* String ⇒

Java strings are Object that allows us to store sequence of character. Which may contain alpha numeric values enclosed in double quotes ("Ankit").

Note →

i) Strings are Immutable (cannot change) in Java.

ii) It contains methods that can perform certain operations on Strings (concat(), equals(), length() etc .....).

Two ways to create String Object →

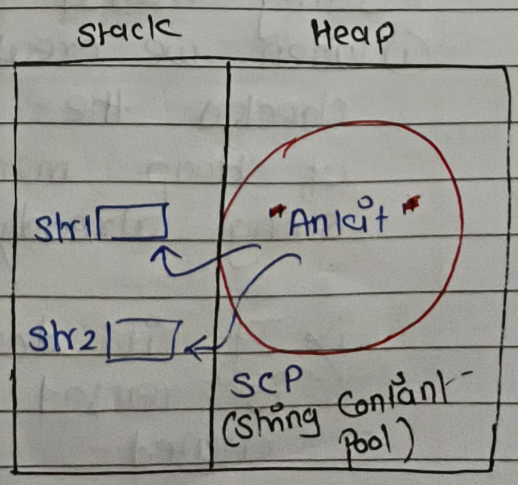
① String literal.
② New keyword. (String Object)

① String literal ⇒

String str1 = "Ankit";

String str2 = "Ankit";

String literal is a string enclosed in double quotes



intern() method used to manage strings in SCP

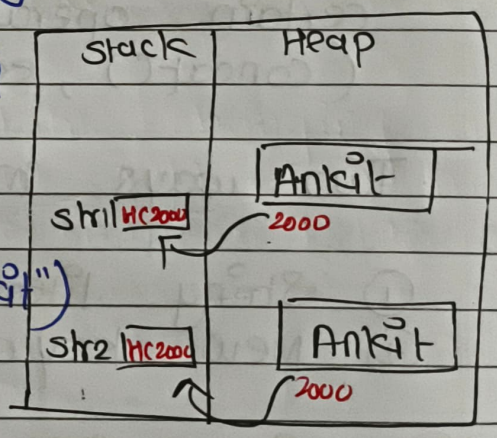
String str = "Ankit";

This is string literal. When we declare string like this, we are actually calling intern() method (intern() method creates an exact copy of string located in the heap memory and stores it in the string constant pool SCP)

② New keyword (string object) ⟹

String str1 = new String ("Ankit");

String object is created explicitly using new keyword



String str2 = new String("Ankit")

** Note →

String literal ⟹

① When we create a string literal, Java checks the String Constant Pool (Part of Heap memory) to see if an identical string already exists or not

i) If it does exist, the existing string is reused and no new object is created

ii) If not, a new string object is created in the pool.

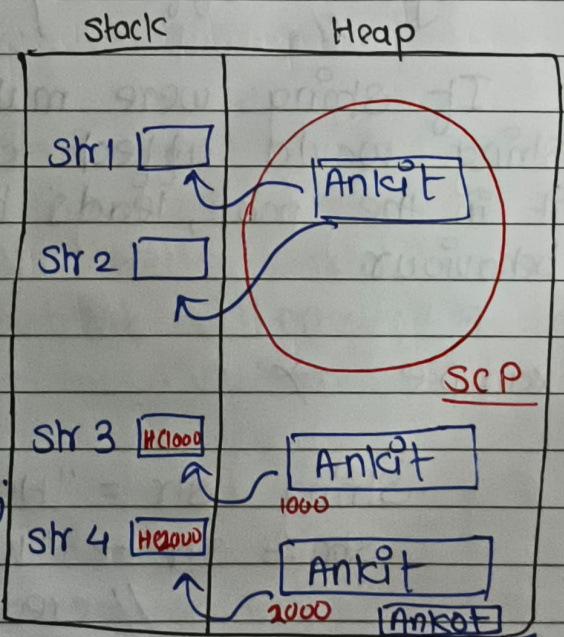
② String literals are implicitly interned

String object ⇒

① It creates a new object in the Heap memory, even if an identical string exists in the pool

② String objects are not automatically interned

```
String str1 = "Ankit";
String str2 = "Ankit";
String str3 = new String("Ankit");
String str4 = new String("Ankit");
str3.replace('i','o');
```



{ When we use replace method to replace 'i' by 'o' in str3 then it store in new dummy object so original string is not change }

(str1 & str2 refference the same string object in the String Pool)

{ (A new string object is created in the Heap separate from the Pool)

As we see in above memory diagram if "Ankit" exists in the String Pool, str3 & str4 refers to a new object in the Heap }

Why String is immutable ⇒

In Java, String is immutable, means once a String object is created, its value cannot be changed

1) Security →

String url = "https://secure-site.com";
(If String is mutable, someone could change it to : "https://false-site.com."

