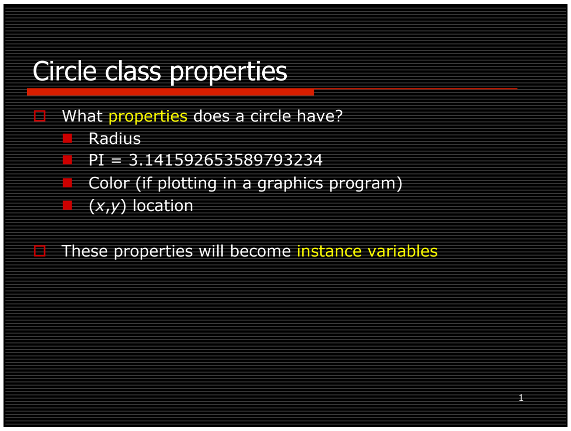
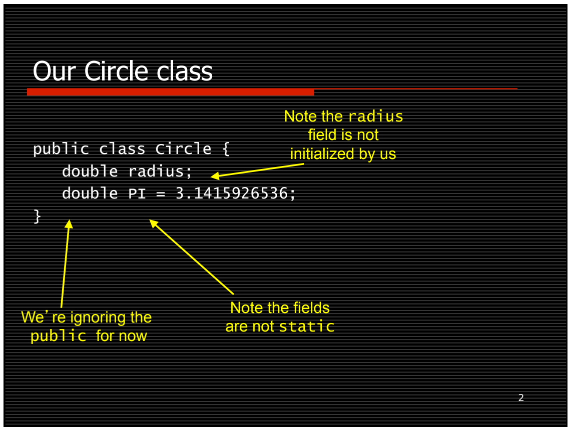


Circle class properties

- What properties does a circle have?
  - Radius
  - PI = 3.141592653589793234
  - Color (if plotting in a graphics program)
  - (*x*,*y*) location
- These properties will become instance variables



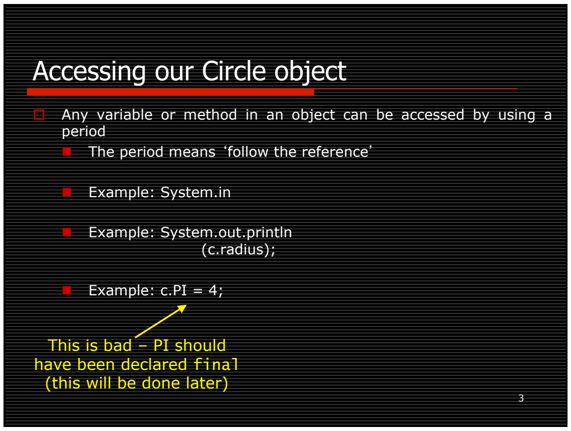
Our Circle class

```
public class Circle {
   double radius;
   double PI = 3.1415926536;
}
```

Note the `radius` field is not initialized by us

We're ignoring the `public` for now

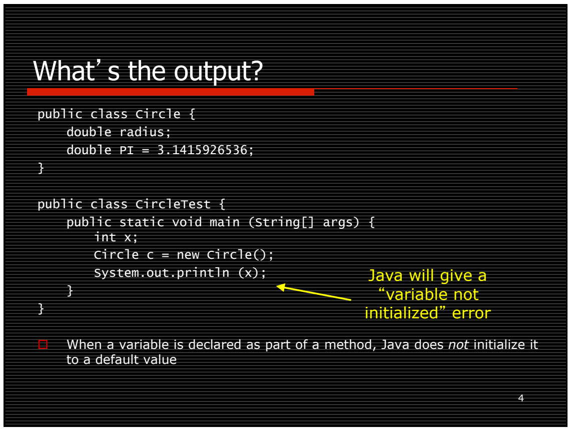
Note the fields are not `static`



Accessing our Circle object

- Any variable or method in an object can be accessed by using a period
  - The period means 'follow the reference'
  - Example: System.in
  - Example: System.out.println (c.radius);
  - Example: c.PI = 4;

This is bad – PI should have been declared `final` (this will be done later)

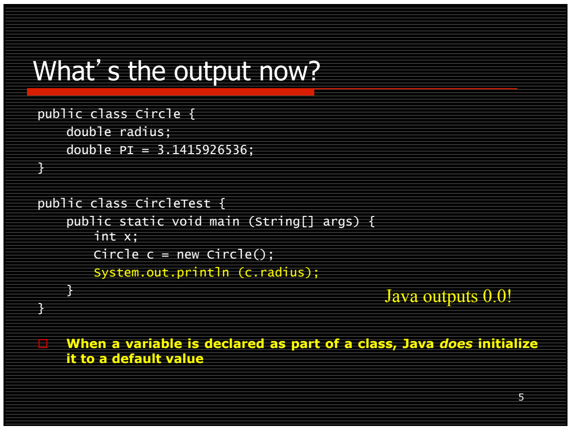


What's the output?

```
public class Circle {
    double radius;
    double PI = 3.1415926536;
}

public class CircleTest {
    public static void main (String[] args) {
        int x;
        Circle c = new Circle();
        System.out.println (x);
    }
}
```

Java will give a "variable not initialized" error

- When a variable is declared as part of a method, Java does *not* initialize it to a default value

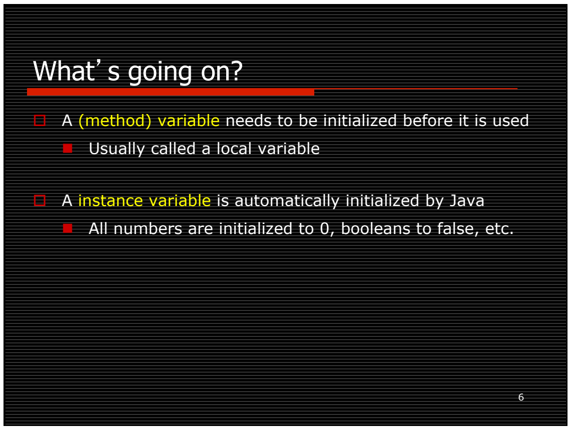


What's the output now?

```
public class Circle {
    double radius;
    double PI = 3.1415926536;
}

public class CircleTest {
    public static void main (String[] args) {
        int x;
        Circle c = new Circle();
        System.out.println (c.radius);
    }
}
```

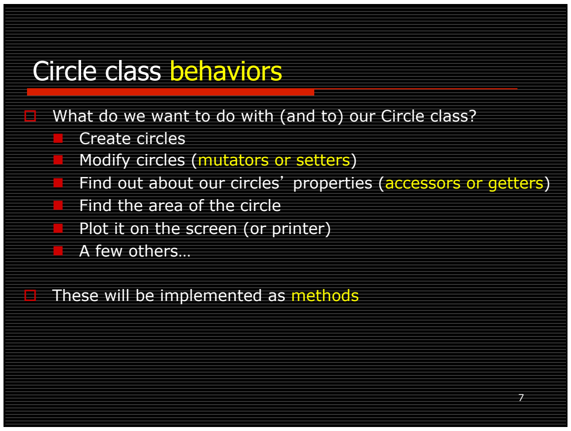
Java outputs 0.0!

- **When a variable is declared as part of a class, Java *does* initialize it to a default value**



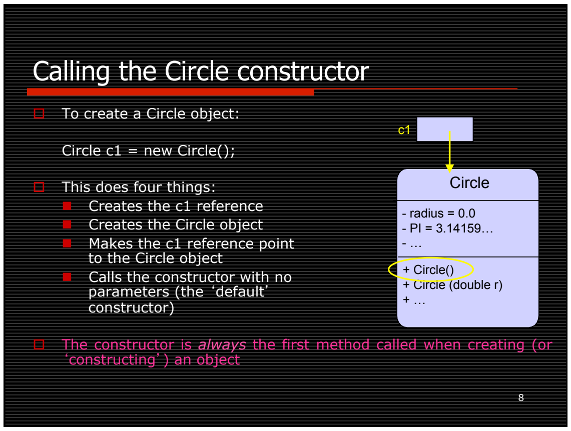
What's going on?

- A (method) variable needs to be initialized before it is used
  - Usually called a local variable
- A instance variable is automatically initialized by Java
  - All numbers are initialized to 0, booleans to false, etc.



