Core JAVA

- Fundamental Concepts
- Bootstrapping
- Basic Language Syntax
- Common Caveats
- Coding Conventions

Core JAVA

- Fundamental Concepts
- Bootstrapping
- Basic Language Syntax
- Common Caveats
- Coding Conventions

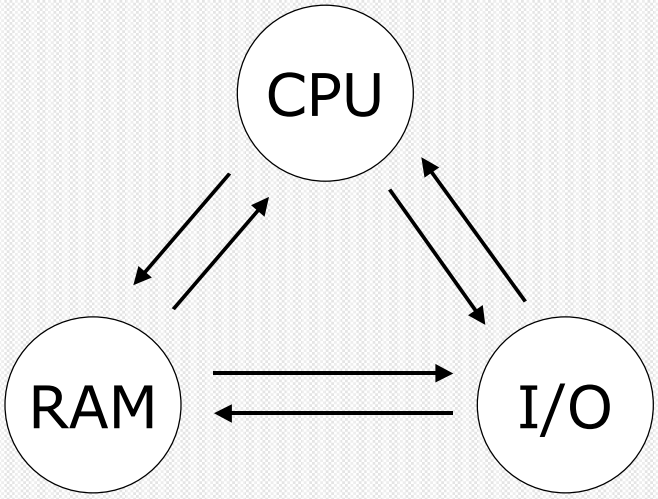
General Purpose Computers

- Most computers that we encounter are application specific…
  - Light switches, microwave oven controller, VCR timer, DirecTV receiver
- GPCs are different…
  - GPCs are built as generic problem solving machines
  - Programming is the bridge from the generic tool to a useful “machine”

GPC (Computer) Organization

- CPU – Central Processing Unit
  - Primary location for computations
- I/O – Input and Output Subsystem
  - Devices and communication bus for user interaction, import/export of data and permanent storage
- RAM – Random Access Memory
  - High speed, volatile, “scratchpad”

Classic Computer Organization



Programming a GPC

- The hardware can be controlled using "machine language"
  - 01001011001010010010010010101
- Assembly language is an attempt to make this more "friendly"
  - MOV AX, BX
  - ADD R3, #32, R9
  - PUSH EAX
  - JZ R25, [R12]

High Level Languages

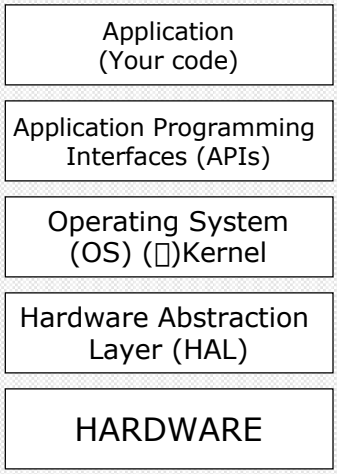
- Machine and Assembly Language are very hard to use…
  - Try computing a 3rd order integral in assembly…
  - How about writing a GUI?
- So we create high level languages and compilers for translating high level programs into assembly

Multiuser/Multitasking

- GPCs are shared…
  - … between multiple programs
  - … between multiple users
- The operating system (OS) governs the computer's hardware resources
  - It allocates time for each program to run
  - It provides a unified interface for all of the hardware devices
  - It might also provide session support for multiple users

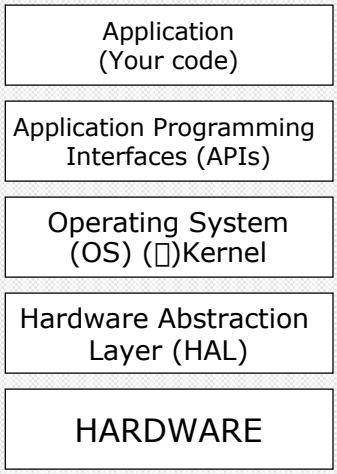
Typical Topology

- Most applications talk to APIs implemented by the OS kernel.
- Most reasonable OS kernels talk to hardware through an HAL



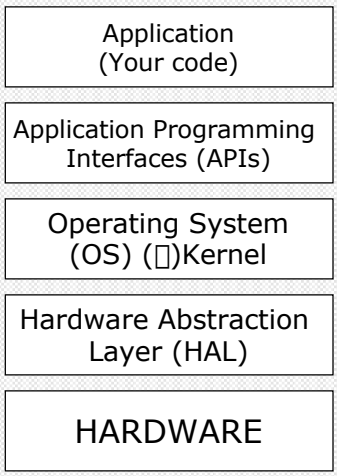
Why so many layers?

- HAL makes all hardware look the "same" to kernel
- The same kernel code that runs on an Intel x86 PC can run on a DEC 21x64 workstation



The same for your code!

- Assume that all OS's agree on a common API
- You can write a single piece of code that can be recompiled onto many platforms



"Recompiled?"

