1. Threading
2. JRE JDK JVM

JVM :

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each [OS](https://www.javatpoint.com/os-tutorial) is different from each other. However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

### JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

### JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and [applets](https://www.javatpoint.com/java-applet). It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



1. JDBC

In Java, both Statement and PreparedStatement are used to execute SQL queries against a database. However, they differ significantly in terms of functionality, performance, and security. Here are the key differences:

**1. SQL Injection Protection**

* **Statement**: Vulnerable to SQL injection attacks because it directly interpolates user input into the SQL query.
* **PreparedStatement**: Protects against SQL injection by using placeholders (?) for parameters and properly escaping inputs.

**2. Performance**

* **Statement**: Compiles the SQL query every time it is executed, which can be slower for frequently executed queries.
* **PreparedStatement**: Pre-compiles the SQL query once and can reuse the compiled query with different parameters, leading to better performance for repeated executions.

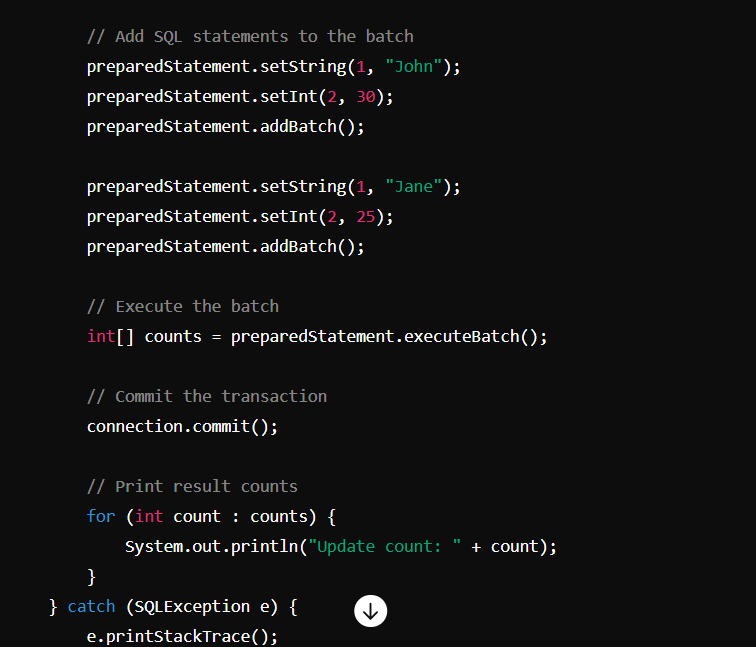
**3. Ease of Use**

* **Statement**: Requires manual concatenation of query strings, which can be error-prone and cumbersome, especially for complex queries.
* **PreparedStatement**: Simplifies setting parameters using setter methods like setInt(), setString(), etc., which makes code easier to read and maintain.

**4. Type Safety**

* **Statement**: No type safety, as all values are concatenated as strings.
* **PreparedStatement**: Provides type safety by allowing parameter values to be set with specific types (int, string, etc.).

**5. Batch Updates**

* **Statement**: Supports batch updates but less efficiently compared to PreparedStatement.
* **PreparedStatement**: More efficient for batch updates due to pre-compilation and parameterized queries.
* 

1. Collections
2. OOPS
3. Singleton
4. Java 8 features
5. Comparator vs comparable
6. Files
7. Synchronization vs Asynchronization

 \*\*Synchronous Programming:\*\* Use when tasks are dependent on each other, and the program logic requires a step-by-step execution. - \*\*Asynchronous Programming:\*\* Use when tasks can be executed independently, and you want to avoid blocking the program while waiting for potentially time-consuming operations

