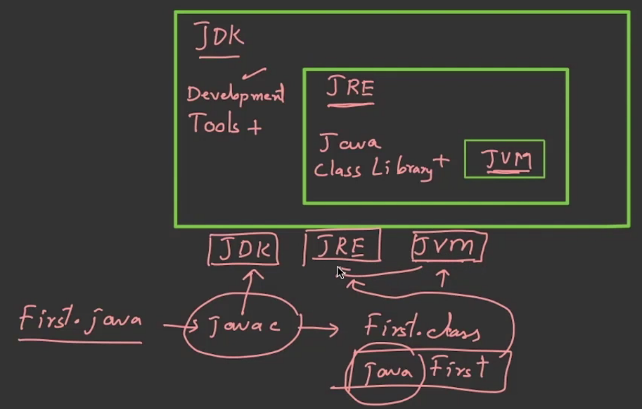
**Java**

**Section 1: Introduction**

  **A diagram of a computer program

AI-generated content may be incorrect.**

**JDK:-**

It contains developer tools like:-  
 javac -> **Java Compiler** – Converts .java source code into .class bytecode.

Java -> **Java Application Launcher** – Runs Java applications by executing .class files.

Jar ->**Java Archive Tool** – Packages multiple .class files and resources into a .jar archive.

Jdb -> **Java Debugger** – Debugs Java applications at runtime.

**Java Compiler**: This tool takes your Java code (written in a language humans understand) and converts it into bytecode.

JRE:-

There are many build in library which helps the code to run.

the JDK is like a full toolbox for both **building** and **running** Java programs, while the JRE is just the part needed to **run** them. If you're a developer, you use the JDK. If you're just running a Java program someone else made, the JRE is enough.

JVM:-

Converts the byte code to machine and it will execute the program.

**Compiler:-**

* **What it Does**: Translates the entire code (written by you) into machine language (code the computer understands) all at once, before the program runs.
* **Output**: Creates a separate file (like .exe or bytecode) that can be executed without the compiler.
* **Speed**:
  + Slower initially because it has to translate the whole code first.
  + Faster during execution since everything is already translated.
* **Example**: Languages like **C**, **C++**, and **Java** (Java compiles to bytecode).

**Real-life analogy**: Imagine translating an entire book from one language to another before reading it. Once translated, you can read it anytime without needing a translator.

**Interpreter:-**

* **What it Does**: Translates and runs the code line by line. It doesn’t create a separate file.
* **Output**: No file is saved; it executes directly.
* **Speed**:
  + Faster to start running since it doesn’t translate the whole code upfront.
  + Slower during execution because it translates line by line.
* **Example**: Languages like **Python**, **JavaScript**, and **Ruby**.

**Real-life analogy**: Imagine a translator standing next to you, translating each sentence as you read it.

Java uses **both**!

* It **compiles** your code into **bytecode** using a compiler.
* Then, the **JVM** uses an **interpreter** (and sometimes a just-in-time compiler) to run the bytecode.

