Topic 1
Java Overview

Applet vs Application

Java Applications vs Applets

- Java Applications
  - Any platform with a Java VM 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
- May include System.exit()
  - action taken with value returned is system dependent
  - abruptly terminates the running program including all threads

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] );
  }
}
```

Hello Program 3

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

Applets

Java Applets

- Java Applets
  - Designed specifically to be loaded and run BY a Web Browser
  - More security constraints than applications

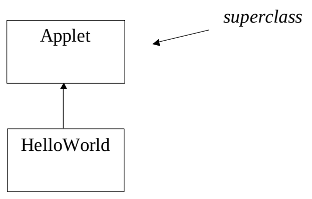
HelloWorld - Java Applet style

```
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



Display an Applet

- Requires an HTML file with the statement

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

Web Page with Applet

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

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

import
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
- Option:
  - 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
Java 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 (IEEE 754-1885)	32-bit floating point
• double (IEEE 754-1885)	64-bit floating point

Variables

```
 int   i = 23;
 byte  b = 88;
 short s = 733;

// 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
```

Other Type Variables...

- boolean b = true;
- char c = 'z';

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

Java is Strongly Typed

double x;
float y;
x = 737;
y = 777;
x = y; // ok
y = x; // not ok!
y = (float)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 but treated special
- String constant:
  “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 reference types
- Handled by reference -- the address is stored and passed to methods
  - primitive types are stored by value

Arrays

Arrays

- Arrays are Java objects
- You must
  - Declare
  - Allocate
    - with keyword **new**
- Cannot be allocated in place 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;**

Alternative style

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, 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?

Looping and Arrays

```
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;
```

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

a function

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();
```

<u>members</u>:
x, y, width, height, computeArea

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

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

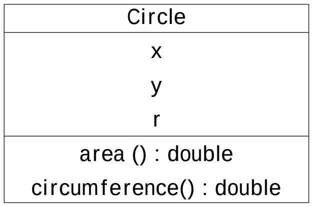
the class Circle

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

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

Define an instance of Circle
(a Circle object):

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

Unified Modeling Language UML


Circle

x

y

r

area () : double

circumference() : double

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();

•Looks like a function.

•A special function/method : constructor

•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 x1, double y1, double r1 ) {
  x = x1;
  y = y1;
  r  = r1; }
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 x1, double y1, double r1 ) {
  x = x1;
  y = y1;
  r  = r1;
}
```

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 ;
// constructor
public Circle (double x, double y, double r ) {
  this.x = x;  this.y = y; this.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 parameter
- 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 ;
// constructor
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 ;
// constructor
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
- 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

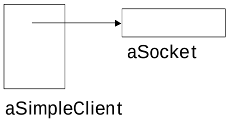
e.g. FileOutputStream class has:

```
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);
  }
}
```

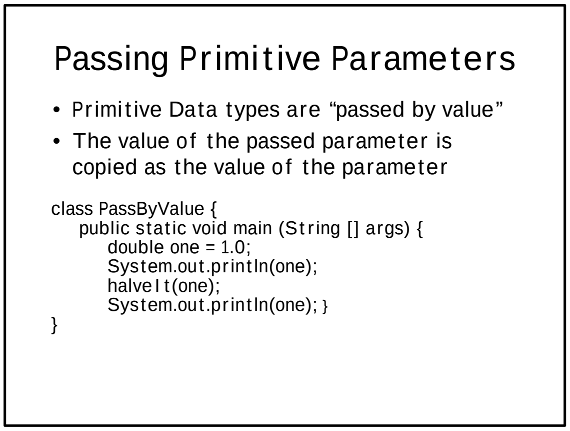
Finalize not always final

- 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

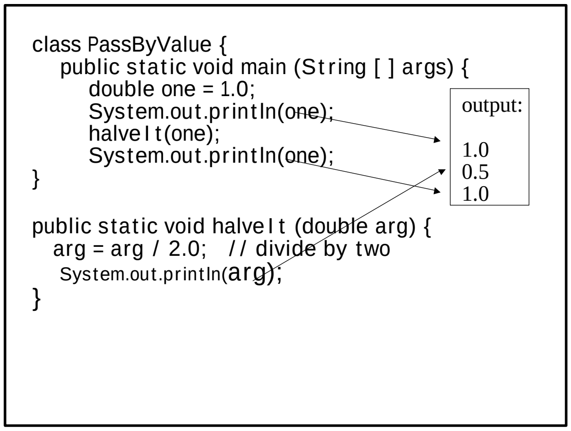
```
import java.awt.*;
… ..
Button p, q;
p = new Button();
q = p
p.setLabel(“STOP”);
String s = q.getLabel();
```



