An Overview of Java

Jeff Offutt
Roger Alexander

http://www.ise.gmu.edu/~offutt/

SWE 432
Design and Implementation of Software for the Web

Java Introduction

- **Developed by Sun Microsystems**
- **Originally intended for development of embedded software for consumer electronics**
- **Re-designed for programming on the Internet**
- **Based on Objective C**

Part I - Java Design Goals

Java Design Goals

- **Object-oriented**
- **Architecture neutral**
- **Portable**
- **Robust**
- **Multi-threaded**
- **Distributed**

Java vs. C++

- **Java is interpreted, not compiled like C++**
  - **Not quite! The Java VM interprets byte-code, not Java**
  - **Java is *compiled* directly to byte-code**
- **Memory allocation in Java**
- **No direct multiple inheritance**
  - **Achieved through** interfaces
- **Type parameterization simulated with inheritance and objects**
- **Java pointers are typed and restricted**
  - **Called** references

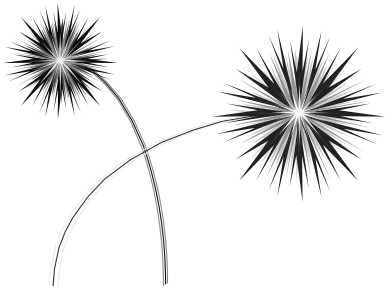
Major Language Features

- **Classes and Interfaces**
- **Inheritance and Polymorphism**
- **Exceptions**
- **Garbage Collection**
- **Multi-threading**

Missing Language Features

- **Multiple inheritance**
- **Assertions**
- **Enumerated types**
- **Call-by-reference parameter passing**
- **Default parameter passing**

Part II - Java Basics



Lexical Elements

- **Identifiers: Sequence of letters (Unicode), numbers, underscore ("_") and Dollar Sign ("$").**
  - **First character must be a letter, $, or _**
  - **Case is significant**
- **Reserved Words: Keywords cannot be redefined by programmer**
- **Operators: Tokens that perform operations upon operands of various types**

49 Java Reserved Words

abstract	**class**	**extends**	**implements**
boolean	**const**	**false**	**import**
break	**continue**	**final**	**instanceof**
byte	**default**	**finally**	**int**
case	**do**	**float**	**interface**
catch	**double**	**for**	**long**
char	**else**	**if**	**native**

49 Java Reserved Words (cont'd)

new	**short**	**throws**
null	**static**	**transient**
package	**super**	**true**
private	**switch**	**try**
protected	**synchronized**	**void**
public	**this**	**volatile**
return	**throw**	**while**

Java Operator Precedence

Postfix operators	**[] . (*params*) *expr*++ *expr*--**
Unary operators	**++*expr* --expr –*expr* +*expr* -*expr* ~ !**
Creation or cast	**new (*type*) *expr***
Multiplicative	*** / %**
Additive	**+ -**
Shift	**<< >> >>>**
Relational	**< > >= <= instanceof**

Java Operator Precedence (cont'd)

Equality	**== !=**
Bitwise AND	**&**
Bitwise Exclusive XOR	^
Bitwise Inclusive OR	|
Logical AND	**&&**
Logical OR	||
Conditional	**?:**
Assignment	**= += -= *= /= %=**
	>>= <<= >>>= &= ^= |=

Variables, Methods, and Classes

- **All variables must be declared before use**
- **They must be used in a manner consistent with their type**
- **Classes combine data objects and functions to provide <u>Abstract Data Types</u>**

Definitions and Declarations

- **Definitions: Places where a variable or function is created or assigned storage**
- **Declarations: Places where the nature of a variable is stated, but no storage is allocated**
- **Allocations: Places where storage is allocated**
- **Variables and functions must be declared for each function that wishes to access them**

Statements

- **Every statement ends in a ';'**
- **A compound statement is a sequence of statements enclosed by braces:**

```
'{'
    [ decl-list ]
    [ stmt-list ]
'}'
```

Comments

Java supports three different comment styles

- **Single-line comments:**

```
// This comment starts with // and ends at the end of the line.
```

- **Traditional C bracketed comments:**

```
/*
   This is a multi-line Java comment.
   No nesting allowed!
*/
```

- **JavaDoc comments (later)**

A well commented program is a sign of a good programmer

Boolean Expressions

- **All logical expressions are of type** boolean
- **Java uses <u>short circuit evaluation</u>**

<u>Examples</u>

X < 10 Z <= X+10 Y > 20

Y = (-5) X >= X+Y

(X < Y) && (Z == 10) (Z == 10) || (Y >= 5)

! (X > 0)

Short-Circuit Evaluation

- **Java uses short circuit evaluation: evaluation stops as soon as the final result can be determined.**
- A && B
  If A is false, produce false.
  Otherwise, evaluate B.
- A || B
  If A is true, produce true.
  Otherwise, evaluate B.

