# What is the difference between usual String and StringBuilder?

**Immutability**:

**String**: Strings in Java are immutable, which means that once a **String** object is created, its content cannot be changed. Any operation that appears to modify a **String** actually creates a new **String** object with the modified content. This immutability is a key characteristic of **String** objects.

**StringBuilder**: **StringBuilder** is mutable. It represents a sequence of characters that can be modified without creating a new object. You can append, insert, replace, or delete characters from a **StringBuilder** without creating new instances.

When we explicitly create a **String** object using **new** keyword, an object is created in the heap.

When you create a **StringBuilder** object using the **new** keyword, an object is created in the heap memory:

# For what are used marker interfaces?

Marker interfaces, also known as "tagging interfaces" in Java, are interfaces that don't declare any methods but are used to indicate a certain capability or attribute of a class. They serve as markers or flags to inform the compiler or runtime environment about specific characteristics of the class that implements them. Here are some common uses of marker interfaces in Java:

**Serialization:** The most famous marker interface in Java is **java.io.Serializable**. When a class implements this interface, it signals that instances of that class can be serialized, meaning they can be converted into a stream of bytes and later deserialized to reconstruct the object. The Java serialization framework uses this marker interface to determine whether an object can be serialized.

**Cloning:** Another marker interface is **java.lang.Cloneable**. When a class implements this interface, it indicates that instances of that class can be cloned using the **Object.clone()** method. It's important to note that using **clone()** correctly requires deep copying of object contents, and not all classes support it.

**Custom Annotations:** In some cases, marker interfaces are used as part of custom annotation processing. A custom annotation might specify that annotated classes must implement a specific marker interface to enable some behavior or functionality.

1. **Runtime Behavior:** Marker interfaces can also be used to affect the runtime behavior of classes. For example, some frameworks or libraries may use marker interfaces to identify classes that should be treated differently during certain operations.
2. **Documentation and Self-Documentation:** Marker interfaces can serve as a form of documentation, indicating the intended use or constraints of a class. Developers can see that a class implements a specific marker interface and understand its implications.

The difference between Iterable and Iterator**.**

In Java, **Iterator** and **Iterable** are related concepts used for working with collections, but they serve different purposes:

1. **Iterable:**
   * **Iterable** is an interface in the **java.lang** package.
   * It's the higher-level interface that represents a collection of elements that can be iterated (looped) over.
   * The primary purpose of **Iterable** is to provide a way to obtain an **Iterator** for the collection.
   * It contains a single abstract method, **iterator()**, which returns an **Iterator** for the implementing class.
   * Implementing the **Iterable** interface allows an object to be used in enhanced for loops (**for-each** loops).

Example of implementing **Iterable**:

public class MyIterableCollection<T> implements Iterable<T> {

// Implement required methods

public Iterator<T> iterator() {

// Return an Iterator for this collection

return new MyIterator<T>();

}

}

**Iterator:**

* **Iterator** is also an interface but in the **java.util** package.
* It represents an object that can iterate (loop) through elements of a collection sequentially.
* An **Iterator** provides methods like **hasNext()** to check if there are more elements, and **next()** to retrieve the next element.
* The **Iterator** interface is used for iterating over elements of a collection, especially in a way that hides the underlying data structure.

Example of using **Iterator**:

List<String> myList = new ArrayList<>();

myList.add("A");

myList.add("B");

myList.add("C");

Iterator<String> iterator = myList.iterator();

while (iterator.hasNext()) {

String element = iterator.next();

System.out.println(element);

}

# Lambda Expressions

Lambda expressions are a feature introduced in Java 8 that allow you to write concise and expressive code for defining and implementing functional interfaces (interfaces with a single abstract method) directly inline.

# Sockets

In Java, ServerSocket and Socket are two classes used for creating network applications, specifically for implementing communication between client and server applications using sockets. However, they serve different purposes in the network communication process:

**ServerSocket:**

**Purpose:** ServerSocket is used on the server side of a network application. It listens for incoming client connection requests and creates a new Socket for each accepted connection. It allows a server to accept multiple client connections simultaneously.

**Usage:** You create a ServerSocket object on the server side to listen on a specific port for incoming connection requests.

**Example**

ServerSocket serverSocket = new ServerSocket(8080);

Socket clientSocket = serverSocket.accept(); // Accept a client connection

**Socket:**

**Purpose:** Socket is used on both the client and server sides. On the client side, it's used to initiate a connection to the server. On the server side, it's used to represent an established connection to a client. Socket allows bidirectional communication between the client and server.

**Usage:** On the client side, you create a Socket object to connect to a specific server IP address and port. On the server side, after accepting a connection with ServerSocket, a new Socket object is created to manage the communication with the connected client.

Socket clientSocket = new Socket("serverIpAddress", 8080);

In summary, **ServerSocket** is used on the server side to listen for incoming connections, while **Socket** is used on both the client and server sides to establish and manage connections for data exchange. They are fundamental classes for building networked applications in Java.

**Creation:** An instance of Socket is created either by the server application to listen for incoming data or by the client application to initiate a connection.

