Mathematical Functions, Characters, and Strings

CSE 114, Computer Science 1
Stony Brook University

http://www.cs.stonybrook.edu/~cse114
Static methods

- Remember the main method header?
  ```java
  public static void main(String[] args)
  ```
- What does `static` mean?
  - associates a method with a particular class name
  - any method can call a `static` method either:
    - directly from within same class OR
    - using class name from outside class
The Math Class

- Class constants:
  - PI
  - E

- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods
Trigonometric Methods

- \( \sin(\text{double } a) \)
- \( \cos(\text{double } a) \)
- \( \tan(\text{double } a) \)
- \( \text{acos}(\text{double } a) \)
- \( \text{asin}(\text{double } a) \)
- \( \text{atan}(\text{double } a) \)

Radians

**Examples:**

Math.sin(0) returns 0.0
Math.sin(Math.PI / 6) returns 0.5
Math.sin(Math.PI / 2) returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6) returns 0.866
Math.cos(Math.PI / 2) returns 0

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Exponent Methods

- **exp(double a)**
  Returns $e$ raised to the power of $a$.

- **log(double a)**
  Returns the natural logarithm of $a$.

- **log10(double a)**
  Returns the 10-based logarithm of $a$.

- **pow(double a, double b)**
  Returns $a$ raised to the power of $b$.

- **sqrt(double a)**
  Returns the square root of $a$.

**Examples:**
- `Math.exp(1)` returns 2.71
- `Math.log(2.71)` returns 1.0
- `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
Rounding Methods

• **double ceil(double x)**
  x rounded up to its nearest integer. This integer is returned as a double value.

• **double floor(double x)**
  x is rounded down to its nearest integer. This integer is returned as a double value.

• **double rint(double x)**
  x is rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double.

• **int round(float x)**
  Return (int)Math.floor(x+0.5).

• **long round(double x)**
  Return (long)Math.floor(x+0.5).
Rounding Methods Examples

Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.round(2.6f) returns 3
Math.round(2.0) returns 2
Math.round(-2.0f) returns -2
Math.round(-2.6) returns -3
min, max, and abs

- **max(a, b) and min(a, b)**
  Returns the maximum or minimum of two parameters.

- **abs(a)**
  Returns the absolute value of the parameter.

- **random()**
  Returns a random double value in the range [0.0, 1.0).

**Examples:**

Math.max(2, 3) returns 3
Math.max(2.5, 3) returns 3.0
Math.min(2.5, 3.6) returns 2.5
Math.abs(-2) returns 2
Math.abs(-2.1) returns 2.1
The **random** Method

Generates a random **double** value greater than or equal to 0.0 and less than 1.0 \((0 \leq Math.random() < 1.0)\)

Examples:

\[(\text{int})(\text{Math.random()} \times 10)\]  
Returns a random integer between 0 and 9.

\[50 + (\text{int})(\text{Math.random()} \times 50)\]  
Returns a random integer between 50 and 99.

In general,

\[a + \text{Math.random()} \times b\]  
Returns a random number between \(a\) and \(a + b\), excluding \(a + b\).
Generating Random Characters

\[(\text{char})(\text{(int)'a'} + \text{Math.random()} \times (\text{(int)'z'} - (\text{int)'a'} + 1))\]

- All numeric operators can be applied to the char operands
  - The char operand is cast into a number if the other operand is a number or a character.
- So, the preceding expression can be simplified as follows:

\[(\text{char})('a' + \text{Math.random()} \times ('z' - 'a' + 1))\]
# ASCII Code for Commonly Used Characters

<table>
<thead>
<tr>
<th>Characters</th>
<th>Code Value in Decimal</th>
<th>Unicode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0' to '9'</td>
<td>48 to 57</td>
<td>\u0030 to \u0039</td>
</tr>
<tr>
<td>'A' to 'Z'</td>
<td>65 to 90</td>
<td>\u0041 to \u005A</td>
</tr>
<tr>
<td>'a' to 'z'</td>
<td>97 to 122</td>
<td>\u0061 to \u007A</td>
</tr>
</tbody>
</table>
if (ch >= 'A' && ch <= 'Z')
    System.out.println(ch + " is an uppercase letter");

if (ch >= 'a' && ch <= 'z')
    System.out.println(ch + " is a lowercase letter");

if (ch >= '0' && ch <= '9')
    System.out.println(ch + " is a numeric character");
# Methods in the Character Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isDigit(ch)</code></td>
<td>Returns true if the specified character is a digit.</td>
</tr>
<tr>
<td><code>isLetter(ch)</code></td>
<td>Returns true if the specified character is a letter.</td>
</tr>
<tr>
<td><code>isLetterOrDigit(ch)</code></td>
<td>Returns true if the specified character is a letter or digit.</td>
</tr>
<tr>
<td><code>isLowerCase(ch)</code></td>
<td>Returns true if the specified character is a lowercase letter.</td>
</tr>
<tr>
<td><code>isUpperCase(ch)</code></td>
<td>Returns true if the specified character is an uppercase letter.</td>
</tr>
<tr>
<td><code>toLowerCase(ch)</code></td>
<td>Returns the lowercase of the specified character.</td>
</tr>
<tr>
<td><code>toUpperCasse(ch)</code></td>
<td>Returns the uppercase of the specified character.</td>
</tr>
</tbody>
</table>
The String Type

- The char type only represents one character.
- To represent a string of characters, use the data type called String.

