

CMPS 12A/L
Introduction to Programming
(Accelerated)

Winter 2009

Don Chamberlin

<http://www.soe.ucsc.edu/classes/cmcs012a/Winter09/>

Topic 1: Expressions

Reading: JBD Chapter 2

Java is:

- A programming language
- The Java Virtual Machine
- A large library of useful classes (the “Java Platform”)

The promise of Java:

- “Write once, Run anywhere”
- Windows, Mac, Unix, browsers, cell phones, . . .

Compared to other languages, Java is:

- Object-oriented
- Strongly typed
- Safe (no pointer arithmetic, automatic garbage collection)
- Network-oriented
- Simple, powerful, and elegant

History of Java

- Invented in 1995 at Sun Microsystems by James Gosling and others.
- Embedded in early browsers (Netscape, IE); became de facto standard for web programming
- Version 1.0, 1.1, 1.2, 1.3, 1.4, 5.0, 6



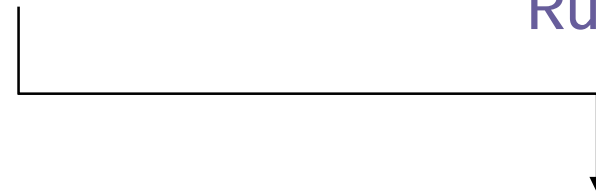
Compile time (uses JDK)

Your Program:
`Hello.java`



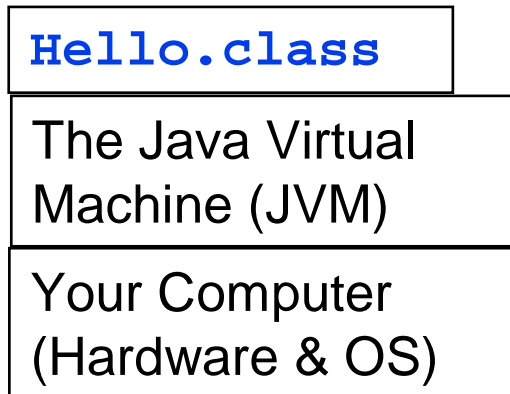
`javac Hello.java`

Java Compiler



Run time (uses JRE)

`java Hello`



- Java is case-sensitive:
 - `world`, `World`, `WORLD`, `world` are all different names
- There are conventions for using case:
 - Variables and functions have initial lower-case:
`jobCode`, `timeToMarket`
 - Class names have initial capitals
`Customer`, `Student`
 - Constants are all-caps:
`PI`, `MAX_SIZE`
- Follow the conventions in the textbook
 - Also see "Programming Style Guidelines" in Moodle

Skeleton of a simple Java program:

```
class FooBar {  
    public static void main(String[] args) {  
  
        your code goes here  
  
    }  
}
```

By convention, put your program in a file with the same name as the class that it implements:

FooBar.java

- Comments

```
x = x + 1;          // This is a comment
// x = x + 1;     // This line is "commented out"
/* This is a comment */
/**
 * This kind of "block comment" is often
 * used to document the purpose of a program
 * or function.
 */
```

- Whitespace

- Blanks, tabs, line breaks are (mostly) not significant
- Please follow conventions for indentation (see textbook and Programming Style Guidelines)

Names

- Various things in Java (variables, methods, classes, etc.) have names
- Names are made up of letters, digits and underscores
- Names must NOT:
 - begin with a digit
 - be the same as a Reserved Word:

`if`
`for`
`while`
`boolean`
`double`
etc.

Variables

- A variable is like a box that can hold a value
- Each variable must be declared before it is used

```
int age;  
double height;  
boolean married;
```

- Each variable has:

- A name (like **age**)
- A type (like **int**)
- A value (like **47**)
- A scope (the region of the program in which the variable is visible--typically from the declaration to the end of the current block or method)

Name	Value	Type
age	47	int
height	1.28E2	double
married	true	boolean

The assignment (=) operator

- Assigns a value to a variable
- The value must be compatible with the declared type of the variable
- A variable may be initialized with a value when it is declared

```
int x = 47, y = 81;
```

- A variable may be assigned another value later

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

- CAUTION: Do not confuse the assignment (=) operator with the equals (==) operator!

