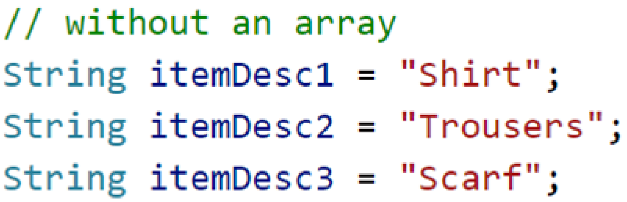
Arrays and Strings

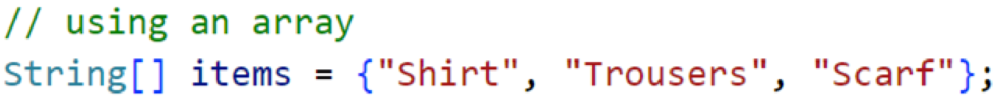
What is an array?

Arrays

An array is an object that stores a collection of values.



```
// without an array
String itemDesc1 = "Shirt";
String itemDesc2 = "Trousers";
String itemDesc3 = "Scarf";
```



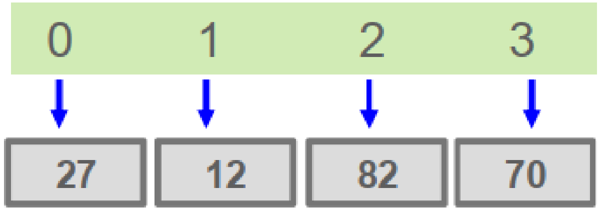
```
// using an array
String[] items = {"Shirt", "Trousers", "Scarf"};
```

Arrays can store two types of data:

- A collection of primitive data types.
- A collection of objects.

Arrays

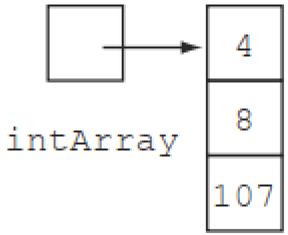
- An array is an indexed container that holds a set of values of a single type.
- Each item in an array is called an element.
- Each element is accessed by its numerical index.
- The index of the first element is 0 (zero).
  - A four-element array has indices: 0, 1, 2, 3



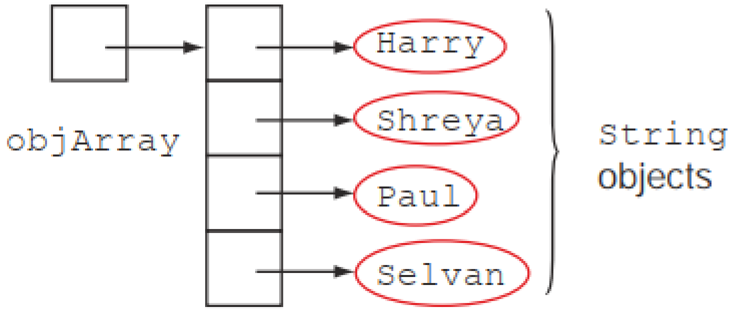
0	1	2	3
27	12	82	70

Arrays of Primitives and References

```
int intArray[] = new int[]{4, 8, 107};

String objArray[] = new String[]{"Harry", "Shreya", "Paul", "Selvan"};
```

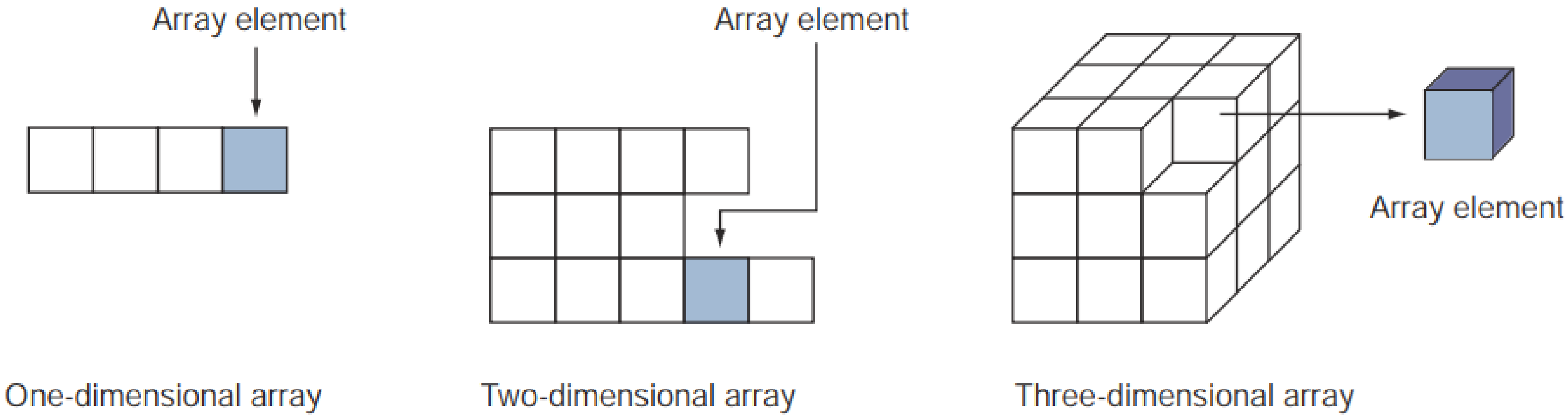


Array of primitive data



Array of objects

Types of Array in Java



Create, declare and initialize an array

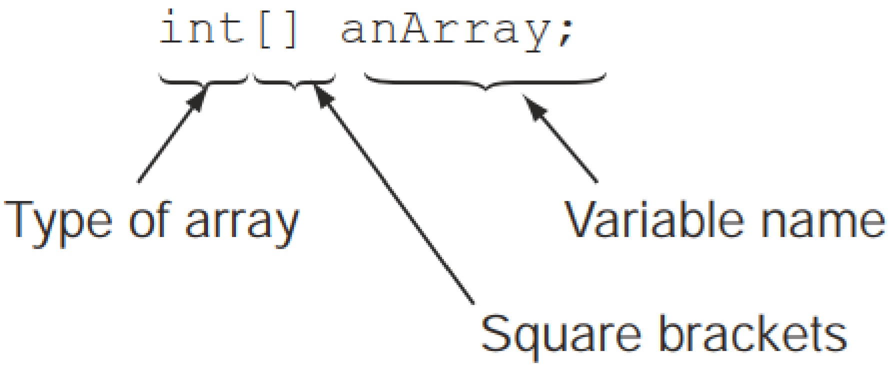
Creating an array

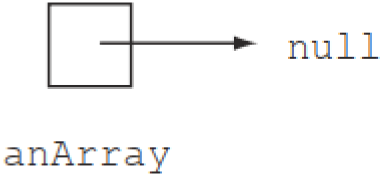
Creating an array involves three steps, as follows:

- Declaring the array;
- Allocating the array;
- Initializing the array elements.

Array declaration

An array declaration includes the **array type** and **array variable**.

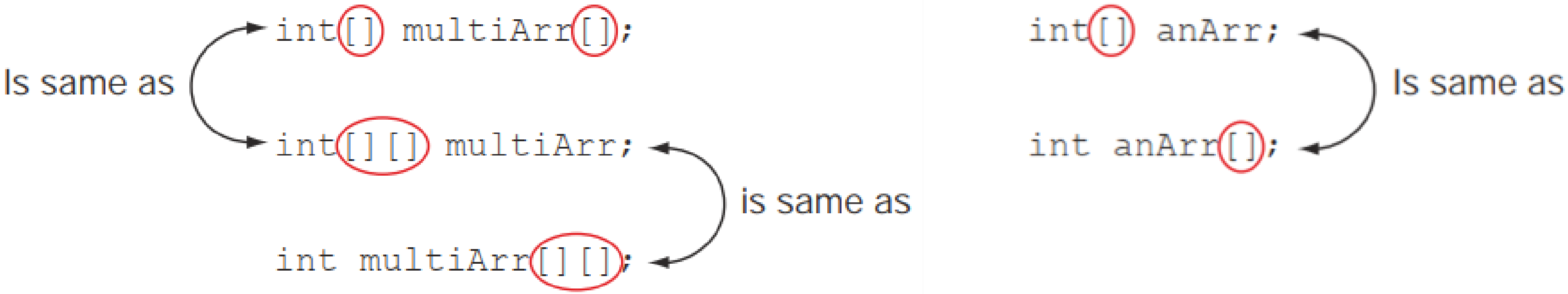




** The array declaration only creates a variable that refers to **null**.*

Array declaration

The number of bracket pairs indicates the depth of array nesting.



```
int[] multiArr[];

int[][] multiArr;

int multiArr[][];
```

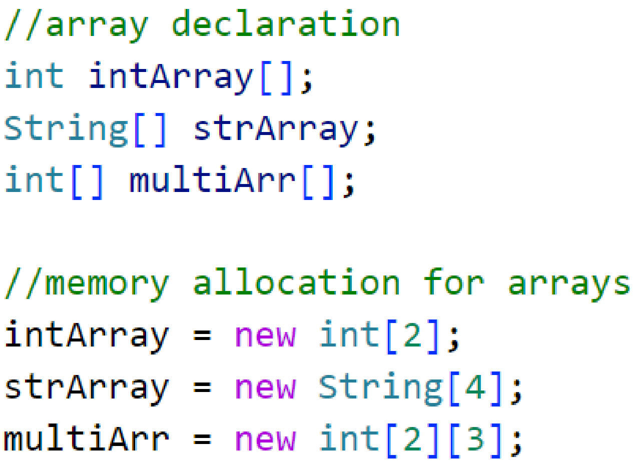
```
int[] anArr;

int anArr[];
```

** In an array declaration, placing the square brackets ыыы to the type (as in `int[]` or `int[][]`) is preferred.*

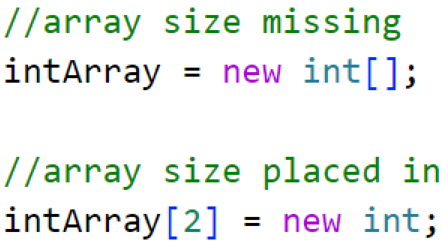
Array allocation

Array allocation will allocate memory for the elements of an array.

