Chapter 8: String, Wrapper Class, and ArrayList

TAO Yida

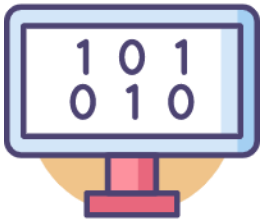
taoyd@sustech.edu.cn

Objectives

- **To create and manipulate strings**
  - Immutable character-string objects of class `String`
  - Mutable character-string objects of class `StringBuilder`
- **Learn wrapper classes of primitive types**
- **Learn ArrayList, whose capacity can be dynamically changed at runtime**

Characters: Fundamental Building Blocks of Java Programs

- Character encoding (字符编码): convert characters to numbers, in order to store and transmit them more effectively
- Unicode: Motivated by the need to encode characters in all languages without conflicts (1-to-1 mapping between character and numbers)



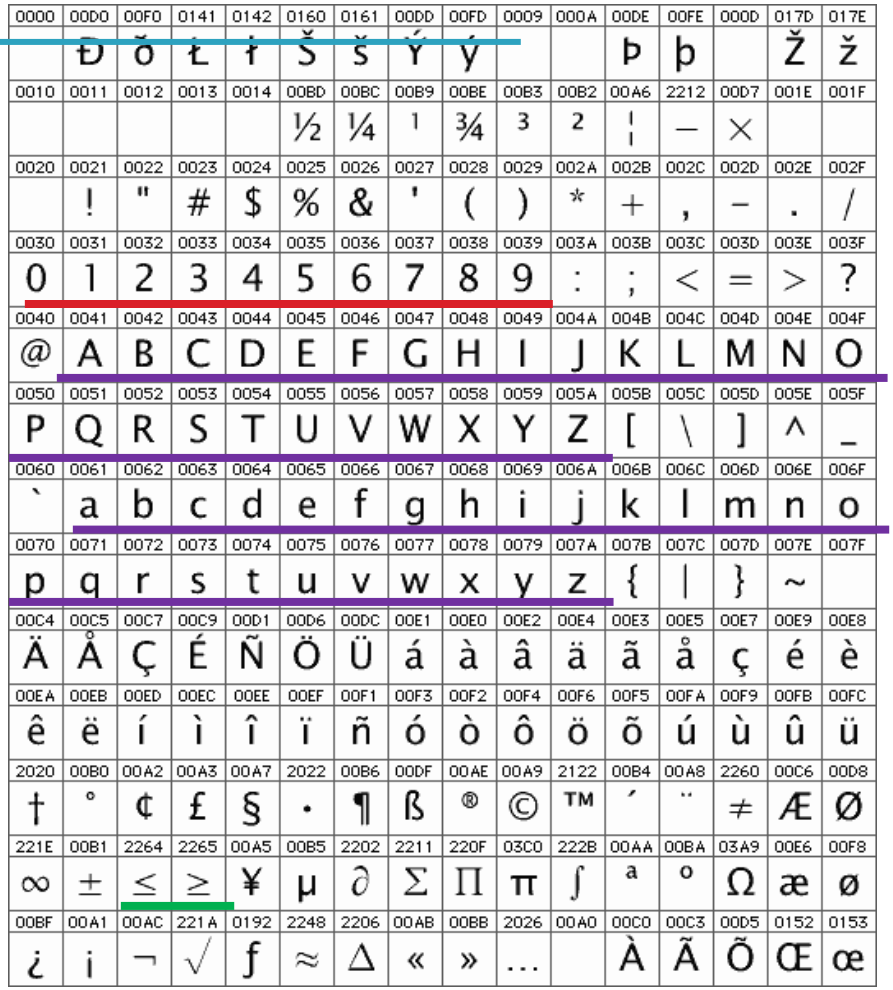
Characters: Fundamental Building Blocks of Java Programs

Line Feed
'\n' (LF)
(a white-space char)

Digits

Letters

Operators

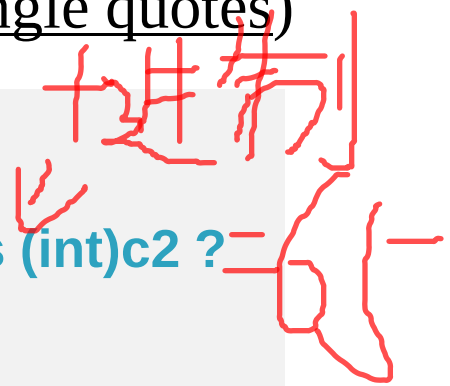


Unicode table
(万国码表)