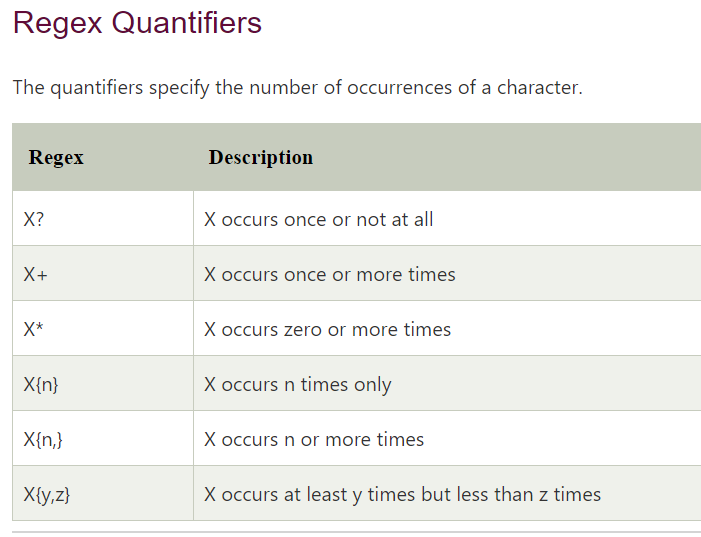
1. StringBuilder vs String Buffer

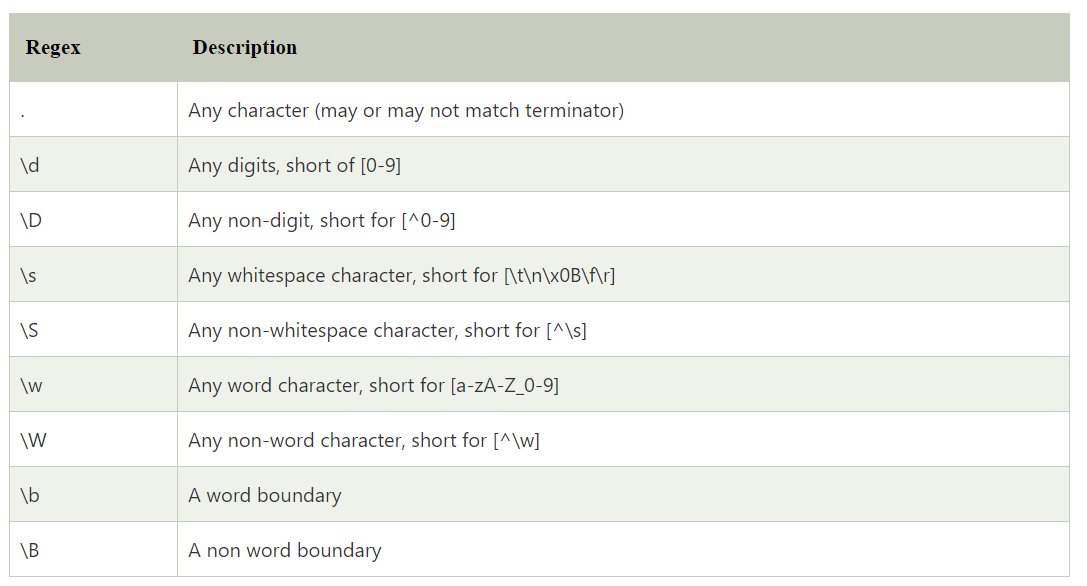
StringBuffer is thread safe

1. Equals and == (Heap memory, String pool)

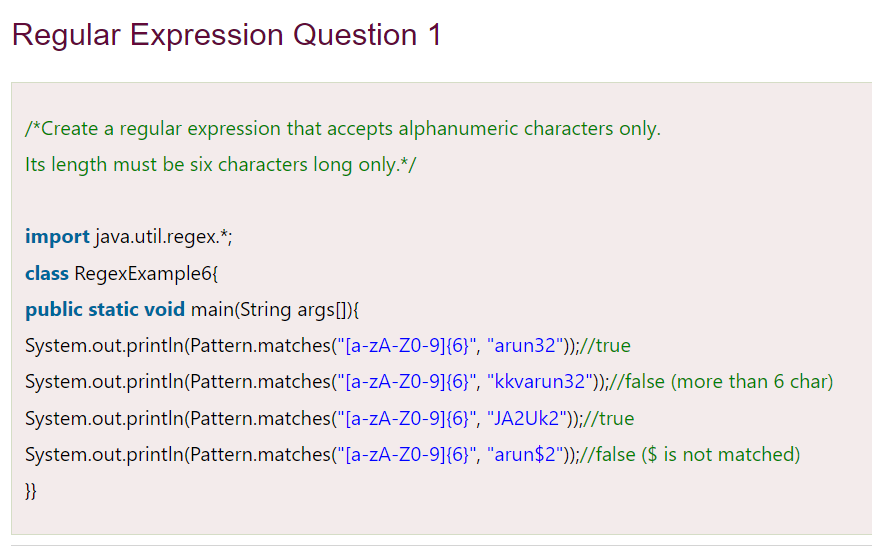
In Java, == is an operator that compares the references or memory addresses of objects to determine if they are the same, whereas . equals() is a method that compares the contents of the objects to check for value equality.

