# CS 66-460
# Introduction to Graphical Human Machine Interfaces

*Java (Part 2)*

MONDAY, APRIL 7, 1997

Instructor: G. Bowden Wise

Rensselaer Polytechnic Institute
Spring 1997

## Outline

- Java language features
- Object-oriented programming
- Java Applets
- Creating user interfaces with Java AWT
- Example programs

# Java
## *Comments*

Java supports to styles of comments, both borrowed from C and C++:

- for comments on one line use `//`:

```
// this is a comment
// this is a second comment
```

- for long comments use `/*` and `*/`:

```
/*
 This is a long comment.

 Which is useful when you have lots
 of text spanning several paragraphs.
 */
```



# Java
## *Data Types*

**Integers**

```
byte     1 byte  [-128, +128]
short    2 bytes [-32,768, +32,768]
int      4 bytes [-2,147,483,648, +2,147,483,647]
long     8 bytes [-9,223,372,036,854,775,808L,
                   +9,223,372,036,854,775,808L]
```

**Floating Point Numbers**

```
float    4 bytes +- 3.40282347E+38F
                 (7 significant digits)
double   8 bytes +- 1.79769313486231570E+308
                 (15 significant digits)
```

**Characters**

```
char     2 bytes  UNICODE
```

**Boolean**

```
boolean  1 byte   true, false
```

## Java

*Conversions Between Numeric Types*

In general, binary operations on numeric types are converted to the largest type when there are variables of mixed numeric types

Java will convert operands in the following order:

- if any operand is a `double`, all are considered `double`
- if any operand is a `float`, all are considered `float`
- if any operand is a `long`, all are considered `long`

Conversions from a larger type to a smaller one require an explicit cast:

```
double x = 9.997;
int    n = x;       // compile time error

int    n = (int) x; // ok: explicit cast
```

Java allows assignments from one type to another provided the resulting variable is large enough:

`byte .. short .. int .. long .. float .. double`

You can always assign a variable of a type that is to the left to the type on its right in the above list.



## Java

*Conversions and Boolean Types*

Java does not allow `boolean` values to be converted to any numeric types. C/C++ programmers may find this a nuisance, no longer can you do this:

```
// function foo() returns an int
if ( foo(a) )
{
}
```

instead you must do this:

```
if ( foo(a) != 0 )
{
}
```

Java does not allow assignments within boolean expressions. It is not possible to do this:

```
if ( a = 10 )
{
}
```

# Java
## *Constants*

- Java is limited when it comes to defining constants
- You cannot make a constant local to a method
- Constants can only be created for the entire class; all methods can access them.
- Prefix a variable declaration with `static final` to create a constant:

```
class MyClass
{
   static final double e=1.602e-19;

   public static void main (String args[])
   {
      System.out.println
      ("fundamental electron charge: " + e);
   }
}
```

- To make a constant `private` to a class

```
   private static final double e=1.602e-19;
```

- Note: `const` is a Java keyword, but it does nothing



# Java
## *Variables*

- Java reserved words cannot be used for variable names
- Names must begin with a letter followed by a sequence of one or more letters or numbers, underscores, or dollar signs.
  - e.g., regular ression: `[a-zA-Z][a-zA-Z0-9_$]*`
- Java follows same conventions as C/C++ for declarations:

```
int     i;
char    c, d;    // comments are helpful
double  amount;
boolean stopped; // is the simulation stopped?
```

# Java
## *Assignment and Initialization*

- It is a good idea to explicitly intitialize all variables
- Avoid un-initialized variables

- you may initialize after a declaration:

```
int i;        // declaration
i = 100;      // initialization
```

- or you may initialize with declaration:

```
int  i = 100;
char c = 'A';
char a = '\u0041'; // decimal 65 is ASCII 'A'
```



