Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Strings

CS2011: Introduction to Programming I

The `String` Type

The `String` Type

- Recall: `char` represents only one character.

- A ***string*** is a sequence of characters.
  - Used to store text information.

- `String` is a predefined class in Java just like `System` or `Scanner`.

- String is a ***reference type*** NOT a ***primitive type***.
  - Any Java class is a reference type.
  - Variables declared by a reference type are called reference variables.
  - You will learn more about reference types in CS-2012.

- Example:
  - `String message = "Welcome to Java!";`
  - Here, message is a ***reference variable*** because String is a ***class / reference type***.

The `String` Type and Objects

- Strings are ***reference / object types*** in Java.
  - objects are the basis for Object Oriented Programming.
  - objects can have methods associated with them which are used process the object in different ways.

- Methods that are part of objects have two general types:
  - ***instance methods****:*
    - methods that can only be accessed from a specific object instance.
    - Syntax: `referenceVariableName.methodName(arguments)`
    - Example:
      - `String s = “This is a string.”`
      - `System.out.println(s.length());` (Here the reference variable s is the instance and .length() is the instance method.
  - ***static (non-instance) methods***:
    - methods that are invoked without using a specific object instance.
    - Syntax: `ClassName.methodName(arguments)`
    - Example: Math.pow()

- You can tell which methods are static and which ones are instance by looking online at the Java API.

String Literals

- A ***String literal*** is a String value typed directly in your source code.

- String literals are always enclosed in double quotes.

- Example:
  - `String s = “Hello World!”;`
  - Here, `“Hello World!”` is the String literal.

- ***instance methods*** can also be invoked directly on a literal.
  - Example: `“Hello World!”.length();`

Common String Methods

.length()

- `.length()`
  - returns the number of characters in the String as an `int`.
  - this includes whitespace characters.
  - is an instance method.

- Example:

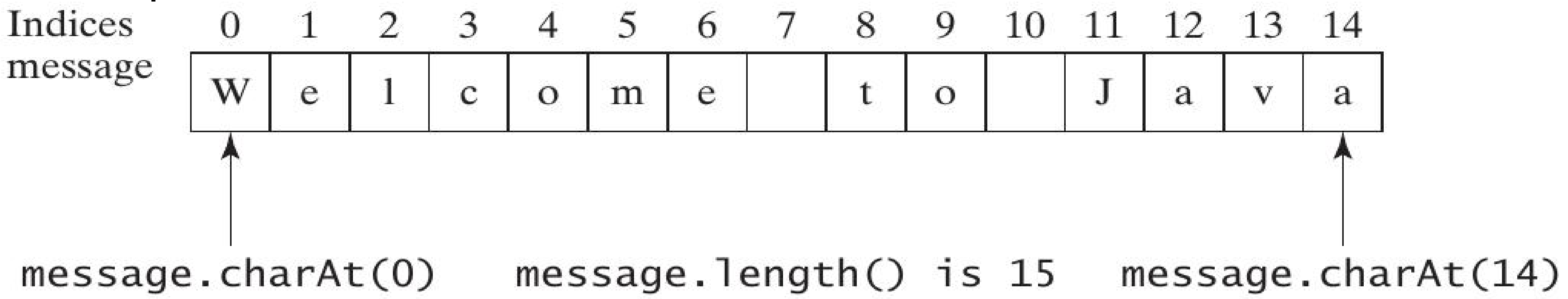
```
String message = “The thing about Strings is...”;
int numChars = message.length();
System.out.println(numChars); //prints 29
```

.charAt()

- `.charAt(int index)`
  - returns the specific character (as a `char` type) in the String at the given index.
  - is an instance method.

- ***index*** of a String
  - each character in a String is numbered starting from 0.
  - the ***index*** refers to a specific positing in the String.
  - index will always be a value between `0` and `.length() - 1`, inclusive.
  - trying to access an index that is out of bounds (negative or greater than .length() - 1), will cause a `StringIndexOutOfBoundsException`.

- Example:



Concatenating (Combining) Strings

- `.concat()`
  - Concatenates (combines) two strings.
  - Examples `s3 = s1.concat(s2);`

- +
  - an alternative is to use the + operator on Strings.
  - Example: `s3 = s1 + s2;`

- +=
  - Can also be used with Strings.

- Any value regardless of its data type that is combined with a String, will first be converted to a String and then merged with the String.

