Strings, Wrapper Classes, and File I/O

Java™ How to Program

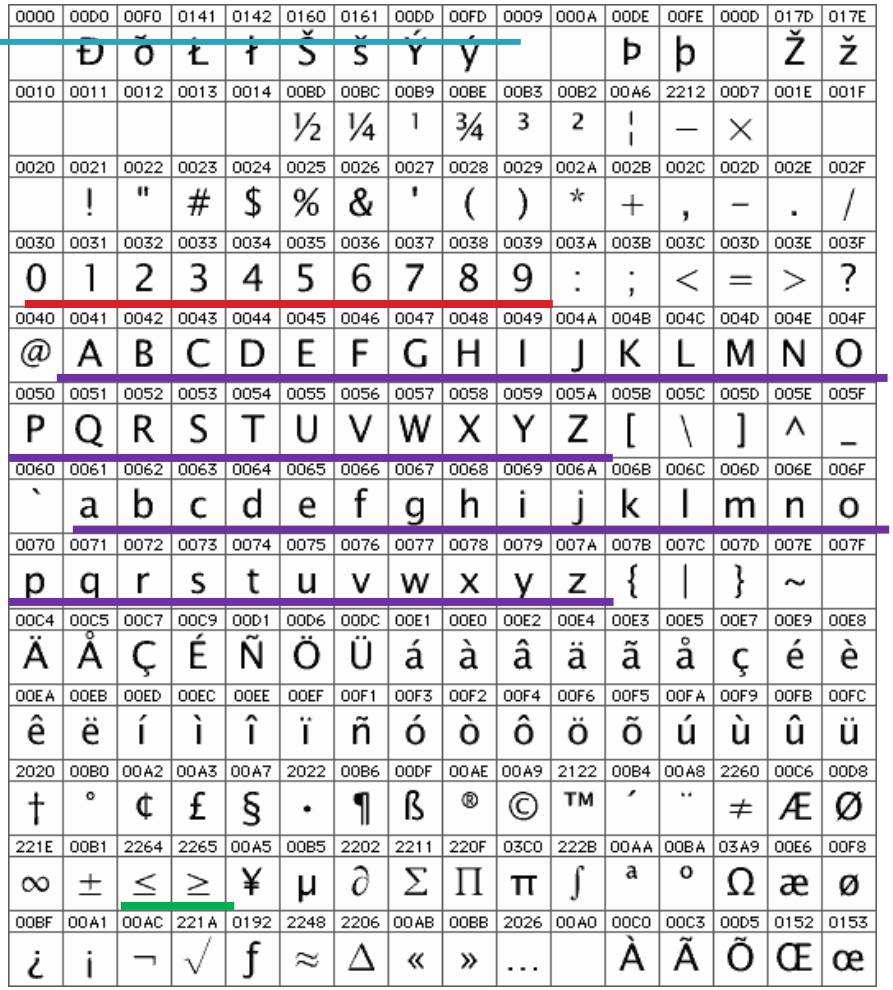
Instructor: Zhuozhao Li

Objectives

- **To create and manipulate strings**
  - Immutable (不可变的) character-string objects of class `String`
  - Mutable (可变的) character-string objects of class `StringBuilder`
- **To create and manipulate objects of class `Character`**
- **Learn wrapper classes of primitive types**
- **File I/O （文件输入、输出）**
  - Creating, reading, updating, and deleting files

Characters: Fundamental Building Blocks of Java Programs

Line Feed
'\n' (LF)



0000	00D0	00F0	0141	0142	0160	0161	00DD	00FD	0009	000A	00DE	00FE	000D	017D	017E
	Đ	ð	Ł	ł	Š	š	Ý	ý			Þ	þ		Ž	ž
0010	0011	0012	0013	0014	00BD	00BC	00B9	00BE	00B3	00B2	00A6	2212	00D7	001E	001F
					½	¼	¹	¾	³	²	¦	−	×		
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
	!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
00C4	00C5	00C7	00C9	00D1	00D6	00DC	00E1	00E0	00E2	00E4	00E3	00E5	00E7	00E9	00E8
Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
00EA	00EB	00ED	00EC	00EE	00EF	00F1	00F3	00F2	00F4	00F6	00F5	00FA	00F9	00FB	00FC
ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
2020	00B0	00A2	00A3	00A7	2022	00B6	00DF	00AE	00A9	2122	00B4	00A8	2260	00C6	00D8
†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø
221E	00B1	2264	2265	00A5	00B5	2202	2211	220F	03C0	222B	00AA	00BA	03A9	00E6	00F8
∞	±	≤	≥	¥	µ	∂	Σ	Π	π	∫	ª	º	Ω	æ	ø
00BF	00A1	00AC	221A	0192	2248	2206	00AB	00BB	2026	00A0	00C0	00C3	00D5	0152	0153
¿	¡	¬	√	ƒ	≈	Δ	«	»	…		À	Ã	Õ	Œ	œ