# Java
## *Operators*

| | | | | |
|---|---|---|---|---|
| 1 | ++ | arithmetic | R | pre or post increment |
| | -- | arithmetic | R | pre or post decrement |
| | +, - | arithmetic | R | unary plus or minus |
| | \~ | integral | R | bitwise complement (unary) |
| | ! | boolean | R | logical complement (unary) |
| | (type) | any | R | cast |
| 2 | *, /, \% | arithmetic | L | multiplication, division, modulus |
| 3 | +, - | arithmetic | L | addition, subtraction |
| | + | String | L | concatenation |
| 4 | << | integral | L | left shift |
| | >> | integral | L | right shift with sign extension |
| | >>> | integral | L | right shift with zero extension |
| 5 | <, <= | arithmetic | L | less than, less than or equal |
| | >, >= | arithmetic | L | greater than, greater than or equal |
| | instanceof | object, type | L | type comparison |
| 6 | == | primitive | L | equal (same values) |
| | != | primitive | L | not equals (different values) |
| | == | object | L | equal (same objects) |
| | != | object | L | not equal (differnt objects) |
| 7 | & | integral | L | bitwise AND |
| | & | boolean | L | boolean AND |
| 8 | ^ | integral | L | bitwise XOR |
| | ^ | boolean | L | boolean XOR |

# Java
## *Operators* (continued)

<table>
<tr><td>9</td><td>|</td><td>integral</td><td>L</td><td>bitwise OR</td></tr>
<tr><td></td><td>|</td><td>boolean</td><td>L</td><td>boolean OR</td></tr>
<tr><td>10</td><td>&&</td><td>boolean</td><td>L</td><td>conditional AND</td></tr>
<tr><td>11</td><td>||</td><td>boolean</td><td>L</td><td>conditional OR</td></tr>
<tr><td>12</td><td>?:</td><td>boolean,any,any</td><td>R</td><td>conditiional (ternary) assignment</td></tr>
<tr><td>13</td><td>=</td><td>variable, any</td><td>R</td><td>assignment</td></tr>
<tr><td></td><td>*=, /=</td><td>variable, any</td><td>R</td><td>assignment with operation</td></tr>
<tr><td></td><td>\%=</td><td></td><td></td><td></td></tr>
<tr><td></td><td>+=, -=</td><td></td><td></td><td></td></tr>
<tr><td></td><td><<=, >>=</td><td></td><td></td><td></td></tr>
<tr><td></td><td>>>>=</td><td></td><td></td><td></td></tr>
<tr><td></td><td>&=, ^=</td><td></td><td></td><td></td></tr>
<tr><td></td><td>|=</td><td></td><td></td><td></td></tr>
</table>



# Java
## *Control Flow*
## Conditional Statements

### `if` statements:

```
if (yourSales >= target)
   quotaMet = true;
```

or use blocks

```
if (yourSales >= target)
{
   quotaMet = true;
   bonus = 1000;
}
```

# Java
*Control Flow*
## Conditional Statements

`if () else` **statements:**

```
if (yourSales >= target)
   quotaMet = true;
else
   quotaMet = false;
```

or use blocks

```
if (yourSales > target)
{
    quotaMet = true;
    bonus = 1000;
}
else if (yourSales == target)
{
    quotaMet = true;
    bonus = 0;
}   else
{
    quotaMet = false;
    paycut = -1000;
}
```



# Java
*Control Flow*
## Indeterminate Loops

`while` **statements:**

```
while (balance < goal)
   balance += 10;
```

or use a block:

```
while (balance < goal)
{
   amt = 10;
   balance += amt;
}
```

`do` **statements:**

```
do
{
   amt = 10;
   balance += amt;
} while (balance < goal)
```

# Java
## *Control Flow*
## Determinate Loops

**`for` and `while` statements:**

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

while (i < 10)
   System.out.println ("i: " + i++);
```

with blocks:

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

while (i < 10)
{
   sum += i;
   System.out.println
        ("i: " + i + "sum: " + sum);
   i++;
}
```



# Java
## *Control Flow*
## Multiple Selections

**`switch` statements:**

```
import jave.util.Date.*;
Date today = new Date();
String day = "";
switch (today.getDay())
{
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
    day = "Tuesday";
      break;
  case 3:
    day = "Wednesday";
      break;
  case 4:
    day = "Thursday";
      break;
  case 5:
    day = "Friday";
      break;
  case 6:
    day = "Saturday";
      break;
  }
System.out.println ("Today is: " + day);
```

