CS193J: Programming in JAVA

January 11, 1999

Lecture #2

Admin

- http://cse.stanford.edu/class/cs193j
  - Grab lecture slides, previous handouts, etc.
- Need to register with Webclass by this Wednesday
- This Friday's section will be usurped for lecture
  - To avoid slow start from MLK holiday
- Textbook
  - Can buy on-line or from Stacey's, Computer Literacy, etc. to avoid delay at Bookstore

Language goals moving C to Java

- Simplify where possible
- Improve robustness, remove unsafe language loopholes
- Guarantee portability
- Add facilities for data abstraction and encapsulation

What does Java look like?

```
/* A simple, but complete Java program. */

public class Test
{
   public static void main(String args[])
   {
      for (int i=0; i<10; i++)
         System.out.println("Hello world!");
   }
}
```

- Syntax familiar from C, but definitely some surprises: class? String? System.out.println?

Java syntax

- Similar to C/C++
- Case-sensitive
- Semi-colon is line terminator
- Curly braces used for block structure
- /* */ used for comments, also // will comment to end of line
- Function parameters enclosed in parentheses and separated by commas

Variable types

- Primitive types are fully specified
  - char (16 bit, UNICODE)
  - short (16 bit signed)
  - int (32 bit signed)
  - long (64 bit signed)
  - float (32 bit IEEE)
  - double (64 bit IEEE)
- Two additional primitive types
  - byte (8 bit signed)
  - boolean (1 bit, true/false)
- Everything else will be an "object"
  - No enums, structs, unions, pointers, typedef, etc.
- (No sizeof operator since not needed)

Booleans

- Real boolean type, true and false are keywords
- Does not consider anything non-zero to be "true"
- Type checking enforced for boolean expressions
  - `while (num)` //not allowed, needs to be num != 0
  - `if (num = 3)` //helps avoid this common mistake...

Variables declared anywhere

- Can declare anywhere, not just at opening of block

```
for (int i = 0; i < 20; i++) {
   System.out.println("Hi");
   char ch = 'A';
}
double pi = 3.14159;
```

  - Variable remains in scope until enclosing block is exited
- Adds convenience and readability
- Encourages initialization at time of declaration
  - Java compilers require initialization before use (only warning in C)

Type narrowing and widening

- C freely assigns between numeric types, but Java requires cast for narrowing conversions
- "Widening" goes from less to more rich type
  - Always okay, doesn't require cast
    e.g. assigning short to int, int to float, etc.
- "Narrowing" goes from more to less rich type
  - Possible loss of information, so cast is required
    e.g. assigning short to byte, double to float
- Parameter passing is same as assignment
- Cannot cast anything to boolean
  - Not okay: `boolean b = (boolean)num;`
    Use instead: `boolean b = (num != 0);`

Arithmetic/logical operations

- Arithmetic operators
  - Binary operators: + - * /%
  - Unary-assignment form: += -= *= /= %=
  - Shorthand increment/decrement: ++ --
- Relational operators
  - All evaluate to boolean: == != < <= > >=
- Logical operations
  - Combine booleans: && || !
- Same precedence and associativity as C
  - Can be overcome with parentheses like usual
  - Order of sub-expression evaluation is mandated left-to-right
- User cannot "overload" operators
  - Although + is overloaded to do string concatenation...

Type promotion

- In arithmetic expressions involving mixed type, all operands promoted to richest type

```
short s; int i; double d;
System.out.print(s+i+d); //result is double
```

- Calculations involving byte, short, char are promoted to int, result may need cast

```
short s = 4, t = 6;
s = (short)(s + t);  //cast required
```

- Cast not needed for shorthand operators

```
s += t;
```

Behavior of arithmetic

- Integers wrap around when range exceeded
  - e.g. adding two large integers overflows to negative result
  - No underflow or overflow reported
- Integer divide or mod by zero raises error
  - FatalArithmeticException thrown
- Floating point arithmetic follows IEEE rules
  - Overflow, divide by zero produce ±INFINITY
  - Underflow goes to zero
  - Invalid expressions (0/0, INF - INF) produce NaN
  - No exceptions thrown

Control flow

- Basically same as C
  - `if (boolean-expr)`
    `statement; (has optional else)`
  - `for (expr; boolean-expr; expr)`
    `statement;`
  - `while (boolean-expr)`
    `statement;  (do-while variant also)`
  - `switch (integer-expr)`
    `case constant: statement; break;`
- break/continue extended to allow break out of nested loops using labels
- No explicit goto
  - (although keyword is reserved...)

