Framework: set of classes and interfaces that provide the framework for building an application

- Framework handles control flow (complex in GUI)
  - Ex. Java framework for developing GUIs: Abstract Window Toolkit (AWT), Swing, & Java Event Model
- Software developer reuses or customizes parts of framework instead of developing everything from scratch
  - Customization by inheritance (creating subclass): developer must understand the inheritance hierarchy of the class that he is extending
  - Customization by composition (implementing interface): no need to understand rest of framework

Benefits of using framework:
- Application using framework may be more reliable because framework has been tested
- Less expertise and time (compared to building from scratch)
- Consistent "look-and-feel" to interface
Disadvantages of using framework:
- Effort to learn framework
- Give up some control over design: e.g. control flow, look and feel
- May not even be able to customize framework to meet requirements of application
- Forced to change application to keep up with new versions of framework

Java version 1.1 AWT: provides GUI components
- Component: visual object that responds to user action
- Ex.: buttons, panels, dialog boxes, menus, text fields, lists
- heavyweight components: implemented by local platform's native GUI system
  - E.g. same button may look different when program runs under Windows & under Macintosh
  - Look-and-feel consistent with OS user is used to
  - Less portable: behavior may vary across platforms

Java version 1.2 (Java 2) Swing components
- more types of components than AWT
- lightweight components: not depend on platform
  - consistent look-and-feel across platforms
  - can be chosen at runtime
- can only be run by Java version 1.2 interpreters (available at UNCG yet?)
Benefits of using Java for GUI development:

1. **Object-oriented GUI framework**

2. **Robustness**: compile-time and run-time error checking (exception handling), automatic garbage collection, no explicit pointers

3. Features supporting Internet applications
   - **Concurrent** programming: perform more than one task at a time (e.g. send data over network while update screen)
   - **Portability**: compile once, run on any machine that has Java Virtual Machine (interpreter)
   - **Security**: programs running on user’s computer via web browser cannot access files on user’s computer
   - Programs can be loaded over network when needed

Disadvantages of using Java for GUI development:
- Not rapid prototyping language - not WYSIWYG
- Designed for programmers

### Development cycle

- **mycode.java** (source code)
- **Compile:**
  - `javac mycode.java`
  - `mycode.class` (byte code)
- **Modify source code**
- **Run on Java Virtual Machine**
  - **Application:** `java mycode`
  - **Applet:** `appletviewer <url of web page>`

**Byte code interpreted (translated into machine instructions) at runtime by Java Virtual Machine**
Two kinds of Java program
1. Stand-alone **application:**
   - Execution starts in method **main** with signature:
     ```java
     public static void main(String args[])
     ```
   - Does not run over Internet, so file IO not restricted
2. **Applet:**
   - **NO** method **main** needed
   - File IO restricted for security
   - not designed to be run on its own, i.e. runs only from webpages referring to compiled applet (e.g. my.class below):
     ```html
     <html>
     ...
     <body>
     <applet code="my.class" ...></applet>
     ...
     </body></html>
     (Warning: HTML not case-sensitive but Java is!)

Note about above diagram: if standard Java classes used by applet are already on the user's machine, then they do not need to be downloaded at runtime over the Internet!

```java
class Block {
    int a, b, c;
    int volume;

    // Constructor
    Block(int i, int j, int k) {
        a = i;
        b = j;
        c = k;
        volume = a * b * c;
    }

    // Return true if ob has same volume as this object
    boolean sameVolume(Block ob) {
        if (ob.volume == volume) return true;
        else return false;
    }

    // Evil method added to class by N. Green - what does it do?
    void flatten(Block ob) {
        ob.a = 0;
        ob.volume = 0;
    }
}
```

```java
class PassOb {
    public static void main(String args[]) {
        Block ob1 = new Block(10, 2, 5);
        Block ob2 = new Block(4, 5, 5);
        System.out.println("ob1 same volume as ob2: "+
            ob1.sameVolume(ob2)); // returns true!
        ob1.flatten(ob2);
        System.out.println("ob1 same volume as ob2: "+
            ob1.sameVolume(ob2)); // returns what now?!
    }
}
```

- simple types like `int` are passed **call-by-value**
- objects (like `ob2` above) and arrays are passed **call-by-reference**
Review of Java variables

Syntax (BNF) for declaring variables:

\[
\begin{align*}
&\{<\text{modifier}>\}^* \ <\text{type}> \ <\text{varname}> \ \{<\text{initialization}>\} \ ; \\
&<\text{modifier}> \ := \ \text{static} | \ \text{final} | \ \text{public} | \ \text{private} | \ldots \\
&<\text{type}> \ := \ <\text{built-in}> \ | \ <\text{class}> \\
&<\text{built-in}> \ := \ \text{int} | \ \text{double} | \ \text{boolean} | \ \text{char} | \ldots \\
&<\text{varname}> \ ::= \ \text{letter} \ \{\ \text{letter} | \ \text{digit} | \ \$ | \ \text{underscore} \ \}^* \\
&<\text{initialization}> \ ::= \ "=\" \ <\text{value}> \\
\end{align*}
\]

Note: all variables must be declared inside a class (unlike C++)

Class variables (static):

- Declared with static keyword in declaration (“static extent”: memory allocated at compile-time)
- Only one value for the whole class, not one for each object

```java
class Demo {
    static int numDemo = 0; // how many Demo objects currently exist (class variable)
    static final int maxDemo = 60; // (class constant)
}
```

Instance variables:

- Stores value of field/member of object
- Declared by absence of static keyword
- Good programming style to declare with private access and define public instance methods for getting/setting

Local variables

- Declared inside block { … }
- Local scope: only visible from point of declaration to end of block
- Shadows other visible variables with same name & wider scope, e.g.:
```java
{ int count = 0;
    int count = 1;
    System.out.println(count); // prints 1
}
System.out.println(count); // prints 0!
```
**Message Passing in Java**

When an object (the **sender** or Client) needs a service, it can send a **message** (and **arguments**) to another object (the Receiver or Server or **target**) that provides the service.

**Syntax (BNF):**

```
{<target>} . <message> "(" {<argList>}* ")"
```

where `<argList>` is one or more arguments separated by ",",

**UML sequence diagram:**

![UML sequence diagram]

**Examples:**

- `panel_1.add(BorderLayout.NORTH, button_1)`
  - sender: an object of the class containing this statement
  - target: panel_1
  - message: add
  - arguments: two objects (a layout and a button)
- `add(BorderLayout.NORTH, button_1)`
  - target: the same object as the sender
- `this.add(BorderLayout.NORTH, button_1)`
  - target: `this` (*this* is Java keyword that always refers to sender, so the sender is sending a message to itself)

**note:** `this` can be passed as an argument too, e.g.,

```java
someObj.someMethod(this);
// passes this object (call-by-reference) to someObj
```