When you allocate memory for an array, you should specify its dimensions, such as the number of elements the array should store.



```
//array declaration
int intArray[];
String[] strArray;
int[] multiArr[];

//memory allocation for arrays
intArray = new int[2];
strArray = new String[4];
multiArr = new int[2][3];
```



```
//array size missing          ⊗
intArray = new int[];

//array size placed incorrectly
intArray[2] = new int;        ⊗
```

** The size of an array can’t expand or reduce once it is allocated.*

Array allocation

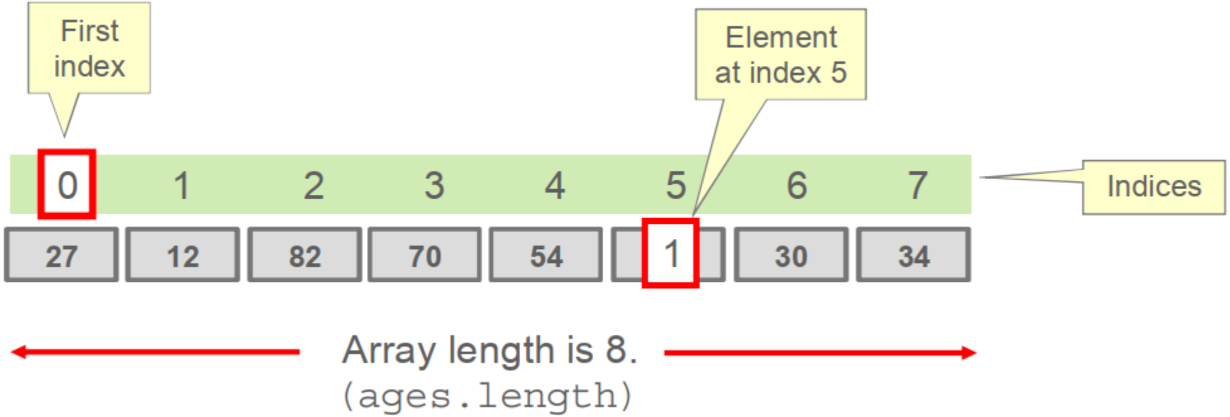
Once allocated, all the array elements store their default values.

Elements of an array that store primitive data types store:

- `0` for integer types (`byte`, `short`, `int`, `long`);
- `0.0` for decimal types (`float` and `double`);
- `false` for boolean;
- `\u0000` for char data;
- `null` for objects.

Array Indices

```
int intArray[] = {27, 12, 82, 70, 54, 1, 30, 34};
```

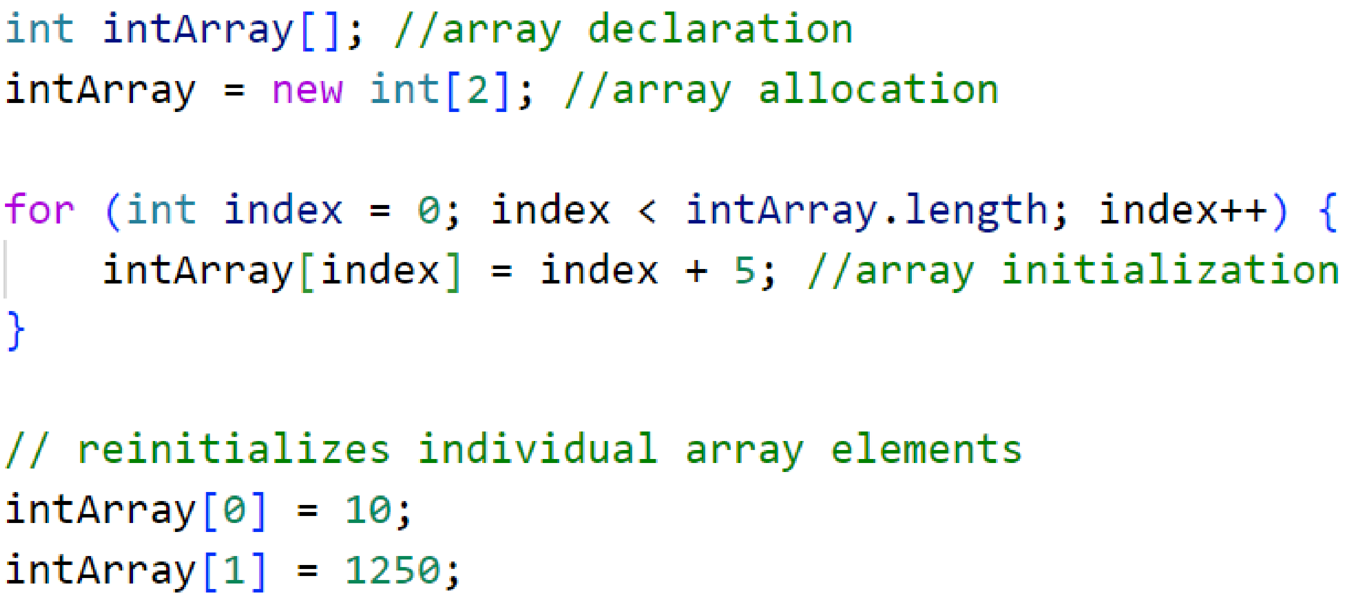


0	1	2	3	4	5	6	7
27	12	82	70	54	1	30	34

Array length is 8.
(ages.length)

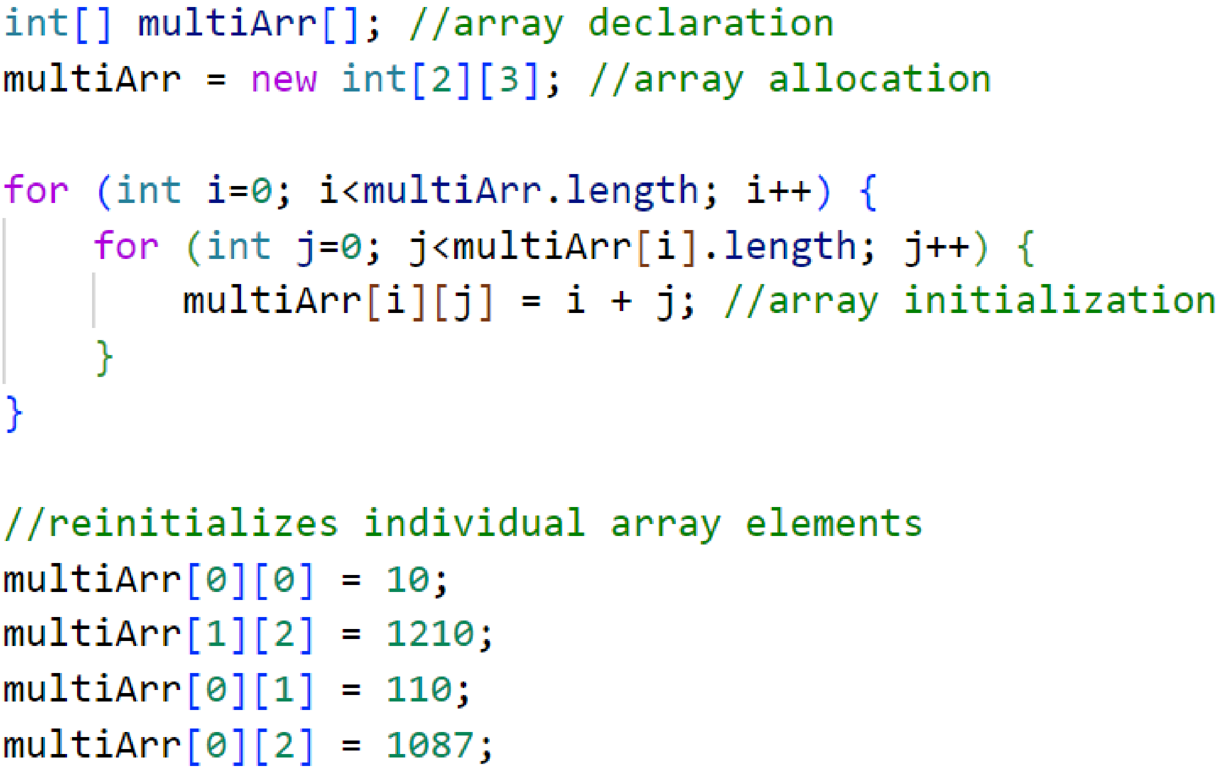
```
int item = intArray[5];
System.out.println(item); // 1
```

