

Chapter 4

Mathematical Functions, Characters and Strings

Programming I --- Ch. 4

1

Objectives

- To solve mathematical problems by using the methods in the **Math** class
- To represent characters using the **char** type
- To encode characters using ASCII and Unicode
- To represent special characters using the escape sequences
- To compare and test characters using the static methods in the **Character** class
- To represent strings using the **String** object, and using some of the common String methods for obtaining string length, for accessing characters in the string, for concatenating strings, for converting a string to upper or lowercases, for trimming a string, for comparing strings and to obtain substrings.
- To read a character and strings from the console
- To format output using the **System.out.printf** method

Programming I --- Ch. 4

2

Common Mathematical Functions

- *Java provides many useful methods in the **Math** class for performing common mathematical functions.*
- A **method** is a group of statements that performs a specific task.
 - You have already used the **pow(a, b) method** to compute a^b in Section 2.9.4,
 - the **random()** method for generating a random number in Section 3.7.
- This section introduces other useful methods in the **Math** class, including the rounding, min, max, absolute methods.
- Since Math is in the java.lang package, it does not need to be imported. java.lang is the "default package" and everything in it is already implicitly imported for you.

Programming I --- Ch. 4

3

Math Class - Exponent Methods

- **pow(double a, double b)**
Returns a raised to the power of b, as a double value.
- **sqrt(double a)**
 - Returns the square root of a.
 - Note that the result is the double value closest to the true mathematical square root of the argument value.

Examples:

```
Math.pow(2, 3) returns 8.0
Math.pow(3, 2) returns 9.0
Math.pow(3.5, 2.5) returns 22.91765
Math.sqrt(4) returns 2.0
Math.sqrt(10.5) returns 3.24
```

Programming I --- Ch. 4

4

Math Class - Rounding Methods

- **ceil(double x)**
Returns the smallest integer value that is greater than or equal to the argument. This integer value is returned as a double.
- **floor(double x)**
Returns the largest integer value that is less than or equal to the argument. This integer value is returned as a double.
- **round(double x)**
Returns the value of the argument rounded to the nearest **int** value.

Examples:

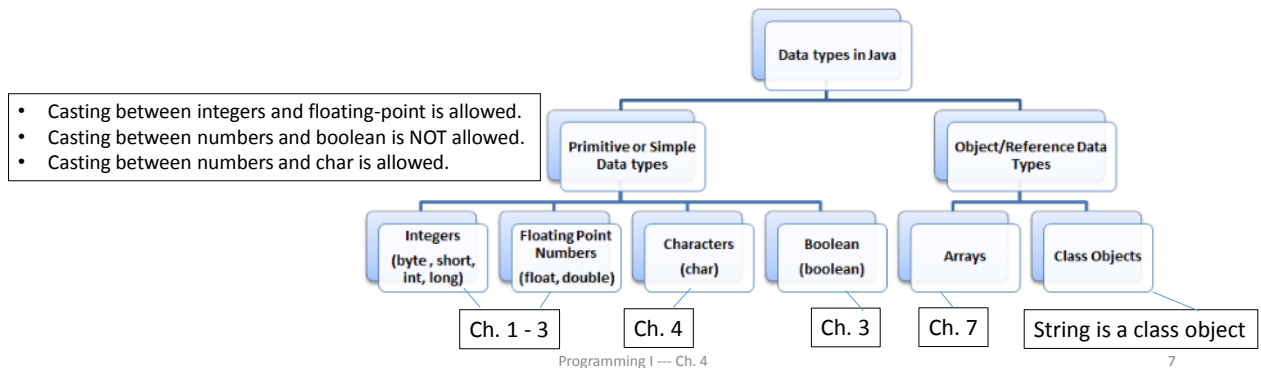
```
Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.7) returns 2.0
Math.floor(2.0) returns 2.0
Math.round(2.1) returns 2
Math.round(2.7) returns 3
```

Math Class - min, max, abs Methods

- The **min** and **max** methods return the minimum and maximum numbers of two numbers (**int**, **long**, **float**, or **double**).
- The **abs** method returns the absolute value of the number (**int**, **long**, **float**, or **double**).
- For example,
 - **Math.max(2, 3)** returns **3**
 - **Math.max(4.4, 5.0)** returns **5.0**
 - **Math.min(2.5, 4.6)** returns **2.5**
 - **Math.abs(-2)** returns **2**
 - **Math.abs(-2.1)** returns **2.1**

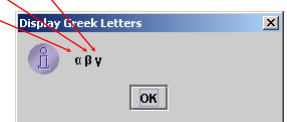
Data Types in Java

- Remember that in Chapter 2, we have mentioned there are 8 primitive data types in Java:
byte, short, int, long, float, double, boolean and char.