Digits

Letters

Operators

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)

```
System.out.print('\u5357');
System.out.print('\u65B9');
System.out.print('\u79D1');
System.out.print('\u6280');
System.out.print('\u5927');
System.out.print('\u5B66');
```

Prints： 南方科技大学

String （字符串）

- 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

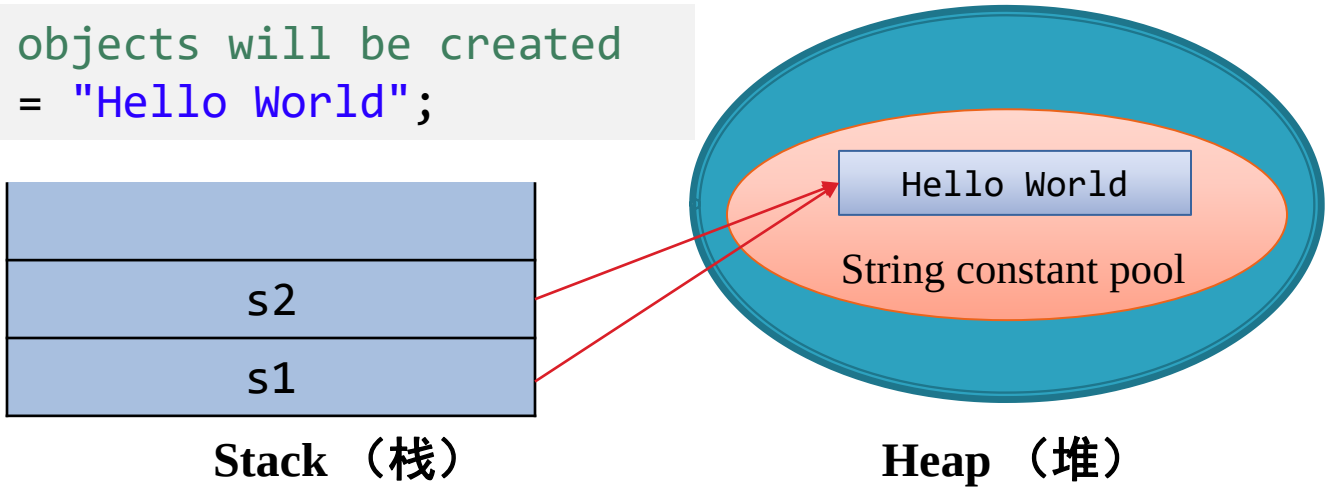
Unicode escape sequence for chars you cannot find on keyboard:
\u + a code point (in hexadecimal，16进制)

Creating String Objects: Two Ways

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

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

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



Creating String Objects: Two Ways

- `String` objects can also 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, 1, 3); // string “ell”
```

More at: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

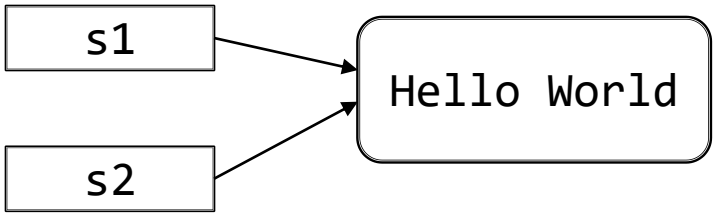
String Assignments

- A string may be assigned to a `String` reference.
  - `String s = "hello world";`
  - The statement initializes `String` variable `s` to refer to a `String` object that contains the string "hello world".
  - `String s2 = s;`
  - 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）

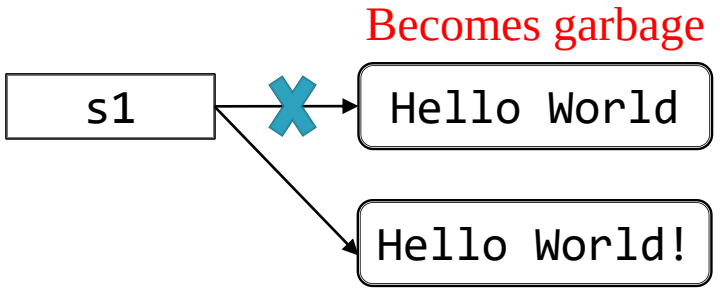
- In Java, `String` objects are immutable. Strings are constants; their values cannot be changed after they are created. Because `String` objects are immutable, they can be shared safely.

