Week 1

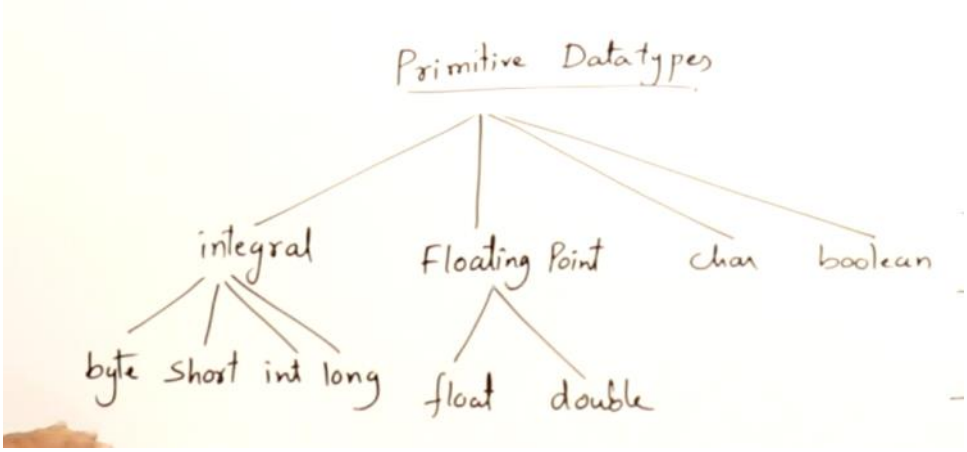
Monday, March 20, 2023 4:18 PM

Week 1:

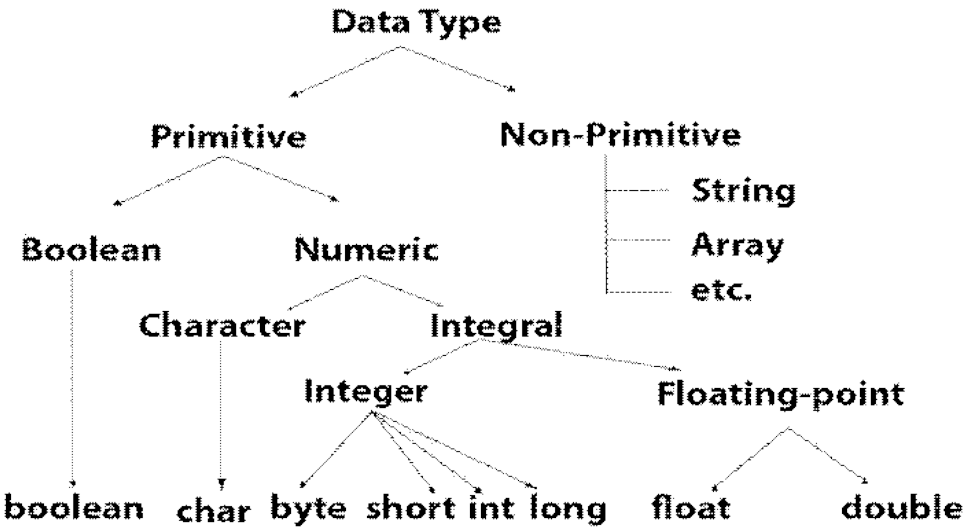
- OOPS concepts (Data Abstraction, Encapsulation, Inheritance, Polymorphism, Generalization, Composition, Aggregation etc.,)
- Basic Java constructs like Enum, loops and data types
- Classes & Objects
- String handling
- Exception handling
- Immutable Class

Data Types :

There are 8 primitive data types in Java.



- Integral -> Stores numerical without decimal point
- Floating point -> Stores decimal numerical
- Char -> For character
- Boolean -> For true or false



Datatypes

int char boolean

ouble

ASCII

Unicode

Type	size	Range	Default
byte	1	-128 to 127	0
short	2	-32768 to 32767	0
int	4	-2147483648 to -2147483647	0
long	8	—	0
float	4	±1.4E-45 to ±3.4E+38	0.0f
double	8	±4.9E-324 to ±1.7E+308	0.0d
char	2	0 to 65535	\u0000
boolean	?	true/false	false

Default value & size of Primitive data types :

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

- **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

- **int**: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc have been added to the Integer class to support the arithmetic operations for unsigned integers.