- (X != 0 && Y/X > 2) // Avoid division by zero.

- (X > 100) || IsPrime (X)) // Avoid expensive function call.

Conditional Branching

- if/else: **General method for selecting an action**

- switch: **Allows efficient selection among a finite set of alternatives**

if **Statement**

```
if (<cond>)

    <stmt1>
  [ else
    <stmt>]
```

```
if (isalpha (c))
{
    if (isupper (c)
        return UPPER;
    else
        return LOWER;
}
else if (isdigit (c))
    return DIGIT;
else if (isprint (c)
    return PRINTABLE;
else
    return UNPRINTABLE;
```


switch **Statement**

- **General form:**

  switch (<expr>) { <cases> }

- **switch only works for constants and scalar variables (integers and characters)**

- **Java uses "fall-through semantics''**

  must use a break **to terminate each case**

switch **Statement**

```
int symbol;

switch (symbol)
{
    case CONST    : print (“constant”); break;
    case SCALAR   : print (“scalar”); break;
    case RECORD : print (“record”); break;
    default             : print (“array or string”); break;
}
```


Iteration Statements

Java has four iteration constructs:

- for**: test at loop top**
- while**: test at loop top**
- do/while**: test at loop bottom**
- **Recursion**

for **Loop**

- **General form:**

  for (<initialize>; <exit test>; <increment>)

  <stmt>
- **Initialization, exit test, and increment are all in one construct**
- **All three loop header sections are optional, and may contain arbitrary expressions**
- **Variables may be declared in the** <initialize> **section**

for **Loop (cont'd)**

Examples:

```
// standard for loop
for (int i=a;  i <= b; i++)

// infinite loop
for ( ; ; )

// compute n!
for (int i = n; n > 1 ; n-- )
    i = i*(n-1);
```

while **Loop**

- **General form:**

```
while (<condition>)
    <stmt>
```

- **Repeats stmt as long as condition is TRUE.**

while **loop (cont'd)**

Examples:

```
// standard while loop
 while (X != 0)

 // infinite loop
 while (true)
```

```
// compute n!
i = n;
while (n >= 0)
{
    n--;
    i = i*n;
}
```

do .. while **Loop**

- **General form:**
  do <stmt> while (<condition>);
- **Always executes body at least once.**
- **Test is at the bottom.**
- **Less common than** for **and** while**.**

break **Statement**

- **Can be used to exit any code block**
- **Allows early termination of loops**
- **Syntax:**

  labelname:
  … loop body …
  break [label];
- Label **allows you to terminate an arbitrary block of code**

break **Statement (cont'd)**

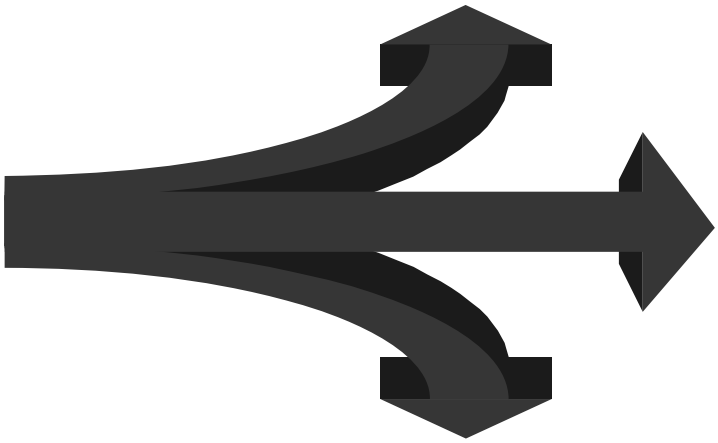
```
public boolean workOnFlag (float flag, int[][] Matrix)
{
     int x, y;
     boolean found = false;
search:
     for ( y = 0; y < Matrix.length; y++ )
          for ( x = 0; x < Matrix[y].length; ++x )
          {
               if ( Matrix[y][x] == flag )
               {
                    found = true;
                    break search;
               }
          }
     return (found);
}
```


continue **Statement**

- **Skips to next iteration of loops**
- **Can specify a label of an enclosing loop**
- **Syntax:**

```
while (true)
{
     … loop body …
     if (<expr>)
          continue;
}
```