**•Connection:** Sockets are then connected to a remote address and port, allowing data transmission.

•**Communication:** Data can be read from or written to the socket.

•**Closure:** After communication is done, sockets should be closed to free up resources.

# JRE JDK JVM

The JVM (Java Virtual Machine), JDK (Java Development Kit), and JRE (Java Runtime Environment) are key components of the Java platform, each serving distinct purposes:

1. **JVM (Java Virtual Machine):**
   * **Primary Function:** The JVM is an integral part of the Java platform responsible for executing Java bytecode. It provides a runtime environment for Java applications to run on various hardware and operating systems without modification.
   * **Key Tasks:**
     + Loading: It loads Java bytecode files (class files) into memory.
     + Verification: It ensures that the loaded bytecode is valid and secure.
     + Execution: It interprets and/or compiles bytecode into native machine code.
     + Memory Management: It manages memory allocation and garbage collection.
     + Security: It enforces security measures such as access control and sandboxing.
     + JIT Compilation: In some cases, it just-in-time (JIT) compiles bytecode for improved performance.
     + Platform Independence: It abstracts hardware-specific details, making Java applications platform-independent.
2. **JDK (Java Development Kit):**
   * **Primary Function:** The JDK is a software development kit used by Java developers to create, compile, and package Java applications and applets.
   * **Key Components:**
     + Compiler: The JDK includes the Java compiler (**javac**) for compiling Java source code into bytecode.
     + Libraries: It provides the standard Java class libraries, APIs, and development tools.
     + Debugger: Debugging tools like **jdb** are included for troubleshooting code.
     + Documentation: It includes documentation, such as the Java API documentation.
     + JRE: The JDK includes a minimal JRE, allowing developers to run their applications during development.
3. **JRE (Java Runtime Environment):**
   * **Primary Function:** The JRE is a runtime environment required to execute compiled Java applications and applets.
   * **Key Components:**
     + JVM: The JRE includes the JVM, which interprets or compiles bytecode into machine code.
     + Standard Class Libraries: It contains the Java Standard Library (Java API), which provides essential classes and packages for common tasks.
     + Supporting Files: Necessary files and configurations for running Java applications.

In summary, the JVM executes Java bytecode, the JDK is used for Java software development, including compiling and packaging, and the JRE is essential for running compiled Java applications on end-user machines. Developers use the JDK to create Java applications, and end-users need the JRE to execute those applications.

Java.lang contains Object, Wrapper classes

We cannot use float, double and long in switch because there is strict accuracy

float f = 0.1f;  
double d = 0.1;  
System.*out*.println(f==d);

The output of this code is false

Wrapper classes are immutable, when we want to change them they create a new object

Dimond operator can contain the array of primitive types

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

* Meta space, integer pool,
* Which is the limit of the integer pool.
* What is class loader and the levels of it  
  loading, linking, initialization
* For what is class path used, where we can use it.
* Wrapper classes

We cannot cast one wrapper class to another

Casts

* Nested classes,
* We cannot write condition in the constructor and call other constructer after it, because the call of the other constructor must be on the first line.
* Access modifiers
* Static dynamic polymorphism
* UML
* Exceptions
* Can we catch errors?
* Tree of collections (sets work on the maps)
* Differences of array list and linked list, performance measures
* Anonymous classes
* Functional interfaces, Marker Interfaces (empty) -> Serializable, Cloneable
* Difference between abstract class and interface (access modifiers)
* Stream API, functions, operation types (intermediate, terminal), can we use intermediate without terminal, examples
* Files
* Annotation (retention, target) types and usage
* Thread, wait, sleep, notify, notify all, interrupt, synchronize, object monitor, thread states

Difference between abstract class and interface.

**Abstract Class:**

1. **Definition**: An abstract class is a class declared with the **abstract** keyword. It can have a combination of abstract (unimplemented) methods, concrete (implemented) methods, instance variables, and constructors.
2. **Methods**: It can contain both abstract and concrete methods. Abstract methods are declared with the **abstract** keyword and have no implementation within the abstract class. Concrete methods have full implementations.
3. **Constructors**: Abstract classes can have constructors, and they are typically used for initializing instance variables and performing common tasks.
4. **Inheritance**: A Java class can extend only one abstract class. This means that if you extend an abstract class, you cannot extend any other class (abstract or not).
5. **Fields**: Abstract classes can have fields (variables) that can be of any access level (public, private, protected, package-private).
6. **Access Modifiers**: Abstract classes can have access modifiers for methods and fields, allowing you to specify visibility and control access to them.

**Interface:**

1. **Definition**: An interface is a contract that defines a set of abstract methods that a class must implement. In Java, interfaces are declared using the **interface** keyword.
2. **Methods**: Interfaces can only declare abstract methods. These methods have no implementation within the interface and are implicitly **public** and **abstract**.
3. **Constructors**: Interfaces cannot have constructors because they cannot be instantiated. They are purely for defining method contracts.
4. **Inheritance**: A Java class can implement multiple interfaces. This allows a class to inherit the method contracts of multiple interfaces, enabling multiple inheritance of behavior.
5. **Fields**: Interfaces can declare constants (public, static, and final variables), but they cannot have instance variables with non-constant values.
6. **Access Modifiers**: All methods declared in an interface are implicitly **public**. Fields declared in an interface are implicitly **public**, **static**, and **final**.