- Platforms will differ in many ways…
  - Static sizes for OS and device interfaces
  - Availability/coding of machine instructions
- Recompilation requires the source…
  - Your competitors will have access to code which took you a very long time to develop
  - Your users may not have a compiler… if they do, they may not know how to use it
  - Source code verification is critical!

The JAVA Way

- Run a JAVA Virtual Machine as a regular application on the OS
- The JVM *simulates* a standard platform (GPC) that all JAVA programs can execute on
- Write once, run anywhere!

Caveats of the JAVA Way

- Performance
  - Clearly, JAVA will always be slower than a natively coded application
  - JIT JVM technology brings most applications within 30% of native code
  - Latest HotSpot JVMs are within 5% of C++
- Touching the hardware
  - Not all local devices will have an interface through the JVM… your favorite USB scanner may simply not work (at least, for now…)

Core JAVA

- Fundamental Concepts
- Bootstrapping
- Basic Language Syntax
- Common Caveats
- Coding Conventions

You need to "install JAVA"

- JAVA environment is like any other program (you need to install it)
- At home, download and install the proper JDK (J2SE SDK) for your platform
  - http://java.sun.com/j2se
- Also get the J2SE documentation
  - http://java.sun.com/docs
- This will have already been done for you in the computer lab

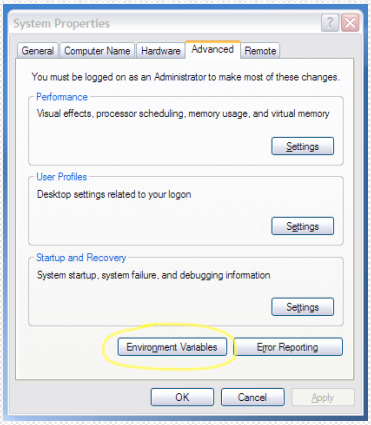
Add JAVA to your PATH

- Under both Windows and UNIX, the JAVA executables reside in the “bin” subdirectory of the installation site
- Add that directory to your PATH
  - Win95/98 – edit your AUTOEXEC.BAT
  - WinNT/2K/ME/XP – edit environment variables found under advanced system properties
  - Most UNIX – edit your .profile or .cshrc
  - MacOS 9 – upgrade to OS X
  - MacOS X – do nothing, it’s preinstalled!

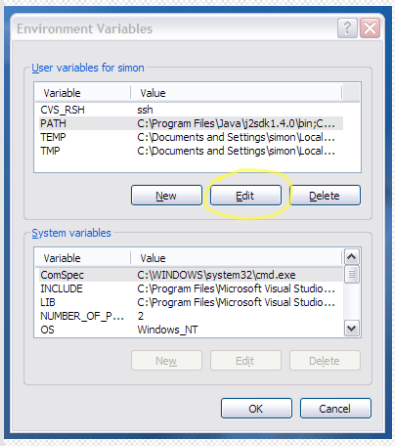
Add JAVA to your PATH

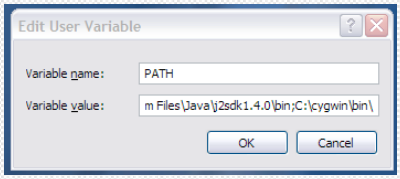
- For example, under Win95/98, add the following statement to the end of your AUTOEXEC.BAT file:
  - SET PATH=C:\JDK1.4.0_01\BIN;%PATH%
- Under UNIX, edit your .profile and add the following statement:
  - EXPORT PATH=$PATH:/opt/jdk1.3/bin
  - Substitute your install path for /opt

WinNT/2K/ME/XP Path Addition



WinNT/2K/ME/XP Path Addition





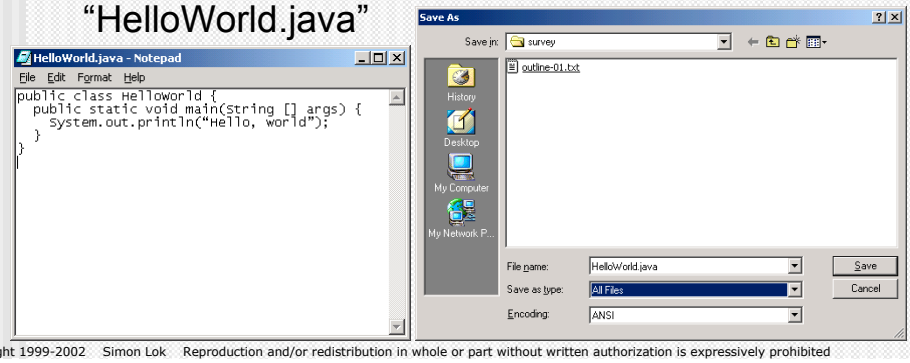
Hello World - Our First Program

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

- All JAVA modules begin with a class definition … classes are “objects”
- The POI (point-of-entry) of a class is the main method

