

Java Introduction

MSIS 531 – Spring 2006

Learning Goals

2

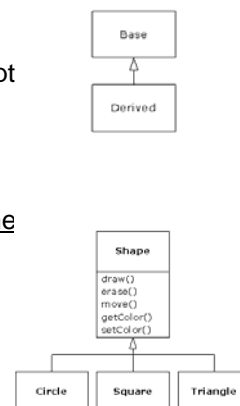
- Objects in Java
- Java Language Syntax
- Building some sample projects as we go
- Program control flow
- Using arrays
- Handling exceptions
- Goal is not to become a “Java architect”, but rather to use Java as a tool

Objects in Java

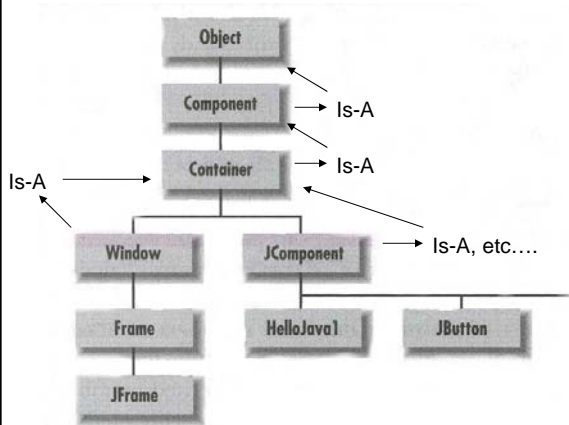
- In Java, (almost) everything is an object
 - A variable that can do things as well as store data
- Programs are groups of interacting objects
 - Interaction occurs by sending messages
- An object lives in memory, and is made of other objects
- Every object has a type
 - An object is an instance (in memory) of a class
- All objects of a particular type can receive the same messages
 - The list of messages constitutes the interface another object uses to interact with the object
 - An object can be thought of as a service provider

Inheritance

- Inheritance allows an object's interface to be reused and/or extended
 - Allows grouping of data, function by concept
- A derived class (subclass) inherits all data, code (methods) from its parent class (superclass)
 - In this case, the derived class is of the same type as the parent class
 - Can receive the same messages, and possibly additional ones
 - In this case, the subclass is said to extend the superclass



Class Relationships

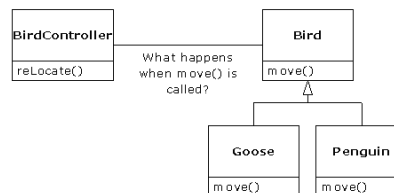


- Subclassing creates an “is-a” relationship
- From more general to more specific
- If we refer to a kind of object, we mean that type of object or any of its subclasses
- We will begin to use objects now, gain understanding of their structure as we go

Polymorphism

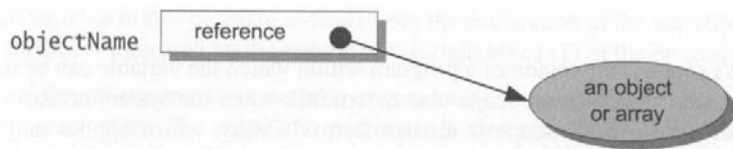
6

- Because of inheritance, we can send messages to a class of the parent type, and have subtype-specific behavior occur automatically
- In the example at right, a BirdController acts on a Bird, but the move() action is bird-specific
- As new subtypes are added, new implementations can be added transparently



Creating Objects

- All objects are manipulated using references
 - Consistent syntax for all object types
 - Sort of a “handle” that attaches to the object
 - Can’t do anything with the handle except pass messages to the object



Primitive Data Types

- The eight exceptions to the “everything is a class” rule:

int	4 bytes	-2 billion to 2 billion
short	2 bytes	-32,768 to 32,768
long	8 bytes	-9,223,372,036,854,775,808-..
byte	1 byte	-128 to 128
float	4 bytes	6-7 significant digits
double	8 bytes	15 significant digits
char	2 bytes	'a', 'b', etc.
boolean	1 byte	true, false

- All other Java data types are objects
- All primitive types have default values (false for boolean, 0 for numerics, null character for char)

Wrapper Classes

- Each primitive data type has a corresponding object
- The data is stored as an immutable field of the object
- Wrappers around the primitive types define useful constants:

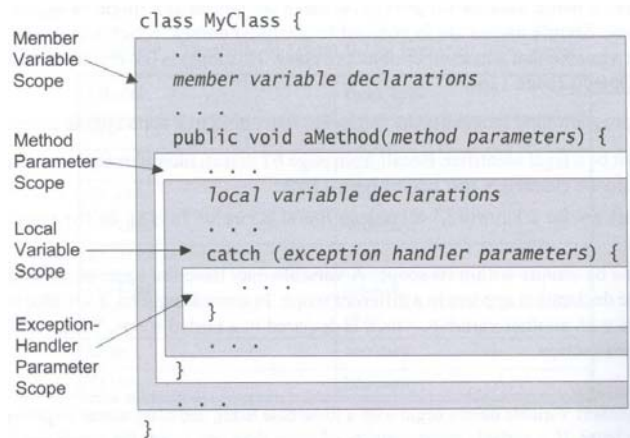
```
Integer.MAX_VALUE
Float.NEGATIVE_INFINITY
```

- They also include data conversion methods:

```
String value = "3.14e6";
double d =
    Double.parseDouble(value);
```

Primitive Data Type	Corresponding Object Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Scope



- Program region where a variable can be referred to by its simple name
- Also determines when memory allocated, deallocated (no need to manually free memory)
- Scope is not the same thing as visibility; more on this later
- Variables in “larger” scopes can be by “smaller” scopes

Java Classes

- Classes are the building blocks in Java
- Contain methods, variables, code, other classes
- Java recognizes a class by:

```
class MyClassName { class stuff }
```

- And we create a MyClassName object using:

```
MyClassName myClass = new MyClassName;
```

- The words “class” and “new” are Java keywords
- Note syntax: text in brackets, structure of assignment which instantiates a new class

Classes and Objects

- So if a class is a cookie cutter, an object is something like a cookie
- You may see the two terms used interchangeably in practice, adding to the confusion
- The class definition doesn't allocate memory, it is just a description (cookie cutter) for the objects (cookies) that will be produced
- Also note that class design is a programming issue (design time), but object instantiation happens at run time

Class Member Scope

- Scope determines where in a program we can use a name without qualification
- For data/code inside a class instance, we can refer to methods/variables directly
- From outside, must qualify the names:

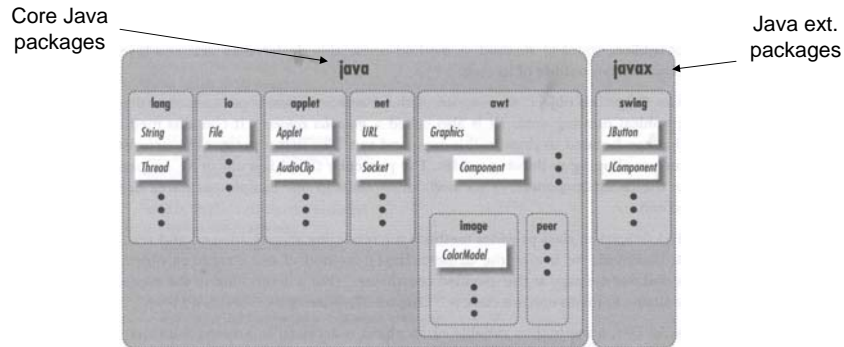
```
myDataElement = objectName.dataElement; // data
objectName.methodName(arg1, arg2, arg3); // method
```

Class Methods

- Methods always appear in class bodies
- Basic structure is as follows:


```
returnType methodName(/* args */) {
    // method body
}
```
- May declare local variables (variable within a method)
 - Local variables exist only for the duration of the method, then are garbage collected
- Must have a return type and fixed number of arguments
 - If nothing is returned, the return type is void
- When a local variable in a method has the same name as an instance variable, the local variable shadows (hides) the instance variable

Use of Packages



```
Public class HelloJava1 extends javax.swing.JComponent {...}
```

- A package is a set of Java classes, grouped by function
- We'll create and use our own classes later in the course

16

The import Statement

- To leverage code outside our own packages, we must make the environment aware of the classes we want to use
- To do this, we import the packages we need, removing need to qualify its members
- Can also just import a subset of a package

```
import java.util.ArrayList;      // just ArrayList class  
import java.util.*              // all java.util classes
```

Static Data Members

- Instance members are unique to each object—static members are shared
 - Use the **static** keyword to identify
- Within the class, no class/member reference required
- From outside, use the classname to qualify the reference
`SalesOrder.MAX_SALES_ORDER_ITEMS`
- Design issue: make sure the item really won't change