1. Date Time
2. Java Regular Expression





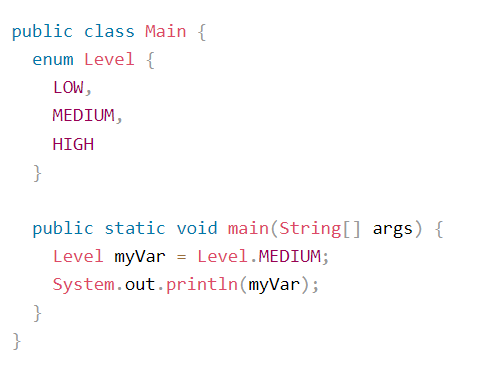
Example :



1. Exception Handling

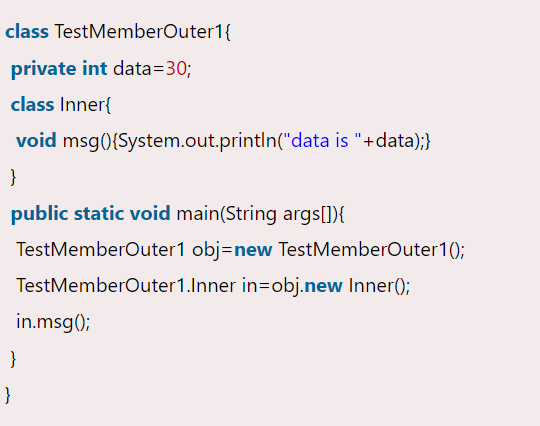
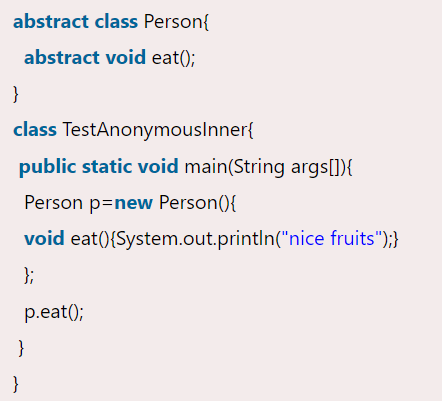
Try, catch, throws, throw, finally

1. Enum

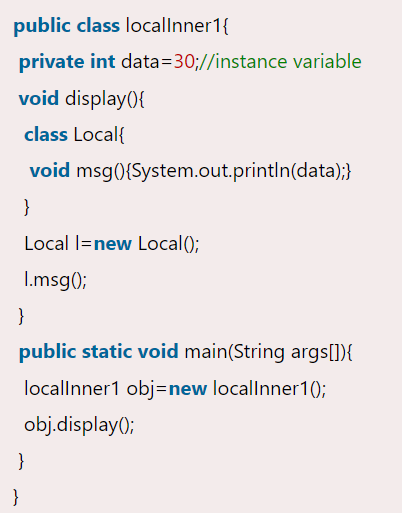
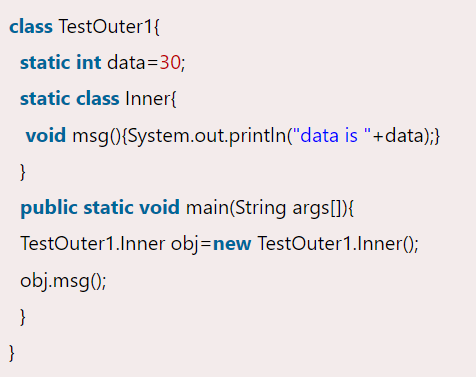


