The Java Language

Course Topics

Elements of the Java Platform

- ***The Java Language***

Java Objects, Methods, and Fields

Java's Object Orientation and I/O

Interfaces, Graphical User Interfaces, and Applets

Topics this week:

- Last Week Highlights
- Pieces that make up the Java Platform
- Java Programs and the Java Platform
- Creating a Java application
- Last Week's Homework
- Reserved Words - Primitive data types
- Primitive Numeric Integer Types
- Primitive Floating Point Number Types
- Other Primitive Types
- String and System objects
- Expressions
- Arithmetic Operators
- Assignment Operators
- Methods
- Fill in the blanks
- Reserved Words - Control Flow
- Control Flow - Branching
- Relation Operators

Last Week Highlights

- Course Guidelines
- Website where class materials are available

  http://cs.gmu.edu/~jdoughty/cs161/class_material.html
- Java Development Kit: JDK 1.1.x, JDK 1.2.x
- Compiling Java source
  Using the *javac* **compiler**
- Running a Java application
  Using the *java* **interpreter** to execute Java **bytecode**
- Introduction to methods
- Kinds of errors
  - User errors
  - Compiler errors
  - Runtime errors

Pieces that make up the Java Platform

Java Language

The syntax and constructs for writing Java code

The Java *compiler* is the tool that understands the language and generates Java *bytecode instructions* from source files.

Java Application Programming Interface (API)

The collection of *packages* of standard Java classes that you use as building blocks.

The API documentation describes the standard Java packages.

Java Virtual Machine (JVM)

The environment in which Java code runs: the program that executes Java code.

The Java *interpreter* or *Just In Time (JIT) compiler* is part of the JVM

Question: *What's an interpreter?*

Question: *What's a JIT?*

Java Programs and the Java Platform

A picture of the previous slide's words:

Java Program (application, applet, servlet) bytecode
Java Application Programming Interface (standard packages; also bytecode)
Java Virtual Machine (executes bytecode)
Hardware, e.g., your PC, OSF1, workstations, rings, ...

- The Java Virtual Machine (JVM, a program) runs on hardware of some sort or another;
- A Java application (like your program) runs in the JVM
- Applications use the API. The API defines standard object types that are common to many programs needs.

Creating a Java application

- Decide what it is you want to accomplish
- Break that down into smaller and smaller pieces
  (Like writing an outline.)

  In *object oriented programming* it helps to think first in terms of what are the nouns that make up your problem: these become your class names.

- Write Java code in one or more source files that fulfills the actions required.

  Think of what the verbs are that you want your nouns to do: these become your methods.

Last Week's Homework

Write the class Hello so that it prints on four lines just:

- Your name
- Your GMU ID
- Your email address
- The date and time

```java
/** Hello.java
 ** Jonathan Doughty - assignment 1
 ** A Java class to produce output that looks like:
 ** Your name
 ** Your GMU ID
 ** Your email address
 ** The date and time
 **/

// Will use a Date object from the java.util package.
import java.util.Date;

public class Hello {

  public static void main(String[] args) {

    //  Make a Hello object ...

    Hello anObject = new Hello();

    // Initialize it

    anObject.initialize();

    // ... and ask it to to identify itself

    System.out.println( anObject.toString() );

    Hello.waitForUser();
  }

  // The rest of this relates to Hello "objects"

  // each Hello object will remember who you tell it is using
  // these "instance variables" named "name", "GMUid", "email"

  String name;
  String GMUid;
  String email;
  Date now;

  public void initialize() {

    // Assign its name from the command line argument

    name = "Jonathan Doughty";
    GMUid = "123-45-6789";
    email = "jdoughty@cs.gmu.edu";
    now = new Date();
  }

  // This will allow Hello objects to identify themselves asked.

  public String toString() {
    String result;
    result = "Name:" + name +
             "\nGMU Id:" + GMUid +
             "\nEmail:" + email +
             "\nDate:" + now;
    return result;
  }

  // The following is only needed for running the program from
  // within the Javaedit application on Windows
  public static void waitForUser() {
    try {
      byte[] line = new byte[80];
      System.out.println("press enter key to end");
      System.in.read(line);
    }
    catch (java.io.IOException e) {
      // ignored
    }

  }
}
```