Converting String Case

- `toLowerCase()`:
  - Returns a new String with all lowercase letters.
  - Example: `s1.toLowerCase();`

- `toUpperCase()`:
  - Returns a new String with all uppercase letters.
  - Example: `s1.toUpperCase();`

- `trim()`:
  - Returns a new string with leading and trailing white space characters removed.
  - Example: `"\t Good Night \n".trim()` would return `Good Night`

Reading Strings from the Console

Reading String Input

- `.next()`:
  - reads from the console a String up to the next white-space character.

```java
Scanner input = new Scanner(System.in);
System.out.print("Enter three words separated by spaces: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

```
Enter three words separated by spaces: Welcome to Java ↵ Enter
s1 is Welcome
s2 is to
s3 is Java
```

Reading String Input

- `nextLine()`:
  - reads a String up to the next Enter key press.
  - **NOTE:** using `nextLine()` after `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, or `next()`, will cause the program to skip the next request for input.
  - Watch in class as to how this should be fixed.

- Example:

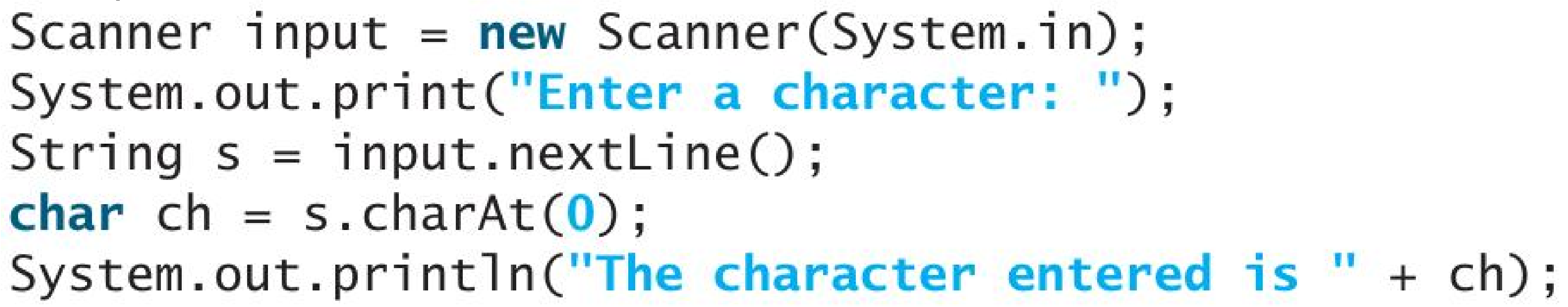
```java
Scanner input = new Scanner(System.in);
System.out.println("Enter a line: ");
String s = input.nextLine();
System.out.println("The line entered is " + s);
```

```
Enter a line: Welcome to Java
The line entered is Welcome to Java
```

Reading a Single Character Input

- You can use the `next()` or `nextLine()` methods to read a string from the console, and then invoke the charAt(0) to return the first character in the string.

- Example:



```java
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

- You can also chain together `next()` or `nextLine()` with `chartAt(0)` all in one statement.
  - This is called method chaining and you will see this technique used a lot in CS-202.

- Example:

```java
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
char ch = input.next().charAt(0);
```