Static Methods

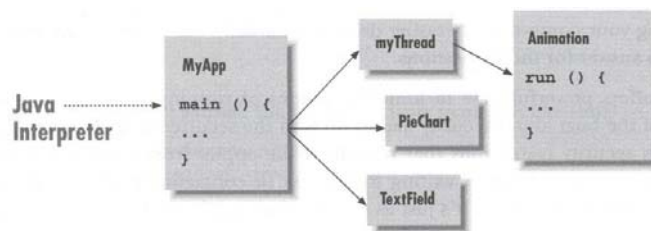
- As with static data members, use the **static** keyword to identify static methods
- These exist independent of a class instance
 - Significant benefit: these methods can be called without instantiating the class
- Useful for utility methods that perform useful work independent of any class instances
- Many JDK classes include static methods for programmer convenience

The main() Method

- The java interpreter must start somewhere...
- Think of main() as an entry point into a bytecode file
- For Clock, main() does the following...
 1. Potentially receives arguments passed as input (the String[] args entry on the first line)
 2. Creates a ClockView object
 3. Makes the clock visible with cv.setVisible
 4. Refreshes the display using a thread (subject for another day)
 5. Deals with any errors (exceptions) that may arise

Example: javaclock

An Executing Program



```
public static void main(String[] args) {...}
```

- Main() takes one argument, an array of strings
- One way for us to pass inputs to the program
- Execution completes when all child threads have completed

Building the Clock program

- For most of the class, we won't build new java objects; we'll try to reuse existing code
- General rule: whatever you need to do, someone else has already done it
- Value add is in combining things in new ways, and doing so quickly
- The clock example would obviously be reusable if you needed a clock elsewhere
- Java bytecode (.class) files can just be copied

The java Interpreter

- Implements the JVM
- Standalone, or embedded in another application (e.g. a browser)
- Loads classes, interprets byte code
- Interpreter runs until main and all its "child threads" have completed
- For a web server application, the JVM is called when on startup
 - The main() routine is not visible (or useful) to us as users of the application

The Class Path

- Put simply, “how Java finds things”
 - List of directories, files where classes may be found
 - Files are in class archive file format (.JAR, .ZIP)
- Similar to Windows or Unix PATH statement
- No need to include java.lang, java.net, etc.—the interpreter knows how to find them
- Class path searched in order
- If no CLASSPATH variable -> current dir only
- Most complex Java programs (e.g. Tomcat) set the classpath themselves, usually during startup

Java Language Syntax

- We’ve seen the high-level view
- Now on to the syntax of the Java language
- We’ll gloss over some aspects, focus on the basics:
 - Data types: type of data we can store
 - Statements: control flow, assignment, etc.
 - Expressions: building complex statements
 - Exceptions: error handling
 - Arrays: groups of items

Unicode

- Java characters, strings are stored in a standard format that allows representation of most languages
 - Significant shortcoming of some other languages
- 16-bit format (65536 unique characters)
- We'll use the ISO8859-1 (Latin-I) encoding, but support for other languages is built in
- <http://www.unicode.org>

Code Comments

- Comments are one of the easiest ways to reduce application lifecycle costs
- Line comments (one line max):

```
// this is a comment
// so is this
```
- Block comments (multiple lines, can't nest):

```
/* hello there */
```
- Javadoc comments (more on this later):

```
/** I'm going into the docs */
```

Data Types

- Types associate data elements (variables, constants) with storage (memory, disk)
- All elements have a type known at compile time (static typing)
- In Java, type information maintained at runtime—type safety assured
- Two categories:
 - Primitive types (8: Boolean, Char, Int Float, etc.)
 - Reference types (everything else)

Naming Data Elements

- Sequence of Unicode characters, starting with a letter
- Can't use Java keywords, 'true', 'false', or 'null' as a name
- Unique within its scope
- May be qualified (OtherPkg.someVar)
- Conventions:
 - Variable: 1st letter lower case, all subsequent words start with an upper case letter (myVar)
 - Constant: All caps, “_” between words (MY_CON)

Declaring/Initializing Variables

- Declare variables in methods (local) or classes (instance)
- Can declare/initialize at same time
- Instance variables are initialized to default values (0, null, false) by default
- Default type for literals: int for integers, double for floating-point

Using Strings

- Strings are objects, and thus reference types
- The Java compiler does some tricks for you
 - Literal string values ("abc") are turned into String objects automatically
 - The "+" operator is overloaded for strings to allow string concatenation •
- Strings are immutable: they can't change once initialized
- Two ways to create string

```
String s = "abcd";           // auto-object creation
String s = new String("abcd"); // explicit creation
```

Operators

- An operator performs a function (e.g. addition) on one, two, or three operands
- Operators may appear before, between, or after operands
 - Prefix, infix, or postfix notation
- Operand categories:
 - Arithmetic
 - Relational/conditional
 - Shift/bitwise
 - Assignment
 - Other

Digression: Servlets

- For most of the examples going forward in the class, we'll use a web browser as our UI
- We need to introduce servlets into the mix to allow this
- NetBeans makes it very easy to create/run them
- For now we'll add and use servlets in the javaintro project, with the option to create new projects later
- To create a new servlet, right-click on the javaintro project, select New->Servlet. Take the defaults. Uncomment the TODO section to display something. Right click NewServlet->Run File. Done!