Passing Primitive Parameters

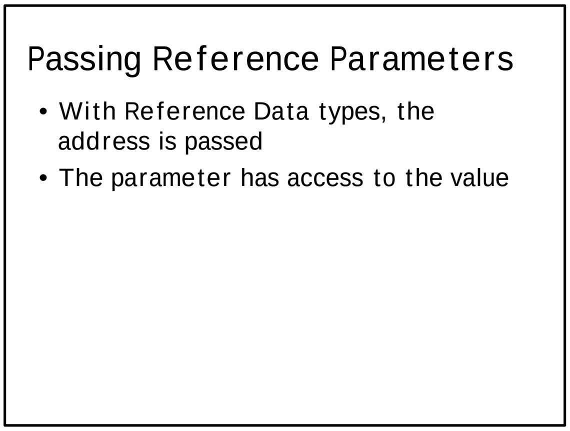
- Primitive Data types are “passed by value”
- The value of the passed parameter is copied as the value of the parameter

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



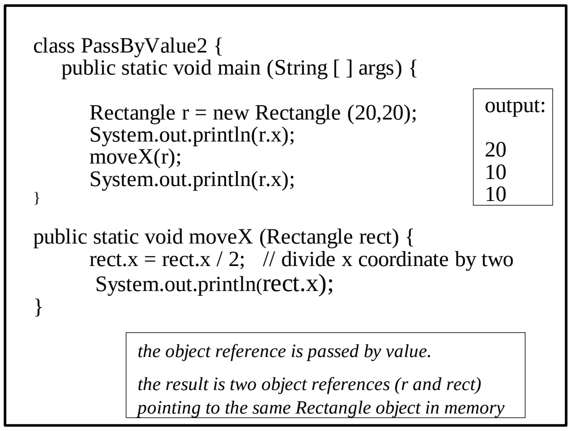
```
class PassByValue {
    public static void main (String [ ] args) {
        double one = 1.0;
        System.out.println(one);
        halveIt(one);
        System.out.println(one);
}

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

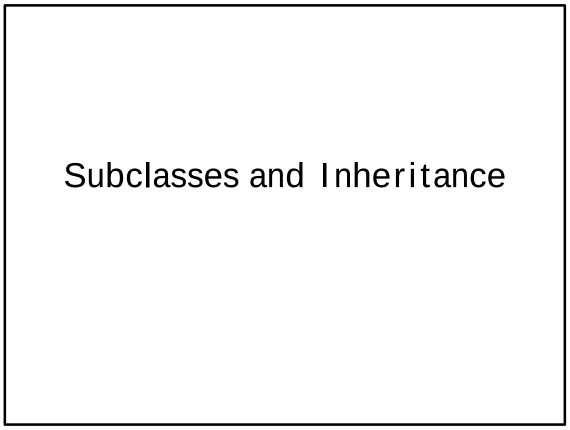


```
class PassByValue2 {
    public static void main (String [ ] args) {

        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);
}
```

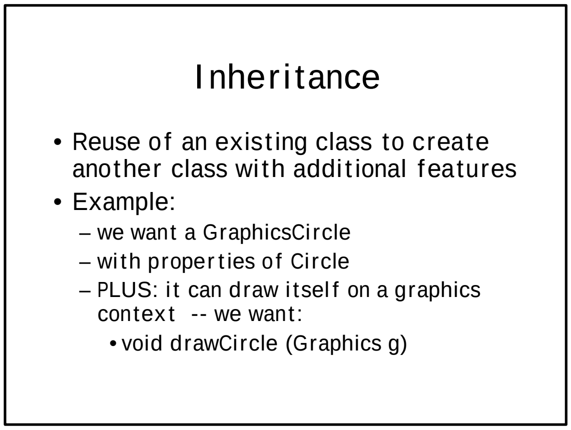
```
output:

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



Subclasses and Inheritance



Inheritance

- Reuse of an existing class to create another class with additional features
- Example:
  - we want a GraphicsCircle
  - with properties of Circle
  - PLUS: it can draw itself on a graphics context -- we want:
    - void drawCircle (Graphics g)

Cut and paste Technique (not recommended)