1. Inner class

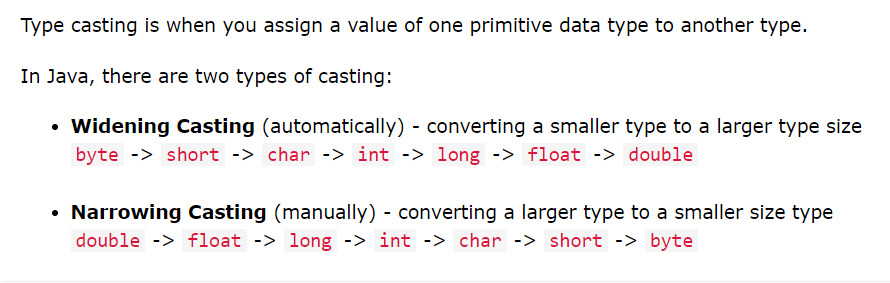
Member inner class Anonymous Inner Class

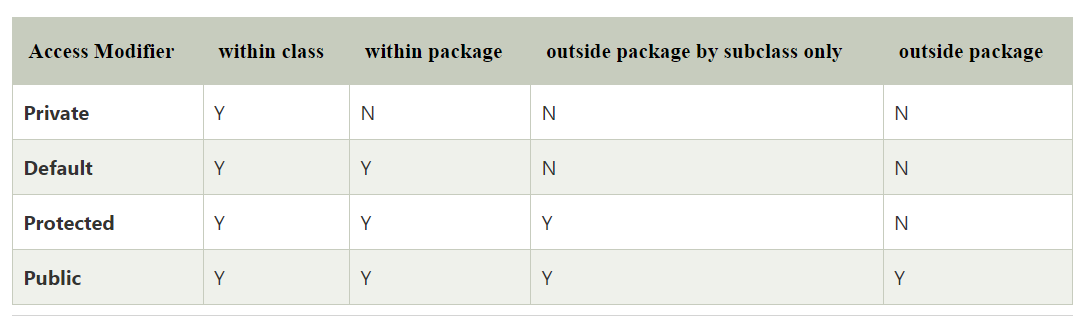
Local Inner Class Static Inner class

1. Typecasting



1. Access modifiers



1. Java IO
2. Primitive vs Non primitive

String, Arrays, Classes – non primitive

1. Wrapper class Auto Boxing vs Un Boxing

[Autoboxing](https://www.geeksforgeeks.org/autoboxing-unboxing-java/) refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class.

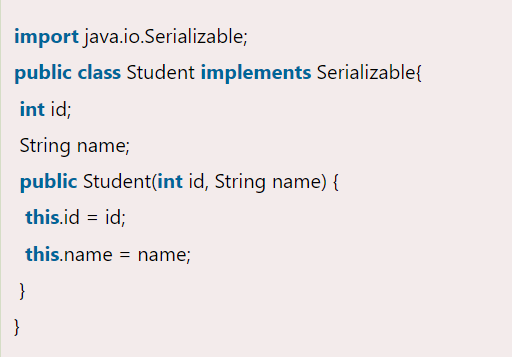
**Unboxing**on the other hand refers to converting an object of a wrapper type to its corresponding primitive value.

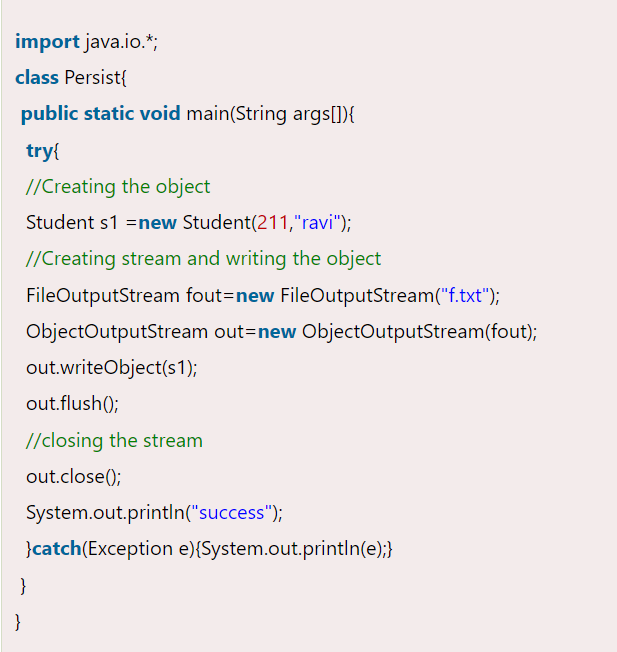
1. Design Pattern – Refer another doc
2. Serialization