Exception Handling



Errors and Exceptions

- **During execution, "stuff" happens**
- **Many kinds of errors**
- **Many kinds of exceptional situations**
- **Varying degrees of severity**
- **Code to handle errors adds complexity**
  - **Often, more code required to handle errors than the original function**

Java Exceptions

- **Language feature provides clean check for exceptions**
- **Mechanism to signal errors directly**
- **Define new exceptional conditions for methods**
- **Automatically checked by run-time system**

Java Exceptions (cont'd)

- **Thrown when an unexpected condition occurs**
- **Caught by a method who can handle exception**
- **Default exception handler handles those not caught by a programmer-supplied method**

Handling Built-in Exceptions

```
try
{   in_str   = in.readLine ();
    answer = Integer.parseInt (in_str);
}
catch (IOException e)
{  // JDK requires this exception to be caught.
    System.err.println ("Could not read input.");
    answer = 0;
}  catch (NumberFormatException e)
{   System.err.println ("Entry must be numeric");
    answer = 0;
}
```

More later . . .

Example Java Program

```
/**
 *  A class for reading a sequence of numbers and report:
 *    - count of positive numbers
 *    - count of negative numbers
 *    - count of zeroes
 *    - sum of positive numbers
 *    - sum of negative numbers
 *
 *  @author  Jeff Offutt
 */

import java.io.*;
public class Count
{ // User enters the length of sequence followed by numbers
public static void main (String args[] ) throws IOException, NumberFormatException
{
    int num_pos = 0, num_neg = 0, num_zero = 0;
    int sum_pos = 0, sum_neg = 0;
    int length, cur_num;
    BufferedReader in = new BufferedReader (new InputStreamReader
                                (System.in));
```

Example Java Program (cont'd)

```
String in_str;

   System.out.print ("How many numbers? ");
   in_str = in.readLine ();
   length = Integer.parseInt (in_str);

// Loop to read and count

for (int i=1; i <= length; i++)  // i  is declared in for loop
{
   System.out.print ("Enter value: ");
   in_str = in.readLine ();
   cur_num = Integer.parseInt (in_str);
   if (cur_num > 0)
   {
      num_pos = num_pos+1;
      sum_pos = sum_pos + cur_num;
   }
```


Example Java Program (cont'd)

```
   else if (cur_num < 0)
   {   // Shorthand operators
       num_pos++;
       sum_pos += cur_num;
   }
   else   // cur_num==0
       num_zero++;
} // end for loop

   // Now print the results.

   System.out.println ("Count of positive numbers: " + num_pos);
   System.out.println ("Count of negative numbers: " +  num_neg);
   System.out.println ("Count of zeroes: " + num_pos);
   System.out.println ("Sum of positive numbers: " + sum_pos);
   System.out.println ("Sum of negative numbers: " + sum_neg);
}  // end of main
}  // end of Count
```

Primitive Types

- boolean - **either** true **or** false
- char - **16-bit Unicode**
- byte - **8-bit signed, -128..127**
- short - **16-bit signed, -32,768..32,767**
- int - **32-bit signed**
- long - **64-bit**
- float - **32-bit, 6 or 7 significant bits**
- double - **64-bit, 15 significant bits**

Initial Values

- **A variable (sometimes called a *field*) can be initialized in its declaration:**

```
double pi = 3.14159;
```

- **Java only assigns default values to class instance variables (more later)**

Constants

- **Variables can be declared as constant using** static **and** final **keywords:**

```
static final double pi = 3.14159
```

- **Examples**
  - 37
  - 045 // Octal, base 8
  - 0x25 // Hex, base 16
  - '5'
  - "Steffi"