- **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. Use this data type when you need a range of values wider than those provided by int. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.

Variables :

Variables are **containers for storing data values**.
A variable is a name given to a memory location.
All of the variables provided by Java (other than the eight primitive variables mentioned above) are reference type.

Enum :

An enum is a special "class" that represents a group of constants
All enums implicitly extend java.lang.Enum. Because a class can only extend one parent, the Java language does not support multiple inheritance of state, and therefore an enum cannot extend anything else.
The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.

Enum vs Class :

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).
An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why enum :
Because it is constant and as it is a final class no class can extend it.

Static & instance block in enum :

https://stackoverflow.com/questions/11419519/enums-static-and-instance-blocks
Since enum values are static fields and they're placed before everything, so instance block followed by constructor for each enum values and only after that static block is executed.

OOPS :

It is a programming pattern that rounds around an object or entity are called **object-oriented programming.**

Class :

A class is a blueprint which defines properties and behavior of similar objects or entities.
It is an user-defined data type.
Class is like a template for creating object.

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Object :

An object is a real-world entity that has attributes, behavior, and properties. It is referred to as an instance of the class.

The state of an object is stored in fields (variables), while methods (functions) display the object's behavior.

Different ways of creating object :

1) **Using new keyword :**
   A obj = new A();

2) **Using newInstance() method :**
   If we know the name of the class & if it has a public default constructor we can create an object Class.forName. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn't create any Object. To create an Object of the Class you have to use the new Instance Method of the Class.

```
Class cls = Class.forName("GFG");
// Creating object of main class
// using instance method
GFG obj = (GFG)cls.newInstance();
```

3) **Using clone() method :**
   Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor. In order to use the clone() method on an object we need to implement Cloneable and define the clone() method in it.
   Note :
   - Here we are creating the clone of an existing Object and not any new Object.
   - Class need to implement Cloneable Interface otherwise it will throw CloneNotSupportedException.

```
class GFG implements Cloneable {

    // Method 1
    @Override
    protected Object clone()
        throws CloneNotSupportedException
    {
        // Super() keyword refers to parent class
        return super.clone();
    }
     GFG obj1 = new GFG();

        // Try block to check for exceptions
     try {

         // Using the clone() method
         GFG obj2 = (GFG)obj1.clone();

         // Print and display the main class object
        // as created above
         System.out.println(obj2.name);
        }
```

4) Deserialization
5) Using new instance on constructor
6) Unsafe class method

If we try print the reference to a object which is not initialized yet will print null.

Dynamic vs Static Memory Allocation :

Dynamic Memory Allocation -> Memory allocation to heap memory is done during runtime.

Static Memory Allocation -> Local variables, etc are allocated to a stack memory during compile time.

Dot . Operator :

Also called as Separator or Period or Member operator.
It denotes the separation of class from a package, separation of method from the class, and separation of a variable from a reference variable. It is also known as separator or period or member operator.

1. It is used to separate a variable and method from a reference variable.
2. It is also used to access classes and sub-packages from a package.
3. It is also used to access the member of a package or a class.

This keyword :