Serialization is the process of converting an object into a stream of bytes, which can then be stored in a file, sent over a network, or persisted in a database. The reverse process, i.e., reconstructing the object from the serialized bytes, is called deserialization. Serialization is primarily used for data persistence and communication between different Java applications or systems.

In Java, serialization is achieved by implementing the **Serializable** interface. Here's an example:

Student.java





Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

In Java, the **flush()** method is used in the context of streams, such as output streams or buffered streams. Here's a simple explanation:

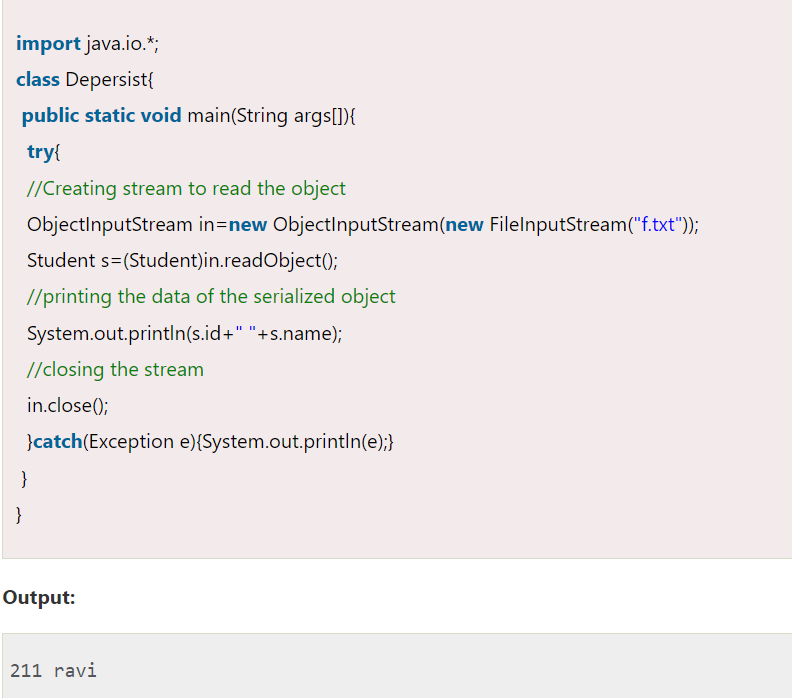
When you write data to a stream (like writing to a file or sending data over a network), Java doesn't immediately send or write the data out. Instead, it stores the data temporarily in a buffer. This buffering helps improve performance because writing data to a file or network can be slow compared to writing to memory.

Now, the **flush()** method comes into play. When you call **flush()** on a stream, you're telling Java to empty that buffer and send out any data that's been stored there immediately. It's like saying, "Okay, I'm done for now. Send out whatever is in the buffer."

Why would you want to do this? Well, sometimes you might want to make sure that data is sent or written right away, instead of waiting for the buffer to fill up naturally. For example, if you're sending data over a network connection and you want to ensure that the other end receives the data promptly, you might call **flush()** to force it to be sent immediately.

In summary, **flush()** is a method used with streams in Java to force any buffered data to be sent or written immediately, instead of waiting for the buffer to fill up naturally. It's handy when you want to ensure data is sent or written promptly.

Top of Form



HashCode vs Equals :

In Java, hashCode and equals are two important methods inherited from the Object class. They are fundamental for objects' comparison and for the proper functioning of hash-based collections like HashMap and HashSet.