Produces:

Name:Jonathan Doughty GMU Id:123-45-6789 Email:jdoughty@cs.gmu.edu
Date:Sat Oct 30 21:31:57 EDT 1999 press enter key to end

Reserved Words - Primitive data types

primitives	**control flow**	**qualifier**	**other**	**unused**
boolean	break	abstract	**false**	byvalue
byte	case	class	instanceof	cast
char	catch	extends	new	const
double	continue	final	null	future
float	default	import	super	generic
int	do	implements	this	goto
long	else	interface	**true**	inner
short	finally	native		operator
	for	package		outer
	if	private		rest
	return	protected		var
	switch	public		
	throw	static		
	try	synchronized		
	while	throws		
		transient		
		void		
		volatile		

Primitive Numeric Integer Types

Memory

Type								
byte								
short								
int								
long								

Details

Type	Size	Min value	Max value
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807

Example

```
int answer = 42;
long aLongValue = 5875685732167641L;  // note 'L'
short sval = 32766;
byte bval = -101;
```

Primitive Floating Point Number Types

Memory

Type								
float								
double								

Details

Type	Size
float	32 bits
double	64 bits

Example

```
float fval = 1.0f;
double dval = 1.0;
double pi = 3.14;
double betterPi = Math.PI;  // from java.lang.Math
```

Other Primitive Types

Type	Size	Values
boolean	1 bit	true or false
char	16 bit	Unicode characters

Example

```
boolean tOrF = true;
char c = 'a';
```

boolean

Java has special reserved words related to boolean types

true and ***false*** are reserved words in Java

Used to save the result of a *relation* or *conditional* operators (discussed later.)

char

A *char* variable (field) can hold a *single* **Unicode** character

Any of over 34,000 distinct characters from the written languages of the Americas, Europe, the Middle East, Africa, India, Asia, and Pacifica.

Java supports using a set of *escape sequences* to represent frequently used special character values:

Escape sequence	Character value
\b	Backspace
\t	Tab character
\n	Newline
\r	Carriage return
\"	Double quote
\’	Single quote
\\	Backslash
\uxxx	The Unicode character xxxx

String and System objects

String - not a primitive, an Object

- A special Object with some built-in language support.
- Over 20 methods (many more if you count overloading), e.g.,
  - equals(String);
  - equalsIgnoreCase(String);
  - indexOf(int);
  - indexOf(String);
  - lastIndexOf(String);
  - substring(int);
- Quoted strings are made into String objects automatically:

```
"This is a String"
```

- Strings *overload* the + operator to mean "concatenate" (and create a new String)
- String objects are *immutable* once created they never change; you can only create a new string

  Tip: If you need to change a String's contents a lot, use a StringBuffer object.

```
public String toString() {
  StringBuffer result = new StringBuffer();
  result.append( "Name:" );
  result.append( name );
  result.append( "\nGMU Id:" );
  result.append( GMUid );
  result.append( "\nemail:" );
```

```
    result.append( email );
    result.append( "\ndate:" );
    result.append( now );
    return result.toString();
  }
```

is better than

```
  public String toString() {
    String result;
    result = "Name:" + name +
             "\nGMU Id:" + GMUid +
             "\nemail:" + email +
             "\ndate:" + now;
    return result;
  }
```

The latter creates 8 String objects as a result of all the + operators.

System

A JDK provided class that contains useful class fields and methods.