# Java
*Control Flow*
## Break

**Un-labeled `break` statements:**

```
while (years <= 100)
{
   balance = (balance + payment) *
             (1      + interest);
   if (balance > goal)
      break;
   years++;
}
```

**Labeled `break` statements:**

```
test: if (check(i))  {
         for (int j=0; j < 10; j++) {
            for (k=0; k < 10; k++) {
               if (a[i][j] == null)
                  break test;
            }
         }
      }
```



# Java
*Control Flow*
## Continue

**Un-labeled `continue` statements are used to break to the top of a loop:**

```
while (years <= 100)
{
   balance = (balance + payment) *
             (1      + interest);

   if (balance ==  500)
   {
      balance = 550;
      continue;
   }
   years++;
}
```

**Labeled `continue` statements:**

- are similar to labeled `break` statements

# Java
## *Arrays*

- Java arrays are **first class objects**
- Once an array has been created, you cannot change its size.
  - to change the size of arrays at run-time; use `vector`s
- You must explicitly allocate memory for arrays with `new`
- You do not need to worry about freeing that memory
- Arrays indices start from `0`

Examples:

```
int[]  I = new int[100];
for (int i=0; i < 100; i++)
   I[i] = i; // fill array with 0 to 99

int lookupTable[]  = { 1, 10, 15, 20, 25, 30};
```



# Java
## *Arrays* (continued)

- Java arrays are manipulated by reference
  This means when we pass them as parameters to a routine, the routine can change the elements

```
public class ShellSort
{
   public static void sort(int[] a)
   {
      int n    = a.length;
      int incr = n / 2;
      while (incr >= 1)
      {
         for (int i=incr; i < n; i++)
         {
            int temp = a[i];
            int    j = i;
            while ( j>= i  && temp < a[j-incr])
            {
               a[j] = a[j-incr];
               j -= incr;
            }
            a[j] = temp;
         }
         incr /= 2;
      }
   }
```

```
   public static void main (String[] args)
   {
      int[] a = new int [10];
      for (int i=0; i < a.length; i++)
         a[i] = (int) (Math.random() * 100);

      System.out.print ("Unsorted: ");
      for (int i=0; i < a.length; i++)
         System.out.print (a[i] + " ");
      System.out.println (" ");

      sort (a);

      System.out.print ("Sorted:   ");
      for (int i=0; i < a.length; i++)
         System.out.print (a[i] + " ");
      System.out.println (" ");
   }
}
```



# Java
# *Arrays* (continued)

- Java arrays can be returned from methods

```
public class LotteryDrawing
{
   public static int[] drawing (int high, int number)
   {
      int numbers[] = new int [high];
      int result [] = new int [number];

      // fill in array with numbers 1 2 3 ... high
      for (int i=0; i < high; i++) numbers[i] = i+1;

      for (int i=0; i < number; i++)
      {
         int j = (int) (Math.random() * (high - i));
         result[i] = numbers[j];
         numbers[j] = numbers[high - 1 - i];
      }
      return result;
   }

   public static void main(String[] args)
   {
      int   high = 50;
      int number = 5;

      int[] a = drawing (high, number);

      for (int i=0; i < a.length; i++)
         System.out.println (a[i]);
   }
}
```

# Java
# The `String` Class

- Java does not provide a built-in string class
- Instead, the standard Java library contains a string class

```
String empty = "";
String hello = "Hello";

// concatenation
String hello2 = hello + " world";

// substrings
String world = hello2.substring(6, 5);

// characters:
char w = world.charAt(0);

// cannot change individual characters
// compile time error
world[0] = 'W';

// but can replace whole string:
world    = "World";

// for comparing two strings:
if (world.equals(hello2))

// DO NOT USE THE == operators
// it will compare the addresses of the objects
if ( world == hello2 )
```



# Java
# The `StringBuffer` Class

- `String`s are immutable; you cannot change individual characters at run-time, only whole strings
- Use a `StringBuffer` if you must modify characters inside a string