Substrings

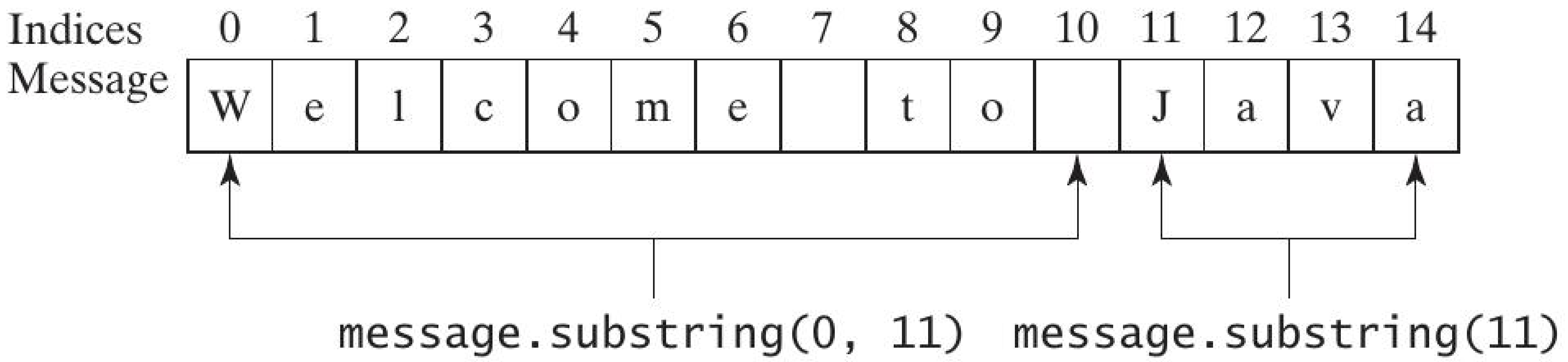
Substring

- the `substring()` method returns a substring from a given string and a given `beginIndex` (and optionally an `endIndex`)
  - NOTE: if `beginIndex > endIndex`, this is a runtime error.

```
String message = "Welcome to Java";
String message = message.substring(0, 11) + "HTML";
```

The string `message` now becomes `Welcome to HTML`.

Method	*Description*
`substring(beginIndex)`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 4.2. Note that the character at `endIndex` is not part of the substring.



Finding Specific Characters

Method	*Description*
`index(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

Converting Strings to Numbers

Converting Between Strings and Numbers

- `Integer.parseInt(string)`:
  - Converts a string to an integer, assuming that the result will be a valid integer.

- `Double.parseDouble(string)`:
  - Converts a string to a floating-point value, assuming that the result will be a valid floating-point value.

- NOTE: Strings can be converted to other numeric data types. The commands follow the same pattern of `Datatype.parseDatatype(string)`.

- You can also convert a number to a string very easily.
  - Example: `String s = "" + number;`

Other String Details

Strings are Immutable

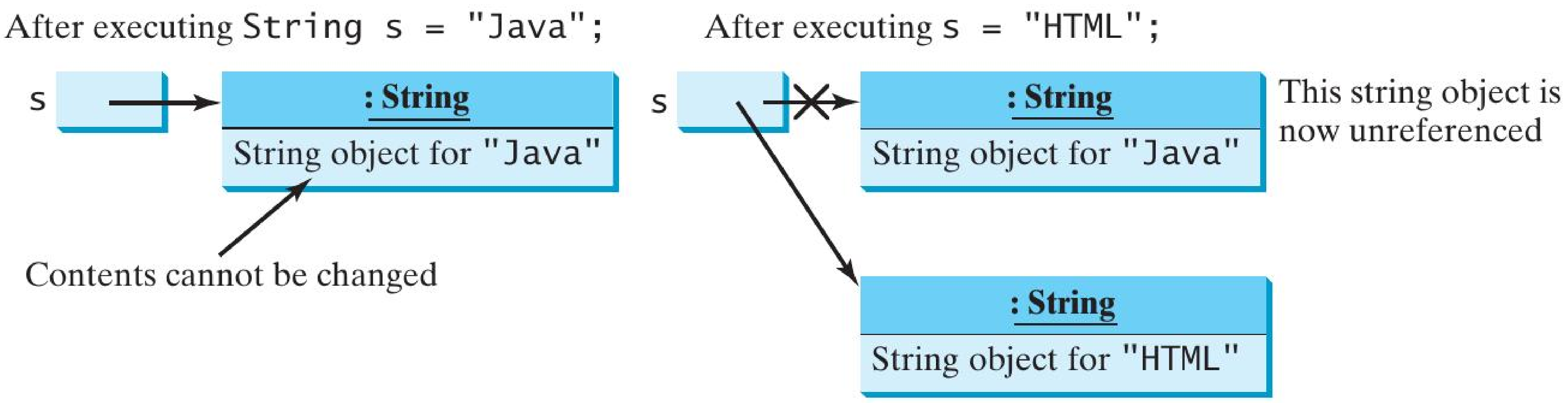
- A `String` object is immutable; its contents cannot be changed.
-
- Example: Does the following change the contents of the ***original*** string?

```
String s = "Java";
s = "HTML";
```

- Answer: No!
  - The first line creates a String object with content "Java" and assigns its reference to s.
  - The second line creates a new String object with the content "HTML" and assigns its reference to s.

Strings are Immutable

- What happens to the first string object?
  - it still exists, but can no longer be accessed. (will be Garbage Collected)


After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

Strings are immutable; once created, their contents cannot be changed.

