Using Java Classes

15-110 Summer 2010
Margaret Reid-Miller
The Math Class

- The Math class is one of many classes in the Java class libraries with predefined code. It contains
  - mathematical constants and
  - methods that perform common mathematical operations.
- These methods require argument (data) on which to perform their actions, and return a result that can be used in an expression in your program.
- A complete description of the Math class is in the Java API online.
  - http://java.sun.com/j2se/1.5.0/docs/api
Square Root

static double sqrt(double n)

- **static** indicates that we call this method using the name of the class.
- **double** indicates the data type of the answer the method returns.
- **sqrt** is the name of the method.
- **(double n)** indicates that the method requires one double argument to do its job.
- **BEHAVIOR:** returns the square root of the number supplied in the argument.

Example:

double answer = Math.sqrt(16.0);  // returns 4.0


Exponentiation

\[ \text{static double pow(double num, double power)} \]

- **static** indicates that we call this method using the name of the class.
- **double** indicates the data type of the answer the method returns.
- **pow** is the name of the method.
- **(double num, double power)** indicates that this method requires two double arguments on which to compute its result.
- **BEHAVIOR:** pow returns num raise to the specified power.

**Example:**

\[
\text{double answer = Math.pow(2.0, 3.0);} \quad // \text{returns 8.0}
\]
The Math Class

Math constants:
The constants $e$ and $\pi$ are defined in the Math class. By convention, names of constants are in all upper case:

- Math.E and Math.PI

Some Math methods:

```java
static double floor(double num)
static double ceil(double num)
static double sqrt(double num)
static int abs(int num)
static double abs(double num)
```

This is an example of method overloading, where abs is defined two ways.
Examples

double area = Math.PI * radius * radius;

double circumference = 2.0 * Math.PI * radius;

double side;
side = Math.sqrt(2.0 * Math.pow(radius, 2.0));

System.out.println("The square area is "
    + "at least " + Math.floor(side * side));
Calling Methods

• The argument to a method is be a literal, variable, or an expression that evaluates to a value.

• The argument value must match the data type specified in the method header.

• Multiple argument values must match the number and order specified in the method header.

• The results of these methods should be the argument to another method (e.g., print), assigned to a variable, or used as part of a larger expression.

Example:

Math.abs(-4.0);  // WRONG: has no effect!
Generating Random Numbers

- The `random` method of the `Math` class generates a random number in the range \([0.0,1.0)\).  
  
  \[
  \text{includes } 0.0 \quad \text{excludes } 1.0
  \]

- The number is not truly random; it is \text{pseudo-random}.
- The number is (approximately) \text{uniformly distributed} in the range.

Example:

\[
\text{double randNum} = \text{Math.random}();
\]
Generating Random Numbers

- To generate a random number in a different range, we can scale (multiply) and/or translate (add) the random number to get a new random number in that range.

Examples:
- Generate a random double in [0.0, 8.0):
  \[
  \text{double } \text{randNum} = \text{Math.random()} * 8.0;
  \]
- Generate a random double in [10.0, 25.0):
  \[
  \text{double } \text{randNum} = \text{Math.random()} * 15.0 + 10.0;
  \]
Generating Random Integers

We can use Math.random to generate a random integer in some range: generate a random number in the range and then typecast to an integer.

Example:

- To generate a random integer in \{0, 1, 2, \ldots, 12\}:
  - How many different integers do we want to generate? 13.
  - Generate a random double in \([0.0, 13.0)\).
  - Each range \([0,1), [1,2), \ldots, [12,13)\) corresponds to integers 0, 1, 2, \ldots, 12, respectively.

\[
\text{int randNum} = (\text{int}) (\text{Math.random()} * 13.0);
\]
Generating Random Integers

Generate a random integer in \{5, 15, 25, \ldots, 75\}:

1. How many different integers do we want to generate?

\[
\text{Math.random(); } \quad [0.0, 1.0);
\text{Math.random() * 8.0; } \quad [0.0, 8.0);
\text{(int) (Math.random() * 8.0) } \quad \{0,1, 2,\ldots, 7\}
\]

2. What is the difference between pairs of numbers?

\[
\text{(int) (Math.random() * 8.0) * 10; } \quad \{0,10, 20,\ldots, 70\}
\]

3. What is the first number?

\[
\text{(int) (Math.random() * 8.0) * 10 + 5; } \quad \{5, 15, 25, \ldots, 75\}
\]
Java Data: Primitive vs Objects

Primitive data:

- Data uses a small **fixed** amount of memory.
- There are exactly eight primitive data types. 
  
  byte, short, int, long, float, double, char, boolean
- Primitive data types names are in all lower case.
- A primitive is only data and has no other special abilities.
- You cannot define new primitive data types.
Java Data: Primitive vs Objects

Objects:

- An object has both *state* and *behaviors*.
- An object’s current *state* (data) is defined by the values for its attributes. These values are stored internally and may require a little or a lot of memory.
- An object’s *behaviors* (methods) are the actions it can perform.
- The *type* (or category) of an object is its *class*.
- Java has many classes already defined, *e.g.*, String, System, Scanner.
  (Recall: Class names start with a capital letter.)
Class as a type

- A class is like a blueprint from which an object is created.
- We can create many objects from the class.
- The differences among these objects are the attribute values (data) that define the objects’ state.
- For example, a class `Student` might be used to create a student object.
  - All such objects would have attributes common to students (e.g., name, andrewId, courses enrolled…).
  - But each object would have its own values for these attributes, depending on which student it represents.

  (Later we will see how we can define new classes.)
Object State

- What state (data) an object holds internally often is *hidden* from us, the user of the object. We cannot access the data directly.

- For example, a `String` object holds the string of characters in the object. But it might hold other hidden information relating to the string, *e.g.*, the length of the string.

- Another example is the `System.out` object that holds information about how and to where to write text to the console.
  - We do not need to access these data in order to use the object; we just need to call the `print` or `println` methods.
Object Behaviors

• To use an object, we need to know only the *behaviors* of an object.
• An object’s behaviors are defined by a set of methods associated with the object.
• For example, methods may enable you to access or change an object’s attribute values, or to ask the object to perform a task.
• These methods are known as the *interface* to the object.
String Objects

• An **object** of type `String` holds a sequence of (unicode) characters.

• When we **declare** a variable of type `String`, it does not create an object. All you get is a way to refer to the object:

  ```java
  String founder;
  ```

• To **create** an object we use the `new` operator:

  ```java
  founder = new String("Carnegie");
  ```

  * constructor: sets up the object

• Strings have a shortcut way of creating them:

  ```java
  String founder2 = "Mellon";
  ```
Object vs Primitive Data

- A primitive variable holds an actual value:

  ```java
  int count = 15100;
  ```

- An object variable holds a reference (address) to the object (how to find the object).

  ```java
  String founder = new String("Carnegie");
  ```
Escape Sequences

- How do you include a " in a String literal?
- You cannot have a String literal break across lines. How do you include a line break?
- Solution: An escape sequence is a two-character sequence that represent a single special character.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>tab character</td>
</tr>
<tr>
<td>\n</td>
<td>newline character</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>\’</td>
<td>single quote</td>
</tr>
<tr>
<td>\</td>
<td>backslash character</td>
</tr>
</tbody>
</table>
String Length

```java
int length()
```

**BEHAVIOR:** Returns the number of characters in this string.

- () indicates the `length` method needs no argument values to do its job.
- Because `String` objects may have different lengths, you need to ask the object for its length. (It is non-static method.)
- **Example:**
  ```java
  String founder = "Carnegie";
  int numChar = founder.length();
  ```
Getting a single character

char charAt(int index)

BEHAVIOR: Returns the character at a specified index.