Array initialization: One-dimensional array



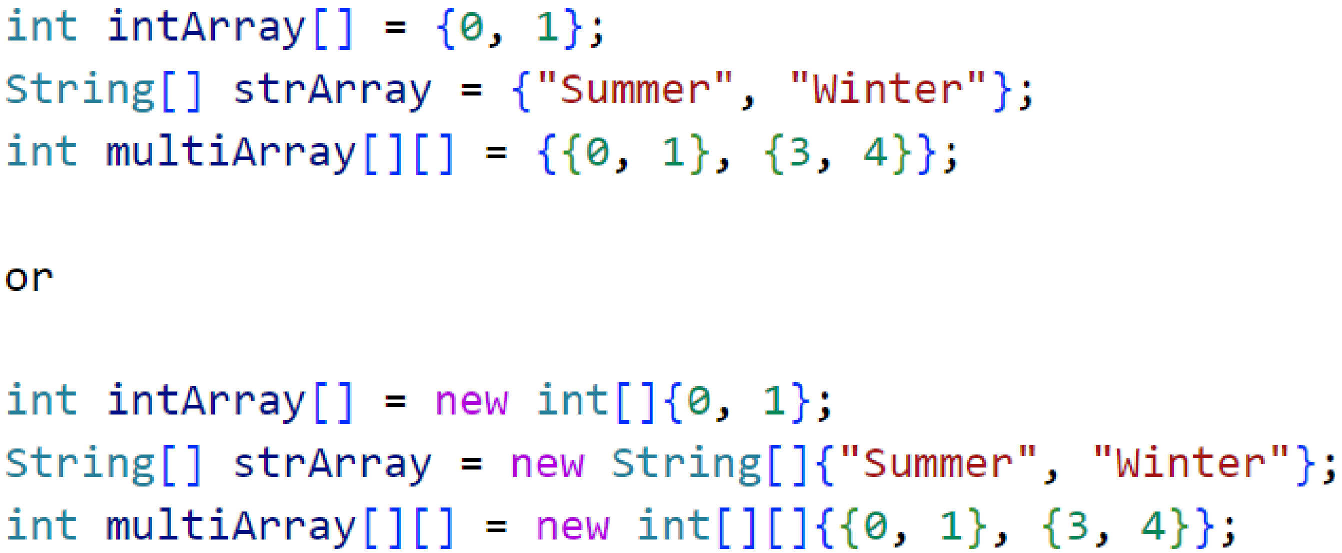
```
int intArray[]; //array declaration
intArray = new int[2]; //array allocation

for (int index = 0; index < intArray.length; index++) {
    intArray[index] = index + 5; //array initialization
}

// reinitializes individual array elements
intArray[0] = 10;
intArray[1] = 1250;
```

Array initialization: Two-dimensional array



```java
int[] multiArr[]; //array declaration
multiArr = new int[2][3]; //array allocation

for (int i=0; i<multiArr.length; i++) {
    for (int j=0; j<multiArr[i].length; j++) {
        multiArr[i][j] = i + j; //array initialization
    }
}

//reinitializes individual array elements
multiArr[0][0] = 10;
multiArr[1][2] = 1210;
multiArr[0][1] = 110;
multiArr[0][2] = 1087;
```

Combining array declaration, allocation, and initialization



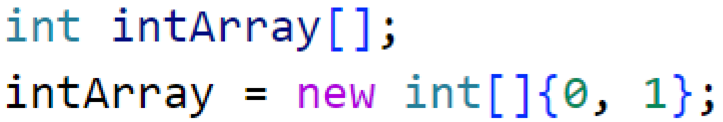
```
int intArray[] = {0, 1};
String[] strArray = {"Summer", "Winter"};
int multiArray[][] = {{0, 1}, {3, 4}};

or

int intArray[] = new int[]{0, 1};
String[] strArray = new String[]{"Summer", "Winter"};
int multiArray[][] = new int[][]{{0, 1}, {3, 4}};
```

* *When you combine an array declaration, allocation, and initialization in a single step, you can’t specify the size of the array.*

Combining array declaration, allocation, and initialization

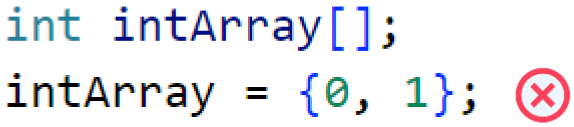
If you declare and initialize an array using two separate lines of code, you’ll use the keyword **new** to initialize the values.

The following lines of code are correct:



```
int intArray[];
intArray = new int[]{0, 1};
```

But you can’t miss the keyword **new** and initialize your array as follows:

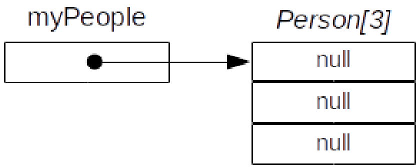


```
int intArray[];
intArray = {0, 1}; ⊗
```

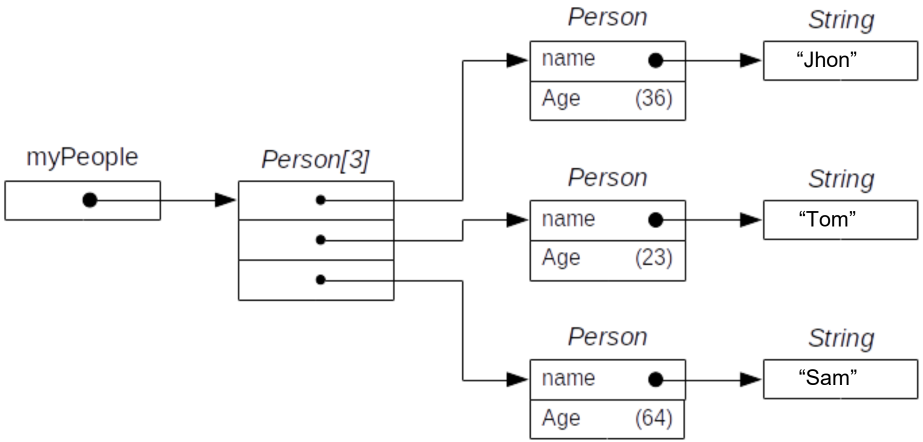
Arrays of Objects

But you can’t miss the keyword **new** and initialize your array as follows:

```
Person[] myPeople;
myPeople = new Person[3];
```



```
myPeople[0] = new Person("John", 36);
myPeople[1] = new Person("Tom", 23);
myPeople[2] = new Person("Sam", 64);
```



String class

String class

The class `String` defined in the Java API in the `java.lang` package.

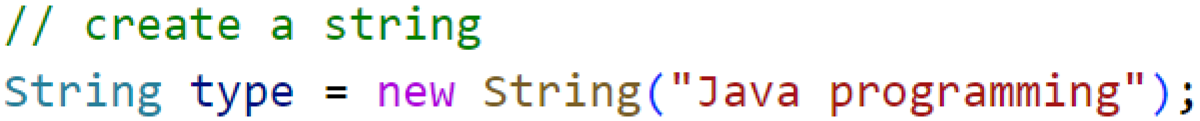
The `String` class represents **character strings**.

Most commonly used methods:

- `indexOf();`
- `substring();`
- `replace();`
- `charAt()`
- …

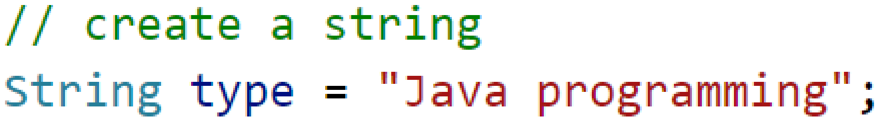
Creating String objects

You can create objects of the class `String` by using the **new** operator



```
// create a string
String type = new String("Java programming");
```

or by using `String` **literal values** (values within double quotes)

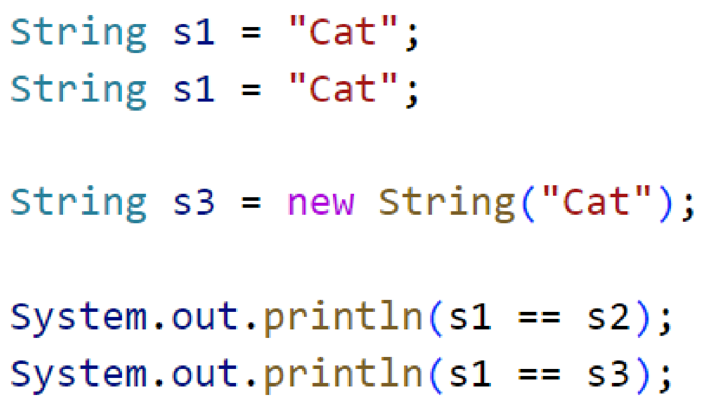


```
// create a string
String type = "Java programming";
```