HelloWorld under Windows

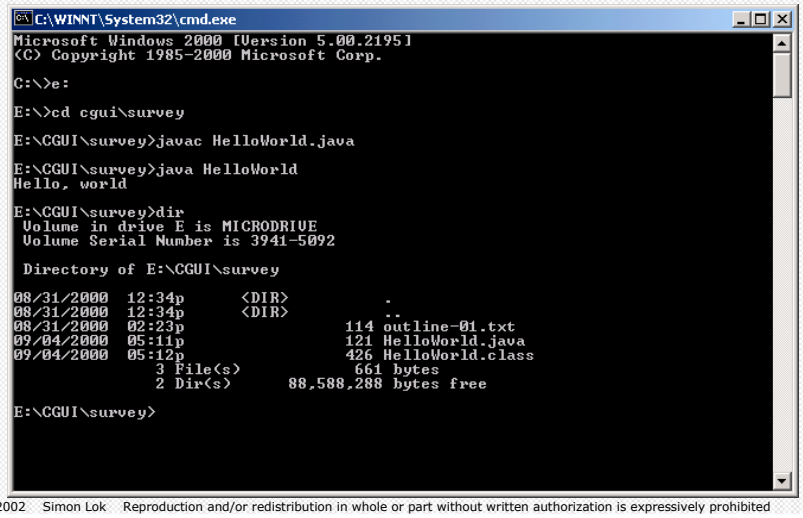
- Start :: Accessories :: Notepad
- Type in HelloWorld as given
- Save as type “All Files” with name “HelloWorld.java”



HelloWorld under Windows

- Start a command prompt
  - Win98: Start :: Run :: DOSPRMPT
  - WinNT/2K: Start :: Run :: CMD
- Change to the proper directory
- Compile and Execute
  - JAVAC HelloWorld.Java
  - JAVA HelloWorld
- Watch the case!

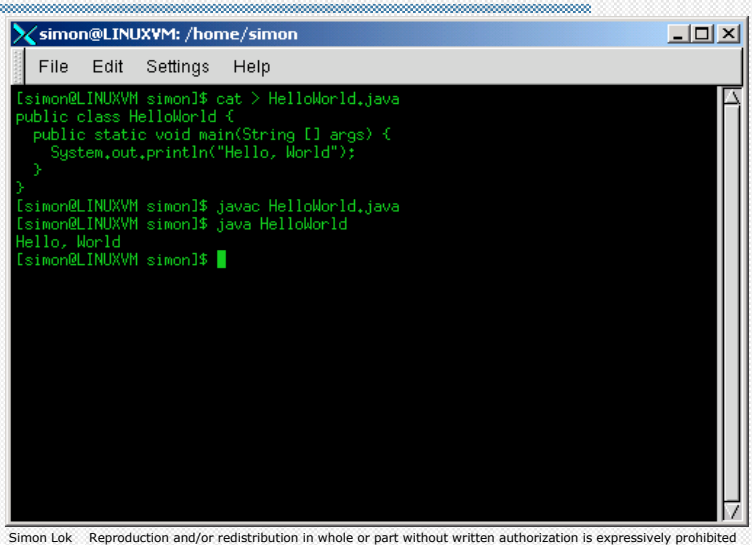
HelloWorld under Windows



HelloWorld under UNIX

- Start your favorite text editor
  - EMACS, PICO, VI or just use CAT
- Type in HelloWorld as given
- Save and exit the editor
  - Use filename “HelloWorld.java”
- Compile and Execute
  - JAVAC HelloWorld.Java
  - JAVA HelloWorld

HelloWorld under UNIX



The “Real World”

- Text editors with command line compilation are “stone age” tools for program development
- Contemporary software engineering is accomplished using RAD (rapid application development) tools and IDEs (integrated development environments) with inline debuggers