**Section 2: Data Type**

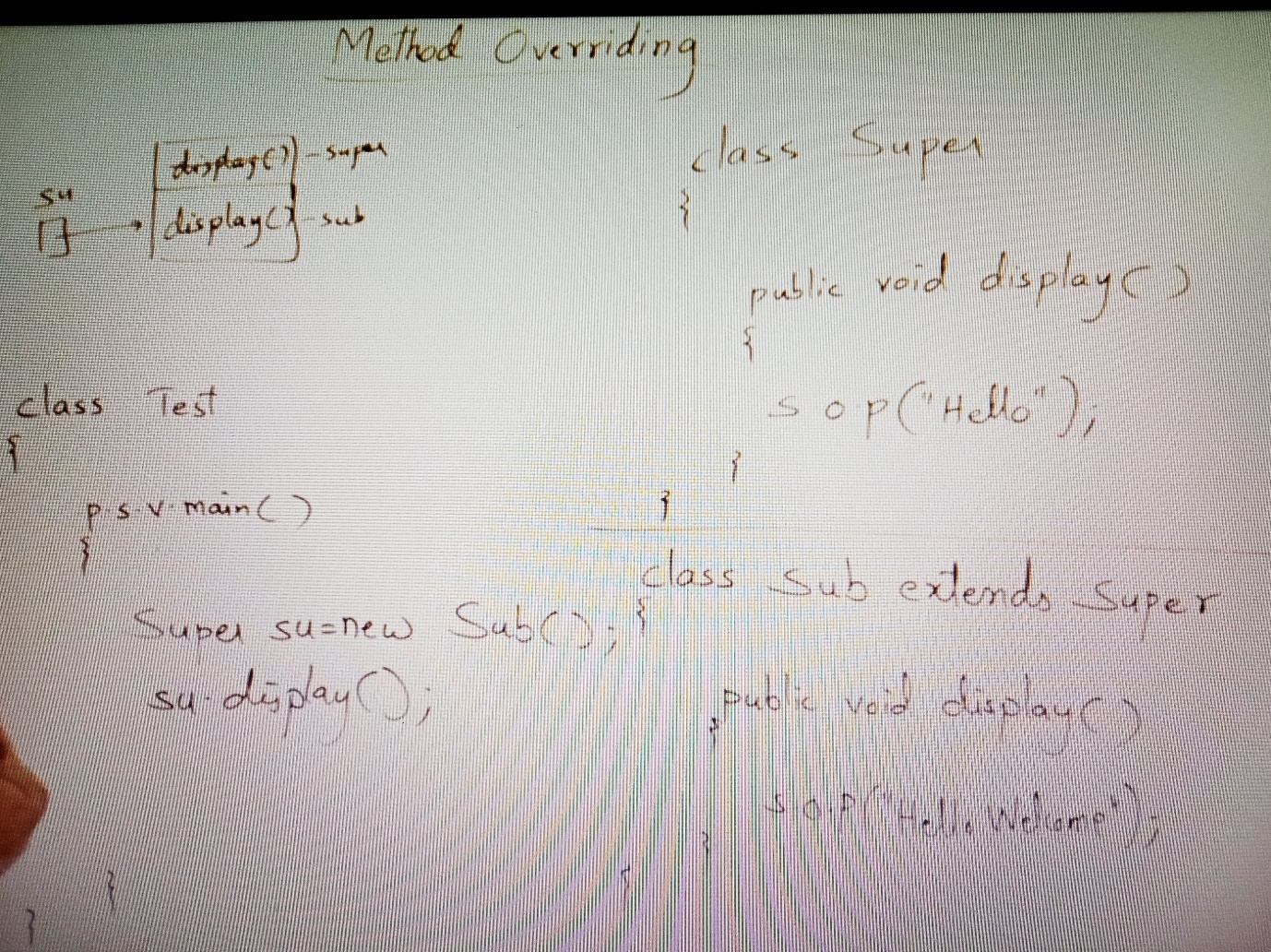
**Data Types:-**

A black background with text overlay

Description automatically generatedA black table with white text

Description automatically generated

**Dynamic Method Dispatch:-**



When creating a object if the reference is superclass and the object is subclass and a method is Overriding then the method of sub class is called.

When creating a object if the reference is superclass and the object is subclass we can call only the methods which are in the superclass or the method in the superclass extended or implemented by the subclass.

**Exception Handling: -**

Types of Errors:

Syntex Error -> Compiler

Logical Error -> Debugging

Runtime Error -> Exception Handling

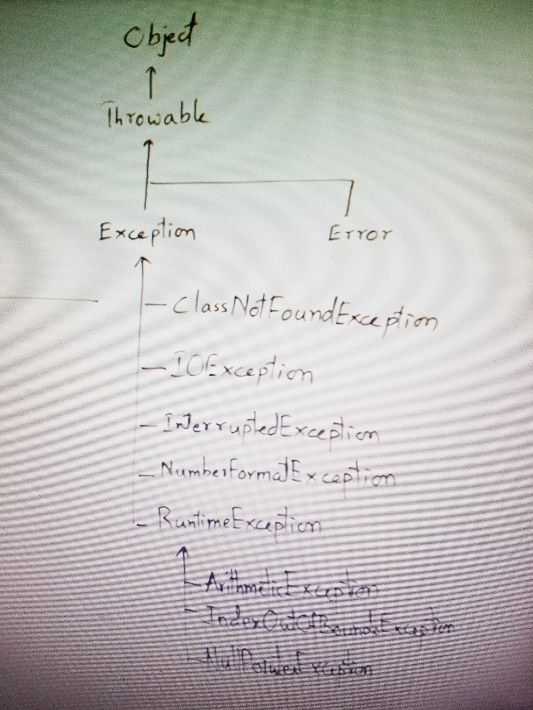
Run time errors are called Exception

Built in class in Exception:-

String getMessage() 🡪 sop(e.getMessage()) 🡪will print the exception message

String toString() 🡪 sop(toString)🡪Same as above

Void printStackTrace()



Checked Exception- must handle

Unchecked Exception- choice

**Final:-**

Final Variable:-

It cant be modified.