Interned Strings

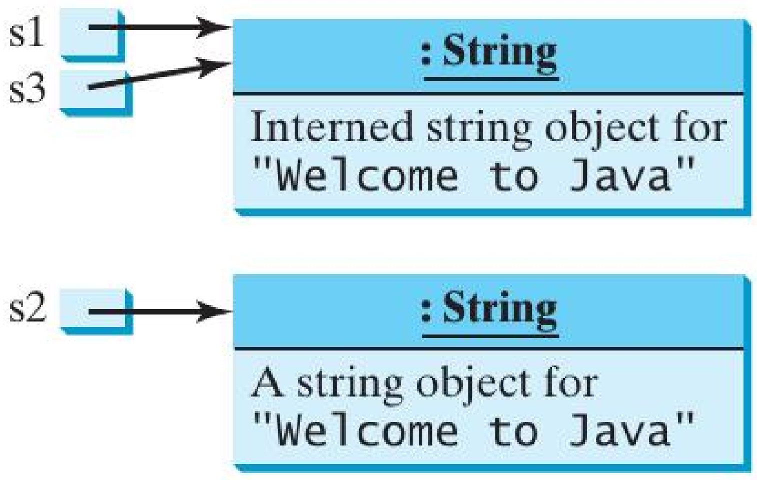
- In order to improve memory efficiency the JVM keeps a "pool" of the most recently used String literals.
- If two String literals are the exact same string, they both refer to the one instance in the "pool" instead of being separate instances in memory.

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

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


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

display

```
s1 == s2 is false
s1 == s3 is true
```

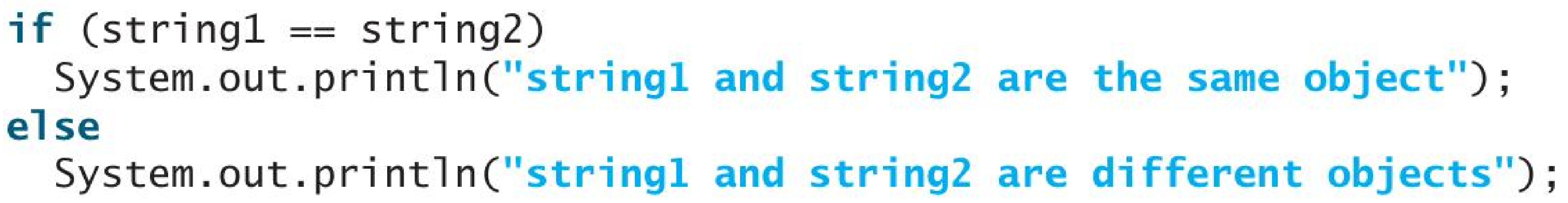
Comparing Strings

Comparing Strings

Method	*Description*
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.
`contains(s1)`	Returns true if s1 is a substring in this string.

Comparing Strings

- Strings CANNOT be compared using any of the relational operators (==, !=, <, <=, >, >=)



```
if (string1 == string2)
   System.out.println("string1 and string2 are the same object");
else
   System.out.println("string1 and string2 are different objects");
```

- == checks to see if both string objects are stored in the same memory location, NOT if they two string objects are equal.

- NEVER NEVER NEVER use == to compare Strings.

Comparing Strings

- instead use the `equals()` method of the String class.

```
if (string1.equals(string2))
  System.out.println("string1 and string2 have the same contents");
else
  System.out.println("string1 and string2 are not equal");
```

```
String s1 = "Welcome to Java";
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

Comparing Strings

- Also, never use <, <=, >, >=, or != to compare two strings.
- You should use the `compareTo()` method instead.
  - `s1.compareTo(s2);`
    - returns 0 if the two strings are equal
    - returns a value less than 0 if s1 is lexicographically (Unicode ordering) less than s2
    - returns a value greater than 0 if s1 is lexicographically greater than s2.
- the actual value returned depends on the difference between the unicode values of the different characters.
- Example:

```
String s1 = "abc";
String s2 = "abg";
s1.compareTo(s2) returns -4
```

- first a and a are compared, they are the same so move to the next character.
- compare b and b, they are the same so move to the next character
- compare c and g, since c and g are different, return c - g .

Comparing Strings

- Other versions of equals() and compareTo() exist to ignore the case of a string
- Recall that "A" is considered a different character than "a"
- `equalsIgnoreCase()` and `compareToIgnoreCase()` perform the same way as `equals()` and `compareTo()` but ignore the case of all letters in the string (compares the two strings as if they are all lowercase)

Comparing Strings

- There are also three other methods to check the prefix, suffix, and whether or not a string contains a specified substring.
  - `str.startsWith(prefix)` checks whether string str starts with the specified prefix
  - `str.endsWith(suffix)` checks whether string str ends with the specified suffix
  - `str.contains(s1)` checks whether string str contains the string s1.

Formatting Console Output

Formatting Console Output

- You already know two methods from the System class to display console output
  - `System.out.print()`
  - `System.out.println()`

- There is a third method available which allows you to format how your output will look.
  - `System.out.printf()`

Formatting Console Output

- `System.out.printf()` can be used for the following:
  - display floating point values to a certain number of decimal places
  - align your output to columns by adjusting the width of each item in the column

- Syntax:
  - `System.out.printf(format, item1, item2, …, itemk)`
    - format is a format specifier which indicates HOW an item should be displayed
    - an item can be a numeric value, character, Boolean value, or a string.

Format Specifiers

- A format specifier consists of a percent sign followed by the conversion code.
- NOTE: The percent sign DOES NOT mean percentage or modulus when used as a format specifier.

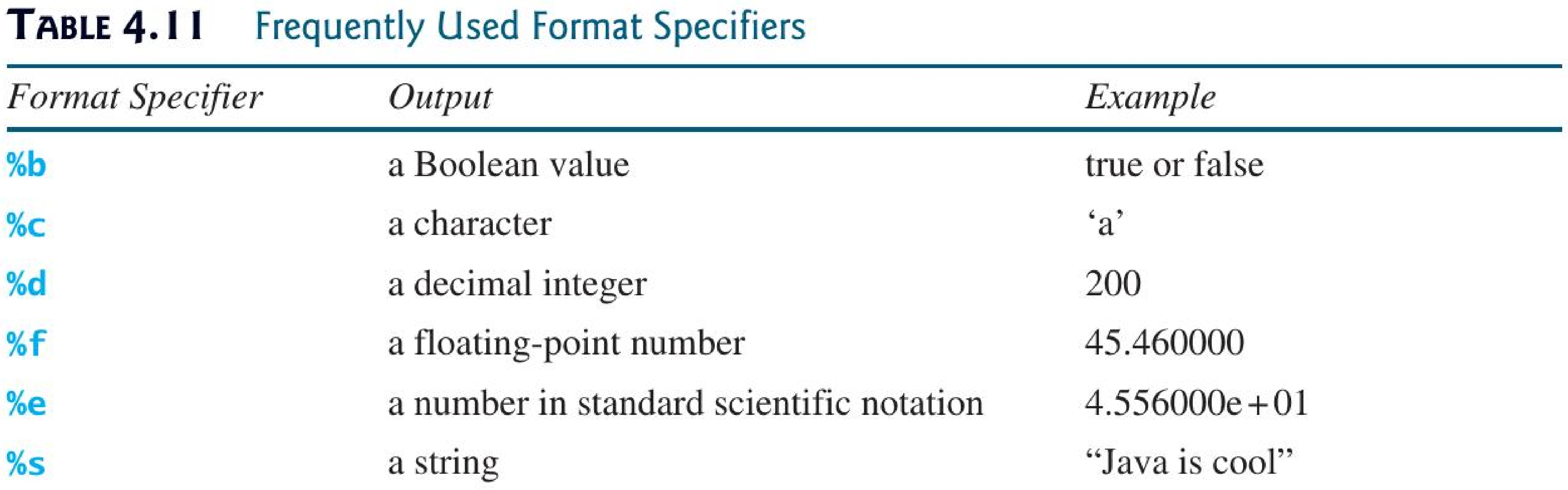
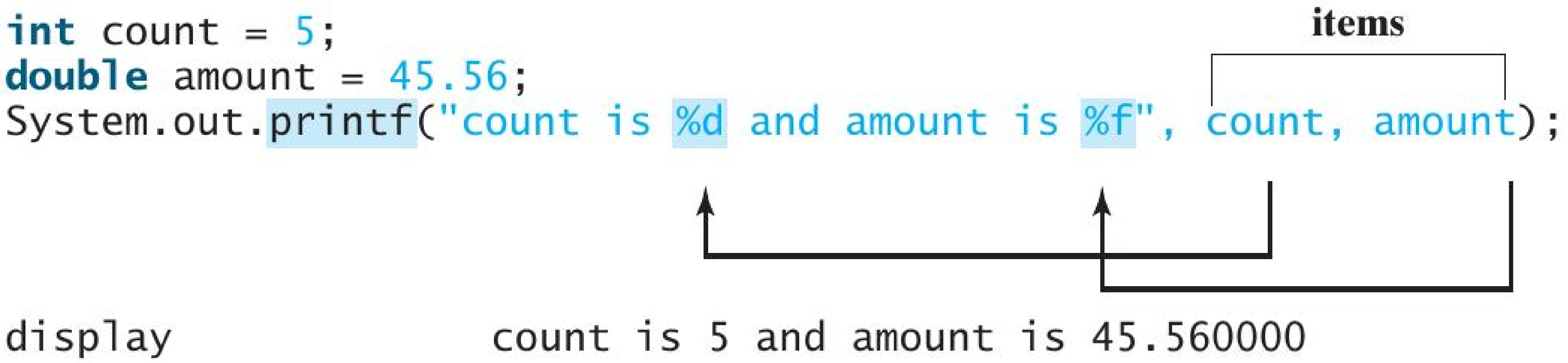


TABLE 4.11 Frequently Used Format Specifiers

Format Specifier	*Output*	*Example*
%b	a Boolean value	true or false
%c	a character	‘a’
%d	a decimal integer	200
%f	a floating-point number	45.460000
%e	a number in standard scientific notation	4.556000e+01
%s	a string	“Java is cool”

Format Specifiers

- Example: Uses a format specifier to insert an integer and a floating-point number into an output string. Notice how the format specifiers are replaced with the values stored in each item.
- NOTE: The items MUST match the format specifiers in order, in number and in exact type. By default a floating-point value is displayed with six digits after the decimal point.



