Welcome to: **Introduction to Java**

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies

Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java Fundamentals

- OOP vs Procedural
- Java Development Tools
- Java Program Structure
- Java File Structure
- Comments
- Types of Java Programs
- Interpreted vs Compiled
- Compiling and Running
- Java Naming Conventions

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

OOP vs Procedural

- Different paradigm
- Identify objects and behaviors
- Attributes and behavior encapsulated in classes
- Java is based on classes

Copyright © 2007 WolfWare, Inc. All rights reserved.

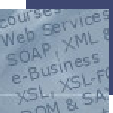
Object oriented programming is a different methodology. It is not a panacea. It is not possible to solve problems using OOP that couldn't be solved with another methodology, but it may be easier.

OOP has also weathered the test of time well. It is not a new fad. Over time it has become a favored approach to large software systems.

In Procedural programming, we broke down the problem into a sequence of tasks. In OOP, we break it down into the objects that exist in the problem statement. For these objects, we then identify their attributes and behavior. The attributes are the features of the objects and the behaviors are the tasks we want the objects to be able to perform. The attributes are defined much like other variables and the behaviors are implemented as methods (or functions).

Java is a pure OO language in the sense that all of the code in Java must exist in classes. That's not to say that you can't write procedural style code in Java. In fact, until we learn how to build classes later on, we will be doing just that. However, it is different than languages that only partially support the features necessary for OOP, or languages that give the programmer the option of procedural or OO programming (like C++).

Copyright © 2007 WolfWare, Inc. All rights reserved.



Java Development Tools

- JDK - now part of the Java SDK
  - version
  - J2EE, J2SE, J2ME
- Forte
- Other IDE's
  - Visual Café
  - JBuilder
  - Visual Age

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 5

Sun provides 2 types of Java development tools. The JDK is a simple command line oriented compiler and virtual machine. With this, the user creates class files in any editor and compiles and runs them from the command line. When using the JDK, you should take note of which version of the JDK you have. This is free for download.

There are several different types of development tools. J2SE encompasses the SDK. J2EE is the enterprise version, and J2ME is the micro version.

Sun also provides an integrated development environment called Forte, which is also free for download.

Other vendors provide integrated development environments for Java. These include Jbuilder (Inprise), Visual Age for Java (IBM), and Visual Café (Symantec).

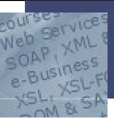
Speaking of versions,

Java 1.0 was first released in 95/96. It was a very basic version.

Java 1.1 quickly replaced 1.0 with major enhancements including a new event handling model.

Java 1.2 added the Java Foundation Classes, including the Swing components for UI development. It also has major enhancements to the Collection classes.

Copyright © 2007 WolfWare, Inc. All rights reserved.



Java Program Structure

- One or more source files
- Each file typically contains one class
- Each class can have a main method
- For applications, at least one class must have a main method
- Applets use init, start, stop, and/or paint

Copyright © 2007 WolfWare, Inc. All rights reserved.

A program often is comprised of many classes, and hence many files.

There are special circumstances which allow more than one class per file. The actual rule is that there can be only one top-level public class per file. This will be covered more later in the course.

For applications, there must be at least one main method in one of the classes, however, each class is allowed to have a main method.

Applets do not need a main method, although they can have one. Instead, they use other methods that are called by the browser in which the applet is run.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java File Structure

- A file typically has only one class
- The file can also contain a line that specifies the package to which the class belongs
- The file can also contain import statements that refer to other packages or classes
- The order must be package, import, class
- The main method is in the class definition

Copyright © 2007 WolfWare, Inc. All rights reserved.

For now we'll focus on files that only have one class. In its simplest form, it contains only a class definition. Classes are organized into related or cooperating groups called packages. If you want to specify which package the class belongs to, use a package statement which must be the first real line of code in a Java file if present. Since classes often make use of other classes, import statements are also helpful. We'll learn more about these later. For now, suffice it to say that the import statement(s) come after the package statement, and before the class definition. So a file might look like:

```
package myClasses;
import java.util.*;
public class MyProgram {
  ….
   public static void main(String [] args) {
       //This can be the entry point of our program
      ….
   }
}
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Comment Formats

- Single line:
  - int count;   // number of students in class
- Multi line:
  - /* comment out as many
  -     lines as necessary. */
- Javadoc:
  - /** special javadoc comment */

Copyright © 2007 WolfWare, Inc. All rights reserved.

The single line comment will cause the compiler to ignore everything that follows it on the same line.

The multi line comment will cause the compiler to ignore everything between the /* and the next */.

The javadoc comments allow us to generate excellent documentation to go with our programs, and to maintain the documentation in the same file as our programs. We'll see this more later.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Types of Java Programs

- Applications
  - stand-alone programs
- Applets
  - require a browser to run
- Servlets
  - faceless server programs

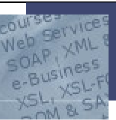
Copyright © 2007 WolfWare, Inc. All rights reserved.

Java applications are stand-alone java programs. It is still possible to have a single piece of software run on various platforms, but each is a stand-alone application. We will be focusing on these most of the time, since they are the easiest way to illustrate the facilities we will be using.

Java applets are small programs that are accessed via a browser. The browser contains a java virtual machine in which the applet can run. These are downloaded to the client when accessed, so they should be small. These are also limited in what they can do for security reasons so that they will not harm the client's computer.

Java servlets are faceless java programs that run on a server. They are typically accessed via a browser, and as part of its execution, a servlet often constructs an HTML page to send back to the client.

Copyright © 2007 WolfWare, Inc. All rights reserved.



Intepreted vs. Compiled

- Compiled languages require a compiler to turn source code into machine code
- In interpreted languages each statement is interpreted as is
- Java is an interpreted language
- However, java source files must be compiled into byte code

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java achieves cross-platform functionality by compiling source code into byte code. This byte code can be run on any machine that has a Java Virtual Machine running on it.

Interpreted programs are typically slower than compiled programs, however, there are techniques for getting around this, like native compilers and JIT compilers.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Compiling and Running

Source Code: MyClass.java

Java Compiler: javac ↓

Byte Code: MyClass.class

JVM: java ↓

Output

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 11

If you are using the command line JDK, you will need 2 commands, javac to compile and java to run.
The javac command takes one or more full file names as arguments. You can also use wild cards.
The java command takes the class name as an argument.

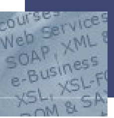
IMPORTANT: The file name must be exactly the same as the class name contained in the file. The file must also have an extension of .java.

For the above, you would execute the following commands:

```
javac MyClass.java
java MyClass
```

If the compilation is successful, it will produce a file, MyClass.class which is the byte code.

Copyright © 2007 WolfWare, Inc. All rights reserved.



Java Naming Conventions

- Class names are mixed case, beginning with uppercase: MyClass
- package names are typically lowercase: java.util
- method and variable names are mixed case, beginning with lowercase: myMethod
- Constants are all uppercase: MAX_NUM

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java programmers are quite consistent with this, so you might want to adopt this habit early on.

However, if you are working on a project that has different naming conventions established, you should of course use those.

The Sun Java site also has recommended coding guidelines, including naming conventions available for download.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Exercise

- Write a simple application to print out your name.
- To do this you will need to use the following in your main method:
  - System.out.println(“Your name here”);

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java
XMaLpha Technologies, LLC.,
Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,
Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Overview of the Basics

- Variables and Data Types
- Operators
- Casting and Converting data
- Conditionals
- Loops
- Arrays
- String
- StringBuffer

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Primitive Data Types

- Integral
  - byte, short, int, long
- Floating Point
  - float, double
- Other
  - boolean
  - char

Copyright © 2007 WolfWare, Inc. All rights reserved.

In Java, as with many other languages, a variable must be defined to be of a specific type. The options for this type are either one of the primitive types, or an object type. We'll look at objects later; for now we'll look at the primitives.

Data Types are fixed-size:

byte	1 byte	
short	2 bytes	
int	4 bytes	
long	8 bytes	
float	4 bytes	
double	8 bytes	
boolean	1 bit	// Value is either true or false
char	2 bytes	

Java is strongly typed and programmers need to be aware of what data types they are working with, and what those types are compatible with.

Integral and floating point types are pretty self-explanatory; we'll look at boolean and character more closely.

There are no unsigned integers in Java.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Boolean

- Strictly true or false
- Cannot be converted to or from other types
- C, C++, and VB programmers beware

Copyright © 2007 WolfWare, Inc. All rights reserved.

Similar to some languages, the type boolean is strictly either true or false (these are Java reserved words). This is not similar to C, C++, and VB, and programmers used to those languages will need to be aware of the distinction.

Logical operations (seen shortly) result in boolean values.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Character

- 16 bit Unicode characters
- Can handle many foreign languages
- Enclose literal values in single quotes: ‘a’
- Special escape characters are valid: ‘\n’
- Can also express as unicode literal: ‘\u0064’

Copyright © 2007 WolfWare, Inc. All rights reserved.

Unlike other languages, Java used 16 bits to represent characters in order to support international alphabets.

When we work with streams later on in the course, you’ll see that this is very transparent to the programmer. The stream classes handle converting these unicode characters back and forth to the local character, which is often ascii for us.

ASCII printable characters have the same binary value in Unicode.

Java also supports UTF-8 character formatting. We’ll see in the Stream section that there are methods for reading and writing UTF characters. These methods convert Unicode characters back and forth to UTF. UTF is an efficient way to input/output Unicode characters that are also ASCII characters, since these only require one byte. Another great benefit of these methods is that you can read strings without having to read character by character. Again, we’ll see this later.

Characters can function as unsigned integer values.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Defining Variables

- Formats:
  - DataType identifier;
  - DataType id1, id2, id3;
  - DataType id1 = const1, id2 = const2;
- Examples:
  - int age;
  - int age, weight;
  - int age =20, index = 0;

Copyright © 2007 WolfWare, Inc. All rights reserved.

The basic rule for declaring variables is that each must be associated with a data type.

The general format for declaring variables is to put the data type first, then the variable identifier.

You can declare more than one variable per statement, as long as all are of the same type. A statement ends at the semicolon.

You can initialize variables at the same time as you declare them.

Variable identifiers:

- must begin with a letter and be a sequence of letters, digits, or underscores
- letters can be foreign letters as well as underscores and $
- no other symbols or spaces
- length is essentially unlimited
- as with everything in Java, variable identifiers are case-sensitive
- cannot use a Java reserved word
- The Character wrapper class provides methods for checking the validity of characters to be used in identifiers.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Literals

- Integral, assumed to be int, unless magnitude requires a long
- Floating Point assumed to be double
- Can use suffixes/prefixes to specify type:
  - L or l for long e.g. size = 233455632L;
  - D or d for double e.g. max = 100.6D;
  - 0x or 0X for hexidecimal e.g. mask = 0x24F3;
  - 0 for octal e.g. mask = 023;

Copyright © 2007 WolfWare, Inc. All rights reserved.

Literals are values that are specified literally in a program. If the literal is integral, it is assumed by the compiler to be an int. If it is a floating point value, it is assumed to be a double. This is something that newcomers to Java must get used to. A common novice error is to define a variable as a float, and then assign a floating point literal (which is a double) to it.

If working with hexadecimal values, these are specified by preceding the hex value with a zero and and x, either in upper or lower case. It is also possible to specify values in octal. For these, use a zero as a prefix.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Operators

- Basic Binary: +, - , *, /, %
- Bitwise Binary: |, &, <<, >>, >>>
- Basic Unary: +, -
- Bitwise Unary: ~, ^
- Unary with Side Effect: ++, --
- Logical Unary: !
- Logical Binary: |, ||, &, &&

Copyright © 2007 WolfWare, Inc. All rights reserved.

Basic Binary: addition, subtraction, multiplication, division, and modulus, respectively. Modulus gives the remainder of a division operation.

Bitwise Binary: Or, And, Left Shift, Right Shift, Unsigned Right Shift, respectively. These are useful primarily with embedded systems, network programming, and some interesting mathematical problems.

Basic Unary: negation and the rather useless positive.

Bitwise Unary: Complement and Exclusive Or

Unary with Side Effect: These have an implicit assignment as well. We will look more at these in a bit.

Logical Unary: Not - negates the boolean value of a conditional.

Logical Binary: Used to form complex conditionals.

We'll look more at the logical operators in the control flow section. Note that logical operations result in boolean values.

VB Programmers: There is no ^ operator for exponentiation. Use the Math class instead.

Be aware of your order of precedence and associativity. Most texts include a table for these. In general, multiplication and division before addition and subtraction. Use parentheses when in doubt of precedence. Most operations occur left to right, but assignments occur right to left.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Operators

- Cast: (type)
- Object: instanceof
- Ternary: ?:
- Assignment: =
- Compound Assignment: +=, -=, *=, /=, %=
- (also works with bitwise operators)

Copyright © 2007 WolfWare, Inc. All rights reserved.

The cast operator will temporarily change the data type of a variable. The variable itself does not change type. A temporary variable is created behind the scenes to hold the new type. We'll do quite a bit with casting in this course.

The instanceof operator is used to test if an object is an instance of a particular class or interface (or array). This may not make sense at this point, but we will use it later in the course.

The ternary operator is a shorthand conditional. We'll see an example of this as well.

The single equal sign is used to assign a value to a variable. This is different than the double equal sign which tests for equality.

The compound assignment operators are shorthand for doing an operation and then assigning the result. The format is: LHSop operator= RHSop

which translates into: LHSop = LHSop operator RHSop

For example: x += 3 is the same as x = x + 3

Copyright © 2007 WolfWare, Inc. All rights reserved.

Comparison Operators

- Equality:
  - ==
  - !=
- Relativity:
  - <
  - >
  - <=
  - >=

Copyright © 2007 WolfWare, Inc. All rights reserved.

The double equal sign is used to test for equality. This is not the same as the single equal sign which is used for assignment. As in any language, be careful of testing floating points for equality or inequality due to roundoff errors.

The != is used to test for inequality.

Relativity: Less than, greater than, less than or equal, and greater than or equal, respectively.

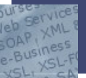
VB Programmers: Note that Java uses different operators for assignment and equality. = is not the same as ==

C and C++ programmers: Since conditions must be boolean, typos such as

```
if (x = 0)
```

are no longer a problem, since this triggers a compiler error.

Copyright © 2007 WolfWare, Inc. All rights reserved.



Unary Side Effect Operators

- ++ and --
- Shorthand for increment and decrement
- Have a built-in implicit assignment
- 2 forms: postfix and prefix
- Can be a bit tricky at first

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 12

These are the same as adding or subtracting one to/from a variable.

In their simplest form, they are not tricky, but when used as part of a bigger statement, it is necessary to understand how these work.

Postfix or prefix does not matter if this is the only thing in the statement. This does matter if this operation is part of a bigger statement.

So, x++; is functionally the same as ++x;

but y = x++; is not the same as y = ++x;

Three simple code fragments illustrate this point:

a = 0;	a = 0;	a = 0;
b = a + 1;	b = a++;	b = ++a;

After each of these executes the values for a and b are as follows:

a = 0	a = 1	a = 1
b = 1	b = 0	b = 1

Copyright © 2007 WolfWare, Inc. All rights reserved.

Integral Arithmetic

- As with some other languages, be aware of integral division as remainder will be lost
- Cast operator can help in some situations
- Programmer must be aware of what is happening

Copyright © 2007 WolfWare, Inc. All rights reserved.

Dividing 2 integers will result in the loss of the remainder if there is one. There is no indication that this has occurred. There are several ways to remedy the situation, one of which is the cast operator.

Examples: (x, y, and z are ints)

```
x = 7;
z = 2;
y = x / 2;          // y's value will be 3
```

Just defining y as a float or double will not matter since the remainder is lost in the division and before the assignment. However, if y is defined as a float, the following will work:

```
y = x / 2.0;    // This will force floating point arithmetic, and y will be 3.5
```

You can cast to do the same thing:

```
y = (float)x / 2;
```

Casting is particularly useful, when specifying a literal is not possible:

```
y = x / (float) z;
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Conversion and Casting

- Casting is when the programmer explicitly casts a variable using (type)
- Conversion is done implicitly by the compiler
- Since Java is strongly typed, you may have to cast more often than you expect
- Can occur in assignments, arithmetic, and method calls

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 14

We saw an example of casting on the previous page. Conversion happens implicitly, such as:

```
double d = 3.2;
int j = 3;
d = j;          // No cast is required
```

This is implicit because there is not a risk of losing data. However, since Java is strongly typed, you would need to cast to do the reverse:

```
j = d;          // Not legal - will not compile
```

However, `j = (int) d;`

is legal, but you will lose the fraction. Since there is a potential loss of data, a cast is required.

Actually, conversions can also occur when returning values from a method, but it is considered poor style to not have these match.

Permitted conversions are char -> int and

byte -> short -> int -> long -> float -> double

Assignment Conversions

- With variables:
  - int j = 2;
  - double d = j;
- With literals:
  - double d = 2;

Copyright © 2007 WolfWare, Inc. All rights reserved.

As with other languages, the RHS is evaluated and then assigned to the variable on the LHS.

If the RHS evaluates to a smaller data type than the LHS, an implicit conversion will occur.

If the RHS evaluates to a bigger data type than the LHS, you will need to cast, or you will get a compiler error.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Arithmetic Conversions

- Balancing and Promotion
  - both operands must be the same type
  - if not, the one with the 'smaller' type is promoted (implicit conversion)
  - result is the same type as the 'bigger' type
- Operands of type byte, char, and short are automatically converted to int. The compiler doesn't do math with these types.

Copyright © 2007 WolfWare, Inc. All rights reserved.

As with other languages, operations occur one at a time. For each binary operator, both operands must be of the same type, or one will need to be promoted to match the other. This is an implicit conversion.

Example:

```
int a = 4, b = 5;
double d = 6.2;
float f = 3.1;
…
double total = a * d + f * b;
```

The order of operations is a * d, then f * b, then those intermediate results are added together.

The result of a * d is of type double since a had to be promoted.

The result of f * b is of type float since b had to be promoted.

The result of the addition is double since the intermediate result of f * b had to be promoted.

There is no additional conversion for the assignment, since the RHS evaluates to type double.

NOTE: the smallest operation type is int.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Method Conversions

- When passing values to methods:
  - type must match
  - OR
  - implicit conversion may take place
  - OR
  - may need a cast

Copyright © 2007 WolfWare, Inc. All rights reserved.

This is very similar to assignments.
Example:
m1 is a method that takes an argument of type double: void m1(double x) …
j is an integer.
If you call m1 and pass in j, an implicit conversion will take place.

If instead, j was a double, no conversion would be necessary.

And finally, if x was of type int and j was of type double, you would need to cast j to pass it to m1:

```
SomeObject.m1( (int) j);
```

For now, ignore the SomeObject part; we'll learn about that later.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Optional Exercise

- Try out some of the techniques we've shown in this section, such as casting and converting.
- Suggestion:
  - start out with 5 integers and calculate their average, out to 2 decimal places

Copyright © 2007 WolfWare, Inc. All rights reserved.

If Java is looking a bit odd already, or if you just want to give your programming environment another try, write some simple programs to do some arithmetic.

If you are comfortable with the material so far, you'll probably want to skip this exercise.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Conditional Statements

- Used in:
  - if statements
  - loops
  - ternary operator
- Must evaluate to a boolean value
- Usually need to be encased in parentheses
- Product of relational or instanceof operator

Copyright © 2007 WolfWare, Inc. All rights reserved.

Conditional statements control whether or not a particular line or block of code will be executed.

In Java, they *must* evaluate to a boolean value, either true or false. If you are used to C or C++, this is quite different. However, it also eliminates potentially problematic code.

Most often conditionals are enclosed in parentheses, but the ternary operator doesn't actually require them.

The way to form a conditional statement is to use relational or equality operators such as < or ==, such as (x > 3). You can also use the instanceof operator, such as (obj instance of MyClass). We'll see this more in later sections.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Compound Conditionals

- For AND:
  - && - short circuit logical AND
  - & - logical AND that forces full execution
- For OR:
  - || - short circuit logical OR
  - | - logical OR that forces full execution
- For Exclusive OR: ^
- For NOT: !

Copyright © 2007 WolfWare, Inc. All rights reserved.

There are two forms of the logical OR and AND. The single character version of each will stop execution as soon as it reaches an answer. For example, true OR anything is still true, so if the first operand evaluates to true, it will never evaluate the second. This is usually the behavior you want. First of all, it is more efficient. Secondly, it allows you to easily verify that a reference or value is valid before using it.

The double character form of the these operators forces both terms to be evaluated. This is useful if the term has some sort of side effect that you need to happen. This is a much less common case.

NOTE: The single form of the AND and OR operators is the same as the bit-wise AND and OR. The difference is in the type of the operands. If the operands are numeric, the operation is bit-wise. If the operands are boolean, the operation is logical.

For AND: Both operands must be true for the result to be true.

For OR: One or both operands can be true to result in true.

For Exclusive OR: One and only one operand can be true to result in true.

