Fundamental Java Programming Structures

CS 278
Sam Houston State University
Dr. Tim McGuire

What You'll Learn

- A Simple Java Program
- Comments
- Data Types
- Variables
- Assignments and Initializations
- Operators
- Strings
- Control Flow
- Class Methods (User-defined Functions)
- Arrays

Note:

- You are presumed to already be familiar with programming, so we will go rapidly
- We'll make enough "reassuring noises" along the way so you will realize what you already know
- We'll point out the differences between Java and other languages, notably C and C++

A Simple Java Program

- We'll start with a simple application -- applets come later
- It simply prints a message to the console

```
// A first program in Java
public class Welcome1
{
    public static void main( String args[])
    {
        System.out.println( "Hello, world!" );
        /* yes, "Hello, world!" yet again */
    }
}
```

What to note about our example

- Java is *case sensitive,* like C, so watch your spelling and capitalization
- The keyword **public** is called an *access modifier*
    - These modifiers control what other parts of a program can use this code
- The keyword **class** is there because, unlike C++, *everything* in a Java program is enclosed in a class

Names in Java

- Following the keyword class is the name of the class
    - The rules for names are similar to that of C++
        - Names must begin with a letter and after that they can have any combination of letters and digits
        - The length is essentially unlimited
        - You cannot use a Java reserved word (e.g. if, public, � like C++'s reserved words)
    - The convention is that class names begin with initial caps

- The file name for the source code must be exactly the same as the name of the public class, with the extension .java appended (i.e., **Welcome1.java**, not **welcome1.java**)

The main() method

- Every Java application *must* have a **main()** method
- Curly braces ({}) mark the beginning and end of a block
- All Java main methods are **public static void**

Java classes and methods

- Java classes are similar to C++ classes, but there are a few differences
- In Java, ***all*** functions are member functions of some class (and they are called *methods*)
- In Java, you have a shell class for the **main** method
- In Java, as in C++, *static member functions* are those defined inside a class, but do not operate on objects
- The **main** method in Java is always **static**

The Body of main()

- Contains a statement that outputs a single line of text to the console
- We use the **System**.**out** object and ask it to use its **println** method
- Java always uses the general syntax
  - **object**.**method**(***argument*_list**) for its function calls
- The **println** method works with a string and displays it on the console
- There is also a **print** method in **System**.**out** which doesn't add a newline at the end

Note about command-line arguments:

- Although in Java, as in C++, the main method receives an array of command-line arguments, the array syntax is different
- A **String[]** is an array of strings, and so **args** denotes an array of strings
- Unlike C++, the name of the program is not stored in the args array
- If we ran a Java application in **Doctor**.**class** with

```
java Doctor McGuire
```

  - **args[0]** will be **McGuire**, not **Doctor** or **java**

Comments

- Java has three styles of comments
- // begins a comment that runs from the // to the end of the line
- /* begins a comment that terminates with the character pair */
- /** and **/ delimit a special type of comment which is processed by the **javadoc** automatic documention tool
- /* */ comments do not nest in Java

Data Types

- Java is a *strongly typed* language
- Each variable must have a declared type
- There are eight *primitive types*
  - six are numeric types
    - 4 integer and 2 floating-point types
  - one is the character type **char**
  - one is a **boolean** type

Integers

Type	Size	Range
int	4 bytes	-2^{31} to $+2^{31}-1$ (just over 2 billion)
short	2 bytes	-32,768 to +32,767
long	8 bytes	-2^{63} to $+2^{63}-1$
byte	1 byte	-128 to +127

- These ranges are the same no matter what machine you are running on (unlike C++)
- Java does not have any **unsigned** types
- Long integer numbers have the suffix L (**40000000L**)
- Hexadecimal numbers have a prefix of 0x (**0xCAFE278**)

Floating-Point Types

Type	Size	Range
float	4 bytes	+/-(10^{-38} to 10^{+38}) (6 to 7 significant digits)
double	8 bytes	+/-(10^{-308} to 10^{+308}) (15 significant digits)

- You will usually want to use double because of its extended range and greater precision
- Numbers of type float have a suffix F
- Numbers without a suffix are always considered double, but you may use a D suffix
- All floating-point types follow the IEEE 754 specification

The Character Type (char)

- Apostrophes are used to denote char constants
- The char type denotes characters in the 2-byte Unicode encoding scheme
- ASCII code is the first 255 characters of Unicode
- Unicode characters are most often expressed in terms of a hexadecimal encoding scheme that runs form '\u0000' to '\uFFFF', with ASCII codes being '\u0000' to '\u00FF' -- e.g. '\u2122' is ^{TM}
- Check out http://www.unicode.org/