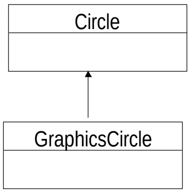
```
public class GraphicsCircle {
  public double x,y, r ;

public double circumference ( ) {… }
public double area ( ) {… }
public void drawCircle (Graphics g) {… }
}
```

Inheritance

```
public class GraphicsCircle extends Circle {

 public void drawCircle (Graphics g) {… }

}
```

UML

- Circle is the superclass
- GraphicsCircle is the subclass



Creating Instances

- Circle c = new Circle();
- GraphicsCircle gc = new GraphicsCircle();
- double a1 = c.area();
- double a2 = gc.area();
- gc.drawCircle(g);

Super to Subclass Assignment (Downcasting)

- GraphicsCircle IS-A Circle

Circle c = new Circle();

GraphicsCircle gc = new GraphicsCircle();

Circle c2 = gc;

BUT you can only access public data and methods of Circle

DOWNCASTING

Subclass to Superclass Upcast Assignment (requires cast)

- Circle IS-NOT--A GraphicsCircle

Circle c = new Circle();

GraphicsCircle g2 = c; // can't do!

Requires Cast

Circle c does not have a drawCircle() method

GraphicsCircle g2 = (GraphicsCircle)c;

Subclasses and Parameters

- void foo (Circle cp) {… }

```
Circle c = new Circle();
GraphicsCircle gc = new GraphicsCircle();
foo(c);
foo(gc); // ok because gc IS-A Circle
```

final classes

- Prevents a class from being extended
- No subclasses allowed

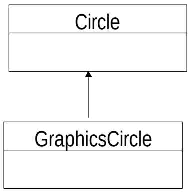
```
public final Hexagon {… ..
```

the class Object

- the root of all classes
- every class is a subclass of Object
- the methods of class Object are available to all classes in Java
  - String toString()
  - boolean equals (Object obj)
  - int hashCode()
  - void wait()
  - void notify()

Subclasses & Constructors

- A constructor will be called for each class in an inheritance hierarchy
- Either implicit or explicit



super

```
public class GraphicsCircle extends Circle {
 Color myColor;
 public GraphicsCircle (double x, double y,
                                   double r, Color c) {
  super(x,y,r);
  this.myColor = c;
 }
 public void drawCircle (Graphics g) {… }
```

if used, must be the first statement in subclass constructor

```
public GraphicsCircle (double x, double y,
                                   double r, Color c) {
 super(x,y,r);

 this.myColor = c;
}
```

POTENTIAL GOTCHA!

if superclass constructor is NOT called, Java will call the no-arg constructor super( )

Data Hiding

- Data Hiding
  - keep the data of an object invisible to users of the class
  - allows changes to code without impacting users of the code
- Object-Oriented Principle
  - keep the data elements PRIVATE not PUBLIC

Accessor Methods: get

```
public class GraphicsCircle {
  private double x,y, r ;
 public double getX() { return x;}
 public double getY() {return y;}
 public double getR() {return r;}

public double circumference ( ) {... }
...
}
```

Accessor Methods: set

```
public class GraphicsCircle {
  private double x,y, r ;
 public double getX() { return x;}
 public double getY() {return y;}
 public double getR() {return r;}

public void setX(double x) { this.x. = x;}
public void setY(double y) { this.y = y;}
...
}
```

JavaBeans - Naming Rules for get & set

- Use lowercase
  - get
  - set
- Convert first letter of variable (lowercase) to UPPERCASE
- getX
- setX
- getTotalWidgetsSold
  - int totalWidgetsSold

Visibility Modifiers

public
protected
private
package

public vs private

- public double x,y,r;

```
Circle c = new Circle();
c.x = 20.0;  // OK
```

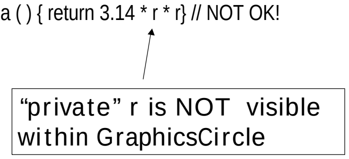
- private double x,y,r;

```
c.x = 20.0 ; //NOT ALLOWED!
```

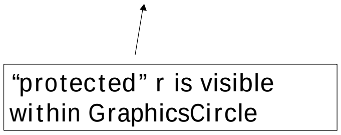
protected

- visible to methods within the class
- visible to methods in any subclass
- NOT visible to external users of the class

```
public class Circle {
  private double x,y, r ;
  public double circumference ( ) {… }
}

public class GraphicsCircle  extends Circle{
  private Color c;
  public double area ( ) { return 3.14 * r * r} // NOT OK!
}
```


