Mathematical Functions, Characters, and Strings

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 3

Announcements

- Assignment 1 is due Oct 5, 11:59 PM
- Quiz 1 is Oct 7
- Assignment 2 will be released Oct 5
  - Due Oct 12, 11:59 PM
- Educational research study
  - Oct 7, weekly survey
- Reading
  - Liang
    - Chapters 2 and 4

Mathematical functions, characters, and strings

- Numerical data types (e.g., an integer)
- Numeric operations (e.g., addition)
- Mathematical functions (e.g., cosine)
- Reading numbers from the console
- Character data type (i.e., `char`)
- Comparing and testing characters
- String data type (i.e., `String`)
- Simple string methods (e.g., number of characters in the string)
- Reading a character and string from the console

Numerical data types

Name	Range	Storage Size
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

Number literals

- A literal is a constant value that appears directly in the program

```
int i = 34;
long x = 1000000;
double d = 5.0 + 1.0;
```

`34`, `100000`, `5.0`, and `1.0` are literals

Integer literals

- An integer literal can be assigned to an integer variable as long as it can fit into the variable
- A compilation error will occur if the literal is too large for the variable to hold
  - For example, the statement `byte b = 1000` would cause a compilation error, because `1000` cannot be stored in a variable of the `byte` type
- An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (equals -2147483648) to $2^{31}-1$ (equals 2147483647)
- To denote an integer literal of the `long` type, append it with the letter `L` or `l`
  - `L` is preferred because `l` (lowercase `L`) can easily be confused with `1` (the digit one)

Floating-point literals

- Floating-point literals are written with a decimal point
- By default, a floating-point literal is treated as a `double` type value
  - Example: `5.0` is considered a `double` value, not a `float` value
- You can make a number a `float` by appending the letter `f` or `F`, and make a number a `double` by appending the letter `d` or `D`
  - Example: you can use `100.2f` or `100.2F` for a float number, and `100.2d` or `100.2D` for a double number

Scientific notation

- Floating-point literals can also be specified in scientific notation
  - Example: `1.23456e+2` (same as `1.23456e2`) is equivalent to `123.456`, and `1.23456e-2` is equivalent to `0.0123456`
- E or e represents an exponent

Numeric operations

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

`double` vs `float`

- The `double` type values are more accurate than the `float` type values
  - For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.3333333333333333`

16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.33333334`

7 digits

Floating-point accuracy

- Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy
- For example,

  `System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);`

  displays `0.5000000000000001`, not `0.5`, and

  `System.out.println(1.0 - 0.9);`

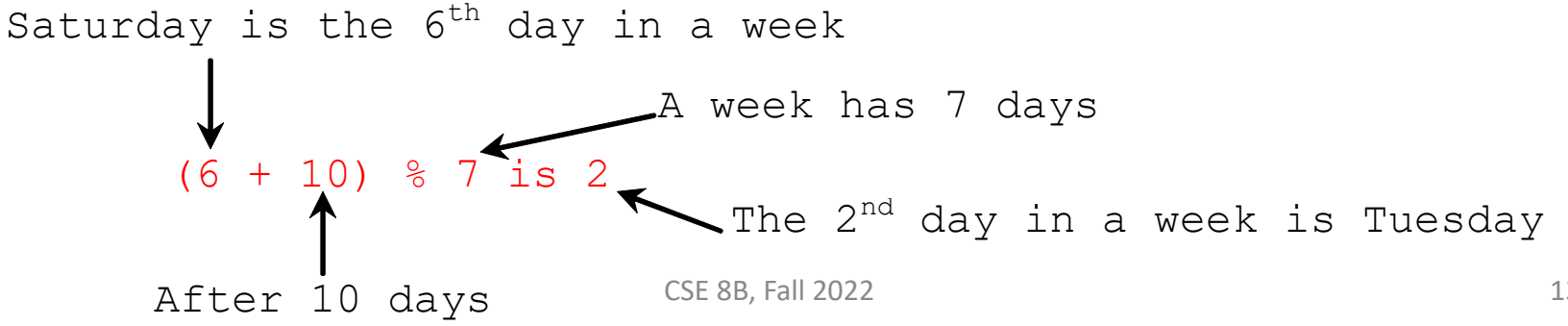
  displays `0.09999999999999998`, not `0.1`
- Integers are stored precisely
  - Calculations with integers yield a precise integer result

Integer division

- Warning: resulting fractional part (i.e., values after the decimal point) are **truncated, not rounded**
  - Example: 5 / 2 yields an integer 2

Remainder operator

- Example: an even number % 2 is always 0 and an odd number % 2 is always 1
  - You can use this property to determine whether a number is even or odd
- Example: If today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? You can find that day is Tuesday using the following expression.