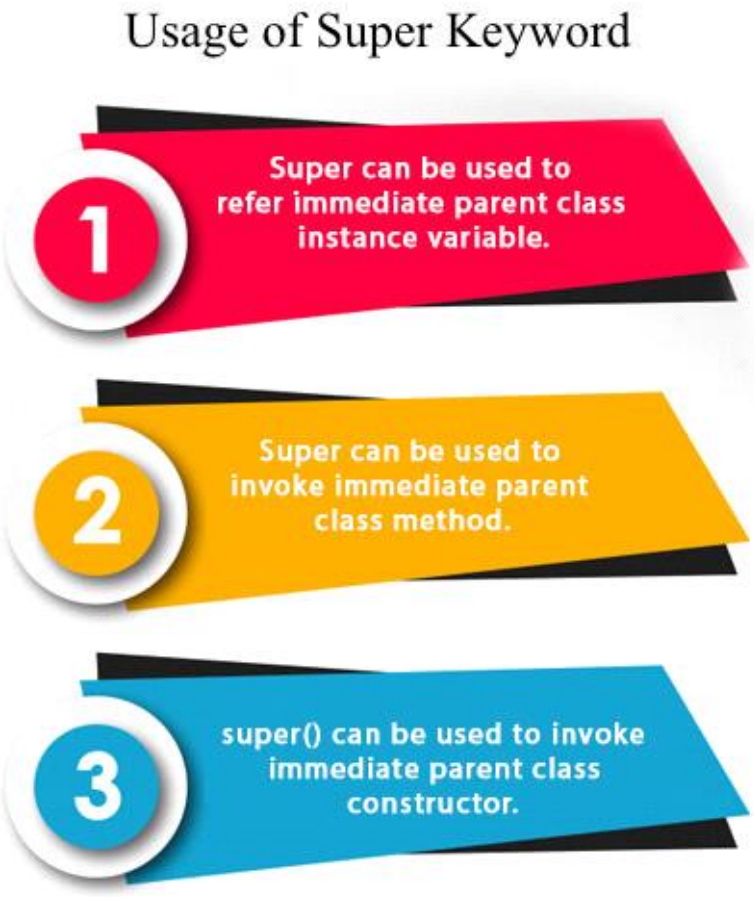
The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

this keyword is also used for calling one constructor from another constructor.
Example: this(rollNo, name);

https://www.geeksforgeeks.org/this-reference-in-java/

Super Keyword :

The super keyword in Java is a reference variable which is used to refer immediate parent class object.
Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.



Super vs This Keyword :

"super" and "this" in Java are two predefined keywords, that cannot be used as an identifier. "super" in Java is used to refer to methods, static and instance variables, constructors of an immediate parent class.
"this" in Java is used to refer to methods, static and instance variables, constructors of a current class.

Final Keyword :

- Used to declare constants.
- Convention is to use CAPITAL letters for variable names.
- If you declare a final variable later on you cannot modify or, assign values to it.
- Therefore, it is mandatory to initialize final variables once you declare them.

Static keyword :

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

3) **Java static block**

- Is used to initialize the static data member.
- It is executed before the main method at the time of class loading.

Instance Block :

Just like constructors, instance blocks can be used to initialize the instance variables during object creation.

If any instance block exists in the program, then for every object during its creation, first of all that instance block will be executed and then its specific constructor will be executed.

Because instance blocks are executed whenever the object of any kind is created, so if we want to write a logic which we want to execute on the creation of all kinds of objects, then using instance blocks is a good idea to avoid writing the same logic inside every constructor.

Garbage Collection :

Garbage collection happens in Java automatically.
We cannot manually destruct an object.
But we can specify what to actions to do when object is going to be taken away by garbage collector.
This can be done using finalize() method.

```
@Override
protected void finalize() throws Throwable {
    System.out.println("Object is destroyed");
}
```

Runtime vs Compile time polymorphism :

https://www.linkedin.com/pulse/why-method-overriding-called-runtime-polymorphism-java-omar-ismail/?trk=pulse-article

Encapsulation :

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

Achieved by making use of access modifiers. Private for data and Public for methods.