```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

```
count is 5 and amount is 45.560000
```

Specifying Width and Precision

- You can also specify the width and precision with a format specifier.

- If you specify a width and precision, then your format specifier has the following syntax:
  - `%<width>.<precision><conversion_code>`
    - replace `<width>` with an integer literal to specify the width of the value when printed
    - replace `<precision>` with an integer value to specify how many decimal places to display
    - replace `<conversion_code>` with the code letter corresponding to the data type of the item you are formatting.

Width and Precision Examples

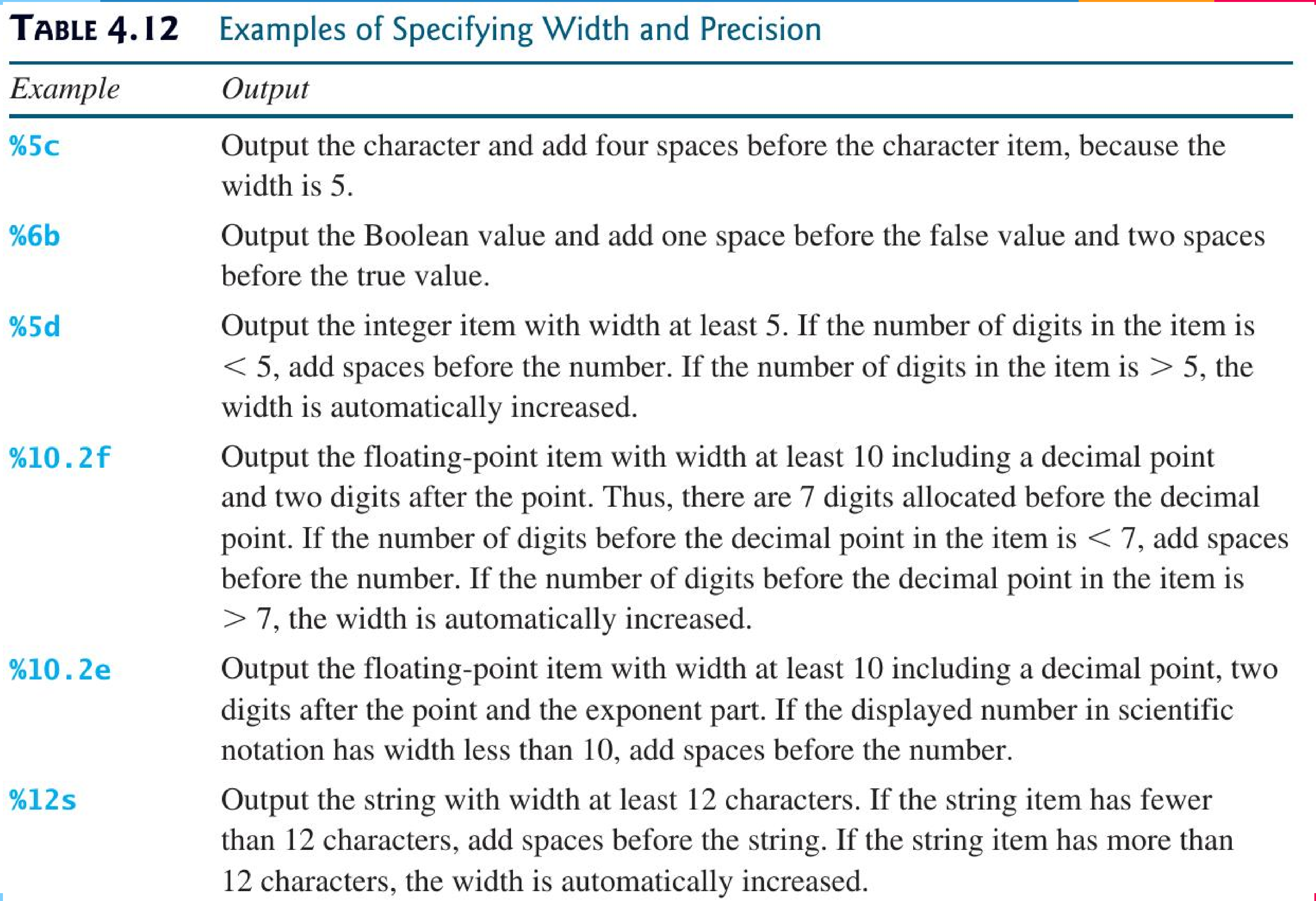


TABLE 4.12 Examples of Specifying Width and Precision

Example	*Output*
`%5c`	Output the character and add four spaces before the character item, because the width is 5.
`%6b`	Output the Boolean value and add one space before the false value and two spaces before the true value.
`%5d`	Output the integer item with width at least 5. If the number of digits in the item is < 5, add spaces before the number. If the number of digits in the item is > 5, the width is automatically increased.
`%10.2f`	Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus, there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is < 7, add spaces before the number. If the number of digits before the decimal point in the item is > 7, the width is automatically increased.
`%10.2e`	Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.
`%12s`	Output the string with width at least 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