Augmented assignment operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Increment and decrement operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

Conversion rules

- When performing a binary operation involving two operands of *different* types, Java automatically converts the operand based on the following rules
  1. If one of the operands is `double`, the other is converted into `double`
  2. Otherwise, if one of the operands is `float`, the other is converted into `float`
  3. Otherwise, if one of the operands is `long`, the other is converted into `long`
  4. Otherwise, both operands are converted into `int`

Type casting

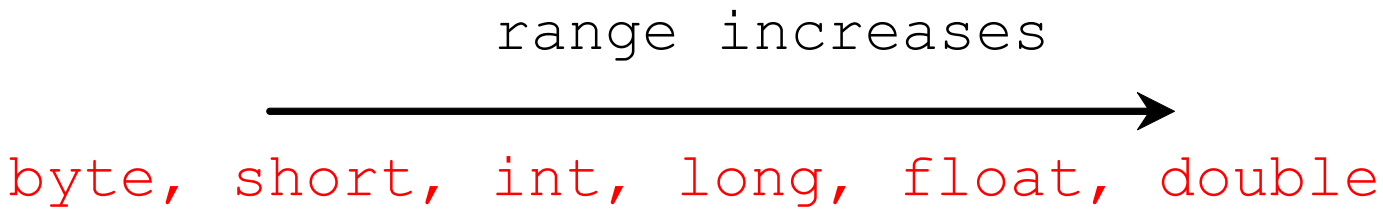
Implicit casting

```
double d = 3;
```
(type widening)

Explicit casting

```
int i = (int)3.0;
```
(type narrowing)

```
int i = (int)3.9;
```
(fraction part is truncated, not rounded!)



Operator precedence

- `()`, `var++`, `var--`
- `++var`, `--var`, `+`, `-` (unary plus and minus), `!` (not)
- (type) casting
- *, /, % (multiplication, division, and remainder)
- +, - (binary addition and subtraction)
- <, <=, >, >= (relational operators)
- ==, != (equality)
- ^ (exclusive or)
- && (and)
- || (or)
- =, +=, -=, *=, /=, %= (assignment operators)

Relational and logical operators will be covered next lecture

Operator associativity

- When two operators with the same precedence are evaluated, the associativity of the operators determines the order of evaluation
- All binary operators except assignment operators are left-associative

  `a - b + c - d` is equivalent to `((a - b) + c) - d`
- Assignment operators are right-associative

  `a = b += c = 5` is equivalent to `a = (b += (c = 5))`

Operator precedence and associativity

- The expression in the parentheses is evaluated first
  - Parentheses can be nested, in which case the expression in the inner parentheses is executed first
- When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule
- If operators with the same precedence are next to each other, their associativity determines the order of evaluation

Reading numbers from the console

1. Create a Scanner object

```
Scanner input = new Scanner(System.in);
```

2. Use the method nextDouble() to obtain to a double value. Example:

```
System.out.print("Enter a double value: ");
Scanner input = new Scanner(System.in);
double d = input.nextDouble();
```

Reading numbers from the console

```
Scanner input = new Scanner(System.in);
int value = input.nextInt();
```

Method	Description
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Explicit import and implicit Import

- At top of source file

```
import java.util.Scanner; // Explicit Import

import java.util.*; // Implicit import
```

Mathematical functions

- Java provides many useful methods in the `Math` class for performing common mathematical functions
- `Math` class constants

  `PI`

  `E`
- Math class methods
  - Trigonometric methods
  - Exponent methods
  - Rounding methods
  - `min`, `max`, `abs`, and `random` methods