“private” r is NOT visible within GraphicsCircle

```
public class Circle {
  protected double x,y, r ;
  public double circumference ( ) {… }
}

public class GraphicsCircle  extends Circle{
  private Color c;
  public double area ( ) { return 3.14 * r * r}   //OK!
}
```


“protected” r is visible within GraphicsCircle

package (default when nothing specified)

```
public class Circle {
  double x,y, r ;
  public double circumference ( ) {… }
}

public class GraphicsCircle  extends Circle{
  Color c;
  public double area ( ) { return 3.14 * r * r}   //OK!
}
```

package

- Visible to all methods in all classes that are in the same package
- Not visible outside the package

- If a package is not specified, “default unnamed” package is assumed

Visibility

Visible to:	public	protected
Same class	Yes	Yes
Class in same package	Yes	Yes
Subclass in diff package	Yes	Yes
Non-subclass, other package	Yes	No

Visibility

Visible to:	private	package
Same class	Yes	Yes
Class in same package	No	Yes
Subclass in diff package	No	No
Non-subclass, other package	No	No

Abstract Classes

- Used to structure an inheritance hierarchy
- If we want a family of shape classes
  - Circle
  - Rectangle
  - Ellipse
  - Triangle
- We want force each subclass to implement area() AND perimeter()

Abstract class: Shape

```
public abstract class Shape {
  public abstract double area ( );
  public abstract double perimeter();
}

public class Circle extends Shape {…
public class Triangle extends Shape {..
```

```
public class Triangle extends Shape {
  private double s1, s2, s3;

  public double perimeter () {
   return s1 + s2 + s3;
  }
  public double area ( ) {
    return …..;
  }
}
```

Triangle must implement the abstract methods of Shape

Abstract classes

- Any class with an abstract method is an abstract class and must declare itself abstract
- A class may be declared abstract even with no abstract methods
- An abstract class cannot be instantiated
- If a subclass does not implement all abstract methods, the subclass is abstract

Declaring abstract variables

```
Shape [] shapes = new Shape[10];
shape[0] = new Rectangle();
shape[1] = new Circle (10.0, 10.0, 5.0);
shape[2] = new Triangle(2.0,2.0,3.0);
```

No cast required --
a Circle IS-A Shape

Polymorphism

```
Shape [] shapes = new Shape[10];
shape[0] = new Rectangle();
shape[1] = new Circle (10.0, 10.0, 5.0);
shape[2] = new Triangle(2.0,2.0,3.0);
double a1 = shape[0].area();
double a2 = shape[1].area();
double a3 = shape[2].area();
```

Subclasses in Action

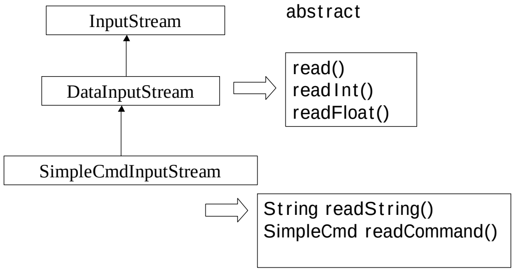
Farley Example 1-1

```
public abstract class SimpleCmd {
  protected String arg;

  public SimpleCmd(String inArg) {
    arg = inArg;
  }

  public abstract String Do();

}
```

```
class GetCmd extends SimpleCmd {

public GetCmd(String s) {
    super(s);
  }

public String Do () {
    String result = arg + "Gotten\n";
    return result;
  }
}
```

```
class HeadCmd extends SimpleCmd {
  public HeadCmd(String s) {
    super(s);
  }

public String Do () {
    String result = "Head \"" + arg + "processed\n";
    return result;
  }

}
```

Farley Ex 1-2



Java Interface

- An alternative to abstract classes
- An interface specifies only method signatures:
  - method name
  - return value
  - parameters and types
- Abstract class can define:
  - data variables
  - concrete methods

interface

```
public interface Drawable {
  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(Graphics g);
}
```

OPTIONAL!

interface

```
public abstract interface Drawable {
  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(Graphics g);
}
```

interface variables

(rarely seen)

```
public interface Drawable {
  private static final prefColor = Color.red;

  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(Graphics g);
}
```