In summary, abstract classes are used when you want to provide a common base class with some shared behavior and allow derived classes to extend it. Interfaces are used when you want to define a contract that multiple classes can implement, enabling polymorphism and multiple inheritance of method contracts.

**does metaspace located in the heap?**

ChatGPT

No, the Metaspace is not located in the Java heap. Metaspace is a separate memory area in the Java Virtual Machine (JVM) that is used to store class metadata, such as class definitions, method definitions, field definitions, and other information related to the structure of loaded classes.

Here are some key points about Metaspace:

1. **Separate from the Heap**: Metaspace is distinct from the Java heap, which is used for storing objects created by your Java program. The heap is where instances of classes (objects) are allocated and managed.
2. **Class Metadata Storage**: Metaspace is primarily used to store class metadata. This includes information about the classes and methods loaded by the JVM, as well as method bytecode.
3. **Dynamically Sized**: Unlike the older PermGen space that it replaced (in pre-Java 8 JVMs), Metaspace is dynamically sized. It can grow or shrink based on the requirements of the loaded classes and metadata. This helps prevent issues like "OutOfMemoryError: PermGen space" that were common in older JVMs.
4. **Garbage Collection**: Metaspace has its own garbage collection mechanism that's separate from the garbage collection of the Java heap. The purpose of Metaspace garbage collection is to reclaim memory used by class metadata that is no longer needed.
5. **Native Memory**: Metaspace is typically implemented using native memory (memory outside the Java heap) and is managed by the operating system. This means that Metaspace memory is not subject to Java heap-related issues like fragmentation.

To summarize, Metaspace serves as the storage for class metadata and is separate from the Java heap, which is used for storing objects. It provides more flexibility and better memory management compared to the older PermGen space. The introduction of Metaspace in Java 8 was a significant improvement in terms of JVM memory management.

In Java, the "Integer pool," sometimes referred to as the "Integer cache," is a feature that optimizes memory usage and performance for small integer values. It's a mechanism by which a fixed set of integer values (typically in the range of -128 to 127) are cached and reused, rather than creating new **Integer** objects for each occurrence of those values.

Here's how the Integer pool works:

1. **Caching Common Values**: Java maintains a cache of commonly used integer values within the specified range (-128 to 127). When you create an **Integer** object with a value within this range, the JVM checks if it already has an **Integer** object with the same value in the pool.
2. **Reusing Existing Objects**: If an **Integer** object with the same value is found in the pool, the JVM reuses that existing object rather than creating a new one. This optimization reduces memory consumption and improves performance because it reduces the overhead of creating and garbage collecting many **Integer** objects.
3. **Beyond the Pool Range**: For integer values outside the cached range, new **Integer** objects are created for each occurrence. These objects are not part of the pool.
4. **User-Defined Pools**: It's worth noting that the integer pool is for a specific range that's commonly used in practice. Users can define their own integer pools if needed for specific ranges by using techniques like object pooling or custom caching.
5. Integer a = 10;
6. Integer b = 10;
7. // Both 'a' and 'b' reference the same cached Integer object with value 10
8. System.out.println(a == b); // true
9. Integer x = 128;
10. Integer y = 128;
11. // 'x' and 'y' reference different Integer objects (outside the pool range)
12. System.out.println(x == y); // false

The primary goal of the integer pool is to optimize memory usage and object creation for frequently used small integer values. It's particularly helpful in scenarios where many small integer values are created, such as loop counters or array indices.

In the example above, **a** and **b** reference the same cached **Integer** object with the value 10 from the pool, while **x** and **y** reference different **Integer** objects because they are outside the cached range.

# Spring framework

Pom.xml file I mej <dependency> neri mej grume nq ayn librarynery voronq uzum enq ogtagorcel,dependencyneri mej import enq anum bolor librarynery voronq anhrajesht en spring frameworkin

Mtnum enq maven repository

Qashum enq spring-core, spring-beans, spring-context

Sranc verjin versioni vra enq sexmum, copy enq anum dependency ev dnum enq pom.xml filei dependencies neri mej

Bolor filery voronq gtnvum en resources directoryum hasaneli en mer cragrin

Ayn object vory stexcvum e classic kochvum e bean => ete class enq grum voric heto object enq stexcelu anuny dnum enq orinak TestBean

Ete uzum enq springin asel stexcir TestBean I object => resourcesi mej stexcum enq nor file applicationContext.xml anunov, ayntex dnum enq github I codey, vori arajin masy configurationner en, isk erkrord masum bean stexcelu coden e, vori mej grum enq class = junit.TestBean(ayn classi path y vori object enq uzum stexcel)

applicationContext.xml filey petq e partadir gtnvi resources directory mej vory mark e arvac vorpes resources as root, vorpeszi ClasspathXmlApplicationContext y karoxana gtnel vor beany petq e sarqi

getBean() method in poxancum enq beani ID n ev TestBean classy TestBean.class

applicationContext.xml y ogtagorceluc heto anpayman pakum enq