Unicode: Four hexadecimal digits

- A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *encoding*. There are different ways to encode a character.
- Java characters** use *Unicode*, a *16-bit encoding scheme* established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. E.g. character A is represented as 00000000 01000001 (which has a decimal value of 65)
- However, it turned out that the 65,536 characters possible in a 16-bit encoding are not enough to represent all the characters in the world.
- The Unicode standard therefore has been extended to allow up to 1,112,064 characters. Those characters that go beyond the original 16-bit limit are called *supplementary characters*.
- For simplicity, the textbook considers only the original 16-bit Unicode characters. Unicode \u03b1 \u03b2 \u03b3 for 3 Greek letters
- A 16-bit Unicode takes two bytes, preceded by **\u**, expressed in four hexadecimal digits that run from **\u0000 to \uFFFF** (value of 65535 in decimal). E.g. character A is represented as **\u0041** (which has a decimal value of 65)



ASCII Character Set – Hexadecimal index

- *ASCII (American Standard Code for Information Interchange)* is an 8-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters.
- ASCII Character Set is a subset of the Unicode from \u0000 to \u007f (i.e. $16 \times 7 + 15 = 127$)

TABLE B.2 ASCII Character Set in the Hexadecimal Index

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	del

16 columns * 8 rows = 128

Programming I --- Ch. 4

9

ASCII Character Set – Decimal index

TABLE B.1 ASCII Character Set in the Decimal Index

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

10 columns * 12 rows + 8 = 128

- Table 4.4 shows the [ASCII code](#) for some commonly used characters.

TABLE 4.4 ASCII Code for Commonly Used Characters		
Characters	Code Value in Decimal	Unicode Value
'0' to '9'	48 to 57	\u0030 to \u0039
'A' to 'Z'	65 to 90	\u0041 to \u005A
'a' to 'z'	97 to 122	\u0061 to \u007A

Programming I --- Ch. 4

- Hexadecimal value of \u0030 is equivalent to decimal value of 48: $16 * 3 + 0 = 48$
- Hexadecimal value of \u0041 is equivalent to decimal value of 65: $16 * 4 + 1 = 65$
- Hexadecimal value of \u0061 is equivalent to decimal value of 97: $16 * 6 + 1 = 97$

10

ASCII - Binary Character Table

For Reference

Letter	ASCII Code	Binary	Letter	ASCII Code	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
d	100	01100100	D	068	01000100
e	101	01100101	E	069	01000101
f	102	01100110	F	070	01000110
g	103	01100111	G	071	01000111
h	104	01101000	H	072	01001000
i	105	01101001	I	073	01001001
j	106	01101010	J	074	01001010
k	107	01101011	K	075	01001011
l	108	01101100	L	076	01001100
m	109	01101101	M	077	01001101
n	110	01101110	N	078	01001110
o	111	01101111	O	079	01001111
p	112	01110000	P	080	01010000
q	113	01110001	Q	081	01010001
r	114	01110010	R	082	01010010
s	115	01110011	S	083	01010011
t	116	01110100	T	084	01010100
u	117	01110101	U	085	01010101
v	118	01110110	V	086	01010110
w	119	01110111	W	087	01010111
x	120	01111000	X	088	01011000
y	121	01111001	Y	089	01011001
z	122	01111010	Z	090	01011010

11

Character Data Type

- The character data type, **char**, is used to represent a single character.
- A character literal is enclosed in single quotation marks.

```
char letter = 'A';
```

```
char numChar = '4';
```

```
char letter = '\u0041'; // (Unicode: Character A's Unicode is 0041, Table 4.4)
```

```
char numChar = '\u0034'; // (Unicode: Four hexadecimal digits)
```

- You can use ASCII characters such as '**X**', '**1**', and '**\$**' in a Java program as well as Unicode.

Casting from Numeric Types to **char**

- Recall that integer is 32 bits whereas char is 16 bits.
- Converting integer to char requires [explicit casting](#).
- When a floating-point value is cast into a **char**, the floating-point value is first cast into an **int**, which is then cast into a **char**.

```
char ch = (char)65.25; // Decimal 65 is assigned to ch
System.out.println(ch); // ch is character A
```