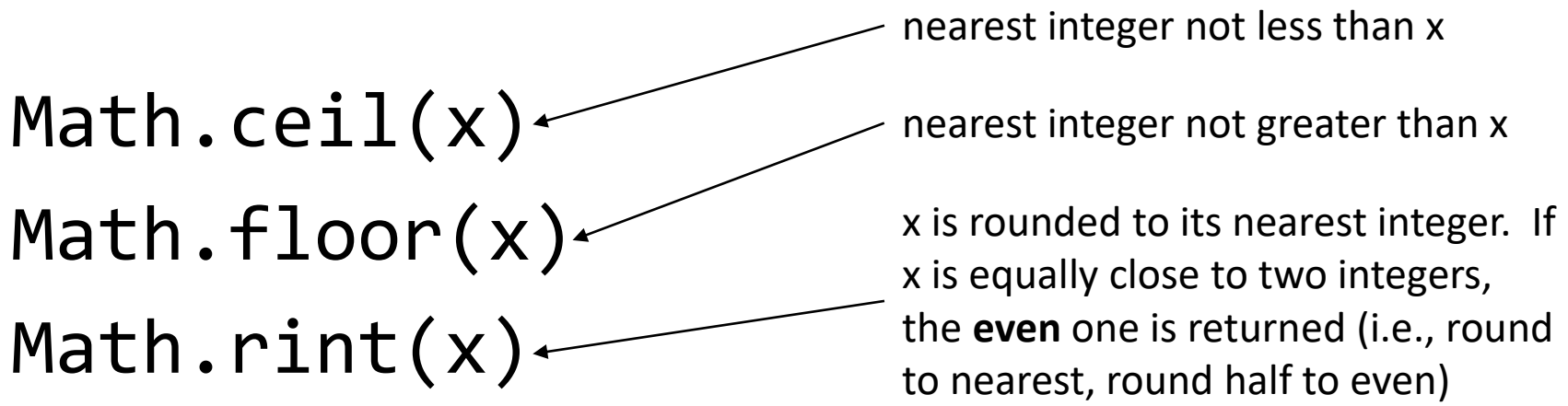
Trigonometric methods

```
Math.toDegrees(radians)
Math.toRadians(degrees)
Math.sin(radians)
Math.cos(radians)
Math.tan(radians)
Math.acos(a)
Math.asin(a)
Math.atan(a)
```

Exponent methods

`Math.exp(a)`	e^a
`Math.log(a)`	$\log_e(a)$
`Math.log10(a)`	$\log_{10}(a)$
`Math.pow(a, b)`	a^b
`Math.sqrt(a)`	$\sqrt{a}$

Rounding methods



- If you want to return an integer type, then

  `int Math.round(float x)`

  - Returns `(int)Math.floor(x + 0.5f)`

  `long Math.round(double x)`

  - Returns `(long)Math.floor(x + 0.5)`

`min`, `max`, `abs`, and `random` methods

```
Math.min(a, b)
Math.max(a, b)
Math.abs(a)
Math.random()
```

- Returns a random `double` value in the range [0.0, 1.0)

char data type

```
char letter = 'A'; // ASCII
char numChar = '4'; // ASCII
char letter = '\u0041'; // Unicode
char numChar = '\u0034'; // Unicode
```

- 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
- Unicode takes two bytes, preceded by `\u`, expressed in four hexadecimal numbers that run from `\u0000` to `\uFFFF`
  - Unicode can represent 65536 characters

Common and special 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

Escape Sequence	*Name*	*Unicode Code*	*Decimal Value*
`\b`	Backspace	`\u0008`	8
`\t`	Tab	`\u0009`	9
`\n`	Linefeed	`\u000A`	10
`\f`	Formfeed	`\u000C`	12
`\r`	Carriage Return	`\u000D`	13
`\\`	Backslash	`\u005C`	92
`\"`	Double Quote	`\u0022`	34

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");
```

- Methods in the `char` class

Relational and logical operators will be covered next lecture

Method	Description
`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOfDigit(ch)`	Returns true if the specified character is a letter or digit.
`isLowerCase(ch)`	Returns true if the specified character is a lowercase letter.
`isUpperCase(ch)`	Returns true if the specified character is an uppercase letter.
`toLowerCase(ch)`	Returns the lowercase of the specified character.
`toUpperCase(ch)`	Returns the uppercase of the specified character.

Casting between char and numeric data types