Circle class behaviors

- What do we want to do with (and to) our Circle class?
  - Create circles
  - Modify circles (mutators or setters)
  - Find out about our circles' properties (accessors or getters)
  - Find the area of the circle
  - Plot it on the screen (or printer)
  - A few others…
- These will be implemented as methods



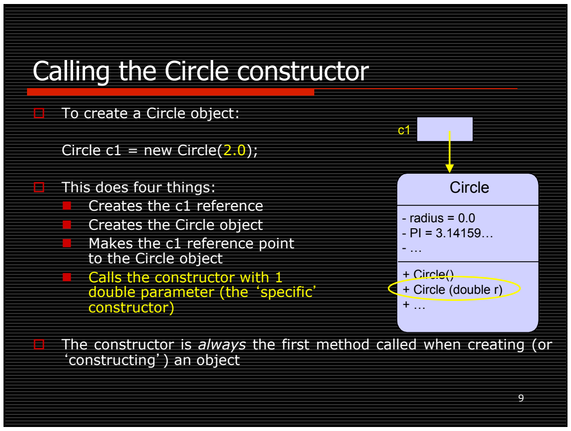
Calling the Circle constructor

- To create a Circle object:

  Circle c1 = new Circle();

- This does four things:
  - Creates the c1 reference
  - Creates the Circle object
  - Makes the c1 reference point to the Circle object
  - Calls the constructor with no parameters (the 'default' constructor)

- The constructor is *always* the first method called when creating (or 'constructing') an object



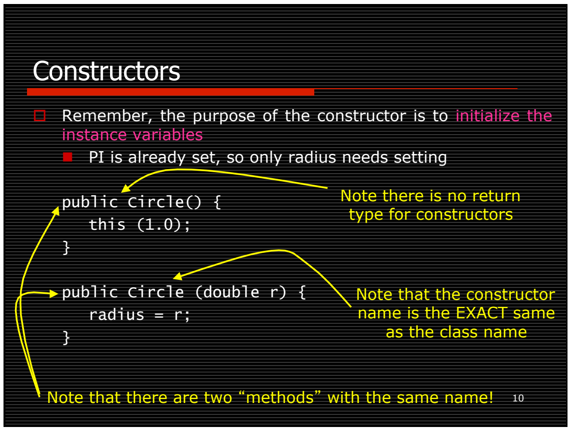
Calling the Circle constructor

- To create a Circle object:

  Circle c1 = new Circle(2.0);

- This does four things:
  - Creates the c1 reference
  - Creates the Circle object
  - Makes the c1 reference point to the Circle object
  - Calls the constructor with 1 double parameter (the 'specific' constructor)

- The constructor is *always* the first method called when creating (or 'constructing') an object

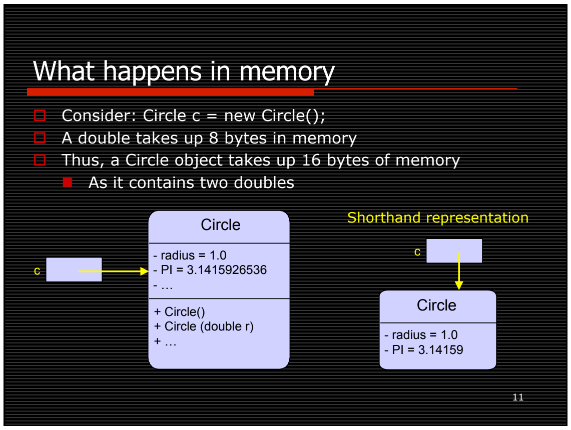


Constructors

- Remember, the purpose of the constructor is to initialize the instance variables
  - PI is already set, so only radius needs setting

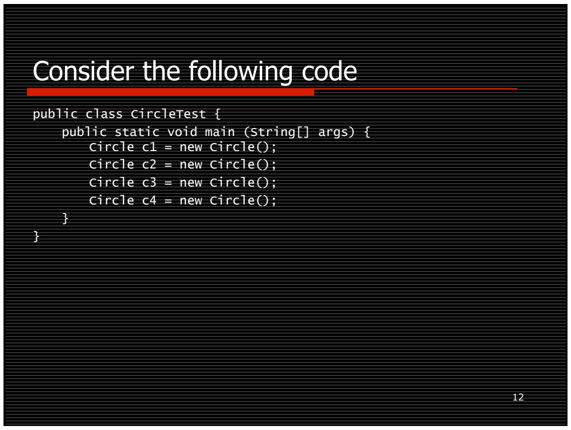
```
public Circle() {
    this (1.0);
}

public Circle (double r) {
    radius = r;
}
```

Note there is no return type for constructors
Note that the constructor name is the EXACT same as the class name
Note that there are two "methods" with the same name!



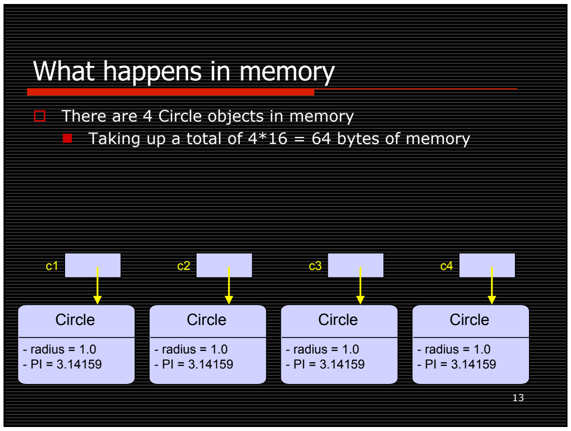
What happens in memory

- Consider: Circle c = new Circle();
- A double takes up 8 bytes in memory
- Thus, a Circle object takes up 16 bytes of memory
  - As it contains two doubles



Consider the following code

```
public class CircleTest {
    public static void main (String[] args) {
        Circle c1 = new Circle();
        Circle c2 = new Circle();
        Circle c3 = new Circle();
        Circle c4 = new Circle();
    }
}
```



What happens in memory

- There are 4 Circle objects in memory
  - Taking up a total of 4*16 = 64 bytes of memory

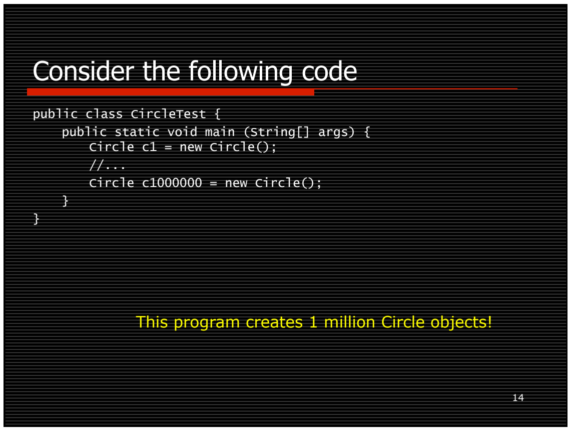
c1 c2 c3 c4

Circle
- radius = 1.0
- PI = 3.14159

Circle
- radius = 1.0
- PI = 3.14159

Circle
- radius = 1.0
- PI = 3.14159

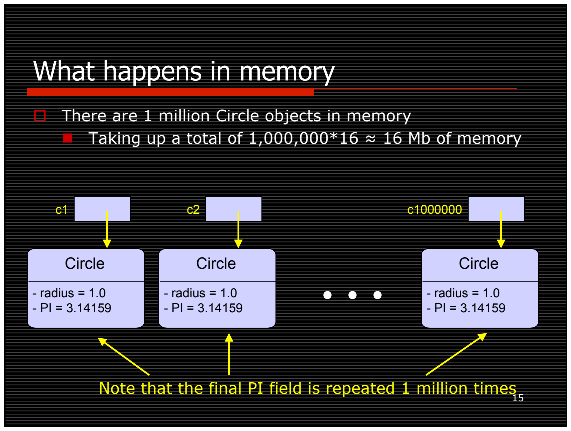
Circle
- radius = 1.0
- PI = 3.14159



Consider the following code

```
public class CircleTest {
    public static void main (String[] args) {
        Circle c1 = new Circle();
        //...
        Circle c1000000 = new Circle();
    }
}
```

This program creates 1 million Circle objects!