Programming I --- Ch. 4

13

Casting from **char** to Numeric Types

- A **char** can be cast into a numeric type, with or without explicit casting.:

```
int i = (int)'A';
System.out.println(i); // i is 65
```
- [Implicit](#) casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of 'a' is **97**, which is within the range of a byte, these implicit castings are fine:

```
byte b = 'a'; // b is 97
int i = 'a'; // i is 97
byte x = '\u007f'; // 16x7+15 = 127
```
- But the following casting is incorrect, because the Unicode **\u0080 (decimal value of 128)** cannot fit into a byte:

```
byte y = '\u0080'; // invalid
```
- To force this assignment, use [explicit](#) casting, as follows:

```
byte y = (byte)'\ u0080'; // overflow: y is -128
byte z = (byte)'\ u0081'; // overflow: z is -127
```

Programming I --- Ch. 4

14

Adding characters

- The increment and decrement operators can be used on char variables to get the next or preceding Unicode character. For example, the following statements display character b.

```
char ch = 'a';
System.out.println(++ch); // output 'b' after adding the int to the Unicode of the char
```
- You can also use `ch+=1` which can be considered as `ch=(char)(ch+1)`; As stated on slide 55 of Chapter 2, in Java, an augmented expression of the form `x1 op= x2` is implemented as `x1 =(T)(x1 op x2)`, where `T` is the type for `x1`. Hence, for `ch+=1`, casting is already handled automatically.
- But NOT `ch=ch+1` (syntax error complaining that cannot convert from int to char) unless you carry out casting, i.e. `ch=(char) (ch+1)`;
- However, `ch='a'+1` is a valid statement!!!!!! **Be Aware!!!!**

Implicit *narrowing primitive conversion*

- According to the binary promotion rules, if neither of the operands is double, float or long, both are promoted to int.
- Consider the following examples:

```
char x = 'a' + 'b'; // OK, char: Ã, ASCII value of 195
char y = 'a';
char z = y + 'b'; // Compilation error
```
- When a constant expression appears in an assignment context, the Java compiler computes the value of the expression and sees if it is in the range of the type that you are assigning to. If it is, then an implicit narrowing primitive conversion is applied.
- In the first example, `'a' + 'b'` is a constant expression, and its value will fit in a char, so the compiler allows the implicit narrowing of the int expression result to a char.
- In the second example, `y` is a variable so `y + 'b'` is NOT a constant expression. So even though the value will fit, the compiler does NOT allow any implicit narrowing, and you get a compilation error saying that an int cannot be assigned to a char.

Subtracting characters

- When you subtract 2 characters, it will return you the offset as an integer.

```
char answer = 'f' - 'A'; // 102-65=37, which is implicitly cast to % symbol because of char data type
System.out.println(("f-'A'") + " + " + answer); // 37 is cast to a string because of "", it displays 37%
System.out.println(("f-'A'") + answer); // displays 74 which is 37 + 37
```

Escape Sequences for Special Characters

- The following statement has a compile error. The compiler thinks the second quotation character is the end of the string and does not know what to do with the rest of characters.

```
System.out.println("He said "Java is fun");
```
- To overcome this problem, Java uses a special notation to represent special characters. This special notation, called an *escape sequence*, consists of a *backslash (\)* followed by a character or a combination of digits.
- So, now you can print the quoted message using the following statement: `System.out.println("He said \"Java is fun\");`

Escape Sequences

- The backslash `\` is called an *escape character*. It is a special character.
- To display this character, you have to use an escape sequence `\\`.
- For example, the following code

```
System.out.println("\t is a tab character");
System.out.println("\\t is a tab character");
```

displays

is a tab character

`\t` is a tab character

TABLE 4.5 Escape Sequences

Escape Sequence	Name	Unicode Code
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	Linefeed	<code>\u000A</code>
<code>\f</code>	Formfeed	<code>\u000C</code>
<code>\r</code>	Carriage Return	<code>\u000D</code>
<code>\\</code>	Backslash	<code>\u005C</code>
<code>\"</code>	Double Quote	<code>\u0022</code>

Programming I --- Ch. 4

19

Comparing and Testing Characters

- Two characters can be compared using the relational operators just like comparing two numbers. This is done by comparing the Unicode values of the two characters. For example,
 - `'a' < 'b'` is true because the Unicode value for `'a'` in decimal (**97**) is less than that for `'b'` (**98**).
 - `'a' < 'A'` is false because the Unicode value for `'a'` in decimal (**97**) is greater than that for `'A'` (**65**).
 - `'1' < '8'` is true because the Unicode value for `'1'` in decimal (**49**) is less than that for `'8'` (**56**).
- For convenience, Java provides the following methods in the **Character** class for testing characters as shown in Table 4.6.

