**CS112 Lecture: Introduction to Java**

Last Revised 1/13/04

**Objectives:**

1. To introduce the Java programming language
2. To introduce the basic structure of a Java application
3. To introduce the notion of an applet and how it differs from an application
4. To introduce drawing using the paint() method and a Graphics object

**Materials:**

1. Transparency of Java Implementation Structure
2. Examples from Wu ch. 2: Ch2Sample1.java, Ch2ShowMessageDialog.java plus compiled .class files for each
3. Example of an applet: AppletDemo.java, .class ; html file
4. Transparency of Wu figures 2.4 and 2.5
5. Transparency of Wu Table 5.4
6. Demonstration of some of the draw and fill methods of Graphics: DrawingDemo.java

**I. Introduction to Programming Languages**

A. At the start of the course, we defined programming. What was that definition?

**ASK**

“Programming is the process of spelling out the steps needed to accomplish some task in a form that can be *interpreted* by a computer system.”

B. This, of course, creates the question “what form should a program take in order to be capable of being interpreted by a computer system?”

1. In CS111, you learned that computer hardware (a CPU chip) is capable of interpreting programs written in binary machine language, and you saw a specific example of such a language - the language of the (hypothetical) Pip system. Although real CPU's are much more sophisticated than Pip, there is no major difference in kind between the machine languages of real CPU chips and Pip - i.e. the fundamental operations in both cases involve things like copying a number from one location to another, performing a basic arithmetic operation on two numbers, or comparing two numbers.
2. As you no doubt observed in lab, writing programs in machine language has lots of disadvantages:

ASK

a) It is tedious.

b) It is error-prone.

c) Correcting such programs when errors are found is extremely difficult.

d) Modifying such programs is extremely difficult.

e) Perhaps most important of all, the machine language of each family of CPU chips is unique to that family. Thus, a program written for one family of chips (e.g. the PowerPC used in Macintoshes or the Intel 80x86 used with Wintel systems and most Linux systems) is totally useless for some other family.

3. For this reason, very early in the history of computer science, various higher-level programming languages were developed. Though still not as natural as English, they are a whole lot simpler to use than binary machine language. In CS111, you worked with one such language - Java Script - and the text mentions about 15 others.

In addition to being much easier to use, higher level languages have the advantage of being designed to be cross-platform - i.e. the same program should be capable of being used on multiple families of CPU.

C. With the use of higher-level languages, a new problem arises. The CPU can only interpret its binary machine language - so how does it execute a program written in a higher-level language?

Two approaches to solving this problem have been widely used.

1. Software interpretation. A special program called an interpreter is written for a particular chip, in the binary machine language of that chip. This program reads the higher-level language program one step at a time and interprets and carries out that step.

2. Compilation. A special program, called a compiler, is used to translate the higher-level language program into the binary machine language of the target system.
3. The key difference between the approaches is how often a given higher-level language statement is “understood”. With an interpreter, it is “understood” each time it is executed. With a compiler, it is “understood” once, and then translated.

Example:

Say to a student who knows German: “Heben sie die hand, bitte” (“Raise your hand, please”)

Ask the same student to translate into English, hand the translation to another student to interpret.

4. Often, one approach or the other is the preferred approach for a given higher-level language.

Example: JavaScript is always handled by a software interpreter (normally incorporated into a web browser). Most languages (including C, C++, Pascal, and Ada) are handled by a compiler.

However, with some languages (e.g. BASIC, Prolog) both approaches are used in different cases.

D. Note that, in either case, the relevant software (software interpreter or compiler) is specific to both the higher-level language being used and the platform.

Example: a BASIC interpreter that runs on a VAX is useless on a Pentium-based system and vice-versa. A BASIC interpreter for any system is useless if your program is written in JavaScript.

Example: a C++ compiler that produces binary machine language for a Pentium would be useless if you wanted your program to run on a Macintosh, and vice versa, and a C++ compiler for any machine would be useless if your program were written in Ada.

II. Introduction to Java

A. Although this course is intended as an introduction to computer programming, it is necessary for us to use a specific programming language as a vehicle for accomplishing our goal. Since the language we are using is Java, it seems good to say a few introductory things about this language.

B. The Basic Approach of Java Contrasted with other Programming Languages
1. Java’s approach to some key issues differs in some significant ways from the approach taken by my most programming languages that have proceeded it. These differences have to do with the relationship between the programming language and the computer system, or platform on which the program is run.