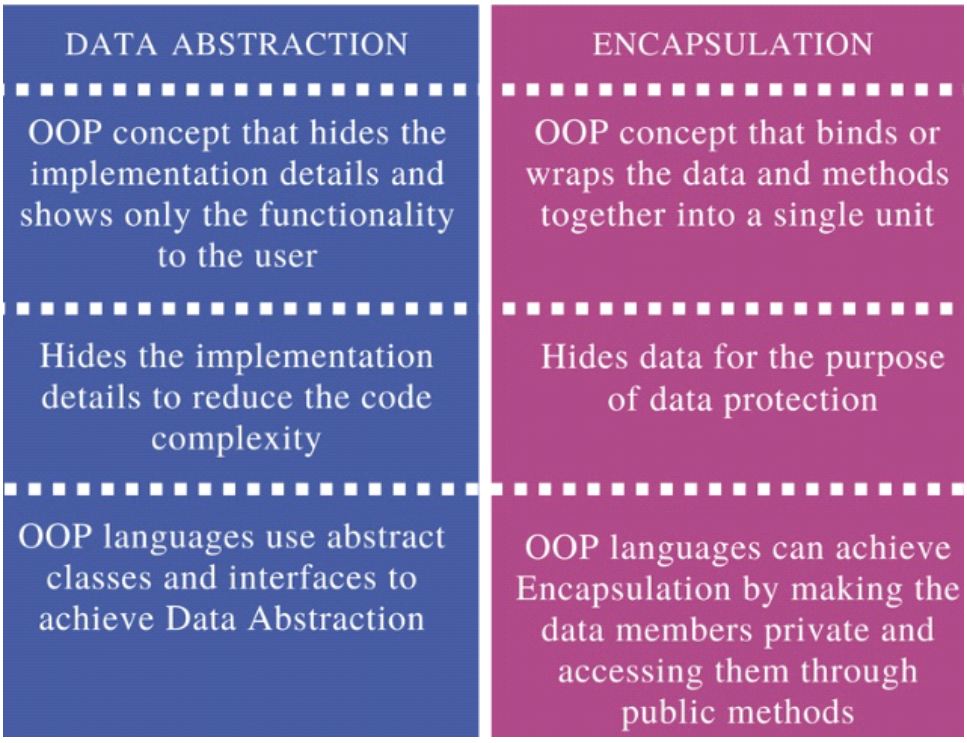
Access Modifiers :

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Abstraction :

Process of hiding certain details and showing only essential information to the user.
In Java, achieved by using Abstract class and Interface.
Interface achieves 100% abstraction.

Encapsulation vs Abstraction :



Marker Interface :

An interface that does not have any methods, fields, or constants, i.e, an empty interface in java is known as Marker or Tag Interface.
Why do we need Marker Interface :
The main use of the Marker Interface in Java is to convey to the JVM that the class implementing some interface of this category has to be granted some special behavior.
Example : When a class implements Cloneable Interface, then it indicates to the JVM that the objects of this class can be cloned.
https://www.scaler.com/topics/marker-interface-in-java/

Polymorphism :

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

1) Method Overloading -> Compile time polymorphism
2) Method Overriding -> Runtime polymorphism

Inheritance :

It is the mechanism by which one class inherits the features of another class i.e. the child class can use the fields and methods of parent class.
"extends" keyword is used in Java to implement inheritance.

Wrapper Class :

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Wrapper classes are fundamental in Java because they help a Java program be completely object-oriented. The primitive data types in java are not objects, by default. They need to be converted into objects using wrapper classes.

Need of Wrapper Classes :

- They convert primitive data types into objects. The classes in java.util package (Collections framework, etc.,) handles only objects and hence wrapper classes help in this case.
- An object is needed to support synchronization in multithreading.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Autoboxing :

The automatic conversion of primitive types to the object of their corresponding wrapper classes is

known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

```
char ch = 'a';
// Autoboxing- primitive to Character object
// conversion
Character a = ch;
```

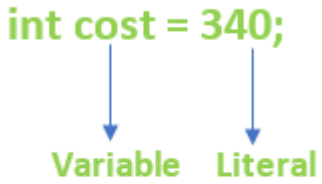
Unboxing :

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

```
Character ch = 'a';
// unboxing - Character object to primitive
// conversion
char a = ch;
```

Literals :

In Java, literals are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals.



Types of Literals in Java

There are the majorly four types of literals in Java:

1) Integer Literal
2) Character Literal
3) Boolean Literal
4) String Literal

In Java, literals are the fixed values or constants that are explicitly written in the source code of a program. For example, 1, 3.14, true, false, and "Hello" are all literals.

There are different types of literals in Java, including integer literals (e.g., 1, 10, -5), floating-point literals (e.g., 3.14, -2.5f, 6.02e23), character literals (e.g., 'a', '\n', '\'), string literals (e.g., "Hello", "world"), boolean literals (e.g., true, false), and null literals (e.g., null).