- Each character in a string has an **index**

<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>C</td>
<td>a</td>
<td>r</td>
<td>n</td>
<td>e</td>
<td>g</td>
<td>i</td>
<td>e</td>
<td>M</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>

Example:

```java
String school = "Carnegie Mellon";
char firstChar = school.charAt(0);
```

**WARNING:** You cannot assign a char to a object of type String without first converting the char to a String object!

  ```java
e.g., String initial = "" + firstChar;
  ```
Substrings

String substring(int startIndex, 
    int endIndex)

**BEHAVIOR:** Returns a new string consisting of the substring starting at `startIndex` (inclusive) and ending at `endIndex` (exclusive).

Example:

```java
String school = "Carnegie Mellon";
String founder = school.substring(0, 8);
String founder2 = school.substring(9, 15);
```

**Note:** length of substring is `endIndex - startIndex`
Substrings

String substring(int startIndex)

BEHAVIOR: Returns a new string consisting of the substring starting at startIndex and ending at the last character in the string.

Example:

String school = "Carnegie Mellon";
String founder2 = school.substring(9);
returns "Mellon"
Replacing Characters

String replace(char oldChar, char newChar)

BEHAVIOR: Returns a new String object resulting from replacing every occurrence of oldChar with newChar.

- The original String object is unchanged. (Strings are immutable!)

Example:

String founder = "Carnegie";
System.out.println(
    founder.replace('e', 'E'));
System.out.println(founder);

OUTPUT:
CarnEgiE
Carnegie
### Changing Case

**String toUpperCase()**

**BEHAVIOR:** Returns a new `String` object with all letters converted to uppercase.

**String toLowerCase()**

**BEHAVIOR:** Returns a new `String` object with all letters converted to lowercase.

**Example:**

```java
String founder = "Carnegie";
String upper = founder.toUpperCase();
String lower = founder.toLowerCase();
```

**Immutable:** You need to print or assign the result to a variable!
Method order

- The dot(.) operator is evaluated from left to right.
- If a method returns an object, you can invoke one of the returned object’s methods.
- Example:

```java
String school = "Carnegie Mellon";
System.out.println(
    school.substring(9).toLowerCase());
```

"Mellon"

"mellon"
Reading User Input

- The **Scanner** class has methods for reading user input values while the program is running.
- The **Scanner** class is in the *java.util* package.
  - Related classes are grouped into *packages*.
  - Most of the classes we use are in the *java.lang* package, which is always available.
  - To use classes in other packages we must tell the compiler about these packages by using an *import declaration* before the classheader:

  ```java
  import java.util.Scanner;
  ```

  ```java
  public class interactiveProgram { ...}
  ```
Scanner Object

• First, we need to create a Scanner object using the new operator:

```java
Scanner console = new Scanner(System.in);
```

• `console` is a variable that refers to the Scanner object.
• `Scanner()` is the constructor that helps set up the object.
• `System.in` is an object that refers to the standard input stream which, by default, is the keyboard.
Scanner Methods

String nextLine()

- Reads and returns the next line of input.

String next()

- Reads and returns the next token (e.g., one word, one number).

double nextDouble()

- Reads and returns the next token as a double value.

int nextInt()

- Reads and returns the next token as an int value.

These methods pause until the user has entered some data and pressed the return key.
import java.util.Scanner;

Scanner Example

Scanner console = new Scanner(System.in);
System.out.print("What is the make of your vehicle? ");
String vehicleMake = console.nextLine();

System.out.print("How many miles did you drive? ");
int miles = console.nextInt();

System.out.print("How many gallons of fuel did you use? ");
double gallons = console.nextDouble();
Scanner Caveats

- `nextInt`, `nextDouble`, and `next` read one token at a time.
  - Tokens are *delimited* by whitespace (space, tab, newline characters)
  - Several values can be on the same input line or on separate lines.
- `nextLine` reads the rest of the line and moves to the next line. It may return a string of no characters if it is called after calling one of the above methods.
- What happens if the user does not enter an integer when we use `nextInt`?