TABLE 4.6 Methods in the Character Class

Method	Description
<code>isDigit(ch)</code>	Returns true if the specified character is a digit.
<code>isLetter(ch)</code>	Returns true if the specified character is a letter.
<code>isLetterOrDigit(ch)</code>	Returns true if the specified character is a letter or digit.
<code>isLowerCase(ch)</code>	Returns true if the specified character is a lowercase letter.
<code>isUpperCase(ch)</code>	Returns true if the specified character is an uppercase letter.
<code>toLowerCase(ch)</code>	Returns the lowercase of the specified character.
<code>toUpperCase(ch)</code>	Returns the uppercase of the specified character.

For example,
`System.out.println("isDigit('a') is " + Character.isDigit('a'));`
`System.out.println("isLetter('a') is " + Character.isLetter('a'));`

displays
 isDigit('a') is false
 isLetter('a') is true

Programming I --- Ch. 4

20

Comparing and Testing Characters

```

if (ch >= 'A' && ch <= 'Z')
    System.out.println(ch + " is an uppercase letter");
else if (ch >= 'a' && ch <= 'z')
    System.out.println(ch + " is a lowercase letter");
else if (ch >= '0' && ch <= '9')
    System.out.println(ch + " is a numeric character");

```

Programming I --- Ch. 4

21

The String Type

- The **char** type represents only one character. To represent a sequence of characters, use the data type called **String**.
- A string literal must be enclosed in quotation marks (" "). A character literal is a single character enclosed in single quotation marks (' '). Therefore, "A" is a string, but 'A' is a character.
- For example, the following code declares **message** to be a string with the value "Welcome to Java".

```
String message = "Welcome to Java";
```
- The **String** type is **NOT** a **primitive type**. It is known as a *reference type*. Reference data types will be discussed in detail in Chapter 9, Objects and Classes.
- Table 4.7 lists the **String** methods for obtaining string length, for accessing characters in the string, for concatenating strings, for converting a string to upper or lowercases, and for trimming a string.

TABLE 4.7 Simple Methods for *String* Objects

Method	Description
length()	Returns the number of characters in this string.
charAt(index)	Returns the character at the specified index from this string.
concat(s1)	Returns a new string that concatenates this string with string s1.
toUpperCase()	Returns a new string with all letters in uppercase.
toLowerCase()	Returns a new string with all letters in lowercase.
trim()	Returns a new string with whitespace characters trimmed on both sides.

"Welcome".toLowerCase() returns a new string welcome.
 "Welcome".toUpperCase() returns a new string WELCOME.

22

instance method vs static method

- Strings are objects in Java. The methods in Table 4.7 can only be invoked from a specific string instance, and these methods are called *instance methods*.
- The syntax to invoke an *instance method* is **reference-variable.methodName(arguments)**.

```
String message = "Welcome";
System.out.println(message.length());
```
- A non-instance method is called a *static method*. A static method can be invoked without using an object. All the methods defined in the **Math** class are static methods.
- Recall that the syntax to invoke a *static method* is **ClassName.methodName(arguments)**. For example, the **pow** method in the **Math** class can be invoked using **Math.pow(2, 2.5)**
- **Are the methods in the Character class, as discussed in Table 4.6 (slide 21) static methods or instance methods?**
- **Suppose we have the following declarations. Write the two statements to convert ch and s1 to uppercase.**

```
char ch = 'a';
String s1 = "a";
```

Programming I --- Ch. 4

23

Getting String Length

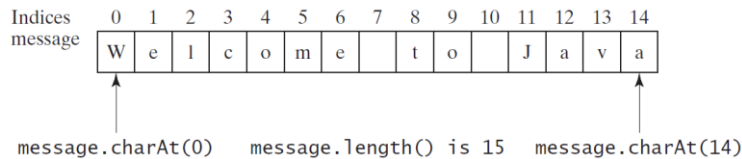
```
String message = "Welcome to Java";
System.out.println("The length of " + message + " is " +
    message.length());
```

- Java allows you to use the string literal to refer directly to strings without creating new variables.
- Thus, **"Welcome to Java".length()** is correct and returns **15**.
- Note that **""** denotes an *empty string* and **"".length()** is **0**.
- **" ".length()** is **1**

Programming I --- Ch. 4

24

Getting Characters from a String



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

```
System.out.println("The first character in message is " + message.charAt(0));
```

- Note that the index is between **0** and **message.length()-1**.
- For example, **message.charAt(0)** returns the character **W** and **message.charAt(message.length())** would cause a **StringIndexOutOfBoundsException**.
- A method may have many arguments or no arguments. For example, the **charAt(index)** method has one argument, but the **length()** method has no arguments.