Literals represent values that are used in the program, and their values cannot be changed once the program is compiled and running.

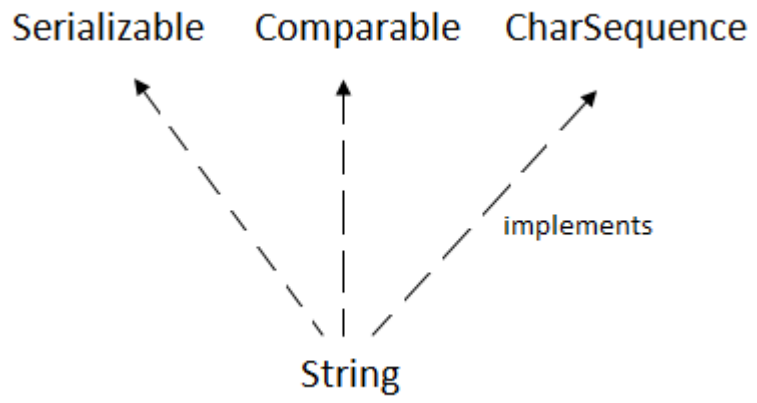
Strings :

string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.



CharSequence Interface :

The CharSequence interface is used to represent the sequence of characters.
String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.

BufferedReader :

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

BufferedReader(Reader rd)

```
FileReader fr=new FileReader("D:\\testout.txt");
BufferedReader br=new BufferedReader(fr);
```

`BufferedReader` buffers the input, just as the name says. This means that it reads from the input source into a buffer before passing it onto you. The buffer size here refers to the number of bytes it buffers.

Reading input from most sources is very slow. A buffer of just 2 bytes is going to hurt performance, as your program is very likely going to be waiting on input most of the time. With a buffer size of 2, a read of 100 bytes will result in reading 2 bytes from the in-memory buffer (very fast), filling the buffer (very slow), reading 2 bytes from the buffer (very fast), filling the buffer (very slow), etc - overall very slow. With a buffer size of 100, a read of 100 bytes will result in reading 100 bytes from the in-memory buffer (very fast) - overall very fast. This is assuming the buffer is contains the 100 bytes when reading though, which in a case like yours is a reasonable assumption to make.

Unless you know what you're doing, you should use the default buffer size which is quite large. One reason for a smaller buffer is when you are running on a limited-memory device, as the buffer consumes memory.

BufferedReader is synchronized (thread-safe) while Scanner is not.

String Constant Pool :

The String constant pool is a special memory area. When we declare a String literal, the JVM creates the object in the pool and stores its reference on the stack. Before creating each String object in memory, the JVM performs some steps to decrease the memory overhead.

StringBuffer :

A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

public StringBuffer() -> Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

Exception and Exception Handling :