Only static final variables are allowed in an interface

classes implement interfaces

```
public class Triangle implements  Drawable {
  public void setColor(Color c) {
    // code;
  }
  public void setPosition(double x, double y) {
    ….;}
  public void draw(Graphics g) {
    ….;}
}
```

extend only one class implement multiple classes

```
public class Triangle extends Shape
   implements  Drawable, Serializable {
  ….
}
```

Must implement all methods in interfaces

Interface as Data Type

```
Drawable myShape;
myShape = new Triangle();

Drawable [] shapes = new Drawable[5];
shapes[0] = new Triangle();
shapes[1] = new Circle();
```

Assumes Circle and Triangle both implement Drawable

```
Drawable [] shapes = new Drawable[5];
shapes[0] = new Triangle();
shapes[1] = new Circle();

shapes[1].setColor(Color.blue);
a1 = shapes[1].area();   // NOT OK!!
```

Can only execute methods defined as part of the interface

String and StringBuffer

Strings

- NOT an array of charaacters
- Based on the class String
- String are immutable
- BUT special treatment
  - "hello world"
    - creates a String instance object
  - concatenation operator +

String Constructors

- String (String value)
- String (char [] value)
- String (char [] val, int offset, int length)
- String (byte [] bytes)
- String (StringBuffer stringbuffer)

String Methods

- int length()
- char charAt(int index)
- boolean equals(String s)
- boolean equalsIgnoreCase(String s)
- int indexOf(char c)
- String subString(int beginIdx, int endIdx)

String static methods

- String valueOf(Object obj)
- String valueOf(char [] data)
- String valueOf(char c)
- String valueOf(int i)
- String valueOf(long l)
- String valueOf(float f)
- String valueOf(double d)

StringBuffer

- Contents can be modified
- Grows in length as needed
- Can modify in place with:
  - setCharAt()
  - append()
  - insert()
- Convert to string with
  - toString()

StringBuffer used in Farley p 12 Example 1-2

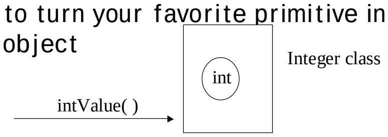
Vectors

the class Vector

- An "array" of objects that grows when necessary
- VERY useful when you don't know in advance how many objects you need to track
- To use: **`import java.util.*;`**
- Does <u>not</u> store primitive types (int, float, etc.) - only objects!!
- Vector elements can be null

Vectors and Primitive Types

- You can't create a vector of ints, floats or any other Java primitive type
- Java has wrapper classes specifically to turn your favorite primitive into an object



Wrapper Classes (one for each primitive type)

- Integer
- Float
- Double
- Long
- Character
- ...

Wrapper Examples

- Integer myInt = new Integer(3);
- Float myF = new Float(33.44f);
- int j = myInt.intValue()
- float f = myF.floatValue();

What can a Vector do?

- add an element
  - **`void addElement( Object obj);`**
- return an object at some index position
  - **`Object elementAt( int index);`**
- tell you the index position of some object
  - **`int indexOf( Object elem);`**
- tell you how may elements in the vector = **`int size();`**

Other Vector capabilities

- **`void removeElementAt( int index);`**
- **`boolean contains (Object elem);`**
- **`boolean isEmpty();`**
- **`int capacity ()`**
  - how many elements can the Vector hold before expansion is necessary

Declaring an Object

- Like other variable declarations, object declarations can also appear alone, like this:

`Vector v;`

But more common is...

Creating a Vector instance

Vector v = new Vector( );

Creating a Vector instance

<u>Vector v</u> = <u>new Vector( );</u>

Declaration

Allocation &
Assignment

Creating a Vector instance

Vector v = new Vector( );

Constructor

a method executed when
the object is created

Creating a Vector instance

Vector v = new Vector(20);

Constructor

creates a Vector with 20 slot capacity

Creating a Vector instance

Vector v = new Vector(20,10);

Constructor

creates a Vector with 20
slots and when more
space is needed, allocates
new slots 10 at a time

More on Vector Access Methods

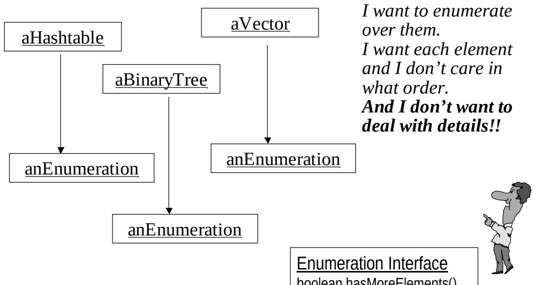
- boolean <u>contains</u> (Object obj)
  - determines if an object is in the vector
  - the two object references must refer to the SAME object
- Object <u>elementAt</u> (int idx)
  - retrieves the element at the specified index
  - if idx is not valid, throws ArrayIndexOutOfBounds exception

Vector Iteration

```
public static void printVec (Vector vec) {

if (vec.isEmpty() )
  System.out.println(“Vector is empty”);
else
 for (int i=0; i< vec.size(); i++)
  System.out.println(vec.ElementsAt(i));
}
```

reusable?