Strong typing

- Every variable has a type
- The type of a variable does not change
- The value of a variable must be an instance of its type
- Advantages of strong typing
 - Helps to catch errors at compile time
 - Helps compiler to generate efficient code
- Not all languages are strongly typed
 - Python is "weakly" or "dynamically" typed

Primitive Types

- Java has eight *primitive types*.
- Some of these types have *literals*.
- We will work mainly with the types shown in blue:

`boolean` `true, false`

`char` `'a', '\n'`

`byte`

`short`

`int` `47, -9`

`long`

`float`

`double` `26.5, -7.49E-2`

Conversions

- Java doesn't mind making *widening conversions*:

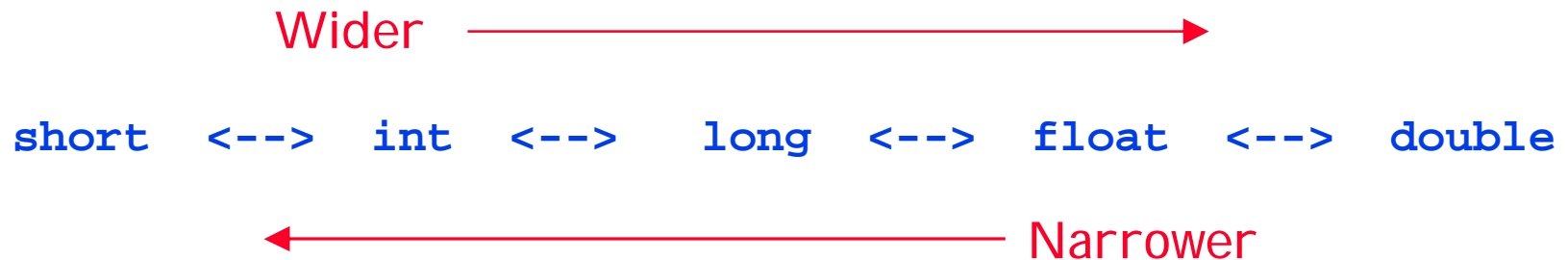
```
long x = 42;    // OK
```

- Java doesn't like *narrowing conversions*:

```
float x = 7.5; // error!
```

- But you can force a narrowing conversion by a *cast*:

```
float x = (float)7.5 // OK
```