```
String s1 = "Hello World";
String s2 = s1;
```



- Any modification creates a new `String` object

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



`String` Methods

- `length` returns the length of a string (i.e., the number of characters)
- `charAt` helps obtain the character at a specific location in a string
- `getChars` helps retrieve a set of characters from a string as a `char` array
- *These are instance methods that interact with the specific data of objects. Calling them requires an 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);
        System.out.print("\nThe string reversed is: ");
        for(int count = s1.length() - 1; count >=0; count--) {
            System.out.printf("%c", s1.charAt(count));
        }
    }
}
```

```
s1: hello world
The string reversed is: dlrow olleh
```

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[5];
        System.out.printf("s1: %s\n", s1);
        s1.getChars(0, 5, charArray, 0);
        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 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
```

- Uses lexicographical comparison （字典序，Unicode）—it compares the integer Unicode values that represent each character in each `String`.

```
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?
```

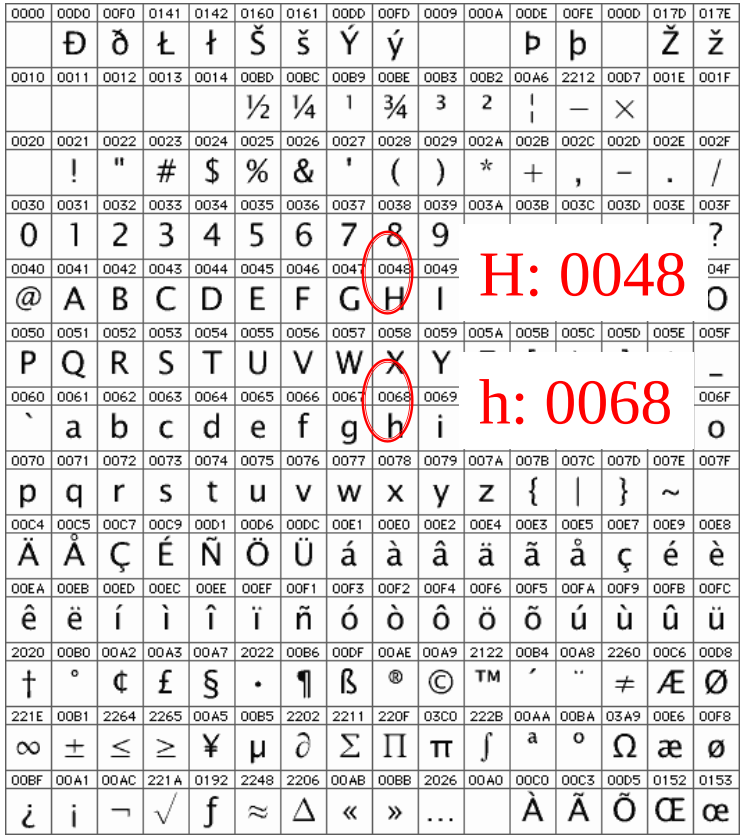
`compareTo` compares two strings (lexicographical comparison):

- Returns 0 if the `String`s 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 = 0x0068 – 0x0048 (s1 > s2)

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

-3 (s1 < s2)

```
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
```

The Method regionMatches

```
String s1 = "Hello World";
String s2 = "hello world";
boolean result = s1.regionMatches(0, s2, 0, 5); // true or false?
```

`regionMatches` compare portions of two `String`s for equality:

- The first argument is the starting index in the `String` that invokes the method `(s1)`.
- The second argument is a comparison `String`.
- The third argument is the starting index in the comparison `String`.
- The last argument is the number of characters to compare between the two `String`s.
- Returns `true` only if the specified number of characters are lexicographically equal.

The Method regionMatches

```
String s1 = "Hello World";
String s2 = "hello world";
boolean result = s1.regionMatches(true, 0, s2, 0, 5); // true
```

`regionMatches` is overloaded (it has a five-argument version):

- When the first argument is `true`, the method ignores the case of the characters being compared.
- The remaining arguments are identical to those described for the four-argument `regionMatches` method

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).

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.
- An overloaded version of method `replace` enables you to replace substrings rather than individual characters.

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` 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 (回车).

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
```

`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 valueOf` methods that take an argument of any type and convert it to a `String` object.

```
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
```

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

```
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 overloaded `append` methods to allow values of various types to be appended (追加) to the end of a `StringBuilder` object.
- Versions are provided for each of the primitive types, and for character arrays, Strings, Objects, and more.

```
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 lastBuffer = new StringBuilder("last buffer");
10. StringBuilder buffer = new StringBuilder();

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.