```
int i = 'a'; // Same as int i = (int)'a';

char c = 97; // Same as char c = (char)97;
```

`String` type

- The `char` type only represents one character
- To represent a string of characters, use the `String` type
- `String` is a predefined class in the Java library (just like the System class and Scanner class)

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

- The `String` type is not a primitive type; it is known as a reference type
  - Any Java class can be used as a reference type for a variable

Simple `String` methods

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.

- These methods can only be invoked from a specific string instance
  - These methods are called instance methods

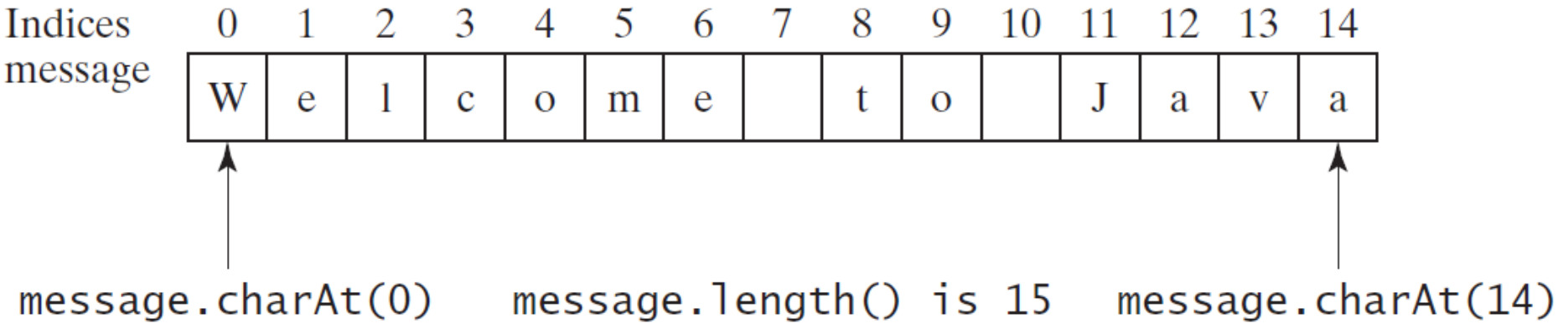
Instance methods vs static methods

- These methods can only be invoked from a specific string instance
  - These methods are called instance methods
  - The syntax to invoke an instance method is
    `referenceVariable.methodName(arguments)`
- A non-instance method is called a static method
  - **A static method can be invoked without using an object** (i.e., they are not tied to a specific object instance)
  - The syntax to invoke a static method is
    `ClassName.methodName(arguments)`
  - For example, all the methods defined in the `Math` class are static methods

Methods will be covered next week

Getting characters from a string

```
String message = "Welcome to Java";
System.out.println("The first character in message is "
    + message.charAt(0));
```



String concatenation

```
String s3 = s1.concat(s2); // These two are
String s3 = s1 + s2;       // equivalent

// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

Reading a string from the console

```
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);
```