Reference Types

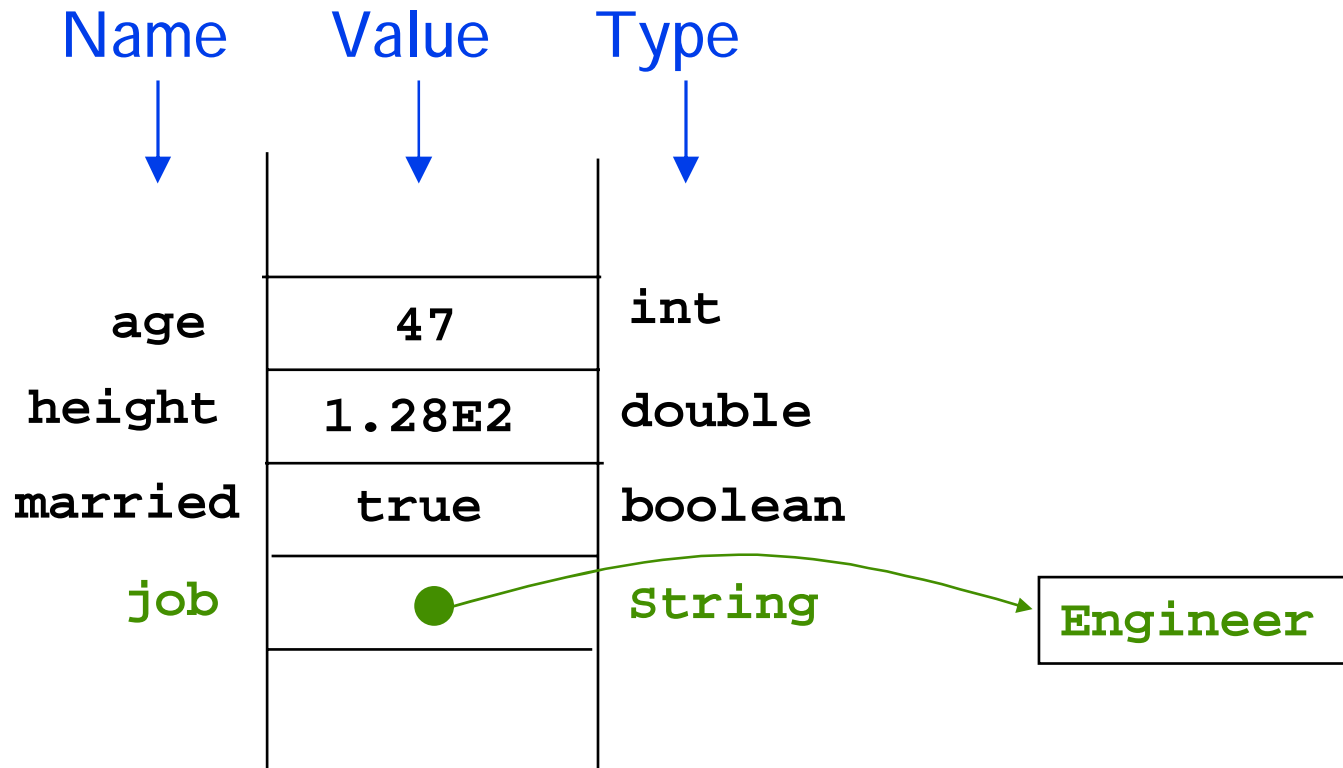
- In addition to the eight *primitive types*, Java has *reference types*
 - Some are built in.
 - The built-in `String` type has its own literal:

```
String job = "Engineer"
```
 - Later we will learn how to define our own reference types.
- Reference types have some important differences from primitive types:
 - Instances of reference types are called *objects*
 - They are handled differently in memory
 - They have *methods* (invoked by a dot-notation)
 - Objects have *identity* but primitive values do not

```
int age = 47;
double height = 1.28E2;
boolean married = true;
String job = "Engineer"
```

Stack

Heap



The Plus (+) operator on Strings

- Concatenates Strings
- Returns a NEW String
- Leaves the original Strings unchanged
- If one operator is a String and the other is a primitive, converts the primitive to a String, then concatenates
- Example:

```
String s1 = "Hello";  
String s2 = s1;  
String s3 = "Goodbye";  
s1 = s1 + s3 + 47;
```

The Equals (==) operator on Strings

- Returns true only if both operands have the same identity
- CAUTION!
 - The == operator does not test for equal content
 - Do not confuse the == operator with the = operator
- Examples:

```
String s1 = "Hello";  
String s2 = s1;  
String s3 = "Hello";  
s1 == s2           // true  
s1 == s3           // false  
"Hello" == "Hello" // false
```

Some methods of the String type:

```
String s1, s2;  
s1.length()      // returns length of s  
s1.equals(s2)    // true if contents are the same  
s1.charAt(n)     // returns nth char, starting at 0  
s1.substring(n1, n2) // returns substring  
                    // from position n (inclusive)  
                    // to position n2 (NOT inclusive)  
s1.trim()        // removes leading & trailing blanks
```

- Examples:

```
String s1 = "Hello", s2 = "Hello";  
s1 == s2      // false  
s1.equals(s2) // true
```

- For many more String methods, see:

java.sun.com/javase/6/docs/api/java/lang/String.html

More Operators

- We have already discussed some operators: + = ==
- Operators can be overloaded (the effect of the operator depends on the types of its operands)
 - Example: + adds numbers, but concatenates Strings
- In general, an operator can:
 - Have a value (it returns something)
 - Have a side-effect (it does something)
- We will discuss some Java operators
 - See a Java reference for a more complete list (including shifting and bitwise operators)

Arithmetic operators: + - * / %

- Value: Computed by rules of arithmetic
- Side effects: None
- Example: `5.0 % 1.5` (value is `0.5`)
- Special cases:
 - When at least one operand is a String, the `+` operator converts all operands to String and concatenates
 - If all operands are `int`, arithmetic operators always return an `int`, truncating if necessary:
`5 / 3` returns `1`
 - Arithmetic on `float` and `double` can return `Infinity` or `NaN`:
`5.0 / 0.0` returns `Infinity`
`0.0 / 0.0` returns `NaN`