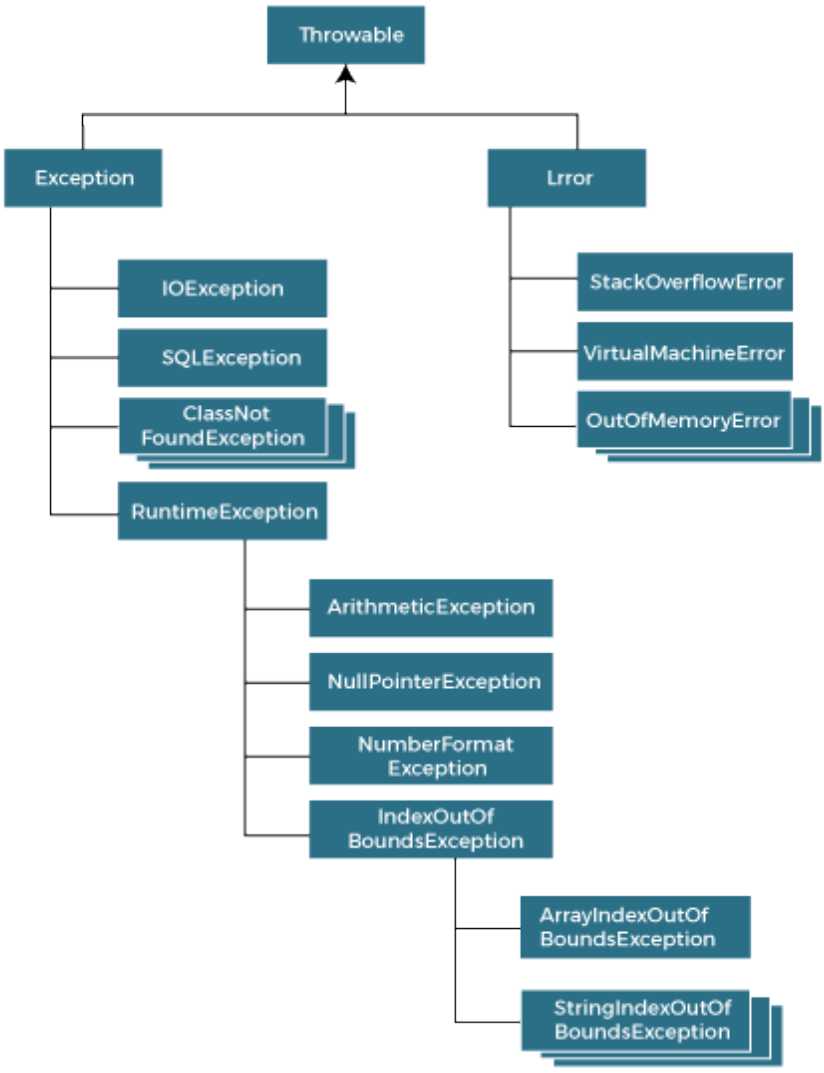
Exception :

An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. It stops the program execution.

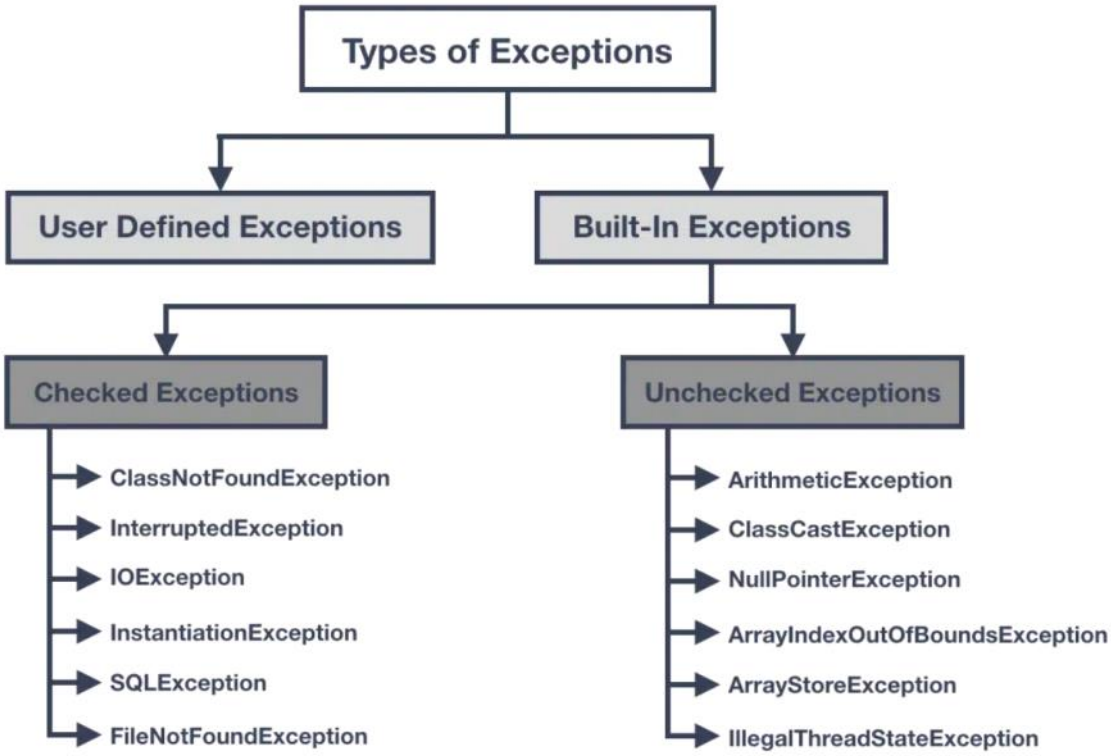
Exception Handling :

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.