Final Methods:-

Cant be overridden

Final Class:-

Cant be extended

**Interface:-**

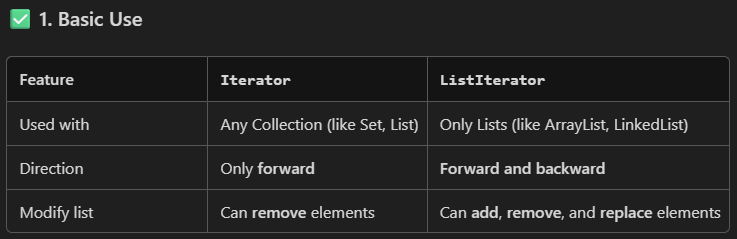
An **interface** in Java is a blueprint for a class that defines a set of methods (behaviors) that a class must implement. It is a way to enforce a contract that a class has to follow, without specifying how the methods should be implemented.

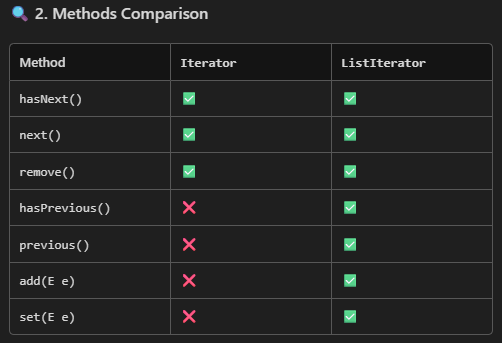
It is a class with all abstract methods

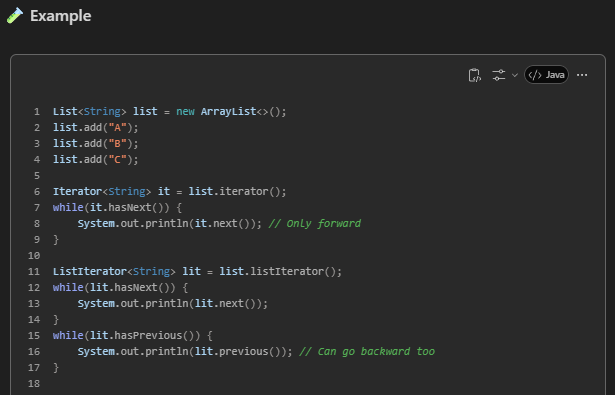
In Interface there is no method with body, to wright body we should declare it with **static** or **Default**.

**Iterator vs ListIterator**

Both Iterator and ListIterator are used to **traverse elements** in a collection (like a list), but they have **different capabilities**.