What happens in memory

- There are 1 million Circle objects in memory
  - Taking up a total of 1,000,000*16 ≈ 16 Mb of memory

c1 c2 c1000000

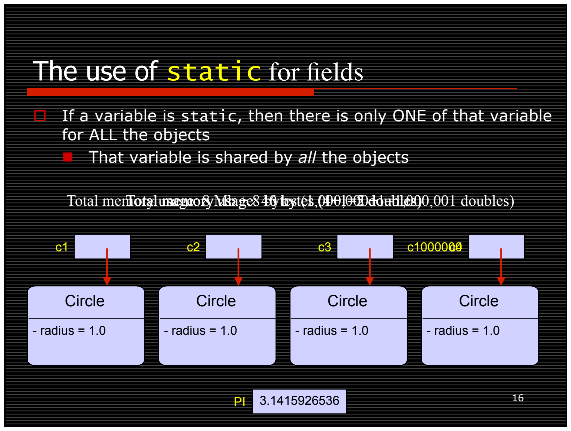
Circle
- radius = 1.0
- PI = 3.14159

Circle
- radius = 1.0
- PI = 3.14159

• • •

Circle
- radius = 1.0
- PI = 3.14159

Note that the final PI field is repeated 1 million times



The use of `static` for fields

- If a variable is `static`, then there is only ONE of that variable for ALL the objects
  - That variable is shared by *all* the objects

Total memory usage: 8 Mb + 8 bytes (1,000,001 doubles)

Total memory usage: 40 bytes (4+1=5 doubles)

c1 c2 c3 c1000000

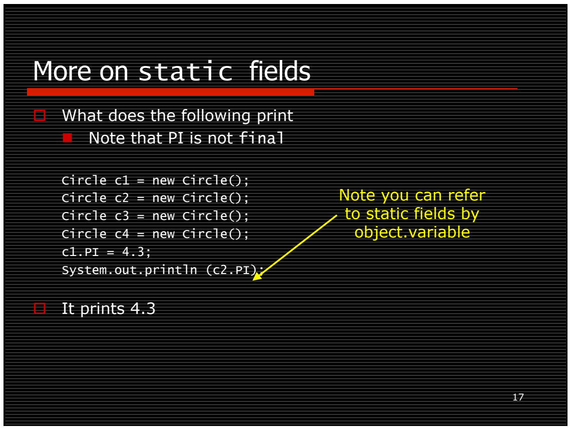
Circle
- radius = 1.0

Circle
- radius = 1.0

Circle
- radius = 1.0

Circle
- radius = 1.0

PI 3.1415926536



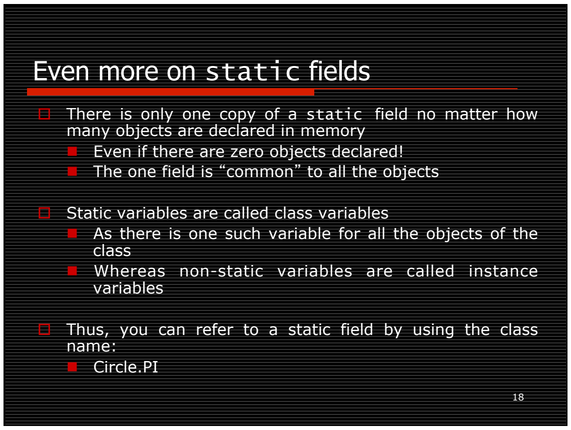
More on `static` fields

- What does the following print
  - Note that PI is not `final`

```
Circle c1 = new Circle();
Circle c2 = new Circle();
Circle c3 = new Circle();
Circle c4 = new Circle();
c1.PI = 4.3;
System.out.println (c2.PI);
```

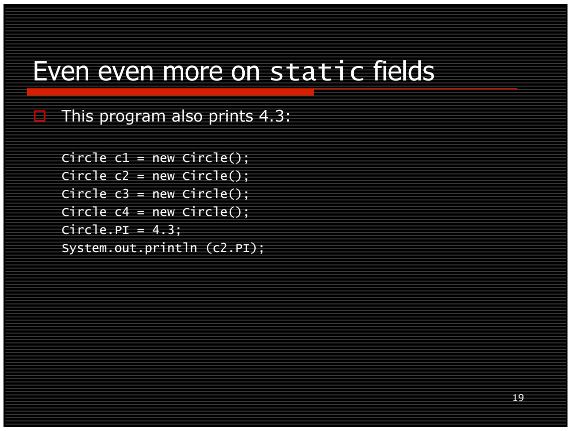
Note you can refer to static fields by object.variable

- It prints 4.3



Even more on `static` fields

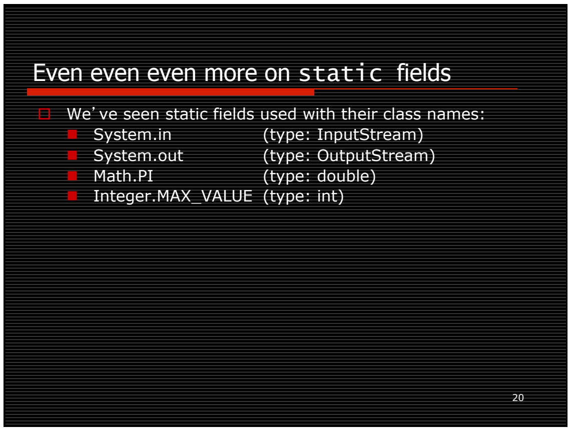
- There is only one copy of a `static` field no matter how many objects are declared in memory
  - Even if there are zero objects declared!
  - The one field is “common” to all the objects
- Static variables are called class variables
  - As there is one such variable for all the objects of the class
  - Whereas non-static variables are called instance variables
- Thus, you can refer to a static field by using the class name:
  - Circle.PI



Even even more on static fields

- This program also prints 4.3:

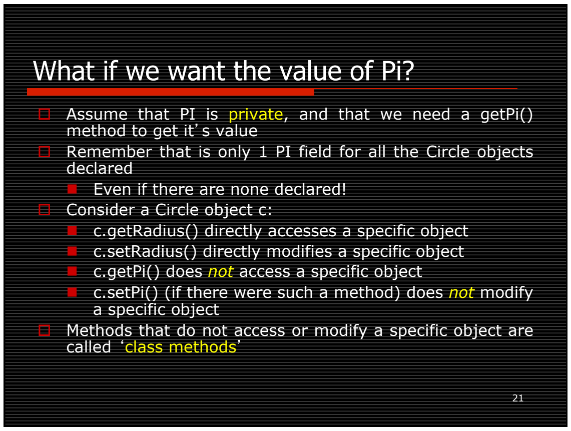
```
Circle c1 = new Circle();
Circle c2 = new Circle();
Circle c3 = new Circle();
Circle c4 = new Circle();
Circle.PI = 4.3;
System.out.println (c2.PI);
```



Even even even more on static fields

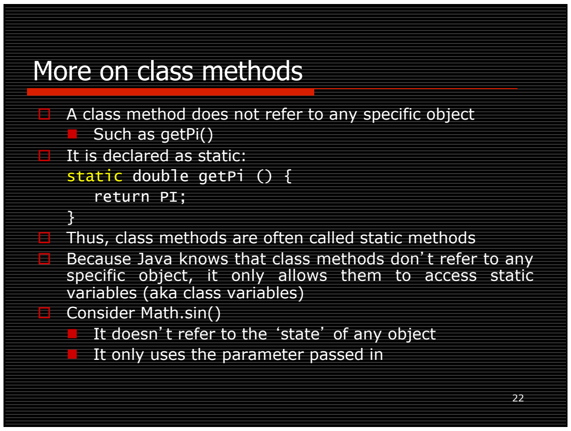
- We’ve seen static fields used with their class names:

System.in	(type: InputStream)
System.out	(type: OutputStream)
Math.PI	(type: double)
Integer.MAX_VALUE	(type: int)



What if we want the value of Pi?