The Primitive Type char

- The `char` data type is a single 16-bit Unicode character
  - '\u0000' – '\uffff': 65536 characters, covering characters for almost all modern languages, and a large number of symbols
- Programs often contain character literals (in single quotes)

```
char c1 = '\u0030';
char c2 = '\u0041';
char c3 = '\u4e2d';
char c4 = '\u56fd';
System.out.printf("%c %c %c %c", c1, c2, c3, c4);
```



Prints: 0 A 中 国

String

- A string is a sequence of characters

```
"I like Java programming"
```

- A string may include letters, digits and various special characters, such as +, -, *, / and $.

```
"I \u2665 Java programming"
```

```
I ♥ Java programming
```

Unicode escape sequence for chars you cannot find on keyboard:
\u + a code point in hexadecimal（十六进制码位）

Creating String Objects: Two Ways

- `String` objects can be created by using the `new` keyword and various `String` constructors
  - `String s1 = new String("hello world");`
  - `String s2 = new String(); // empty string (length is 0)`
  - `String s3 = new String(s1);`
  - `char[] charArray = {'h', 'e', 'l', 'l', 'o'};`
  - `String s4 = new String(charArray);`
  - `String s5 = new String(charArray, 3, 2); // string “lo”`

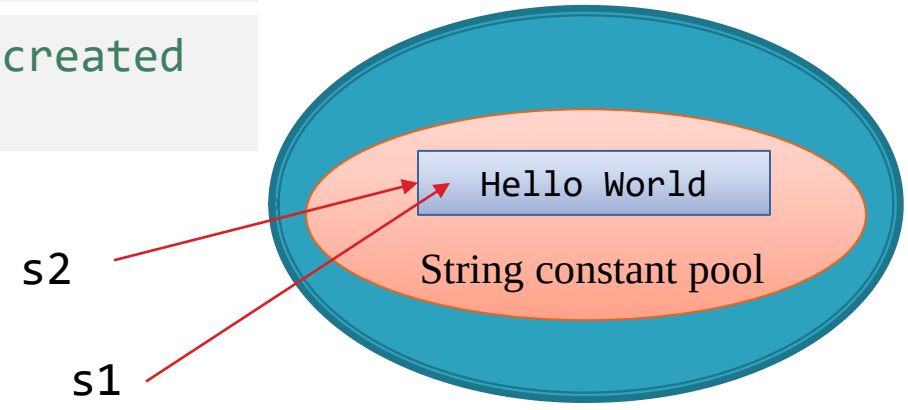
More at: https://docs.oracle.com/javase/10/docs/api/java/lang/String.html

Creating String Objects: Two Ways

- A string is an object of class `String`
- `String` objects can also be created by string literals (字面常量，a sequence of characters in double quotes)

```
String s1 = "Hello World";
```

```
// no new objects will be created
String s2 = "Hello World";
```



Heap
(memory area for storing objects and arrays)