2) String Pool Efficiency →

If string were mutable, changing one string would affect all refferences to it in the pool, leads to unpredictable behaviour

Example →

```
String str = "Hello";
str = str + "world";
        // creates new object
           "Hello world"
```

The original "Hello" string remains unchanged

The refference str is updated to point to the new object

* String class in Java ⇒

The String class in Java is a part of the java.lang package and represents a sequence of character.

Important Methods in String class →

① Length ()

Returns the number of characters in the string.

E.g → public class Main {

```
    public static void main (String[] args)
{
    String str = "Hello World";
    System.out.println ("Length: "+ str.length());
                    // output 11
}
```

② charAt (int index)

Returns the character at the specified index. Index starts from 0.

E.g → public class Main {

```
    public static void main (String[] args) {
        String str = "Hello";
        System.out.println("character at index
                1 :"+str.charAt(1));
} }
```

output e

③ Substring (int beginIndex) →

Returns a substring starting from the given index to the end

E.g →

```
public class Main {

public static void main (String [] args) {

    String str = "Programming";
                 0 1 2
    System.out.println ("Substring : " +
                        str.substring (3));
}       // output : gramming
}
```

④ Substring (int beginIndex, int endIndex)

Returns a substring starting from beginIndex to endIndex - 1

E.g →

```
public class Main {

public static void main (String[] args) {

    String str = "Programming";

    System.out.println ("Substring : " +
                        str.substring (3,8));
}       //Output : gram.
}
```

⑤ Concat (String str) →

Concatenates the specified string to the end of current string.

E.g →

```
public class Main {

  public static void main (String[] args)
  {
    String str1 = "Hello";
    String str2 = "World";

    System.out.println("Concatenated: + str1.concat-
                                (str2));
    }      // Output : Hello World.
}
```

⑥ equals (Object obj)

Compares the content of two strings. Returns true if they are equal.

E.g →

```
public class Main {
  public static void main (String[] args)
  {
    String str1 = "Java";
    String str2 = "Java";
    String str3 = "java";
    Syso( str1.equals(str2)); // true
    Syso ( str1. equals (str3)); // False
}}
```

⑦ equalsIgnoreCase (String str) →

Compares two strings, ignoring case differences.

E.g →

```
String str1 = "JaVA";
String str2 = "java";
Sys0 (str1.equalsIgnoreCase (str2));
    // true.
```

⑧ CompareTo (String anotherString) →

Compares two strings and Returns accordingly
i) '0' if equal.
ii) Positive value if string is greater.
iii) Negative value if string is smaller.

E.g →

```
String str1 = "Apple";
String str2 = "Banana";

Sys0 (str1. compareTo(str2));
    // Negative value.
```

⑨ toUpperCase () →

Converts all characters of the string to uppercase.

E.g → String str = "Hello"; // HELLO
Sys0 ("str. toUpperCase ());

(10) toLowerCase () →

Convert all characters of the String to lowercase

E.g →

Syso (str.toLowerCase ()); (consider HELLO)

//hello

(11) trim () →

Removes leading and trailing spaces from the string.

E.g → String str = " Hello " ; space

Syso.out.println (Str.trim () ~~+ "~~);

(12) replace (char oldChar , char newchar) →

Replaces all occurrences of a character with another.

E.g → String str = "banana";
Syso (str.replace('a', 'o'));

(13) contains (charSequence seq) →

Checks if the string contains the specified sequence.

E.g → String str = "Hello world";
Syso (str.contains ("World"));
// true.

(14) startsWith (String prefix) →

Checks if the string starts with the specified prefix

E.g → String str = "Ankit";
Syso (str.startWith ("An"));
// true.

(15) endsWith (String suffix) →

Checks if the string ends with the specified suffix

E.g → String str = "Ankit";
Syso (str.endsWith ("t"));
// true

(16) isEmpty () →

Checks if the string is empty

E.g → String str = "";
Syso (str.isEmpty()); // true.

* For each loop →

The for each loop in Java (also called an enhanced for loop) is a simpler and more readable way to iterate over arrays or collections, like lists & sets.

Syntax →

```
for(dataType element : collection)
{
    // code to be executed for each
       element.
}
```

where,

dataType is the type of elements in the array or collection.

element is a variable that represents the current element in loop

collection is array or collection you want to iterate over

(Array) Example →

```
public class Main {
    public static void main (String[] args)
{
    int[] numbers = {10, 20, 30, 40, 50};

    for (int num : numbers) {

        Syso (num);
} } }
```

ArrayList<E> or List<E>

(Type of Parameter)

* What is Generic in Java ?