- in - the "standard" input stream
- out - the "standard" output stream
- err - the "standard" error output stream
- Over 18 methods itself, over 32 counting those of *in*, *out*, and *err*

String and System are part of the Java API's java.lang package

Browsable at http://cs.gmu.edu/~jdoughty/cs161/jdk1.2.2/docs/api/

StringComparisons.java

```java
/** This StringComparisons class contains examples of comparing Java
 ** String values.
 **
 ** @author Jonathan Doughty
 **/

public class StringComparisonsInst {

  public static void main(String args[]) {

    if (args.length != 2) {
      System.out.println(
        "usage: java StringComparisons \"string1\" \"string2\"");
    }
    else {
      StringComparisonsInst instance = new StringComparisonsInst();

      instance.compare( args[0], args[1] );
    }
  }

  public void compare(String string1, String string2) {

    // Compare input arguments in several ways

    if (string1.equals(string2))

      System.out.println("Strings are the same");

    else if (string1.equalsIgnoreCase(string2))

      System.out.println("Strings are the same, ignoring case");

    else if (string1.startsWith(string2))

      System.out.println( string2 + " matches the beginning of " +
                          string1);

    else if ( string1.equalsIgnoreCase("Java") ||
              string2.equalsIgnoreCase("Java"))

      System.out.println( "one of the arguments matches 'Java'");

    else if (string1.endsWith(string2))

      System.out.println( string2 + " matches the end of " + string1);

    else {
      int compareValue = string1.compareTo(string2);
      if ( compareValue > 0)

        System.out.println( string1 + " is 'greater' than  " +
                            string2);

      else

        System.out.println( string1 + " is 'less' than  " +
                            string2);
    }
  }
}
```

Expressions

*Definition: An **expression** is a series of variables, operators, and method calls (constructed according to the syntax of the language) that **evaluates** to a single value.*

from: *The Java Tutorial*, Campione & Walrath, 1998

Example

```
// Examples of declaration and expression evaluation

int i;
 ...
i = 1;

int j;
 ...
j = i + 1;

CreditCard card = myCard.dependentsCard();

float whatIOwe = whatIOwedLastMonth +
                 visaCard.balance() +
                 mastercardCard.balance() +
                 discoverCard.balance();
```

The code is always of the form:

```
destination = expression_to_be_evaluated ;
```

First time programmers often write

```
expression to be evaluated = destination ;
```

It doesn’t work that way.

Arithmetic Operators

Operator	Description
op1 + op2	Adds op1 to op2
op1 - op2	Subtracts op2 from op1
op1 * op2	Multiplies op1 by op2
op1 / op2	Divides op1 by op2
op1 % op2	Returns remainder of dividing op1 by op2
op++	Post increment of op (pre-increment: ++op)
op--	Post decrement op (pre-decrement: --op)

Assignment Operators

And shortcut assignment operators (Side effect operators)

d = e	d is assigned the value of e
d += e	d is assigned the value of d + e
d -= e	d is assigned the value of d - e
d *= e	d is assigned the value of d * e
d /= e	d is assigned the value of d / e
d %= e	d is assigned the value of d % e
For completeness...	
d &= e	d is assigned the value of d & e
d |= e	d is assigned the value of d | e
d ^= e	d is assigned the value of d ^ e
d <<= e	d is assigned the value of d << e
d >>= e	d is assigned the value of d >> e
d >>>= e	d is assigned the value of d >>> e

Methods

Called functions, subroutines, procedures among other names in other programming languages

"A method is a named sequence of instructions ..."

An action that you want an object to perform one or more times

```
public class Something {

   // instance variable definitions, if any

   // method definitions that pertain to actions that Something can be
   // asked to do

   public ReturnDataType action name {
      ...
   }

}
```

Fill in the blanks

```java
/** Examples of operations on primitive Java data types
 **/

public class Example1 {

  public static void main (String [] args ) {

    Example1 instance = new Example1();

    // Assumes an int value is given on the command line

    int argval = Integer.parseInt(args[0]);

    System.out.println("squared:" + instance.square(argval));
    System.out.println("area:" + instance.areaOfCircle(argval));

  }

   // instance variable definitions, if any

   // method definitions that pertain to actions that Something can be
   // asked to do


  public int square(int arg) {
    int result;

    // Add code here to compute arg * arg and save the result in
    // the local variable result

    return result;
  }

  public double areaOfCircle( int radius ) {

    double result;

    // Add code here to compute the area if a circle whose radius
    // is given: area = Pi * (radius squared)
    // Hint: Java defines a constant Math.PI

    return result;
  }

}
```

