# UNIX Systems Programming

## Java Programming Language Fundamentals

Dr. Kivanç Dinçer
CENG-332 Lectures
Spring 2000

# Java Applications vs Applets

- Java Applications
  - Any platform with a Java Virtual Machine (JVM) interpreter can run a Java program just as one can run a Fortran, C or Cobol program
- Java Applets
  - Designed specifically to be loaded and run by a Web Browser

# Java Application

- Requires a main() method
- Cannot have a return statement, but instead may include System.exit()
  - abruptly terminates the running program including all threads
  - action taken with value returned is system dependent

# Hello Program 1

```
public class Hello {
  public static void main (String[] args) {
     System.out.println(”Hello World”);
     System.exit(0); // not required
  }
}
```

*Interpretation left up to the Operating System*

# Hello Program 2

```
public class Hello {
  public static void main (String[] args) {
    for (int i=0; i<args.length; i++)
       System.out.println("args[" + i + "] = "
                                   + args[i]);
  }
}
```

# Java Applets

- Designed specifically to be loaded and run by a Web Browser
  - Have more security constraints than applications

# HelloWorld - Java Applet style

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

public class HelloWorld extends Applet {

  public void paint (Graphics g) {
    g.drawString(“HelloWorld Applet”,25, 25);
  }

}
```

# HelloWorld - in Color

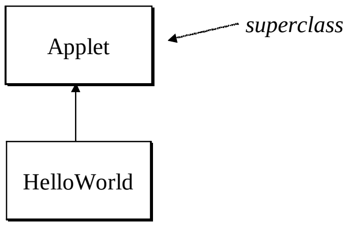
```
import java.applet.*;
import java.awt.*;

public class HelloWorld extends Applet {

  public void paint (Graphics g) {
    g.setColor (Color.red);
    g.drawString(“Hello Applet World”,25,25);
  }

}
```

# extends ⇨ Inheritance



Applet

*superclass*

HelloWorld

# Displaying an Applet on a Web Page

- Requires an HTML file with the statement:

```
<APPLET code="FirstApplet.class"
        width=150
        height=100>
</APPLET>
```

# A Web Page with an Applet

```
<HTML>
  <HEAD>
    <TITLE> My Web Page </TITLE>
  </HEAD>

  <BODY>
    <APPLET CODE="HelloWorld.class"
          WIDTH=150   HEIGHT=25>
    </APPLET>
  </BODY>
</HTML>
```

# Java Packages

- The Java API consists of over twenty packages each with classes and interfaces
  - java.applet
  - java.awt
  - java.beans
  - java.io
  - java.lang
  - java.net
  - . . .

# java.net Package

- java.net.Socket
- java.net.ServerSocket
- java.net.URL
- . . .

# import

- To use the classes of any package (except Java.lang) you must import the packages:
  - `import java.net.Socket;`

  OR
  - `import java.net.*;`

# Classpath

- Java knows where to look to find system classes
- The Classpath variable is used to tell java where to look for user classes

```
set CLASSPATH=.;C:\joe\apps;D:\myjava
```

↑

*current directory*

# Basic Java

# Comments

- Standard C style

```
/* ... until */
```

- End of line

```
// ... until end of line
```

- java doc style comments

```
/** ... until */
```

# Constants: final variables

- No C style constants in java
  - A final variable cannot be changed
  - A final class cannot be subclassed

```
public final class Math {
    public static final double PI = 3.14159…;
    public static final double E = …;
}
```

# Two kinds of data types

- Primitive
  - int, float, char, …
- Non-Primitive
  - objects
  - arrays

# Java Primitive Data Types

| | |
|---|---|
| • boolean | true or false |
| • char | 16 bit Unicode character |
| • byte | 8-bit integer (signed) |
| • short | 16-bit integer (signed) |
| • int | 32-bit integer (signed) |
| • long | 64-bit integer (signed) |
| • float<br>(IEEE 754-1885) | 32-bit floating point |
| • double<br>(IEEE 754-1885) | 64-bit floating point |

# Variables

```
int     i = 23;
byte    b = 88;
short   s = 733;
boolean b = true;
char    c = ‘z’;

// not ok -- compiler catches!
byte b1 = 7373;
```

# Floating Point Variables

```
double d = 44.494;
float  x = 44.33;   // can’t do!

float x = 44.33f;  // float constant
```

# Java is Strongly Typed

```
int x;
short y;
x = 737;
y = 777;
x = y;   // ok - automatic coercion done!
y = x;   // not ok ! might lose precision
y = (short)x;  // requires cast
```

# Other casts