Arithmetic Operators

- +, -, *, /, % (modulus)
- Use '%' to compute remainder
- Regular arithmetic operators are infix
op1 * op2 (operator in the middle)
myVar1 + myVar2
- If data types of operands are different, the result is promoted to the larger type
- Shortcut operators (op++, ++op, op--, --op) to increment, decrement by 1
 - These are prefix/postfix operators

Example: MathOps.java

Relational Operators

- Compares two values, returns a boolean result indicating their relationship
- >, >=, <, <=, == !=
 - Note “equal” is two equal signs
- Also have conditional operators: &&, ||, !
 - op1 && op2 true if both ops are true
 - op1 || op2 true if either op is true
 - !op1 true if op1 is false, true otherwise
- We're skipping the shift/bitwise operators

Assignment Operators

- Assignment: store a result in a variable
 - `myInt = 27`
 - Remember data types must match or be promotable, or it's a compile-time error
- Java gives shortcuts for arithmetic assignment
 - `+=`, `-=`, `*=`, `/=`, `%=`
 - `myInt *= 4` (same as `myInt = myInt * 4`)

Other Operators

- ?: (shortcut if/else)
 - `myInt = (a > b) ? a : b`
- [] (used to declare arrays)
- . (dot: used to qualify names)
- (params) (comma-separated list of parameters)
- (type) (convert a value to the specified type)
- new (create a new object/array)
- instanceof (true if op1 is an op2)

Statements & Expressions

- Java statements appear in methods and classes
 - Describe all activities of a Java program
 - Declarations and assignments are statements
- Expressions describe values
 - Method calls, object creation, math expressions
 - Variables, operators, method calls that evaluate to a single value
- Statements, expressions must be structured according to the Java language spec

Expression Evaluation

- Expression result type depends on type of data (literals, variables, method results) contained in it
- Compound expressions possible
 - $x * y * z$
 - $x + y / 7$
- In 2nd expression above, what is the result?
 - Need to consider operator precedence
 - By default, it's $x + (y / 7)$
 - Use parentheses (i.e. $(x + y) / 7$) if alternate precedence is desired

Java Statements

- Statement roughly similar to a statement in a natural language
 - Forms a complete unit of execution
 - Terminated with a semicolon
- Examples:

```

aValue = 8933.234;           // assignment
aValue++;                   // increment
System.out.println(aValue); // method
MyObj newObj = new MyObj(); // object
double aValue = 8933.234;   // declaration

```

Statement Blocks

- Statements, expressions appear in code blocks
 - Zero or more statements enclosed by “{“ and “}”
- Methods are code blocks that take inputs (arguments) and can be referred to by name
- Variables are limited in scope to the enclosing code block
- Most common use of code blocks is to group statements to perform conditional or iterative processing

Control Flow Statements

- Without control flow, statements execute top to bottom
- Control flow statements alter this execution pattern

Statement Type	Keywords
Looping	while, do-while, for
Decision making	if-else, switch-case
Exception handling	try-catch-finally, throw
Branching	break, continue, label:, return

The while and do-while statements

```
while (expression) {      do {
    statement(s)          statement(s)
}                          } while (expression);
```

- While (expression) is true, statement(s) are executed
- 'while' tests at the top, 'do-while' at the bottom
 - 'do-while' statement(s) execute at least once; 'while' statements may not execute
- Note semicolon at end of do-while statement

The for statement

```
for (initialization; termination; increment) {
    statement(s)
}
```

- Initialization, termination, and increment are all expressions
- Initialization used to set values used in the loop
- Termination decides when loop ends
- Increment modifies loop parameters

```
for (int i = 1; i <= 5 ; i++) {
    System.out.println(i + " squared is " + i*i );
}
```

The if-else statement

```
if (expression) {
    statement(s)
} [[ else if (expression) {
    statement(s)
} ] else {
    statement(s)
} ]
```

- If (expression) is true, execute statements
- The 'else' clause (in '[']) is optional—if included, it's executed if expression is false
- Can have more than one 'else if' clause, but only one 'else' clause