Special Characters

Escape	Name	Unicode value
Sequence		
\b	backspace	\u0008
\t	tab	\u0009
\n	linefeed	\u000A
\r	carriage return	\u000D
\"	quote mark	\u0022
\'	apostrophe	\u0027
\\	backslash	\u005C

- In C++, char denotes an integral type (1 byte integer) in the range 0..255 or -128..127
- In Java, char is not a number -- converting numbers to characters requires an explicit cast, however, characters are automatically promoted to integers without a cast

The Boolean Type

- The **boolean** type has two values, **false** and **true**
- It is similar the C++ type **bool**, except that in Java, you cannot convert between numbers and Boolean values, ***not even with a cast***

    - Why not? Have you ever been bitten by

      if (a = b) instead of **if (a == b)**?

    - So, in Java it is *not* true that 0 is **false** and non-zero is **true**

Variables

- Java uses variables in a way similar to that of C++
- Naming rules:
    - Variable name must begin with a letter and be a sequence of letters or digits
    - A "letter" is A-Z, a-z, _ (underscore), or *any* Unicode character that denotes a letter in a language -- Germans could use 'ä', Greeks could use 'µ', etc.
    - A "digit" is 0-9 or any Unicode character that denotes a digit in a language
- All characters in the name of a variable are significant and case is significant
- There is no upper limit to the length of a variable name

Conversions between Numeric Types

- Automatic promotions of numeric types work as they do in C++
- Demotion is done through explicit casts in the same way that C does it (the C++ form is not available)

```
double x = 9.9997;
int nx = (int) x; // notint(x) ala C++
```

    - this stores the value 9 in **nx**
- You cannot cast between **boolean** values and any numeric types

Constants

- In Java, you use the keyword **final** to denote a constant.

```
public class UsesConstants
{

     public static void main(String[] args)
     {

          final double CM_PER_INCH = 2.54;
          double paperWidth = 8.5;
          System.out.println("Paper width in cms: " +
          paperWidth*CM_PER_INCH);

     }

}
```

- The reserved word **final** indicates that you can assign to the variable once, then its value is set once and for all

Static Constants

- It is probably more common in Java to want a constant that is available to multiple methods inside a single class (a ***class constant***)
- Class constants are defined with the keywords **static final**

```
public class UsesConstants
{
```

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

               final double CM_PER_INCH = 2.54;
               double paperWidth = 8.5;
               System.out.println("Paper width in cms: " + paperWidth*CM_PER_INCH);

          }

     }
```

- Note that the definition of the class constant appears outside the main method
- Note: **const** is a reserved Java keyword, but it is not currently used for anything

Operators

- The usual operators `+ - * / %` are used in Java
- As in C, `+=, -=, *=, /=, %=` are also available
- The usual way of doing exponentiation is via the statement:
    - `double y = Math.pow(x,a);`
- Note: the **Math** class in Java has a large number of functions an engineer would need (constants for pi and *e*, *sqrt* , *ln*, *exp*, rounding, *abs*, *max*, *min*, etc.)

Other Operators

- Increment and Decrement Operators
    - The ++ and **--** operators are used as in C
- Relational and Boolean Operators
    - ==, !=, >, <, <=, >= are used in comparisons
    - **&&**, ||, and **!** are the Boolean operators
    - Boolean operators are evaluated in short circuit fashion
- Unlike C, Java does not have a comma operator
    - However, you can use a comma-separated list of expressions in the first and third slot of a **for** statement

Bitwise Operators

- **& ("and"),** | ("or"), ^ ("xor"), ~ ("not") are the bitwise operators
- e.g.,

  ```
      int fourthBitFromRight = (foo & 8) / 8;
  ```

  gives you a 1 if bit 3 of **foo** is 1, and 0 if it is 0
- >> and << are the right and left bit shift operators

  ```
      int fourthBitFromRight = (foo & (1 << 3)) >>3;
  ```
- >> does right shift with sign extension
- >>> does right shift with zero-fill
- (There is no >>> operator in C, and the action of >> is not well defined, so it only works correctly for positive integers)

Strings

- Java does not have a built-in string type
- The standard Java library contains a predefined class called **String**
- Each quoted string is an instance of the **String** class

```
    String e = "";     // an empty string
    String greeting = "Hello";
```

Concatenation

- Java allows you to use the + sign to concatenate two strings together

```
String part1 = "Hello";
String part2 = "world";
String message = part1 + part2;
```

- The above code makes the value of the string object **message** `"Helloworld"`
- When you concatenate a string with a value that is not a string, the latter is converted to a string