```
char  c = 'a';
short s = (short)c;
byte  b = (byte)c;
```

stores the value 97 (ascii value of ‘a’)

# Strings

- Not primitive type but treated special
- String constants:

  "hello"
  "java land"
  **System.out.println("hello" + " world");**
  where + is string concatentation

- String is a class

  **String s = "hello world"**

# Reference Types

- Arrays and Objects are of reference types
- Handled by reference
  - the address is stored and passed to methods by reference
  - primitive types are stored(?) by value

# Arrays

# Arrays

- Arrays are Java objects
- You must
  - Declare
  - Allocate
    - with keyword **new**
- Cannot be <u>allocated in place</u> as in C/C++

# Array Declaration, Allocation and Assignment

**Declare:**

- **int [] scores; /* array not created*/**

**Allocate:**

- **scores = new int[10];**

**Assign:**

- **scores[0] = 33;**
- **scores[9] = 56;**

<u>Alternative styles:</u>

**int [ ] scores;** OR **int scores [];**

# Array Idioms

**Declare & Allocate:**

- **int [] scores = new int[20];**

**Declare & Allocate & Init:**

```
int [ ] scores = {1, 2, 3+5, 7};
```

# Arrays

- All elements of an int, float, double, or long array are initialized to zero
- Arrays begin at index 0
- Arrays are always checked for bounds correctness:
  - ArrayIndexOutOfBounds exception will be thrown

    **scores[j] = 34;** // exists?

Traversing Array Elements:

```
for (int i=0; i< scores.length; i++) {
   System.out.print(scores[i]);
}
```

# Classes and Objects

# Data Records (the C struct)

```
struct Rectangle {
  int x;
  int y;
  int width;
  int height;
}

Rectangle r;
r.x = 10;
r.y = 20;
r.width = 15;
```

*a function:*

```
int computeArea (Rectangle r)
{
  return(r.width * r.height);
}
```

# The C++ struct: move code near its data

```
struct Rectangle {
  int x;
  int y;
  int width;
  int height;

  int computeArea () {
    return (width * height);
  }
}
```

```
Rectangle myRect;
myRect.width  = 20;
myRect.height = 30;
area = myRect.computeArea();
```

*note that the members of a struct are visible. They are* **public** by default

*members:*
*x, y, width, height, computeArea*

# Class

- A collection of data and methods (functions in C/C++) that operate on that data
- The data and methods define an object
- Each object instance has its own copy of the data

# Circle class

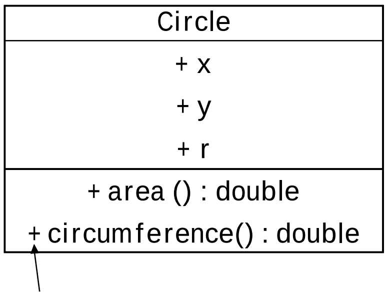
```
public class Circle {
  public double x,y; // center
  public double r;   // radius

 // methods that use the data
  public double area ()
  {
    return 3.14159*r*r;
  }
  public double circumference ()
  {
    return 2*3.14159*r;
  }
}
```

Define an instance of Circle (a Circle object):

```
Circle c;
c = new Circle();
```

# Unified Modeling Language (UML) Representation of Circle Class



| Circle |
|---|
| + x<br>+ y<br>+ r |
| + area () : double<br>+ circumference() : double |

*represents public visibility*

# Accessing Object Data

```
Circle c = new Circle();
c.x = 4.0;
c.y = 3.0;
c.r = 10.2;
…
System.out.println("radius=" + c.r);
```

# Calling Object Methods

```
Circle c = new Circle();
double aa;
c.r = 12.2;
aa = c.area();
```

*not: area(c);*

# Object Creation

- **`Circle c = new Circle();`**
  - A special function/method : constructor
    - Looks like a function.
    - Has same name as the class
    - Purpose: initialize an object
  - Java provides a default constructor that does no initialization

## Defining a Constructor

```
public class Circle {
  public double x,y, r ;

  // constructor
  public Circle (double _x,
                 double _y, double _r )
  {
     x = _x;
     y = _y;
     r = _r;
  }
  public double circumference ( ) {…}
  public double area ( ) {…}
}
```

## Using the “arg” constructor