Example:

```
class ReverseString {
  public static String reverseIt(String source) {
              int i, len = source.length();
              StringBuffer dest = new StringBuffer(len);

              for (i = (len - 1); i >= 0; i--) {
                  dest.append(source.charAt(i));
              }
              return dest.toString();
  }
}

public class Reverse
{
 static public void main (String[] args)
  {
    String s = "Reverse Me";
    String r = ReverseString.reverseIt (s);

    System.out.println ("  String: " + s);
    System.out.println ("Reversed: " + r);
  }
}
```

# Java
*Other Data Structures*

Other data structures are available from the `java.util` package:

- `Vector`
- `BitSet`
- `Hashtable`
- `Stack`



# Object Oriented Programming
*Procedural vs. Object-Oriented*

**Procedural programming**

- A program consists of a bunch of procedures and sub-procedures which operate on the data within a program
- Programmers first identify tasks and then:
  - *stepwise refinement* is used to break larger tasks into smaller ones (and those into further smaller ones), this is called the **top-down** approach
  - procedures are written for the smaller tasks, which are then used to implement the larger tasks; this is called the **bottom-up** approach.

# Object Oriented Programming
*Procedural vs. Object-Oriented*

## Object-oriented programming

- A program consists of a number of objects which have their own data which it knows how to manipulate
- Objects send messages to other objects to make them do something.
- Programmers:
  - analyze the problem domain to identify objects (e.g, nouns) and methods/tasks (e.g., verbs)
  - implement classes for the different kinds of objects
  - group similar classes into hierarchies



# Object Oriented Programming

One definition of an object-oriented langage is that the langauge provides support for the following

- **Encapsulation and data hiding**
  combining data and behavior in one package, in such a way that the underlying data is hidden
- **Inheritance**
  enables a class to obtain behavior and data from another class
- **Polymorphism**
  ensures that the correct method is called within the hierarchy since a subclass may have the method or its parent (or its parent's parent) may have the method.

  The process of deferring of which method to call is called **late binding** or **dynamic binding**

# OOP
*Some Definitions*

- A **class** is a collection of data and methods that operate on that data.
- An **object** is a particular (run-time) instance of a class.
- Objects are sent **messages** which tell them to invoke a particular task (method).
- Objects have a lifetime from their creation to their destruction.
- A class `A` may be derived from another class `B`. We say that class `A` **inherits** from `B`. Furthermore, we say that `A` is a sub-class of `B` and that `B` is class `A`'s parent or superclass.



# OOP
*Class Relationships*

There are three common relationships among classes:

- **uses**
  Class `A` *uses* class `B` if `A` manipulates or accesses methods of class `B`.
- **containment or "has-a"**
  If a class `A` has a data member of type `B`, then `A` *contains* or `has-a B`
- **inheritance or "is-a"**
  Inheritance denotes *specialization* or a refinement of an existing class. For example, a `Rectangle` is a specialed kind of `Shape`.

## Java
### *Support for OOP?*

- Everything in Java is a **class** or part of one.
- Classes have methods and data
- Does Java support OOP?
  - **Encapsulation and data hiding**
    Java provides several levels of visibility for classes: `public, protected, private protected, private, default`
  - **Inheritance**
    Java classes may inherit from another class (`class A extends B`)
  - **Polymorphism**
    Java uses late binding and dynamic method lookup to ensure the correct methods are called for an object within a class hierarchy



## Java OOP
### *Classes*

What goes in a class:

- public features
- package scope features (protected)
- private features

Within each section, we can have:

- constants
- constructors
- finalizers
- methods
- static methods
- instance variables
- static variables

# Java OOP

*The Life Cycle of an Object*

- Objects are created with `new`
- You do not have to explicitly delete an object; Java's garbage collector takes care of freeing memory used by objects which are no longer used.

When implementing classes, you implement:

- **constructors**
  there must be at least one, you may have more than one constructor, which take different arguments. This is called **overloading**.