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

String is a predefined class in the Java library just like the System class.

http://java.sun.com/javase/8/docs/api/java/lang/String.html

- The String type is NOT a primitive type.
  - The String type is a reference type.
    - A String variable is a reference variable, an "address" which points to an object storing the value or actual text.
More about Strings

- Each character is stored at an index:

```java
String sentence = "A statement";
```

012345678910

- The String class (from J2SE) has methods to process strings:

```java
System.out.println("charAt(6) is " + sentence.charAt(6));
System.out.println(sentence.toUpperCase());
System.out.println(sentence.substring(0,7) + sentence.substring(14));
```
Strings are immutable!

- There are no methods to change them once they have been created.
- Any new assignment will assign a new String to the old variable.

```java
String word = "Steven";
word = word.substring(0, 5);
```

- The variable `word` is now a reference to a new String that contains "Steve".
String Concatenation

• “+” is used for making a new string by concatenating strings:

// 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
Useful String functions

- `charAt`, `equals`, `equalsIgnoreCase`, `compareTo`, `startsWith`, `endsWith`, `indexOf`, `lastIndexOf`, `replace`, `substring`, `toLowerCase`, `toUpperCase`, `trim`

- `s.equals(t)`
  - returns `true` if `s` and `t` have same letters and sequence
  - `false` otherwise
Special Characters

- `\n` — newline
- `\t` — tab
- `\"` — quotation mark

Example:

```java
String s = "<img src="./pic.jpg" />";
System.out.print(s + "\n");
```
Getting Characters from a String

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

<table>
<thead>
<tr>
<th>Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>W</td>
<td>e</td>
<td>l</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>e</td>
<td>t</td>
<td>o</td>
<td>J</td>
<td>a</td>
<td>v</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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.print("The character entered is "+ch);
Comparing Strings

• Don’t use ‘==’ to compare Strings
  • it compares their memory addresses and not actual strings (character sequences)
• Instead use the `equals` method supplied by the String class
Comparing Strings

String word1 = new String("Hello");
String word2 = new String("Hello");
if (word1 == word2){
    System.out.println(true);
} else {
    System.out.println(false);
}

Result?
Comparing Strings

String word1 = new String("Hello");
String word2 = new String("Hello");
if (word1 == word2) {
    System.out.println(true);
} else {
    System.out.println(false);
}

• Two different addresses:
  false
Comparing Strings

String word1 = new String("Hello");
String word2 = new String("Hello");
if (word1.equals(word2)) {
    System.out.println(true);
} else {
    System.out.println(false);
}
Comparing Strings

String word1 = "Hello";
String word2 = "Hello";
if (word1 == word2){
    System.out.println(true);
} else {
    System.out.println(false);
}

true

• Interned Strings: Only one instance of “Hello” is stored
• word1 and word2 will have the same address
# Comparing Strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals(s1)</td>
<td>Returns true if this string is equal to string s1.</td>
</tr>
<tr>
<td>equalsIgnoreCase(s1)</td>
<td>Returns true if this string is equal to string s1; it is case insensitive.</td>
</tr>
<tr>
<td>compareTo(s1)</td>
<td>Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or greater than s1.</td>
</tr>
<tr>
<td>compareToIgnoreCase(s1)</td>
<td>Same as compareTo except that the comparison is case insensitive.</td>
</tr>
<tr>
<td>startsWith(prefix)</td>
<td>Returns true if this string starts with the specified prefix.</td>
</tr>
<tr>
<td>endsWith(suffix)</td>
<td>Returns true if this string ends with the specified suffix.</td>
</tr>
</tbody>
</table>
## Obtaining Substrings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>substring(beginIndex)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>substring(beginIndex, endIndex)</code></td>
<td>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 9.6. Note that the character at <code>endIndex</code> is not part of the substring.</td>
</tr>
</tbody>
</table>