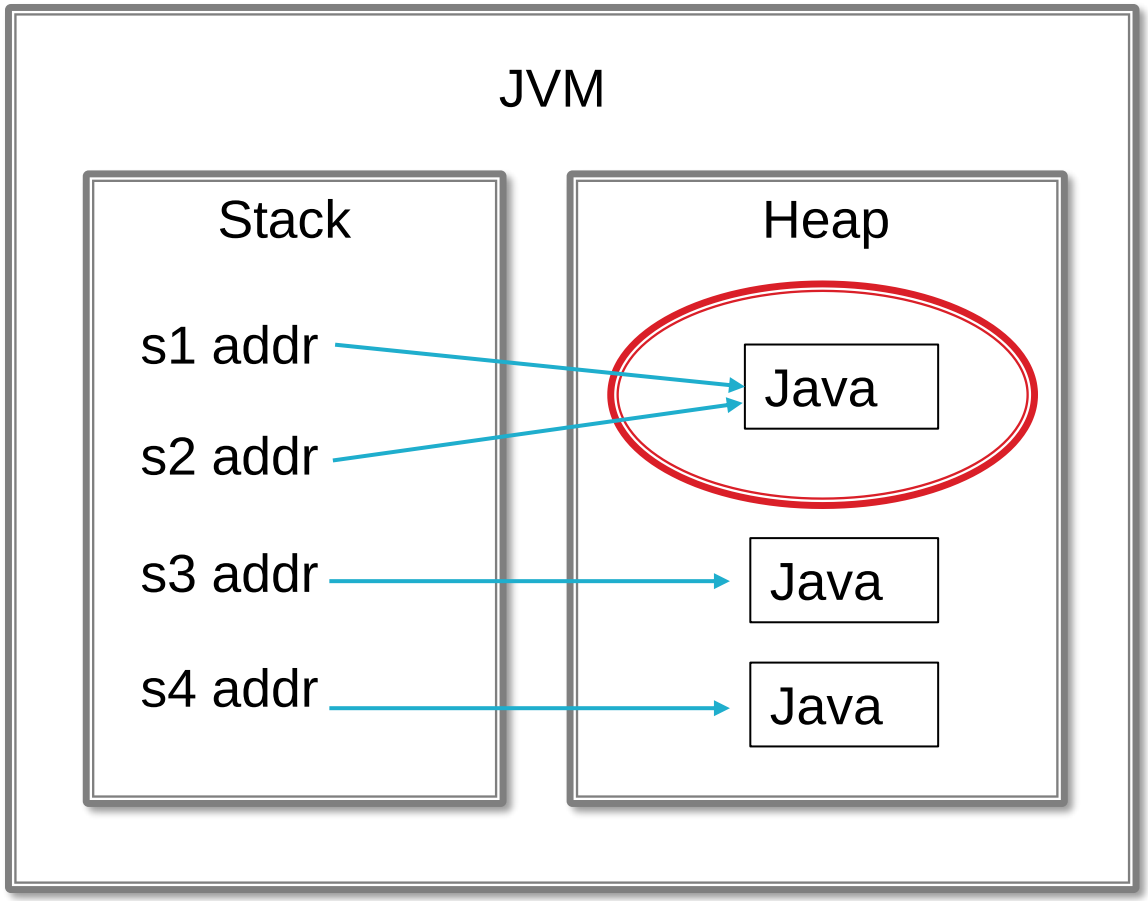
Using String literal vs new keyword

```
String s1 = "Java";
String s2 = "Java";

String s3 = new String("Java");
String s4 = new String("Java");
```

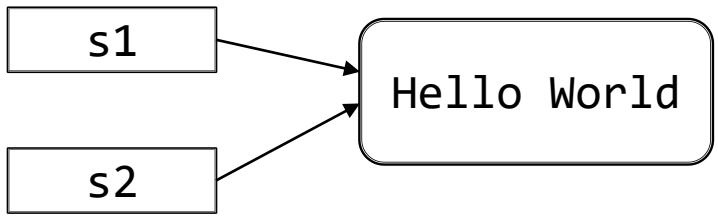
System.out.println(s1 == s2); // true
System.out.println(s3 == s4); // false

String Constant Pool:
Store string objects created by string literals



String Assignments

- A string may be assigned to a `String` reference.
  - `String s1 = “Hello World";`
  - The statement initializes `String` variable `s` to refer to a `String` object that contains the string “hello world”.
  - `String s2 = s1;`
  - The statement makes `s2` and `s` to refer to (sometimes we say “point to”, they mean the same thing) the same `String` object.

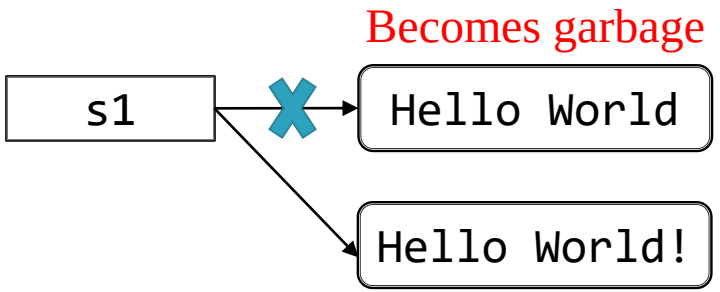


Immutability（不可变性）

- In Java, `String` objects are immutable. Strings are constants; their values cannot be changed after they are created.

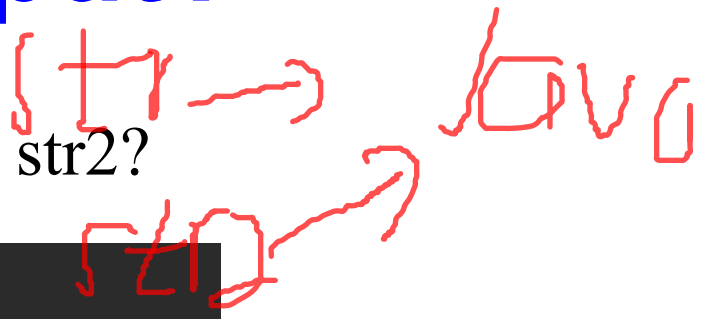
- Any modification creates a new `String` object

```
String s1 = "Hello World";
s1 = s1.concat("!");
```



What's the output?