Reserved Words - Control Flow

primitives	***control flow***	**qualifier**	**other**	**unused**
boolean	**break**	abstract	false	byvalue
byte	**case**	class	instanceof	cast
char	**catch**	extends	new	const
double	**continue**	final	null	future
float	**default**	import	super	generic
int	**do**	implements	this	goto
long	**else**	interface	true	inner
short	finally	native		operator
	for	package		outer
	if	private		rest
	return	protected		var
	switch	public		
	throw	static		
	try	synchronized		
	while	throws		
		transient		
		void		
		volatile		

Control Flow - Branching

"if" statements

Execute some statements when a condition is satisfied (or not)

```
if (expression that evaluates to a boolean result) {
   // The enclosed statements are performed if true
...
}

if (expression that evaluates to a boolean result) {
   // The enclosed statements are performed if true
  ...
}
else {
   // The enclosed statements are performed if true
  ...
}
```

"switch" statements

Select one of a group (or several groups) of statements to execute

```
switch (expression that results in an integer value) {
case one_possible_value:
  ...
  break;
case another_possible_value:
  ...
  break;
...
```

```
default:
  ...
  break;
}
```

Question: *Why do I say "(or several groups)"*

IfDemo.java

```
/** An example of "if" ... "else" ... branching control flow
 ** @author Jonathan Doughty
 **/

public class IfDemo {
  public static void main(String args[]) {
    if (args.length == 0) {
      System.out.println("usage: java IfDemo [other arguments]");
    }
    else {
      System.out.println("You gave me " + args.length + " arguments");
    }
  }
}
```

SwitchDemo.java

```
/** An example of the "switch" branching control flow statement
 ** @author Jonathan Doughty
 **/

public class SwitchDemo {
  public static void main(String args[]) {
    switch (args.length) {

    case 0:
      System.out.println("usage: java IfDemo [other arguments]");
      break;

    case 1:
      System.out.println("I didn't come here for an argument");
      break;

    default:
      System.out.println("You gave me more then 1 argument");
      break;
    }
  }
}
```

Relation Operators

Return boolean values - true or false

Operator	Description
(op1 > op2)	Is op1 greater than op2?
(op1 >= op2)	Is op1 greater than or equal to op2?
(op1 < op2)	Is op1 less than op2?
(op1 <= op2)	Is op1 less than or equal to op2?
(op1 == op2)	Is op1 equal to op2?
(op1 != op2)	Is op1 not equal to op2?

```
if (args.length == 0) { ...
```

Conditional Operators

evaluate booleans - return boolean

Operator	Description
(op1 && op2)	Are op1 and op2 **both** true (op2 not evaluated if op1 true)
(op1 || op2)	Is op1 or op2 true (op2 evaluated only if op1 false)
(! op)	The opposite of op
boolean ? op1 : op2	expression has value op1 if boolean true, else op2

Introducing Objects

- Putting all your code in *main* is a bad idea
- Try to write small (< 50 lines) methods that do what their name implies: actions (verbs) that you want to be performed.
    - Any time you find yourself writing the same, or almost the same, lines of code (even if it is only a few lines), think about making that code a method instead.
    - Any time you can give a name for a sequence of lines of code, think about making that sequence into a method.
- Think of *objects*: things (nouns) that can perform various actions (verbs) that will be the object's methods.
- Try to think of collections of things each of which is independent of other things in the collection.