Defining functions

- Parameter syntax same as in C
- Return statement also unchanged
  - Compiler requires non-void function to have return statement
- Factorial example

```
long factorial(int n)
{
   long result = 1;
   for (int i = n; i > 1; i--)
      result *= i;
   return result;
}
```

- No variable argument functions
  - Cannot be implemented safely, which is unacceptable for Java

Function overloading

- Can re-use names for functions of different type

```
void sort(int[] array);
void sort(double[] array);
```

- Can have different numbers of arguments, as well as different types of arguments

```
int indexOf(char ch);
int indexOf(String s);
int indexOf(char ch, int startPosition);
int indexOf(String s, int startPosition);
```

- No default arguments (unlike C++)

Resolution of an overloaded call

- Compiler finds best match to usage
  - Prefers exact type match over all others
  - Finds "closest" approximation
  - Only considers widening conversions, not narrowing
- Process is called "resolution"

```
void binky(int i, int j);
void binky(double d, double e);
```

`binky(10, 8)` will use (int, int)
`binky(3.5, 4)` will use (double, double)

- Can use cast to force choice or disambiguate

```
void winky(int i, double d);
void winky(double d, int i);
```

`winky(1, 10)` ambiguous so compiler error, requires cast
`winky(3.5, 4.2)` requires narrowing conversion, must cast

More info on overloading

- Cannot overload solely on return type

  Not allowed: **`boolean sort(int[] array)`**

  `void sort(int[] array)`

- Advantages of overloading
  - Opens up namespace
  - Aids readability, usually means shorter names
  - Nice for libraries, fewer distinct names to learn
- Disadvantages
  - Can accidentally invoke wrong version (leaving off parameter, forgetting a cast)
  - If poorly used, can lead to confusing interfaces (methods with same name, but not same functionality)

No preprocessor

- No #define, #include, #ifdef
- Good riddance, preprocessor is mostly a hack
- Source files are both implementation and interface
  - One less thing to get wrong
  - Encapsulation is tightly maintained, so no danger
- Constants supported with additional keyword
  - Final means "read-only", similar to C++ const
  - **`final int MaxStudents = 100;`**
  - Less error-prone than #define

No pointers!

- No explicit pointer types
  - (Later we will see that all objects are implemented as references, which are effectively pointers)
- No & or * operators
- No pointer arithmetic
- No function pointers
- Greatly improves robustness and security
  - No wild pointers
  - Can't accidentally return address of stack variable
  - Can't use cast to subvert type checking or object encapsulation

Memory management

- Managing storage is problematic in C
- Dynamic allocation exists in Java, but made easier
- malloc replaced by new
  - malloc error-prone: size calculation can be off, no type-checking
  - new is type-safe, raises exception on failure, compatible with Java's "constructors" for proper initialization
  - Example use: `int[] arr = new int[100];`
- free replaced by garbage collector
  - No leaks, double-frees, use of freed memory, etc.
  - Tracks references to each piece of dynamic storage
  - Might reclaim immediately when last reference goes out of scope or wait until more storage needed at later time

Input/Output

- System.out, System.in, System.err are standard
  - Stream objects
  - stdin, stdout, stderr equivalents
- Operations for output are print and println
  - Differ in whether newline is appended at end
  - Overloading used so they work for any type of variables

```
void print(int i);
void print(char ch);
void print(float f); //and so on...
```

- Typically build output using string concatenation

```
System.out.println("Your name is " + name +
        " and you are " + num + " years old.");
```

- Formatted output handled through string formatters
- InputStreams somewhat complex, will cover later

Array allocation and initialization

- Look like C arrays, but are quite different
- Can't specify size when declaring array

```
int arr[3];  //not legal in Java!
int arr[];   //okay
int[] arr;   //okay (means same as previous line)
```

- Arrays dynamically allocated using new

```
arr = new int[3];
```

- All elements are zeroed when array allocated
- Shorthand to declare, allocate, and initialize

```
int[] arr = {5, 10, 15, 20};
```

  - Values don't have to be constant, can be expressions, function calls, etc.