- What's the value of str and str2?



```
String str = "Java";
String str2 = str;
str = str.concat(" course");
```

```
String str = "Java";
String str2 = str;
str.concat(" course");
```

Think: Why String is immutable?

`String` Methods

- `length` returns the length of a string (i.e., the number of characters)
- `charAt` obtains the character at a specific location in a string
- `getChars` retrieves a set of characters from a string as a `char` array
- *These are instance methods that can be invoked on specific objects. Calling them requires a non-null object reference.*

The Method length

`int length()`	Returns the length of this string.

```
public class StringExamples {
    public static void main(String[] args) {
        String s1 = "hello world";
        System.out.printf("s1: %s", s1);
        System.out.printf("\nLength of s1: %d", s1.length());
    }
}
```

```
s1: hello world
Length of s1: 11
```

The Method charAt

`char`	**`charAt`**`(int index)`	Returns the `char` value at the specified index.

```
public class StringExamples {
    public static void main(String[] args) {
        String s1 = "hello world";
        System.out.printf("s1: %s", s1);

        for(int count = s1.length() - 1; count >=0; count--) {
            System.out.printf("%c", s1.charAt(count));
        }
    }
}
```

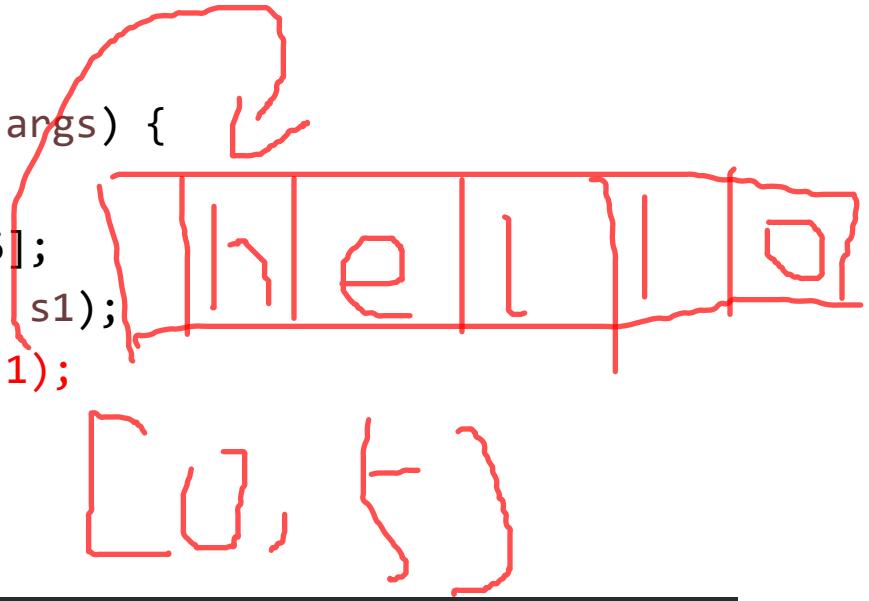
What's the output?

The Method getChars

```
void  getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Copies characters from this string into the destination character array.

```
public class StringExamples {
    public static void main(String[] args) {
        String s1 = "hello world";
        char[] charArray = new char[6];
        System.out.printf("s1: %s\n", s1);
        s1.getChars(0, 5, charArray, 1);
        for(char c : charArray) {
            System.out.print(c);
        }
    }
}
```




```
s1: hello world
hello
```

Comparing Strings

- When primitive-type values are compared with ==, the result is `true` if both values are identical.

```
int a = 2, b =2;
if (a == b) System.out.println("a = b"); // prints a = b
```

- When references (memory addresses) are compared with ==, the result is `true` if both references refer to the same object in memory.

```
String s1 = "Hello World";
String s2 = "Hello World";
if(s1 == s2) System.out.println("s1 = s2"); // prints s1 = s2
```

Comparing Strings

```
String s1 = "Hello World";
String s2 = s1 + "";
if(s1 == s2) System.out.println("s1 = s2"); // prints s1 = s2?
```

- No. The condition will evaluate to `false` because the `String` variables `s1` and `s2` refer to two different `String` objects, although the strings contain the same sequence of characters.
- To compare the actual contents (or state information) of objects (strings are objects) for equality, a method `equals` must be invoked.

The Method `equals`

- Method `equals` tests any two objects for equality—the strings contained in the two `String` objects are identical.

```
String s1 = "Hello World";
String s2 = s1 + "";
if(s1.equals(s2)) System.out.println("s1 = s2"); // true
```

```
String s1 = "hello";
String s2 = "HELLO";
if(s1.equals(s2)) System.out.println("s1 = s2"); // false
```

The Method `equalsIgnoreCase`

- Method `equalsIgnoreCase` ignores whether the letters in each String are uppercase or lowercase when performing a comparison.

