Mathematical Functions, Characters, and Strings

CSE160, Computer Science A: Honors
Stony Brook University

http://www.cs.stonybrook.edu/~cse160
Static methods

- Remember the **main** method header?
  ```java
  public static void main(String[] args)
  ```
- What does **static** mean?
  - all inputs will be provided as parameters
  - 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 – e.g.,
      `Math.sin(0)`
The Math Class

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

- Also some class constants (final static):
  - PI
  - E
Trigonometric Methods

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

**Radians**

- **Examples:**
  - \( \text{Math.sin(0)} \) returns 0.0
  - \( \text{Math.sin(Math.PI / 6)} \) returns 0.5
  - \( \text{Math.sin(Math.PI / 2)} \) returns 1.0
  - \( \text{Math.cos(0)} \) returns 1.0
  - \( \text{Math.cos(Math.PI / 6)} \) returns 0.866
  - \( \text{Math.cos(Math.PI / 2)} \) returns 0
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
• `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 ($0 \leq \text{Math.random()} < 1.0$)

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 \(a + b\).
# 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>
Generating Random Characters

(char)((int)'a' + Math.random() * ((int)'z' - (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:

(char)('a' + Math.random() * ('z' - 'a' + 1))
Comparing and Testing Characters

```java
if ('A' <= ch && ch <= 'Z')
    System.out.println(ch + " is an uppercase letter");

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

if ('0' <= ch && ch <= '9')
    System.out.println(ch + " is a numeric character");
```
There are also 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>toUpperCase(ch)</code></td>
<td>Returns the uppercase of the specified character.</td>
</tr>
</tbody>
</table>
The String Class/Type

- The `char` type only represents one character
- To represent a string of characters, we 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 API: [http://java.sun.com/javase/11/docs/api/java/lang/String.html](http://java.sun.com/javase/11/docs/api/java/lang/String.html)

- The `String` type is NOT a primitive type
- The `String` type is a reference type
  - That is, a `String` variable is a reference variable, an "address" which points to an object storing the value (the 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 reference to the old variable

```java
String word = "Steven";
word = word.substring(0, 5);
```
  
- the variable word is now a new reference to a new String that contains "Steve"
String Concatenation

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

```java
// 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

\" – quotation mark
\n – newline
\t – tab

• Example:

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

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

Indices

<table>
<thead>
<tr>
<th></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>

message.charAt(0)  message.length() is 15  message.charAt(14)
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/1` 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);
}

false

- Two different addresses
Stack

```java
int i = 5;
String s1 = "Paul";
String s2 = "Paul";
```

Global segment

```
String
-value: ['P', 'a', 'u', 'l']
```

```
s1==s2
true
s1.equals(s2)
true
```
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);
}

true
There is one exception

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

true
```

- Compiled Strings are **Interned**: only one instance of “**Hello**” is actually stored
  - `word1` and `word2` will have the same address
  - only for Strings assigned during compilation
Comparing Strings

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

- `.equals` still works, so use it
String \( s^3 \) = new String ("Paul");

String \( s^4 \) = new String ("Paul");

\( s^3 == s^4 \)
false

\( s^3 . \)equals (\( s^4 \))
true
Comparing Strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equals(s1)</code></td>
<td>Returns true if this string is equal to string s1.</td>
</tr>
<tr>
<td><code>equalsIgnoreCase(s1)</code></td>
<td>Returns true if this string is equal to string s1; it is case insensitive.</td>
</tr>
<tr>
<td><code>compareTo(s1)</code></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><code>compareToIgnoreCase(s1)</code></td>
<td>Same as <code>compareTo</code> except that the comparison is case insensitive.</td>
</tr>
<tr>
<td><code>startsWith(prefix)</code></td>
<td>Returns true if this string starts with the specified prefix.</td>
</tr>
<tr>
<td><code>endsWith(suffix)</code></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.</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>. Note that the character at <code>endIndex</code> is not part of the substring.</td>
</tr>
</tbody>
</table>

## Indices

<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>Welcome to Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

```java
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;
Formatting Output

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>

You can also specify dimensions, such as, the number of digits after decimal period

```java
int count = 5;
double amount = 45.5678;
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: `&`
  - The AND of two corresponding bits yields a 1 if both bits are 1
    - `10101110 & 10010010` yields `10000010`
Bitwise OR: 

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

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

Output: 29 because

```
12    0000 1100 |
25 is 0001 1001 =
29 is 0001 1101
```
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

• 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
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

• Feature new since 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__01010111L;
• _ is just ignored
Bitwise operations in Java

- Right shift with **sign extension**: `>>`
  - `-0b1010 >> 1 (-10) yields -0b101 (-5)`
  - `0b1010 >> 1 (10) yields 0b101 (5)`
  - 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**: `>>>`
  - `-0b1010 >>> 1 (-10) yields the integer 2147483643`
    - because `-0b1010` is represented inside with a 1 for the sign bit (the left most bit)
  - `0b1010 >>> 1 (10) yields 0b101 (5)`
  - 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 octal and hexadecimal format

```c
int x = 010;    //octal for 8
int y = 0xff;   //hexadecimal for 255
```