Enumeration



I want to enumerate over them.
I want each element and I don't care in what order.
And I don't want to deal with details!!

Enumeration Interface
boolean hasMoreElements()

Object nextElement()

Sets and Collections

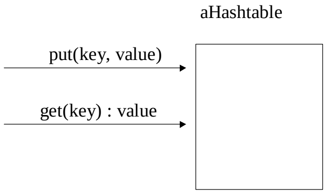
Vector Enumeration

- Vectors & Hashtables know how to create Enumeration objects
- Just ask them! -- use the **`elements()`** method
  – returns an Enumeration of all elements

```
public static void printVec (Vector vec) {
 Enumeration e = vec.elements();
 while (e.hasMoreElements() )
  System.out.println("\t" + e.nextElement() );
}
```

HashTable

Hashtables - when We don't want to search Vectors or Arrays



Hashtable in use:

```
Hashtable numbers = new Hashtable();
 numbers.put ("one", new Integer(1));
 numbers.pu t("two", new Integer(2));
 numbers.put ("three", new Integer(3));
```

To retrieve a number

..use the following code:

```
Integer n = (Integer)numbers.get("two");
if (n != null) {
   System.out.println("two = " + n);
}
```

the class: HashTable

- Implements a hashtable, which maps keys to values.
- Any non-null object can be used as a key or as a value.
- To successfully store and retrieve objects from a hashtable, the objects used as keys must implement:
  - the hashCode method
  - the equals method.

Exceptions and Exception Handling

What's an Exception

- A signal that indicates an *exceptional condition* (something unexpected) has happened in your program
- To *throw an exception* is to signal that an exceptional condition has occurred
- To *catch an exception* is to handle the exception - to take whatever action is necessary
  - *sometimes you can't do anything*

Why Exceptions?

- Exceptions allow the programmer to treat error conditions outside the main logic flow
- Most programming languages (without exceptions) handle errors by passing return codes as error indicators

Exception Example 1 the FileInputStream class

<u>constructor</u>

```
public FileInputStream (String s) throws IOException;

String s = "myfile.dat";
FileInputStream fis = new FileInputStream (s);
```

will not compile unless...

we deal with the possibility of an IOException

Exception Example 2
URL class

URL constructor

```
public URL (String s) throws MalformedURLException;

String  webPageString  = 'http://www.yahoo.com";
URL myURL = new URL(webPageString);
```

will not compile unless…

Exception Example 3
the Thread class

Thread static method

```
public static void sleep (long millis)
                    throws InterruptedException;

Thread.sleep(1000);
```

will not compile unless….

Unless we… .

- List the exception in our own method header

OR

- Catch the exception in our method

```
class ThreadSleepTest {
  public static void main (String [] args)
                   throws InterruptedException {

       for (int i=0; i< 10; i++) {
         if ( i == 5) Thread.sleep(1000);
                System.out.print(i + ".. ");
       }
  }
```

```
>java ThreadSleepTest
0.. 1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9..
```

one second pause

```
class ThreadSleepTest2 {
  public static void main (String [ ] args) {

        try {
          for (int i=0; i< 10; i++) {
              if ( i == 5) Thread.sleep(2000);
              System.out.print(i + ".. ");
          }
        }
        catch (InterruptedException e) {
        }

  }
```

Exceptions II

Handling Exceptions the complete story

```
try {
  // code that might
  // throw an exception
} catch (ExceptionType variable) {
   // handle the exception if thrown
} finally {
  // .. always do this
}
```

```
// assume t is a Thread object
```

```
try {
      t.sleep(1000);

} catch (InterruptedException e) {
      System.out.println("an exceptions was thrown");

} finally {
      System.out.println("finally");
}
```

Used for cleanup - close files, release resources...