```
Circle c;
c = new Circle(10.0, 20.0, 5.2);
```

OR

```
Circle c = new Circle(10.0,20.0,5.2);
```

## Constructor Gotcha

*NO return value specified -- not even void*

```
public Circle(double _x,
              double _y,double _r)
{
  x = _x;
  y = _y;
  r = _r;
}
```

## This is NOT a Constructor

```
public void Circle(double x1,
                   double y1,
                   double r1)
{
  x = x1;
  y = y1;
  r  = r1;
}
```

The compiler will compile this as a method and you will think you have a constructor

## Java Keyword: this

```
public class Circle {
  public double x,y, r ;

  // constructor
  public Circle (double x,
                 double y, double r )
  {
     this.x = x;
     this.y = y;
     this.r = r;
  }
  public double circumference ( ) {…}
  public double area ( ) {…}
}
```

## Multiple Constructors

```
public class Circle {
  public double x,y, r ;

  // constructors
  public Circle (double _x,
                 double _y, double _r )
  { x = _x; y = _y; r = _r; }
  public Circle (double r)
  { x = 1.0; y = 1.0; this.r = r; }
  public Circle (Circle c)
  { x = c.r; y = c.y; r = 10.0 }
  public double circumference ( ) {…}
  public double area ( ) {…}
}
```

## Method Overloading

- Methods with the same name but different
  - parameter lists
  - number of parameters
  - type of parameters

```
void foo (int, int);
void foo (int, double)
void foo (Circle);
```

- BUT can’t do
  `double foo (int, int)`

## Constructor Gotcha!!

```
Circle r = new Circle (10.0, 20.0, 5.0);
Circle s = new Circle (40.2, 50.3, 6.0);
Circle t = new Circle ();
```

*You cannot use the default constructor if you define your own constructor!*

*If you want a no-arg constructor, then YOU must define one!*

# Design Pattern Multiple Constructors

```
public Circle (double x,double y,double r) {
  this.x = x;  this.y = y; this.r  = r; }

public Circle (double r){
  this(1.0, 1.0, r);
}

public Circle (double x, double y) {
  this(x, y, 10.0);
}
public Circle () {
  this(1.0, 1.0, 10.0);
}
```

this = constructor call. note: if used, must be the first statement in a constructor

NO ARG Constructor

# Class Variables

```
public class Circle {
  public double x,y,r ;

public Circle (double x,double y,double r )
{
  this.x = x;  this.y = y; this.r  = r;
}

public double circumference ( ) {…}
public double area ( ) {…}
}
```

x,y,r are instance variables -- each instance of Circle has its own version of x,y and r

# Class Variables

```
public class Circle {
 static int numCircles = 0;
 public double x,y, r ;

public Circle (double x,double y,double r){
  this.x = x;  this.y = y; this.r  = r;
  numCircles++;
}
. . .
}
```

Only one copy of numCircles associated with the class Circle

Tracks how many circles have been created

# Accessing static variables

- Must use the class name

```
System.out.println(Circle.numCircles);
System.out.println(Math.PI);
```

Must use the class name outside the class

# Class Methods

- Not associated with object instances
- Closest thing to “global” methods
  - **`Math.sqrt(double)`**
  - **`Math.sin(double)`**
- Also called static methods

- Have access only to static variables

# Hello Program

```
public class Hello {
  static String s = “hello”;

  public static void main (String[] args)
  {
    System.out.println(s);
    for (int i=0; i<args.length; i++)
         System.out.println("args["+i+"]="
                                  + args[i]);
  }
}
```

# Initialization

- Variables

  ```
  static int numCircles = 0;
  float r = 22.33;
  ```

- Methods
  - Instances = constructors
  - Class = static initializers

# Static Initializer

- Called when the class is loaded
- For initializing static variables
  - No return value
  - No arguments
  - No name

**`e.g., static { ….}`**

```
public class Circle {
static private double sines[] = double [1000];

static {
 double x, deltaX;
 deltaX = (Math.PI/2)/(1000-1);
 for (int i=0, x=0.0; i<1000; i++, x+=deltaX)
 {
     sines[i] = Math.sin(x);
 }
} // end static initializer
```

# Garbage Collection

- Java periodically frees memory no longer needed.
- Garbage collector runs as low-priority thread - *synchronously* or *asynchronously* depending on the system

# Forced Forgetting

```
public static void main (String [ ] args)
{
  int big [] = new int [10000];
  double result = compute(big);

  for (;;) {
    // do something with result
}
```

# Forced Forgetting...