```
String rating = "PG" + 13;
```

  - sets **rating** to the string `"PG13"`

Substrings

- You can extract a substring from a larger string with the substring method of the String class

```
String machine = "computer";
String s = machine.substring(1,4);
```

  - creates a string consisting of the characters "**omp**"
- The first argument to **substring** is the starting position of the substring
- The second argument is the first position that you ***do not*** want to copy
- This seems peculiar, but it is easy to compute the length of the substring
- s.**substring(a,b)** always has length **b-a**

String Editing

- The length of a string is found by the **length** method

```
String machine = "computer";
int n = machine.length(); // is 7
```

- To edit a string, you need to use the **substring** method

```
String software = machine.substring(0,4)+"il" + machine.substring(6,8)
```

  - gives us "**compiler**" in software

```
machine = machine.substring(0,1) + machine.substring(4,8);
```

  - makes the "**computer**" "**cuter**"

Why can't you change characters directly?

- A Java string is not "just an array of chars"
- The Java documentation refers to string objects as being *immutable* -- that is, the string "**Hello**" always contains the sequence **'H','e','l','l','o'**
- You can, however, change the contents of the string variable message and make it refer to a different string
- Although immutable strings seem to be less efficient, they have one great advantage: they can be *shared*

How string sharing works

- Think of the various strings as sitting on the heap
- String variables then point to locations on the heap
- The substring **machine.substring(1,4)** is just a pointer to the existing **"computer"** string, together with the range of characters that are used in the substring

- Overall, the designers of Java decided that the efficiency of string-sharing outweighs the inefficiency of immutability

Further notes on Java strings

- C programmers sometimes have difficulty adjusting to Java strings, because they think of strings as arrays of characters:

```
char greeting[] = "Hello";
```

- That is the wrong analogy; a better one is

```
char *greeting = "Hello";
```

- But what happens when we make another assignment to greeting?

```
greeting = "Howdy";
```

- In C, we would have a memory leak
- In Java, automatic garbage collection is performed; the unreferenced memory will eventually be recycled

Testing Strings for Equality

- To test if two strings are equal, use the **equals** method

```
s.equals(t)
```

  returns **true** if the strings **s** and **t** are equal, **false** otherwise

- **s** and **t** can be string variables or string constants

```
"Hello".equals(greeting)
```

  is a perfectly legal boolean expression

- Do not use the `==` operator to test if two strings are equal
  - `==` only determines if the strings are stored in the same location

Comparing Strings

- The **compareTo** method is the exact analog of **strcmp** in C
- You can use

```
    if (greeting.compareTo("Hello") == 0)
```

  (although it seems clearer to use **equals** instead)

- compareTo has the heading

```
int compareTo(String other)
```

  - returns a negative value if the string comes before **other** in Unicode order, a positive value if the string comes after **other**, or 0 if the strings are equal

Useful methods from java.lang.String

char charAt(int index)

  - returns the character at the specified location

boolean endsWith(String suffix)

  - returns **true** if the string ends with **suffix**

boolean startsWith(String prefix)

  - returns **true** if the string ends with **prefix**

boolean equalsIgnoreCase(String other)

  - returns **true** if the string equals **other**, except for upper/lowercase distinction

int indexOf(String str)
int indexOf(String str, int fromIndex)

  - returns the start of the first substring equal to **str**, starting at index 0 or at **fromIndex**

int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)

  - returns the start of the last substring equal to **str**, starting at index 0 or at **fromIndex**

String replace(char oldChar, char newChar)

  - returns a new string that is obtained by replacing all characters **oldChar** in the string with **newChar**

String toLowerCase()

  - returns a new string containing all characters in the original string, with uppercase characters converted to lowercase

String toUpperCase()

  - returns a new string containing all characters in the original string, with lowercase characters converted to uppercase

String trim()

  - returns a new string by eliminating all leading and trailing spaces in the original string

Control Flow

- Java control flow constructs are very similar to those of C and C++, with a few minor exceptions
- We will emphasize what the exceptions are
- Java has no **goto**, but it does have a "labeled" version of **break** that you can use to break out of a nested loop (about the only place where a **goto** should be used anyway)

Block Scope

- A block (or compound statement) is a sequence of statements enclosed in curly braces
- A block allows you to use multiple statements as a unit
- In Java, unlike C, it not possible to declare identically named variables in two nested blocks

```
public static void main(String[] args)
{
    int n;
    �
    {
       int k;
       int n; // error -- can't redefine n in inner block

         �
    }

}
```

Conditional Statements

- **if** and **if-else** statements are identical to those of C/C++
- Because of the short-circuit boolean expression evaluation built-in to Java,