Multiple Catch Blocks..

```
try {
  someObject.test();
  anotherObject.foo();
} catch (InterruptedException e1) {
   // do something with e1
} catch (IOException e2) {
  // do something with e2
} catch (NullPointerException e3) {
  // do something with e3
}
```

checked in order until match is found!

GOTCHA!

```
try {
  someObject.test();
  anotherObject.foo();
} catch (Exception e1) {
   // do something with e1
} catch (IOException e2) {
  // do something with e2
} catch (NullPointerException e3) {
  // do something with e3
}
```

matches ALL Exceptions and subclasses of Exception

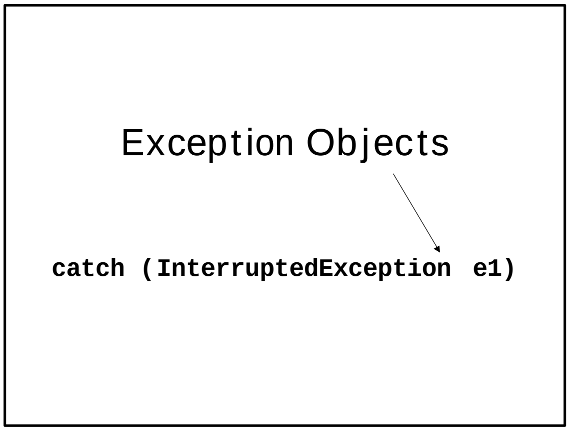
Rule -- Multiple Catch Blocks

```
try {
  someObject.test();
  anotherObject.foo();
} catch (IOException e1) {
   // do something with e1
} catch (NullPointerException e2) {
  // do something with e2
} catch (Exception e3) {
  // do something with e3
}
```

list most specific Exceptions first

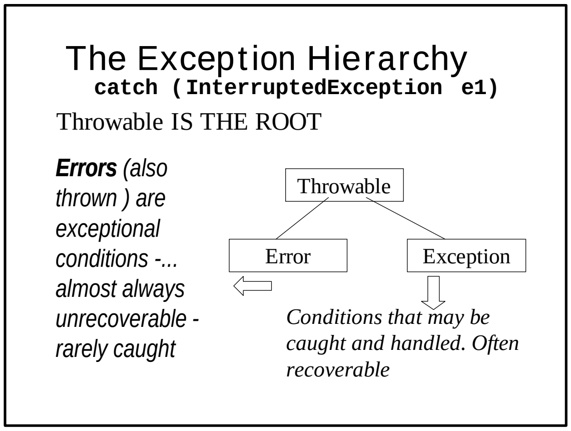
list most general Exceptions last

Exception Objects



Exception Objects

```
catch (InterruptedException  e1)
```



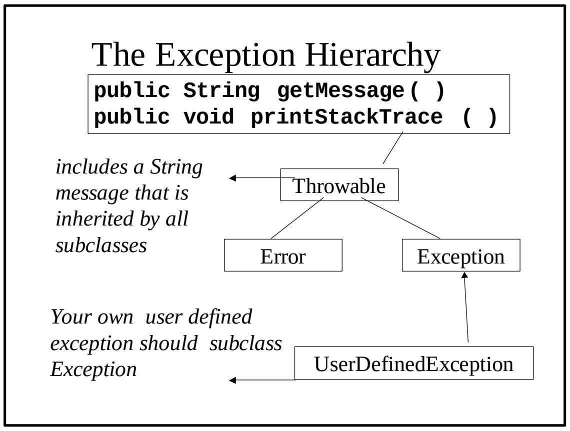
The Exception Hierarchy

```
catch (InterruptedException  e1)
```