```
public static void main (String [ ] args) {
  int big [ ] = new int [10000];
  double result = compute(big);

  big = null;  // Garbage collector able to collect array

  for (;;) {
    //do something with result
 }
```

# Object Finalization

- Garbage collection only frees the memory allocated for an object
- Objects may be holding onto resources
  - file descriptors
  - sockets
- Finalizer methods are used to free resources prior to object garbage collection

# Finalizer Method

- Must be:
  - non-static
  - return no value -- void
  - named finalize

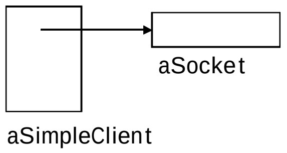
e.g. FileOutputStream class has:

```
protected void finalize() throws IOException
{
    if (fd != null) close();
}
```

fd is file descriptor object

# Finalize

- Finalize is called when the garbage collector determines that an object is to be garbage collected



# Ex. Farley p 13
# Using finalize to close socket connection

```
public synchronized void finalize () {

  System.out.println(“Closing down..”);

  try { serverConn.close(); }  // socket
  catch (IOException e) {
    System.out.println(“SimpleClient:”+e);
    System.exit(1);
  }
}
```

## Finalize not always final

- The finalize method may store its object's reference (i.e. this) somewhere, preventing garbage collection

```
public synchronized void finalize () {
  AppVector.addElement(this);
}
```

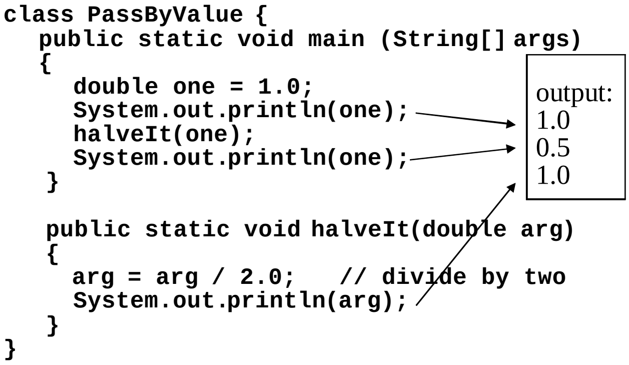
## Assignment: Reference Types

```
import java.awt.*;
…..
Button p, q;
p = new Button();
q = p
p.setLabel("STOP");
String s = q.getLabel();
```

## Passing Primitive Parameters

- Primitive Data types are "passed by value"
- The value of the passed parameter is copied as the value of the parameter

```
class PassByValue {
   public static void main (String[] args)
   {
      double one = 1.0;
      System.out.println(one);
      halveIt(one);
      System.out.println(one);
   }
}
```



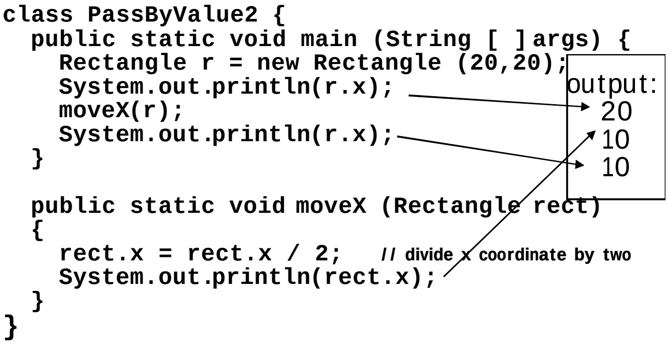
```
class PassByValue {
   public static void main (String[] args)
   {
      double one = 1.0;
      System.out.println(one);
      halveIt(one);
      System.out.println(one);
    }

    public static void halveIt(double arg)
    {
      arg = arg / 2.0;   // divide by two
      System.out.println(arg);
    }
}
```

```
output:
1.0
0.5
1.0
```

## Passing Reference Parameters

- With Reference Data types, the address is passed
- The parameter has access to the value



```
class PassByValue2 {
  public static void main (String [ ] args) {
    Rectangle r = new Rectangle (20,20);
    System.out.println(r.x);
    moveX(r);
    System.out.println(r.x);
  }

  public static void moveX (Rectangle rect)
  {
    rect.x = rect.x / 2;   // divide x coordinate by two
    System.out.println(rect.x);
  }
}
```

```
output:
20
10
10
```

*the object reference is passed by value.*

*the result is two object references (r and rect) pointing to the same Rectangle object in memory*