- Assume that PI is private, and that we need a getPi() method to get it’s value
- Remember that is only 1 PI field for all the Circle objects declared
  - Even if there are none declared!
- Consider a Circle object c:
  - c.getRadius() directly accesses a specific object
  - c.setRadius() directly modifies a specific object
  - c.getPi() does *not* access a specific object
  - c.setPi() (if there were such a method) does *not* modify a specific object
- Methods that do not access or modify a specific object are called ‘class methods’

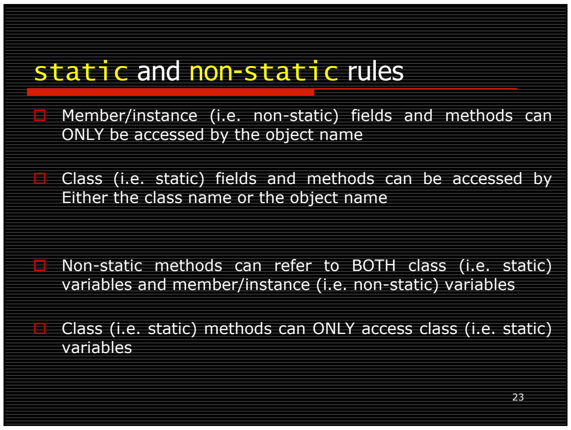


More on class methods

- A class method does not refer to any specific object
  - Such as getPi()
- It is declared as static:

```
static double getPi () {
   return PI;
}
```

- Thus, class methods are often called static methods
- Because Java knows that class methods don’t refer to any specific object, it only allows them to access static variables (aka class variables)
- Consider Math.sin()
  - It doesn’t refer to the ‘state’ of any object
  - It only uses the parameter passed in



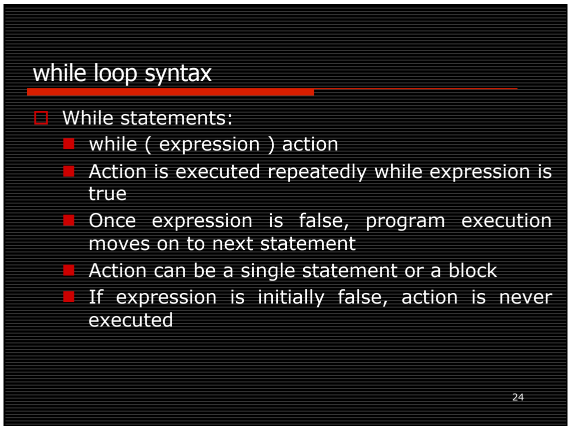
static and non-static rules

- Member/instance (i.e. non-static) fields and methods can ONLY be accessed by the object name

- Class (i.e. static) fields and methods can be accessed by Either the class name or the object name

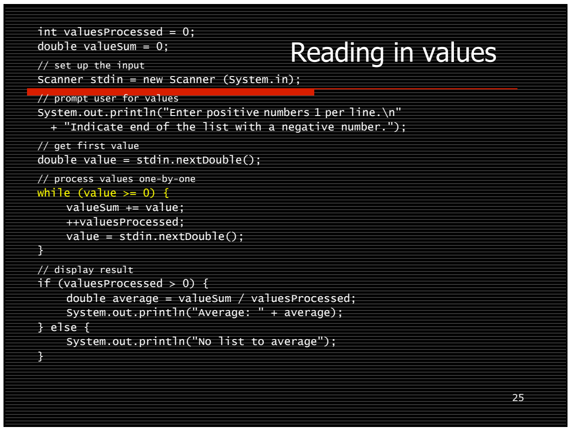
- Non-static methods can refer to BOTH class (i.e. static) variables and member/instance (i.e. non-static) variables

- Class (i.e. static) methods can ONLY access class (i.e. static) variables



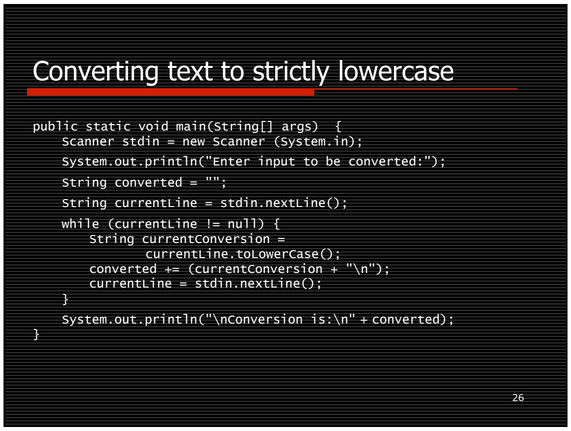
while loop syntax

- While statements:
  - while (expression) action
  - Action is executed repeatedly while expression is true
  - Once expression is false, program execution moves on to next statement
  - Action can be a single statement or a block
  - If expression is initially false, action is never executed



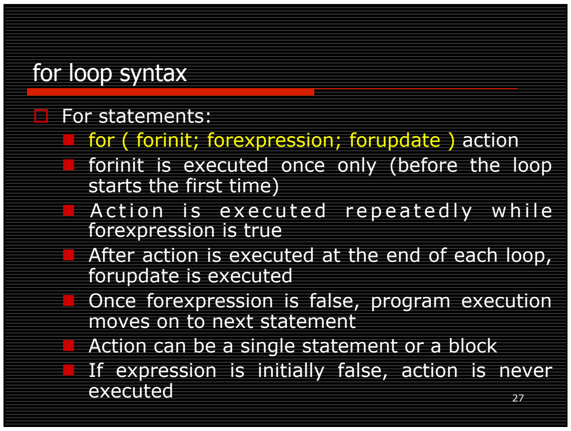
Reading in values

```
int valuesProcessed = 0;
double valueSum = 0;
// set up the input
Scanner stdin = new Scanner (System.in);
// prompt user for values
System.out.println("Enter positive numbers 1 per line.\n"
  + "Indicate end of the list with a negative number.");
// get first value
double value = stdin.nextDouble();
// process values one-by-one
while (value >= 0) {
     valueSum += value;
     ++valuesProcessed;
     value = stdin.nextDouble();
}
// display result
if (valuesProcessed > 0) {
     double average = valueSum / valuesProcessed;
     System.out.println("Average: " + average);
} else {
     System.out.println("No list to average");
}
```



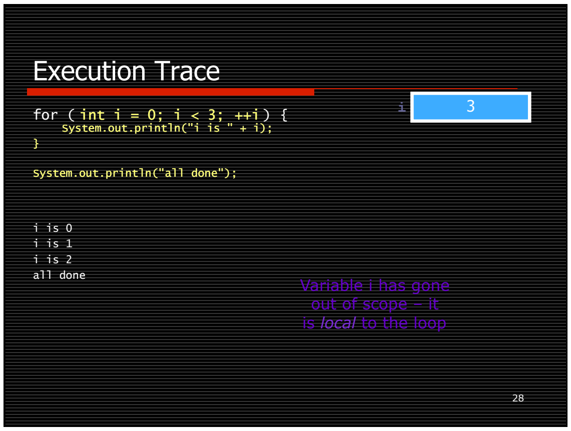
Converting text to strictly lowercase

```
public static void main(String[] args)  {
     Scanner stdin = new Scanner (System.in);
     System.out.println("Enter input to be converted:");
     String converted = "";
     String currentLine = stdin.nextLine();
     while (currentLine != null) {
          String currentConversion =
                    currentLine.toLowerCase();
          converted += (currentConversion + "\n");
          currentLine = stdin.nextLine();
     }
     System.out.println("\nConversion is:\n" + converted);
}
```



for loop syntax

- For statements:
  - for (forinit; forexpression; forupdate) action
  - forinit is executed once only (before the loop starts the first time)
  - Action is executed repeatedly while forexpression is true
  - After action is executed at the end of each loop, forupdate is executed
  - Once forexpression is false, program execution moves on to next statement
  - Action can be a single statement or a block
  - If expression is initially false, action is never executed

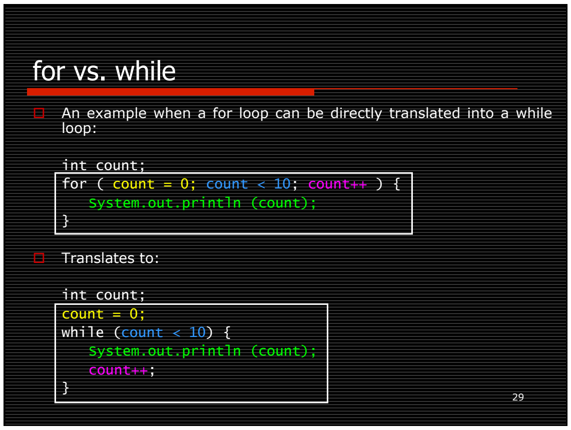


Execution Trace

```
for ( int i = 0; i < 3; ++i) {
    System.out.println("i is " + i);
}

System.out.println("all done");
```

```
i is 0
i is 1
i is 2
all done
```