2. One such issue has to do with the way that a Java program is translated into an interpretable form.

   a) Java is distinctive among programming languages in that it uses a combination of compilation and interpretation.

      (1) Part of the Java standard is a definition for the machine language of a Java virtual machine (JVM) - a hypothetical CPU which does not actually exist in hardware (but which presumably could if someone chose to build one - indeed hardware implementations of a subset of this language do exist)

      (2) The java compiler translates Java programs into the binary machine language of the JVM - commonly called bytecodes or J–codes. This language is the same regardless of what actual physical CPU will run the program.

      Therefore, one Java compiler can be used for any machine (though there are multiple compilers available due to competition, etc.)

      (3) To run Java bytecodes on a given platform, one needs an implementation of the JVM for that machine. This is actually a software interpreter for the “machine language” of the JVM.

      Example: On recent Macintoshes, the JVM is included as part of the Mac Operating system. The various development environments we are using in lab serve as interfaces to the built in JVM.

      (4) Note that the same class files will run on any platform that has Java support. This makes programs written in Java truly portable across platforms.

3. A second key distinction of Java has to do with the user interface model.

   a) Most programming languages do not provide direct support for GUI-related tasks such as displaying a window or responding to mouse clicks. Instead, such tasks are handled through an application programming interface (API) that is part of the operating system of the host platform.
(1) Macintoshes come with an API called the Macintosh toolbox.

(2) On Windows platforms, one uses an API called the Microsoft Foundation Classes

(3) On Unix/Linux systems, there is a windowing API called X–Windows.

A consequence of this is that even though there are compilers for major languages for all major platforms, the portion of a program that deals with the user interface must be totally rewritten when porting a program from one platform to another.

b) Java, however does provide direct support for GUI-related tasks through a standard set of class libraries that are present on all Java systems. Thus the same Java code that puts up a window on a Macintosh will put up a similar window on a Wintel machine or a Linux box.

Actually, in the evolution of the Java, there have been two different approaches taken to this issue.

(1) From the beginning, the Java libraries have included a package called the abstract windowing toolkit (AWT), which provides “glue” between Java and the native platform API. Each of the facilities provided by the AWT is actually implemented by a native “peer” - e.g. a Java Button is implemented by a native “ButtonPeer” which is platform-specific.

This has two consequences:

(a) The functionality provided by AWT is limited to capabilities that are available on all platforms on which Java runs.

(b) The exact appearance of the interface (look and feel) will vary from platform to platform.

(2) Several years ago, a new package called “Swing” was added to Java. Swing relies on a native peer only for top level GUI windows - everything else is implemented in Java. Again, this has two consequences.

(a) The set of capabilities provided by Swing is not limited by what is common to all platforms on which Java runs. Swing has a much wider set of capabilities than AWT.
(b) The look and feel of the interface is not platform-dependent. By default, Swing imitates the look and feel of the platform on which the program runs (e.g. Swing looks “Maccy” on a Macintosh). But it is also possible to change the look and feel of a Swing-based interface to look like some other platform, or to use a distinctive “Java” look and feel (called “Metal”). (However copyright issues prevent Swing from being made to look like a Macintosh on any platform other than a Macintosh or like Windows on any platform other than Windows).

(3) Classes that are part of the Swing package often have names identical to that of corresponding classes in the AWT package, except that the name begins with “J” to indicate that the class is written in pure Java.

Examples: Frame, JFrame; Button, JButton

(4) Prior editions of the textbook - and hence of this course - have used AWT. This year, for the first time, both the book and the course are using Swing.

4. As an aside, sometimes people think - from the similarity of the names - that there is some relationship between Java and JavaScript. Actually, JavaScript was originally developed by Netscape, who planned to call it LiveScript. The change of name to JavaScript was a marketing ploy (according to Flanagan - JavaScript, the Definitive Guide.) The languages do have a lot of surface similarity due to both syntaxes being derived from C++, but the similarities can be deceiving.

C. The Java system consists of three major components:

1. The Java language itself.

2. The Java Virtual Machine - an architecture for a computer whose machine language contains just the features needed for running programs written in a language like Java.