### Indices Message

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>e</td>
<td>l</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>e</td>
<td>t</td>
<td>o</td>
<td>J</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
message.substring(0, 11) message.substring(11)
```
### Finding a Character or a Substring in a String

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>indexOf(ch)</code></td>
<td>Returns the index of the first occurrence of <code>ch</code> in the string. Returns <code>-1</code> if not matched.</td>
</tr>
<tr>
<td><code>indexOf(ch, fromIndex)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>indexOf(s)</code></td>
<td>Returns the index of the first occurrence of string <code>s</code> in this string. Returns <code>-1</code> if not matched.</td>
</tr>
<tr>
<td><code>indexOf(s, fromIndex)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>lastIndexOf(ch)</code></td>
<td>Returns the index of the last occurrence of <code>ch</code> in the string. Returns <code>-1</code> if not matched.</td>
</tr>
<tr>
<td><code>lastIndexOf(ch, fromIndex)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>lastIndexOf(s)</code></td>
<td>Returns the index of the last occurrence of string <code>s</code>. Returns <code>-1</code> if not matched.</td>
</tr>
<tr>
<td><code>lastIndexOf(s, fromIndex)</code></td>
<td>Returns the index of the last occurrence of string <code>s</code> before <code>fromIndex</code>. Returns <code>-1</code> if not matched.</td>
</tr>
</tbody>
</table>
Finding a Character or a Substring in a String

int k = s.indexOf(' '); // 3
String firstName = s.substring(0, k);
String lastName = s.substring(k + 1);
Conversion between Strings and Numbers

String intString = "15";
String doubleString = "56.77653";

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

String s2 = "" + intValue;
The `printf` statement:

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

format is a string that may consist of substrings and format specifiers

- A format specifier begins with a percent sign and specifies how an item should be displayed: a numeric value, character, boolean value, or a string
# Frequently-Used Specifiers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%b</td>
<td>a boolean value</td>
<td>true or false</td>
</tr>
<tr>
<td>%c</td>
<td>a character</td>
<td>'a'</td>
</tr>
<tr>
<td>%d</td>
<td>a decimal integer</td>
<td>200</td>
</tr>
<tr>
<td>%f</td>
<td>a floating-point number</td>
<td>45.460000</td>
</tr>
<tr>
<td>%e</td>
<td>a number in standard scientific notation</td>
<td>4.556000e+01</td>
</tr>
<tr>
<td>%s</td>
<td>a string</td>
<td>&quot;Java is cool&quot;</td>
</tr>
</tbody>
</table>

```java
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %.2f", count, amount);
```

Displays: count is 5 and amount is 45.56
Bitwise operations in java

- To write programs at the machine-level, often you need to deal with binary numbers directly and perform operations at the bit-level

- Java provides the bitwise operators and shift operators
  - The bit operators apply only to integer types (byte, short, int, and long)
  - All bitwise operators can form bitwise assignment operators, such as =: |=, <<=, >>=, and >>>=

- Bitwise AND: &
  - 10101110 & 10010010 yields 10000010
  - The AND of two corresponding bits yields a 1 if both bits are 1
Bitwise operations in java

- **Bitwise OR:** |

- The OR of two corresponding bits yields a 1 if either bit is 1

- \(10101110 \lor 10010010\) yields \(10111110\)

```java
class BitwiseOR {
    public static void main(String[] args) {
        int number1 = 12, number2 = 25, result;
        result = number1 | number2;
        System.out.println(result);
    }
}
```
Bitwise operations in java

- Bitwise exclusive OR: ^
  - 10101110 ^ 10010010 yields 00111100
  - The XOR of two corresponding bits yields a 1 only if two bits are different.

- One’s complement: ~
  - ~10101110 yields 01010001
  - The operator toggles each bit from 0 to 1 and from 1 to 0.

- Left shift: <<
  - 10101110 << 2 yields 10111000
  - The operator shifts bits in the first operand left by the number of bits specified in the second operand, filling with 0s on the right.
Bitwise operations in java

- Right shift with sign extension: `>>`
  - `10101110 >> 2` yields `11101011`
  - `00101110 >> 2` yields `00001011`
  - The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with the highest (sign) bit on the left.

- Unsigned right shift with zero extension: `>>>`
  - `10101110 >>> 2` yields `00101011`
  - `00101110 >>> 2` yields `00001011`
  - The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with 0s on the left.
Constants in binary format

byte fourTimesThree = 0b1100;
byte data = 0b0000110011;
short number = 0b1111111111111111;
int overflow = 0b10101010101010101010101010101011;
long bow = 0b101010101010101010101010101010111L;

• Just be careful not to overflow the numbers with too much data, or else you'll get a compiler error:
byte data = 0b1100110011;
// Type mismatch: cannot convert from int to byte

• New feature in Java 7 known as numeric literals with underscores:
int overflow = 0b1010_1010_1010_1010_1010_1010_1010_1010_1011;
long bow = 0b1__01010101__01010101__01010101__010101111L;
Constants in octal and hexadecimal format

```c
int x = 06;    //octal
int y = 0xff;  //hexadecimal
```