  ```
  if (x !=0 && 1/x > 0) // no division by zero
  ```

  does not evaluate **1/x** if **x** is zero, and so cannot lead to a divide-by-zero error

- Java also supports the ternary ? operator

  ```
  condition ? e1 : e2
  ```

Looping Structures

- The **while** loop does the test at the top, just as it does in C
- The **do-while** does the test at the bottom, also as in C
- The **for** loop is used for counter-controlled loops, as in C
- Be careful about testing for equality of floating-point numbers
  - A **for** loop that looks like this:

    ```
    for(x = 0; x! = 10; x += 0.01)
    ```

    may never end (due to roundoff errors, the value 10.0 is not reached exactly)

Multi-way selection

- **if**s may be nested, or you may use the **switch** statement in Java
- Unfortunately, the **switch** is as limited in Java as it is in C
- You may only select on a **char** or any integer type (except **long**) and you cannot use a range of values
- You still must use the **break** at the end of each **case** (unless you *want* to fall through into the next case)

Labeled Breaks

- goto is a reserved word in Java, but it is not used in any legal statement
- The only real reason for goto is to jump out of a nested loop upon an error condition
- The labeled break adds this capability without opening up the possibility of "spaghetti code"

Labeled break example

```
stop: while(...)
{

    for (...)
    {    . . .

        if (error_condition)

            break stop; // jump out of the loop labeled stop

        . . .
```

```
        }

    }
```

- Note that the label must precede the outermost loop out of which you want to break
- There is also a labeled **continue**, but it is of dubious utility

Class Methods
(User-Defined Functions)

- *Method is the "new and improved" terminology for function*
- *A method definition must occur inside a class*
- *It can occur anywhere inside the class, although (unlike C) Java custom places all the other methods of a class before the* **main** method
- Also unlike C, Java does not have "global" functions
- **Static** methods do not operate on objects and so are the same as normal C functions

An Example: Lottery Odds

- If you must match six numbers from the numbers 1 to 50 then there are (50· 49· 48· 46· 45)/(1· 2· 3· 4· 5· 6) possible outcomes, so your chance is 1 in 15,890,700 (good luck! or rather *fat chance*!)
- Just in case someone wants you to participate in a pick *k* out of *n* lottery, here is a method to compute the odds:

```
public static long lotteryOdds(int n, int k)
{

    long r = 1;
    int i;
    for (i = 1; i <= k; i++)
    {

        r = r * n / i;
        n--;

    }
    return r;

}
```

Calling the Method

- See Lottery.java
- This application uses the Java Swing API which we will cover later (because doing raw input in Java is a bearcat -- not to be confused with a BearKat, of course �)

Comments on Methods

- Methods in Java are similar but not identical to functions in C++
- There is no analog to C++ function prototypes in Java
- They are not required because methods can be defined after they are used -- the compiler makes multiple passes through the code
- Pointer and reference arguments do not exist in Java -- you cannot pass the location of a variable

Class Variables

- Occasionally, it is useful to declare a variable that will be accessible by all the methods in a class
- This is not quite the same as a global variable because it has class scope rather than global scope
- Class variables are declared like class constants, outside any method:

```
public class Employee
{

     private static double socialSecurityRate = 0.0775;
     public static void main(String[] args)
     {
          . . .
     }

}
```

Recursion

- Recursion is a general method of solving problems by reducing them to simpler problems of a similar type
- The general framework for a recursive solution to a problem looks like this:

```
solve_recursively(Problem P)
{ if (P is trivial) return the obvious answer;

     P1 = simpler problem;
     S1 = solve_recursively(P1);
     S = solution of P using the solution S1
     return S;

}
```

Recursion Example

- Factorial function
  - n! = n * (n-1)! n > 0
  - 0! = 1

```
public static long factorial(int n)
{

     if ( n <= 1 ) // trivial case

          return 1;

     else

          return n * factorial(n-1);

}
```

Arrays

- In Java, unlike C, arrays are "first-class" objects
- You are better off not thinking about how arrays are implemented in Java -- just accept them as objects that exist in and of themselves
- You can assign one array to another (but then both array variables refer to the same array
- Once you create an array, you cannot change its size easily
- If you need to change the size dynamically, you can use a Java vector object (discussed later)

Creation of Arrays

- Arrays are the first example of objects whose creation must be explicitly handled by the programmer
- This is usually done through the **new** operator:

```
int [] arrayOfInt = new int[100];
```

  - sets up an array that can hold 100 integers
  - The array entries are numbered from 0 to 99

Range Checking

- If you try to access the 101st element of a 100 element array, your source code will compile without an error or warning
- Your program will stop when it attempts to access any array element that goes outside the declared bounds of the array