Reading a character from the console

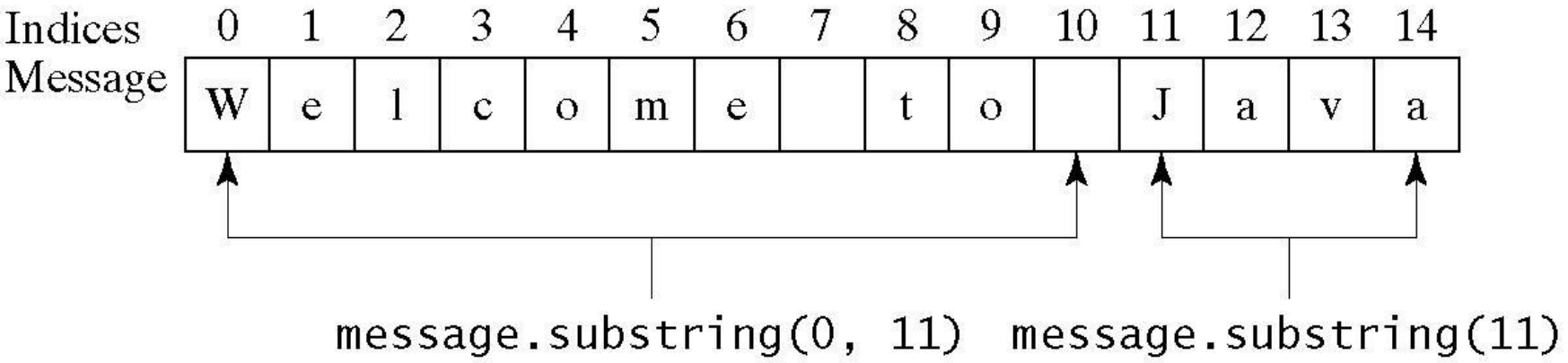
```
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

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.

Substrings

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.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Note that the character at `endIndex` is not part of the substring.

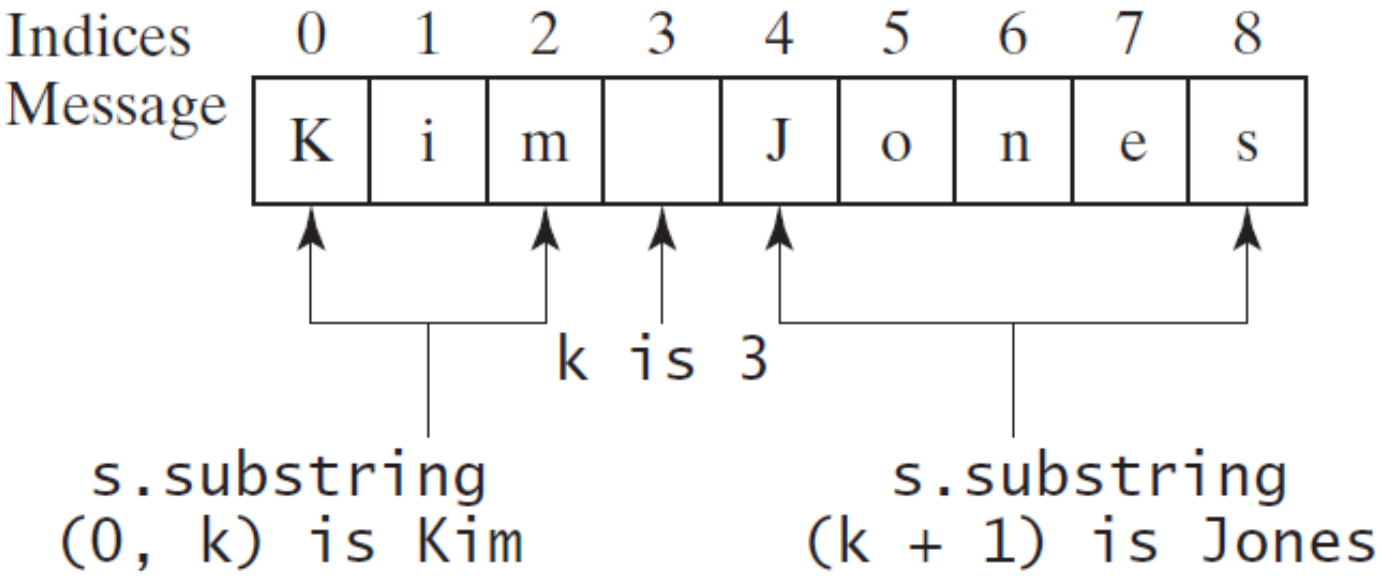


Finding a character or a substring in a string

Method	Description
`indexOf(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.

Finding a character or a substring in a string

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



Conversion between strings and numbers

```
int intValue =
  Integer.parseInt(intString);
double doubleValue =
  Double.parseDouble(doubleString);

String s = number + "";
```

Formatting output

- Use the `printf` statement

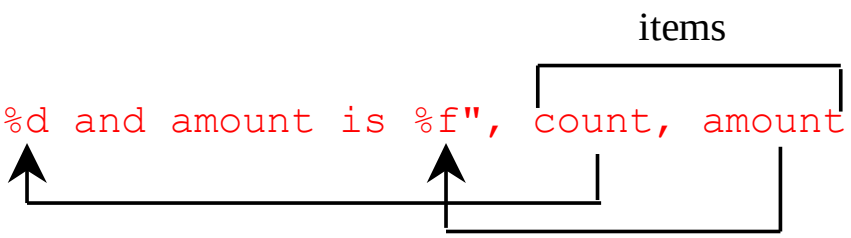
  ```
  System.out.printf(format, items);
  ```

- Where `format` is a string that may consist of substrings and format specifiers
  - A format specifier specifies how an item should be displayed
  - Each specifier begins with a percent sign
  - An item may be a numeric value, character, Boolean value, or a string

Common specifiers

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
```

Next Lecture

- Selections
- Loops
- Reading
  - Liang
    - Chapters 3 and 5