The ! toggles the operand's value.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Code Blocks

- Used in:
  - if statements
  - loops
  - method and class bodies
  - initializers
- Syntax:

```
{
    body of block
}
```

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 21

Code blocks are the way to specify that more than one line of code should be treated as a single section of code. So far you've already used these to delineate the body of a class and the body of a method. In this section, we'll use them to mark the body of code you want executed in certain conditions.

The syntax is simple - just curly braces around the code.

Actually, code blocks don't require any of the cases listed. You can actually just surround random code with a code block and it will compile, although it is not considered good style. The reason you can do this is that you might want to limit the scope of a variable to a small code block.

Any variable defined inside a code block is limited in visibility to the block in which it is defined.

NOTE: There cannot be multiple variables with the same name in scope at the same time. This is different than C++.

if Statement

- Key words: if, else
- Syntax:
- if (some condition)
- execute this statement or code block
- else
- execute this statement or code block

Copyright © 2007 WolfWare, Inc. All rights reserved.

In this form, only one line of code is executed in either condition. If you want more than one line of code, you must use code blocks.

The else clause is optional.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- if (condition 1)
-       execute this statement or block
- else if (condition 2)
-       execute this statement or block
- …
- else
-       execute this statement or block

Copyright © 2007 WolfWare, Inc. All rights reserved.

This can go on indefinitely.

It is important to note that any else statement goes with the closest if, which may not be what you expect if the indentation is misleading:

```
if (condition 1)
   line a
else if (condition 2)
   if (condition 3)
      line b
else
   line c
```

Here, “else line c” goes with the “if condition 3” clause. Of course, indentation and white space are meaningless to the compiler. This can be fixed by putting a code block around lines 4 and 5.

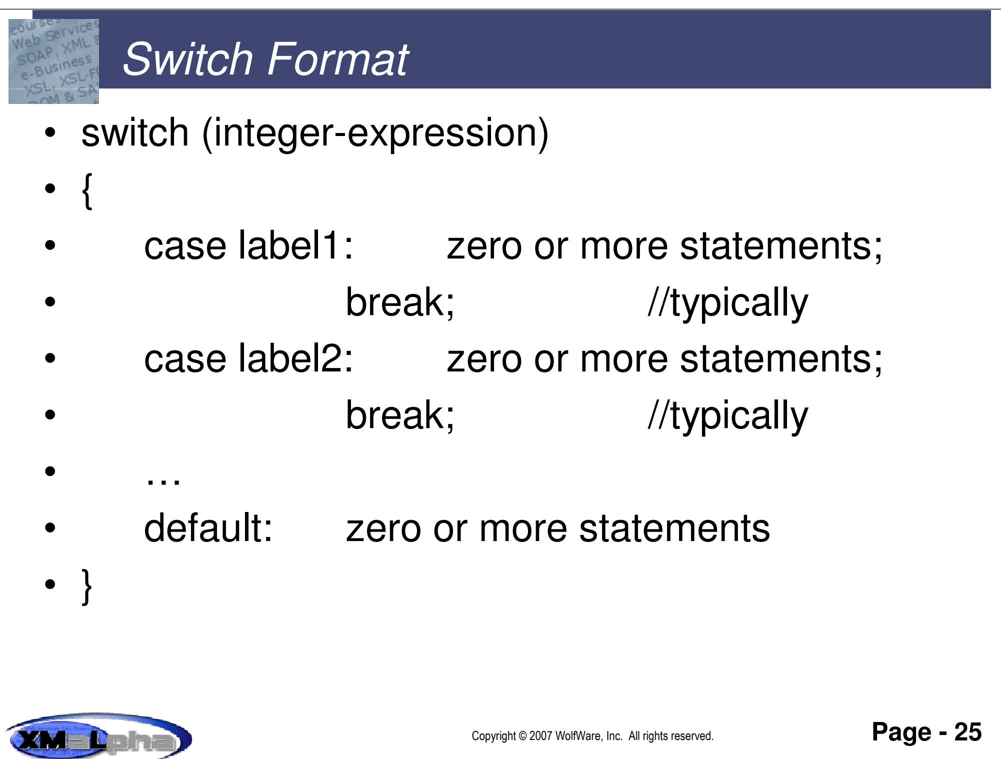
Copyright © 2007 WolfWare, Inc. All rights reserved.

Switch-Case Statement

- An alternative for some nested if-else statements
- Very limited in its application
- Actually a variant of a limited goto statement
- Key words: switch, case, default
- Requires a code block body

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.



The integer expression must be of type byte, char, short, or int.

The case labels must be unique, and the values of the labels must be constants, either literals or variable constants, because it must be determined at compile time.

The default label is optional, but recommended. It is executed if none of the cases match.

Since the *case XXX*: syntax is really a label, this functions somewhat like a goto. Therefore, when a case is matched, execution picks up at the matching label, and continues from there. Therefore, although the breaks are optional, they are required to stop execution from continuing on into the next case section. Of course, you may want it to continue on.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Conditional Operator

- Unusual looking shortcut for a simple conditional statement
- Often embedded as part of an assignment statement
- Format:
  - condition ? then-clause : else-clause;

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 26

This operator should be familiar to C and C++ programmers, and probably looks odd to others.

As with all conditions, this one must be boolean.

Although it can legally be used many ways, the most common is to have it embedded within an assignment. You've all probably had to write code like:

```
if (x == true)
  y = 1;
else
  y = 0;
```

This can be abbreviate as:

```
y = (x == true) ? 1 : 0;
```

Or, since x itself is a boolean, even more concise:

```
y = x? 1 : 0;
```

The key to recognizing this when reading code is the question mark. It is the only time it is legal Java syntax.

Subexpressions (in this case 1 and 0) must be compatible to the LHS operand (in this case y). Here, all are integers.

Oddly, the parentheses are not required around conditions with this operator.

Copyright © 2007 WolfWare, Inc. All rights reserved.

While Loop

- Simplest loop
- Body executes zero or more times
- Syntax:
- while (boolean condition)
- single statement or code block
- Key word: while

Copyright © 2007 WolfWare, Inc. All rights reserved.

While loops are often useful when you know under what condition to execute the body, but you don't know how many times you will execute the loop. For example, you are reading lines of input and you don't know how many there are.

It is worth noting that all loops can be converted to and from each other by adding enough extra code to force entry and exit at the appropriate times. However, this can result in some ugly code.

Example:

```
number = 0;                         // need to make sure condition is true
while ((number < 1) || (number > 10))
{
   System.out.println("Enter a number between 1 and 10");
   // read input from user here….
}
```

Be careful not to put a semicolon after the condition. It is syntactically legal, but terminates the loop.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Do-While Loop

- Condition checked at end of loop
- Body executes one or more times
- Syntax:
- do
- single statement or code block
- while (boolean condition);
- Key words: do, while

Copyright © 2007 WolfWare, Inc. All rights reserved.

Do loops are useful when you know you need to execute the loop at least once, but don't know how many times you will execute it. Asking for user input and continuing to do so until the input is valid is an example.

The previous example could be done in a do loop without having to force the condition to be true the first time:

```
do
{
   System.out.println("Enter a number between 1 and 10");
   // read input from user here….
} while ((number < 1) || (number > 10));
```

This is the one time you need the semicolon after the condition, since you actually do want to terminate the loop there.

Copyright © 2007 WolfWare, Inc. All rights reserved.

For Loop

- Slightly more complex loop
- Easy to specify exactly how many times to execute loop
- Syntax:
- for (initializations; condition; updates)
-     single statement or code block
- Key word: for

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 29

For loops are useful when you know how many times you want to execute a loop, or when you have updates to be done each time through the loop. Again, recall that it is often a toss-up when choosing which loop to use. In fact, the previous loop could be done with a for loop:

```
for (number = 0; number <1 || number > 10; )    // no update section
{
   System.out.println("Enter a number between 1 and 10");
   // read input from user here….
}
```

This is generally not considered the best choice of loop for this problem, however. For loops are better suited for something like:

```
for (j = 0; j < 10; j++)
   System.out.println("I love Java!");
```

Be careful not to put a semicolon after the (…). It is syntactically legal, but it terminates the loop.

Copyright © 2007 WolfWare, Inc. All rights reserved.

For Loop

- All sections are optional
- The initialization and update sections can have more than one part
- Any variables defined in the initialization section or the body of the loop are limited in scope to the loop

Copyright © 2007 WolfWare, Inc. All rights reserved.

The following is syntactically legal:

```
for (; ; )
    // this line would execute forever…
```

Also legal:

```
for (x = 0, y = 1; x<10;  x+=2, y+=3)
    // some body here
```

Notice the commas between the parts of the initializations and updates.

```
for (int x = 0; x < 10; x ++)
{
   int y = x *2;
  // other stuff here
}
```

NOTE: Here, x and y are only valid within the body of the for block. This is different than C++ for x.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Break and Continue

- Alter flow of control of loops
- Not recommended (similar to goto)
- Often used to patch poor designs
- Continue causes the current iteration of the loop to be abandoned
- Break causes the loop to be exited
- Both can be labeled as well

Copyright © 2007 WolfWare, Inc. All rights reserved.

The author feels strongly enough that these are not good programming practice, that we will not dwell on them. They are never necessary, and can always be replaced by reworking the conditions on the loops.

Note that continue and break are key words.

Continue:

In while and do loops, the condition is checked right after the continue statement.

In for loops, the update section is executed right after the continue statement, then the condition is checked.

Break:

The immediate loop is exited.

Each can be used in conjunction with labels to specify which loop to continue with or break out of if you have nested loops.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Arrays

- Group of
  - primitives
  - objects
  - other arrays
- Ordered
- All items in array must be the same type
- Work differently than C and C++

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 32

Items in an array must be of the same type, although we will see that this is quite a loose requirement with objects.

As with Cand C++, multi-dimensional arrays are really one-dimensional arrays where each element is another array.

NOTE: Arrays are considered first class objects in Java. However, it is sometimes easier to think of arrays are neither primitives nor objects. They are groups of these things. Hence an array variable is not compatible with an object or primitive variable, but an element within the array may be. They are also syntactically and functionally unlike any other object in Java since they are implemented by the compiler and not a class in the Java library.

NOTE: There are classes in the Java libraries for arrays as well. We'll look at these later.

Array Definition

- 2 steps
  - define the array variable
  - construct the array
  - these can be combined into one statement
- Example:
- int [] values;   // OR: int values[];
- values = new int [900];
- OR: int []values = new int [900];

Copyright © 2007 WolfWare, Inc. All rights reserved.

Note that no dimension is specified when defining the array variable. Arrays are dimensioned at runtime, so a variable can be used to set the dimension:

```
int []lockers = new boolean[size];
```

where size is an integer variable.

You must do both steps before trying to use the array. The first step just gives you a variable that is allowed to refer to an array. The second step actually gives you the array. Whenever you see the keyword new, you are asking for memory dynamically.

Array elements have default values:

0 for numerical arrays

false for boolean arrays

null for object arrays

Copyright © 2007 WolfWare, Inc. All rights reserved.

Initializing Arrays

- It is possible to initialize arrays when you define them
- Useful if you know the initial values at definition time
- Syntax:
-     int [] values = {10, 20, 30, 40};
- This replaces the construction step

Copyright © 2007 WolfWare, Inc. All rights reserved.

Although it is not often that we know what the values of an array are going to be when we define it, sometimes it is useful to do so. Since we are specifying a specific number of elements for the array, the compiler will count them and dimension the array appropriately.

It is probably more common to initialize the array in a loop:

```
for (int j = 0; j < values.length; j++)
   values[j] = (j + 1) * 10;
```

We'll see why this loop works in a bit.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Array Length

- It is always possible to find the size or length of an array:
  - int size = values.length;   // here, size is 4
- length is not a method, but can be thought of as an attribute
- NOTE: there are no parentheses after length (it's an attribute)

Copyright © 2007 WolfWare, Inc. All rights reserved.

It may not make sense at this point why we make a special note of the lack of parentheses. This will become more evident when we learn about Strings. Parentheses are also a way to distinguish between methods and attributes for objects. We'll learn more about this later.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Array Indices

- Array elements are always numbered starting from 0
- The valid index of an array is always in the range 0..(arrayName.length - 1)
- A variable used to index into an array must be of type int
- Index is checked for validity at runtime

Copyright © 2007 WolfWare, Inc. All rights reserved.

If you try to access outside the bounds of an array, you will get an ArrayIndexOutOfBounds runtime exception. We'll learn more about exceptions later.

XMaLpha Technologies
"Using XML technology to turn structured data into intelligent business knowledge"TM
Website: http://XMaLpha.com - E-Mail: Courses@XMaLpha.com
Copyright © 2007 WolfWare, Inc. All rights reserved.

Array References

- The name of an array is just a reference to a dynamically allocated piece of memory
- This reference can be reused to refer to a different piece of memory
- int values = new int[22];
- Assigning references does not make a copy of the array, instead both references refer to the same array

Copyright © 2007 WolfWare, Inc. All rights reserved.

In the code in the second bullet, if there are no other references to the original values array, it will be lost (and available to the garbage collector).

Assigning array references does not make a copy of the array:

```
int [] nickname = values;
```

Now there are 2 references that refer to the same array in memory.

If you are trying to make a copy of an array, use the System.arraycopy() method.

Copyright © 2007 WolfWare, Inc. All rights reserved.

2D Arrays

- Really a 1D array of 1D arrays
- Use 2 sets of [] in the definition and construction:
- char [][] board = new char [3][3];
- Can be initialized instead:
- char [][] board ={ { ‘A’, ‘B’, ‘C’},
- { ‘D’, ‘E’, ‘F’},
- { ‘G’, ‘H’, ‘I’}}

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 38

This is extendable to more dimensions. Use an additional set of square brackets for each additional dimension.

We can think of a 2D array as a table, where the first dimension is the number of rows and the second is the number of columns.

Note that there is still no dimension in the definition part.

To get the number of rows in a 2D array:

array Name.length

To get the number of columns in any given row:

arrayName[rowIndex].length

Multi-dimensional arrays do not need to be rectangular; each row can have a different length:

```
int [][]values = new int [5];        // specifies a 2D array with 5 rows
values[0] = new int[5];              // specifies 5 elements in the first row
values[1] = new int[3];              //specifies 3 elements in the second row
…
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Optional Exercise

- There are 900 students and lockers at the local school. The first student enters and opens all the lockers. The 2nd student enters and closes every 2nd locker. The 3rd student enters and flips every 3rd locker (opens it if closed, closes it if open). The 4th student does the same to every 4th locker, and so on. Which lockers are open after all 900 students have entered?

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Strings

- Pre-defined class in the Java library
- Sequence of Unicode characters
- Objects
- Immutable
- Can be concatenated to other strings or to primitives using the + operator:
- System.out.println("Age = " + 25);

Copyright © 2007 WolfWare, Inc. All rights reserved.

This is our first deliberate use of objects. String is a class in the Java library. It is in the package java.lang, so we automatically have access to it in our programs.

Strings are a sequence of Unicode characters, although this is pretty transparent to the programmer.

What will probably seem the strangest to programmers in other languages is that Strings are immutable. Once the value is set, it cannot be changed. To manipulate string values, use StringBuffer, which we will see shortly.

Strings can be concatenated (of course, this forms a new, often temporary, String) by using the + operator. The rule is that if one of the operands is a String, the JVM does concatenation. If the operands are both numeric, it does addition.

C and C++ Programmers: Strings are not arrays of characters!!!

Copyright © 2007 WolfWare, Inc. All rights reserved.

Defining Strings

- 2 steps
  - – define the String variable
  - – construct the String or initialize with a literal
  - – these are most often combined in one step
- Literals are enclosed in double quotes
- Example:
- String motto = “Learning Java is fun”;

Copyright © 2007 WolfWare, Inc. All rights reserved.

This example of defining and initializing a String shows a shortcut that is only available to String objects. It is not possible to create any other object in Java this way. The stereotypical way to create an object is:

ClassName objectName = new ClassName(…);

We could actually create String objects this way also:

String motto = new String (“Learning Java is fun”);

However, it is not only more cumbersome, it is also less efficient since it creates an additional temporary String object. What is really happening in the shortcut is that JVM is creating a String object whenever it encounters a String literal. So the long form creates a String object just to create the String object!

NOTE: If creating an object in 2 steps, it is important to note that the first step does not result in an object, merely a variable that is allowed to refer to an object:

```
 ClassName objectName;                    // no object yet
ObjectNname = new ClassName(…);           // now have an object
```

Whenever we create a object, we are actually invoking a constructor for that class. A constructor is a special kind of method responsible for initializing objects. We’ll learn more about these in the classes section. We know we are invoking a constructor when we use the word *new* (except for arrays, which are just different animals altogether).

Copyright © 2007 WolfWare, Inc. All rights reserved.

String Equality

- String variable is just a reference to a location in memory
- == only checks if both references refer to the same spot in memory
- Use equals() or equalsIgnoreCase() to check for equality of Strings

Copyright © 2007 WolfWare, Inc. All rights reserved.

This should sound familiar from the arrays section. The variable is just a reference to a location in memory. It can be reused just like with arrays, but the reference to the first String may be lost, just like with arrays.

Checking for equality is also the same for arrays as well. In fact, it is the same for any type of object. Using == only checks to see if the 2 variables refer to the same spot in memory, not if the contents are equal. Many classes have an equals() method defined to check for equality. Developers often include an equals method for their own classes as well.

C and C++ Programmers: The String class has a method compareTo() that works like strcmp. If you want to compare Strings (for sorting for example), you may want to use this method.

Copyright © 2007 WolfWare, Inc. All rights reserved.

String References

- A String variable (reference) can be reused to refer to a different piece of memory
- String motto = “I love Java”;
- As with arrays, assigning references does not make a copy of the String, instead both references refer to the same String:
- String motto2 = motto;

Copyright © 2007 WolfWare, Inc. All rights reserved.

Since Strings are immutable, the code in the second bullet is somewhat useless.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Java API documentation

- Excellent documentation of the Java class library
- Free for download
- Easy to use
- Example of javadoc documentation

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Java libraries are well documented with javadoc comments. These comments are used by the javadoc tool to create excellent HTML documentation. This API documentation is free for download from the Sun Java site. We'll look at how to use this in class to investigate the String class more closely.

Look at the API documentation for the String class to see what methods are available.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Using Objects to Invoke Methods

- Java programs are composed of classes
- Some of these we implement, others already exist in the library and we make use of them
- Classes allow us to create objects
- From objects, we can invoke methods:
- String motto = “I love Java”;
- int size = motto.length();

Copyright © 2007 WolfWare, Inc. All rights reserved.

Now that we’ve seen the methods available to Strings, let’s look at how to use them. This is also called invoking a method. In OOP, we typically need an object in order to call or invoke a method. Later, we’ll see that we can use a class name instead in certain circumstances.

The format for invoking methods is:

```
objectID.methodName(possible argument list)
```

This is different than accessing attributes for an object:

```
objectID.attribute
```

Note there are no pareatheses. This is what we did to get the length of an array. For strings, however, length is a method, so we need parentheses. There are no arguments for the length method, so the argument list is empty.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Command Line Arguments

- If running from the command line
- Passed to main via an array of Strings
- Example:
  - java SomeProgram arg1 arg2
  - arg1 and arg2 are automatically put into an array of Strings and passed to main

Copyright © 2007 WolfWare, Inc. All rights reserved.

There are several ways to run a Java program. We've seen that applets are run from within a browser. Typically applications have a user interface with windows and such. Sometimes however, it is useful to have programs that are run from the command line. If you've ever used Unix commands, many are essentially C programs. When you supply various arguments to those commands, they also are put into an array and passed to the program.

This is why when you write a main method in Java, you must specify that the parameter list is an array of Strings:

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

args is an arbitrary name that is most commonly used for this, but you may choose any name you like.

For the example in the slide, args[0] would be set to arg1, and args[1] would be set to arg2. To get the number of arguments, just ask for the length of the array: args.length

Copyright © 2007 WolfWare, Inc. All rights reserved.

StringBuffer

- Mutable strings
- Has methods for modifying the contents
- Memory is managed by the class, not the user (the programmer)
- Must use standard construction:
- StringBuffer sb = new StringBuffer(…);

Copyright © 2007 WolfWare, Inc. All rights reserved.

Since Strings are immutable, there is another class for strings that need to be modified, StringBuffer. StringBuffer objects have the ability to hold a string as contents, but also have extra buffer space to hold additional characters. If it runs out of buffer space, it automatically adds more without the programmer needing to do anything.

Strings and StringBuffers are not compatible types. To compare one to the other, you will have to convert one to the other.

StringBuffer is sometimes more efficient to work with, even though your code may seem just as simple, if not easier when working with Strings. This is because working with Strings, especially when concatenating, creates temporary Strings behind the scenes.

Look at the API documentation for StringBuffer to see what constructors and methods are available.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Exercise

- Modify the exercise from the Overview section to define your first name as a StringBuffer variable. Append your last name onto it, with a space in between. Write a loop to print out this name 5 times, each time appending an additional '!' onto it.

- If you have time, change the program to use a different type of loop.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

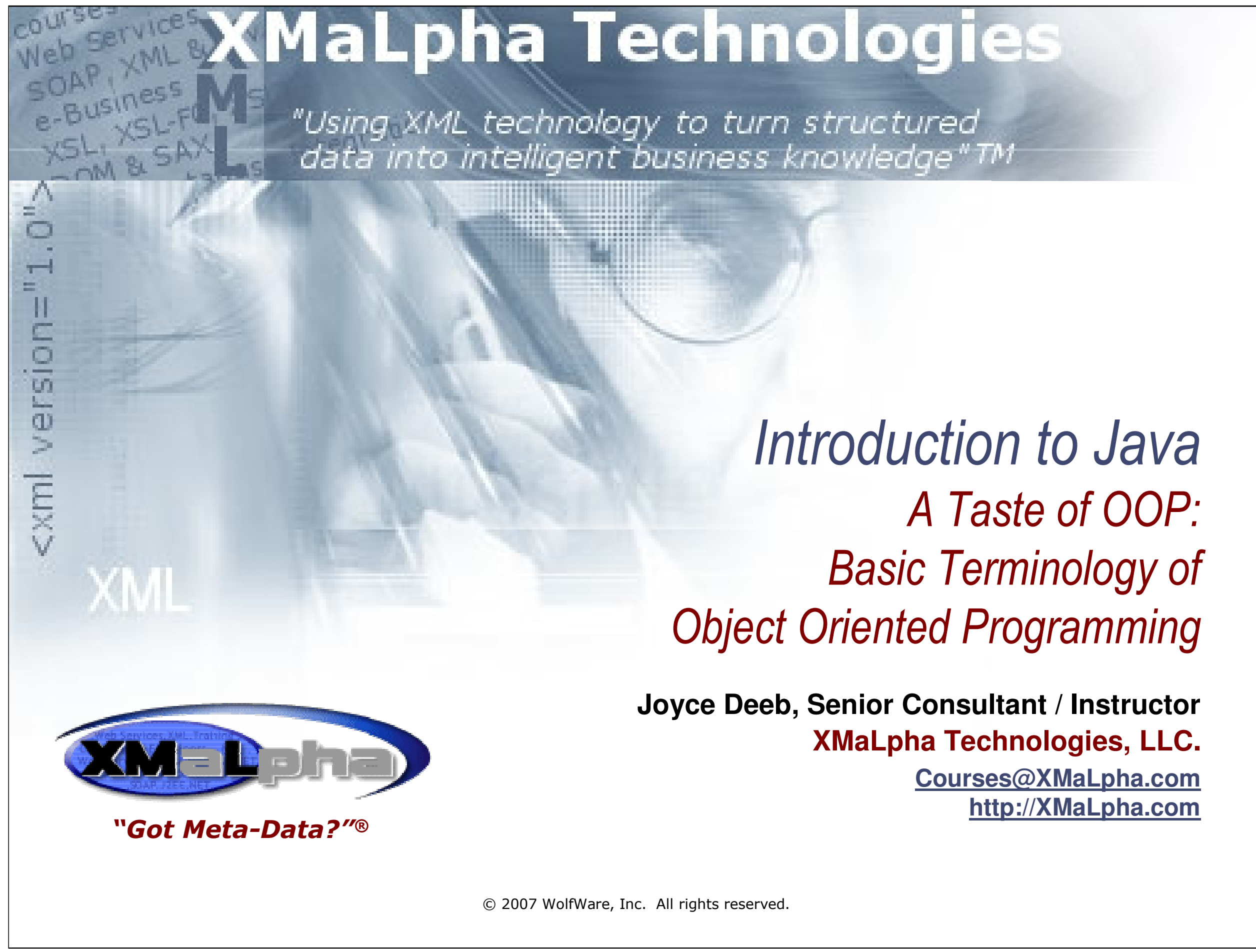
<Xselerator>
Partner

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.



Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

XMaLpha

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 2

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

OOP Terminology

- Class, Object, Instance
- Attributes
- Methods, Behaviors
- Message Passing, Invoking Methods
- Abstraction
- Relating classes to other classes
- Inheritance
- Composition
- Polymorphism

Copyright © 2007 WolfWare, Inc. All rights reserved.

Every new topic seems to come with its own vocabulary, and so does OOP. You may already be familiar with these, and we've already been using some of them. In this section, we'll learn to speak OOP.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Class

- Basic building block of OOP
- Should correspond to real-world entities
  - Physical
  - Conceptual
- Encapsulates state and behavior
- Specifies what objects look like and how they behave

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 4

Key questions from CoreJava:
- What is the object's behavior?
- What is the object's state?
- What is the object's identity?

Two Order objects may contain the same OrderItems, but they still need a unique identity (PO # in the real world).

State is sometimes referred to as the data or the attributes. Behavior is the functionality or methods.

One common way to begin to look for objects in a problem statement is to look for the nouns. This isn't a surefire method, but it is often a good start.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Object

- Instance of a Class
- Examples:
  - Class: Person   Object: John Doe
  - Class: Car   Object: John Doe's Car
  - Class: Circle   Object: circle123

Copyright © 2007 WolfWare, Inc. All rights reserved.

The specific instances of the generic class. You can think of the class as a blank form and the object as the form after you have filled it out. Now it is personalized with you information.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Attributes

- Features of an Object
- Specified in general in the class
- Given specific values in the instance
- Examples:
  - Person: name, SSN
  - Car: make, model, color
  - Circle: radius

Copyright © 2007 WolfWare, Inc. All rights reserved.

The attributes are the state or data part of the class definition.

XMaLpha Technologies
"Using XML technology to turn structured data into intelligent business knowledge"™
Website: http://XMaLpha.com - E-Mail: Courses@XMaLpha.com
Copyright © 2007 WolfWare, Inc. All rights reserved.

Methods

- The tasks an object can do
- Also called behavior, messages or functions
- Specified in the class
- Typically invoked through objects
- Examples:
  - Car: accelerate, stop, start
  - Circle: getRadius, calcArea, draw

Copyright © 2007 WolfWare, Inc. All rights reserved.

The collective group of public methods for a class is often referred as the API for the class. This is because OOP is based on the concept that data is kept hidden, and interaction with an object is controlled by the public methods defined in the class.

Methods fall into 2 categories:

-Accessor, where the method only accesses data in the object, but does not change the data.

- Mutator, where the method actually changes the data in the object.

The Java naming convention is to use getAttributeName() as the name of accessor methods, and setAttributeName() as the name of mutator methods,

where AttributeName is the name of the specific attribute.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Message Passing

- OOP generic term
- Objects send messages to each other to cause actions (methods to execute)
- Implemented by invoking a method on an object
- Example:
  - circle123.calcArea();

Copyright © 2007 WolfWare, Inc. All rights reserved.

In true OO-speak, client objects send messages to server objects. This really boils down to an object invoking a method on another object.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Abstraction

- Hide the details
- Interface specified at a more abstract level
- Simplify interaction
- Requires robust implementation

Copyright © 2007 WolfWare, Inc. All rights reserved.

The idea behind OOP is to build robust, reusable software components. If we think of these as building blocks, then we can build a bigger structure (program) by putting the smaller components together correctly, without having to worry about the details of those smaller components. Hardware vendors have been doing this for years. Seldom do they build from scratch. They more often design around pre-built components. As long as the components behave as specified, they can more easily build larger components.

In OOP, we typically choose to hide the data, and control access to it through public methods. This allows us to centralize data validation and ensure consistency. It also makes it easier to narrow the scope of bugs when debugging.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Relating Classes to Other Classes

- Class A Uses Class B
  - when class A calls a method in class B
  - when class A creates, receives, or returns objects of type class B
- Inheritance: Class A IS-A Class B
  - when class A is a subclass of class B
- Composition: Class A HAS-A Class B
  - when class A contains object of type class B

Copyright © 2007 WolfWare, Inc. All rights reserved.

Building discrete classes only gets us part way to where we want to go. We also need to be able to specify how those classes interact with one another. In practice, the less interaction, the easier it is to maintain the code in the long run. However, if we have hidden the implementation and can abide by the public API for those classes, then we should still be ok.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance

- Way of relating classes to each other
- Forms hierarchies of classes
- Generalize common features in ancestors
- Specify unique information in descendants
- Derived class extends Base Class
- Allows Polymorphism

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance provides the ability to organize hierarchies of classes. This allows us to place common features and functionality in parent classes, and to put specialized information in the child classes.

Further vocabulary:

Parent class is also the ancestor, superclass, or base class.

Child class is also the descendent, subclass, or derived class.

In Java, we implement inheritance via the *extends* keyword: The derived class extends the base class.

Inheritance is one of the facilities necessary for implementing polymorphism, which we'll talk more about in a bit.

Inheritance is a major feature of Java, and we will cover it more later in the class.

Examples of inheritance:

Manager IS-A type of Employee

RushOrder IS-A type of Order

GradStudent IS-A type of Student

Copyright © 2007 WolfWare, Inc. All rights reserved.

Composition

- Class is composed of one or more other types of objects
- Examples:
  - An Order has OrderItems in it
  - A Car has an Engine in it

Copyright © 2007 WolfWare, Inc. All rights reserved.

Composition is when a class is a composite of other types of objects.

Composition should pass the HAS-A test, whereas Inheritance should pass the IS-A test.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Polymorphism

- Literally, many forms
- When a single line of code can result in different behavior based on the type of object
- Very powerful programming technique

Copyright © 2007 WolfWare, Inc. All rights reserved.

Polymorphism is a powerful programming technique in OOP. We can write code that will look the same, yet function according to the type of object it is working on. Ultimately, it allows us to write more generic code, thereby writing less code. Anytime we can write less code, it will not only get done faster, it will be easier to maintain.

Polymorphism requires inheritance and dynamic binding. We'll see more about this in a later section.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Exercise

- Your eccentric cousin Zeke is a music fanatic. He's got an extensive CD collection and even more digital music files on his computer. Trouble is, he can't find what he's looking for on his computer. He'd love to turn his computer into his own personal Music Central, where he could select tracks for programs that fit a specific purpose, such as party music, background music, relaxation, etc. You decide to sign on to help with his endeavor. Right off the bat, one class should jump out at you. Identify the class, it's attributes and behaviors. For each attribute, identify possible values or ranges of values. For each behavior, identify what kind of information will be required as input and what should be the output.
- If you look a little more carefully, another class is evident as well. This class is an exercise in composition, as it's composed of other objects. Identify this class in the same manner.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

XMaLpha

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java Classes

- Components of a Java Class
- Instance Data
- Class Data
- Constructors
- Creating Objects
- Instance Methods
- this Reference
- Class Methods
- …

Copyright © 2007 WolfWare, Inc. All rights reserved.

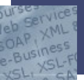
Copyright © 2007 WolfWare, Inc. All rights reserved.

Java Classes

- Overloaded Methods
- The main() Method
- Pass by Value vs Pass by Reference
- Qualifiers
- Garbage Collection
- Finalizers, Instance Initializers, Static Initializers
- Packages
- A Peek at UML

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.



Components of a Java Class

- State + Behavior == Data + Methods
- Data
  - Instance Data
  - Class Data
- Methods
  - Constructors
  - Instance Methods
  - Class Methods
- Other
  - Initializers, Finalizers

Copyright © 2007 WolfWare, Inc. All rights reserved.

Components of a class are also called **members** or **fields**.

There are two basic types of components, which are data and methods.

There are also two types of data: instance and class.

There are basically the same two types of methods as well, but constructors are a special type of method.

The other components that you might find in a class are instance and static initializers.

Finalisers are really just instance methods, but they are also somewhat special.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance Data

- Separate copy for each instance
- Created when object is instantiated
- Have default values
  - 0 for numeric
  - false for boolean
  - null for objects
- Can be initialized where defined instead
- Can be accessed from within Instance Methods

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance data is data that has a separate copy for each instance of the class that is created. When an object is created, memory is set aside for this data. If no initial values are specified (we'll see how to do this), the data will have default values according to the data type.

Instance data can only be accessed from witin instance methods.

Example: Suppose you have a MoneyMarketAccount class and for each instance, you want to keep track of the balance in that account, and perhaps the owner of that account.

```
public class MoneyMarketAccount {
   double balance;
   String owner;
}
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Class Data

- One copy for entire class
- Created when class is loaded
- Have same default values as Instance Data
- Can be initialized when defined instead
- Can be accessed from within Instance Methods or Class Methods
- Keyword: static
- Example:
  - static int MAX = 999;

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 7

Suppose for that same MoneyMarketAccount you want to specify the interest rate. Typically, the interest rate is the same for all MoneyMarketAccounts at your particular bank, so there is no need to have a copy of this data associated with every object. Instead, just have one copy associated with the class as a whole.

Class data is also called **static** data. It is created when the class is loaded, which is basically the first time the JVM encounters the class name in code. Class dtat can also be initialized when it is defined, or it will also use the default values corresponding to the data type.

Class or static data can be accessed from either instance or class methods.

We recognize class data in code by the word *static*.

```
public class MoneyMarketAccount {
   public static double rate = 0.04;
   …
}
```

C and C++ Programmers: static has several meanings in C and C++. In Java, it only has this one.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Constructors

- Special type of method
- Purpose is to initialize objects
- Invoked when object is created via new
- Must have same name as Class name
- Must not have a return type
- Format:
  - [qualifiers] ClassName ([arguments])
  - { … }

Copyright © 2007 WolfWare, Inc. All rights reserved.

Constructors are special methods. Constructors are called when an object is created via *new*. The purpose of these methods is to set up the object in some desired initial state.

There are strict rules for defining constructors. The constructor must have the exact same name as the class itself, case-sensitive as always. It also must not have a return type. Unfortunately, the error message is a bit obscure if you accidentally specify a return type.

Note in the format that the qualifiers and the arguments are in square brackets, indicating that they are optional. They parentheses are **not** optional.

Constructors are methods and they must have a code block that encases the body of the constructor, even if the block is empty.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Default Constructors

- Constructor with no arguments
- Provided by the compiler if no constructor is defined
- Simply allocates memory for the data
- Member data is initialized according to data type

Copyright © 2007 WolfWare, Inc. All rights reserved.

Syntactically, a class does not need a constructor defined in it. IF there is no constructor defined, the compiler will provide one, but it will only do this if there are **no** constructors defined. This is called the default constructor. However, you should be aware that in some texts, default constructor takes on 2 meanings:

1. The constructor provided by the compiler in the absence of any other.
2. Any constructor that takes no arguments.

We'll see later in the course why this second meaning is sometimes used.

XMaLpha Technologies
"Using XML technology to turn structured data into intelligent business knowledge"TM
Website: http://XMaLpha.com - E-Mail: Courses@XMaLpha.com
Copyright © 2007 WolfWare, Inc. All rights reserved.

Overloaded Constructors

- More than one constructor per class
- Each must have a unique argument list
  - number of arguments
  - data type of arguments
  - order of arguments

Copyright © 2007 WolfWare, Inc. All rights reserved.

It is also possible, and fairly common, to have more than one constructor per class. These are called overloaded constructors. The rule is that each one must have an argument list that the compiler can uniquely identify. So, whenever the compiler encounters code that tries to create an object, it must know exactly which constructor to call. To distinguish between the multiple constructors, the compiler looks at the argument list.

The following are unique argument lists:

MoneyMarketAccount(double amount, String name)

MoneyMarketAccount(String name, double amount)

MoneyMarketAccount(double amount, String name, String ssn)

MoneyMarketAccount(double amount, long ssn, String name)

However, the following would not be allowed along with the others:

MoneyMarketAccount(double amount, String name, String jointName)

because there already is a constructor that takes arguments double, String, String. Even though the arguments have different identifiers and represent different values, the compiler cannot distinguish between these.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Constructors Calling Constructors

- If there are multiple constructors in a class, it is possible to have one constructor call another
- The first line of the constructor must use the keyword this to call another constructor
- Example:
  - MoneyMarketAccount() {
  - this(0, “”); //call MMA(double, String)
  - }

Copyright © 2007 WolfWare, Inc. All rights reserved.

If there is a significant piece of initialization code that several of the constructors need to do, it may be easier to put it in one constructor and have the others call that constructor. The alternative would be to have a completely separate method with this common code and have all of the constructors call it.

For one constructor to call another, it must use the keyword *this*. This is one of two ways that the *this* keyword is used. The other is to refer to the invoking object, which we'll see shortly.

When a constructor (call it A) is called:

1. All data members are initialized, either to their default values or initial values if specified.
2. All instance initializers are executed in the order in which they appear.
3. If the constructor (A) calls another constructor (B), that constructor (B) is executed.
4. The rest of the constructor (A) is executed.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Creating Objects

- Done through the use of new
- Format:
  - ClassName objectId = new ClassName(arg list);
- Invokes the constructor corresponding to the argument list
- If no constructor matches the argument list, result is a compiler error

Copyright © 2007 WolfWare, Inc. All rights reserved.

Just a quick refresher on how to create objects.

Keep in mind that constructor calls must match a constructor definition, just like any other method call. The constructor that matches the argument list will be invoked.

Object definition can be done in separate steps:

```
Circle c1;              // no object yet, just a reference
c1 = new Circle(3);     // now there is an object
```

IMPORTANT: After the first step, there is no object yet. This only defines a variable, c1, that is allowed to refer to a Circle object. After the second statement, c1 actually refers to an object.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance Methods

- Method that is associated with individual instances
- Must be invoked from an object
- Can access both instance and class data
- Format:
- [qualifiers] returnType methodName ([arguments])
- { body }

Copyright © 2007 WolfWare, Inc. All rights reserved.

These are the most common type of methods in a class definition. Their purpose is to act on an individual object.

Generally, these are broken down into 2 categories:

- accessors, which only access data

- mutators, which can also change the data

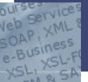
However, there is no syntactical differentiation between these.

In order for these methods to act on objects, they need to access instance data. However, they can also access class data. These methods access instance data through the implicit *this* reference.

Example:

```
public class Circle {
   int radius;
   Circle (int r { radius = r; }                                  // Constructor
   double area() { return 3.14 * radius * radius; }       // instance method
}
```


Copyright © 2007 WolfWare, Inc. All rights reserved.



Accessing Members with '.'

- The dot operator allows us to access data and method members
- Format:
  - instanceName.instanceData
  - instanceName.instanceMethod()
  - instanceName.classData //not preferred
  - instanceName.classMethod() //not preferred
  - ClassName.classData //preferred
  - ClassName.classMethod() //preferred

Copyright © 2007 WolfWare, Inc. All rights reserved.

The general format for the dot operator has a class name or instance name on the left side and a member on the right side. We can tell the difference between data members and method members by whether or not there are parentheses.

When working with class data or class methods, it is preferable to access them through the class name because that makes it more clear in the code that you are dealing with class members.

Example:

circle123. Area();

Copyright © 2007 WolfWare, Inc. All rights reserved.

this Reference

- Refers to the invoking object
- Available to every instance method
- Occasionally necessary to get to the invoking object
- Sometimes preferred to explicitly state invoking object
- this is a keyword

Copyright © 2007 WolfWare, Inc. All rights reserved.

Each instance methods has access to an implicit reference named *this*. This reference refers to the object through which the method was invoked. So when we have code like:

```
circle123.area()
```

the area() method will work with the data from circle123, which is the invoking object. In the area() method, *this* refers to circle123.

In the previous example, area() could have been written as:

```
double area() { 3.14 * this.radius * this.radius; }
```

Furthermore, when there is naming ambiguity, *this* is necessary. If the parameter in the constructor was also named radius:

```
Circle(int radius) { this.radius = radius; }
```

There is some disagreement as to whether this is a good style. On one hand, radius is always radius, and you don't have to remember which is r, and which is radius. On the other hand, if you forget to use *this*, it can be a tough bug to find.

Another place where *this* is necessary is if you nee to pass the invoking object to another method.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Class Methods

- Method that can be invoked from an object, or from the class name
- Preferred style to invoke from class name
- Does not have an implicit reference to an invoking object (no this)
- Cannot access instance data
- Can access class data
- Keyword: static

Copyright © 2007 WolfWare, Inc. All rights reserved.

Class methods are methods that can be invoked without having an object. This is done by invoking it through the class name instead. Since there is no requirement to have an object, there is no *this* reference. If there's no invoking object, how can there be a reference to it? Since they don't have a *this* reference, they can't access instance data.

These methods can be invoked through objects, but it is preferred to invoke them through the class name since is is clearer in the cod that you are dealing with a class method.