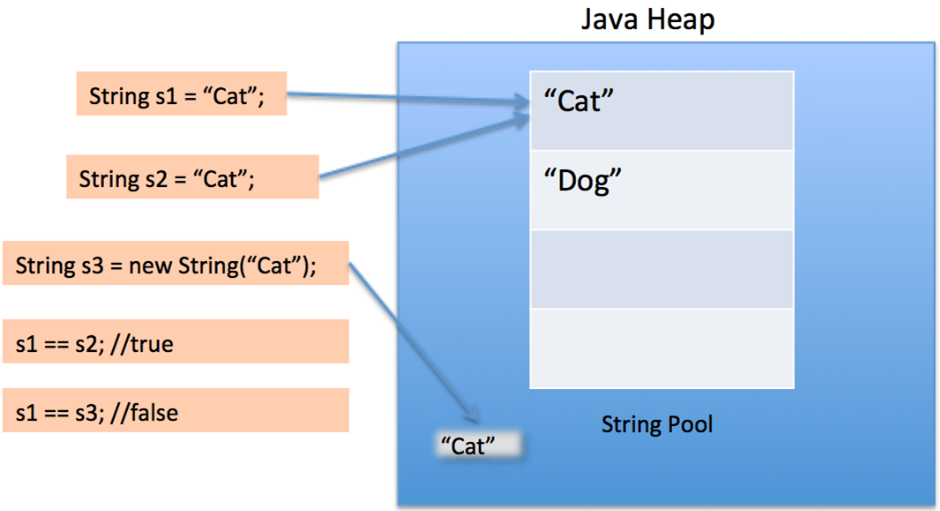
String Constant Pool

The objects are created and stored in a pool of String objects.

Before creating a new object in the pool, Java searches for an object with similar contents.



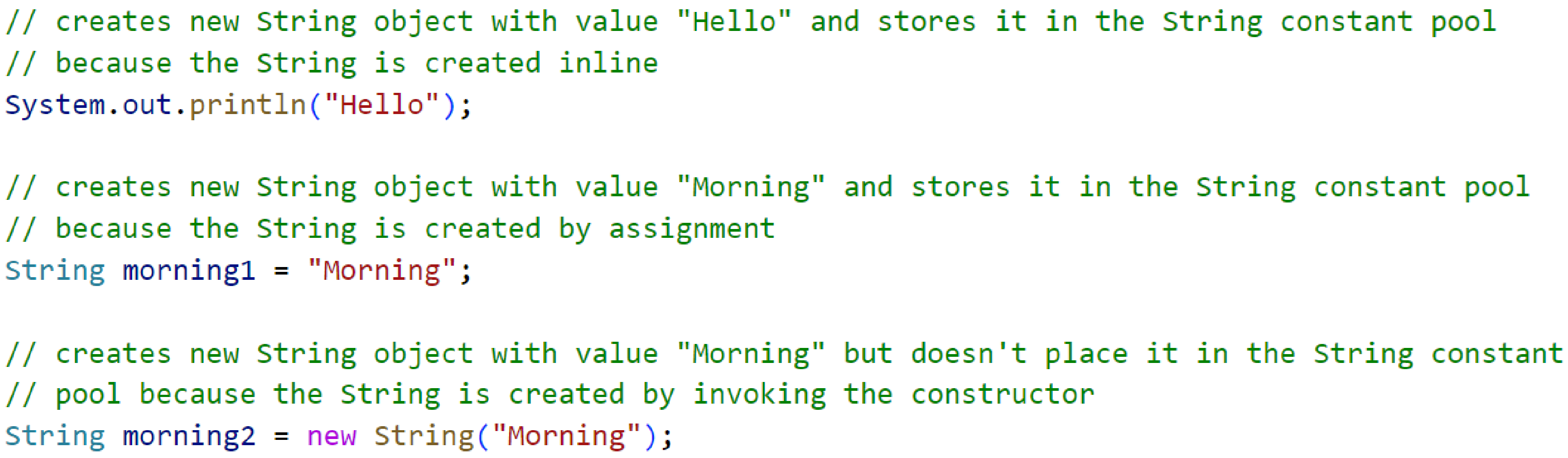
```
String s1 = "Cat";
String s1 = "Cat";

String s3 = new String("Cat");

System.out.println(s1 == s2);
System.out.println(s1 == s3);
```



```
String s1 = “Cat”;
String s2 = “Cat”;
String s3 = new String(“Cat”);
s1 == s2; //true
s1 == s3; //false
```

** The operator == compares the addresses of the objects referred to by the variables **s1** and **s3**, and not their values.*

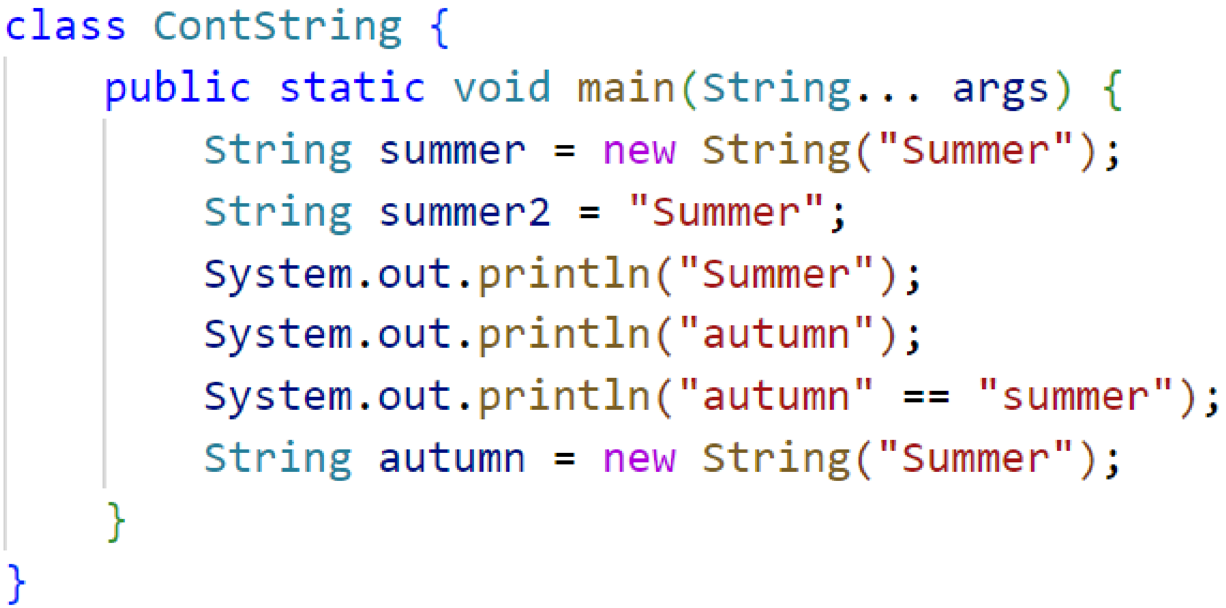
Creating Strings



```
// creates new String object with value "Hello" and stores it in the String constant pool
// because the String is created inline
System.out.println("Hello");

// creates new String object with value "Morning" and stores it in the String constant pool
// because the String is created by assignment
String morning1 = "Morning";

// creates new String object with value "Morning" but doesn't place it in the String constant
// pool because the String is created by invoking the constructor
String morning2 = new String("Morning");
```

Counting string objects

Count the total number of `String` objects created in the following code:



```
class ContString {
    public static void main(String... args) {
        String summer = new String("Summer");
        String summer2 = "Summer";
        System.out.println("Summer");
        System.out.println("autumn");
        System.out.println("autumn" == "summer");
        String autumn = new String("Summer");
    }
}
```

The class String is immutable

- Once created, the contents of an object of the class `String` can never be modified.

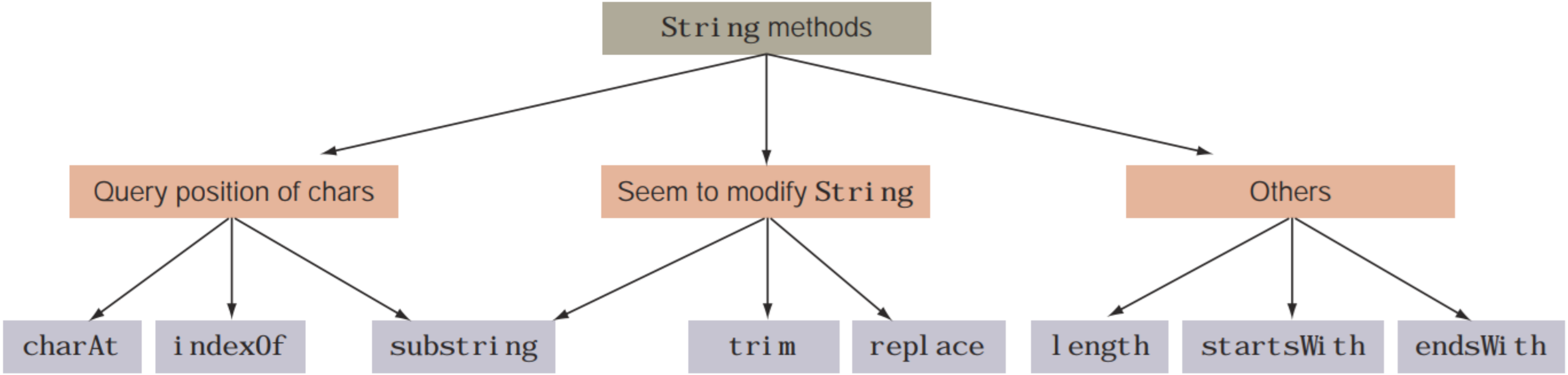
- The immutability of `String` objects helps the JVM reuse `String` objects.