```
String s1 = "hello";

String s2 = "HELLO";

if(s1.equalsIgnoreCase(s2)) System.out.println("s1 = s2");
```

The condition evaluates to `true` and the program prints "`s1 = s2`"

The Method compareTo

```
String s1 = "hello";
String s2 = "HELLO";
int result = s1.compareTo(s2); // value of result?
```

compareTo compares two strings (lexicographical comparison):

Uses lexicographical comparison (字典序, Unicode表是字母表):
Compare the integer Unicode values that represent each character in each String.

The Method compareTo

```
String s1 = "hello";
String s2 = "HELLO";
int result = s1.compareTo(s2); // value of result?
```

compareTo compares two strings (lexicographical comparison):

- Returns 0 if the Strings are equal (identical contents).
- Returns a negative number if the String that invokes compareTo (s1) is less than the String that is passed as an argument (s2).
- Returns a positive number if the String that invokes compareTo (s1) is greater than the String that is passed as an argument (s2).

Comparing Strings

- What does it mean when we say a string `s1` is greater than another string `s2`?
  - When we sort last names, we naturally consider that "Jones" > "Smith", because the letter 'J' comes before 'S' in the alphabet of 26 letters.
  - All characters in computers are represented as numeric codes. The characters form an ordered set (a very large alphabet).
  - When the computer compares `String`s, it actually compares the numeric codes of the characters in the `String`s.

Comparing Strings


```
String s1 = "hello", s2 = "HELLO";
int result = s1.compareTo(s2));
```

32 = 0068 (HEX) – 0048 (HEX) (s1 > s2)

```
String s1 = "HE", s2 = "HELLO";
int result = s1.compareTo(s2));
```

-3 (s1 < s2, s2 has three more letters)

```
String s1 = "HEL", s2 = "HEllo";
int result = s1.compareTo(s2));
```

-32 (s1 < s2)

Methods startsWith & endsWith

The methods `startsWith` and `endsWith` determine whether a string starts or ends with the method argument, respectively

```
String s1 = "Hello World";
if(s1.startsWith("He")) System.out.print("true"); // true
```

```
String s1 = "Hello World";
if(s1.startsWith("llo", 2)) System.out.print("true"); // true
```

```
String s1 = "Hello World";
if(s1.endsWith("ld")) System.out.print("true"); // true
```

Locating Characters in Strings

```
String s = "abcdefghijklmabcdefghijklm";
System.out.println(s.indexOf('c'));  // 2
System.out.println(s.indexOf('$'));  // -1
System.out.println(s.indexOf('a', 1));  // 13
```

- `indexOf` locates the first occurrence of a character in a `String`.
  - If the method finds the character, it returns the character's index in the `String`; otherwise, it returns `-1`.
- Two-argument version of `indexOf`:
  - Take one more argument: the starting index at which the search should begin.

Locating Characters in Strings

```
String s = "abcdefghijklmabcdefghijklm";
System.out.println(s.lastIndexOf('c')); // 15
System.out.println(s.lastIndexOf('$')); // -1
System.out.println(s.lastIndexOf('a', 8)); // 0
```

- `lastIndexOf` locates the last occurrence of a character in a `String`.
  - The method searches from the end of the `String` toward the beginning.
  - If it finds the character, it returns the character's index in the `String`; otherwise, it returns –1.
- Two-argument version of `lastIndexOf`:
  - The character and the index from which to begin searching backward.

Locating Substrings in Strings

```
String s = "abcdefghijklmabcdefghijklm";
System.out.println(s.indexOf("def"));     // 3
System.out.println(s.indexOf("def", 7));  // 16
System.out.println(s.indexOf("hello"));   // -1
System.out.println(s.lastIndexOf("def")); // 16
System.out.println(s.lastIndexOf("def", 7));  // 3
System.out.println(s.lastIndexOf("hello"));   // -1
```

- The versions of methods `indexOf` and `lastIndexOf` that take a `String` as the first argument perform identically to those described earlier except that they search for sequences of characters (or substrings) that are specified by their `String` arguments.

Extracting Substrings from Strings

```
String s = "abcdefghijklmabcdefghijklm";
System.out.println(s.substring(20));  // hijklm
System.out.println(s.substring(3, 6)); // def
```

- `substring` methods create a new `String` object by copying part of an existing `String` object.
- The one-integer-argument version specifies the starting index (inclusive) in the original `String` from which characters are to be copied.
- Two-integer-argument version specifies the starting index (inclusive) and ending index (exclusive) to copy characters in the original `String`.

Concatenating (拼接) Strings