Assignment operator: =

- Side effect: assigns value of RHS to variable on LHS
- Value: returns value of LHS, after assignment
- Value must conform to declared type of variable (possibly after a widening conversion)
- Examples:

```
x = 47
```

```
x = y = 47
```

```
int n = 4;
```

```
double d = 7.0;
```

```
d = n;    // OK
```

```
n = d;    // not OK
```

Modify operators: `+=` `-=` `*=` `/=` `%=`

- Side effect: modifies variable on LHS by value on RHS
- Value: returns value of LHS, after modification
- Remember: `x += y` is equivalent to `x = x + y`
- Examples:

```
int x = 4;
```

```
x += 1 // sets x to 5, returns 5
```

```
int y = 7;
```

```
y *= 2 // sets y to 14, returns 14
```

Increment/decrement operators: **++** **--**

- Side effect: adds or subtracts 1 to a variable
- Value: returns value of variable
 - If used as prefix, returns value of variable after change
 - If used as suffix, returns value of variable before change
- Examples:

```
int x = 47;
x++      // returns 47, then sets x to 48
++x      // sets x to 48, then returns 48
x--      // returns 47, then sets x to 46
double y = 1.5;
++y      // sets y to 2.5, then returns 2.5
char z = 'a';
++z      // sets z to 'b', then returns 'b'
```

Comparison operators: == != < <= > >=

- Value: boolean (true or false)
- Side-effect: None
- On numeric types: numeric comparison
- On Strings: == tests for identity, < and > are not supported
 - To compare unequal Strings `s1` and `s2`, use:
`s1.compareTo(s2)` or `s1.compareToIgnoreCase(s2)`
(comparison based on Unicode code sequences)
- On float and double: == is dangerous due to roundoff
 - Example: `4.7 % 2.3` returns `0.1000000000000000053`
- Examples:
`x == y` `x > 47` `myName.compareToIgnoreCase("J")`

Logical operators: `&&` `||` `!` (and, or, not)

- Value: boolean (true or false)
- Side-effect: None
- Operands must be booleans
- If LHS is false, `&&` does not evaluate RHS
- If LHS is true, `||` does not evaluate RHS
- CAUTION: Do not confuse `&&` and `||` with `&` and `|` (bitwise and, or) -- see text for details
- Examples:

```
y != 0 && x / y > 5
```

```
color.equals("Red") || color.equals("Blue")
```

```
gender == 'F' && !married
```

Expressions

- Expressions are built up from operators and operands
- The operands of an expression can be other expressions
- *Orthogonality principle:*
Wherever a value is expected, an expression can be used that returns a value of an appropriate type

Operator Precedence

- The order of execution of operators in an expression is determined by their precedence
- See JBD Appendix B for a complete precedence table
- Some examples you should remember:
 - * and / come before + and -
 $a * b + c * d$
 - arithmetic comes before comparison
 $a + b > c + d$
 - comparisons come before logical operators
 $a == b \ \&\& \ c < d$
 - ! comes before &&, which comes before ||
 $a \ \&\& \ !b \ || \ c \ \&\& \ !d$

Operator Precedence

- Use parentheses when you need to override normal precedence (or to be safe, if you can't remember it)

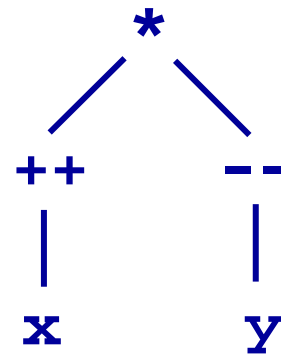
`a * (b + c) * d`

`a == (b && c < d)`

`a && !(b || c) && !d`

- Think of an expression as an "operator tree," possibly with multiple side effects

`x++ * y--`



- Using operator precedence and the orthogonality principle, you should be able to predict the value and side effects of any expression.

```
int x = 1, y = 2;
```

```
x+=y++
```

y is set to 3 but evaluated as 2
x is increased by 2 (set to 3)
value of the expression is 3

```
x++ + ++y
```

x is set to 2 but evaluated as 1
y is set to 3 and evaluated as 3
value of the expression is 1+3 = 4

```
z = x+=--y
```

y is set to 1 and evaluated as 1
x is increased by 1 (set to 2)
z is set to 2
value of the expression is 2