A Student Object

```
/** Encapsulate information relating to a single student.
 ** @author: Jonathan Doughty
 **/

public class Student1 {

  public static void main(String args[]) {

    Student1 aStudent = new Student1();

    // When the above line is executed, a single, anonymous
    // Student1 object is created, briefly.
  }
}
```

A Student Object with associated data

```java
/** Encapsulate information relating to a single student.
 ** @author: Jonathan Doughty
 **/

public class Student2 {

  public static void main(String args[]) {

    Student2 aStudent = new Student2();

    aStudent.setName("Jonathan Doughty");
    aStudent.setId("123-45-6789");

    // When the preceding lines are executed, a Student2 object is
    // created and its two fields are set.

    // What does the following do?  What does the result look like?
    System.out.println(aStudent);

    Student2.waitForUser();
  }

  // Instance variables (fields) that will be associated with each student

  private String GMU_Id;
  private String name;
  private int    homeworkGrade;

  // An accessor method to set the Student's name field
  public void setName( String studentName ) {
    name = studentName;
  }

  // An accessor method to set the Student's GMU_Id field
  public void setId( String id ) {
    GMU_Id = id;
  }

  // The following is only needed for running the program from
  // within the Javaedit application on Windows
  public static void waitForUser() {
    try {
      byte[] line = new byte[80];
      System.out.println("press enter key to end");
      System.in.read(line);
    }
    catch (java.io.IOException e) {
      // ignored
    }
  }

}
```

Try compiling and running this

A Student Object that identifies itself

```java
/** Encapsulate information relating to a single student.
 ** @author: Jonathan Doughty
 **/

public class Student3 {


  public static void main(String args[]) {
    Student3.tryIt();
    Student3.waitForUser(); // only needed for Windows/Javaedit
  }

  public static void tryIt() {
    Student3 aStudent = new Student3();

    aStudent.setName("Jonathan Doughty");
    aStudent.setId("123-45-6789");
    aStudent.setGrade( 100 );

    // When the preceding lines are executed, a Student3 object is
    // created and its two fields are set.

    // Does the following do something different?  What does the
    // result look like now?
    System.out.println(aStudent);
  }

  // Instance variables (fields) that will be associated with
  // each student

  private String GMU_Id;
  private String name;
  private int    homeworkGrade;

  // An accessor method to set the Student's name field
  public void setName( String studentName ) {
    name = studentName;
  }

  // An accessor method to set the Student's GMU_Id field
  public void setId( String id ) {
    GMU_Id = id;
  }

  // An accessor method to set the Student's homeworkGrade field
  public void setGrade( int grade ) {
    homeworkGrade = grade;
  }

  // Using the toString method to enable an instance of an
  // object to identify itself usefully.
  public String toString() {
    String result = name + ":" + GMU_Id + " grade:" + homeworkGrade;
    return result;
  }

  // The following is only needed for running the program from
  // within the Javaedit application on Windows
  public static void waitForUser() {
    try {
      byte[] line = new byte[80];
      System.out.println("press enter key to end");
      System.in.read(line);
    }
    catch (java.io.IOException e) {
      // ignored
    }
  }
}
```

Constructors

Constructors are the mechanism by which new *instances* of a class are created from the blueprint of the class definition.

The purpose of a constructor is to initialize a new object.

- Constructors look something like method definitions *except*
  - They always have the same name as the class
  - They never return any type of value

- You "call" a constructor using the **new** operator and supplying any needed constructor arguments.

- Every class has, by default, a constructor:

  ```
  public ClassName() {
  }
  ```

  that takes no arguments and does no special initialization.

- If you don't define one, Java will create a default, no-arg constructor.

- If you define any constructor, with or without arguments, Java assumes you know what you are doing and defines *no* default constructor.

- **Recommendation:** If you define *any* constructors that take arguments, always define a "no argument" constructor too.