Variable i has gone out of scope – it is *local* to the loop



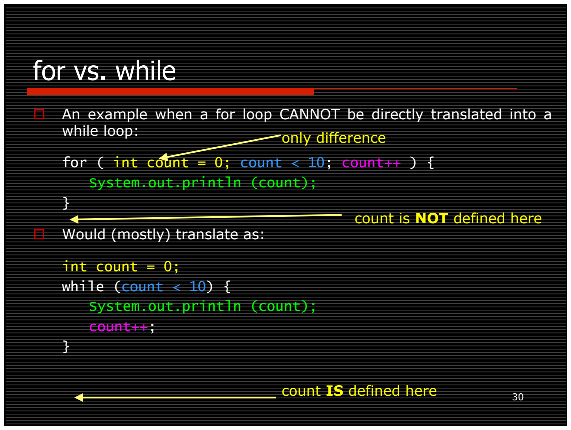
for vs. while

- An example when a for loop can be directly translated into a while loop:

```
int count;
for ( count = 0; count < 10; count++ ) {
    System.out.println (count);
}
```

- Translates to:

```
int count;
count = 0;
while (count < 10) {
    System.out.println (count);
    count++;
}
```



for vs. while

- An example when a for loop CANNOT be directly translated into a while loop:

only difference

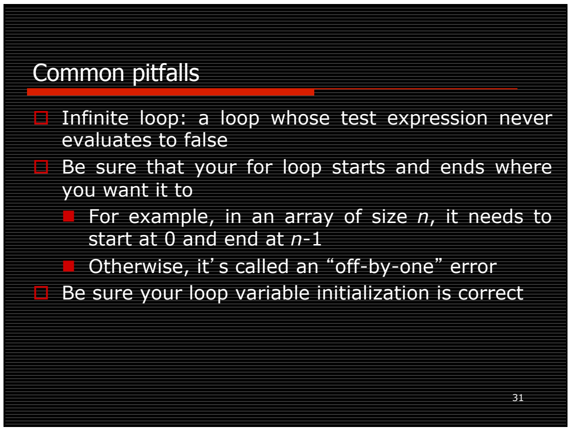
```
for ( int count = 0; count < 10; count++ ) {
    System.out.println (count);
}
```

count is **NOT** defined here

- Would (mostly) translate as:

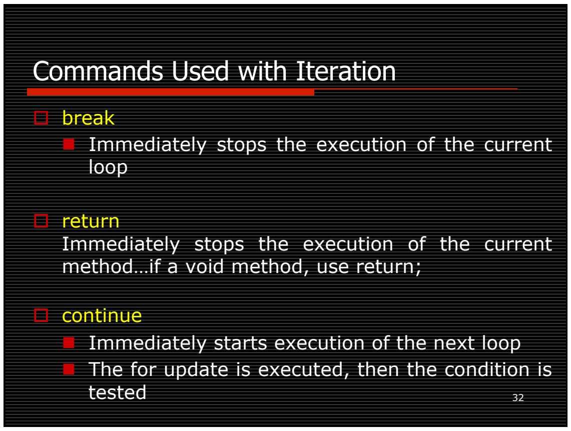
```
int count = 0;
while (count < 10) {
    System.out.println (count);
    count++;
}
```

count **IS** defined here



Common pitfalls

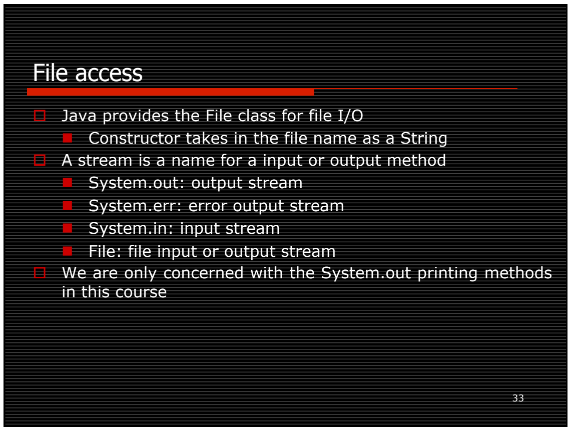
- Infinite loop: a loop whose test expression never evaluates to false
- Be sure that your for loop starts and ends where you want it to
  - For example, in an array of size *n*, it needs to start at 0 and end at *n*-1
  - Otherwise, it’s called an “off-by-one” error
- Be sure your loop variable initialization is correct



Commands Used with Iteration

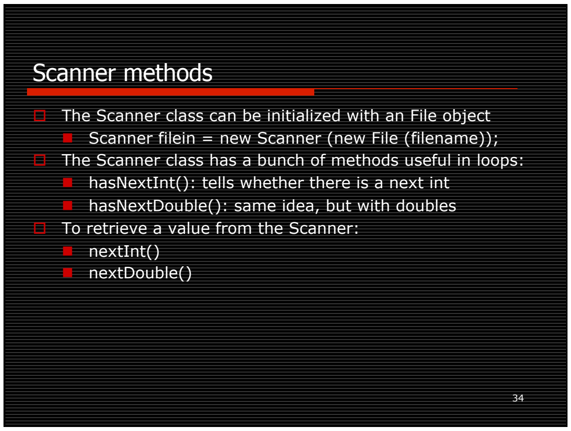
- break
  - Immediately stops the execution of the current loop
- return

  Immediately stops the execution of the current method…if a void method, use return;
- continue
  - Immediately starts execution of the next loop
  - The for update is executed, then the condition is tested



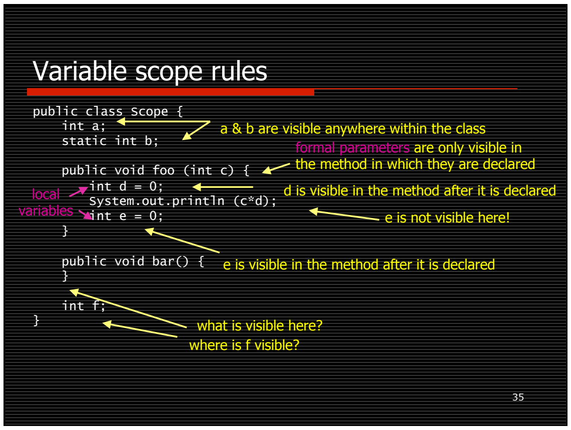
File access

- Java provides the File class for file I/O
  - Constructor takes in the file name as a String
- A stream is a name for a input or output method
  - System.out: output stream
  - System.err: error output stream
  - System.in: input stream
  - File: file input or output stream
- We are only concerned with the System.out printing methods in this course



Scanner methods

- The Scanner class can be initialized with an File object
  - Scanner filein = new Scanner (new File (filename));
- The Scanner class has a bunch of methods useful in loops:
  - hasNextInt(): tells whether there is a next int
  - hasNextDouble(): same idea, but with doubles
- To retrieve a value from the Scanner:
  - nextInt()
  - nextDouble()



Variable scope rules

```
public class Scope {
    int a;
    static int b;

    public void foo (int c) {
        int d = 0;
        System.out.println (c*d);
        int e = 0;
    }

    public void bar() {
    }

    int f;
}
```

a & b are visible anywhere within the class

formal parameters are only visible in the method in which they are declared

local variables

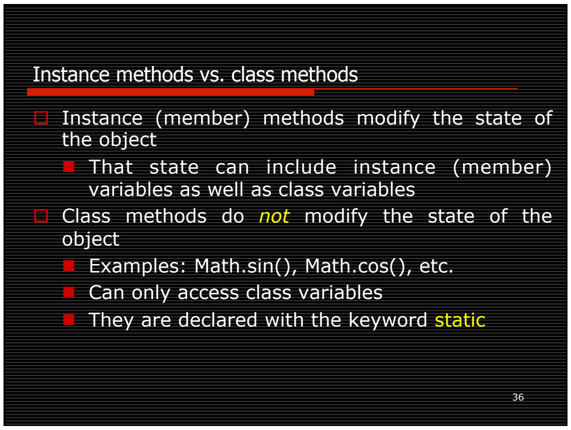
d is visible in the method after it is declared

e is not visible here!

e is visible in the method after it is declared

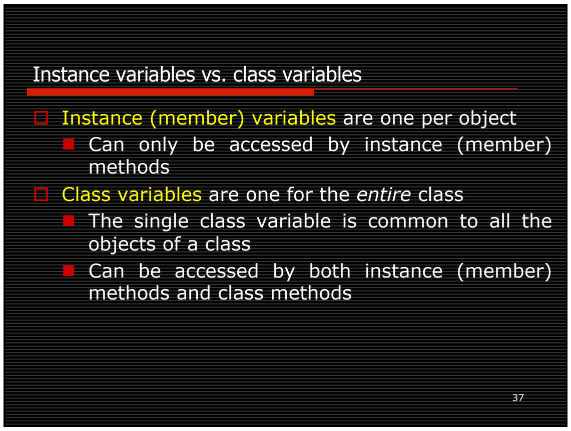
what is visible here?

where is f visible?