Type Casting

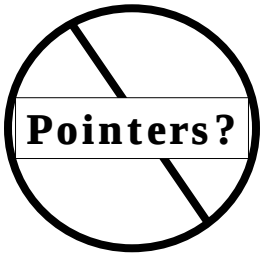
Expressions can be "cast" or converted to a different type:

```
int int_var;
double doub_var;
int_var = doub_var; // Error!

int_var = (int) doub_var;  // Correct!
doub_var = (double) int_var * 37;
```

Checked at compile-time and run-time

Java Pointers



Not really …

- **Java pointers are strongly typed and called references**
- **They are much more like Ada pointers than C/C++ pointers**
- **They are based on C++ references**
- **Typed pointers help programmers avoid many problems**

Java Pointer References

- **No math operations**
- **Assignment (=), equality testing (==, !=)**
- null **– does not point to anything**
- **No dereferencing necessary (***p, p->**)**
- **Access elements with the dot operator**
  - **Variables**
  - **Methods**
  - p.x

Java References (Cont'd)

- **An object is a non-primitive type**
- **A reference variable is a name for an object**
- **Reference variables hold the memory locations of the object**
- **Objects have:**
  - **Variables (fields)**
  - **Methods (functions)**

Dot Operator

The "*dot operator*" is used to select variables and methods within an object:

```
theCircle.area();   // Call a method
theCircle.radius;  // Access a variable
```

Using References

```
Watch w;   // declares w to be a reference to
                // objects of type Watch
w = new Watch ();  // Allocates space for w
w.SetTime (5, 15, PM);  // Calls method SetTime
```

When we call "w.SetTime ()**", we say**

"The method SetTime **executes in the context of the Watch instance w".**

In C, this would be:

```
SetTime (w, 5, 15, PM);
```


Garbage Collection

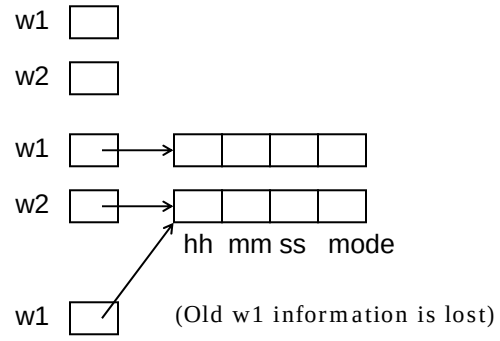
- **Java automatically recovers space that is not referenced**
- **This is convenient, but it does add hidden execution time**

Reference Assignments

Watch w1, w2;

w1 = new Watch();
w2 = new Watch();

w1 = w2;



The Watch object should provide a <u>copy</u> operation, called "clone ()**" in Java**

Reference Comparison

Watch w1, w2, w3;

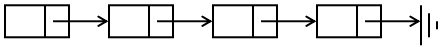
w1 = new Watch();
w2 = new Watch();

w1.SetTime (10, 41, AM);
w2.SetTime (10, 41, AM);
w3 = w1;
(w1 == w3) is True
(w1 == w2) is False

To compare two objects, the convention is to define a method called "equals ()**"**

Recursive Data Structures (Lists)

```
class Node
{   int Value;
    Node Next;
}
```



```
Node list, cur;
list = new Node ();
list.Value = 1;
cur = list;
for (int i = 2; i <= 5, i++)
{  // Initialize list of 5 integers.
    cur.Next = new Node ();
    cur = cur.Next;
    cur.Value = i;
}
```

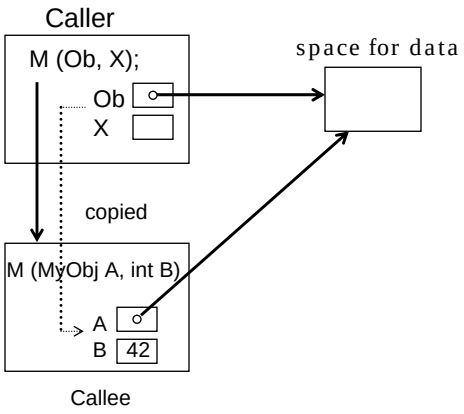

Subprograms - Methods

Access Type Name (params)

- **Parameters are call-by-value**
- **To get call-by-reference, pass a reference**
  - **Does not work with scalar variables**
- **References to objects are also passed by value, not the object itself**
- **Example:**

```
public static int Max (int a, int b)
{
    if (a >= b)
        return (a);
    else
        return (b);
}
```

