Labs and Tutorials
This week is Lab #2. It has a pre-lab!

Midterms – Save the Dates!
- Midterm #1 is 5:30-6:30pm on February 10 (Tuesday) in Woodward IRC 2
- Midterm #2 is 6-7pm on March 11 (Wednesday) in Woodward IRC 2

You must notify me well in advance (I will specify deadlines as the dates approach) if you have an unavoidable conflict with a midterm.

Learning Goals
By the end of the next several lectures you will be able to...
- Explain the concepts of “class”, “object”, and “method” as used in computer programming.
- Find pre-defined classes in the Java library, understand the documentation, and use the methods for basic classes.
- Write programs that use common Java classes like BigInteger, String, and Scanner.

Reading
Your textbook is *Big Java* (3rd Ed).

This Week’s Reading: Ch 2.6-2.10, Finish Ch 4

Learning Goals
By the end of class today you will be able to...
- Describe what classes, objects, and methods are.
- Find the on-line documentation for Java’s library of pre-defined classes.
- Use constructors to create new objects, and explain why this is necessary.
- Explain what syntax, runtime, and logical errors are.
Last Time...
- We got more comfortable with the Java primitive types and operators.
- We learned the rules for when Java will convert types automatically.
- We learned how to force conversion ("cast") from one type to another.
- We learned how to declare constants and the rationale for doing so.

Finishing Off from Last Time...
- We got more comfortable with the Java primitive types and operators.
- We learned the rules for when Java will convert types automatically.
- We learned how to force conversion ("cast") from one type to another.
- We learned how to declare constants and the rationale for doing so.

Casting
- **Casting:** force Java to convert from one type to another, even with information loss
- Converting from real to integer
  - `int shoes = (int) 1.5;`
- Truncation: fractional part thrown away
  - `int shoes = (int) 1.75;`
  - `int shoes = (int) 1.25;`
- Rounding: must be done explicitly
  - `shoes = Math.round(1.99);`

Converting Between Types
```java
//*****************************************
// Feet.java   Author: Tamara
// What type of things can be put on feet?
//*****************************************
public class Feet
{
    public static void main (String[] args)
    {
        int shoes = 2;
        int socks = (int) 1.75;
        System.out.println("shoes = " + shoes + " socks = " + socks);
        int toes = Math.round(1.99);
        System.out.println("toes = " + toes);
    }
}
```

What's wrong? Let's try it...

Converting Between Types
```java
//*****************************************
// Feet2.java   Author: Tamara
// What type of things can be put on feet?
//*****************************************
public class Feet2
{
    public static void main (String[] args)
    {
        int shoes = 2;
        int socks = (int) 1.75;
        System.out.println("shoes = " + shoes + " socks = " + socks);
        long toes = Math.round(1.99);
        System.out.println("toes = " + toes);
    }
}
```