Wrapper Classes （包装类）

- 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();
```

Other Useful Methods

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

- Java documentations, search online, etc.
  - https://docs.oracle.com/javase/8/docs/api/

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
```

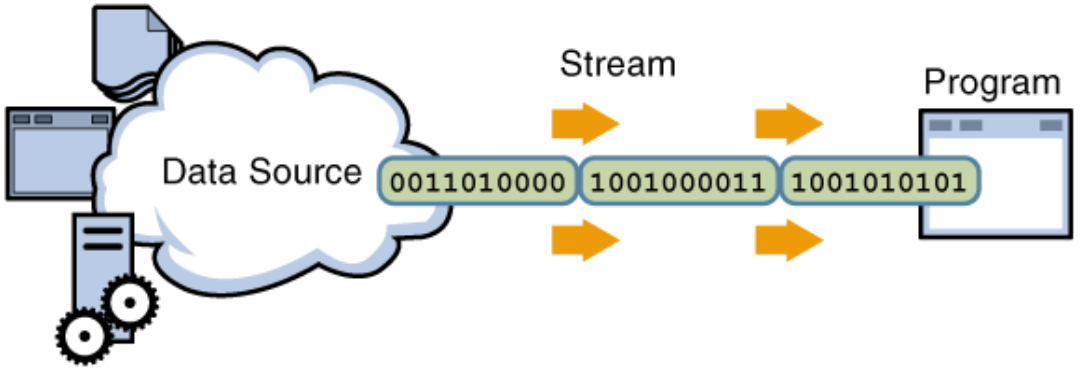
Java identifiers can only star with a letter, an underscore (_), or a dollar sign ($)

I/O Streams

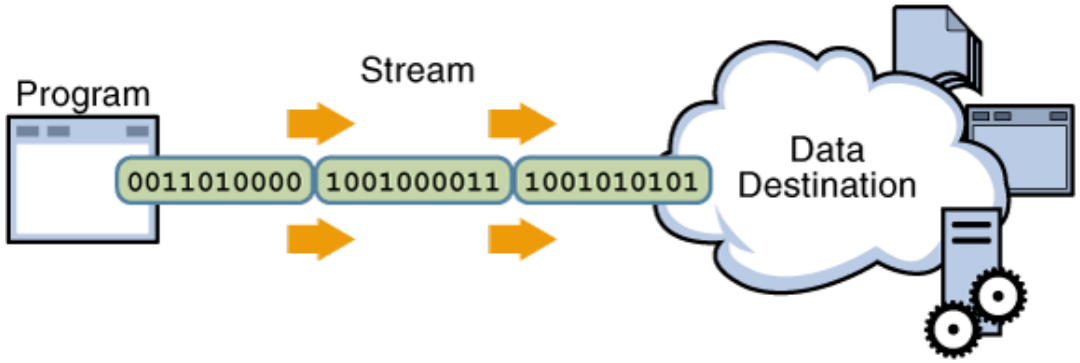
- I/O = Input/Output
  - Input can be from keyboard or a file
  - Output can be to display (screen) or a file
- An I/O stream is a communication channel that a program has with the outside world. It is used to transfer data items in succession
- A stream connects a program to an I/O object
  - System.out connects a program to the screen
  - System.in connects a program to the keyboard

I/O Streams

- Input stream



- Output stream



File I/O

- Variables, arrays, objects, etc.
  - Data stored in variables and arrays is temporary — it's lost when a local variable goes out of scope or when the program terminates.

- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

Stream Types

- All data and programs are ultimately zeros and ones
  - Each digit can have one of two values, hence binary
  - Bit （比特） is one binary digit, byte （字节） is a group of eight bits
- Byte streams
  - Byte streams （字节流） perform input and output of 8-bit bytes in binary format. They read and write data one byte at a time
- Character streams
  - Character streams（字符流） are used to perform input and output for 16-bit Unicode, 2 bytes
- Files created using byte streams -> binary files
- Files created using character streams -> text files

Text Files versus Binary Files

- Text files are more readable by humans

- Binary files are more efficient
  - Computers read and write binary files more easily than text

Absolute and Relative Paths

- A file or directory's path specifies its location on disk
- An absolute path （绝对路径）contains all directories, starting with the root directory that lead to a specific file or directory
- A relative path （相对路径） is "relative" to another directory
  - For example, a path relative to the directory in which the application began executing

File Class

- Acts like a wrapper class for file names

- File has some very useful methods
  - exists: tests if a file already exists
  - canRead: tests if the OS will let you read a file
  - canWrite: tests if the OS will let you write to a file
  - delete: deletes the file, returns true if successful
  - length: returns the number of bytes in the file
  - getName: returns file name, excluding the preceding path
  - getPath: returns the path name—the full name

Create a File

```
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Write a File

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Read a File

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Get File Information

```
import java.io.File;  // Import the File class

public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

Delete a File

```
import java.io.File;  // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

Assignment 3

- Submit on OJ
- 11.2.2021 – 11.14.2021