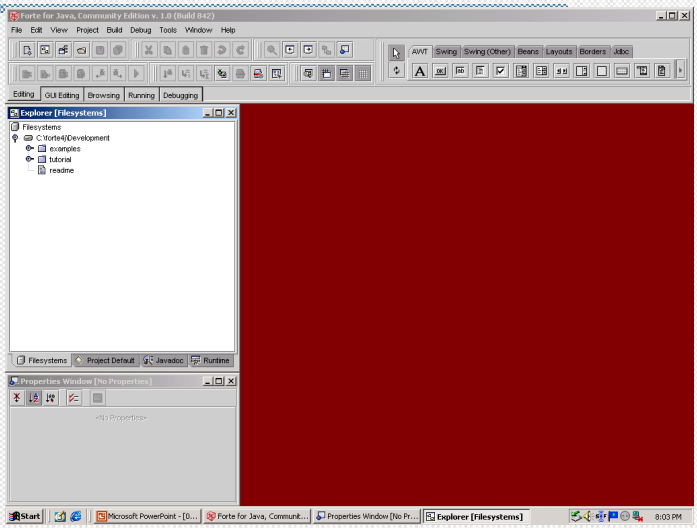
JAVA RAD Tools and IDEs

- Many are available…
  - Symantec Visual Cafe
  - Borland J-Builder
  - Microsoft Visual J++ (EOL), J#
  - Sun Forte / NetBeans
- Recommendation: Sun Forte / NetBeans
  - It’s free
  - It’s the official Sun IDE
  - It produces “clean code”
  - It’s got modules for RMI and other cool stuff

The NetBeans / Forte IDE

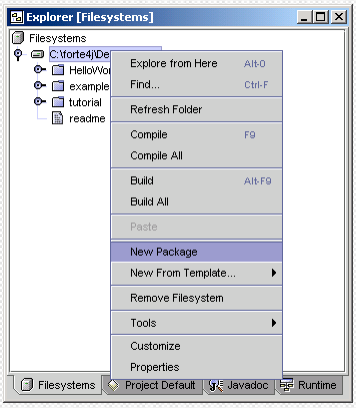
- You must download and install Netbeans / Forte as a separate package:
  - http://www.netbeans.org
  - http://www.sun.com/forte/ffj/ce/
- Prerequisites
  - J2SE SDK
  - J2SE Documentation (recommended)
  - Installer automatically detects the location of your JDK and documentation during the installation process

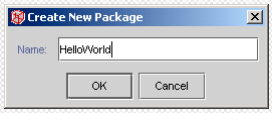
The Main IDE Screen



Hello World in Forte/NetBeans

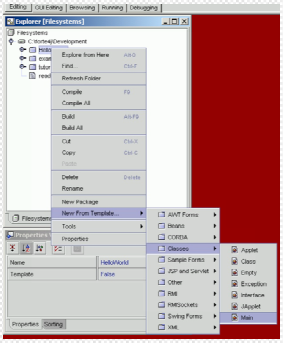
- Create a new package
  - Right click on the explorer window…

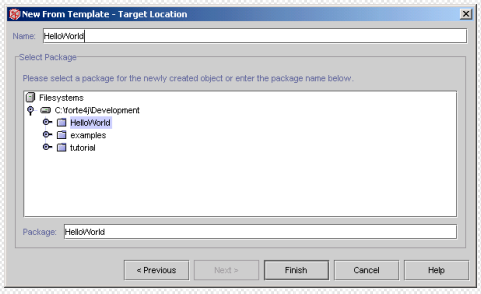




Hello World in Forte/NetBeans

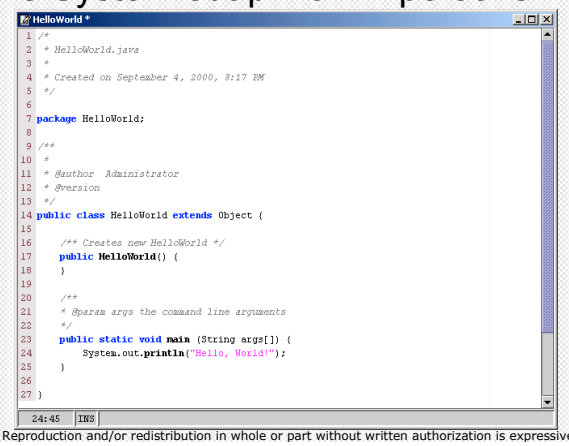
- Create a new class
  - Right click on the name of the new project that you just created…





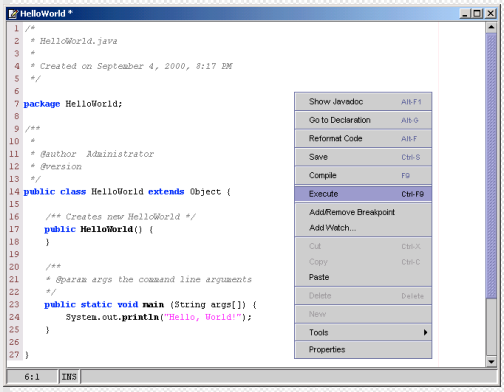
Hello World in Forte/NetBeans

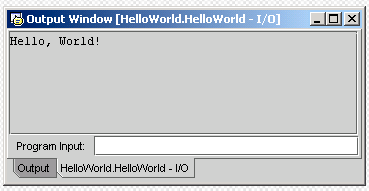
- The template does most of the work, just add the System.out.println imperative



Hello World in Forte/NetBeans

- Compile and run
  - Right click on the name of the class…





Features of Forte/NetBeans

- RAD (rapid application development)
  - "Drag-and-drop" programming of GUIs
  - Clean (pure JAVA) code generation
- Integrated debugger
  - Real time variable watches
  - Single click breakpoints
- Powerful templates
  - You only need to write the "core" code
- … and much much more.