```
String s1 = "Happy ";
String s2 = "Birthday";
System.out.println(s1.concat(s2)); // Happy Birthday
System.out.println(s1); // Happy
```

- `String` method `concat` concatenates two `String` objects and returns a new `String` object containing the characters from both original `String`s.
- The original `String`s to which `s1` and `s2` refer are not modified (recall that `String`s are immutable).

Recall The Immutability of Strings

```
String s1 = "Hello";
s1.concat(" world");
System.out.println(s1); // prints "Hello"
```

- Any attempt to modify a `String` object (e.g., the call to the `concat()` method above) creates a new object. The original `String` object remain unchanged.

Why Strings are Immutable?

- In the String constant pool, a String object is likely to have one or many references.
- If several references point to same String without even knowing it, it would be bad if one of the references modified that String value.

Mainly for security reasons (database username, password are often passed as Strings in Java programming, imagine the consequence if `String` objects are mutable)

String Method replace

```
String s1 = "Hello";
System.out.println(s1.replace('l', 'L')); // HeLLo
System.out.println(s1.replace("ll", "LL")); // HeLLo
```

- `replace` returns a new `String` object in which every occurrence of the first character argument is replaced with the second character argument.
- Another version of method `replace` enables you to replace substrings rather than individual characters (every occurrence of the first substring is replaced).

String Case Conversion Methods

```
String s1 = "Hello";
System.out.println(s1.toUpperCase());  // HELLO
System.out.println(s1.toLowerCase());  // hello
```

- `String` method `toUpperCase` returns a new `String` object with uppercase letters where corresponding lowercase letters exist in the original.
- `String` method `toLowerCase` returns a new `String` object with lowercase letters where corresponding uppercase letters exist in the original.

String Method trim

- `trim` returns a new `String` object that removes all white-space characters at the beginning or end of the `String` on which `trim` operates.

```
String s1 = " spaces ";

System.out.println(s1.trim());  //prints “spaces”
```

String Method toCharArray

- `toCharArray` creates a new character array containing a copy of the characters in the string.

```
String s1 = "hello";
char[] charArray = s1.toCharArray();
for(char c : charArray) System.out.print(c);
```

The `for` loop prints each of the five chars in "`hello`"

Tokenizing Strings (分词)

- When you read a sentence, your mind breaks it into tokens—individual words and punctuation marks that convey meaning to you.

- `String` method `split` breaks a `String` into its component tokens, separated from each other by delimiters (分隔符), typically white-space characters such as space, tab, new line, carriage return.

Tokenizing Strings

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a sentence and press Enter");
String sentence = input.nextLine();
String[] tokens = sentence.split(" ");
System.out.printf("Number of tokens: %d\n", tokens.length);
for(String token : tokens) System.out.println(token);
input.close();
```

```
Enter a sentence and press Enter
This is a sentence with seven tokens
Number of tokens: 7
This
is
a
sentence
with
seven
tokens
```

How about sentence.split("is")?

`String` Method `valueOf`

- Every object in Java has a `toString` method that enables a program to obtain the object's `String` representation.
- Unfortunately, this technique cannot be used with primitive types because they do not have methods.
- Class `String` provides `static` methods (associated with class, no need to create objects for their invocation) that take an argument of any type and convert it to a `String` object.

```
static String   valueOf(boolean b)
static String   valueOf(char c)
static String   valueOf(char[] data)
static String   valueOf(char[] data, int offset, int count)

static String   valueOf(double d)
static String   valueOf(float f)
static String   valueOf(int i)
static String   valueOf(long l)
```

```
boolean booleanValue = true;
char charValue = 'Z';
int intValue = 7;
long longValue = 10000000000L;
float floatValue = 2.5f;
double doubleValue = 33.3333; // no f suffix, double is default
char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
System.out.println(String.valueOf(booleanValue));
System.out.println(String.valueOf(charValue));
System.out.println(String.valueOf(intValue));
System.out.println(String.valueOf(longValue));
System.out.println(String.valueOf(floatValue));
System.out.println(String.valueOf(doubleValue));
System.out.println(String.valueOf(charArray));
```

```
true
Z
7
10000000000
2.5
33.3333
abcdef
```


Objectives

- **To create and manipulate strings**
  - Immutable character-string objects of class `String`
  - Mutable character-string objects of class `StringBuilder`
- **Learn wrapper classes of primitive types**
- **Learn ArrayList, whose capacity can be dynamically changed at runtime**

Class `StringBuilder`

- `String` objects are immutable. Can we create mutable character-string objects in Java?
- Yes. The class `StringBuilder` helps create and manipulate dynamic string information—that is, modifiable strings.
- Every `StringBuilder` is capable of storing a number of characters specified by its capacity.
- If a `StringBuilder`'s capacity is exceeded, the capacity automatically expands to accommodate additional characters.