Programming I --- Ch. 4

25

Concatenating Strings

- You can use the **concat** method to concatenate two strings. For example:
`String s3 = s1.concat(s2);`
- You can use the plus (+) operator to concatenate two strings, so the previous statement is equivalent to `String s3 = s1 + s2;`
- Recall that the + operator can also concatenate a number with a string. In this case, the number is converted into a string and then concatenated.
- If **i = 1** and **j = 2**, what is the output of the following statement?
`System.out.println("i + j is " + i + j);`
- The output is **"i + j is 12"** because **"i + j is "** is concatenated with the value of **i** first. To force **i + j** to be executed first, enclose **i + j** in the parentheses, as follows:
`System.out.println("i + j is " + (i + j));`
- Note that at least one of the operands must be a string in order for concatenation to take place.
- The augmented += operator can also be used for string concatenation. For example,
`message += " and Java is fun";`

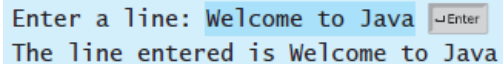
Programming I --- Ch. 4

26

Reading a String from the Console

- To read a string from the console, invoke the **next()** method or the **nextLine()** method on a **Scanner** object.
 - The **next()** method reads a string ending at a delimiter (default is whitespace character).
 - It does not read the delimiter (usually whitespace or the Enter Key) after the token.
 - You can use the **nextLine()** method to read an entire line of text. The **nextLine()** method reads a line that ends with the *Enter* key pressed.
 - The line separator is read, but it is not part of the string returned by **nextLine()**.

```
Scanner input = new Scanner(System.in);
System.out.print("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
```

Programming I --- Ch. 4

27

Reading a Character from the Console

- Scanner class in Java supports **nextInt()**, **nextLong()**, **nextDouble()** etc. But there is no **nextChar()**
- To read a char, we use **next().charAt(0)** or **nextLine().charAt(0)**.
- **next()** function returns the next token/word in the input as a string and **charAt(0)** function returns the first character in that string.
- For example, the following code reads a character from the keyboard:


```
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);
```

Programming I --- Ch. 4

28

Some notes on how Scanner class works

- `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` are called **token-based input** methods.
 - They read input separated by delimiters (whitespace characters by default or the Enter Key).
 - A token-based input first skips any delimiters, then reads a token ending at a delimiter.
 - It does not read the delimiter after the token.
- The token is automatically converted into a value of the byte, short, int, long, float, or double type for `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`, respectively.
- For the `next()` method, no conversion is performed. The token is expected to be a string.
- A runtime exception `java.util.InputMismatchException` will be thrown if the token does not match the expected type.

Programming I --- Ch. 4

29

Testing for valid input

- A Scanner has methods to see what the next token will be.

Method	Description
<code>hasNext()</code>	returns <code>true</code> if there are any more tokens of input to read (<i>always true for console input</i>)
<code>hasNextInt()</code>	returns <code>true</code> if there is a next token and it can be read as an <code>int</code>
<code>hasNextDouble()</code>	returns <code>true</code> if there is a next token and it can be read as a <code>double</code>
<code>hasNextLine()</code>	returns <code>true</code> if there are any more <u>lines</u> of input to read (<i>always true for console input</i>)

To be discussed in Chapter 5 when we read input from a file.

- These methods do not consume input. They give information about what the input is waiting.

Programming I --- Ch. 4

30

An example

- Suppose from the keyboard, you enter **34**, press the *Enter* key, then enter **567** and press the *Enter* key for the following code:

```
int intValue = input.nextInt();
String line = input.nextLine();
```

- You will get **34** in **intValue** and an empty string in **line**!!!
- The token-based input method **nextInt()** reads in **34** and stops at the delimiter, which in this case is a line separator (the *Enter* key).
- The **nextLine()** method ends after reading the line separator and returns the string read before the line separator.
- Since there are no characters before the line separator, **line** is empty.
- If the **nextLine()** method is invoked after a token-based input method, this method reads characters that start from this delimiter and end with the line separator.
- For this reason, *avoid using a line-based input after a token-based input.*

Programming I --- Ch. 4

31

An example: possible solutions

- One possible solution is to read the complete line as string and convert it to an integer:

```
int intValue = Integer.parseInt(input.nextLine());
String line = input.nextLine();
```

- Another possible solution is to use token-based input for both inputs, but this will NOT work if the string input consists of multiple words:

```
int intValue = input.nextInt();
String line = input.next();
```

Programming I --- Ch. 4

32

Conversion between Strings and Numbers

- You can convert a numeric string into a number. To convert a string into an **int** value, use the **Integer.parseInt** method, as follows:

```
int intValue = Integer.parseInt(intString);
```

 where **intString** is a numeric string such as **"123"**.
- To convert a string into a **double** value, use the **Double.parseDouble** method, as follows:

```
double doubleValue = Double.parseDouble(doubleString);
```

 where **doubleString** is a numeric string such as **"123.45"**.
- If the string is not a numeric string, the conversion would cause a runtime error.
- The **Integer** and **Double** classes are both included in the **java.lang** package, and thus they are automatically imported.
- You can convert a number into a string, simply use the string concatenating operator as follows: **String s = number + "";**

Programming I --- Ch. 4

33

Comparing Strings

- The **String** class contains the methods as shown in Table 4.8 for comparing two strings.