Core JAVA

- Fundamental Concepts
- Bootstrapping
- Basic Language Syntax
- Common Caveats
- Coding Conventions

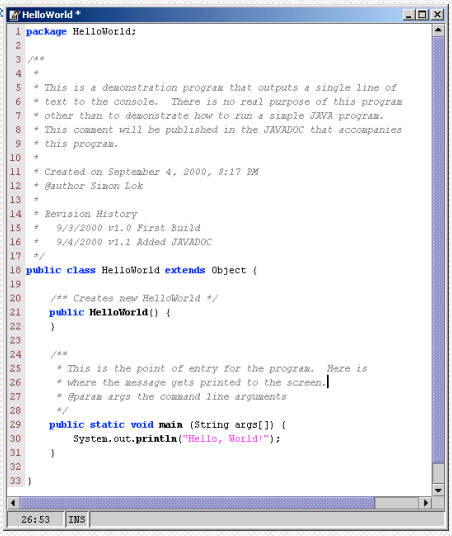
Inline Comments

```
public class HelloWorld {
  public static void main(String [] args) {
    // Next line prints out a message to the console
    System.out.println("Hello, world");
  }
}
```

- Denoted by // (same as C++)
- Everything between // and EOL is not compiled
- Write short notes about what this particular piece of code is doing

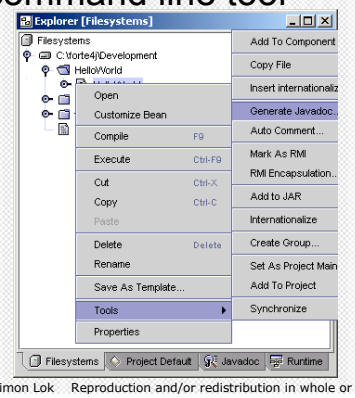
JAVADOC Comments

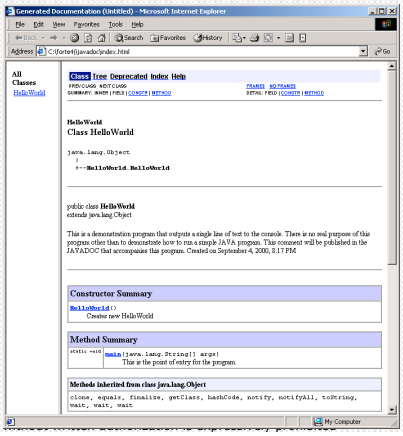
- JAVADOC comments are begun by the sequence /**, continued with a * at the beginning of each line and terminated by the */ sequence
- JAVADOC comments are “official” documentation of your code



JAVADOC Comments

- JAVADOC comments can be compiled into HTML files via Forte or via the JAVADOC command line tool





Primitive Variables

- A *variable* is an item of data named by an identifier
  - Variable declaration is manipulation of the computer's scratchpad (RAM)
  - We are reserving a space in the scratchpad and giving that space an easy-to-use name
- Examples:
  - int x = 0;
  - float f = 3.14159265;

Fixed Point Data Types

- Byte - byte b = 16;
  - 8-bits, -127 to 127
- Short - short s = -1543;
  - 16-bits, -32767 to 32767
- Int - int i = 100340;
  - 32-bits, -2 billion to 2 billion
- Long - long l = -123456789123;
  - 64-bits, absurdly large numbers

Fixed Point Data Types

- Used when representing integral numeric data (like 4 or 5)
- Common misconception:
  - Fixed point types can/is not used to represent fractional values
- Used to represent data where the decimal point position stays constant
  - Example: money … $18.45

Floating Point Data Types

- Used when data may take on wildly different values or when scientific precision must be preserved
- Float
  - float f = 3.14159265;
  - 32-bits (max value ~10^38)
- Double
  - double d = 5.6243*Math.pow(10,250);
  - 64-bits (max value ~10^308)

Why use fixed point?

- Why bother with implicit decimal points?
  - You might forget about the point…
  - Somebody else might modify your code…
- First guess: it's the size
  - 8 bits versus 32 or 64 bits…
  - No… because of alignment issues
- The real reason… SPEED!

Fixed vs. Floating Point

- On a MIPS R4000 class processor (found in 1990 SGI Indy's and Y2000 PDAs like the Casio Cassiopeia)…
  - Floating point division takes ~ 70 cycles
  - Fixed point division takes ~ 13 cycles
- This is even more apparent with SIMD instruction sets…
  - MMX/SSE/3DNow, etc. can improve fixed point performance by 4 to 16 times!