StringBuilder Constructors

- We demonstrate three widely-used constructors

Default initial capacity is 16 chars

```
StringBuilder buffer1 = new StringBuilder();

StringBuilder buffer2 = new StringBuilder(10);

StringBuilder buffer3 = new StringBuilder("hello");

System.out.printf("buffer1 = \"%s\"\n", buffer1);

System.out.printf("buffer2 = \"%s\"\n", buffer2);

System.out.printf("buffer3 = \"%s\"\n", buffer3);
```

```
buffer1 = ""

buffer2 = ""

buffer3 = "hello"
```


StringBuilder Method append

- Class `StringBuilder` provides several `append` methods to allow values of various types to be appended to the end of a `StringBuilder` object.

- Overloaded append() are provided for each of the primitive types, and for character arrays, Strings, Objects, and more.

```
append(boolean b)
append(char c)
append(char[] str)
append(char[] str, int offset, int len)
append(double d)
append(float f)
append(int i)
append(long lng)
append(CharSequence s)
append(CharSequence s, int start, int end)
append(Object obj)
append(String str)
append(StringBuffer sb)
```

```
1.  String string = "goodbye";
2.  char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
3.  boolean booleanValue = true;
4.  char charValue = 'Z';
5.  int intValue = 7;
6.  long longValue = 10000000000L;
7.  float floatValue = 2.5f;
8.  double doubleValue = 33.3333;
9.  StringBuilder buffer = new StringBuilder();
10. StringBuilder lastBuffer = new StringBuilder("last buffer");

11. buffer.append(string); buffer.append("\n");
12. buffer.append(charArray); buffer.append("\n");
13. buffer.append(charArray, 0, 3); buffer.append("\n");
14. buffer.append(booleanValue); buffer.append("\n");
15. buffer.append(charValue); buffer.append("\n");
16. buffer.append(intValue); buffer.append("\n");
17. buffer.append(longValue); buffer.append("\n");
18. buffer.append(floatValue); buffer.append("\n");
19. buffer.append(doubleValue); buffer.append("\n");
20. buffer.append(lastBuffer);

21. System.out.printf("buffer contains:\n%s", buffer.toString());
```

```
buffer contains:
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.3333
last buffer
```

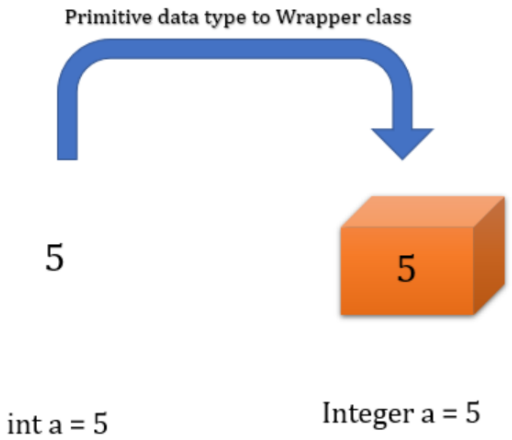
Here we still use the same `StringBuilder` object reference, because `StringBuilder` objects are mutable.

Objectives

- To create and manipulate strings
  - Immutable character-string objects of class `String`
  - Mutable character-string objects of class `StringBuilder`
- **Learn wrapper classes of primitive types**
- **Learn ArrayList, whose capacity can be dynamically changed at runtime**

Wrapper Classes

- Java has 8 primitive types: `boolean`, `char`, `double`, `float`, `byte`, `short`, `int` and `long`
- Java also provides 8 type-wrapper classes—`Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` and `Long`—that enable primitive-type values to be treated as objects.