**equals Method**

The equals method is used to compare two objects for equality. By default, the equals method inherited from Object compares the memory addresses of the objects, i.e., it checks for reference equality. However, this method can be overridden to compare objects based on their content.

**hashCode Method**

The hashCode method returns an integer hash code value for the object. This value is used by hash-based collections. The general contract of hashCode is:

* Whenever hashCode is invoked on the same object more than once during an execution of a Java application, it must consistently return the same integer, provided no information used in equals comparisons on the object is modified.
* If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
* It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.
* Override equals to compare objects based on their content.
* Override hashCode to ensure that equal objects have the same hash code.
* Always override hashCode if you override equals to maintain the general contract.

Properly implementing hashCode and equals ensures correct behavior in collections and avoids bugs related to object equality and hashing.

Why Strings are immutable

Strings are immutable in many programming languages, including Java, Python, and C#. There are several reasons why strings are designed to be immutable:

1. \*Thread Safety\*: Immutability ensures that strings cannot be modified after creation. This property makes strings inherently thread-safe because multiple threads can safely read the same string without worrying about concurrent modifications.

2. \*Caching\*: Since strings are immutable, they can be cached and reused. This optimization can improve performance and reduce memory usage, especially in scenarios where the same string is used multiple times.

3. \*Security\*: Immutable strings prevent unintended modifications, which can be crucial in security-sensitive applications. For example, if strings representing sensitive data (such as passwords or cryptographic keys) were mutable, they could be inadvertently modified, compromising security.

4. \*Hashing\*: Immutable strings allow for efficient hashing and comparison operations. Once a string is hashed, its hash value remains constant, simplifying hash table implementations and string comparison algorithms.

5. \*Predictability\*: Immutability leads to predictable behavior. Since strings cannot change, developers can rely on their values remaining constant throughout the program's execution, which reduces complexity and potential bugs.

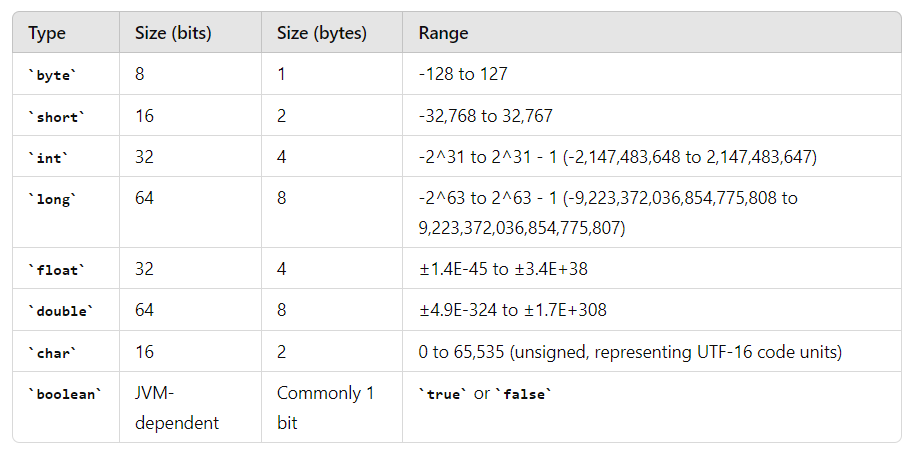
6. \*Optimizations\*: Immutable strings enable compiler and runtime optimizations. For example, string interning techniques can be used to optimize memory usage by reusing identical string literals.

Overall, immutability provides several benefits in terms of performance, thread safety, security, and predictability, making it a common design choice for strings in many programming languages.

**Stack vs heap memory :**

Stack memory is a sort of memory allocation that the OS continuously manages and uses to store local variables in a LIFO order. On the other hand, heap memory is a type of dynamic memory allocation used for storing objects and data structures that require a longer lifespan than stack memory.

**Primitive data types :**

****

**UTF -16 :**

Unicode is a universal character encoding standard that assigns a unique numeric value to each character in many of the world's writing systems. UTF-16 (16-bit Unicode Transformation Format) is one of the encoding methods used to represent Unicode characters.