- `String` objects can be shared across multiple reference variables without any fear of changes in their values.

- All the `String` methods (`substring`, `concat`, `toLowerCase`, `toUpperCase`, …) that return a modified `String` value return a new `String` object with the modified value. The original `String` value always remains the same.

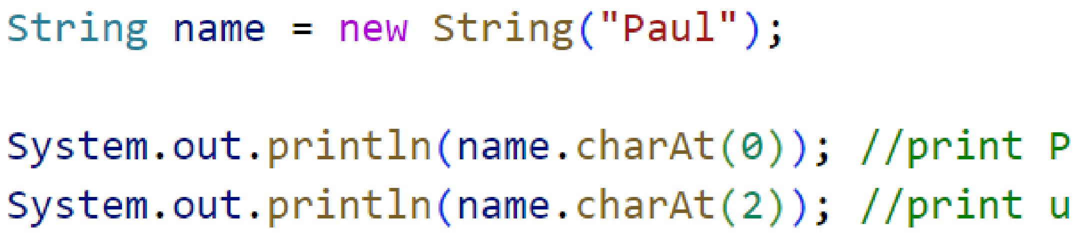
Manipulate String objects

Methods of the class String

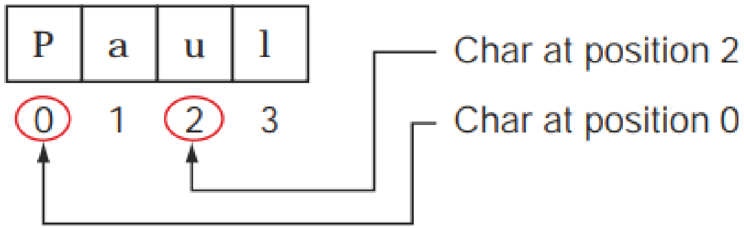


charAt()

You can use the method **charAt(int index)** to retrieve a character at a specified index of a String:



```
String name = new String("Paul");

System.out.println(name.charAt(0)); //print P
System.out.println(name.charAt(2)); //print u
```



Char at position 2

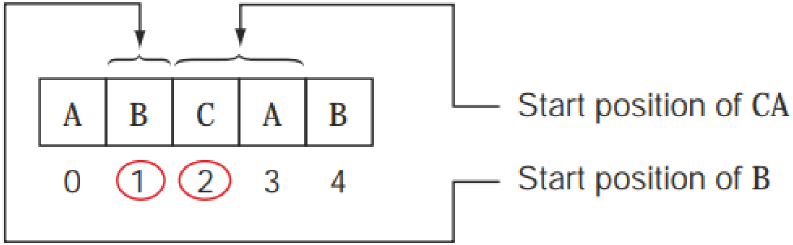
Char at position 0

indexOf()

You can search a String for the occurrence of a char or a String.

If the specified char or String is found in the target String, this method returns the first matching position; otherwise, it returns -1.

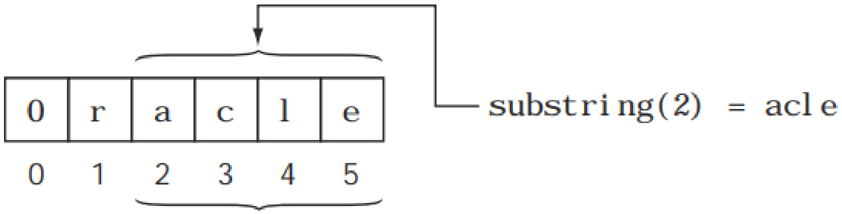
```
String letters = "ABCAB";
System.out.println(letters.indexOf('B')); //prints 1
System.out.println(letters.indexOf("S")); //prints -1
System.out.println(letters.indexOf("CA")); //prints 2
```



substring()

The **substring()** method is shipped in two flavors. The first returns a substring of a String from the position you specify to the end of the String.

```
String exam = "Oracle";
String sub = exam.substring(2);
System.out.println(sub); //print acle
```



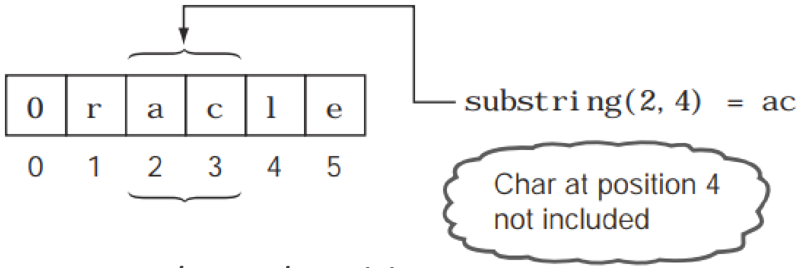
O r a c l e

0 1 2 3 4 5

substring(2) = acle

You can also specify the end position with this method:

```
String exam = "Oracle";
String result = exam.substring(2, 4);
System.out.println(result); //print ac
```



O r a c l e

0 1 2 3 4 5

substring(2, 4) = ac

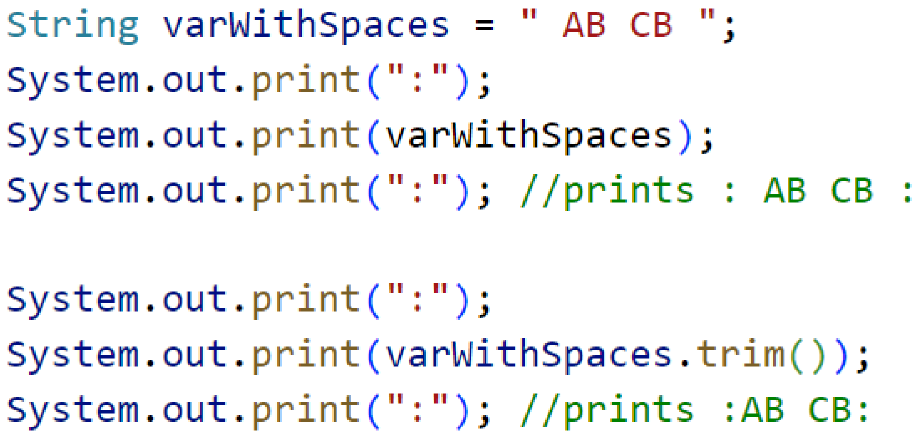
Char at position 4 not included

* *The substring method doesn't include the character at the end position.*

trim()

The **trim()** method returns a new String by removing all the leading and trailing white space in a String.

White spaces are blanks (new lines, spaces, or tabs).



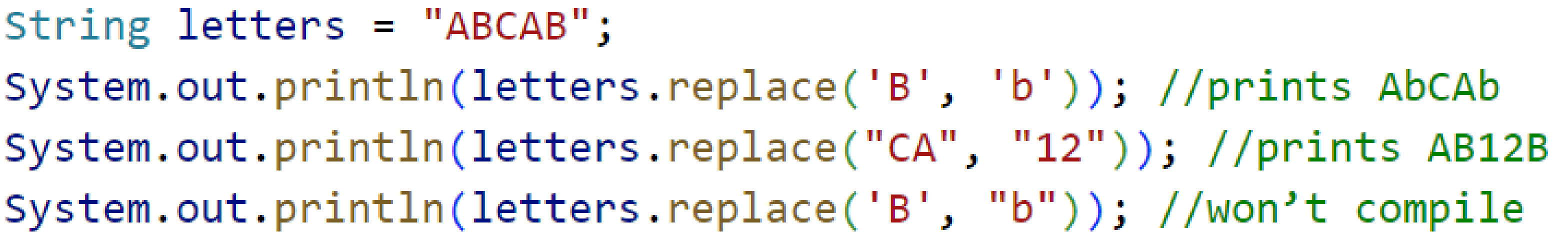
```
String varWithSpaces = " AB CB ";
System.out.print(":");
System.out.print(varWithSpaces);
System.out.print(":"); //prints : AB CB :

System.out.print(":");
System.out.print(varWithSpaces.trim());
System.out.print(":"); //prints :AB CB:
```

** This method doesn’t remove the space within a String.*

replace()

This method will return a new String by replacing all the occurrences of a char with another char.