Generics in Java allow you to create classes, interfaces, and methods that operate on a parameterized type.

This enables type safety, code reusability, and eliminated of explicit type casting.

* Problem without Generics

Before Java introduced generics, collections stored objects of type object, which require explicit typecasting

```
import java.util.*;

public class WithoutGenerics {
    public static void main(String[] args)
{
    List list = new ArrayList(); // No typecasting
    list.add("Hello");                safty
    list.add(10); // No restriction, can
                     add any type.
    String s = (String) list.get(0);
                          // Type casting
    System.out.println(s);          required.
}
}
```

{ No type safety (adding Integer to a String list is possible) }

* Solution with Generics :→

Generics enforce type safety at compile time

```
import java.util.*;

public class WithGenerics {

public static void main(String[] args){

List<String> list =new ArrayList<>();
                                // Type safe
list.add ("Hello");
// list.add(10); // Compile-time error.

String s = list.get(0);
                                // No type casting
                                needed.
System.out.println(s);

}
}
```

{ Type safety, Only string elements can be added

{ There are ~~some~~ also Generic classes, Generic methods, Generic Interfaces }

{ String Builder & Buffer also used for method chaining. i.e we can call multiple methods by using "." }

* String Buffer and String Builder in Java

In java, String Buffer and String Builder are used to create mutable (Modifiable) sequences of characters, unlike String which is immutable

Both are part of the java.lang package and are used when frequent modifications (such as appending, inserting, or deleting characters) are required.

① String Buffer :→

i) StringBuffer is thread-safe and synchronized, meaning it can be safely used in a multi-threaded environment.

ii) Since it is synchronized, it performs slower compared to StringBuilder in a single-threaded environment.

iii) It provides methods like append(), insert(), replace(), delete(), and reverse().

Example :→

```
public class StringBufferExample{
    public static void main(String[] args){
        StringBuffer sb = new StringBuffer("Hello");
        sb.append("World"); // modify original String
        System.out.println(sb); //output → Hello World
    }
}
```

② String Builder :→

i) String Builder is not synchronized, meaning it is not thread-safe.

ii) It is faster than String Buffer because it doesnot have synchronization overhead.

iii) It has the same methods as String Buffer but it is preffered when thread safety is not required

Example :→

```
public class StringBuilderExample {
public static void main(String args[]){
StringBuilder sb = new StringBuilder
                    ("Hello");
sb.append("World"); // Modify original String
System.out.println(sb); // HelloWorld
```

* When to use what ?

i) Use String Buffer when multiple threads are modify the same String object.

ii) Use String Builder when working in a single threaded environment for better performance

* Example (StringBuffer and StringBuilder Methods)

① String Buffer Methods and
② String Builder Methods.

```
StringBuffer sb = new StringBuffer("Hello");
```

OR

```
StringBuilder sb = new StringBuilder("Hello");
```

// append() - Adds string at the end.

```
sb.append("world");
System.out.println("After apend:" + sb);
// Output : Hello world
```

// insert() - Inserts string at given index.

```
sb.insert(5, "Java");
System.out.println("After insert:" + sb);
// output : Hello Java world
```

// replace() - Replace characters from index start to end with given string

```
sb.replace(6,10,"c++");
System.out.println("After replace:" + sb);
//output : Hello c++ world.
```

// delete() - Delete characters from index start to end.

```
sb.delete(6,10);
System.out.println("After delete:"+sb);
```

Output : Hello World

// reverse() - Reverse the string

```
sb.reverse();
System.out.println("After reverse:"+sb);
```

Output : dlroW olleH

// capacity() - Return current capacity of buffer or builder

```
System.out.println("Capacity:"+sb.capacity());
```

// Default capacity : 16 + length of string.
for both StringBuffer & StringBuilder

// length() - Returns the length of string

```
System.out.println("length:"+sb.length());
```

Output : 12

// setCharAt() - Sets a character at a specified index.

```
sb.setCharAt(6,'x');
System.out.println("After setCharAt:"+sb);
```

Output: Hello Xorld.

// ensureCapacity() - Ensures that the capacity is at least equal to specified minimum

```
sb.ensureCapacity(50);
System.out.println("After ensureCapacity(50).
    Capacity:"+sb.capacity());
```

Output: >=50

* Difference Between

String Buffer	String Builder
i) Mutable	i) Mutable
ii) ~~Yes~~ Thread Safety because it is Synchronized	ii) No Thread Safety (Not Synchronized)
iii) Slower (due to Synchronizaⁿ)	iii) Faster (due to No Synchronizaⁿ)
iv) Multi-Threaded environment	iv) Single-threaded Applications.