3. The Java applications programming interface (API) - a collection of predefined classes that support useful data types and operations, input-output, and a platform-independent graphical user interface.

   a) The bulk of the Java API is written in Java.
b) There are about 22 classes in the Java API that contain host architecture dependent code that must be written specifically for each Java implementation.

4. Summary: TRANSPARENCY

5. Interestingly, the Java language itself is perhaps the least crucial part of the package. Compilers have been developed to translate other languages into Java bytecodes, with bindings to the Java API. In particular, several such packages are available for Ada.

6. A final note: the Java language and API is evolving.

   a) Java versions are commonly referred to by names like JDK 1.0.2, JDK 1.1, etc. JDK stands for "Java Development Kit", and is the name Sun has used for its releases of the Java compiler and other tools.

      (1) JDK 1.0.2 was the first version to be widely used.
(2) There were significant changes made to the language and the API between JDK 1.0.2 and JDK 1.1.x - but the JVM was unchanged.

(3) A smaller set of modifications led to the creation of JDK 1.2.x - which Sun now prefers to call “Java 2”. The differences between 1.1.x and 1.2.x are all in extensions to the API

(4) 1.2 has been succeeded by 1.3 and 1.4.

(5) The most recent version is JDK 1.4.2

(6) We are currently using 1.4.1.

b) There is ongoing discussion about extending the Java language to provide features not currently present.

D. Java systems provide support for two sorts of executable programs:

1. Java applications, that run stand-alone on an individual machine. With one exception, these are no different in principle from applications written in other languages - they can be bought, downloaded, or developed locally. (The key difference is that a JVM implementation and library must reside on the platform on which the application is to be run - but these are either standard or readily available for modern platforms.)

2. Java applets - mini-applications that run in the context of another program such as a web browser, in a window provided by that program.

   a) These are typically downloaded over the net on demand when the user clicks on an appropriate link.

   b) Unlike Java applications, Java applets downloaded over the net have restricted capabilities (enforced the the Java implementation on the client platform) that are designed to prevent malicious actions - e.g.

      (1) A bytecode verifier checks the J-Code as it is downloaded.

      (2) The applet is limited to accessing files on the server from which it was downloaded - it may not access files on the client system or elsewhere on the net.

E. In CS112, our focus will be on the Java language, with some limited discussion of the API. We will focus on developing applications, though we will do some work with applets. (The book deals with applications exclusively). We will do a lot more with the API in CS211.
III. Introduction to Writing Applications in Java

A. As we have already seen from our work with Karel, a Java application always has a class that contains a class method called main(). At the time this method is invoked, there are no objects in existence (hence the reason why it must be a class method). This main method must

1. Create the objects needed by the program (or at least some of them)

2. Initiate computation.

We will now talk about each of these tasks in turn

B. The number and type of objects needed by a given program varies widely from program to program. In general, though these objects can be placed into several categories, and most programs will have one or more objects in each category.

1. A *boundary object* is used to manage interaction between the program and the outside world (human user or another system).

   *Example:* What kind(s) of objects fulfilled this function in the Karel programs we were working with?

   *ASK*

   a) The window representing the robot’s world that we saw on the screen was a boundary object.

   b) The window containing the speed and resume/stop controls was actually a window object containing two other objects: a button object and a slider object. All were boundary objects.

   c) A GUI application will always have at least one boundary object - the top-level window (frame) in which the application is displayed. (It is possible to write non-GUI applications in Java, too, though we seldom will.)

2. An *entity object* is used to represent a thing in (concrete or abstract) in the domain of the program, with methods providing operations on it.

   *Example:* What kind(s) of objects fulfilled this function in the Karel programs we were working with?

   *ASK*
a) The object(s) representing the robot(s)

b) The objects representing the world. (We didn’t work with these directly; however, the call to World.readWorld() resulted in creating the objects needed to represent a specific scenario.

3. A control object is used to manage the flow of computation, using the other objects.

Example: What kind(s) of objects fulfilled this function in the Karel programs we were working with?

ASK

a) The robot programs we wrote were part of a control object. They were part of the class that contained the main method. The main method created an instance of the control object, and arranged for its run() method to be called to manage the computation.

b) In this case, it would have also been possible to have the robot object(s) do double duty (as both entity and control objects) by having our subclass of Robot include an appropriate method that could have been started from main(). (In this case, no object of the class containing main would ever have been created.)