We recognize class methods in Java by the keyword *static*.

```
public static void getMAX() { … }
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

- Scope is limited to the code block in which they are defined
- Must be initialized explicitly - default values do not apply here
- Trying to use a local variable that has not been initialized will result in a compiler error
- Be careful if naming the same as instance data
- Applies to both class methods and instance methods

Copyright © 2007 WolfWare, Inc. All rights reserved.

As with variables defined inside of any other code block, variables defined within a method are limited to scope in that method.

Unlike instance data, local variables must be explicitly initialized. If code uses a local variable that has not been initialized, a compiler error will result.

As we saw when we looked at *this*, local variables that have the same name as instance data will overshadow the instance data. Either avoid naming local variables the same as instance data, or be very careful when doing so.

These rules apply to local variables in both instance and class methods.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Overloaded Methods

- More than one method with the same name in the same class
- Must have unique argument lists
  - number of arguments
  - type of arguments
  - order of arguments

Copyright © 2007 WolfWare, Inc. All rights reserved.

Just as we can have overloaded constructors, we can have overloaded methods. The same rules apply for determining uniqueness.

The method name along with the argument list is sometimes called the signature. However, texts vary somewhat in the definition and sometimes include the return type as part of the signature. Return type is irrelevant to overloading methods. The following is not allowed:

```
int getBalance()
double getBalance()
```

Even though the return types are different, these methods have the same name and argument list.

Copyright © 2007 WolfWare, Inc. All rights reserved.

main() Method

- At least one is necessary per application
- Each class can have one
- Convenient for test code for class
- Format:
  - public static void main(String [] args)

Copyright © 2007 WolfWare, Inc. All rights reserved.

We used main() in the small programs that we've written so far. All applications must have at least one main() method. When we run programs from the command line with the *java* command, we are actually specifying which class' to begin running with. Because of this, it is possible for each class to have a main() method, since we will choose which one to run. IN fact, this is one technique used for testing. Each class' main() method can contain code to test out that class. This code would not be run as part of the overall application, but could be used when testing modifications to the class.

The main() method must have the format shown on the slide with very little variation. It is public to allow the method to be invoked from outside the class. It is static to allow the method to be invoked without having an instance of the class. The return type must be void. The name must be main, all in lowercase. The argument list must be an array of Strings. You can name the array anything you like.

We'll see more about the qualifier *public* soon.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Pass by Reference vs Pass by Value

- All parameters are passed by value in Java
- However, objects and arrays look as though they are passed by reference
  - Can be changed if the class definition allows it
  - Class definition is responsible for determining what can be done to an object
  - Cannot set reference to refer to a different object

Copyright © 2007 WolfWare, Inc. All rights reserved.

This probably seems strange to programmer coming from other languages. The philosophy behind this is that the class definition is responsible for determining what can and cannot be done to an object.

Trick: If you need to modify the value of a primitive in a method, pass it in as an array of 1 item. Values in arrays can be changed. Actually, arrays work much the same as the examples we'll see on the following couple of slides.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Pass by Reference vs Pass by Value

- Example: Swapping Circles

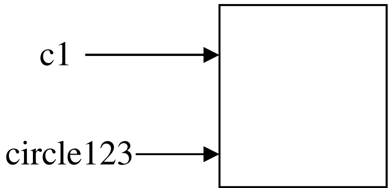
```
public void swap (Circle c1, Circle c2) {
   Circle temp = c1;
   c1 = c2;
   c2 = temp;
}
```

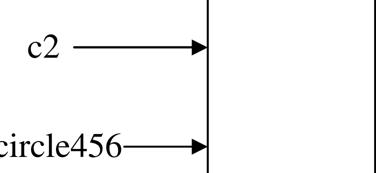
This will not change the arguments passed in:

```
swap (circle123, circle 456);
```

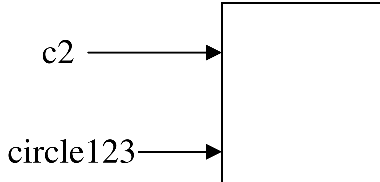
Copyright © 2007 WolfWare, Inc. All rights reserved.

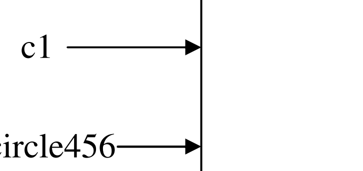
Start:





End:





The references c1 and c2 are local variables that refer to an object. Changing the object they refer to inside of the method does not change what circle123 and circle456 refer to.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Pass by Reference vs Pass by Value

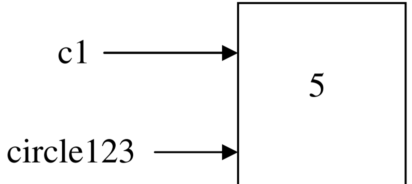
- Example: Using references to modify objects

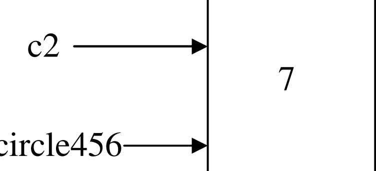
```
public void grow (Circle c1, Circle c2) {
   c1.setRadius(c2.getRadius() + c1.getRadius());
}
```

This will change the first argument passed in:

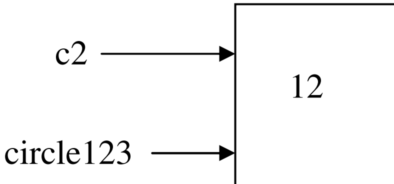
```
   grow (circle123, circle 456);
```

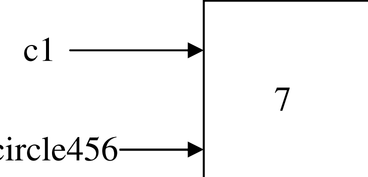
Copyright © 2007 WolfWare, Inc. All rights reserved.





End:





The Circle class is responsible for controlling access to and modification of circle objects. As long as it has methods for getRadius and setRadius, the grow method can change the radius of the arguments passed in as shown. Actually, it can do it without needing these methods, which we'll see shortly.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Qualifiers

- Modify:
  - class definitions
  - methods
  - data
- Qualifiers:
  - public
  - protected
  - private
  - final
  - static
  - native

Copyright © 2007 WolfWare, Inc. All rights reserved.

Qualifiers can modify class definitions, methods, and data. We've already seen one of these qualifiers at work: **static** for defining class methods and class data.

The qualifiers public, protected, and private allow us to control access to methods and data. The qualifier public also applies to classes.

The final qualifier takes on a variety of meanings depending on its use, which we will see shortly.

The native qualifier is used when you are incorporating methods written in another language.

There is another qualifier, *abstract*, which we will see later in the course.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Access Qualifiers

- Public
  - gives access to anything
- Protected
  - gives access to descendents
- Private
  - limits access to the methods of the immediate class
- If no modifier is specified
  - defaults to package protection, which gives access to everything else in the same package

Copyright © 2007 WolfWare, Inc. All rights reserved.

There are actually 4 levels of access, even though there are only 3 qualifiers for this. The levels, in order of most access to least access, are:

public - Most access. Item is available to the rest of the program.

package - Default if no qualifier is present. Item is available to any other class in the same package. **Note:** *package* is a keyword, but it is not used in the this context. It is used to package classes together.

protected - Used in conjunction with inheritance. Item is available to descendents. Does not apply to classes. **Note:** Item is still available to other classes in the same package.

private - Least access. Item is only available to the methods in the same class. Does not apply to classes.

As state before, data is typically private, methods are typically public.

Protected should be used with caution, since you don't know who may extend your class. This will make more sense in the inheritance section.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Private Data, Public Methods

- Stereotypical Setup
- Many exceptions
- Necessary to preserve encapsulation
- Easier to debug, maintain, enhance
- Doesn't account for inheritance (protected)

Copyright © 2007 WolfWare, Inc. All rights reserved.

This is the stereotypical recommendation for class visibility. The reason for this is to control access to the data, and keep the details hidden. Managing access and manipulation through controlled methods tends to make it easier in the long run as the code starts to grow.

Suppose we have order objects with a data member for the paid status. If we restrict modifications to this field through methods that know how to handle it, we can reduce potential problems down the road. For example, suppose code somewhere changed the status to cancelled even though it had already been paid. This would be harder to maintain than to only have one method that controls the status to make sure it is logical. Also, suppose you add an additional status for this field after much code is already written. Now all of the code needs to be checked to see if it is still correct, instead of just the member methods that control the status.

Of course there are circumstances where we want public data, like a global constant. Sometimes we have methods that are utilitarian for other methods in the class. Fi we don't want to maintain those for use outside of the class, we can make them private.

This stereotype doesn't take into account the other visibility options of protected and package. These are certainly useful as well. What we want to avoid is public data.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance Methods Returning Objects

```
Public class Student {
   private String name;
   private StringBuffer major;
    …
    public String getName()
       { return name; }
    public StringBuffer getMajor()
       { return major; }   // CAREFUL!
    …
}
```

Copyright © 2007 WolfWare, Inc. All rights reserved.

Caution: Be aware of the consequences of returning a mutable object from a method.

In the above example, suppose you have code that uses this student class as such:

```
Student s1 = new Student( … );
String sName = s1.getName();
StringBuffer sMajor = s1.getMajor();
```

Now, sMajor and major refer to the same object in memory. Should you change sMajor, you also change major. This violates encapsulation, and also is probably not what you want to do. The attribute major is no longer really private after all. It's not a problem for name, because name is a String and Strings are immutable. To fix this, you should return a copy of major. In the inheritance section, we'll see how to use the clone() method to do this.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance Methods and Private Data

```
Public class Circle {
   private int radius;
    …
   public void growBoth (Circle c2) {
     int total = c2.radius + this.radius;    //access c2.radius
     this.radius = total;
     c2.radius = total;                          //modify c2.radius
   }
   …
}
```

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 27

Another interesting characteristic of access protection is that a class method can modify the private data of any object of the same type. In the example on the slide, suppose we invoke the growBoth method():

```
Circle circle123 = new Circle(5);
Circle circle456 = new Circle(7);
circle123.growBoth(circle456);
```

Now both circles will have a radius of 12. You may have expected that this method could only access private data from the invoking object (circle123), but in fact, it can access private data from *any* Circle object.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Final Qualifier

- Applied to Class
  - class cannot be extended
- Applied to Method
  - method cannot be overridden
- Applied to Primitive Variable
  - data is a constant
- Applied to Object Variable
  - reference is constant
  - object can mutate, depending on class definition

Copyright © 2007 WolfWare, Inc. All rights reserved.

The final qualifier has slightly different meanings depending on how it is applied.

If a class is final, that class cannot be extended.

If a method is final, that method cannot be overridden in descendent classes. All methods in a final class are final by definition.

If final is applied to a piece of data, that data is a constant and its value cannot be changed.

```
final int MAX = 999;
```

We saw a similar example earlier where max was a static variable. Actually, it doesn't make much sense to have a final value that is not also static. If the value is final, it is constant. Why have a constant value replicated for each object of that particular class? It's useless to have multiple copies of a constant value. Instead, if the data was also static, there would be only one copy of it, so the best definition of the constant MAX would be:

```
static final int MAX = 999;
```

BTW, qualifiers must come first, but they can actually come in any order, so although static usually precedes final, they can be reversed.

Qualifying an object with final makes the reference final, so it cannot refer to any other object once it is set. The object can be modified, however, depending on what the class definition allows.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Static Qualifier

- Applied to Method == Class Method
  - method can be invoked without an object, through the class name
- Applied to Data == Class Data
  - there is one copy of the data for the entire class, not one copy per object of that class type

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 29

This is just a refresher; we've seen static in use already.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Native Qualifier

- Used for incorporating code that was written in another language.
- Instead of a method definition, there is only a prototype for the method, such as:
  - public native double calcSomething();
- Once defined as such, there is no difference in code between using this function, or any other Java function.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Incorporating native code, such as C or C++ into a Java Program is a somewhat involved task that is beyond the scope of this course. For now, just realize the meaning of the native qualifier, and that this is possible in Java.

Website: http://XMaLpha.com - E-Mail: Courses@XMaLpha.com
Copyright © 2007 WolfWare, Inc. All rights reserved.

Garbage Collection

- Creating objects dynamically allocates memory on the heap
- In many languages, it is up to the programmer to release this memory for reuse
  - common area of programmer error
- Java manages the reclamation of memory with the garbage collector
  - low priority thread runs in the background
  - reclaims memory when there are no longer any references to it

Copyright © 2007 WolfWare, Inc. All rights reserved.

If you've used dynamic memory allocation in other languages, you probably have suspected that the use of *new* in Java could cause some memory use problems. That's because in many other languages, the programmer must manage dynamically allocated memory, which includes giving it back when it's no longer needed. Unfortunately, this has been an area of common programmer error.

Java uses a garbage collector to reclaim unused memory. The garbage collector is a low priority thread that runs in the background. When the CPU is available, it fires up and goes through the heap looking for memory that is no longer in use. If there are no references to an object, it is available for collection.

For local variables, when they go out of scope, they become eligible for collection.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Garbage Collection

- Part of the JVM
  - variability to exactly how it is implemented
- Cannot determine exactly when the garbage collector will reclaim memory
- Can specify that an object is available for collection by setting the reference to null
- Can request that the garbage collector run by:
  - System.gc();
- Guaranteed to run the object's finalizer before reclaiming it

Copyright © 2007 WolfWare, Inc. All rights reserved.

If you want to make an object available for collection, you can set the reference to null:

```
Circle c1 = new Circle(5);
…
c1 = null;                    // now c1 is eligible for collection
```

It is not possible to force the garbage collector to run, or to know exactly when it will run. It is possible to request it to run by making a call to the gc() method in the System class:

```
System.gc();
```

The garbage collector will run the object's finalize method before reclaiming the memory.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Finalizers

- Somewhat like destructors in other languages
- Most common use in other languages was to release memory for reuse
- Due to the garbage collector, there is little need for these any longer
- Still used to free other resources
- Format:
  - public void finalize()
  - { … }

Copyright © 2007 WolfWare, Inc. All rights reserved.

In languages like C++, we used destructors to clean up after we were through with an object. The most common action (by far) of these methods was to give back dynamically allocated memory. They were also used to force dynamic binding.

Since Java uses garbage collection and dynamic binding, there is little use for such a method. However, if there is a need for this, the method is called *finalize*. It is actually specified in the Object class and is inherited by all other classes. We'll talk more about this in the inheritance section. If you need a *finalize* method, you will have to override it in your class definition. We'll learn how to override methods in the inheritance section as well.

If you do implement finalize() in your class, you should have it call its parent's finalize() method as well. We'll see why in the inheritance section. For now, just know that the last line of your finalize method should be a call like:

```
super.finalize();
```

super is a keyword. We'll see it again in the inheritance lesson.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance Initializers

- Can be more than one in a class
- Run in the order of appearance each time an instance is created
- Sometimes considered a preferred way to initialize instance data (vs. constructors)
- Can be useful with anonymous classes
- Format is a code block at the top-level of a class definition

Copyright © 2007 WolfWare, Inc. All rights reserved.

Instance initializers are another way to initialize newly created objects. They are an alternative to constructors. One school of thought is that it may be more clear to initialize each piece of data near where it is defined, instead of in a constructor, which may be farther away from the definition. In this case, there would be a short initializer after each data member definition. This is not the more common approach.

When we see anonymous classes, we will see why instance initializers can be useful with them.

The way to recognize an instance initializer is that it is a code block that is at the top-level of the class definition. This doesn't mean how near the beginning of the file it is. Top-level means that it is only encased in the class definition code block, not nested any deeper.

```
public class Circle {
    int radius;
    {   double value = Math.random();// instance initializer
        radius =  (value < 0.5) ? 6 : 8;
    }  …
}
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Static Initializers

- Can be more than one in a class
- Run once in the order of appearance when the class is loaded
- Useful for tasks that need to be done once per class
  - loading native libraries
  - initializing complex class data
- Format is a code block at the top-level of a class definition with the keyword static in front of it

Copyright © 2007 WolfWare, Inc. All rights reserved.

Static initializers are useful for initializing complex static data, since the static initializer is run only once when the class is loaded. Consider the MoneyMarketAccount example. Suppose the interest rate needed to be read in from a database.

They are also useful for tasks that need to be done in order for the class to function, but again, not to be done more than once. An example of this is loading native code libraries. In order to use a native function, the library has to be loaded. Since this is not part of the JVM mechanism, this must be done manually. However, you don't want to do it more than once. Therefore, classes that use native functions also use static initializers to load the native code library.

The format is identical to the instance initializer, except that the static initializer code block is preceded with the keyword static.

```
public class Circle {
    …
    static {  code block here … }
    …
}
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Class Design Hints

- Always keep data private
- Always initialize data
- Don’t use too many basic types in a class
- Not all fields need individual field accessors and mutators
- Use a standard form for class definitions
- Break up classes with too many responsibilities
- Make the names of your classes and methods reflect their responsibilities

Copyright © 2007 WolfWare, Inc. All rights reserved.

These hints are from Core Java, Volume 1 by Horstmann and Cornell.

Private data maintains encapsulation.

Relying on default values can cause bugs that are difficult to find.

If you have a group of attributes that belong together, make them into a separate class. The classic example is breaking out the fields for an address into a separate object. We do the same in relational databases.

If it doesn’t make sense to provide access to a piece of data, don’t.

Be consistent in your class definitions. This will make them easier to read and maintain. One template sugggestion: Separate public, package, and private members. Within each of these areas, keep order according to: constants, constructors, methods, static methods, instance variables, static variables. This is based on the reasoning that readers are more interested in the public API than the private internals.

If a class seems to be doing too much stuff, it may need to be broken down into more classes.

Use good names, and follow either the Java Naming Conventions, or your own. Accessor methods should begin with *get* and mutator methods should begin with *set*.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Packages

- Packages are groups of related classes
- The Java library is organized into classes
- Provides an additional level of collaboration amongst classes
- If no package is specified, class belongs to an unnamed default package
- If not using packages
  - classes must be in the current directory
  - current directory (“.”) must be on the CLASSPATH

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 37

Java classes are organized into groups called packages. The Java class library is a collection of such packages.

For most of the code we write in this course, it won’t make much difference if we package the classes, but in practice, all real classes should be packaged. If a class is not packaged, it must be in the current directory to access it.

IMPORTANT: Packages are not necessarily a good level of protection. Anyone can add classes to a package, and by doing so, gain access to all data and methods in that package that have package protection. Therefore, be careful not to forget to put private on data that you want private. Defaulting to package access is probably not what you want to do.

Look at the Java API documentation. The upper left corner is a list of the packages. The window below that is a list of classes and interfaces in the selected package. The window on the right side is the selected class.

Packages

- Keyword: package
- Format:
  – package myPackage;   //must be 1st line of code
- Recommended naming convention includes domain name in reverse:
  – com.wolfwareinc.wwclasses
- Provides for unique class names, even across developers and companies
- Packages are hierarchical and correspond to a directory structure

Copyright © 2007 WolfWare, Inc. All rights reserved.

The format for specifying that a class belongs to a particular class is to put a package statement as the first line of code (excluding comments) in the class file.

Sun recommends using a domain name to name your packages since this should guarantee uniqueness.

Packages correspond to a directory structure.   We'll see more about this in a bit.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Importing Classes or Packages

- Keyword: import
- Format:
  - import myPackage.MyClass;   // single class
  - import myPackage.*;   // all classes in myPackage
- Can only import a single package per statement
- import statements must come after the package statement if there is one, and before the class definition(s)

Copyright © 2007 WolfWare, Inc. All rights reserved.

The full name of a class includes the package in which it exists. For instance, the full name for String is java.lang.String since it is part of the java.lang package. When using classes, we have the option of specifying the full class name, or the abbreviated part, such as String. If we want to use the abbreviated form, we must import the package. So why didn't we have to import java.lang to use String? Because java.lang is assumed. The compiler always searches it and you never need to import it. This is the only package that is treated this way. All of the standard java classes are within the package hierarchy beginning with *java*. Classes considered to be extensions to the java library are within a package hierarchy beginning with *javax*.

You can only import a single package per import statement. SO if you need classes in both java.awt and java.util, you need an import statement for both of them. You can't just import java.*. This last statement would only find classes in the java directory off of a location on the CLASSPATH not classes in any of its subdirectories.

The statement *import myPackage.MyClass;* will look for a file called MyClass.class in a directory named myPackage, where this directory can be in any of the locations specified on the CLASSPATH.

Tired of hearing about CLASSPATH yet? It's on the next page...

Copyright © 2007 WolfWare, Inc. All rights reserved.

CLASSPATH

- Operating System environment variable
- Comes properly set, unless you need to add locations for additional classes
- If setting explicitly, be sure to include the current directory (".")
- Items on this path should be the location where the directories of your classes are located, NOT the package directory itself.

Copyright © 2007 WolfWare, Inc. All rights reserved.

CLASSPATH is on OS environment variable. Setting it depends on the OS you are working on.

If you download the SDK, CLASSPATH is already set to include the current directory. Should you set your own CLASSPATH variable, be sure to include the current directory, or you will have to package everything. If you name a package *MyPackage*, you will nee to put all of the classes in this package in a directory called MyPackage. Suppose you put this directory at:

C:\javastuff

Then the entry on the CLASSPATH should be:

C:\javastuff

If you add the MyPackage directory to this, the compiler will look in the MyPackage directory for the MyPackage directory.

Adding the current directory as well, you could end up with something like:

CLASSPATH=.;C:\javastuff

This is Windows notation. Unix uses the opposite slash and colons instead of semicolons.

Besides directories, CLASSPATH entries can alsobe .jar or .zip files.

Copyright © 2007 WolfWare, Inc. All rights reserved.

A Peek at UML

- Unified Modeling Language
- Popular means of specifying OO designs
- Tools:
  - Rational Rose
  - Together J
- Collection of views of the system
- Each view is a slightly different perspective
- Views must be consistent with each other

Copyright © 2007 WolfWare, Inc. All rights reserved.

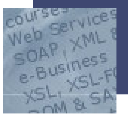
The Unified Modeling Language was developed by Grady Booch, Jim Rumbaugh, and Ivar Jacobson, sometimes referred to as the Three Amigos. They all now work for Rational Corporation. The UML is a notation for specifying OO designs. Again, there is nothing magical about it, but since it has become the prevalent notation for OO designs, it's worth looking at.

There are various views in UML. Each is a view of the system from a different point of view. Together, these views should present a complete picture of the system.

UML is a notation, not necessarily a process. For this, Rational provides the Rational Unified Process (RUP). With all of this notation and process, it is easy to get overwhelmed. One response to this "analysis paralysis" is Extreme Programming(XP), which is at the opposite end of the spectrum. While each of these may be the best choice for certain projects, it is often more useful to tailor a process to a given project.

UML, RUP, and XP are all beyond the scope of this course, but we will take a look at one of the UML diagrams, the Class Diagram.

Copyright © 2007 WolfWare, Inc. All rights reserved.



UML Class Diagram

Class Name
attribute attribute:dataType attribute:dataType = value ...
operation operation(argList) : returnType ...

Argument list is specified as: (id1:dataType, id2: dataType, …)

The entries in the second and third sections are options. You should be consistent with whichever one you choose.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Common Library Classes

- java.lang.Math
- java.text.DecimalFormat
- java.util.Calendar
- Wrapper Classes
  - java.lang.Integer -- Byte, Short, Long also
  - java.lang.Double -- Float also
  - java.lang.Boolean
  - java.lang.Character
- The javadoc Utility

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

java.lang.Math

- Class of Mathematical Methods and Constants
- Cannot be instantiated - all members are static, so don't need to instantiate
- Has common arithmetic functions
  - sqrt, round, min, max, random, ...
- Has constants: PI and E
- Access is through the class name:
  - Math.PI
  - Math.random()

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Math class has utilities for common mathematical functions. See the Java API for a complete listing and details.

Math is a final class, so it cannot be extended. Also, the constructors are private, so it cannot be instantiated. That doesn't matter, though, because all of the member data and methods are static, so they are all accessible through the Math class name.

The area() method in the Circle class that we saw earlier in thecourse could have been written as:

```
public double area() {
    return radius * radius * Math.PI;
}
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

java.text.DecimalFormat

- Utility class to format numeric values
- Has extensive capabilities, making it somewhat complex
- Uses local settings to determine how to format:
  - 45,123.76 \\ US format
  - 45.123,76 \\ German format

Copyright © 2007 WolfWare, Inc. All rights reserved.

The DecimalFormat class API can be somewhat intimidating. That's because it has some really fancy capabilities, including representing values according to the local convention. However, it can also be easy to use, if you just want to handle some simple formatting.

The FormatDecimals.java file shows 2 examples.

DecimalFormat can also be used to parse Strings into numeric values:

```
String s = "123.4";
double x = new DecimalFormat().parse(s.trim()).doubleValue();
```

Here's what happens: First trim the string of any leading or trailing whitespace. You would do this if the string was gotten from somewhere out of your control. Then pass this new string to the parse method of DecimalFormat. That returns an object of type Number, which is actually an abstract class. The actual object is either a Long or a Double depending on the type of numeric value that the string contained. However, doubleValue() is defined for the Number class, so you can invoke it on the returned object.

Copyright © 2007 WolfWare, Inc. All rights reserved.

java.util.Date

- Represents an instance in time, in milliseconds
- Two constructors
  - Date() - contains time at which instance was created
  - Date(long) - creates a Date that is n milliseconds after 1/1/1970
- Cannot be broken down into year, month, day, etc.
- Superseded by DateFormat for parsing date Strings
- Superseded by Calendar for obtaining components of date

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 6

The Date class is no longer particularly useful, since most of its methods have been deprecated. It has a deprecated constructor that took a String and created a Date object. This has been replaced by the DateFormat.parse() method.

It also has deprecated methods for retrieving various components of the Date, and those have been superseded by Calendar. The documentation for the deprecated methods tell you which class and method to use instead.

Copyright © 2007 WolfWare, Inc. All rights reserved.

java.util.Calendar

- Abstract Class, but has static methods that returns a GregorianCalendar with the current date/time:
  - Calendar now = Calendar.getInstance()
- Utility class for date/time values
- Has methods for extracting parts of the date, and advancing dates
  - set() - does not recompute fields until a get, getTime, or getTimeInMillis is executed
  - add() - does an immediate recomputation
  - roll() - does an add() without affecting other fields

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Calendar class is actually an abstract class, but it has a static method, getInstance(), that can be invoked from Calendar. It returns an object of type GregorianCalendar.

The Calendar class is useful for extracting the components out of a date. It is also useful for changing dates where you want to roll a date forward n days for example. There are 3 ways to modify the Calendar object. Each has a group of methods that correspond to it. Whenever a field in the object is modified, there is the issue of modifying the other fields to correspond to it. For example, if you move a date forward 1 year, the day of the week will not be the same.

If you use the set() methods, the other fields are not recomputed until the get(), getTime(), or getTimeInMillis() method is invoked.

If you use the add() methods, the fields are immediately recomputed.

If you use the roll() methods, you can roll a single component without affecting any of the other fields.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Wrapper Classes

- Corresponding class for each primitive data type:
  - Byte, Short, Integer, Long, Float, Double
  - Boolean, Character
- The first 6 extend Number
- Number has methods for getting the value of the object as any of the 6 primitive numeric data types
- Objects of these types are immutable
- Most useful for static utility methods like those that parse Strings into numeric values

Copyright © 2007 WolfWare, Inc. All rights reserved.

If you want to create a collection of primitives, you could use the associated wrapper class to create objects from each of the primitive values. When you retrieve one of these from the collection, you will have to then invoke a method to get the numeric value out of the object. This isn't pretty code, and is probably less efficient than just using arrays in the first place.

However, these classes are useful for the static methods that a String as an argument and return the numeric value. These all throw exceptions, since there is a risk that the String provided didn't really represent a number.

The ParseNumbers.java file shows an example of using the wrapper classes to parse strings into numbers.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The javadoc Utility

- Utility that scans code files for special comments
- Produces HTML documentation from comment contents
- Allows code and documentation to be maintained in the same file
- The Java API documentation is a product of javadoc

Copyright © 2007 WolfWare, Inc. All rights reserved.

The javadoc utility is an excellent tool for producing documentation for your code. Once you get used to the common tags, it's quite simple to add the appropriate javadoc comments to your code. In fact, you could put empty javadoc comments in your class template files. If you are using an IDE, it may automatically insert some javadoc comments into your code.

By allowing the code and documentation to be maintained in the same file, there is a much better chance that the 2 will be in synch with each other.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Exercises

- Use Wrapper class utilities to parse command line arguments into numeric values. For example, take in 2 values from the command line and add them together. Handle the exception appropriately and test it by giving a non-numeric argument.
- Write a program to print out a table of raises. Use 197.53 as a starting amount (salary) for each case. Print out a table of the results if it received a 4,5,6,7, and 8% raise. Use decimal values, as below:
  - Percent Starting Salary Increase Amount Ending Salary
  - 4 197.53 15.80 213.33
- Make sure you only print out 2 decimal places
- Add javadoc comments to at least part of your jukebox code. Run javadoc on your files and check out the results. Try to use a variety of tags at least once.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Relating Classes to Other Classes

- Introduction to Inheritance
- The extends Keyword
- Overriding Methods
- Access Modifier: protected
- Introduction to the Object Class
- Casting and Converting Objects
- The instanceof Operator
- Comparing Objects
- Revisiting Polymorphism
- Abstract Classes
- Composition

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance

- Allows derivation of new classes from existing classes
  - take advantage of code already written
  - hierarchies often model the real world
- Generalize common features into parent
- Specify distinct features in various children

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance allows us to derive new classes from existing classes. The obvious benefit of this is that we can reuse the implementation of the parent class. There are other benefits of inheritance that aren't quite as obvious, but we'll see that it allows us to write more generic code as well.

When building an inheritance hierarchy, common features are consolidated into the parent class, and unique ones are put into the children.

Inheritance Vocabulary:

If class B inherits from class A, then

A is called the Base, Parent, Super, or Ancestor class

B is called the Derived, Child, Sub, or descendant class

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance Examples

- Shape
  - Circle, Rectangle, Triangle, …
- Employee
  - Manager, Executive, Salaried, Hourly, …
- Animal
  - Mammal, Bird, Reptile, …
    - Canine, Feline, …
- Garden
  - Rock garden, Vegetable garden, Flower garden, …

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance is not limited to one level, in fact there really is no limit to the number of levels in theory. In practice, more than 4 or 5 levels becomes cumbersome. In the animal example, we show 2 levels of inheritance, and there could potentially be many more. Be aware however, that you should only add levels to the hierarchy if you actually need them. If you were building a piece of software that tracked the animals up for adoption at the local pound, you probably wouldn't need this extensive hierarchy. The levels in the hierarchy and the classes themselves are very dependent on the particular application at hand.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The extends Keyword

- Java uses extends to specify inheritance
- Format:
  - class Child extends Parent {
  - Child class definition …
  - }
- If no extends clause is specified, class extends Object by default
- Child class inherits data and methods from parent
- Child can define additional data and methods
- Child can override inherited behavior

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java uses the extends keyword to implement inheritance. The syntax for implementing inheritance is actually very simple. Designing the hierarchy is the more difficult part.

We'll see more about the Object class soon.

The Child class inherits all members, data and methods, from the parent. It can choose to override inherited methods by giving them a different implementation. It is also common for the child to add additional data and/or methods. In fact, if it didn't add anything, inheritance wouldn't be particularly useful. However, sometimes child classes don't add anything; this is called **pure inheritance**. It is done for consistency in the hierarchy. We'll see an example of this later in the course.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Constructors and Inheritance

- Constructors are executed from the root down to the class being created
- By default, the constructor with no arguments is invoked in the ancestors
  - this behavior is transparent to the programmer
- Explicit parent constructor calls are necessary to execute a constructor that takes arguments
  - requires use of the keyword super

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 7

Whenever *new* is used to create an object, it invokes a constructor from that class. If the object is of any type other than *Object*, it also must invoke constructors in all of the ancestors, all the way up to Object itself. Although the code does not necessarily reflect this, you can think of it as a call to the parent constructor as the first line of each child constructor. The result is that the constructors are executed from the root down to the object being created. This mechanism can be completely transparent in code if there are constructors that take no arguments all the way up the chain, *and* if those are the ones that you want to use. If this is not the case, then an explicit call to the parent constructor is necessary. This is done using the keyword *super*. It is sometimes recommended to always put the call in your constructors, even if you are just using the default behavior.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Constructors and Inheritance Example

```
public class RockGarden {
    …
    public RockGarden() {
        super();            // not necessary, is there implicitly anyway
        …
    }
}

OR:
public class RockGarden {
    …
    public RockGarden() {
        super(100);         // call Garden(100) constructor instead
        …
    }
}
```

Copyright © 2007 WolfWare, Inc. All rights reserved.

For the code on this slide, assume the following:

```
public class Garden {
    …
    public Garden () { … }              // constructor takes no args
    public Garden(int sqFt) { … }    // constructor takes int arg
    …
}
```

The call to super() **must** be the first line of code in the child constructor. Since it's the first line, the constructors execute from the root down.

You might expect the same thing to happen for finalizers, but it doesn't. If you override finalize() in your class, the last line should be a call to the parent's finalize() method:

```
super.finalize();
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

Overriding Methods

- Method in the parent may not be adequate for the child
- Overriding Methods != Overloaded Methods
  - Overloaded must have different signatures
  - Overridden must have the same signature and return type
- Example:
  - Invoking pay() on any Employee should work differently for salaried and hourly employees

Copyright © 2007 WolfWare, Inc. All rights reserved.

Suppose there is a method in the parent that is not adequate or correct for the child. It is possible for the child class to override that method and provide a different implementation for it.

It is important to understand the difference between overloading methods and overriding methods. Overloaded methods are when you have multiple methods with the same name, but different signatures. The idea behind this is to provide a variety of behaviors for the same method for the same class. So a user of that class can choose any one of the implementations dependent on the situation. Overridden methods must have identical signatures and return types. The idea is that you want to block the inherited method from being used by users of your class. In a way, you are modifying the behavior instead of adding another implementation of it.

It is possible to invoke an overridden method through the use of *super*. It is only possible to invoke the method in the immediate parent, meaning you cannot chain together calls to super to go higher up the hierarchy. The keyword super is used as though it is an invoking object. So, in HourlyEmployee, if you want to invoke Employee.pay(), you would do:

```
super.pay();
```

Copyright © 2007 WolfWare, Inc. All rights reserved.

Access Modifier: protected

- The qualifier protected applies to inheritance hierarchies
- Members that are protected can be accessed by any descendant's methods, even if the descendant is not in the same package
- Be aware that any classes that extend descendants will also have access

Copyright © 2007 WolfWare, Inc. All rights reserved.

The *protected* access modifier is only relevant in the context of inheritance. It allows descendant classes access to members without opening up access to everything else. Any member that has protected visibility is accessible to all methods in all descending classes.

It is recommended that this modifier be used with caution for data. The reason is that you have no control over who will extend your classes, and they will have access to protected members. Protected methods, however, can be very useful.

By definition, protected access is more visible than package access, because anything with protected access is also available to anything in the same package.

C++ Programmers: This is different than how it works in C++.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Introduction to the Object Class

- Java has a single-rooted hierarchy where Object is the root of all other classes
- If no extends clause is specified, the class extends Object by default
- This mechanism can be very powerful
  - allows very generic code (e.g. Collections)
  - allows very generalized behavior (e.g. toString())

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 11

Java has a single-rooted hierarchy, where there is one class, *Object*, that is the ancestor of all other classes. Even though you have not used an *extends* clause in your programs, your classes have extended *Object*.

The Object class contains general purpose methods that can be useful to any class. Furthermore, it enables the implementation of utilities that can work on any object because they can count on the fact that any object will have certain methods, either directly implemented or inherited from Object. An example of this is the concatenation operation on Strings. If you concatenate any object to a String, the concatenation mechanism invokes a method called *toString()* on the object in order to concatenate it. This method is inherited from Object. While it is a good idea to override this method, even if you don't, the concatenation mechanism will still work.

The single-rooted hierarchy also allows for very generic code. In some languages, if you build a structure such as a linked list for one type of object, you have to rebuild it for use with another type of object. C++ has implemented a template syntax to get around this, but the syntax is pretty convoluted. Since all objects are really of type Object, you can build the linked list to work on Object, and use it for any type of object. In fact, that is exactly what the Collection utilities do. We'll see those later.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Methods in the Object Class

- toString()
  - converts the object to a string
- equals()
  - tests equality of objects
- clone()
  - produces a duplicate of the object
- getClass()
  - used for Reflection
- others …

Copyright © 2007 WolfWare, Inc. All rights reserved.

The toString() method by default prints out the name of the class with a hashcode that is created from the values of the fields in the object. This is not particularly useful. You should think about overriding this to do something more useful.

The equals() method by default does a shallow comparison. You should think about overriding this to do a deep comparison.

The clone() method is intended to produce a duplicate object. It is a non-trivial task to implement properly, and is beyond the scope of this lesson.

The getClass() method is used to find out the class of an object. It is useful as part of the Reflection facility.

These are just some of the methods provided in Object. It is worthwhile for Java developers to become familiar with the Object class.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Casting and Converting Objects

- Similar to casting and converting primitives
- Can only be done within an inheritance hierarchy
- Upcasting: from descendant to ancestor
  - == conversion, since this is always safe
  - doesn't require an explicit cast
- Downcasting: from ancestor to descendant
  - can be dangerous if you're not careful
  - requires an explicit cast

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 13

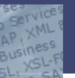
We saw how to cast and convert primitives in an earlier lesson. Similar rules apply for casting and converting objects. In general, if there is no danger involved, implicit conversions are fine. If there is potential for problems, an explicit cast is required.

First rule: the class types in question must be in the same inheritance hierarchy in order to cast or convert. This means that one must be the ancestor of the other.

Suppose we have a class for Shape, and a class for Circle which extends Shape:

```
Shape s = new Circle(6);
```

This is legal since it is really just another way to express a conversion. While we probably wouldn't do it directly like this, we very well may have a method that takes an argument of type Shape, and we pass it a Circle. This would be essentially the same thing.



Casting and Converting Objects

- A variable of an ancestor type can refer to an object of a descendent type
  - conversion; no explicit cast required
- A variable of an ancestor type can be assigned to a variable of a descendent type
  - explicit cast is required
  - type should be checked with instanceof first

Copyright © 2007 WolfWare, Inc. All rights reserved.

This is another way to express the rules for converting and casting objects.

```
Shape s;
Circle c = new Circle(5);
s = c;          //legal since c is really a Shape
```

This is a conversion which is legal because there is no question that c is a Shape. It happens to be a Circle in specific, but it is still a Shape.

Suppose we try the opposite: (c2 if of type Circle)

```
c2 = s;         // not legal because there's a possibility of error
```

In this case, we know that this is okay because s refers to a Circle object. But the compiler doesn't know that, so we have to be explicit:

```
c2 = (Circle)s;
```

This is syntactically legal, but not considered robust. In general, whenever downcasting, you should check to make sure the cast is legal. What if s actually referred to a Triangle? This last line would throw an exception. The instanceof operator can prevent this error:

```
if (s instanceof Circle)
    c2 = (Circle)s;
```


Copyright © 2007 WolfWare, Inc. All rights reserved.

The instanceof Operator

- The compiler allows upcasts because a derived variable really is a base object as well
- The compiler requires explicit casts for downcasting because a base variable may not be a derived object
- The instanceof operator tests the object type to see if the cast is legal
  - Shape s1 = new …
  - if (s1 instance of Circle)
  - r = ((Circle) s1).getRadius();

Copyright © 2007 WolfWare, Inc. All rights reserved.

Upcasting is always safe because:

Manager IS-A Employee

Circle IS-A Shape

Downcasting can be dangerous because some random Shape object may not necessarily be a Circle; it could be a Triangle instead. Recall that a variable of type Shape can refer to an object of any of its descendents. Hence, downcasting can be dangerous.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Comparing Objects

- DO NOT USE == TO TEST CONTENTS
- Remember: == just tests to see if the two variables refer to the same spot in memory
- Classes inherit a method equals() from Object
- As is, it only does a shallow comparison
- You should override this method to do a deep comparison if you need to compare objects of your own class types

Copyright © 2007 WolfWare, Inc. All rights reserved.

We should know by now not to use == with anything except primitives if we want to test the contents for equality. Since all classes inherit an equals() method, we might be tempted to use it as is. However, it only does a shallow comparison, similar to ==.

If you are going to use equals() to compare objects of you own class types, you should override it in those classes to do a proper comparison.

See the CircleCompare.java example in the code addendum for a simple example of this.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Revisiting Polymorphism

- Various definitions
  - overloaded methods or operators
  - overridden methods that behave differently depending on the object type
  - an object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy (from Core Java)
- Requires inheritance and dynamic (late) binding
  - slight performance hit
  - provides excellent generic code versatility

Copyright © 2007 WolfWare, Inc. All rights reserved.

It seems that every text has a slightly different definition of polymorphism. Sometimes it is used to refer to any overloaded method. The definition from Core Java seems as good as any other. Typically a piece of polymorphic code has a variable of a parent class type, which refers t o a variety of child class type objects over time. There may be a single line of code that will behave differently depending on what type of object the parent variable refers to.

Example:

Suppose Rectangle, Circle, and Triangle each have a method for area(). Obviously each one is different. Yet you can have a single line of code that can execute any one of the area methods:

```
Shape s;
…                          // in here, s will be set to refer to some child object
total = s.area();   // will take on different behavior depending on what s is
```

Obviously, the requires inheritance. But it also requires dynamic or late binding. This means that the actual method to invoke is not determined until runtime. Doing something at runtime instead of at compile time means taking a performance hit. However, in languages where programmers can choose binding, they typically choose dynamic binding anyway. There are ways to improve performance, such as using final methods where appropriate.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Abstract Classes

- Used as a parent in order to group together subclasses
- Has limited or no implementation details
- Cannot create objects of an abstract class
- Format:
  - public abstract class Shape {
  - public abstract double area();
  - …
  - }

Copyright © 2007 WolfWare, Inc. All rights reserved.

Abstract is a qualifier that canbe applied to classes and methods.

Think back to the example of calculating the area of various shapes using a variable of type Shape. This was a great way to get some generic code. We saw that each of the child classes had to have their own implementation of area(). But what is the definition of area() in the Shape class? We don't know, because we cant' calculate the area of a Shape. SO why would we put an area() method in Shape at all? Because of a simple rule:

Whatever is on the right side of the dot must be available to the variable on the left side of the dot.

So, for the following line to work:

```
s.area()
```

area() must be available to s, and since s is of type Shape, the area() must be defined for Shape. It can either be defined directly in the class, or inherited by the class. So we have to define a method that has no meaning. We make this an abstract method, because they have no body.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Abstract Methods

- Has no body, just a prototype
- Any class containing an abstract method must be declared abstract
- Must be implemented by subclasses, or they too must be declared abstract
- Abstract classes can also have:
  - methods that are not abstract
  - data members
  - These will be inherited by the descendents

Copyright © 2007 WolfWare, Inc. All rights reserved.

Continuing with our example, we would define area() as an abstract method in Shape. Each of the children would have to provide an implementation for it. If they do not, they must also be defined as abstract.

You can put data and complete method definitions in abstract classes. In fact, you should do this whenever possible, rather than implementing them in each individual subclass.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Composition

- When an object is comprised of other objects
- HAS-A relationship
- Contained objects are instance members
  – recall: initialized to null
- Special consideration needed for:
  – copying (cloning)
  – comparing
  – serializing

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 20

We've seen that we use inheritance to model an IS-A relationship. We use composition to model a HAS-A relationship. There really is no special syntax for implementing composition. It's really just defining data members that are of another class type. Since these data members are objects, we have an object that is at least partially made up of other objects. When we do this, the contained objects are not copied into the outer object. Instead, the data member variables refer to those contained objects somewhere else in memory. For the most part, it looks as though they are inside our outer object, but we'll see a couple of instances where we have to be aware of contained objects.

Recall that data members have default values and the default value for object variables is null. This means you don't have an object yet.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Concerns with Composition

- Deep Comparison
  - must make sure that each contained object also has a properly functioning equals() method, or we will have to compare its components as well
- Deep Copy
  - the same issue arises in clone() as in equals()
- Serializing
  - an analogous situation arises when serializing composite objects as well

Copyright © 2007 WolfWare, Inc. All rights reserved.

We saw that we need to write an equals method to handle comparing contents of our objects. However, our example only had primitive data members. If those members were also objects, we would have to invoke equals() on them as well. This means that contained objects must have a working equals() method in order for our equals() method to work

Although cloning is a bit more complex than comparing, the same issue needs to be considered. Each contained object must be cloned as well, which means that those classes must be able to clone themselves.

We will see serializing in the Streams section. This is when we send an object somewhere else, like over a network connection, or to a file. Java has a fantastic mechanism for serializing objects which makes it very easy. However, we still have to make sure that contained objects know how to serialize themselves as well.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Inheritance Design Hints

- Place common operations and fields in the superclass
- Use inheritance to model the IS-A relationship
- Don’t use inheritance unless all inherited methods make sense
- Use polymorphism, not type information

Copyright © 2007 WolfWare, Inc. All rights reserved.

These hints are from Core Java, Volume 1 by Horstmann and Cornell.

It is always a good idea to consolidate as much implementation as possible into the superclass.

Make sure the relationship is truly an IS-A relationship before using inheritance. Sometimes we are tempted to use it where it is not appropriate.

Make sure that all of the methods and data in the superclass apply to the subclass. Sometimes there is a true IS-A relationship, but the way the parent is implemented does not completely apply to the child. Suppose Circle had a grow method that allowed you to expand the circle. Then suppose you wanted to extend Circle with a FixedCircle class that represents a fixed size Circle. Although FixedCircle is a Circle, it would not work to have it inherit the grow method.

If you have code that is checking an object’s type in order to choose which method to invoke on it, you should see if polymorphism is applicable instead.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 “The Good, The Bad, and The Ugly”™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java Interfaces

- Interfaces as Pure Abstract Classes
- The implements Keyword
- Implementing Multiple Interfaces
- Defining Interfaces
- Interface Hierarchies
- Tag Interfaces
- Interfaces as Data Types
- Cloning and Comparing

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Interfaces as Pure Abstract Classes

- Interfaces are like pure abstract classes
  - can contain abstract methods
  - can contain constants
- Alternative to multiple inheritance
- Great for behavior that crosses hierarchies
- Keywords: implements, interface

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java does not allow a class to extend more than one class (multiple inheritance). Historically, the pros and cons of multiple inheritance have been hotly debated. One of the problems with it is the possibility of inheriting conflicting behaviors. It also puts a much bigger burden on the compiler. Java uses interfaces as an alternative to multiple inheritance. Interfaces are like pure abstract classes because they have no implementation in them. They can only contain abstract methods and constants.

Why would you want multiple parent classes anyway? If a class needs to inherit 2 sets of characteristics, why not just put both sets in the parent class? Because one of those sets might be applicable to classes outside of your immediate hierarchy. In fact multiple inheritance and interfaces work best for features that are generic enough to cross hierarchies, yet don't necessarily apply to all classes. In fact, if you look at the interfaces in the Java library you'll see this a lot. Interfaces such as Cloneable, Comparable, and Serializable are features that could apply to virtually any class that you can think up. Yet, you might not want your class to have these features. By making these interfaces, you can choose if you want your classes to have these behaviors.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The implements Keyword

- Java uses implements to specify that a class implements an interface
- Format:
  - class NewClass implements AnInterface {
  - NewClass definition …
  - }
- NewClass must implement all methods in AnInterface, or must be declared abstract

Copyright © 2007 WolfWare, Inc. All rights reserved.

Since interfaces are like pure abstract classes, any class that implements an interface must provide definitions for all of the (abstract) methods in that interface. If it doesn't, then it must be declared abstract because it is not a complete class definition.

NewClass can provide whatever implementation it wants for the abstract method(s) in AnInterface. The interface doesn't care what the implementation is. It only cares that the signature is correct. It is concerned with guaranteeing that any object that implements AnInterface will be able to perform any of the tasks in AnInterface. In practice, this means that it must be legal to invoke any of the methods in AnInterface on any of the objects that implement this interface.

NOTE: If a parent class implements an interface, then all of the children automatically implement that interface (they inherit the abstract method implementations) without having to specify an *implements* clause.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Implementing Multiple Interfaces

- A class can implement more than one interface
- Format:
  - class NewClass implements Interface1, Interface2 {
  - NewClass definition …
  - }
- NewClass must implement all methods in both interfaces or must be abstract

Copyright © 2007 WolfWare, Inc. All rights reserved.

It is possible to have a class implement more than one interface. This would make sense because there could certainly be more than one set of generic behaviors that you want your class to have. Perhaps you want it to be Cloneable and Serializable.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Defining Interfaces

- Similar to defining abstract classes
- Format:
  - public interface AnInterface {
  - Constants or Method Prototypes
  - }
- Constants are implicitly static and final
- Methods are implicitly abstract
- Class that implements interface can access constants in abbreviated form

Copyright © 2007 WolfWare, Inc. All rights reserved.

Besides using interfaces that are already defined, you can define your own as well. The syntax for doing so is much like that for defining classes. In fact, interfaces are a lot like classes.

Interfaces must be defined in their own file with the same name as the interface itself and a .java extension.

Interfaces are compiled the same way as classes are. This will also produce a byte code file with a .class extension.

All methods defined in an interface are implicitly abstract.

All data defined in an interface is implicitly static and final. Although including these qualifiers in the definition will not hurt.

A class does not need to implement an interface in order to access the constants defined in that interface. It can use the fully qualified name to access the constants:

    InterfaceName.CONSTANT_NAME

However, if it does implement the interface, then it can use the constants without the interface name as a prefix.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Interface Hierarchies

- It is possible to build an inheritance hierarchy of interfaces
- Similar syntax to class hierarchies
  - public interface ChildInterface extends ParentInterface {
  - … }
- Interfaces can extend more than one interface
  - public interface Child extends Interface1, Interface2 {
  - … }

Copyright © 2007 WolfWare, Inc. All rights reserved.

It is also possible to build a hierarchy of interfaces. There are few examples of this, and it would require a complex design to warrant it, but it can be done. Interface hierarchies can have multiple inheritance. This is allowed because implementation details are not involved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Tag Interfaces

- Empty Interface
- Used as a flag to indicate class meets some particular criteria
- Also called Marker Interfaces
- Examples in the Java Class Library
  - Cloneable
  - Serializable

Copyright © 2007 WolfWare, Inc. All rights reserved.

Tag interfaces are somewhat odd in that they are completely empty. They are used as a flag to indicate that a class meets the criteria that the interface is responsible for. There are 2 popular examples in the Java library, Cloneable and Serializable. Cloneable is a marker interface that serves as a flag for the Object.clone() method. It tells this method that objects of this class type are allowed to be cloned. Serializable is similar in that it tells the serialization mechanism that objects of this class type are allowed to be serialized. In both cases, it is another way of saying that the developer of the class has analyzed the situation and has approved the class for the task at hand.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Interfaces as Data Types

- Interfaces are valid data types
  - AnInterface xyz;
  - xyz can refer to any object that implements AnInterface
- Can use with the instanceof operator
  - if (xyz instanceof AnInterface)
  - will be true if xyz refers to an object that implements AnInterface
- Often used as method argument type
  - allows methods to work on objects from different hierarchies

Copyright © 2007 WolfWare, Inc. All rights reserved.

Interfaces are valid data types. So a variable can be declared to be of an interface type:

```
Surroundable s;
```

Now, s can refer to any object that implements the Surroundable interface.

Why would we want to do this? Well we probably wouldn't want to do it just like this. But we do want to be able to write methods that take arguments of an interface type. That way we can write methods that can work on objects from across multiple hierarchies. This is another example of writing generic code.

Since an interface is a valid data type, we can also use it with the *instanceof* operator:

```
if (s instanceof Surroundable)
```

This is handy if we are working with several different types of objects, and we only want to do a particular operation on those that are capable of doing it. This statement guarantees that we can invoke any of the methods declared in Surroundable on the object s. We don't need to know anything else about s, only that it is Surroundable. We'll look at a code example of this in class.

Remember the rule that the thing on the RHS must be available to the variable on the LHS. So when using the variable s, you can only invoke methods that are defined in Surroundable.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Problems with Cloneable

- Clone() is a protected method in Object
  - only methods in the class can clone objects of that class type
- Only works as is for primitives and Strings
  - does a bit-by-bit copy
  - Strings are immutable so both can refer to the same one
- If object refers to other objects, clone will also refer to those exact objects
  - not really a clone

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 11

We mentioned earlier that clone is a method provided by Object. However, overriding clone() can be somewhat complex. By default, clone() does a bit-by-bit copy of the object. This is fine for primitives, but if the object contains references to other objects, it actually copies the references. The result is that the new object refers to the same contained objects as the original. For Strings, you may decide that this is acceptable, since Strings are immutable, you can probably allow both to refer to the same one. This brings up a **key point**: It is up to the programmer to determine whether to allow cloning, and if so, to determine the correct method implementation.

Cloning Options

- Accept the default clone method as is
  - Implement the Cloneable interface
  - Override clone()
    - give it public access
    - call super.clone()
- Override clone() to provide correct behavior
  - do same as above, plus
  - call clone on contained objects as necessary
- Do not allow cloning (default)

Copyright © 2007 WolfWare, Inc. All rights reserved.

To accept the default clone() method, you probably still wanr to override it to get public access. This could be something like the following:

```
public class Circle implements Cloneable {
   …
   public Object clone() {
      try  {
         return super.clone();          // call the clone method in Object
      }
      catch (CloneNotSupportedException e) {
         return null;
      }
   }
}
```

Ignore the try-catch-exception stuff for now. We'll learn about that later. The point is, you have to override it to get public access, but you don't need to change the implementation, so just call the default clone method. This only works for classes that are direct children of Object. If they are not, you will have to set up cloning throughout the hierarchy.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Cloning Concerns

- Implementing Cloneable requires attention to detail
- Must make sure contained objects are either cloneable, or can be ignored
- Additional consideration needed if there is are multiple levels in the inheritance hiearchy

Copyright © 2007 WolfWare, Inc. All rights reserved.

Suppose your object contained a Circle object. Then you might do:

```
public class CompositeObject implements Cloneable {
   Circle c;                              //Instance variable
   …
   public Object clone() {
      try  {
         CompositeObject co = (CompositeObject)super.clone()   // Note 1
         e.circle = (Circle)circle.clone()                                   // Note 2
         return co;
      }
      catch (CloneNotSupportedException e) {
         return null;
      }
   }
}
```

Note 1: The cast is required because clone() returns an object of type Object.

Note 2: This requires that Circle also implement Cloneable.

Again, ignore the exception stuff for now.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Contains an abstract method
  - public int compareTo(Object obj)
  - This should return
  - 0 if the objects are ‘equal’
  - a negative number if the invoking object is ‘less’
  - a positive number if the invoking object is ‘greater’
- Used by the Arrays utility class for sorting arrays of objects

Copyright © 2007 WolfWare, Inc. All rights reserved.

Cloneable showed us an example of a tag interface. Comparable is an example of an interface that has abstract methods. In particular, it contains a method called compareTo(). Any class that implements Comparable is guaranteed to provide this method. In theory, this method should compare the invoking object against another object of the same type, and return what their ordering should be. In practice, it is only as good as the programmer makes it. Going back to the Circle class, this method might look like:

```
public int compareTo (Object obj) {
    return  this.radius - ((Circle)obj).radius;
}
```

In reality, this should do more error checking to make sure obj is of the correct type.

The sort routines in *Arrays* use this method to determine how to order an array of objects. These routines don’t care what type of objects they are sorting. They only care that they can invoke compareTo on them and get back an answer that will tell them how to order the objects.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Example: Sorting Circles

```
import java.util.Arrays;
public class SortCircles {
   public static void main (String [] args) {
      Circle [] circles = new Circle[10];
      for (int j = 0; j < circles.length; j++)
         circles[j] = new Circle( (int) (Math.random() * 9))
      for (int j = 0; j < circles.length; j++)
         System.out.println( circles[j].getRadius());
      Arrays.sort(circles);
      for (int j = 0; j < circles.length; j++)
         System.out.println( circles[j].getRadius());
   }
}
```

Copyright © 2007 WolfWare, Inc. All rights reserved.

This class shows how to use the *Arrays* class to sort objects in an array. As long as you can provide an implementation for compareTo() in your class, you have the ability to sort an array of your objects.

The Arrays class also has utilities that will search for an object in a sorted array. These use a binary search, and as such, are quite efficient. If working with arrays, it is worthwhile to become familiar with the utilities in the Arrays class.

Both of these examples are in the code addendum.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Streams

- Stream Basics
- Binary Output Streams
- Character Output Streams
- Binary Input Streams
- Character Input Streams
- The RandomAccessFile Class
- Serializing Objects
- The File Class
- The FileNameFilter Interface

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Streams

- A general purpose input/output class
- Provide a common way of doing I/O, regardless of source/destination
  - network connection, monitor, keyboard, thread, file
- Follow the Decorator Pattern
- To get such flexibility, dealing with Streams can become complex
- There are over 60 stream classes in the Java Library
  - must import from java.io

Copyright © 2007 WolfWare, Inc. All rights reserved.

Streams are the way Java handles input and output. The idea is that input and output will work the same, regardless of the source or destination.

The Stream classes are set up according to the Decorator pattern. The idea is to add functionality by attaching it to a barebones class. You can add as much functionality as you need. In Streams, you will start out with a barebones Stream class and add on functionality by using that Stream to construct a more functional Stream. This is called filtering.

There are a lot of different Stream classes, making it a pretty tough feature to wade through. However, we can take some cues from the names of the Stream classes:

Piped => Threads

Buffered => Efficiency

File => dealing with a physical file

Data => methods for dealing with primitives

Object => methods for dealing with serializing objects

Zip => dealing with compressed zip format

We'll look at a few basic streams and see examples of how to read and write binary data, character data, and objects.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Using Streams

- Choose the Stream to correspond to the basic task you are trying to do
- Determine what features you want to add to that Stream
- Create the Stream
  - Constructors typically throw IOException
- Use the Stream
- Close the Stream

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 5

We'll look at several examples of choosing streams for particular applications, and how to add functionality to them by filtering them.

For any Stream, closing is the same:

```
streamVar.close()
```

We should be sure to close Streams as soon as we are done using them.

Stream Characteristics

- 2 Types of Streams
  - Binary or Byte Streams
  - Character or Text Streams
- 2 Types of Operations (mainly)
  - Reading (Input)
  - Writing (Output)
- First step in choosing which Stream to use is to pick one of these 4 possibilities

Copyright © 2007 WolfWare, Inc. All rights reserved.

One way to break down the group of Stream classes is to split it into streams that deal with binary data, and those that deal with character data. For the most part, streams either do input or output as well. So by determining which one of these 4 combinations you want to do, you have a starting point for choosing your Stream class.

Copyright © 2007 WolfWare, Inc. All rights reserved.

4 Base Abstract Classes

- InputStream - base class for doing binary input
- OutputStream - base class for doing binary output
- Reader - base class for doing text input
- Writer - base class for doing text output
- Most other Stream classes descend from these
- Related Classes that are not in the Stream family
  - RandomAccessFile
  - File

Copyright © 2007 WolfWare, Inc. All rights reserved.

These are abstract base classes, which contain low-level methods for reading and writing. To make use of these capabilities, use descending classes that implement higher-level methods that are easier to use. The InputStream and OutputStream classes have methods for reading and writing bytes. The Reader and Writer classes have methods for processing Unicode characters.

There are a couple of closely related classes as well. The RandomAccessFile class acts like a Stream, but it not part of the Stream hierarchy. It has capabilities for non-sequential file access.

Another useful related class is File. The File class provides representation for physical files and directories.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Binary Output Streams

- Start with a type of OutputStream
  - FileOutputStream if sending output to a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedOutputStream for efficiency
    - typically want this
  - DataOutputStream for methods to output primitives
- The outermost Stream should be the one that has the methods you want to use

Copyright © 2007 WolfWare, Inc. All rights reserved.

FileOutputStream has the capability to write binary data to a physical file. A common setup for writing binary data to a file starts with a FileOutputStream and then adds buffering for efficiency. Finally, the useful methods are in DataOutputStream, which has methods to output primitives.

The FileOutputStream constructor takes a String or File as an argument. There is also one to specify whether or not to append. By default, it does not append.

The BufferedOutputStream constructor can take a FileOutputStream as an argument. This is how we "decorate" a stream with additional functionality. These streams use a buffer so that each operation does not require accessing the drive. Data is written when the buffer is full, or when the stream is closed.

The DataOutputStream constructor can take a BufferedOutputStream as an argument. It provides methods for writing primitives in a portable binary format.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Binary Output Stream

- Example:
- DataOutputStream dos = new DataOutputStream(
- new BufferedOutputStream(
- new FileOutputStream(“store.bin”)));
- The dos reference will allow you to use methods available to DataOutputStream:
  – writeDouble(), writeFloat(), writeUTF(), writeInt(), …
  – dos.writeDouble(myDoubleValue);
- Argument to constructor can contain a relative or absolute path, otherwise it’s assumed to be in the current directory

Copyright © 2007 WolfWare, Inc. All rights reserved.

Here is a common setup for writing binary data to a file. Although it is somewhat cumbersome to set up the streams, once this is done, the Stream variable is easy to use.

Take notice of the writeUTF() method. This allows us to write out a String that can then be read back in with a single readUTF statement. This is much handier than reading strings character by character.

See BinaryOutput.java for an example of this.

NOTE: The directory separator character is not consistent across platforms. If this is a concern, do not use a String literal that has a path in it. You have 2 options. You can create file objects successively for each directory and it’s immediate subdirectory. You can also use a platform independent file separator to construct your path String. This is stored as an attribute in the File class:

File.separator

You may also want to know what the user’s current working directory is:

System.getProperty(“user.dir”)

and the platform dependent line separator is in:

System.getProperty(“line.separator)

BTW, println() automatically uses this property.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Character Output Streams

- Automatically convert Unicode characters to the local character scheme
  - for US, this is ASCII
- Start with a type of Writer
  - FileWriter if sending output to a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedWriter for efficiency
    - typically want this
  - PrintWriter for methods to output text
    - println() and print()
    - Constructor requires an OutputStream or a Writer

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 10

The stream classes that write character data have built in functionality to convert the Java Unicode characters to the local character encoding scheme. These schemes correspond to local languages, and each JVM has a default encoding scheme. The stream classes write data based on that local scheme.

Unicode can represent most characters, but not the Chinese alphabet. Java also has the means to write UTF characters. UTF is an encoding scheme that will use 1, 2, or 3 bytes to write out a single character. For ASCII characters, it only uses a single byte, so it's very space efficient. For some foreign alphabets it requires 2 bytes, and for others it requires 3.

FileWriter has the capability to write character data to a physical file. A common setup for writing character data to a file starts with a FileWriter and then adds buffering for efficiency. Finally, the useful methods are in PrintWriter, such as println().

The FileWriter constructor takes a String or File as an argument. There is also one to specify whether or not to append. By default, it does not append.

The BufferedWriter constructor can take a FileWriter as an argument. It is analogous to the BufferedOutputStream.

ThePrintWriter constructor can take a BufferedWriter as an argument. It provides methods for writing text.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Character Output Streams

- Example:
```
PrintWriter pw = new PrintWriter(
                      new BufferedWriter(
                      new FileWriter("log.txt")));
```
- The pw reference will allow you to use methods available to PrintWriter:
  - print(), println(), …
- String argument to constructor can contain a relative or absolute path, otherwise it's assumed to be in the current directory

Copyright © 2007 WolfWare, Inc. All rights reserved.

Here is a common setup for writing text to a file.

Note: PrintWriter replaces the PrintStream class. PrintStream was intended to do the same thing as PrintWriter, but there are some limitations in its implementation. In particular, it assume ASCII as the translation destination format. You should use PrintWriter instead of PrintStream. However, there are some leftover implementations of it. Most noticeably, System.out is a PrintStream object.

See the CharacterOutput.java file for an example of this.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Binary Input Streams

- Start with a type of InputStream
  - FileInputStream if getting input from a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedInputStream for efficiency
    - typically want this
  - DataInputStream for methods to input primitives

Copyright © 2007 WolfWare, Inc. All rights reserved.

This setup corresponds very closely to the Binary Output Stream setup.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Binary Input Streams

- Example:
- DataInputStream dis = new DataInputStream(
- new BufferedInputStream(
- new FileInputStream(“store.bin”)));
- The dis reference will allow you to use methods available to DataInputStream:
  - readDouble(), readFloat(), readUTF(), readInt(), …
  - double myDoubleValue = dis.readDouble();
- String argument to constructor can contain a relative or absolute path, otherwise it’s assumed to be in the current directory

Copyright © 2007 WolfWare, Inc. All rights reserved.

Here is a common setup for reading binary data from a file. The read methods take a variable of the corresponding type as an argument and put the read data into that variable.

See BinaryInput.java for an example of this.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Character Input Streams

- Automatically convert the local character scheme to Unicode characters
- Start with a type of Reader
  - FileReader if getting input from a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedReader for efficiency
    - typically want this
- BufferedReader actually gives us a useful method:
  - String readLine()

Copyright © 2007 WolfWare, Inc. All rights reserved.

FileReader has the capability to read character data from a physical file. A common setup for reading character data from a file starts with a FileReader and then adds buffering for efficiency. Diverging from the pattern we've seen so far, BufferedReader actually gives us a method that we probably want to use. The readLine() method will read a line of text from input and return it as a String.

The FileReader constructor takes a String or File as an argument.

The BufferedReader constructor can take a FileReader as an argument. It is analogous to the BufferedInputStream.

XMaLpha Technologies
"Using XML technology to turn structured data into intelligent business knowledge"TM
Website: http://XMaLpha.com - E-Mail: Courses@XMaLpha.com
Copyright © 2007 WolfWare, Inc. All rights reserved.

Character Input Streams

- Example:
- BufferedReader br = new BufferedReader(
- new FileReader(“log.txt”));
- The br reference will allow you to use methods available to BufferedReader:
  - String getLine()
- String argument to constructor can contain a relative or absolute path, otherwise it’s assumed to be in the current directory

Copyright © 2007 WolfWare, Inc. All rights reserved.

Here is a common setup for reading text from a file. The getLine() method will return you a line from the file as a String.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Reading Text from Standard Input

- Use an InputStreamReader as the base stream
- Pass System.in to its constructor
- Filter it with BufferedReader to get the readLine() method
- Example
  - BufferedReader stdin = new BufferedReader(
  - new InputStreamReader(System.in));
  - System.out.println("Enter a line of text");
  - String memo = br.readLine();
  - System.out.println(memo);

Copyright © 2007 WolfWare, Inc. All rights reserved.

Reading data from the keyboard is an example of reading character data. We don't use a FileReader because we're not dealing with files. Instead we use a class called InputStreamReader and pass it System.in. This stream can then be passed to the BufferedReader constructor to get buffering and the readLine() method.

See the StandardInput.java file for an example of this.

A couple of utility classes that are useful with Character Input Streams are StreamTokenizer and StringTokenizer. These classes have methods for helping to parse a series of characters into tokens.

Copyright © 2007 WolfWare, Inc. All rights reserved.

RandomAccessFile

- Can read or write to the same file
- Read and write operations do not need to be sequential.
- RandomAccessFile is not part of the Stream hierarchy
- Has the same API as DataInputStream and DataOutputStream
  – methods for reading and writing primitives

Copyright © 2007 WolfWare, Inc. All rights reserved.

RandomAccessFile is not part of the extensive Stream class hierarchy. However, it has similar functionality to the Stream classes. Additionally, it allows operations to jump around in the file. We'll see that you can ask for the current location, move on, and then return to that location later.

You can use this class for use with physical files, but it does not apply to network connections, since these must be sequential access. For that, you need Streams.

Copyright © 2007 WolfWare, Inc. All rights reserved.

RandomAccessFile

- Constructors take either a String or a File and an additional String indicating read-only (“r”) or read-write (“rw”)
- Methods:
  - long getFilePointer()
  - long length()
  - void seek(long bytesIntoFile)

Copyright © 2007 WolfWare, Inc. All rights reserved.

The constructors for RandomAccessFile take either a String or a File object, and a second String that indicates whether you want to open the file for reading only or for reading and writing.

The getFilePointer() method will return the current location in the file as a long which is the number of bytes into the file.

The length() method will return the length of the file in bytes.

The seek() method will move the specified number of bytes into the file, always starting from the beginning. If you call seek(), any subsequent read/write operations will occur at the new location in the file.

See RandomAccess.java for an example of this.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Serializing Objects

- The process of sending/receiving objects across a Stream
- Used to be a tedious task
- Java has 2 classes to support serialization:
  - ObjectInputStream
  - ObjectOutputStream
- These classes have methods for I/O of objects:
  - Object readObject()
  - void writeObject(Object obj)

Copyright © 2007 WolfWare, Inc. All rights reserved.

The task of writing and reading composite objects has traditionally been a tedious task that required considerable design work and attention to detail. This troublesome task is provided for us in the Object Stream classes. These classes also have methods for reading/writing all of the primitive types, but their real strength is in reading and writing composite objects. These classes extend from InputStream and OutputStream.

The readObject() method reads the next object from the InputStream and returns it as type Object. Either you need to know what the object type is, or you need to use Reflection to get the type in order to downcast it for use.

The writeObject() method writes the given argument to the OutputStream.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Serializing Objects

- Requirements for using serialization methods:
  - Class must implement the Serializable interface
  - Contained objects must either be Serializable or declared as transient (a qualifier)
- Recall that Serializable is a tag interface
- Attributes qualified as transient will be ignored by the serialization process

Copyright © 2007 WolfWare, Inc. All rights reserved.

In order to use the Serialization methods, the object's class must implement Serializable. All contained objects must also be serializable in order for the methods to work with them. If they are not serializable, they must be declared as *transient* so that the methods will ignore them.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Object Output Stream

- Example:
- ObjectOutputStream oos = new ObjectOutputStream(
- new FileOutputStream("obj.bin")));
- The oos reference will allow you to use methods available to ObjectOutputStream:
  - oos.writeObject(myObject);
- Object can be read back in using an ObjectInputStream:
- ObjectInputStream ois = new ObjectInputStream(

new

Copyright © 2007 WolfWare, Inc. All rights reserved.

The object serialization methods make a complex task quite simple, by doing all of the recursive serialization of composite objects and all of the bookkeeping for efficiency behind the scenes.

See ObjectOutput.java for an example of this.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The File Class

- Useful for working with Streams
- Represents a file or directory
  - methods will tell you which type it is
- Constructors take either the file name as a single String, or divided up into path and subpath. Can also take a File object and a subpath String
  - File(String filename)
  - File(String path, String subpath-or-file)
  - File(File path, String subpath-or-file)
- Useful methods for working with file systems

Copyright © 2007 WolfWare, Inc. All rights reserved.

A class closely associated with Streams is the File class. The Stream classes are concerned with the contents of a file, while as the File class is concerned with the file's status in the file system. A File object can represent either a file or a directory. The class has methods that you can use to tell which one it is. The class has a nice selection of methods for working with file systems, such as finding out if a file exists, if it is writeable, renaming a file, and making directories.

Be sure to check out the Java API for File to see a complete list of the methods.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The File Class

- Constructor Examples:
  - File f1 = new File(“myFile”);
  - File f2 = new File(“subdir\MyFile”);
  - File f3 = new File(“subdir”, “MyFile”);
  - File f4 = new File(“subdir”);
  - File f5 = new File(f4, “MyFile”);
- Method Examples:
  - if (f4.exists() && f4.isDirectory() && f4.canRead())
  - // returns an array file names in f4
  - String [] contents = f4.list();

Copyright © 2007 WolfWare, Inc. All rights reserved.

It’s handy that we can use File objects to create Stream objects for a couple of reasons. We often need to validate a particular file before actually using it. The File methods allow us to verify the existence of the file/directory and check other information about it before using it to construct a Stream. Creating a file object does not create a physical file on the file system, although there is a method in File for doing so. However, we can also pass the file to a Stream constructor to do the same thing.

The list() method gives us an array of Strings which are the file names contained in the directory invoking object. It is an overloaded method, where we can also specify a means to filter this list.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The FileNameFilter Interface

- Works well with File objects that represent directories
- Filters the contents returned by list() so that you only get specific contents, like all of the .java files
- One method which tells which files to include:
  - boolean accept(File dir, String filename)
- You provide the implementation for accept() to filter whatever you want
  - return true to include the file

Copyright © 2007 WolfWare, Inc. All rights reserved.

Think of the window that pops up when you go to open a file. It shows the contents of a directory, but only those files that match what's in the box at the bottom, like only show .txt files. FilenameFilters help us do the same thing. This interface requires us to write a method that indicates true if we want a particular item included in the directory contents listing. The criteria for including a file is completely dependent on how we write the accept() method. We pass an object of type FilenameFilter to the list() method in order to filter the contents.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The FilenameFilter Interface

- Example:
  - File f = new File(“myDirectory”);
  - if (f.exists() && f.isDirectory()) {
  -   FilenameFilter fnf = new FilenameFilter() {
  -      public boolean accept(File dir, String fn) {
  -            return (fn.endsWith(“.java”)) ? true : false;
  -         }   // end accept method
  -   }   ;   // end anonymous class definition
  -   String [] contents = f.list(fnf);
  - }   // end if statement
  -

Copyright © 2007 WolfWare, Inc. All rights reserved.

Recall that there are several ways to get the behavior needed by an interface. We can have our class implement the interface. Or we can have a helper class that implements the interface. This helper class can either be a separate class or an inner class. We can use any kind of inner class that suits our purpose. In this case, an anonymous inner class is handy, because it puts the filtering criteria code very near the code that uses it. As long as this is the only place where this interface behavior is needed, this will work fine. Suppose this method needs to filter the directory contents several different ways. You couldn't handle that by having the class implement FilenameFilter, because then you could only write a single accept() method. You could do it with helper classes, but you would need one for each type of filtering. The easiest way is to repeat this code fragment for each type of filtering.

Recall that an anonymous inner class combines instantiation (*new*) with the class definition. Since the class has no name, the next thing after *new* either indicates the class it extends or the interface it implements. In this case, it implements FilenameFilter. The next thing is the body of the anonymous class, which only contains one method, accept(). The accept method uses the conditional operator to tell whether to return true or false.

This code is typically somewhat obscure for someone just becoming familiar with Java. However, it is a common way to get interface behavior that you only need to use once.

See FileOps.java for this example.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Exercises

- Make Track and Program Serializable. Their descendents will inherit this behavior.
- Write a driver class to test out this functionality by creating a program, serializing it, and reading it back in.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Welcome to Course: Introduction to Java

XMaLpha Technologies, LLC.,

Copyright © 2007, WolfWare, Inc. All rights reserved.

INTRODUCTION TO JAVA (HANDS-ON)

Technical

This hands-on technical course teaches the basics of Java programming. It is recommended for programmers and web developers who wish to learn the basics of web development using Java. It covers Java language fundamentals, and includes basic application layout, keywords and identifiers and primitive data types, and flow control.

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Overview of Java • Basics • OO Terminology	Morning • Inheritance • Interfaces
Afternoon • Classes • Library Classes	Afternoon • Streams • Collections

Copyright © 2007 WolfWare, Inc. All rights reserved.

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

Collections

- Java 1.1 Collections
- Enumeration
- Java 1.2 Collections
- Iterator
- Collections Class

Copyright © 2007 WolfWare, Inc. All rights reserved.

Collections are required as part of the Sun Java Certification Exam. Aside from certification, they are very useful tools that can save you a lot of programming time.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java Collections

- Alternative to arrays for holding groups of objects
- Implementations for most common data structures
  – stack, queue, linked list, map, hashtable
- Contained items must be objects
- Some collections are ordered (Lists)
- Some collections cannot contain duplicates (Sets)
- Some collections require key-value pairs (Maps)
- Disadvantage - lose type information

Copyright © 2007 WolfWare, Inc. All rights reserved.

Collections are now included on the Sun Java Programmer Certification Exam. There are many different collections, but the exam requires an understanding of 4 basic types:

a simple collection which does not enforce order and allows duplicates

a list which does enforce order and allows duplicates

a set which does not enforce order and does not allow duplicates

a map which supports searching on a key which must be unique

These features can be combined in various permutations as well. Collections also differ in how they are implemented. If you have taken a course on data structures, many of the Java collections will look familiar.

These classes were made possible because Java has a single-rooted hierarchy. Hence, by defining a linked list of Object, you can store any type of Object in the linked list. Since everything in Java is an Object, we can have a linked list of any object. We can also mix up the types of objects in the collection, though. We either need to keep it straight what's in the collection, or use Reflection to find out what the objects are.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java Collections

- Arrays
  - good for random access
  - slow for insertion, deletion, and growing in size
- Linked List
  - good for insertion, deletion, and growing in size
  - slow for random access
- Tree
  - good for insertion, deletion, growing in size, searching
  - slow for random access
- Hashing
  - good for insertion, deletion, growing in size, excellent for searching
  - slow for indexed access, requires overhead

Copyright © 2007 WolfWare, Inc. All rights reserved.

Specific implementations of collections most commonly fall into 4 basic categories: array, linked list, tree, and hash table. Each of these has advantages and disadvantages.

Arrays are good for fast random access, but poor for insertions and deletions. Arrays also cannot easily grow in size. An array can be searched somewhat efficiently, if the programmer knows the right algorithm.

Linked lists are good for insertions and deletions, but must do sequential access. They can easily grow in size. Lists do not need to be ordered, but typically work well for ordering. Linked lists are inefficient for searches.

A tree is good for insertions and deletions, and can also grow in size easily. A tree must be ordered. If constructed with some care, a tree can be searched very efficiently.

A hash table is an example of a map, as it requires a key-value pair. The unique key is used for access and searching. It is somewhat efficient for data access, and allows growth as well. Hash tables require some overhead for using the keys. Because of this, they are not considered worthwhile for small data sets.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Parent Interface is Collection or Map
- List and Set Interfaces extend Collection
- Collection class name can tell you a lot:
  – HashSet - uses hashing, does not permit duplicates
  – TreeMap - ordered map
  – TreeSet - order set, no duplicates, uses tree for storage
- Choosing best implementation to use may require an understanding of data structures and computational complexity

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Collection API class hierarchy is somewhat complex. The root is actually an Interface, either Collection or Map. List and Set are interfaces that extend Collection. There are also several abstract classes that implement these interfaces, which are extended by classes that we actually use. The name of a collection can often tell you quite a bit about its implementation and features. However, choosing the most appropriate implementation for a given problem requires some understanding of common data structures and computational complexity.

The heavy use of interfaces in the Collections API makes it relatively easy to swap one Collection for another. To do this, you need to use the Interfaces as data types instead of the actual collection class.

Copyright © 2007 WolfWare, Inc. All rights reserved.

- Vector
  - – probably most commonly used collection
  - – basically an array that grows in size easily
  - – updated for the 1.2 Collections
- Stack
  - – LIFO structure, subclass of Vector
- Hashtable
  - – typical map features, no null values, not in hierarchy
- BitSet
  - – used for working with individual bits

Copyright © 2007 WolfWare, Inc. All rights reserved.

There were 4 collections in Java 1.1. Actually, there really wasn't a Collection hierarchy then, so these are rather disjoint. However, Stack is a subclass of Vector.

Note that Hashtable is considered all one word, and the 't' is not capitalized.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Enumeration

- Java 1.1 Collections use an Enumeration to traverse the collection
- Can be used with Hashtable, Vector, and Stack
- Get the Enumeration by invoking elements():
  - Enumeration enum = myVector.elements();
- Methods:
  - hasMoreElements() - true if more elements
  - nextElement() - gets next element
- Superseded by Iterator

Copyright © 2007 WolfWare, Inc. All rights reserved.

We won't dwell on the Enumeration interface because it is not recommended any longer, since it has been superseded by Iterator. Vector (and thus, Stack) have been retrofitted to use Iterator as well.

To use Enumeration, invoke the elements() method on the collection. This method returns an object of type Enumeration. From that object, you can traverse the collection. Suppose I have a Vector of Circles:

```
Enumeration enum = circles.elements();
 while (enum.hasMoreElements())
    Circle c = (Circle) enum.nextElement();        // cast is necessary
```

Since collections work on Objects, we have to downcast to assign the element to a variable of its true type.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java 1.2 Collections

- Other collections added in 1.2
- Wide variety of implementations to choose from
- Lists
  - LinkedList, ArrayList, Vector
- Sets
  - HashSet, TreeSet
- Maps
  - HashMap, TreeMap

Copyright © 2007 WolfWare, Inc. All rights reserved.

Java 1.2 added to the collection classes substantially. In fact, the hierarchy of Interfaces and abstract classes was added with 1.2.

Vector was retrofitted to implement the List interface in 1.2. Other concrete classes that implement List are LinkedList and ArrayList. The main difference between these is the computational complexity of various operations.

Website: http://XMaLpha.com - E-Mail: Courses@XMaLpha.com
Copyright © 2007 WolfWare, Inc. All rights reserved.

Iterator

- Replaces Enumeration Interface for Collections
- Defined in List and Set Interfaces
- Get Iterator by invoking iterator():
  - Iterator iterate = myLinkedList.iterator();
- Methods:
  - hasNext() - true if there are more elements
  - next() - returns the next element
  - remove - removes the last item retrieved
- Vector and Stack also implement Iterator

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Iterator interface replaces the Enumeration interface. It is defined in the List and Set interfaces, so anything that implements those interfaces will have Iterator capabilities.

The shorter method names are also considered an improvement. There is also a remove() method that will remove the last item that was retrieved from the collection. If the collection has been modified in any way after the Iterator was gotten, these methods will not work properly.

Since Vector was retrofitted to implement List, Vector and Stack also have Iterator capabilities.

An example of using an Iterator to traverse a Collection is in Iterate.java.

Copyright © 2007 WolfWare, Inc. All rights reserved.

The Collections Class

- Utility class for Collections
- Similar to Arrays class
- Has methods for:
  - sorting (sort())
  - searching (binarySearch())
  - max, min (max() and min())
- Can specify means of Comparison for sorting and searching with a Comparator

Copyright © 2007 WolfWare, Inc. All rights reserved.

Recall the Comparable interface from the Interfaces section. The utility class, Arrays, uses it to sort an array of objects. Well Arrays also makes use of a similar interface, called Comparator. You can use either for dealing with arrays, as long as you implement the corresponding method. Comparable requires a method called compareTo(), and Comparator requires a method called compare(). Their behavior is the same.

Similar to Arrays, there is a Collections class that has utilities for working on Collections. Collections also uses both the Comparable and the Comparator interfaces.

Copyright © 2007 WolfWare, Inc. All rights reserved.

Comparator and Comparable

- Both are interfaces
- Arrays and Collections utility classes can make use of either
  - sort with just one argument uses Comparable
  - sort with an extra Comparator argument uses Comparator
- Comparable requires a functional compareTo() method
- Comparator requires a functional compare() method

Copyright © 2007 WolfWare, Inc. All rights reserved.

Page - 12

For both Arrays and Collections, if you call the sort method that only takes an array or collection, it expects the objects in the array or collection to implement Comparable, and uses the compareTo method to sort. Both classes have another option, where you can specify a Comparator to do the sorting instead.

In many cases, it doesn't matter which way you go, but if you run into a situation where the objects implement Comparable, and you want a different comparison, you would use the method that takes a Comparator and send it your own comparison for the immediate task at hand.

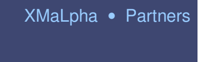
We'll look at some examples using Collections in class. In particular, we'll compare 4 cases:

Arrays, using Comparable - We saw this in the SortCircles example in the Interfaces section.

Arrays, using Comparator - Implemented in SortCirclesComparator.java.

Collections using Comparable and Comparator - Implemented in Iterate.java.

Copyright © 2007 WolfWare, Inc. All rights reserved.



- Any questions?

Copyright © 2007 WolfWare, Inc. All rights reserved.

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Copyright © 2007 WolfWare, Inc. All rights reserved.

</Presentation>

- Please feel free to E-Mail the courseware author at:

Joyce Deeb

Senior Consultant

E-Mail: JDeeb@XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

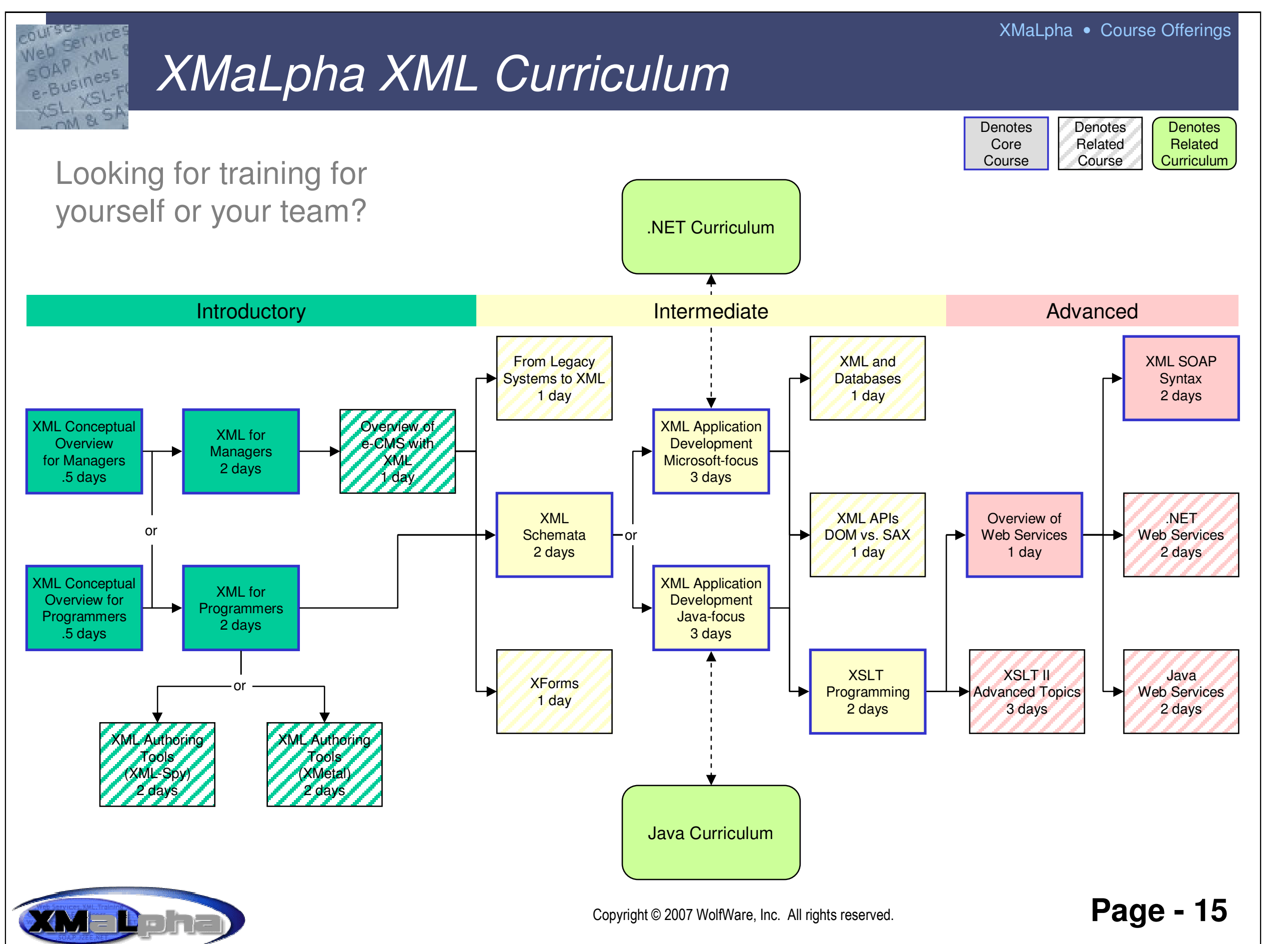
Thank you

If you have questions after the session is over that you think I might be able to help with, please feel free to contact me by E-Mail. In the event that I don't know the answer to your question, I will try and direct you to a Web or other resource for further information.

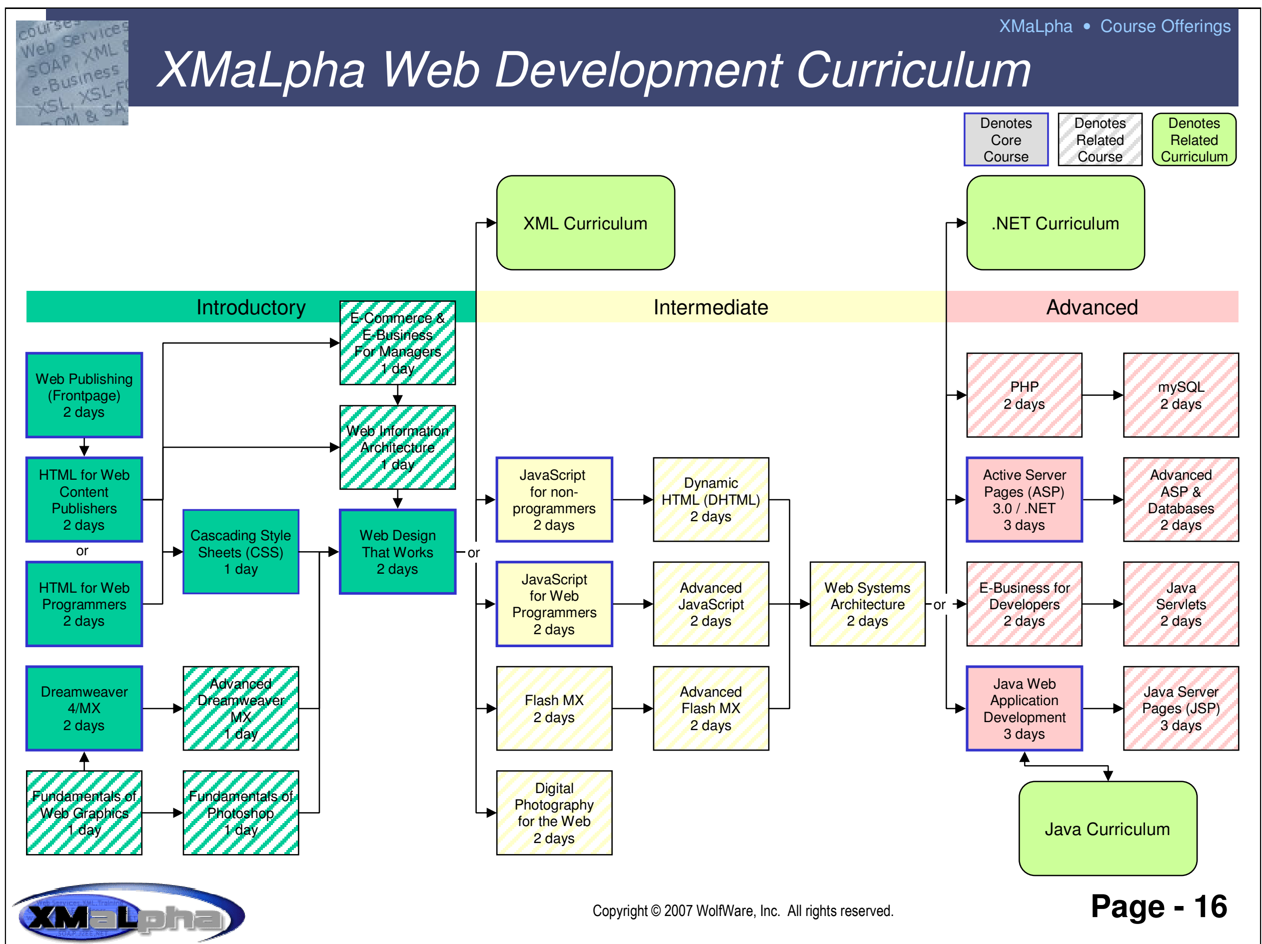
Please visit us on the Web at:

http://XMaLpha.com

Copyright © 2007 WolfWare, Inc. All rights reserved.

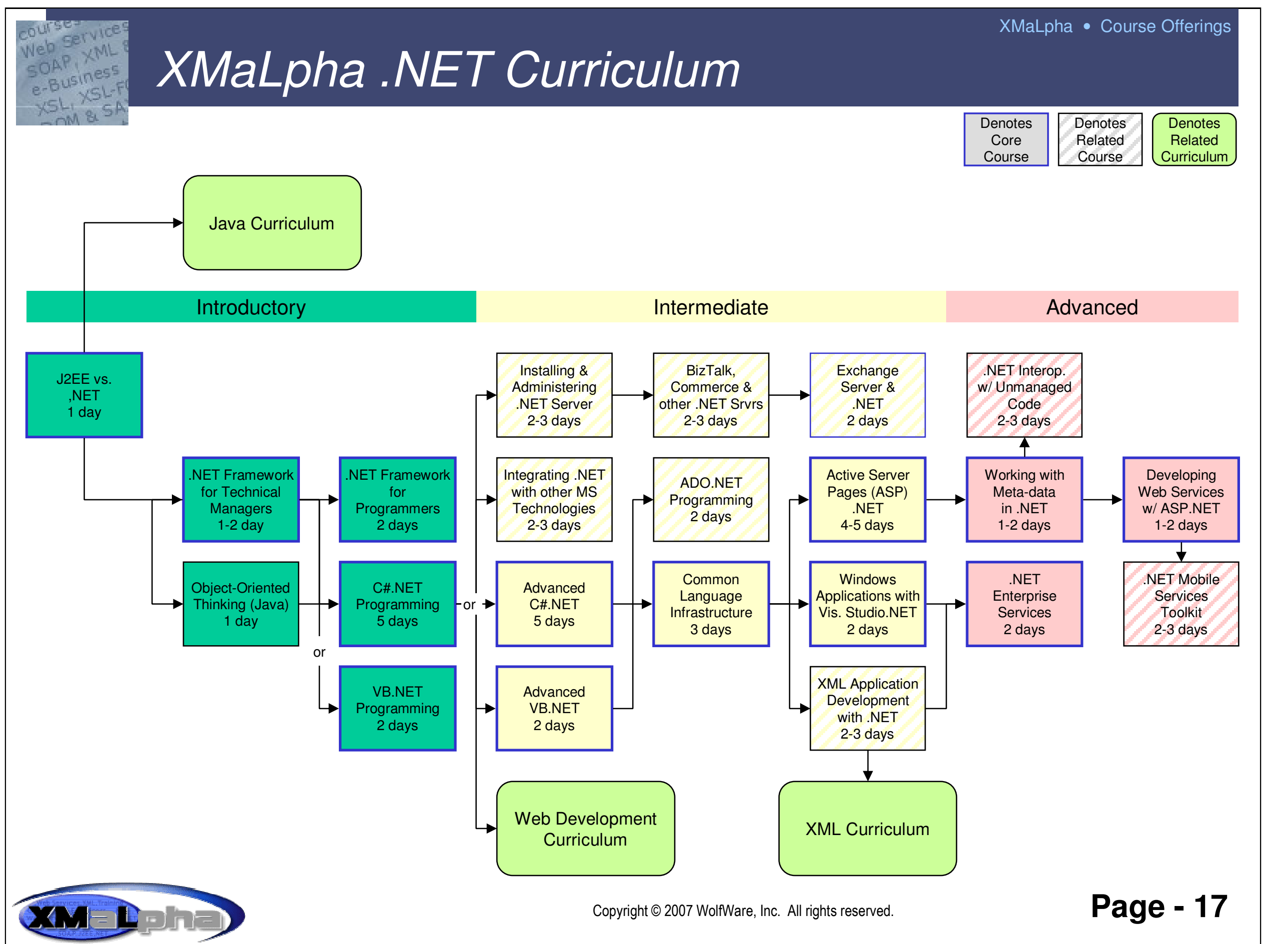


A place for your notes…



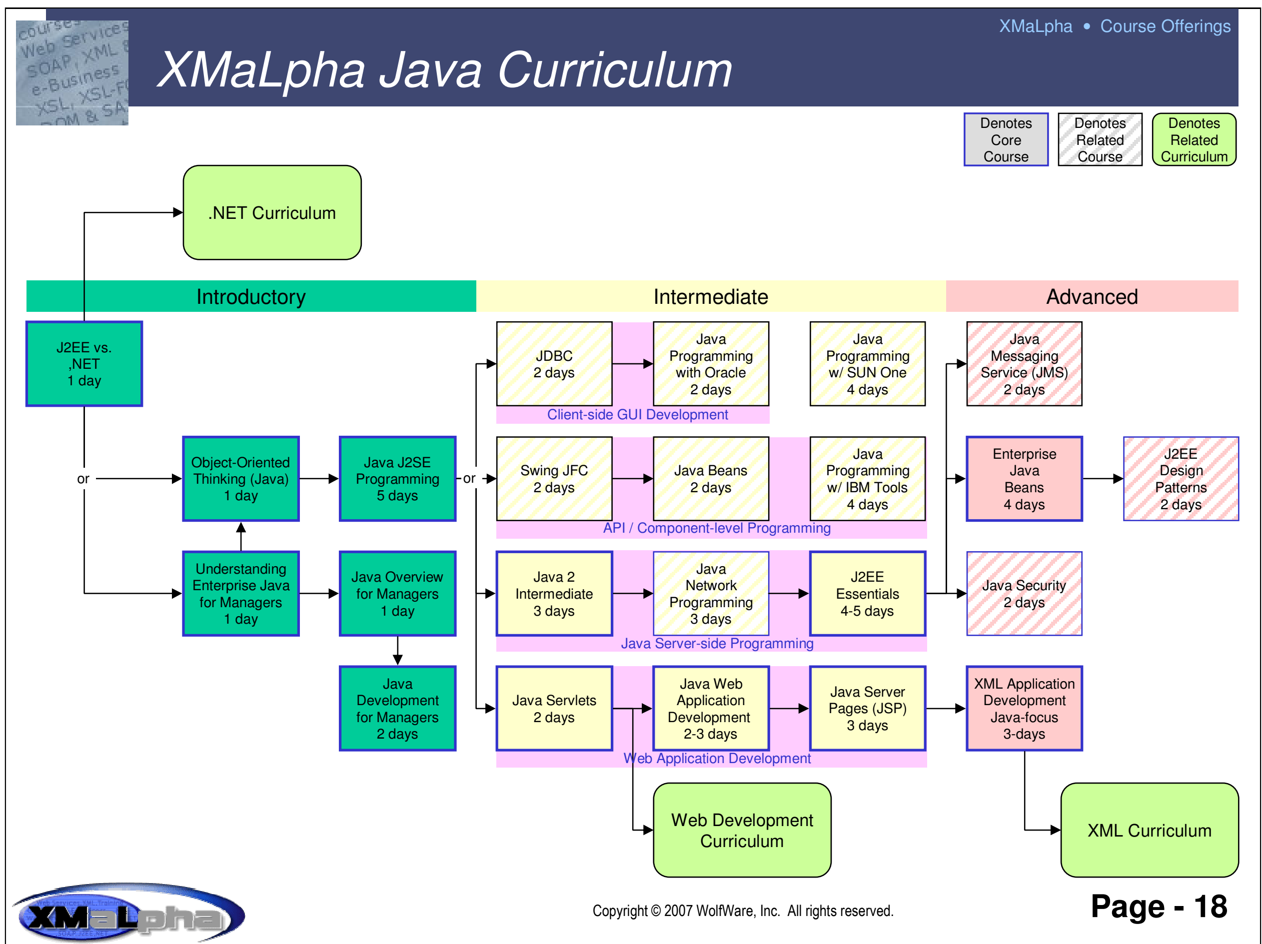
A place for your notes…

Copyright © 2007 WolfWare, Inc. All rights reserved.



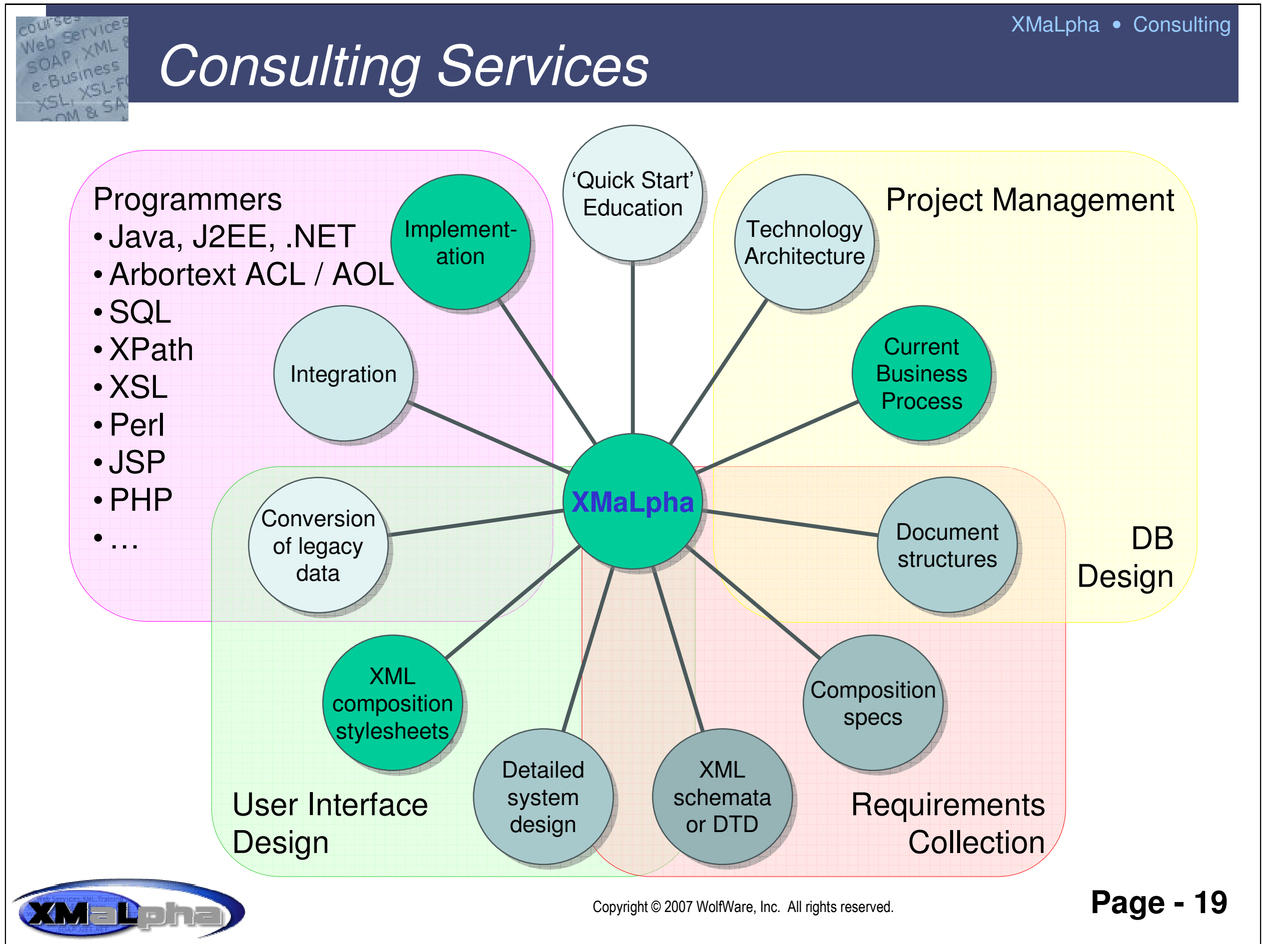
A place for your notes…

Copyright © 2007 WolfWare, Inc. All rights reserved.



A place for your notes…

Copyright © 2007 WolfWare, Inc. All rights reserved.



An independent perspective

A proven track record with extensive XML experience in the legislative environment

XMaLpha Technologies:

- Focus on design and analysis using our experience plus proven techniques
- Build architectural roadmaps and specifications
- Develop the core services using the selected tools and environments
- Design, plan, and deliver integration and implementation

Copyright © 2007 WolfWare, Inc. All rights reserved.