Or Tell Java with a Type Cast
```
//*****************************************
// Feet2.java   Author: Tamara
// What type of things can be put on feet?
//*****************************************
public class Feet2
{
    public static void main (String[] args)
    {
        int shoes = 2;
        int socks = (int) 1.75;
        System.out.println("shoes = " + shoes + " socks = " + socks);
        int toes = (int) Math.round(1.99);
        System.out.println("toes = " + toes);
    }
}
Questions?

What Changes, What Doesn't?

Motivation for Constants

Suppose we want to write a program to compute the distance in miles to various stars, for which we know the distance in light-years.

Constants

Things that do not vary
- unlike variables
- will never change

Syntax:
- final typeName variableName;
- final typeName variableName = value;

Constant names in all upper case
- Java convention, not compiler/syntax requirement

Programming With Constants

public static void main (String[] args)
{
    double lightYears, milesAway;
    final int LIGHTSPEED = 186000;
    final int SECONDS_PER_YEAR = 60*60*24*365;
    lightYears = 4.35; // to Alpha Centauri
    milesAway = lightYears * LIGHTSPEED * SECONDS_PER_YEAR;
    System.out.println("lightYears: " + lightYears + " miles " + milesAway);
    lightYears = 68; // to Aldebaran
    milesAway = lightYears * LIGHTSPEED * SECONDS_PER_YEAR;
    System.out.println("lightYears: " + lightYears + " miles " + milesAway);
}

Programming With Constants

public static void main (String[] args)
{
    double lightYears, milesAway;
    final int LIGHTSPEED = 186000;
    final int SECONDS_PER_YEAR = 60*60*24*365;
    final double ALPHACENT_DIST = 4.35; // to Alpha Centauri
    final double ALDEBARAN_DIST = 68; // to Aldebaran
    lightYears = ALPHACENT_DIST;
    milesAway = lightYears * LIGHTSPEED * SECONDS_PER_YEAR;
    System.out.println("lightYears: " + lightYears + " miles " + milesAway);
    lightYears = ALDEBARAN_DIST;
    milesAway = lightYears * LIGHTSPEED * SECONDS_PER_YEAR;
    System.out.println("lightYears: " + lightYears + " miles " + milesAway);
}
Avoiding “Magic Numbers”

- "magic numbers": numeric constants directly in code
- almost always bad idea!
  - hard to understand code
  - hard to make changes
  - typos possible
- use constants instead

Using Classes and Objects

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>approx (-3.4\times10^{38}) (7 sig.digits)</td>
<td>approx (3.4\times10^{38}) (7 sig.digits)</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>approx (-1.7\times10^{308}) (15 sig. digits)</td>
<td>approx (1.7\times10^{308}) (15 sig. digits)</td>
</tr>
</tbody>
</table>

- Six primitives for numbers
  - fixed size, so finite capacity
  - integer vs. floating point

Operators and Operands

- Programming is all about specifying
  - data that is to be manipulated or acted upon
  - operations that can act upon data
  - order in which operations are applied to data

- So far: specify data using primitive data types
  - come with pre-defined operations like
    - +, -, *, and /

Questions?

Programming with Classes

- What if data we want to work with is more complex than these few primitive data types?
Programming with Classes

- What if data we want to work with is more complex than these few primitive data types?
- We can make our own data type:
  - specifies nature of data we want to work with
  - operations that can be performed on that kind of data
- (Other people can make data types, too, and we can use their work.)

Object-Oriented Terminology

- For historical reasons, there are special terms for these new data types we create:
  - The new data types are called **classes**.
  - Values with the new data type are called **objects**.
  - The operators for the new data type are called **methods**.
- (The programming style we are teaching, which is the dominant style in industry today, is called “object-oriented programming”.)

Primitive Types vs. Classes

<table>
<thead>
<tr>
<th>Primitive Types</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-defined in Java</td>
<td>Written by other programmers or by you</td>
</tr>
<tr>
<td>Simplest things, e.g., int</td>
<td>Can be arbitrarily complex</td>
</tr>
<tr>
<td>Operators: +, -, ...</td>
<td>Methods</td>
</tr>
<tr>
<td>Values belong to types. E.g., 3 is an int, 3.14159 is a double</td>
<td><strong>Objects</strong> belong to classes E.g., a Toyota Prius is a car.</td>
</tr>
</tbody>
</table>

Class Libraries

- Before making new class yourself, check to see if someone else did it already
  - libraries written by other programmers
  - many built into Java
- Examples (built into Java)
  - BigInteger (java.math.BigInteger) lets you compute with arbitrarily big integers.
  - Date (java.util.Date) lets you get the current time.
  - Calendar (java.util.Calendar) does fancy date computations.
- Appendix C in your book lists common ones.

API Documentation

- Online Java library documentation at [http://java.sun.com/javase/6/docs/api](http://java.sun.com/javase/6/docs/api)
  - let’s take a look!
- Everything we need to know: critical details
  - and often many things far beyond current need
- Classes in libraries are often referred to as Application Programming Interfaces
  - or just API

Example: BigInteger

- Let’s write a simple program to print out how much money you’d make if your compensation package is a combination of salary and stock options.
public class GetRichQuick {
    public static void main(String[] args) {
        int salary = 123456;
        int stockOptions = 10000;
        int profitPerShare = 30;

        int optionCompensation = stockOptions * profitPerShare;
        int totalCompensation = salary + optionCompensation;

        System.out.println("Total Compensation = "+ totalCompensation);
    }
}

public class GetRichQuick {
    public static void main(String[] args) {
        int salary = 11122334445556666777888999;
        int stockOptions = 10000000000000000000000;
        int profitPerShare = 3000000000000000000000;

        int optionCompensation = stockOptions * profitPerShare;
        int totalCompensation = salary + optionCompensation;

        System.out.println("Total Compensation = "+ totalCompensation);
    }
}

import java.math.BigInteger; //Tell Java to use BigInteger pkg
public class GetRichQuick {
    public static void main(String[] args) {
        BigInteger salary = new BigInteger("11122334445556666777888999");
        BigInteger stockOptions = new BigInteger("10000000000000000000000");
        BigInteger profitPerShare = new BigInteger("3000000000000000000000");