The switch statement

```
switch (expression) {
  case a: statement(s); break;
  case b: statement(s); break;
  ...
  default: statement(s); break;
}
```

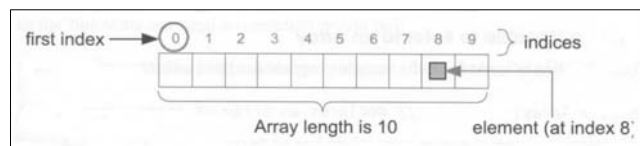
- Choose one option from cases (or default)
- Expression must evaluate to an integer
- Values provided to cases must be unique
- Default case used if no other matches found
- Use break statement to avoid “falling through”
- Could use if statement instead—let readability decide

Branching Statements

Keyword	Effect
break [label];	unlabeled: break out of enclosing loop labeled: break out of labeled statement
continue [label];	unlabeled: skip remainder of loop iteration labeled: skip current iteration of labeled loop
label:	label a statement for use by break or continue
return [value];	return from a method, possibly passing a value

Arrays

- Container for groups of objects of the same type
- Array lengths are static (known at compile time)
- Array element: one of the values
 - Indexed from 0 to array.length - 1
- Arrays are objects; use **new** to create them
- Arrays may hold primitive or reference types



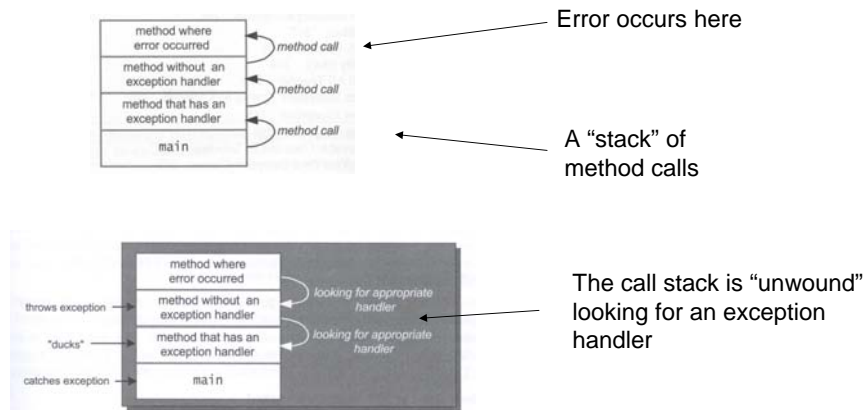
Using Arrays

<code>int[] myArray;</code>	declare an array
<code>myArray = new int[25];</code>	initialize myIntArray
<code>int[] myArray2 = new int[15];</code>	declare, initialize at once
<code>JLabel [] myJ = new JLabel[10];</code>	array of JLabel refs
<code>myJ[0] = new JLabel("Java Rules");</code>	add a JLabel object
<code>x = myArray[3];</code>	4 th element in myIntArray
<code>y = myArray2.length;</code>	$y \leftarrow 15$
<code>String[] test = { "a", "b", "c" };</code>	shortcut declare/initialize
<code>myJ[10].getText();</code>	ERROR
<code>System.arraycopy(myArray, 0, myArray2, 0, 5)</code>	First 5 elements of myArray → myArray2

Handling Exceptions

- Java provides consistent, methodical exception handling
 - Exception: unusual or error condition
 - Interrupts normal program flow
- Control is transferred (“thrown”) to exception handling code
 - Runtime system finds handler code
 - Note that this may be somewhere further up the call stack
- Exceptions are objects (yikes!)

The Call Stack



Exception Advantages

- Separate error-handling code from regular code
 - Details of what to do when problem arises are managed separately
- Call stack error propagation
 - Any method in the chain can handle, or ignore, a particular type of exception
- Grouping of exceptions
 - Exceptions are objects, and are categorized normally in the class hierarchy
 - Extend an object for specific error handling

Catching or Specifying

- For all checked exceptions, a method must either:
 1. Catch the exception by providing an exception handler
 2. Specify that the exception can be thrown
- Checked exception: normal application-level error
 - File not found, device not available
 - Part of object's contract with its users
- Unchecked exception: runtime error

Handling Exceptions

```
try {  
    statement(s)  
} catch (specificexception name) {  
    statement(s)  
} catch (moregeneralexception name) {  
    statement(s)}  
finally {  
    statement(s)  
}
```

- Each catch block handles a specific type of exception
- Catch blocks are evaluated in order—so code from the specific to the general