- and optionally, a `finalize()` method
  The `finalize()` method is called just before the garbage collector reclaims the space for an object. Java will clean up any memory used by other instance variables in your object, but it cannot free up resources such as sockets, file handles, etc. Use the `finalize()` method for these kinds of resources.



# Java OOP

*Controlling Access to Class Members*

Java provides 5 levels of access control for class members – both data and methods:

- `public`
  Members are visible to any object.
- `protected`
  Members are visible only to the class that defines it and its subclasses and also visible to all classes that are in the same package as the class.
- `private protected`
  Members are visible only to the class that defines it and its subclasses.
- `private`
  Members are only visible to the class that defines it.
- `default`
  If a variable or method is declared without any of the above modifiers, it is visible only within the class that defines it, and within classes in the same package.

## Java OOP
*Controlling Access to Class Members*

When to use:

- `public`
  Use for methods, constants, and on *important* variables that should be visible everywhere. You should avoid making variables `public` since that disrupts encapsulation.
- `protected`
  Use for variables and methods that you want classes in the same package to have access to **and** to allow sub-classes outside the package to have access to.
- `private protected`
  Use for methods and variables that are not important to the class but may be important for those extending the class. If you also want those members accessible to all classes within the same package, use `protected` instead.



## Java OOP
*Controlling Access to Class Members*

- `private`
  Use for methods and variables that are only used inside the class and should be hidden from everywhere else.
- `default`
  Use for methods and variables that you want to be hidden outside the package, but which you want cooop-erating classes within the same package to have access to. If you wnat those members to also be accessible to subclasses outside the package, use `protected` instead.

## Java OOP
### *Class Declarations*

When we define a class, we specify

- it's name
- which classes it inherits from (optional)
- which interfaces it implements (optional)
- whether it is `public`, `abstract`, `final` (optional)

In general, a class declaration in Java looks like:

```
[modifiers]  class ClassName [extends SuperClass]
                             [implements InterfaceNames]
{
}
```

Modifiers:

- `public`: a class is visible outside it's package
- `abstract`: a class has abstract methods
- `final`: a class cannot be used as a superclass



## Java OOP
### *Class Data Member Declarations*

When we define class data members, we specify

- it's type and name
- it's visibility
  `public, protected, private protected,`
  `default, private`
- whether it is
  - `static`: is a class (rather than instance) variable, all instances will share this variable
  - `final`: the variable is a constant
  - `volatile`: the variable's value may change asynchronously and the compiler should not do optimizations with it

In general, a class data member in Java looks like:

```
accessSpecifier] [static] [final] [volatile] type variableName
```

Examples:

```
static final double PI = 3.14159; // constant
private double width;             // private instance data
```

# Java OOP
*Class Member Method Declarations*

When we define class methods, we specify

- it's return type, name, and arguments
- it's visibility
  `public, protected, private protected, default, private`
- whether it is
  - `static`: is a class (rather than instance) method, can only access class variables
  - `abstract`: the method is not implemented, subclasses must implement the method
  - `final`: the method cannot be overridden by subclasses
  - `native`: the method is implemented in native code (e.g., C)
  - `synchronized`: indicates the method is a critical section
- exceptions it may throw



# Java OOP
*Class Member Method Declarations*

In general, a class data member in Java looks like:

```
[accessSpecifier] [static] [abstract] [final] [native] [synchronized]
                  returnType methodName ([paramlist])
                  [throws exceptionsList]
```

Examples:

```
public static int getX() { return x; }

public double area() { return width * height; }

public abstract void draw ();

public void OpenFile () throws IOException
{
    ...
}

public void sort (int[] a)
{
    ...
}
```

# Java OOP
## *Abstract Classes*

- Use **abstract classes** to factor out the common operations (and data) to a common base class.
- Abstract contain methods which do not (yet) have an implementation.
- Abstract classes cannot be instantiated.
- Classes derived from an `abstract` class must provide implementations for the `abstract` methods. If it does not, then the derived class is also abstract.
- Similiar to C++ *virtual functions*
- Abstract classes have at least one `abstract` method.