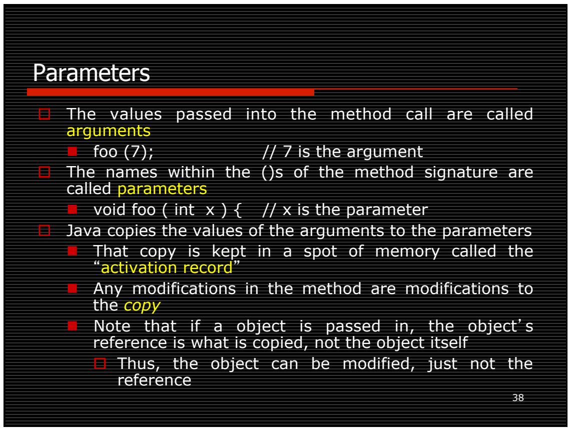
Instance methods vs. class methods

- Instance (member) methods modify the state of the object
  - That state can include instance (member) variables as well as class variables
- Class methods do *not* modify the state of the object
  - Examples: Math.sin(), Math.cos(), etc.
  - Can only access class variables
  - They are declared with the keyword static



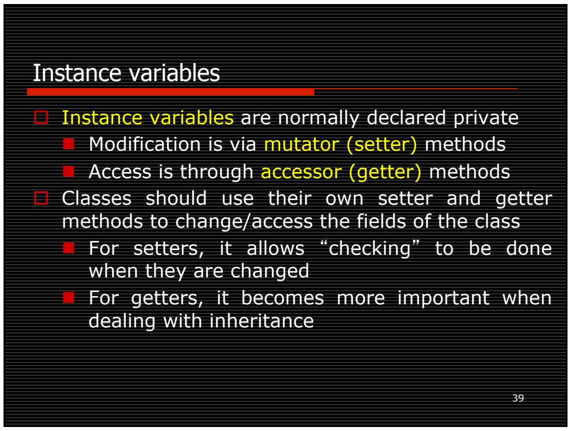
Instance variables vs. class variables

- Instance (member) variables are one per object
  - Can only be accessed by instance (member) methods
- Class variables are one for the *entire* class
  - The single class variable is common to all the objects of a class
  - Can be accessed by both instance (member) methods and class methods



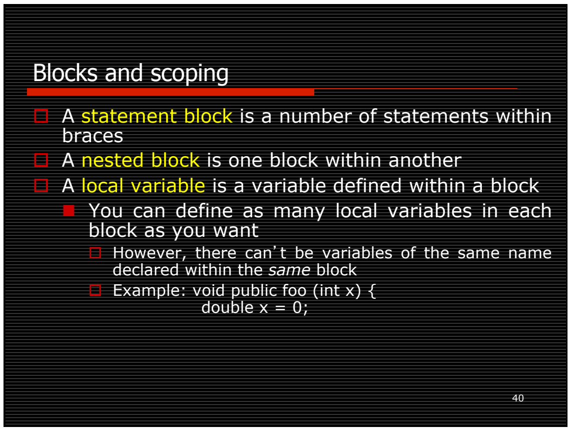
Parameters

- The values passed into the method call are called arguments
  - foo (7);   // 7 is the argument
- The names within the ()s of the method signature are called parameters
  - void foo (int x) {   // x is the parameter
- Java copies the values of the arguments to the parameters
  - That copy is kept in a spot of memory called the "activation record"
  - Any modifications in the method are modifications to the *copy*
  - Note that if a object is passed in, the object's reference is what is copied, not the object itself
    - Thus, the object can be modified, just not the reference



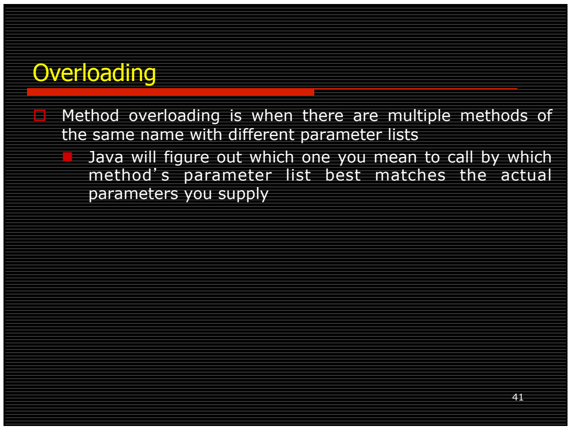
Instance variables

- Instance variables are normally declared private
  - Modification is via mutator (setter) methods
  - Access is through accessor (getter) methods
- Classes should use their own setter and getter methods to change/access the fields of the class
  - For setters, it allows "checking" to be done when they are changed
  - For getters, it becomes more important when dealing with inheritance



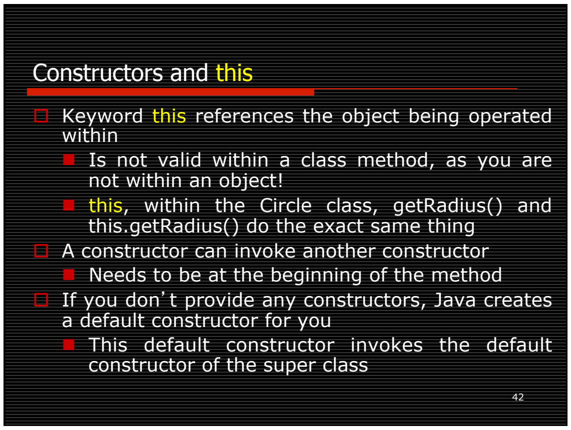
Blocks and scoping

- A statement block is a number of statements within braces
- A nested block is one block within another
- A local variable is a variable defined within a block
  - You can define as many local variables in each block as you want
    - However, there can't be variables of the same name declared within the *same* block
    - Example: void public foo (int x) {
      double x = 0;



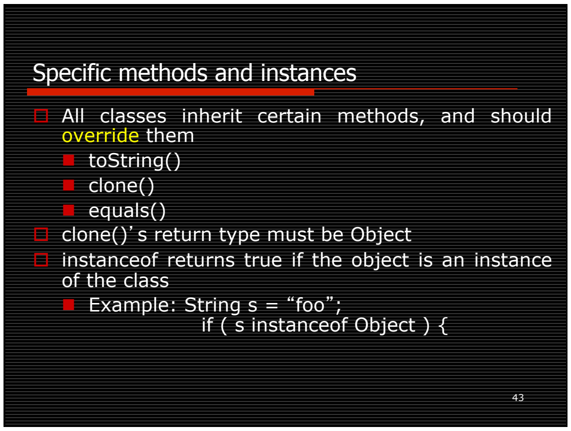
Overloading

- Method overloading is when there are multiple methods of the same name with different parameter lists
  - Java will figure out which one you mean to call by which method's parameter list best matches the actual parameters you supply



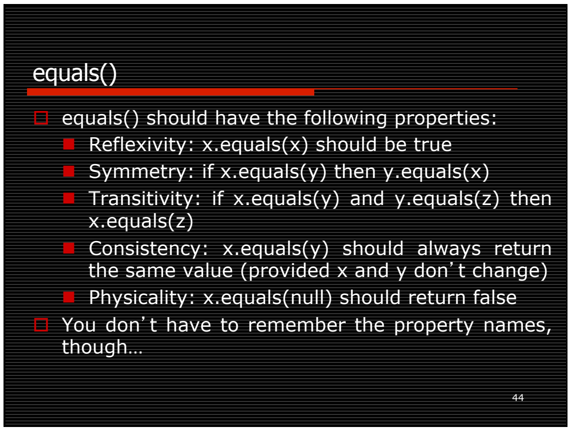
Constructors and this

- Keyword this references the object being operated within
  - Is not valid within a class method, as you are not within an object!
  - this, within the Circle class, getRadius() and this.getRadius() do the exact same thing
- A constructor can invoke another constructor
  - Needs to be at the beginning of the method
- If you don't provide any constructors, Java creates a default constructor for you
  - This default constructor invokes the default constructor of the super class