Other Data Types

- Boolean
  - 1-bit fixed point type
  - Use the words "true" and "false" to assign and compare values
- Char
  - Holds a single unicode character
  - 16-bits (unlike the "usual" 8-bit ASCII)
- Reference
  - Called pointer in C/C++… this holds an address in memory

Literal Data

- How can you tell if 12 is a byte, short, int or long?
- By default, literals w/o a decimal point are int and with a decimal point are double
  - You can use 12345L to make a long
  - 12.3456F can be used for float
  - Byte/Short don't have equivalents

Something to Try…

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

      float f = 3.14159265;  // this is okay.
      int x = 3.14159265;  // is this valid?

      byte b = 32; // this is also okay.
      byte b2 = 130; // … how about this?

   }
}
```

Another Thing to Try…

```
public class Test {
   public static void main(String args[]) {
      boolean firstGuy = true;  //  works.
      boolean secondGuy = 1;    //  this?
      boolean thirdGuy = -1;    //  this?
   }
}
```


Aggregate Types - Arrays

- Easily access groups of variables
  - All variables share the same prefix
  - Variables must be of the same type
- Syntax:

```
int[] myArray = new int[64];
myArray[15] = 9226;
System.out.println(myArray[15]);
```

- Arrays start counting from ZERO!

Something to Try:

```
public class ArrayTest {
   public static void main(String [] args) {
      int [] myArray = new int[5];
      for (int j = 0; j <= 5; j++) {  // ???
          myArray[j] = j*100;
          System.out.println(myArray[j]);
      }
   }
}
```


Type Casting

- If you want to "force" one type into another, you have to "cast" it
- This code will not compile:

```
int x = 123;
byte b = x;
```

- This is the correct code:

```
int x = 123;
byte b = (byte)x;
```

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      int x = 5000;
      byte smallFry = 64;
      long bigGuy = 1234567890;
      x = smallFry;      // will this work?
      x = bigGuy;        // how about this?
      x = (int)bigGuy; // or this?
   }
}
```


Scope

- Variables live within the nearest set of curly braces…

```
public class myStuff {

  int x = 327; // this is visible classwide

  public static void main(String[] args) {
    int y = -33; // visible inside main
  }

}
```

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      int x = 32;
      System.out.println(x);
      {
          int x = 64;  // this won't work
          int y = 74;
          System.out.println(x);
          System.out.println(y);
      }
      System.out.println(x);
      System.out.println(y); // won't work
   }
}
```


Constants

- If you want to reserve a space in memory as being "immutable", use the "final" keyword:

```
final int x = 327;
final double PI = 3.14159265;
```

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      final int x = 32;
      int y = 64;
      System.out.println(x);
      System.out.println(y);
      x = 24;  // this won't work
      y = 32;
   }
}
```


Infix Arithmetic

- The + - / * operators work as you think that they would:

```
int z = y + x;
double fz = fx * fy + fw;
```

- In addition there is the % operator which is called modulo, it divides and takes the remainder

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      int ix = 9;
      double fx = 9.0;
      int iy = 5;
      double fy = 5.0;
      System.out.println(ix/iy);
      System.out.println(fx/fy);
   }
}
```


Prefix/Postfix Arithmetic

- The – operator negates a value:

```
int y = -z;
```

- The + operator promotes:

```
byte x = 32;
int y = +x;
```

- The ++ and -- operators increment and decrement by 1

```
int z = x++;
int y = ++x;
```

++X versus X++?

- Consider the following piece of code:

```
int x = 1;
System.out.println(x);      1
System.out.println(x++);    1
System.out.println(x);      2
System.out.println(++x);    3
System.out.println(x);      3
```

- What's the output?

Relational Operators

- Unlike arithmetic, these process numeric data into a boolean result
- The common ones are:
  - >, >=, <, <=, == and !=
- They work as you would expect

```
int y = 8; int x = 3;
boolean myGuy = (y < x);
System.out.println(myGuy);
```

Combining Relational Ops

- Conditional Combinations
  - &&, ||, ^ - implement the logical AND, OR and XOR functions
  - boolean result = ((x > y) && (x < y));
- Negation
  - The ! operator can prefix any boolean variable or expression
  - It inverts the logical value of the variable or expression that it prefixes

Something To Try:

```
public class Test {
    int x = 32, y = 32, z = 64;
    boolean a = (x > y);
    System.out.println(a);      // output?
    boolean b = (x == y);
    System.out.println(b);      // output?
    boolean c = ((y == x) && (z > y));
    System.out.println(c);      // output?
}
```

Bitwise Operations

- Bitwise Conditional Operations
  - &,| and ^ perform bitwise AND/OR/XOR on numeric data…
  - ```
    int x = 6 & 3;
    int y = 6 | 3;
    System.out.println(x + ", " +y);
    ```
- Remember that 6 is 0110 and 3 is 0011 in binary…

```
  0110      0110      0110
& 0011    | 0011    ^ 0011
  0010      0111      0101
```