TABLE 4.8 Comparison Methods for String Objects

Method	Description
<code>equals(s1)</code>	Returns true if this string is equal to string <code>s1</code> .
<code>equalsIgnoreCase(s1)</code>	Returns true if this string is equal to string <code>s1</code> ; it is case insensitive.
<code>compareTo(s1)</code>	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 <code>s1</code> .
<code>compareToIgnoreCase(s1)</code>	Same as <code>compareTo</code> except that the comparison is case insensitive.
<code>startsWith(prefix)</code>	Returns true if this string starts with the specified prefix.
<code>endsWith(suffix)</code>	Returns true if this string ends with the specified suffix.
<code>contains(s1)</code>	Returns true if <code>s1</code> is a substring in this string.

"Welcome to Java".startsWith("we") returns false.
 "Welcome to Java".endsWith("va") returns true.
 "Welcome to Java".contains("to") returns true.

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

The actual value returned from the **compareTo** method depends on the offset of the first two distinct characters in **s1** and **s2** from left to right.

- The method returns the value **0** if **s1** is equal to **s2**,

```
String s1 = "Welcome to Java";
String s2 = "Welcome to Kava";
System.out.println(s1.compareTo(s2)); // -1 (74-75)
```

Programming I --- Ch. 4

34

String class: equals method

- How do you compare the contents of two strings? You might attempt to use the `==` operator, as follows:


```
if (string1 == string2)
    System.out.println("string1 and string2 are the same object");
else
    System.out.println("string1 and string2 are different objects");
```
- However, the `==` operator checks only whether **string1** and **string2** refer to the same object; it does not tell you whether they have the same contents.
- Therefore, you should not use the `==` operator to find out whether two string variables have the same contents. Instead, you should use the **equals** method.
- The following code, for instance, can be used to compare two strings:

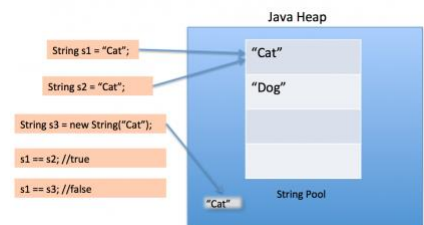

```
if (string1.equals(string2))
    System.out.println("string1 and string2 have the same contents");
else
    System.out.println("string1 and string2 are not equal");
```

Programming I --- Ch. 4

35

Notes on String Objects

- There are two ways to create a string:
 - `String s1 = "Cat";` //String literal goes into String Pool
 - `String s3 = new String("Cat");` //String object
- When we use double quotes to create a String, it first looks for String with the same value in the String pool, if found, it just returns the reference; otherwise, it creates a new String in the pool and then returns the reference.
- If now, you write `String s2 = "Cat";`
This time it checks if "Cat" literal is already available in the StringPool or not. As now it exists, so s2 will refer to the same literal referred to by s1.
- For s3, the new `String(String)` initializes a newly created String object. Using *new* operator, we force String class to create a new String object in heap space.
- So s1 and s2 will have reference to the same String in the pool whereas s3 will be a different object outside the pool, hence the output.



Programming I --- Ch. 4

36

```

1
2 public class CompareString {
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5
6         String s1 = "Welcome";
7         String s2 = "Welcome";
8
9         String s3 = new String ("Welcome");
10        String s4 = new String ("Welcome");
11
12        System.out.println(s1 == s2); // true
13        System.out.println(s3 == s4); //false
14        System.out.println(s1.equals(s2)); // true
15        System.out.println(s3.equals(s4)); // true
16
17        String e = "JDK";
18        String f = new String("JDK");
19        System.out.println(e == f); // False
20    }
21 }

```

Programming I --- Ch. 4

37

Difference between String literal and String object in Java

- We use == operator for reference comparison (**address comparison**) and .equals() method for **content comparison**.
- That is, == checks if both objects point to the same memory location whereas .equals() compares the values.

Programming I --- Ch. 4

38

Comparing Strings: an example

- Listing 4.2 gives a program that prompts the user to enter two cities and displays them in alphabetical order.
- If `input.nextLine()` is replaced by `input.next()` (line 9), you cannot enter a string with spaces for `city1`.
- Since a city name may contain multiple words separated by spaces, the program uses the `nextLine` method to read a string (lines 9, 11).
- Invoking `city1.compareTo(city2)` compares two strings `city1` with `city2` (line 13). A negative return value indicates that `city1` is less than `city2`.