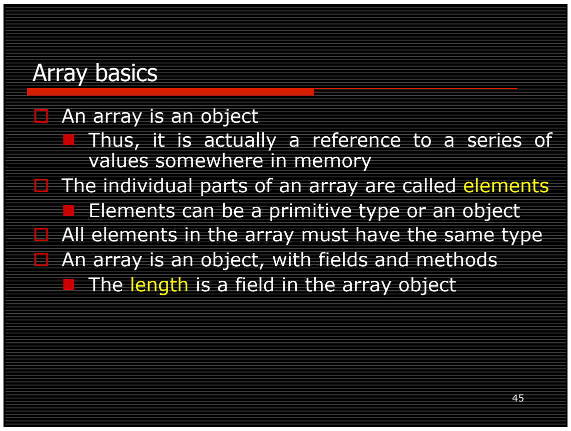
Specific methods and instances

- All classes inherit certain methods, and should override them
  - toString()
  - clone()
  - equals()
- clone()’s return type must be Object
- instanceof returns true if the object is an instance of the class
  - Example: String s = “foo”;
    if (s instanceof Object) {



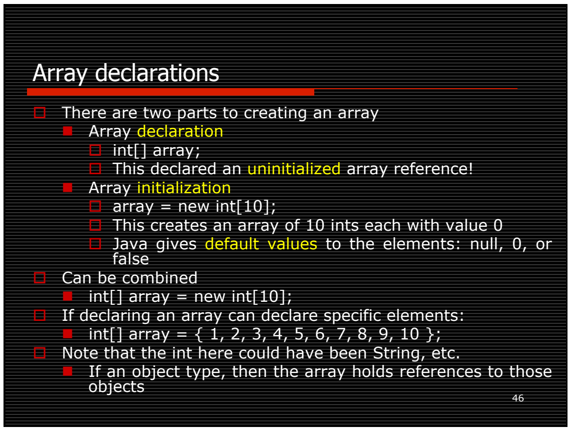
equals()

- equals() should have the following properties:
  - Reflexivity: x.equals(x) should be true
  - Symmetry: if x.equals(y) then y.equals(x)
  - Transitivity: if x.equals(y) and y.equals(z) then x.equals(z)
  - Consistency: x.equals(y) should always return the same value (provided x and y don’t change)
  - Physicality: x.equals(null) should return false
- You don’t have to remember the property names, though…



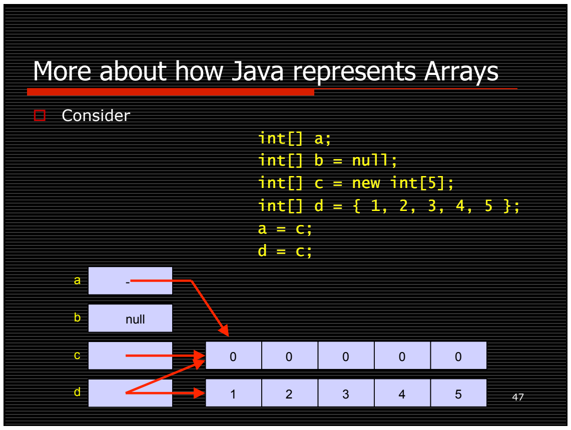
Array basics

- An array is an object
  - Thus, it is actually a reference to a series of values somewhere in memory
- The individual parts of an array are called elements
  - Elements can be a primitive type or an object
- All elements in the array must have the same type
- An array is an object, with fields and methods
  - The length is a field in the array object



Array declarations

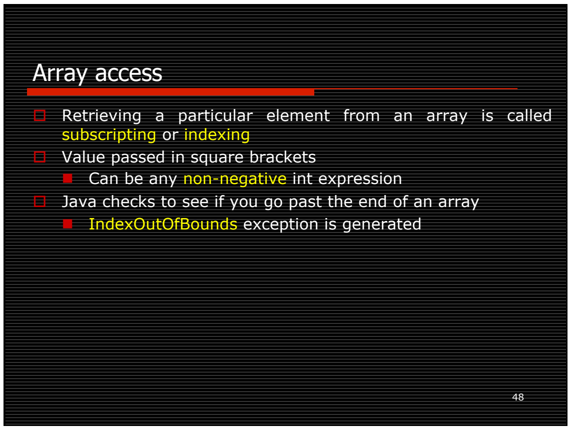
- There are two parts to creating an array
  - Array declaration
    - int[] array;
    - This declared an uninitialized array reference!
  - Array initialization
    - array = new int[10];
    - This creates an array of 10 ints each with value 0
    - Java gives default values to the elements: null, 0, or false
- Can be combined
  - int[] array = new int[10];
- If declaring an array can declare specific elements:
  - int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
- Note that the int here could have been String, etc.
  - If an object type, then the array holds references to those objects



More about how Java represents Arrays

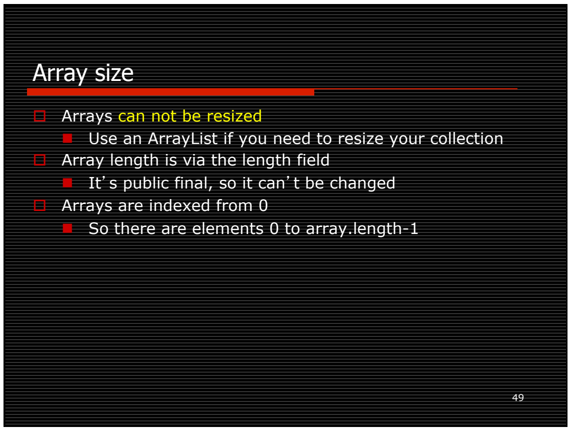
- Consider

```
int[] a;
int[] b = null;
int[] c = new int[5];
int[] d = { 1, 2, 3, 4, 5 };
a = c;
d = c;
```



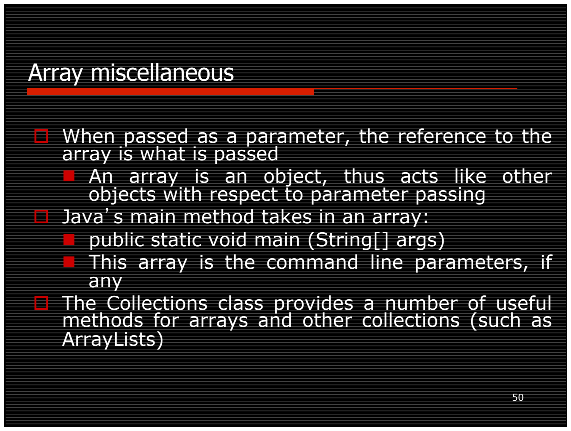
Array access

- Retrieving a particular element from an array is called subscripting or indexing
- Value passed in square brackets
  - Can be any non-negative int expression
- Java checks to see if you go past the end of an array
  - IndexOutOfBounds exception is generated



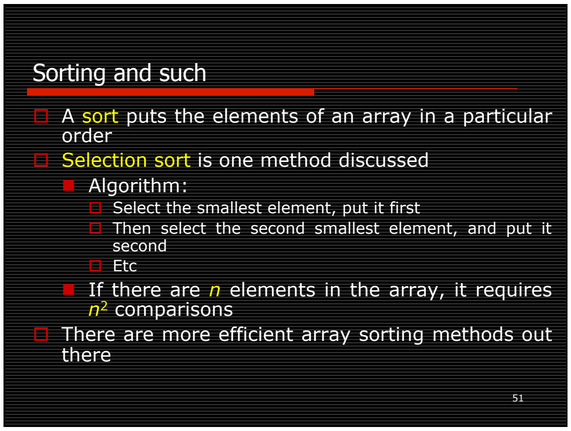
Array size

- Arrays can not be resized
  - Use an ArrayList if you need to resize your collection
- Array length is via the length field
  - It's public final, so it can't be changed
- Arrays are indexed from 0
  - So there are elements 0 to array.length-1