Bit Shifting

- The >>, >>> and << operators move the bits around…
  - ```
    int x = 16 >> 2;
    System.out.println(x);
    ```
- Shifting can be used for quickly multiplying and dividing by two
- >>> differs from >> in that >>> is unsigned… >> simply pads zero

Why bitwise ops?

- Hardware interaction…
  - Most hardware provides a stream of data in the form of bytes that need to be sliced, shifted and otherwise massaged into usable form
- Flags…
  - Rather than having many boolean variables, you can have a fixed point “flag” variable with up to 64 flags

Assignment Operations

- You can assign with the = operator, but you can also combine most other operations…
  - ```
    int x = 0;
    x += 5; // same as x = x + 5;
    ```
- +=, -=, *=, /=, &=, >>=, etc. are all valid assignment operations
- y += 6 bis faster than y = y + 6;
    ```

String Manipulation

- The + infix operator does something slightly different with Strings…

```
String firsGuy = "Hello";
String secGuy = "World";
String sum = firstGuy + " " + secGuy;
System.out.println(sum);
```


String Comparison

- You cannot use == to compare Strings directly!
- Call "compareTo"
  - Returns the lexographic difference
  - Zero means they're the same
- Syntax:

```
if (myString.compareTo("hello") == 0) {
      // executes if myString == "hello"
}
```

Conditional Execution

- Execute a statement block if a certain condition is met…

```
if (x > 0) {
    System.out.println("x is good!");
} else if (x < 0) {
    System.out.println("problem!");
} else {
    System.out.println("borderline!");
}
```


Something to Try:

```
public class Test {
   public static void main(String [] args) {
      double x = 32;
      if(x < 0) {
          System.out.println("x less than zero");
      } else if (x > 0) {
          System.out.println("x greater than zero");
          boolean positiveNumberFlag = true;
      } else {
          System.out.println("x is zero");
      }
      System.out.println(positiveNumberFlag); // ???
   }
}
```

Conditional Execution

- Another alternative:

```
switch(x) {
   case 0: System.out.println("border!");
           break;
   case 1: System.out.println("good");
           break;
   default: System.out.println("BAD!");
  }
```


Something to Try:

```
public class test {
   public static void main(String [] args) {
      int x = 2;
      switch(x) {
          case 1:  System.out.println("one");
                   break;
          case 2:  System.out.println("two");
                   // whoops, forgot the break!
          case 3:  System.out.println("three");
                   break;
          default: System.out.println("unknown");
      }
   }
}
```

Iteration

- To repeat a task a specified number of times, use the “for” construct:

```
for(int i = 0; i < 10; i++) {
   System.out.println(i);
}
```

- To repeat until a condition is met:

```
while(i < 10) {
   System.out.println(i);
   i++;
}
```


More Iteration

- Another variation on the while loop:

```
int i = 0;
do {
   System.out.println(i);
   i++;
} (while (i < 10));
```

- The do/while loop will always run the loop at least once
- This is often used for user input

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      int j = 0;
      // print out all even numbers up to 100
      while (j != 99) {
         System.out.println(j);
         j += 2;
      }
   }
}
```


Changing the flow

- Break and continue can be used to stop/jump iteration blocks
-
```
OUT: for (int j = 0; j < 100; j++)
{
   for (k = 0; k < 100; k++) {
      if ((j % k)==0) continue OUT;
      System.out.println(j);
   }
}
```

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      for (int w = 0; w < 4; w++) {
         MID: for (int y = 0; y < 5; y+= 2) {
            for (int k = 3; k > 0; k++) {
               if ((w + y + k) == 4) break;
               if ((w * y) > 6) continue MID;
            }
         }
      }
   }
}
```


Basic I/O using a CLI

- Soon, we will be building all of our applications with GUIs, but for now, we can take user input from the command line interface
- There are two basic ways to get user input from the CLI
  - The command line arguments
  - Reading from the console

Command Line Arguments

- When you run a program, you often supply it with arguments
  - `dir myfile* /a`
  - `ls –la myfile*`
- You can supply a JAVA program command line arguments as well
  - `java myProgram myFirstArg anotherArg`

Retrieving Arguments

- Recall the declaration of main:
  `public static void main(String [] args)`
- The array “args” can be used to access the parameters
- The scope of the “args” array is inside main
- args.length gives us how many parameters were passed

Something to Try:

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

        for (int j = 0; j < args.length; j++) {
            System.out.println(args[j]);
        }

