

# Intro to Java and the SDK Tools

# Virtual Machine

- Operating System independence.
- Safe, well defined, operating environment.
- Portability
- Performance Issues

# Objects, Objects, Objects

- In Java, *everything* is an Object.
  - I lie! (primitive types are not objects)
- Java source code is based on classes
  - in general: one public class defined in one file.
- There are many pre-defined classes
  - too many !?!
  - *almost* everything is written in Java

# OOP Development

- Object-Oriented *Components*
  - Formalized in J2EE - Java Beans
- Incremental Development
- Unified Modeling Language (UML)

# Features

- Dynamic Memory Management
  - garbage collection
- Threads, synchronization primitives
- Error handling (exceptions)
- Network support
- Security
- Extensive library.

# Java vs. C++

- Similar syntax/control structures.
- No preprocessor or include files.
- No pointers
- No global variables
- No struct or union types.
- All primitive types have well defined size.

# More Java vs. C++

- No operator overloading.
- Single inheritance only
  - there is another approach used - interfaces.
- Error handling is well defined (and somewhat enforced!).
- No memory leaks!

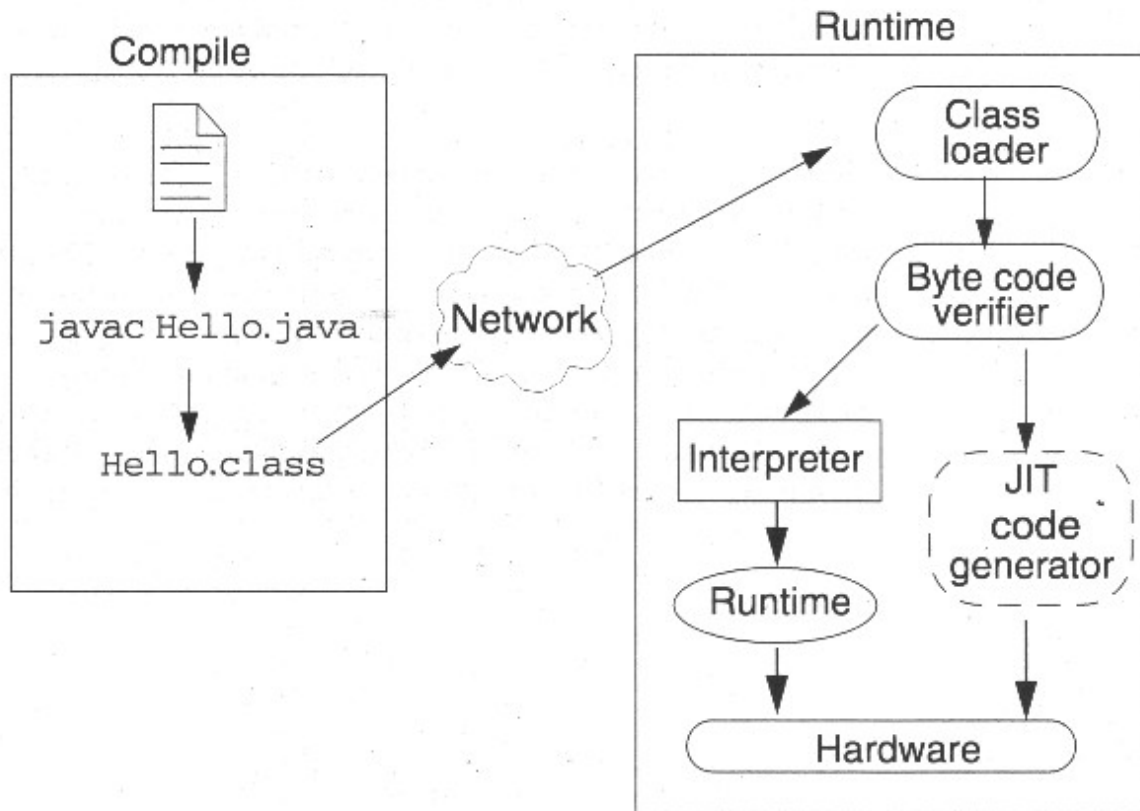
# and more...

- Safety designed in to the language and VM
  - bytecode verification
  - array access bounds checking
  - security manager/ security policies

# Performance Issues

- Using a VM is slower than compiling to *native* instructions.
  - JIT compilers convert Java Bytecode to machine language.
- Safety/Security slow things down
  - all array accesses require bounds check
  - Many I/O operations require security checks

# The big picture



# Java Strengths

- Simplicity (the language itself)
- Networking
- Object model
  - Graphical User Interface (GUI) programming
  - Large and Very Large systems.
  - Portable libraries
- Code Documentation (and other) tools.

# SDK Tools

- **javac**: the Java compiler.
  - Reads source code and generates bytecode.
- **java**: the Java interpreter
  - Runs bytecode.
- **jar**: Java Archive utility
- **javadoc**: create documentation from code.
- **jdb**: Java debugger (command line).
- There are others...

# The Java Compiler

- Usage: **`javac filename.java`**
  - You can also do: **`javac *.java`**
  - Creates **`filename.class`** (if things work)
  - Use **`"-g"`** to compile for use with the debugger.

# The Java Interpreter

- Usage: **java *classname***
  - You tell the interpreter a class to run, not a file to run!
    - It uses the CLASSPATH to find the named class.
    - The named class should have a method with prototype like:
      - **public static void main()**

# jar

- Like Unix tar comand.
- Used to create (and extract from) an archive file:
  - collection of files.
  - compressed.
- Java can find classes (bytecode) that are stored in jar files.

# jar usage

- To extract files:

```
jar xf filename.jar
```

- To list files:

```
jar tf filename.jar
```

- To create an archive:

```
jar cf filename.jar file1 file2 dir1  
dir2 ...
```

# javadoc

- Creates documentation from properly commented Java source code.
- The output of javadoc includes HTML files in the same format as the Java SDK documentation.
  - we all need to get used to this format...
  - learning to find and understand the documentation on classes/methods is 1/2 of learning Java!

# Using the tools

- We will look at these and other tools in more detail later...
- For now (HW1) we just need to be able to compile and execute a simple Java class.