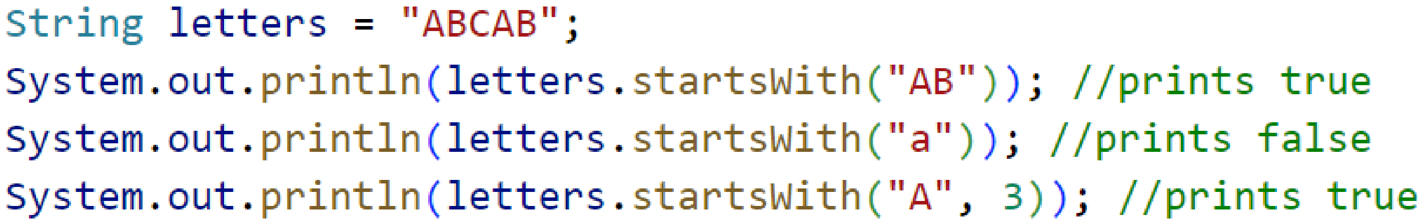
Instead of specifying a char to be replaced by another char, you can also specify a sequence of characters – a String to be replaced by another String.



```
String letters = "ABCAB";
System.out.println(letters.replace('B', 'b')); //prints AbCAb
System.out.println(letters.replace("CA", "12")); //prints AB12B
System.out.println(letters.replace('B', "b")); //won’t compile
```

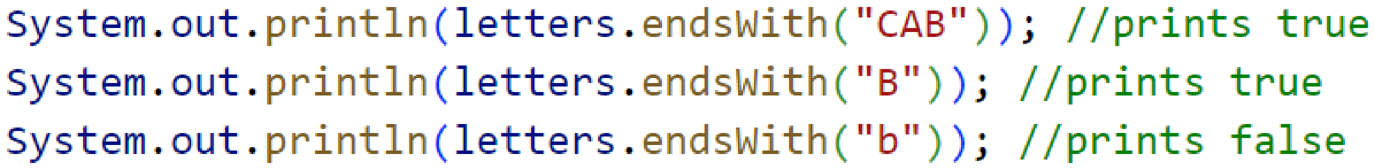
startsWith() & endsWith()

The method **startsWith()** determines whether a String starts with a specified prefix, specified as a String.



```
String letters = "ABCAB";
System.out.println(letters.startsWith("AB")); //prints true
System.out.println(letters.startsWith("a")); //prints false
System.out.println(letters.startsWith("A", 3)); //prints true
```

The method **endsWith()** tests whether a String ends with a particular suffix.

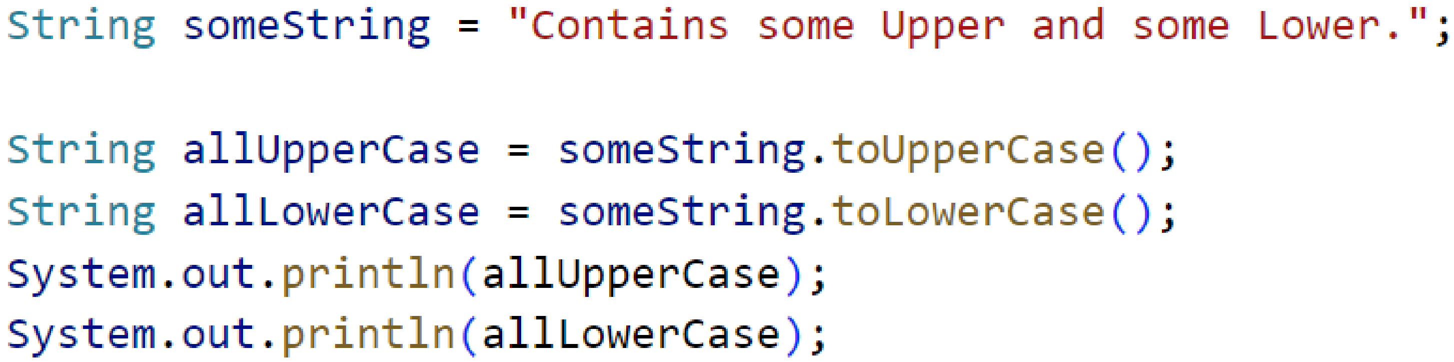


```
System.out.println(letters.endsWith("CAB")); //prints true
System.out.println(letters.endsWith("B")); //prints true
System.out.println(letters.endsWith("b")); //prints false
```

toLowerCase() & toUpperCase()

The method **toLowerCase()** converts all of the characters in a String to lower case.

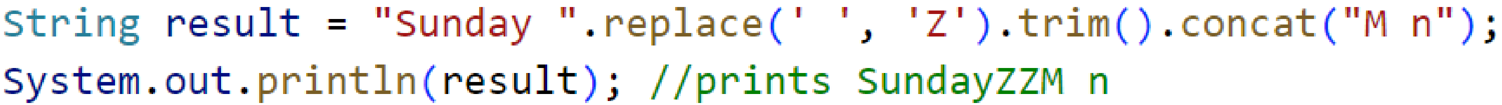
The method **toUpperCase()** converts all of the characters in a String to upper case.



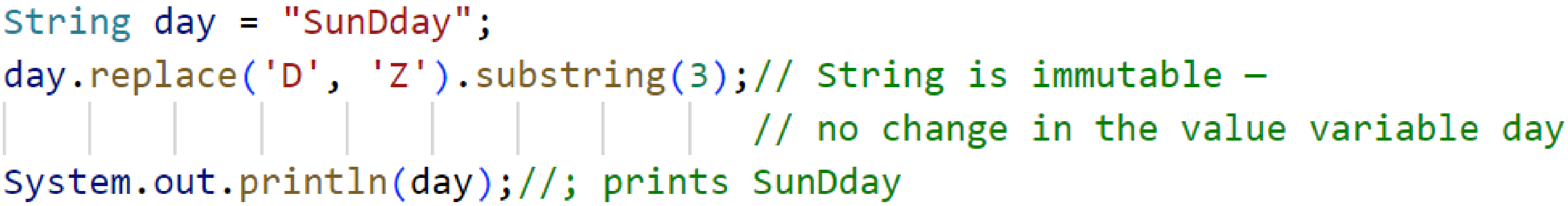
```
String someString = "Contains some Upper and some Lower.";

String allUpperCase = someString.toUpperCase();
String allLowerCase = someString.toLowerCase();
System.out.println(allUpperCase);
System.out.println(allLowerCase);
```

Method chaining

It’s common practice to use multiple String methods in a single line of code, as follows:



```
String result = "Sunday ".replace(' ', 'Z').trim().concat("M n");
System.out.println(result); //prints SundayZZM n
```



```
String day = "SunDday";
day.replace('D', 'Z').substring(3);// String is immutable –
                                   // no change in the value variable day
System.out.println(day);//; prints SunDday
```

* *When chained, the methods are evaluated from left to right.*

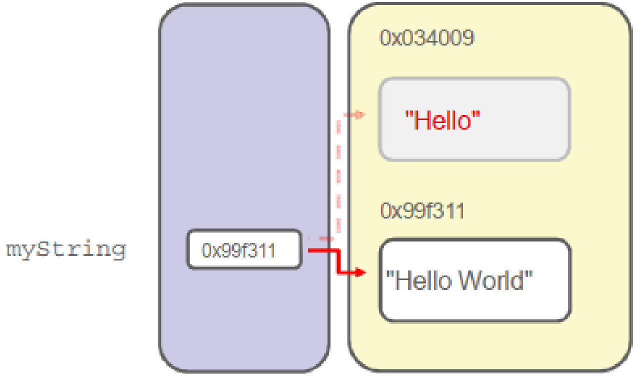
String objects and operators

- Concatenation: + and +=
- Equality: == and !=

Concatenation: + and +=

- You can use the operators + and += to concatenate two String values.
- Because String is immutable, concatenating two strings requires creating a new String.

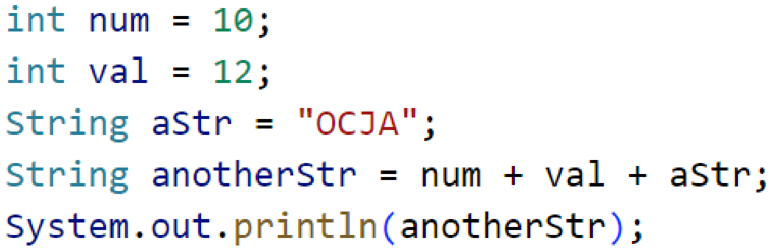
```
String myString = "Hello";
myString = myString.concat(" World"); // myString + " World"
```



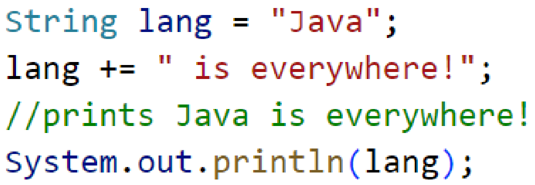
Concatenation: + and +=

- The **+** operator can be used with the primitive values, and the expression `num+val+aStr` is evaluated from left to right.