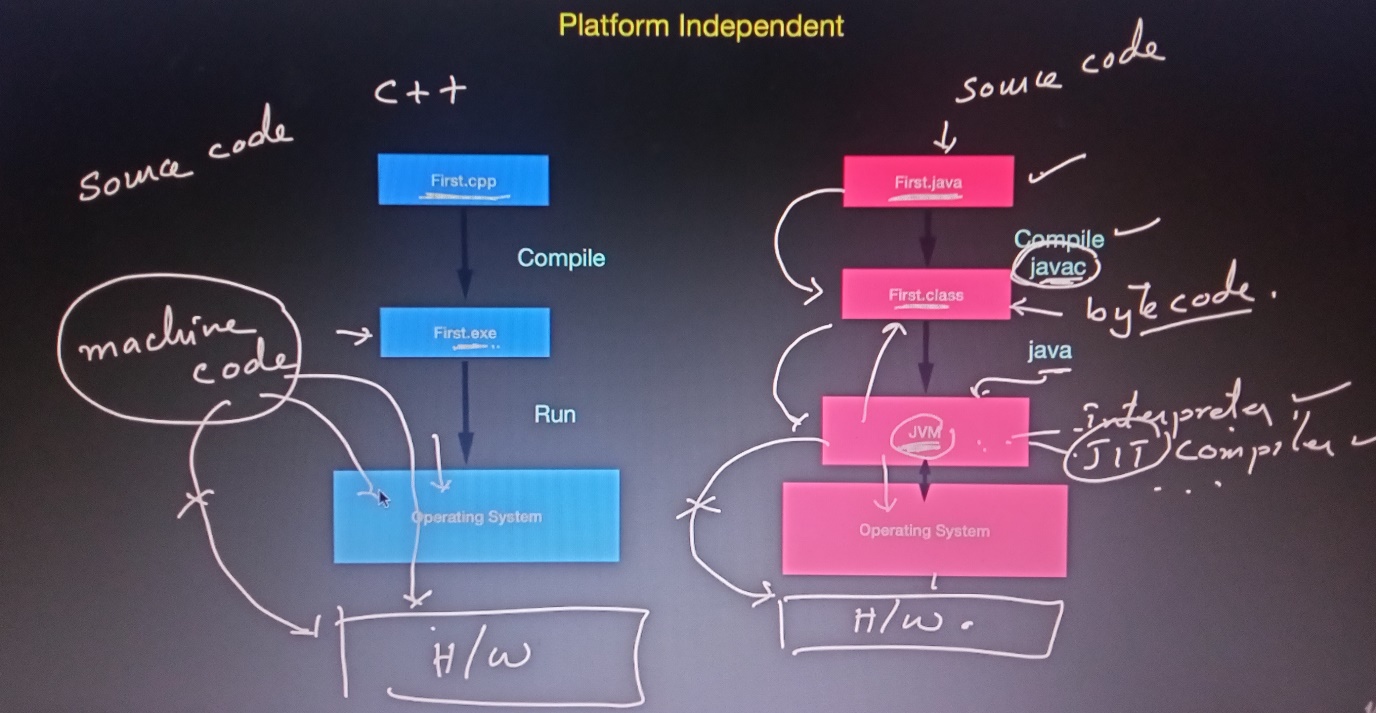






**Lambda Expression:-**

**Platform Independent:-**

****

**Static:-**

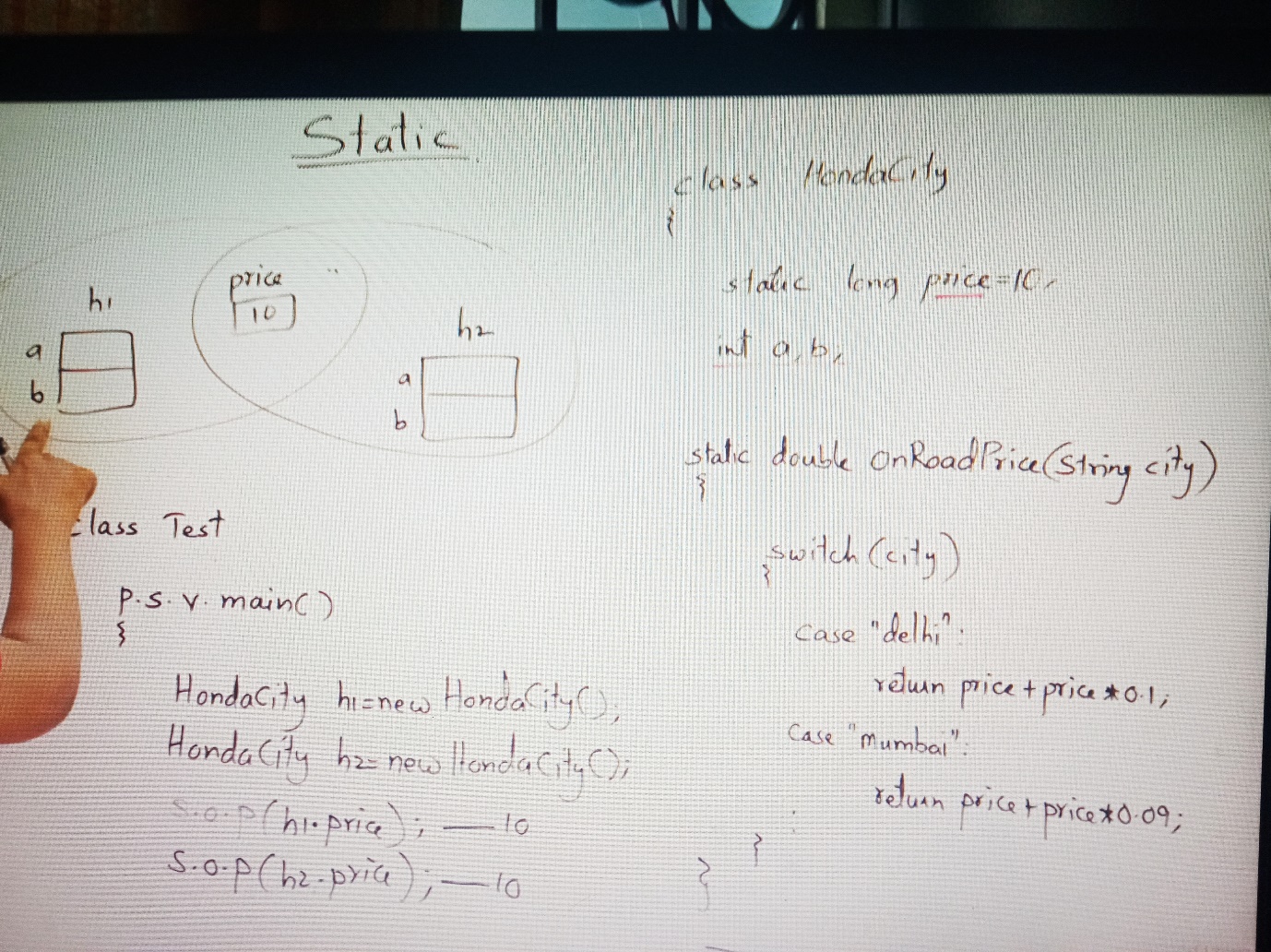
**In Variable-**

Static members belongs to the class.

Their could be a only one cops of a static variable.

Can call the variable without creating a object eg: HondaCity.price.

Created in the method area.

If its changed using one object (h1) the h2 will also hold the same value.

**In method-**

can call a method with out creating a object.

If we are calling a method “some” from a static method “main” the “some” method should also be static.

Can call staic variable from non static method.

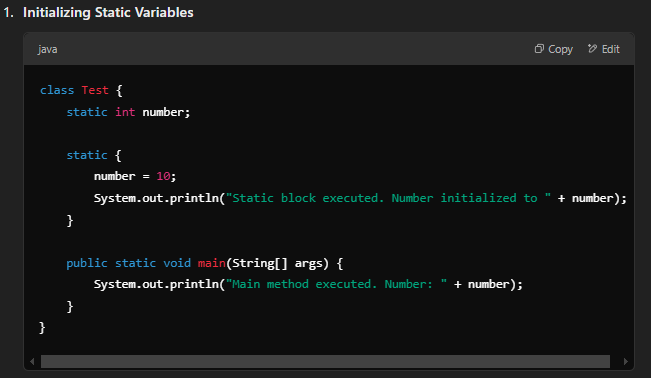
Cannot call non static variable from a static method.

Static members belongs to the class.

**Static Blocks-**

A **static block** in Java is a block of code that runs **once** when the class is **loaded into memory** by the Java ClassLoader. It is used to **initialize static variables** or **execute startup logic** before any object of the class is created.