Example: a hierarchy of shapes with methods for:

- double area()
- double circumference()



# Java OOP
## *Abstract Class* Example

```
abstract class Shape
{
  public Shape (String name) { shName = name; }
  public abstract double area();
  public abstract double circumference();
  public String getName () { return shName; }
  private String shName;
}

class Circle extends Shape
{
  public Circle ()         { super("Circle"); this.r = 1.0; }
  public Circle (double r) { super("Circle"); this.r = r;  }
  public double area()          { return java.lang.Math.PI * r * r; }
  public double circumference() { return 2 * java.lang.Math.PI * r; }
  public double getRadius() { return r; }
  protected double r;
}

class Rectangle extends Shape
{
  public Rectangle ()
  { super("Rectangle");  this.w = 1.0; this.h = 1.0; }
  public Rectangle  (double w, double h)
  { super ("Rectangle"); this.w = w; this.h = h; }
  public double area () { return w*h; }
  public double circumference() { return 2*(w + h); }
  public double getWidth()  { return w; }
  public double getHeight() { return h; }
  private double w;
  private double h;
}
```

```
public class shapes
{
  public static void main(String args[])
  {
   Shape[] shapes = new Shape[3];

   shapes[0] = new Circle (2.0);
   shapes[1] = new Rectangle();
   shapes[2] = new Rectangle(4.0, 2.0);

   double total_area = 0;

   for (int i=0; i < shapes.length; i++)
     {
        total_area += shapes[i].area();
        System.out.println (shapes[i].getName() + ": area= " +
                            shapes[i].area()    + ", circumference= " +
                            shapes[i].circumference());
     }
   System.out.println ("Total area = " + total_area);
  }
}
```



# Java OOP
*Compiling Java Programs*

- The previous shape example is the complete code for a **Java program**, not an applet.
- Save all of the code in a file called: `shapes`
- There can only be **one** public class in a file; the filename must match the class name and is case-sensitive (e.g., `public class MyClass` would have filename: `MyFile.java`)
- Compile it, creating java bytecode: `shapes.class`

```
$javac shapes.java
```

- Run the java bytecode file:

```
$ java shapes
Circle: area= 12.5664, circumference= 12.5664
Rectangle: area= 1, circumference= 4
Rectangle: area= 8, circumference= 12
Total area = 21.5664
$
```

# Java OOP
## *Interfaces*

- Suppose we want to create a shape class that is also drawable.
- We already have our shape classes with methods for `area()` and `circumference()`
- Now we want all drawable shapes to have methods like:
  - `void setColor (Color t)`
  - `void setPosition (double x, double y)`
  - `void draw (DrawWindow dw)`
- Java does not support *multiple inheritance*, only single inheritance, so we cannot create a class `DrawableRectangle` by inheriting from both `Rectangle` and `Drawable`.
- Java's solution is to use **interfaces**

```
public interface Drawable
{
 public void setColor (Color t);
 public void setPosition (double x, double y);
 public void draw        (DrawWindow dw);
}
```



# Java OOP
## Using *Interfaces*

- An **interface** is essentially a way to promise that other classes will implement methods with certain signatures.
- An interface may have variables, but they must be constant (i.e., `static` and `final`)
- Classes use an interface by implementing it; by using the `implements` clause in the class definition
- Classes may implement several interfaces; hence, Java supports multiple inheritance in this way.

Example:

```
public class DrawableRectangle extends Rectangle
                               implements Drawable
{
 public DrawableRectangle ( double w, double h) { super(w,h); }

  public void setColor (Color t) { this.c = t; }
  public void setPosition (double x, double y) { this.x=x; this.y = y; }
  public void draw        (DrawWindow dw)
  { dw.drawRect (x, y, w, g, c); }

  private Color c;
  private double x, y;
}
```

## Java OOP
*Class Design Hints*

- Always keep data private.
- Always initialize data.
- Don't use too many basic types in a class.
- Not all fields need accessors and mutators.
- Use a standard form or style for class definitions.
- Break up classes with too many responsibilities.
- Make the names of classes and the names of methods and data reflect their responsibility.