Throwable is root class for Exception and Error class.
Throwable extends Object class.



Checked Exception or Compile time exception :
A checked exception is caught at compile time so if something throws a checked exception the compiler will enforce that you handle it.
All custom exceptions are checked exception.

Unchecked Exception or Runtime exception :
These are the exceptions that are not checked at compile time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception.

So to summarize; the difference between a checked and unchecked exception is that a checked exception is caught at compile time whereas a runtime or unchecked exception is, as it states, at runtime. A checked exception must be handled either by re-throwing or with a try catch block, a runtime isn't required to be handled. An unchecked exception is a programming error and are fatal, whereas a checked exception is an exception condition within your codes logic and can be recovered or retried from.

Important : -> Exception and inheritance

1) Derived class can throw Runtime exceptions
2) Derived class can throw Compile time or checked exception only if the parent class throws that exception.
3) Eg. If parent class throws IOException, child class can throw IOException or the exception below it in the hierarchy like FileNotFound exception i.e. any exception that extends that particular exception.

Error :
Error is irrecoverable. Some example of errors are StackOverflowError, VirtualMachineError, AssertionError etc.

Try With Resources :

Allows us to declare resources to be used in a try block with the assurance that the resources will be closed after the execution of that block.
The resources declared need to implement the AutoCloseable interface.

Immutable :

Wrapper class are used because they're immutable.
When searching for a bucket, if it is modified hashCode changes. So it should be immutable class.

Principles for making a class private :

To create an immutable class in Java, you need to follow these general principles:

1) Declare the class as final so it can't be extended.
2) Make all of the fields private so that direct access is not allowed.
3) Don't provide setter methods for variables.
4) Make all mutable fields final so that a field's value can be assigned only once.
5) Initialize all fields using a constructor method performing deep copy.
6) Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

Creating a Clone of an object :

1) To create a clone we have to have clone() method.
2) To have clone method we need to implement Cloneable interface.
3) If tried to create a clone() method without implementing Cloneable interface it will throw CloneNotSupportedException.

Creating clone of Date object :

```
public Date getDate() {
    return new Date(date.getTime());
}
```

Why do we need immutable class :

Since the internal state of an immutable object remains constant in time, we can share it safely among multiple threads.
We can also use it freely, and none of the objects referencing it will notice any difference, we can say that immutable objects are side-effects free.

Pass by value & Pass by reference :

In Java, primitive data types are passed by value and all non-primitives (or objects of any class) are passed by reference.