Question: *Why would you want to define a constructor?*

Example of a Constructor

```java
/** Encapsulate information relating to a single student.
 ** @author: Jonathan Doughty
 **/

public class Student3c {


  public static void main(String args[]) {
    Student3.tryIt();
    Student3c.waitForUser(); // only needed for Windows/Javaedit
  }

  public static void tryIt() {
    Student3c aStudent = new Student3c( "Jonathan Doughty",
                                        "123-45-6789" );
    aStudent.setGrade( 100 );

    // When the preceding lines are executed, a Student3c object is
    // created and its two fields are set.

    // Does the following do something different?  What does the
    // result look like now?
    System.out.println(aStudent);
  }

  // Instance variables (fields) that will be associated with
  // each student

  private String GMU_Id;
  private String name;
  private int    homeworkGrade;

  // Constructors for the class

  private Student3c() { // why do you think this is?
  }

  public Student3c( String name, String id) {
    this.name = name;
    GMU_Id = id;
  }

  // An accessor method to set the Student's name field; not
  // needed any more but left in because a student's name could
  // change.
  public void setName( String studentName ) {
    name = studentName;
  }

  // An accessor method to set the Student's GMU_Id field (probably
  // no longer necessary)
  public void setId( String id ) {
    GMU_Id = id;
  }

  // An accessor method to set the Student's homeworkGrade field
  public void setGrade( int grade ) {
    homeworkGrade = grade;
  }

  // Using the toString method to enable an instance of an
  // object to identify itself usefully.
  public String toString() {
    String result = name + ":" + GMU_Id + " grade:" + homeworkGrade;
    return result;
  }

  // The following is only needed for running the program from
  // within the Javaedit application on Windows
  public static void waitForUser() {
    try {
      byte[] line = new byte[80];
      System.out.println("press enter key to end");
      System.in.read(line);
    }
    catch (java.io.IOException e) {
      // ignored
    }
  }
}
```

Assignment for Next Time

Reading

- Chapter 4 - Classes, Objects, and Methods
- Chapter 5 - Programming with Classes and Methods

Homework

Goal:

- Write a simple Java program from scratch;
- Learn how to write a Java class with fields and methods.

Purpose:

- To start thinking like a programmer to break a problem into smaller pieces.
- To start getting the computer to calculate answers. To start learning to be **exact** when giving the computer instructions.

Instructions

Do the following:

- Write a Java class named Employee. This class should have the following *instance* variables:
  - an ID value
  - the name of a person associated with the Employee

- a field to store a value for the annual salary
- a field to store a reference to another Employee object (for the Employee's boss, eventually)

- Write methods with the following signatures
  - a main method to start things off with the signature:

    ```
    public static void main (String [] args)
    ```

    see below for what main should do.

  - a public constructor for the class that will insure that Employee objects always have a name field with the signature

    ```
    public Employee( String name)
    ```

  - a method to set an Employee object's salary field with the signature

    ```
    public void setSalary( double value )
    ```

  - a method to get an Employee object's current salary field value with the signature

    ```
    public double getSalary( )
    ```

  - a method to set an Employee object's field referencing another Employee object with the signature

    ```
    public void setBoss ( Employee boss )
    ```

  - a method to return a String identifying the field contents of an Employee with the signature

    ```
    public String toString()
    ```

- The main method for this class should:
  1. create a single instance of the Employee class,

2. assign a name and a salary value (any value), using the constructor and the setSalaray() methods

3. Access the current salary value (using the getSalary() method) and increase the Employee's salary by 10% using the setSalary method again.

4. then have the Employee object identify itself and it's current salary value.

Hint: Do the reading assignment *before* doing the homework.

In following assignments you are going to create several Employee objects. This assignment's goal is to create an object that will store information about a single employee. Next week we'll explore ways to create a number of Employee objects.

You should *not* do everything in the main method, do only what is asked for in the main method. Look at the in-class examples as a model for how the Employee class should be created.

Do only what is asked above. Do not add any extra code to prompt for input or otherwise add capabilities not asked for, simply use your name and whatever salary value you choose in the program. Be sure to create methods with the signatures requested. Output similar to the following is all the Employee class needs to produce:

```
Joe Student salary: 55000
```

Hint: To start, create the outline of your class definition by defining empty methods with the signatures listed above. Compile it to be sure the outline is in the right form. Then go back and start filling in the details of each method. Compile often so you can correct errors easily.

Hand in listings of your source file, and the usual transcript of its execution.