## Java Applets

- Java **programs** can run from the command line by having the Java interpreter interpret byte codes contained in a class file.
- Java **applets** usually run from within a Web page via a Java-enabled browser.
- You may also run Java applets using the `appletviewer`.

## Java Applets
### *Viewing Applets*

To create and view an applet:

- Create and compile the applet source code: `MyApplet.java`

  ```
  $javac MyApplet.java
  ```

- Create the HTML file for the applet: `myapplet.html`

- There are three ways to view the applet:
  - From the browser by loading the URL: `http://www.cs.rpi.edu/~wiseb/java/myapplet.html`
  - Open the file locally in the browser via `File | Open` menu.
  - Use the `appletviewer` to view the HTML file:

    ```
    $appletviewer myapplet
    ```



## Java Applets
### *HTML*

- Java applets can be loaded from a Web page.
- HTML extensions are used to tell the browser what applet to load.
- Example:

  ```
  <APPLET CODE="HelloWorldApp.class" WIDTH=150 HEIGHT=25></APPLET>
  ```

Required HTML tags:

- `APPLET`
  which must have the following attributes:
  - `CODE`
    The name of the applet's compiled class file.
  - `WIDTH, HEIGHT`
    The width and height of the applet in pixels. You can resize any window the `appletviewer` creates. But *you cannot* resize the applet within a browser window; so make a good guess about how much space your applet requires to show up well for most users.

## Java Applets
### *HTML*

Optional attributes of the APPLET tag:

- CODEBASE
  Specifies the directory below the one where the page is located where the Java class files are located
- ALIGN
  Specifies the alignment of the applet within the Web page; may be one of LEFT, RIGHT, BOTTOM, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, ABSBOTTOM
- VSPACE, HSPACE
  Specifies the number of pixels above and below the applet.

To pass parameters to applets we use the PARAM tag which has two attributes:

- NAME: A string specifying the parameter name.
- VALUE: A string specifying the parameter value.

```
<APPLET CODE="SampleForm.class" WIDTH=150 HEIGHT=25>
<PARAM NAME="Name" VALUE="Joe">
<PARAM NAME="Address" VALUE="1234 Main">
<PARAM NAME="Amount" VALUE="50.00">
</APPLET>
```



## Java Applets
### *The Lifecycle of an Applet*

There are four methods in the Applet class that give you the framework on which to build any serious applet:

- init()
  Called once when the applet is launched for the first time. This is a good place to process PARAM values and create GUI components.
- start()
  Called immediately after init() and whenver the user returns to the page in which the applet exists. Things like resuming threads, or starting animations go here.
- stop()
  Called whenever the user moves off the page on which the applet sits. Time-consuming activities, such as threads or animations, should be stopped here.
- destroy()
  Java calls this method when the browser shuts down normally. Again, since Java takes care of garbage collection, you do not need to destroy applet panels or other GUI components. You do need to reclaim any non-memory resources (sockets, window handles) that you may have consumed.

# Java Applets
*Restrictions*

Because applets are designed to be loaded from a remote site and then executed locally, there are a number of security restrictions for applets.

In general, applets are not allowed to do things like:

- Load or run any local executables or libraries;
- Communicate with any host other than the server from which they were dowloaded from;
- Read/write/delete files on the local system
- Determine environment information like the user's name or path.
- Terminate the Java interpreter; e.g., call `exit()`

Different Web browsers may relax some of these restrictions. Netscape provides less security when loading an applet from a local file than it does for loading an applet via a URL, for example.

In addition, the `appletviewer` does not impose most of the above restrictions.



# Java Applets
*Passing Parameters to Applets*

Parameters may be passed to an applet from the HTML file in which the applet has been embedded.

Here is an example:

```
<HTML>
<HEAD>
<TITLE> An Applet with Parameters </TITLE>
</HEAD>
<BODY>
<APPLET CODE="SampleForm.class" WIDTH=150 HEIGHT=25>
<PARAM NAME="Name" VALUE="Joe">
<PARAM NAME="Address" VALUE="1234 Main">
<PARAM NAME="Amount" VALUE="50.00">
</APPLET>
</BODY>
</HTML>
```

These values are usually retrieved in the applet's `init()` method by calling the applet's `getParameter()` method:

```
String name = getParameter("name");
```

Note that the parameters in the `getParameter` method and the `PARAM` tag must match exactly.