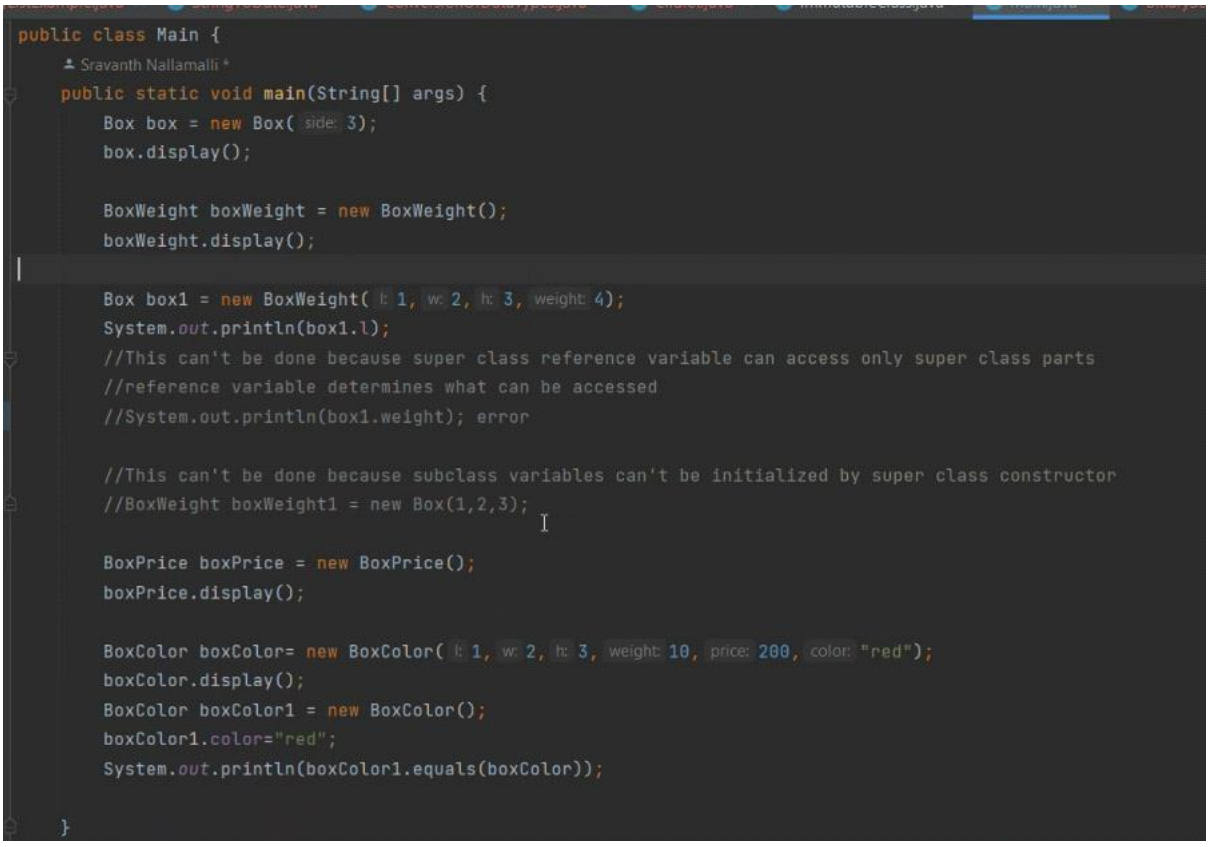
== vs equals() :

Both equals() method and the == operator are used to compare two objects in Java. == is an operator and equals() is method. But == operator compares reference or memory location of objects in a heap, whether they point to the same location or not.
In Java, the String equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are the same, it returns true. If all characters are not matched, then it returns false.
The default implementation of equals method is defined in Java.lang.Object class which simply checks if two Object references (say x and y) refer to the same Object.

hashCode() :

The hashCode() method in Java is used to compute hash values of Java objects.

https://stackoverflow.com/questions/41635837/initializer-block-inside-the-main-method



```
public class Main {
    Sravanth Nallamalli *
    public static void main(String[] args) {
        Box box = new Box( side: 3);
        box.display();

        BoxWeight boxWeight = new BoxWeight();
        boxWeight.display();

        Box box1 = new BoxWeight( l: 1, w: 2, h: 3, weight: 4);
        System.out.println(box1.l);
        //This can't be done because super class reference variable can access only super class parts
        //reference variable determines what can be accessed
        //System.out.println(box1.weight); error

        //This can't be done because subclass variables can't be initialized by super class constructor
        //BoxWeight boxWeight1 = new Box(1,2,3);

        BoxPrice boxPrice = new BoxPrice();
        boxPrice.display();

        BoxColor boxColor= new BoxColor( l: 1, w: 2, h: 3, weight: 10, price: 200, color: "red");
        boxColor.display();
        BoxColor boxColor1 = new BoxColor();
        boxColor1.color="red";
        System.out.println(boxColor1.equals(boxColor));

    }
```

Type Casting :
1) Upcasting
Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. Upcasting is also known as Generalization and Widening.

Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the **"ClassCastException"**. Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

Static, final & class loading :

The compiler optimizes inlineable static final fields by embedding the value in the bytecode instead of computing the value at runtime.
When you fire up a JVM and load a class for the first time (this is done by the classloader when the class is first referenced in any way) any static blocks or fields are 'loaded' into the JVM and become accessible.
https://stackoverflow.com/questions/4343760/when-is-static-variable-loaded-in-java-runtime-or-compile-time