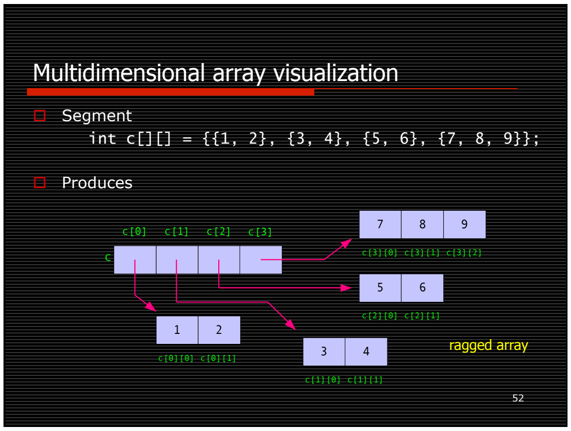
Array miscellaneous

- When passed as a parameter, the reference to the array is what is passed
  - An array is an object, thus acts like other objects with respect to parameter passing
- Java's main method takes in an array:
  - public static void main (String[] args)
  - This array is the command line parameters, if any
- The Collections class provides a number of useful methods for arrays and other collections (such as ArrayLists)



Sorting and such

- A sort puts the elements of an array in a particular order
- Selection sort is one method discussed
  - Algorithm:
    - Select the smallest element, put it first
    - Then select the second smallest element, and put it second
    - Etc
  - If there are *n* elements in the array, it requires n^2 comparisons
- There are more efficient array sorting methods out there

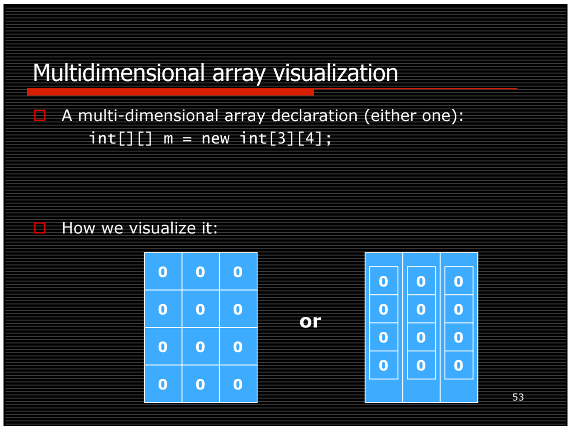


Multidimensional array visualization

- Segment

```
int c[][] = {{1, 2}, {3, 4}, {5, 6}, {7, 8, 9}};
```

- Produces

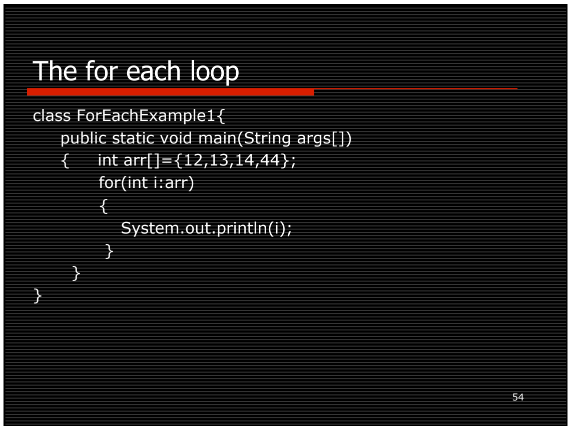


Multidimensional array visualization

- A multi-dimensional array declaration (either one):

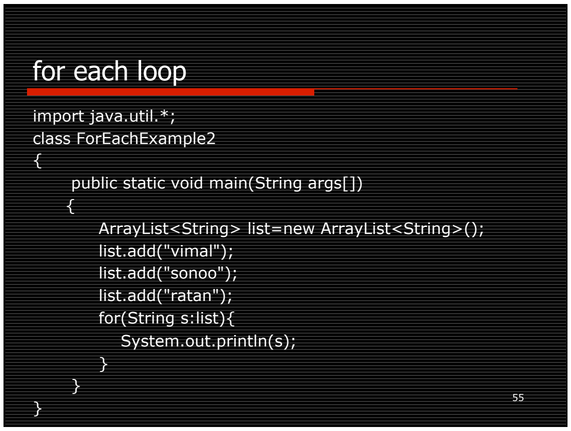
```
int[][] m = new int[3][4];
```

- How we visualize it:



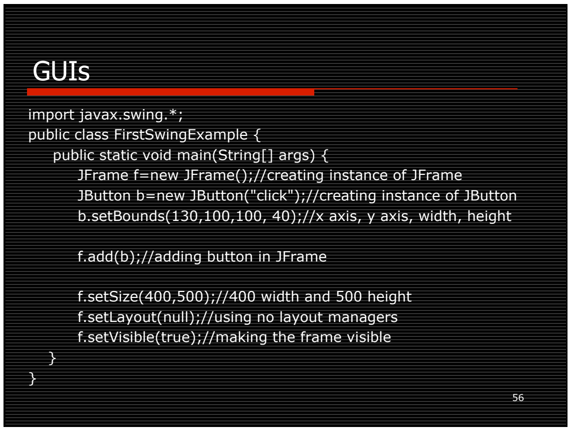
The for each loop

```
class ForEachExample1{
    public static void main(String args[])
    {     int arr[]={12,13,14,44};
          for(int i:arr)
          {
             System.out.println(i);
           }
      }
}
```



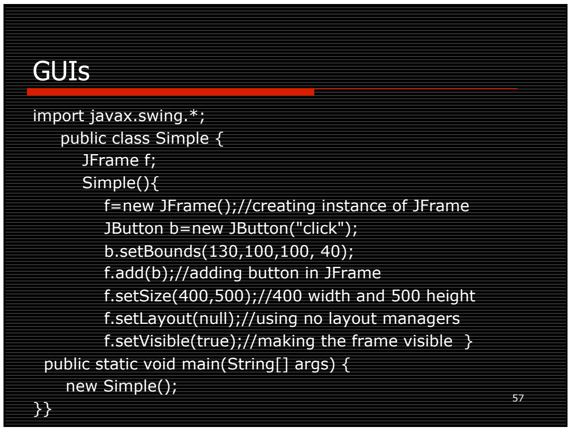
for each loop

```
import java.util.*;
class ForEachExample2
{
       public static void main(String args[])
      {
            ArrayList<String> list=new ArrayList<String>();
            list.add("vimal");
            list.add("sonoo");
            list.add("ratan");
            for(String s:list){
                System.out.println(s);
            }
       }
}
```



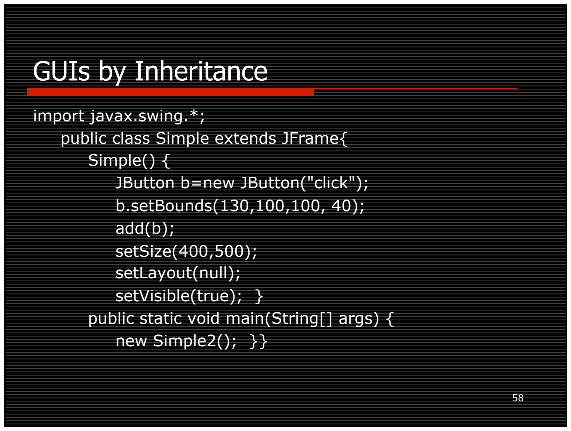
GUIs

```
import javax.swing.*;
public class FirstSwingExample {
     public static void main(String[] args) {
          JFrame f=new JFrame();//creating instance of JFrame
          JButton b=new JButton("click");//creating instance of JButton
          b.setBounds(130,100,100, 40);//x axis, y axis, width, height

          f.add(b);//adding button in JFrame

          f.setSize(400,500);//400 width and 500 height
          f.setLayout(null);//using no layout managers
          f.setVisible(true);//making the frame visible
    }
}
```



GUIs

```
import javax.swing.*;
     public class Simple {
         JFrame f;
         Simple(){
             f=new JFrame();//creating instance of JFrame
             JButton b=new JButton("click");
             b.setBounds(130,100,100, 40);
             f.add(b);//adding button in JFrame
             f.setSize(400,500);//400 width and 500 height
             f.setLayout(null);//using no layout managers
             f.setVisible(true);//making the frame visible  }
  public static void main(String[] args) {
      new Simple();
}}
```



GUIs by Inheritance

```
import javax.swing.*;
     public class Simple extends JFrame{
          Simple() {
               JButton b=new JButton("click");
               b.setBounds(130,100,100, 40);
               add(b);
               setSize(400,500);
               setLayout(null);
               setVisible(true);  }
          public static void main(String[] args) {
               new Simple2();  }}
```