Specifying Width and Precision

- If you specify a width smaller than the actual width of the item, the width is automatically increased.

```
System.out.printf("%3d#%2s#%4.2f\n", 1234, "Java", 51.6653);
```

displays

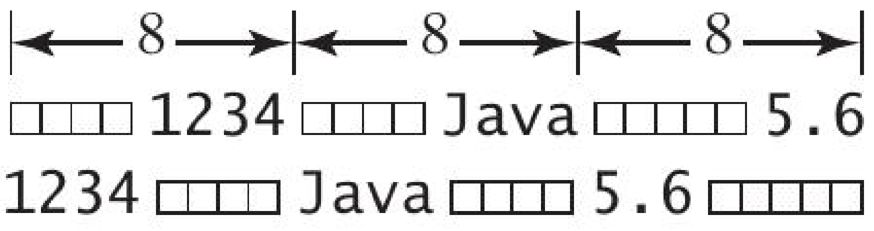
```
1234#Java#51.67
```

Specifying Width and Precision

- If you specify a width larger than the actual width of the item, the item will be padded with spaces.
- The default alignment when padding extra spaces is right justified. If you wish to left justify the item, then you must put a negative sign (-) in front of the width value.

```
System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.63);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.63);
```

display



```
□□□□1234□□□□Java□□□□□5.6
1234□□□□Java□□□□5.6□□□□□
```

where the square box (□) denotes a blank space.

Caution

- The items must match the format specifiers in exact type.
- Example: **`%f`** or **`%e`** must be matched by a floating-point value such as 40.0 NOT 40. An int variable cannot match %f or %e
- If you wish to display a literal percent sign when using printf(), you must use %%

String.format()

- The static `.format()` method can be used to format a string
- Syntax: `String.format(format, item1, item2,..., itemk)`
- Similar to the `printf()` method except `format()` returns a formatted string, `printf()` displays the formatted string.
- Example:

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);

//displays □□45.56□□□□14AB□□                    □ = space
```

References

- Liang, Chapter 04: Math Functions, char Datatype and Strings.