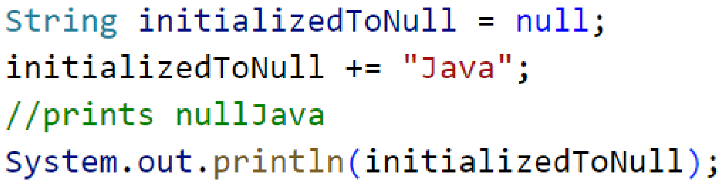
- When you use **+=** to concatenate String values, ensure that the variable you're using has been initialized (and doesn't contain null).



```java
int num = 10;
int val = 12;
String aStr = "OCJA";
String anotherStr = num + val + aStr;
System.out.println(anotherStr);
```



```java
String lang = "Java";
lang += " is everywhere!";
//prints Java is everywhere!
System.out.println(lang);
```



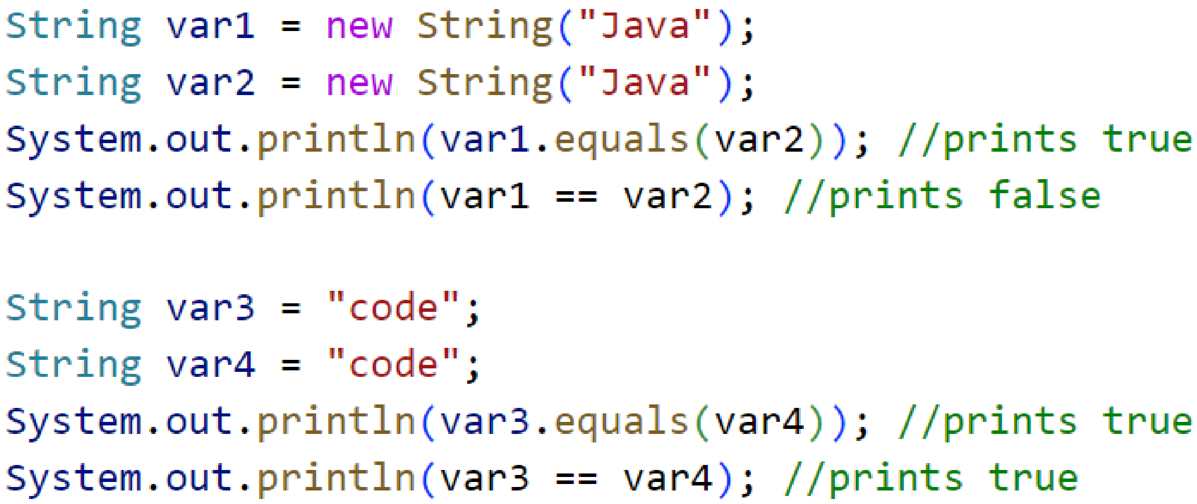
```java
String initializedToNull = null;
initializedToNull += "Java";
//prints nullJava
System.out.println(initializedToNull);
```

Equality of Strings

- The correct way to compare two String values for equality is to use the **`equals`** method defined in the String class.

- This method returns a `true` value
  - if the object being compared to it isn’t null,
  - is a `String` object, and
  - represents the same sequence of characters as the object to which it’s being compared.

Comparing reference variables to instance values

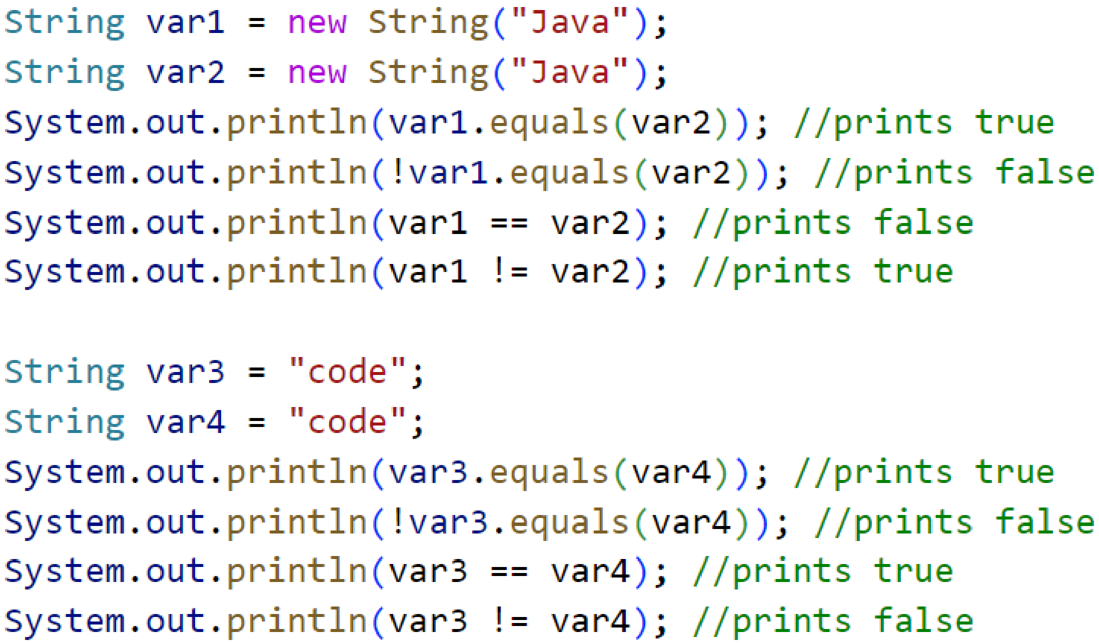


```java
String var1 = new String("Java");
String var2 = new String("Java");
System.out.println(var1.equals(var2)); //prints true
System.out.println(var1 == var2); //prints false

String var3 = "code";
String var4 = "code";
System.out.println(var3.equals(var4)); //prints true
System.out.println(var3 == var4); //prints true
```

** The operator == compares whether the reference variables refer to the same objects, and the method `equals` compares the `String` values for equality.*
*Always use the **`equals`** method to compare two Strings for equality.*
Never use the == operator for this purpose.

Determining inequality of Strings

- You can use the operator != to compare the inequality of objects referred to by `String` variables. It’s the inverse of the operator ==.



```
String var1 = new String("Java");
String var2 = new String("Java");
System.out.println(var1.equals(var2)); //prints true
System.out.println(!var1.equals(var2)); //prints false
System.out.println(var1 == var2); //prints false
System.out.println(var1 != var2); //prints true

String var3 = "code";
String var4 = "code";
System.out.println(var3.equals(var4)); //prints true
System.out.println(!var3.equals(var4)); //prints false
System.out.println(var3 == var4); //prints true
System.out.println(var3 != var4); //prints false
```

Mutable strings
StringBuilder

StringBuilder class

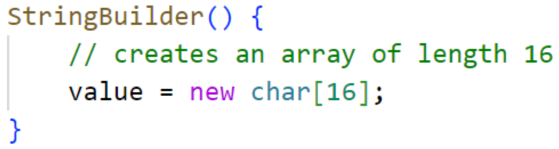
The class `StringBuilder` is defined in the package `java.lang`, and it has a **mutable** sequence of characters.

You should use the class `StringBuilder` when you’re dealing with larger strings or modifying the contents of a string often.

Creating StringBuilder objects

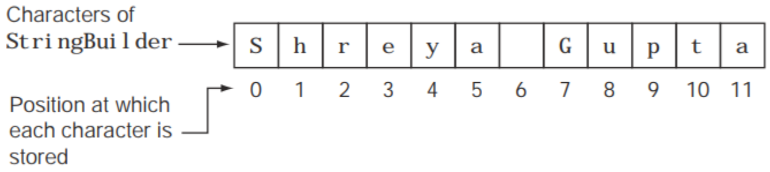
You can create objects of the class StringBuilder using multiple overloaded constructors, as follows:

```java
// no-argument constructor
StringBuilder sb1 = new StringBuilder();
```



```java
StringBuilder() {
    // creates an array of length 16
    value = new char[16];
}
```

```java
// constructor that accepts a StringBuilder object
StringBuilder sb2 = new StringBuilder(sb1);

// constructor that accepts an int value specifying initial capacity of StringBuilder object
StringBuilder sb3 = new StringBuilder(50);

// constructor that accepts a String
StringBuilder sb4 = new StringBuilder("Shreya Gupta");
```

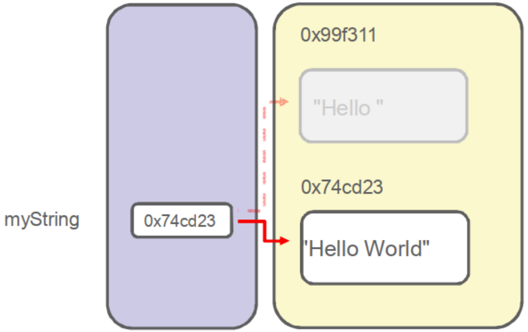

Characters of StringBuilder
Position at which each character is stored

S	h	r	e	y	a		G	u	p	t	a
0	1	2	3	4	5	6	7	8	9	10	11

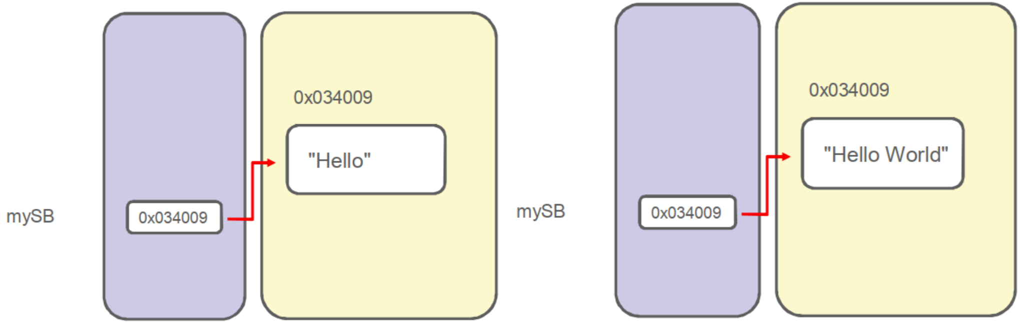
StringBuilder Advantages over String for Concatenation