C. One place where programs differ widely is how the computation itself is managed.

1. In some cases, the main() method also includes the code needed to perform the computation (or at least the highest level of it)

2. In some cases - as we did with Karel - the main() method creates a control object which has a method that manages the computation.

3. In GUI applications, it is common for each of the active GUI components (buttons, menu items, etc.) to be associated with a control object that performs the appropriate computation when the GUI component it is connected to is activated. This style of programming is called event-driven programming, and is something we will introduce in April.

D. An example of a Java application: Ch2Sample1 from Wu ch. 2

1. Show source code

   a) What boundary, entity, and control object(s) does this program have?
ASK

(1) The object myWindow (of class JFrame) is the only boundary object.

(2) There are no entity or control objects. This is a very simple program

b) Simple as this program is, it illustrates a number of features that are - or should be - part of every java application

ASK

(1) Opening comment

(2) import statement

(3) definition of class containing main()

(4) definition of main() method - note standard signature (will always be the case for main())

(5) object declaration

(6) object creation

NOTE: In our Karel programs, we combined object declaration and object creation into one step. That could be done here also - e.g. by

`JFrame myWindow = new JFrame( );`

However, conceptually object declaration and object creation are two distinct things.

TRANSPARENCY - Wu figures 2.4, 2.5

(7) sending a message to an object

2. (Run program). Note that application quits when we click the close box on the main window.

  a) Where is this behavior defined in the source code?

ASK
The behavior is not in the source code - it is part of the JFrame class.

b) This illustrates a key feature of OO programming: wherever possible, we make use of already-existing classes that have the behavior we may need, rather than inventing it ourselves.

E. Ch2ShowMessageDialog.java from Wu ch. 2

1. DEMO Execution

2. Walk-through source code line by line.

IV. Introduction to Writing Applets in Java

A. The other kind of program one can write in Java is an applet - which stands for “little application”. An applet is never a stand-alone program - it is always run by some other program.

1. Most typically, applets are run by web browsers.

2. Most Java implementations also include an applet viewer program which can be used to run applets for testing purposes or when running a full web browser is undesirable.

B. Because an applet is run inside another application, it differs from applications like those we looked at in several ways:

1. An applet is always an instance of a subclass of class java.applet.Applet - often a subclass of javax.swing.JApplet, which is in turn a subclass of java.awt.Applet.

2. An applet is an object, which the application that runs it creates.

3. In fact an applet is an boundary object.

4. An applet does not have to construct its own window; it is displayed in a window provided for it by the application that runs it. (However, applets can create additional windows if they need to.)

5. An applet does not have a main() method - the application that runs it fulfills that responsibility. An applet can, in fact, rely on its host application to invoke several of its methods. These methods are all defined as “do-nothing” methods in class Applet, but can be overridden by the actual subclass the applet belongs to. In particular, the paint() method is always called to draw the applet on the screen.
C. Example of an Applet - AppletDemo.java

1. Walk through source code

2. Demo running it as an applet using BlueJ

   Note how we are asked to specify a size for the window - since an applet always runs in some programs window (in this case, a program called appletviewer used as a tool during applet development).

   Specify 100 high by 300 wide.

3. Demo running it using a web browser

   a) Show AppletDemo.html using BBEdit Lite (in folder html file for applet)

   b) Drag to Safari

V. Some comments on using the paint() method.

A. The applet demo made use of a method called paint() that is used to actually draw the applet on the screen. The application that provides the frame in which the applet is displayed is also responsible for ensuring that this method gets called.

B. This method is not unique to applets. Any visible “widget” (in either AWT or Swing) has such a method that is used to draw it on the screen.

   In particular, a frame has a paint method. The default implementation of it - which can be overridden - is to simply call the paint method of each “widget” that is part of the frame.

C. The basic function of the paint() method is to provide access to a Graphics object, which can, in turn, be asked to do various kinds of drawing operations.

   1. Typically, the Graphics object does drawing on your screen

   2. When you try to print something on a printer, the paint() method is called with a Graphics object that “draws” on the printer instead.
D. The example we just looked at showed one kind of drawing operation a Graphics object can perform. Actually, it can perform about 20 different kinds of drawing operations.

1. TRANSPARENCY - Wu Table 5.4

2. SHOW code for DrawingDemo.java. Note that we override the paint() method of the frame, instead of putting individual widgets in it. We will do something similar in lab 3.

   a) What do you think it will do?

      ASK

   b) DEMO