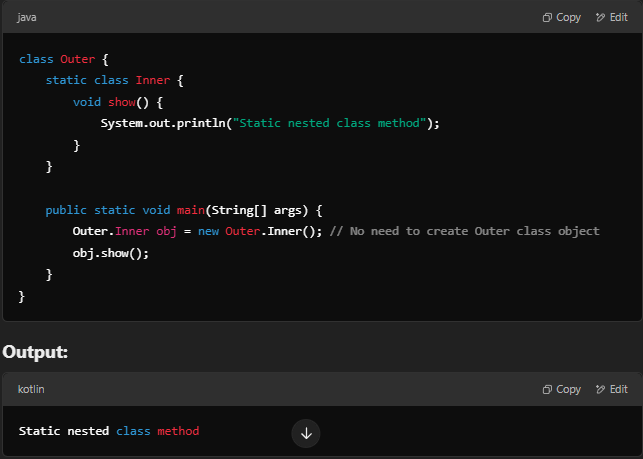
Can call only static members from static block

A static block will execute before the constructor

**Static Inner Class-**

In Java, you cannot make a top-level class static.  
 However, nested (inner) classes can be static.

A static nested class is a class declared inside another class with the static keyword.  
It does not require an object of the outer class to be created.



**String Pool:-**

String str1 = “hello world”;

“hello world” -> in string pool area

Str1-> pointing or holding this object.

String str2 =new String(“Java”);

new String(“Java”) -> constructor created in heap.

Str2 -> reference is holding the object.

Stack – Method Calls and Local Variables, Primitive variables, reference to objects, Metod execution.

Heap – objects, Arrays, Garbage Collection.

Metod area – Class info, Static Variables, Metod And Constructor code, JVM internals.