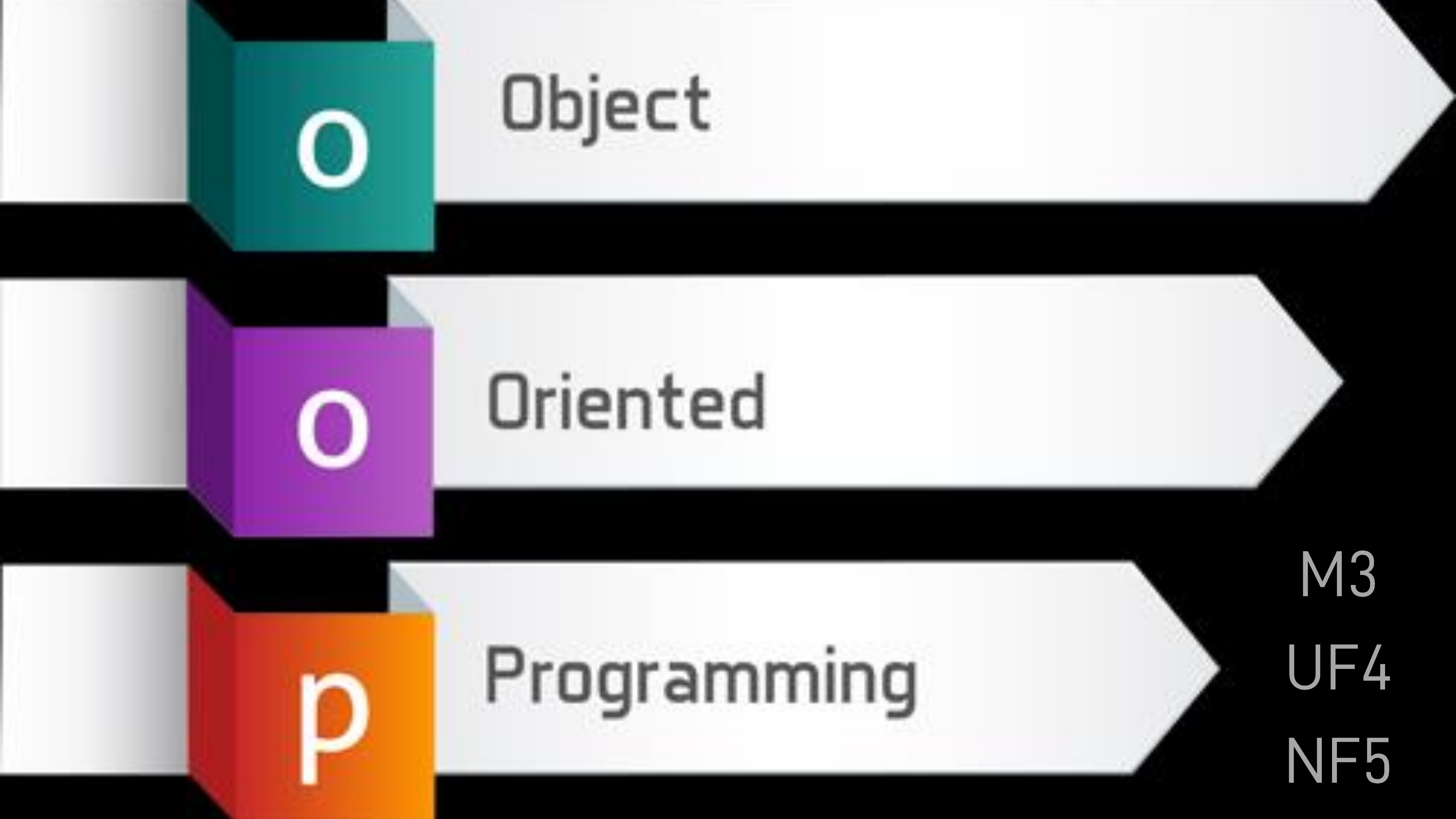


O Object

O Oriented

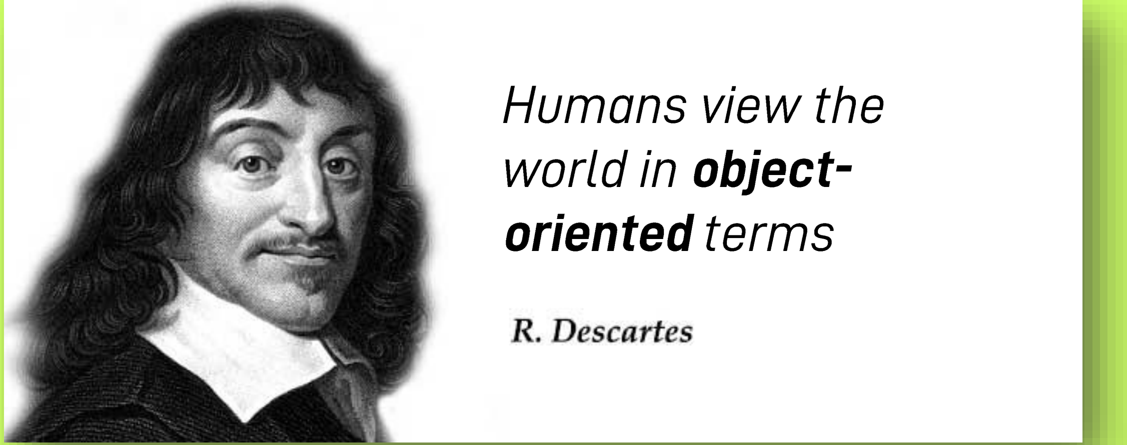
P Programming

M3

UF4

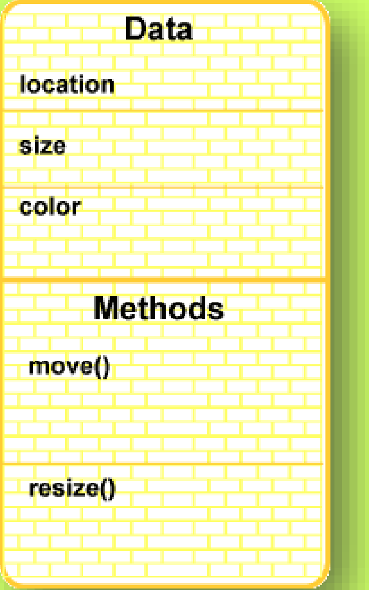
NF5

Objects



Humans view the world in ***object-oriented*** *terms*

R. Descartes



- Organize software in a way that matches the thinking style of our object-oriented brains.
- We want objects that have properties and interact with other objects.