Getting Stuff In

- The name of the keyboard is `System.in`
- A Scanner is an object that can read input from `System.in` or from a file
- Scanner is defined in the `java.util` package. Import all the classes defined in this package so you can use them:

```
import java.util.*
```

- First, create a Scanner with a specific source:

```
Scanner scan = new Scanner(System.in);  
Scanner scan =  
    new Scanner(new File("myFile.txt"));
```

- A Scanner can "tokenize" your input. Tokens are separated by whitespace:

`I will be 29.5 years old
next month.`

- Use the "methods" of your Scanner to read and interpret input:
 - `scan.next()` returns the next token as a String
 - For each primitive type, there is a method to get the next token as an instance of that type:

`scan.nextInt()
scan.nextDouble()
scan.nextBoolean()`

If the next token does not conform to the expected type, you get an error.

- What if you don't want the Scanner to tokenize?
 - `scan.nextLine()` scans from the current position to the end of the current line and returns it all as a String without tokenizing
- Suppose a line of input looks like this: `5 7`
 - `scan.next()` returns "5" as a String
 - `scan.nextInt()` returns 5 as an int
 - `scan.nextDouble()` returns 5.0 as a double
 - `scan.nextBoolean()` raises an error
 - `scan.nextLine()` returns "5 7" as a String

- For each primitive type, Scanner has a method to test whether the next input token conforms to that type.
 - `scan.hasNext()` returns true if there is at least one more token in the input stream.
 - `scan.hasNextInt()` returns true if there is at least one more token in the input stream and the next token conforms to the primitive type `int`.
 - `scan.hasNextDouble()` returns true if there is at least one more token in the input stream and the next token conforms to the primitive type `double`.
- These methods are useful for testing inputs for correctness and for detecting the end of the input stream.

Getting Stuff Out

- The name of the printer is `System.out`
- `System.out` has two important methods:
 - `System.out.print(something)`
prints something but does not start a new line
 - `System.out.println(something)`
prints something and then starts a new line
- Example:

```
System.out.print("a");  
System.out.println("b");  
System.out.println("c");
```

ab
c

- `print` and `println` can operate on any value
 - For primitive types, they convert to a String
 - For Strings, they print the content
 - For other reference types, they print *something*
- `print` and `println` are often used with +

```
System.out.println("x = " + x + ", y = " + y);  
x = 42, y = 57
```
- But be careful!

```
int x = 5, y = 7;  
System.out.println("x + y = " + x + y);  
x + y = 57
```

- The `printf` method provides finer control over print format:

```
double mySalary = 1000.00 / 3;
System.out.println("My salary is " + mySalary);
    My salary is 333.33333333333333
System.out.printf("My salary is %6.2f",
    mySalary);
    My salary is 333.33
```

- Similar to `printf` in C
- See text for details

Getting Stuff In and Out

```
import java.util.*;
/* This program prompts the user for a name
 * and then prints a greeting.
 */
class Greeting {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String person;
        System.out.println("Please enter your name");
        person = scan.next();
        System.out.println("Hello, " + person);
        System.out.println("Have a nice day.");
    }
}
```

Generating Random Numbers

- Computers can't really generate random numbers
- But they can generate "pseudo-random" numbers
- Method 1: If the number stream doesn't need to be reproducible and you want floating-point numbers
 - Just call `Math.random()`
 - You get a random `double` between 0.0 and 1.0
 - If you want random numbers in a different range, apply a scale and offset.
 - Example (generates random doubles between -1.0 and 1.0):
`2.0 * Math.random() - 1.0`

- Method 2:

When the number stream needs to be reproducible or you want random numbers of various types

- Create a Random object with a specific "seed"
- Call methods of the Random object to get random numbers

```
import java.util.*;  
Random rand = new Random(1234);  
double d = rand.nextDouble(); // 0.0 <= d < 1.0  
int n = rand.nextInt(10);      // 0 <= n < 10  
boolean b = rand.nextBoolean();
```

- Every time you execute your program, it will generate the same sequence of random numbers.
(Helpful for testing)