LISTING 4.2 OrderTwoCities.java

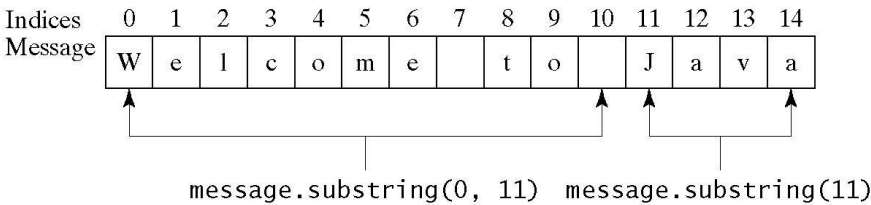
```
1 import java.util.Scanner;
2
3 public class OrderTwoCities {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         // Prompt the user to enter two cities
8         System.out.print("Enter the first city: ");
9         String city1 = input.nextLine();
10        System.out.print("Enter the second city: ");
11        String city2 = input.nextLine();
12
13        if (city1.compareTo(city2) < 0)
14            System.out.println("The cities in alphabetical order are " +
15                               city1 + " " + city2);
16        else
17            System.out.println("The cities in alphabetical order are " +
18                               city2 + " " + city1);
19    }
20 }
```

Obtaining Substrings

- You can obtain a single character from a string using the `charAt` method. You can also obtain a substring from a string using the `substring` method in the `String` class, as shown in Table 4.9.

TABLE 4.9 The `String` class contains the methods for obtaining substrings.

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



Obtaining Substrings (cont'd)

- For example,

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

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

- If **beginIndex** is **endIndex**, **substring(beginIndex, endIndex)** returns an empty string with length **0**.
- If **beginIndex** > **endIndex**, it would be a runtime error.

Programming I --- Ch. 4

41

Finding a Character or a Substring in a String

- The **String** class provides several versions of **indexOf** and **lastIndexOf** methods to find a character or a substring in a string, as shown in Table 4.10.

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

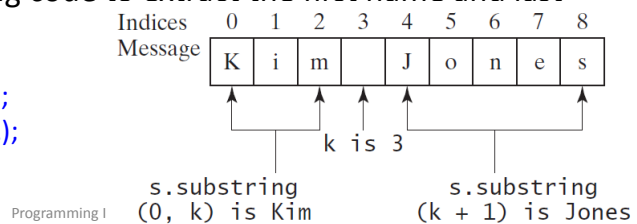
Programming I --- Ch. 4

42

Finding a Character or a Substring in a String - examples

- **For example,**
 - `"Welcome to Java".indexOf('o')` returns **4**.
 - `"Welcome to Java".indexOf('o', 5)` returns **9**.
 - `"Welcome to Java".indexOf("come")` returns **3**.
 - `"Welcome to Java".indexOf("java", 5)` returns **-1**.
 - `"Welcome to Java".lastIndexOf('o')` returns **9**.
- Suppose a string `s` contains the first name and last name separated by a space. You can use the following code to extract the first name and last name from the string:

```
int k = s.indexOf(' ');
String firstName = s.substring(0, k);
String lastName = s.substring(k + 1);
```



An empty string

- The empty string `""` is the [string that has no characters in it](#).
- It is an actual string that has a well-defined length. All of the standard string operations are well-defined on the empty string - you can convert it to lower case, look up the index of some character in it, etc. such as

```
String a = ""; // an empty string
String b = " "; // a string with a space
a.length(); // returns zero
b.length(); // returns one
```

Null Strings

- In **Java** a reference type assigned **null** has no value at all.

```
String c = null;
if (c==null)
    System.out.println("c is null");
```

- null is a literal similar to true and false.
- A **string** assigned "" has a value: an empty **string**, which is to say a **string** with no characters in it.
- When a variable is assigned **null** it means there is no underlying object of any kind, **string** or otherwise.
- "" and **null** are different.
- The null value can *never* have a method invoked upon it.



Programming I --- Ch. 4

45

Testing for an empty string

You can check for an empty String using one of the following:

- 1) Check if String.length() == 0
 - 2) Use String.isEmpty() method to return a boolean value
 - 3) Use String.equals("") method to return a boolean value
- Note that these methods are not null safe and will throw `NullPointerException` if String is null.

```
String a = "";
String c = null;
System.out.println(a.equals(c)); // false
System.out.println(c.equals(a)); // NullPointerException because c is null
```

- Be careful to check if String is null before calling length() to avoid `NullPointerException`. Below is an example:
- ```
if(string != null && string.length() == 0){ return true; }
```

Programming I --- Ch. 4

46

# Formatting Console Output

- You can use the **System.out.printf** method to display formatted output on the console.
- Often, it is desirable to display numbers in a certain format. For example, the following code computes interest, given the amount and the annual interest rate.