Throwable IS THE ROOT

***Errors** (also thrown) are exceptional conditions -... almost always unrecoverable - rarely caught*

Conditions that may be caught and handled. Often recoverable

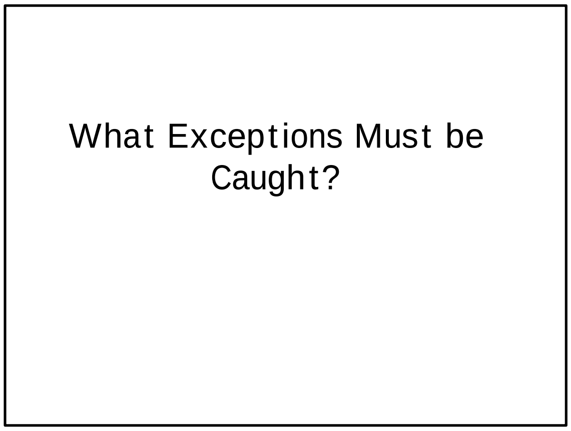


The Exception Hierarchy

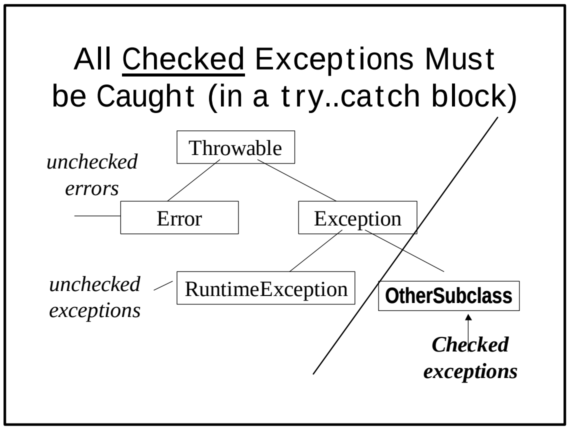
```
public String getMessage( )
public void printStackTrace ( )
```

includes a String message that is inherited by all subclasses

Your own user defined exception should subclass Exception



What Exceptions Must be Caught?

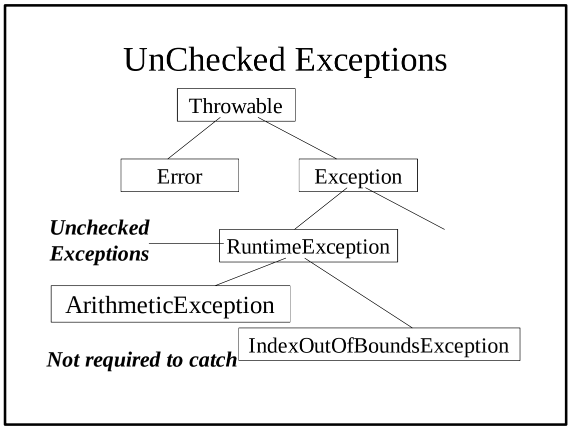


All Checked Exceptions Must be Caught (in a try..catch block)

unchecked errors

unchecked exceptions

Checked exceptions



UnChecked Exceptions

Unchecked Exceptions

Not required to catch

```
public void foo (int k ) {

  int j=1;

  System.out.println( j / k );

}
```

foo (0);

Unchecked Exception is thrown if k = 0

<u>Program aborts with message:</u>

Java.lang ArithmeticException: / by Zero

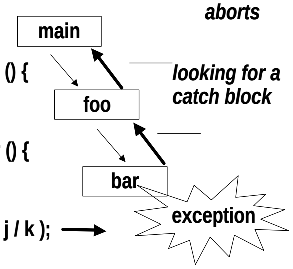
```
public void foo (int k ) {

  int j=1;

  try { System.out.println( j / k );
  }
  catch (ArithmeticException e) {
     System.out.println("Some idiot passed  a zero");
  }

}
```

foo (0);

<u>Program does not abort</u>

Exceptions Travel Up the Call Stack

```
class ExceptionPropagateTest {
 public static void main (String args [ ] ) {

  foo ();
 }
 public static void foo () {
   bar ();
 }
 public static void bar () {
    int j=1, k=0;

    System.out.println( j / k );
 } }
```



program aborts

looking for a catch block

```
class ExceptionPropagateTest {
 public static void main (String args [ ] ) {
  foo ();
 }
 public static void foo () {
   try {  bar (); }
   catch (ArithmeticException e) {
    System.out.println("Caught it in foo ");
   } }
 public static void bar () {
    int j=1, k=0;
    System.out.println( j / k );
 }  }
```

An exception may be caught anywhere in the calling stack

NOTE: This program terminates normally

Writing Your Own Checked Exceptions

```
class BadUserException extends Exception {
  public BadUserException (String name) {
     super(name);
  }
}
```

Exception Summary

- Exceptions are a useful way to structure the normal control flow of an application from the exceptional conditions that may occur
- Checked exceptions must be understood and dealt with by the programmer
- Options are
  - handle in a catch block
  - pass the buck (declare it in your method declaration)

Topic 1 Summary

- Objects are:
  - data
  - methods
- Constructors provide specialized code for object instantiation
- Exceptions allow for specialized treatment of exceptional conditions