        BigInteger optionCompensation = stockOptions.multiply(profitPerShare);
        BigInteger totalCompensation = salary.add(optionCompensation);

        System.out.println("Total Compensation = "+ totalCompensation);
    }
}

import java.math.BigInteger; //Tell Java to use BigInteger pkg
public class GetRichQuick {
    public static void main(String[] args) {
        BigInteger salary = new BigInteger("11122334445556666777888999");
        BigInteger stockOptions = new BigInteger("10000000000000000000000");
        BigInteger profitPerShare = new BigInteger("3000000000000000000000");

        BigInteger optionCompensation = stockOptions.multiply(profitPerShare);
        BigInteger totalCompensation = salary.add(optionCompensation);

        System.out.println("Total Compensation = "+ totalCompensation);
    }
}
Recap: Using classes
- Import the name of the class you want:
  ```java
  import java.math.BigInteger;
  ```
- Declare variables using the class name:
  ```java
  BigInteger salary;
  ```
- Create values (objects) of the new type/class by using constructors:
  ```java
  salary = new BigInteger("111222333");
  ```
- Operate on the objects using methods:
  ```java
  totalCompensation = salary.add(bonus);
  ```

Literals
- With the primitive types, how do you create values with that type?
  E.g., how do we create integer values?
    1. You type some digits, like 3, or 42
    2. You combine integer-valued things with operators that work on integers, e.g.,
       ```java
       3+42*(a-b)
       ```
- A bunch of digits are an integer literal.
  It’s the basic way to create an integer value

More Literals
- How about a value of type double?
  1. You type a bunch of digits with a decimal point, and optionally the letter e or E followed by an exponent
  2. You can combine doubles with operators that work on doubles.

Recap: Using classes
- Import the name of the class you want:
  ```java
  import java.math.BigInteger;
  ```
- Declare variables using the class name:
  ```java
  BigInteger salary;
  ```
- Create values (objects) of the new type/class by using constructors:
  ```java
  salary = new BigInteger("111222333");
  ```
- Operate on the objects using methods:
  ```java
  totalCompensation = salary.add(bonus);
  ```

More Literals
- How about a value of type double?
  1. You type a bunch of digits with a decimal point, and optionally the letter e or E followed by an exponent
  2. You can combine doubles with operators that work on doubles.

Those are literals!
Long Literals
- How about values of type long?
  1. You type a bunch of digits followed by the letter l or L
  2. You combine previously created longs

Literals – General Pattern
- To create values of a primitive type:
  1. There’s some way to type a literal
  2. There are operators that create values of the given type.

Literals for Classes?
- Classes are like primitive types, except they can be defined any way you like, and they can be much more complex.
- How to create a value (an object) of a given class?
  1. Invent some way to type a literal???
  2. Operators that create objects of that class (methods).

Constructors!
- A constructor is the equivalent of a literal for a class. It’s how you create a new object that belongs to that class.
- Examples:
  new BigInteger("999999")
  new Rectangle(10, 20, 30, 40)
  new UBCStudent("Alan Hu", 12345678, …)

Constructor Syntax
- The reserved word new
- Followed by the name of the class
- Followed by an open parenthesis (, followed by any parameters (information needed to construct the object)
- Followed by a closing parenthesis )

Using Constructors
- Use a constructor just as you’d use a literal.
  Example:
  - For the int type:
    int a = 3;
  - For the BigInteger class:
    BigInteger a = new BigInteger("3");
### Primitive Types vs. Classes

<table>
<thead>
<tr>
<th>Primitive Types</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-defined in Java</td>
<td>Written by other programmers or by you</td>
</tr>
<tr>
<td>Simplest things, e.g., <code>int</code></td>
<td>Can be arbitrarily complex</td>
</tr>
<tr>
<td>Operators: <code>+</code>, <code>-</code>, …</td>
<td>Methods</td>
</tr>
<tr>
<td>Values belong to types. E.g., 3 is an <code>int</code>, 3.14159 is a <code>double</code></td>
<td>Objects belong to classes E.g., you are a UBC Student</td>
</tr>
<tr>
<td>Literals</td>
<td>Constructors</td>
</tr>
</tbody>
</table>

### Program Development

- Use an editor to create your Java program
  - often called `source code`
  - code used interchangeably with `program` or `instructions` in the computer world
- Another program, a `compiler` or an `interpreter`, translates `source code` into `target language` or `object code`, which is often `machine language`
- Finally, your computer can execute `object code`

### Syntax

- Rules to dictate how statements are constructed.
  - Example: open bracket needs matching close bracket
  - If program is not syntactically correct, cannot be translated by compiler
  - Different than humans dealing with natural languages like English. Consider statement with incorrect syntax (grammar)
    - for weeks. rained in Vancouver it hasn’t
    - we still have pretty good shot at figuring out meaning

### Semantics

- What will happen when statement is executed
- Programming languages have well-defined semantics, no ambiguity
- Different than natural languages like English. Consider statement:
  - Mary counted on her computer.
  - How could we interpret this?
- Programming languages cannot allow for such ambiguities or computer would not know which interpretation to execute

### Errors

- Computers follow our instructions exactly
- If program produces the wrong result, it’s the programmer’s fault
  - unless the user inputs incorrect data
  - then cannot expect program to output correct results: “Garbage in, garbage out” (GIGO)
- **Debugging**: process of finding and correcting errors
  - Unfortunately can be very time consuming!
Errors

- Error at compile time (during translation)
  - you did not follow syntax rules that say how Java elements must be combined to form valid Java statements

- Error at run time (during execution)
  - Source code compiles
    - Syntactically (structurally) correct
  - But program tried something computers cannot do
    - like divide a number by zero.
  - Typically program will crash: halt prematurely

- Logical error
  - Source code compiles
  - Object code runs
  - But program may still produce incorrect results because logic of your program is incorrect
    - Typically hardest problems to find

Questions?