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is $" + interest);
```

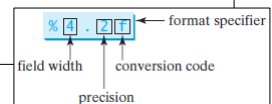
and the output is "Interest is \$16.404674"

```
System.out.println("Interest is $" + (int)(interest * 100) / 100.0);
```

and the output is "Interest is \$16.4"

```
System.out.printf("Interest is $%4.2f", interest);
```

and the output is "Interest is \$16.40"



Programming I --- Ch. 4

## Some Common Format specifier

- The syntax to invoke **System.out.printf** method is `System.out.printf(format, item1, item2, ..., itemk)` where **format** is a string that may consist of substrings and format specifiers.
- A *format specifier* specifies how an item should be displayed. An item may be a numeric value, a character, a Boolean value, or a string.
- A simple format specifier consists of a percent sign (%) followed by a conversion code. Table 4.11 lists some frequently used simple format specifiers.

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" |

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

display                      count is 5 and amount is 45.560000

items

Programming I --- Ch. 4

48



## Format specifier: specifying width and precision

- By default, a floating-point value is displayed with six digits after the decimal point.
- You can specify the width and precision in a format specifier, as shown in the examples in Table 4.12.
- If an item requires more spaces than the specified width, the width is automatically increased. For example, the following code

```
System.out.printf("%3d#%2s#%4.2f\n", 1234,
"Java", 51.6653);
displays
1234#Java#51.67
```

**TABLE 4.12** Examples of Specifying Width and Precision

| Example             | Output                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%5c</code>    | Output the character and add four spaces before the character item, because the width is 5.                                                                                                                                                                                                                                                                                                |
| <code>%6b</code>    | Output the Boolean value and add one space before the false value and two spaces before the true value.                                                                                                                                                                                                                                                                                    |
| <code>%5d</code>    | 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.                                                                                                                                                                                 |
| <code>%10.2f</code> | 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. |
| <code>%10.2e</code> | 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.                                                                                                                                                    |
| <code>%12s</code>   | 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.                                                                                                                                                                  |

## right justified vs left justified

- By default, the output is right justified. You can put the minus sign (-) in the format specifier to specify that the item is left justified in the output within the specified field.
- For example, the following statements

```
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

```

|← 8 →|← 8 →|← 8 →|
□□□□ 1234 □□□□ Java □□□□ 5.6
1234 □□□□ Java □□□□ 5.6 □□□□
```

where the square box denotes a blank space.

## Some notes regarding the use of format specifiers

- The items must match the format specifiers in exact type. The item for the format specifier **%f** or **%e** must be a floating-point type value such as **40.0**, not **40**.
- Thus, an **int** variable cannot match **%f** or **%e**.
- The **%** sign denotes a format specifier. To output a literal **%** in the format string, use **%%**.

```
System.out.printf("The discount is %d%%", 90);
```

## Chapter Summary

- Java provides the mathematical methods **pow**, **sqrt**, **ceil**, **floor**, **round**, **min**, **max**, **abs**, and **random** in the **Math** class for performing mathematical functions.
- The character type **char** represents a single character.
- An escape sequence consists of a backslash (**\**) followed by a character or a combination of digits.
- The character **\** is called the escape character.
- A *string* is a sequence of characters. A string value is enclosed in matching double quotes (**"**). A character value is enclosed in matching single quotes (**'**).
- Strings are objects in Java. A method that can only be invoked from a specific object is called an *instance method*. A non-instance method is called a static method, which can be invoked without using an object.
- The **printf** method can be used to display a formatted output using format specifiers.

# Exercises

Suppose that s1, s2, and s3 are three strings, given as follows:

String s1 = "Welcome to Java";

String s2 = "Programming is fun";

String s3 = "Welcome to Java";

- What are the results of the following expressions?

|                         |  |
|-------------------------|--|
| 1.1 s1.equals(s3)       |  |
| 1.2 s1==s3              |  |
| 1.3 s2.compareTo(s3)    |  |
| 1.4 s1.indexOf('j')     |  |
| 1.5 s1.lastIndexOf('a') |  |
| 1.6 s1.length()         |  |
| 1.7 s1.substring(5)     |  |
| 1.8 s1.substring(5, 11) |  |

Programming I --- Ch. 4

53