* Iterator → (Presents in java.util package)

An iterator in Java is an interface used to traverse or iterate through a collection (like ArrayList, HashSet, LinkedList, etc). one element at a time.

It provides methods to access elements sequencially without exposing the underlying data structure.

Why use an iterator ?

i) Allow traversal of collections without exposing internal implementation

ii) Provides safe element removal during iteration (remove() method).

iii) Works on all collection framework classes (e.g List, Set, Queue).

Iterator Interface Methods.

① boolean hasNext() :→

Returns true if there are more elements to iterate

② E next() :→

Returns the next element in the iteration

③ void remove() :→

Removes the last element returned by next().

Example →

```
ArrayList<String> names = new ArrayList<>
                                  ();
names.add("Ankit");
names.add("Raju");
names.add("Prem");
names.add("Shubham");

// Getting an Iterator

Iterator<String> = names.Iterator();

// Iterating through the list

while (itr.hasNext()) {

    System.out.println(itr.next());
}

}
```

Now suppose we have to remove the "shubham" from the ArrayList then we had to reset the iterator because iterator can only move forward and previous one is exhausted
There is an another approach to use remove() without resetting iterator i.e using <u>ListIterator</u> (for Lists)

(ListIterator is a subinterface of Iterator that allows bidirectional traversal (forward & backward) methods →
boolean hasPrevious(), E previous(), void add(E e)

* Using Iterator :→

```
ArrayList<String> names = new ArrayList<>();
names.add("Ankit");
names.add("Prem");
names.add("shubham");

Iterator<String> itr = names.iterator();

while (itr.hasNext()){
  System.out.println(itr.next());
}

// Getting a new iterator (Previous one is exhausted)

itr = names.iterator();
```

Iterator	ListIterator
① Forward only	① Forward & Backward
② No Modify Elements	② It can modify (set(), add()).
③ All collections	③ Only List
④ No Get Index	④ nextIndex(), previousIndex()

```
// Removing "Shubham"
while (itr.hasNext()) {
    if (itr.next().equals("Shubham")) {
        itr.remove();
    }
}
```

(Again Reset iterator and print remaining elements after removing "shubham" using while loop)

* Using ListIterator :→

(consider ArrayList from Iterator)

```
ListIterator<String> listItr = names.listIterator();

while (listItr.hasNext()) {
    System.out.println(listItr.next());
}

//Moving backward and removing "shubham".

while(listItr.hasPrevious()) { // This avoids the need to resetting
    if (listItr.previous().equals("Shubham")) {
        listItr.remove(); // remove "shubham"
        break;            // stop after removal
    }
}

System.out.println("Updated list");

while(listItr.hasNext()) {

    Systemout.println(listItr.next());
}
}
```

Output : Amit
Prem

* Garbage collector :→

Garbage Collector is the process of automatically freeing memory by destroying objects that are no longer reachable.

(In java, memory is managed by the JVM (Java virtual Machine). Objects are created in heap memory, and the GC takes care of cleaning up unused objects to avoid memory leaks)

Why use Garbage Collector :→

i) Avoid manual memory management like in C/C++ (malloc, free).
ii) Prevent memory leaks.
iii) Ensures efficient memory utilizations.
iv) Enhances program stability and performance.

How does it works ?

i) JVM tracks references to all objects.

ii) If an object is no longer referenced, it becomes eligible for garbage collection.

iii) The Garbage Collector will reclaim memory occupied by those unreferenced objects.

Example :→

An object becomes eligible for GC when no live thread can access it.

```
public class GCDemo {
    public static void main (String[] args)

    GCDemo obj = new GCDemo();
    obj = null;
                    // Now obj is eligible for GC
    System.gc();
                    // Request JVM to run GC.
    }

    @Override

    protected void finalize() throws Throwable
    {
    System.out.println("Garbage Collector called");
    }

}
```

System.gc() is just a request, JVM decides whether to run GC or not.

* finalize() method :→

i) It's a method in the Object class.

ii) Called before an object is garbage collected.

iii) Can be overridden to perform cleanup.

(finalize() is deprecated since Java 9 and removed in Java 18.)
Instead of it use try-with-resources or cleaner API.

What GC can't Do ?

i) Doesn't guarantee when memory will be freed.

ii) Doesn't clean up non-memory resources like file handles or database connections.