Accessing an array

- Java array always knows its length

```
int[] arr = {5, 10, 15, 20};
System.out.println("Len is " + arr.length);
```

- Elements indexed from 0 to length-1, like C
- All array accesses are bounds-checked

```
System.out.print("Val is " + arr[4]);
```

  - Raises exception for "ArrayIndexOutOfBounds"
- Length is set when allocated, cannot be changed
  - No realloc equivalent in Java
  - Have to create new array and copy over if that's what you wanted
- No pointer math for arrays

Array example

```
int sum(int[] arr)
{
   int i, sum = 0;

   for (i = 0; i < arr.length; i++)
      sum += arr[i];
   return sum;
}
```

Arrays are references

- Arrays actually implemented as references
  - Reference is a pointer in disguise
- Before allocation, array variable is null
  - null is the literal in Java for "zero reference"
- Returning array as function result is safe
  - It's a reference to dynamically allocated storage
- Assigning from one array to another makes alias

```
int[] a = {5, 10, 15, 20};
int[] b = a;
b[0] = 3;
System.out.println(a[0]);  //prints 3!
```

Multi-dimensional arrays

- Create array of arrays

```
int[][] matrix = new int[5][10];
int[] firstRow = matrix[0];
int firstElem = matrix[0][0];
firstElem = firstRow[0];
```

- Doesn't have to be rectangular

```
int[][] matrix = new int[5][];
for (int row=0; row < matrix.length; row++)
   matrix[row] = new int[row + 1];
```

  - Creates outer 5 rows (each is null array to start)
  - Each row array is one larger than previous

String type

- Not null-terminated arrays of char or char *
- String constants enclosed in double-quotes

```
String s = "a string literal";
```

- +, += overloaded to do string concatenation

```
s = ”The year is " + 1999 + "!";
```

- Everything can convert to a string representation
  - All primitive types and objects, too
- Functionality similar to 106 strlib:
  - Get length, retrieve characters
  - Search strings for chars and substrings
  - Concatenation, substring, find/replace
  - String comparison/equality

String example

```
String h = "hello";
String w = "world";

System.out.println(h + ' ' + w);
for (int i = 0; i < h.length(); i++)
   System.out.println(h.charAt(i));
w = h.substring(1, 3);
w += "binky";
int pos = w.indexOf("in");
```

Java program structure

- Simplest case: entire program in one file
- File defines a public ***class*** (more on classes later)
- Execution starts at main in that class
  - Function must have exactly the prototype shown below

```
public class MyProgram {
    public static void main(String args[])
    {
        System.out.println("Hello world!");
    }
}
```

- File must be named after class with .java extension
  - Above would be named MyProgram.java

Object-oriented programming

- Many small improvements C to Java, but also a big paradigm shift
- Programs operate on data
- In C, we constructed ADTs to manage data
- OOP ties behavior with data into ultimate ADT
  - Object = data + behavior
- OO program is a set of cooperating objects, messaging each other
- Composition and inheritance allow reuse of objects

Advantages of OOP

- Objects tend to map to real world entities
- Improved reliability
  - Fewer project interdependencies
- Supports teamwork
  - Clear interface/implementation boundaries
- More opportunity for code re-use
- Easier code maintenance
- Enables object-based testing

Basic OO idea

- Look at problem, decompose into objects
- Try to use off-the-shelf components where possible
- Arrange classes in tree for maximum code reuse
- Wire up objects to communicate with one another

OO vocabulary

- Class
  - A data type which describes the pattern of data and behavior
- Object or instance
  - A variable of a particular class
- Messaging
  - We say that you "send a message" to ask object to perform an action
- Method
  - "Function" for a class, it is the code body that implements a message

Tasks for you

- Figure out what it takes to compile, execute, & debug in your Java environment of choice
  - "Assignment #0" (not to hand in, just for you)
  - May want to review last Friday's section if you missed it
- Register with Webclass by Wednesday
  - See link from http://cse.stanford.edu/class/cs193j
- Reading
  - Ch 6 and 7 on classes and objects