# Java Applets
*Dialog Boxes*

- Normally an applet runs within an embedded window of the browser within the dimensions of the applet's `HEIGHT` and `WIDTH` tags specified in the HTML file
- This can be limiting, so you can create pop-up dialogs, which run in their own windows outsidde of the browser window.
- However, browsers such as netscape, usually display a warning and/or decorate the pop-up window so that users do not confuse it with an actual application.



# Java Applets
*Dialog Boxes* **Example**

```
class CalculatorApplet extends Applet
{
   public void init()
   {
      add(new Button ("Calculator");
   }

   public boolean action (Event evt, Object arg)
   {
      if (arg.equals ("Calculator"))
      {
         if (calc.isShowing()) calc.hide();
         else                  calc.show();
      }
      else
         super.action (evt, arg);
      return true;
   }
   Frame calc = new Calculator();
}
```

# Java AWT

The **Abstract Windowing Toolkit** is provided by the `java.awt` package.

The AWT is divided into three categories:

- **Graphics:**
  Classes for colors, fonts, images, polygons, etc.
- **Components:**
  Classes for GUI components: buttons, menus, lists, dialog boxes, etc.
- **Layout Managers:**
  Classes to control the layout of the GUI components.



# Java AWT
## *Graphics*

The AWT provides a number of classes for graphics:

- Color
- Dimension
- Event
- Font
- FontMetrics
- Graphics
- Image
- Insets
- MediaTracker
- Point
- Polygon
- Rectangle
- Toolkit

## Java AWT
### *Interface Components*

The AWT provides a number of GUI components:

- `Button`
- `Canvas`
- `Checkbox`
- `Choice`
- `Container`
  - `Panel`
    - `Applet`
  - `Window`
    - `Dialog`
      - `FileDialog`
    - `Frame`
- `Label`
- `List`
- `Scrollbar`
- `TextComponent`
  - `TextArea`
  - `TextField`



## Java AWT
### *Layout Managers*

- `BorderLayout`
- `CardLayout`
- `FlowLayout`
- `GridBagLayout`
- `GridLayout`

## Java AWT
*Related Classes*

The AWT provides a class for grouping checkboxes:

- `CheckboxGroup`

The AWT provides classes for menus:

- `MenuComponent`
  - `MenuBar`
  - `MenuItem`
    - `Menu`
    - `CheckboxMenu`



## Java AWT
*Some Notes...*

- Note that an `Applet` is a sub-class of `Panel`; it can therefore contain other GUI components
- To sub-divide an the user interface of an applet into a more structured manner, break the applet panel into a number of sub-panels.
- The default layout manager for an `Applet` is `FlowLayout`. To change the layout call the applet's `setLayout()` method.

## Java AWT
### *Events*

- Like most GUI programs, Java uses event programming.
- For events like window events (`WINDOW_DESTROY`), scroll bar movements, override:

```
boolean handleEvent (Event evt);
```

- For events like button clicks, or checbox toggles, choice selections, listbox selections, override

```
boolean action (Event evt, Object arg);
```

- Both of these methods are to return `true` if they handle the event.
- If the event is not handled, it should be passed to the superclass:

```
      return super.action (evt, arg);
or
      return super.handleEvent (evt, arg);
```



## Java AWT
### *Events* Example

```
import java.awt.*;
import java.applet.*;

public class GUI1 extends Applet
{
  public void init()
  {
    b1 = new Button ("Press me");
    field1 = new TextField ("Welcome!", 20);

    setLayout (new GridLayout (2, 1));
    add (b1);
    add (field1);
  }

  public boolean action (Event event, Object arg)
  {
    if (event.target == b1) {
      field1.setText ("Thank you!!!" );
      return true;
    }
    else {
      return super.action (event,arg);
    }
  }

  private Button     b1;
  private TextField  field1;

}
```