    }
}
```


Arguments are Strings!

- Be careful… the command line arguments in the args array is of type String
- You must convert it to a numeric type if you plan on doing arithmetic
  - `int myArg = Integer.parseInt(args[2]);`
  - `float gimme = Float.parseFloat(args[1]);`

Something to Try:

```
public class Test {
   public static void main(String [] args) {
      if (args.length < 2) {
         System.out.println("Must have two args");
         System.exit(-1);
      }
      double a0 = Double.parseDouble(args[0]);
      double a1 = Double.parseDouble(args[1]);
      System.out.println(args[0] + args[1]);
      System.out.println(a0 + a1);
   }
}
```


Reading from the Console

- Unfortunately, this is pretty complicated… the reason is because Sun wants JAVA to be very "clean"
- Refer to the NumberInput.java sample program…
  - Basically you have to open System.in
  - Then you have to readLine and parse
  - You also have to make sure the user types in something that's valid using a loop

One more thing…

- You must import java.io.*;
  - This loads a package, we'll revisit this later
- The try-catch construct is required when doing any kind of I/O
  - ```
    try {
        String input = console.readLine();
    } catch(Exception e) {
        System.out.println("An error occurred.");
    }
    ```
  - This is called an exception handler, we will revisit this later

Something to Try:

```
import java.io.*;
public class Test {
  public static void main(String [] args) {
    InputStreamReader in = new InputStreamReader(System.in);
    BufferedReader con = new BufferedReader(in);
    boolean isGood = false;
    while(isGood != true) {
       try {
          System.out.print("Enter a number:  ");
          double input = Double.parseDouble(con.readLine());
          isGood = true;
       } catch (Exception e) {
          System.out.println("That was not a number!");
       }
    }
    if (input > 0) { System.out.println("positive"); }
    else { System.out.println("not positive");
  }
}
```
```

Core JAVA

- Fundamental Concepts
- Bootstrapping
- Basic Language Syntax
- Common Caveats
- Coding Conventions

`javac: Command not found`

- You have not put the jdk/bin directory into your executable path…
  - Under Win9X/ME, edit autoexec.bat
  - Under WinNT/2K, modify system properties
  - Under UNIX variants, edit .profile/.cshrc
- Better yet, use Forte (or some other IDE) that has a built in compiler

Blah.java:14: ';' expected

- You forgot to end the line with the semicolon character
- You forgot to match your curly braces… for every { you need a }

Can't find class MyStuff.class

- You attempted to run a JAVA program incorrectly:
  - java MyStuff.class
- You should run JAVA programs w/o the trailing .class:
  - java MyStuff

It's a Jungle Out There

- Keep your variables to the absolute minimum scope that they need
- This helps prevent namespace collisions…
  - Namespace collisions are usually quite painful to debug, especially if it's some obscure control flow variable

Infinite Loops

- Loops terminate upon a condition…
  - If you make a blunder on the condition, the loop may never terminate.
- The while loop is prone to this particular problem
- If you know how many times you are going to run a loop at compile time, use a for loop

Comments, who needs those?

- Properly documenting a software engineering project is 100000 times more important than creating the project itself…
- In JAVA, this means proper JAVADOC and inline comments

Test as you write code!

- Design your approach on “paper” first… make a flowchart, etc.
- Write small test programs with code fragments to test your ideas
- Test integrated code as you go along… don’t wait for the last step and hope things will just work

Core JAVA

- Fundamental Concepts
- Bootstrapping
- Basic Language Syntax
- Common Caveats
- Coding Conventions

Why bother? It's arbitrary!

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code easily.
- If you publish your source code, you need to make sure it is as well packaged and clean as any other product you create.

Comments

- We have already talked about this
- JAVADOC comments at the beginning of every class, method and field
- Inline comments every other line to describe what the following line of code does

Line Length

- No line of code should be > 80 characters in length
- Line breaks should make sense…

```
longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                              - longName5) + 4 * longname6; // AVOID
```

Variables

- Initialize all variables all of the time
- Only declare one variable per line
- For local variables, use an inline comment immediately after the variable declaration to describe what the variable is for
- For fields, use a JAVADOC comment

Indentation

- All open curly braces imply that the next line should be indented
- Indentation should be uniform across all files
- Large indentations are a bad idea because you run out of room to nest blocks of code

Parentheses

- Be explicit everywhere
- Order of operations applies, but you should be explicit to make sure that anyone reading your code can easily understand what is going on

Identifiers

- Class names should start with a capital letter and have an additional capital letter for each word in the noun phrase (MyClassName)
- Methods and Variables names do not have a leading capital letter (myVar)
- Constants all all caps with _ breaking the words (MY_CONSTANT)

Clean code is good code

- The vast majority of software defects can be avoided through a combination of:
  - Thorough paper designs
  - Writing clean, standardized code
  - Proper unit testing while coding
  - Meticulous documentation