Be careful: not `Int` or `Char`

Character Class

- The class `Character` is the type-wrapper class for the primitive type `char`
- `Character` provides methods (mostly static ones) for convenience in processing individual `char` values
  - `isDigit(char c)`
  - `isLetter(char c)`
  - `isLowerCase(char c)`

Useful Character Methods

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter a character and press Enter:");
String input = sc.next();
char c = input.charAt(0);

System.out.printf("is digit: %b\n", Character.isDigit(c));
System.out.printf("is identifier start: %b\n", Character.isJavaIdentifierStart(c));
System.out.printf("is letter: %b\n", Character.isLetter(c));
System.out.printf("is lower case: %b\n:", Character.isLowerCase(c));
System.out.printf("is upper case: %b\n", Character.isUpperCase(c));
System.out.printf("to upper case: %c\n", Character.toUpperCase(c));
System.out.printf("to lower case: %c\n", Character.toLowerCase(c));

sc.close();
```

Useful Character Methods

```
Enter a character and press Enter:
A
is digit: false
is identifier start: true
is letter: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

```
Enter a character and press Enter:
8
is digit: true
is identifier start: false
is letter: false
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

Java identifiers can only star with a letter, an underscore (_), or a dollar sign ($)

Other Useful Methods

- `Integer.parseInt(String s)` : parses the string argument as a decimal integer value
  - `Integer.parseInt("123")` returns an integer 123
  - `Integer.pasetInt ("123abc")` returns a `NumberFormatException`

- `Double.parseDouble(String s)...`

- Check the Java API documentation for more details
  - https://docs.oracle.com/javase/10/docs/api/

Autoboxing & Unboxing

- Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
  - For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on.
- If the conversion goes the other way, this is called unboxing.

Autoboxing & Unboxing

```
ArrayList<Double> ld = new ArrayList<>();
// 3.1416 is autoboxed through method invocation.
// Compiler turns it into ld.add(Double.valueOf(3.1416));
ld.add(3.1416);


// Unboxing through assignment
// Compiler turns it into ld.get(0).doubleValue();
double pi = ld.get(0);
System.out.println("pi = " + pi);
```

Objectives

- To create and manipulate strings
  - Immutable character-string objects of class `String`
  - Mutable character-string objects of class `StringBuilder`
- Learn wrapper classes of primitive types
- **Learn ArrayList, whose capacity can be dynamically changed at runtime**

ArrayList

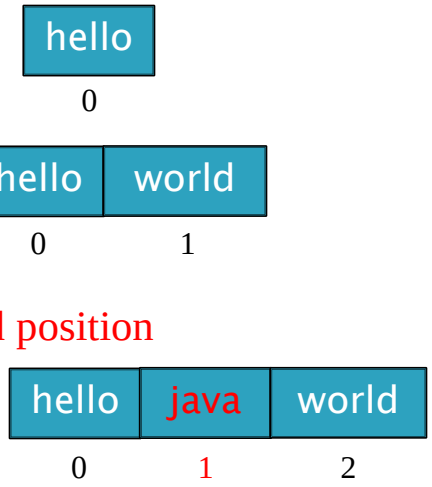
- Arrays store sequences of objects (and primitive values). Arrays do not change their size at runtime to accommodate additional elements.
- `ArrayList<T>` can dynamically change its size at runtime.
- `ArrayList<T>` is a **generic class**, where `T` is a placeholder for the type of elements that you want the `ArrayList` to hold.

```
ArrayList<String> list;
```

Declares `list` as an `ArrayList` collection to store only `String` objects

Adding Elements to ArrayList

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>(); // the list is empty after creation
    printList(list);     // prints nothing since the list is empty
    list.add("hello");   // adding an element to the end of the list
    printList(list);     // prints "hello"
    list.add("world");   // adding one more element to the end
    printList(list);     // prints "hello world"
    list.add(1, "java");// adding one more element to the specified position
    printList(list);     // prints "hello java world"
}
public static void printList(ArrayList<String> list) { // traverse the list
    for(String s : list) System.out.printf("%s ", s); // enhanced for loop
    System.out.println();
}
```



Removing Elements from `ArrayList`

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(null);
list.add(null);
list.add(2);
```

```
for(int i=0;i<list.size();i++){
    if(list.get(i) == null){
        list.remove(i);
    }
}
```

Content of list: [1, null, 2]

Removing Elements from `ArrayList`

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(null);
list.add(null);
list.add(2);
```

```
for (Iterator<Integer> i = list.iterator(); i.hasNext(); ) {
    if (i.next() == null) {
        i.remove();
    }
}
```

Content of list: [1, 2]

Sorting Elements in ArrayList

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(new Integer(5));
    list.add(new Integer(124));
    list.add(new Integer(-8));
    printList(list);
    Collections.sort(list);  // sort the elements in the list into ascending order
    printList(list);
}
public static void printList(ArrayList<Integer> list) {
    for(Integer s : list) System.out.printf("%d ", s.intValue());
    System.out.println();
}
```

ArrayList cannot hold primitive data types

```
-8 5 124
```

`java.util.Collections` class provides ***static*** methods that operate on collections (e.g., `shuffle, reverse, sort`)