Methods - Parameters



Method Name Overloading

Overloading is an abstraction mechanism that allows the same method name to be used with different parameter lists

They are differentiated by their *signatures*, or parameter lists

```
public static int Max (int a, int b);
public static int Max (int a, int b, int c);

max1 = Max (m, n, o);
max2 = Max (x, y);
```

Strings

- **Strings are provided by a standard object, not as a built-in type**
- **Example:**
  String Name = “Steffi”;
- **String variables (“**Name**”) are references to objects**

String Operations

- **Concatenation: “+”**
  String kid = “Steffi” + “ Offutt”;
- **Length:**
  int NameLength = kid.length();
- ***Comparison:***
  string1.equals (string2);

What's an Object?

- **An object is a set of data members (i.e. variables, called "*fields*") and operations on those data members ("*methods*")**
- **Objects are implemented with the Class construct**
- **More later ...**

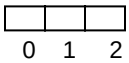
Arrays

Java arrays are objects with operators, but they look very much like C or Pascal arrays

```
int [] my_array;        // declaration, no memory
my_array = new int [10];  // 10 elements
```

- **Java arrays always start at 0!**

```
new int [3]
```



A [3] is an error.

- **Declaration, allocation, and initialization can be done at once:**

```
int [] primes = {1, 3, 5, 7, 11, 13, 17, 19};
```

Using Arrays

- **Arrays are indexed by integers**
- **Java includes bounds checking on arrays**

```
// “length” operator gives number of  elements
for (int i = 0; i < primes.length; i++)
      System.out.println (primes [i]);

pnew = primes;    // Does not copy!
```


Making Arrays Bigger

```
int [] temp;

temp = primes;
primes = new int [temp.length*2];  // double size
for (int i=0; i < temp.length; i++)    // copy
     primes [i] = temp[i];
```

Multi-Dimensional Arrays

```
int [] [] Chess;

Chess = new int [8][8];

for (int i=0; i < 8; i++)
     for (int j=0;  j < 8; j++)
          Chess [i][j] = 0;
```


Handling Command Line Arguments

- **Command line arguments : inputs provided to the program when the program is run**
- **Passed as an array of strings to main**
- **Example:**

```
java   WhoIsHe  Michael Jordan

public static void main (String [] args)
{
    for (int i=0; i < args.length; i++)
         System.out.print (args [i]);
}
```

args [0] == “Michael”
args [1] == “Jordan”

Using Command Line Arguments

- **Assume:** java TCGen -d infile
- **Use** -d **to set a "debug" flag for printing debugging information**
- infile **is the name of the file to open**

Input / Output

- **Input and output is performed with a standard package - java.io**

  import java.io.*;

- **There are three standard I/O "streams":**
  - **Standard input:** System.in
  - **Standard output:** System.out
  - **Standard error:** System.err

Input / Output Examples

System.out.println (“output message”);
System.out.println (37);   // numbers converted to strings

- **Reading an integer**:

```
BufferedReader in = new BufferedReader (new
   InputStreamReader (System.in));
int x;
String in_line;
in_line = in.ReadLine ();
x = Integer.parseInt (in_line);
```


Input / Output Examples

Reading multiple integers per line
E.g.: 37 36 7 4
Use StringTokenizer*:*

```
Buffered Reader in = new BufferedReader (new InputStreamReader
                        (System.in));
int  x;
String in_line;
StringTokenizer  in_str;
int  age_dad,  age_mom,  age_kid1,  age_kid2;
in_line = in.readLine();
in_str  = new StringTokenizer (in_line);
age_dad   =  Integer.ParseInt (in_str.NextToken());
age_mom  =  Integer.ParseInt (in_str.NextToken());
age_kid1  =  Integer.ParseInt (in_str.NextToken());
age_kid2  =  Integer.ParseInt (in_str.NextToken());
```

Input / Output Examples

Reading from files:

```
String fname, in_line;
FileReader my_file;
BufferedReader in_file = null;

fname  = args [0];
my_file = new FileReader (fname);
in_file   = new BufferedReader (my_file);
while ((in_line = in_file.readLine()) != null)
        System.out.println (in_line);

my_file.close ();
```