You can create objects of the class StringBuilder using multiple overloaded constructors, as follows:

```java
String myString = "Hello";
myString = myString + " World";
```

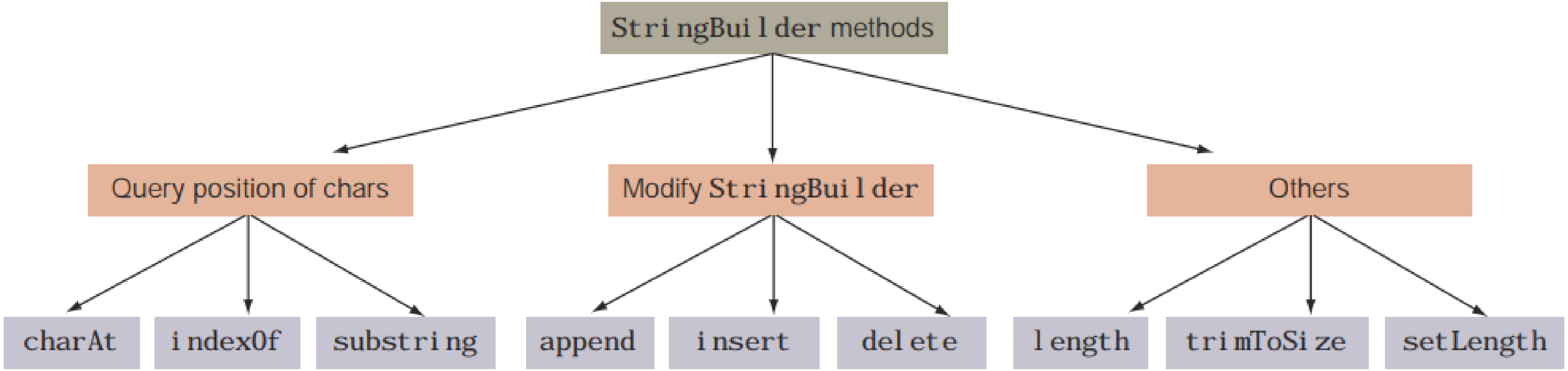


```java
StringBuilder mySB = new StringBuilder("Hello");
mySB.append(" World");
```



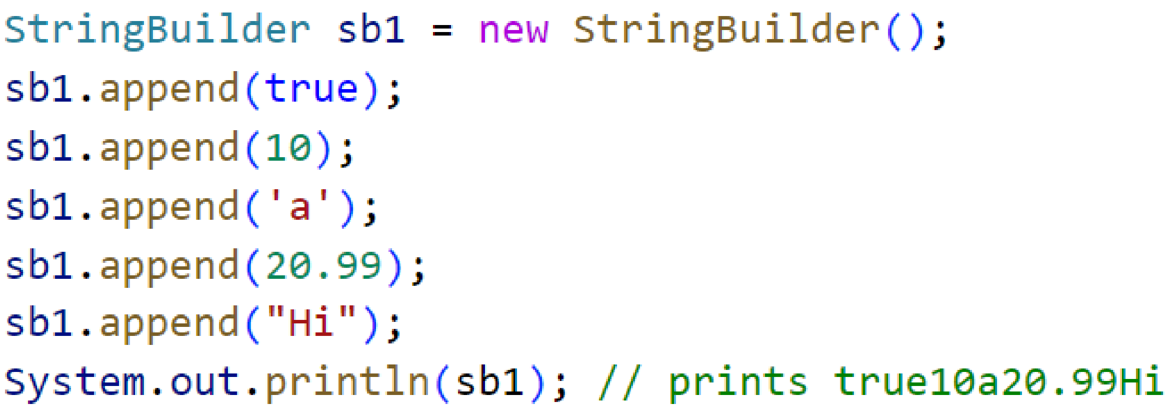
Methods of class StringBuilder

Many of the methods defined in the class `StringBuilder` work exactly like the versions in the class `String` - for example, methods such as `charAt`, `indexOf`, `substring`, and `length`.



append()

The **append** method adds the specified value at the end of the existing value of a `StringBuilder` object. This method accepts all the primitives, String, char array, and Object, as method parameters.

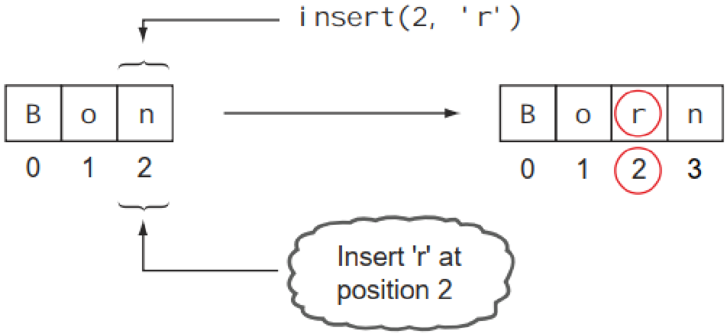


```
StringBuilder sb1 = new StringBuilder();
sb1.append(true);
sb1.append(10);
sb1.append('a');
sb1.append(20.99);
sb1.append("Hi");
System.out.println(sb1); // prints true10a20.99Hi
```

insert()

The **insert** method enables you to insert the requested data at a particular position.

```
StringBuilder sb1 = new StringBuilder("Bon");
sb1.insert(2, 'r'); // inserts r at position 2
System.out.println(sb1); // prints Born
```



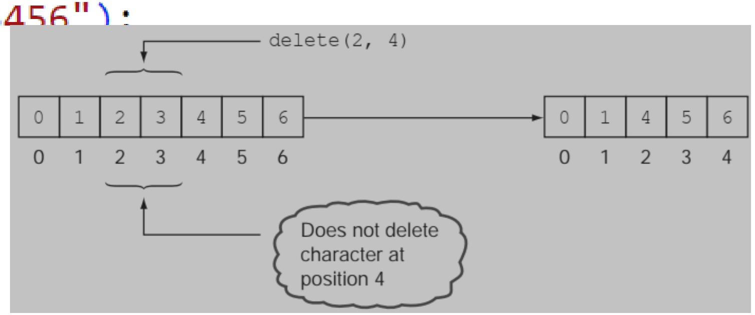
deleteCharAt() & delete()

The method **deleteCharAt** removes the char at the specified position.

```
StringBuilder sb1 = new StringBuilder("0123456");
sb1.deleteCharAt(2);
System.out.println(sb1); // prints 013456
```

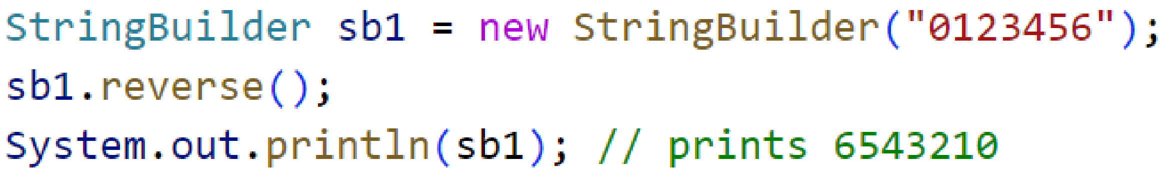
The method **delete** removes the characters in a substring of the specified StringBuilder.

```
StringBuilder sb1 = new StringBuilder("0123456");
sb1.delete(2, 4);
System.out.println(sb1); // prints 01456
```



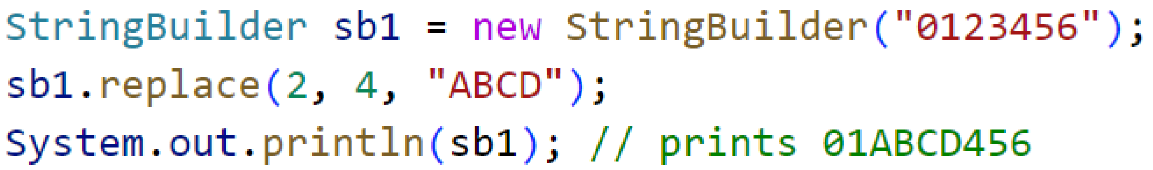
reverse() & replace()

As the name suggests, the **reverse** method reverses the sequence of characters of StringBuilder:



```
StringBuilder sb1 = new StringBuilder("0123456");
sb1.reverse();
System.out.println(sb1); // prints 6543210
```

The **replace** method replaces a sequence of characters, identified by their positions, with another String.



```
StringBuilder sb1 = new StringBuilder("0123456");
sb1.replace(2, 4, "ABCD");
System.out.println(sb1); // prints 01ABCD456
```

Thank you for your attention!