Characteristics of objects:

- **Identity** (each object is distinct)
- **State** (properties)
- **Behavior** (methods)

Computer memory

What are software objects made out of?

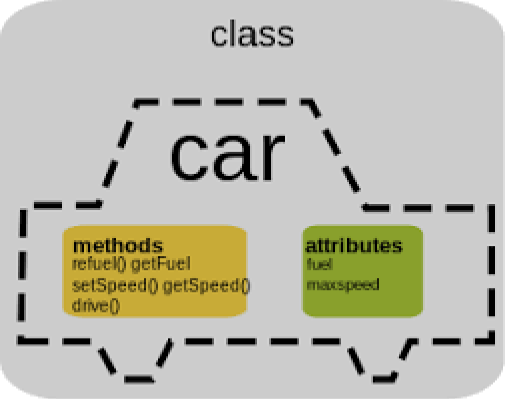
Class

A **class** is a description ect.

A class is merely a plan bject.

Not still an object, only a “place holder”

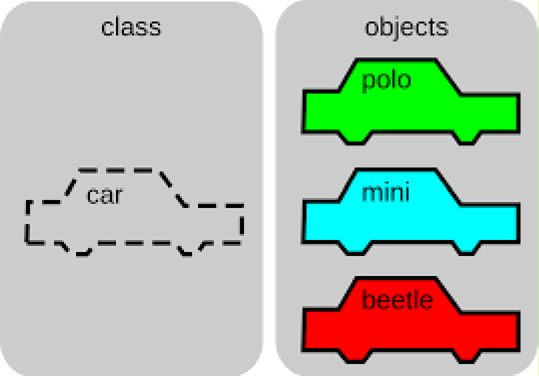
```
String str1;
str1 = new String("Random Jottings");
```



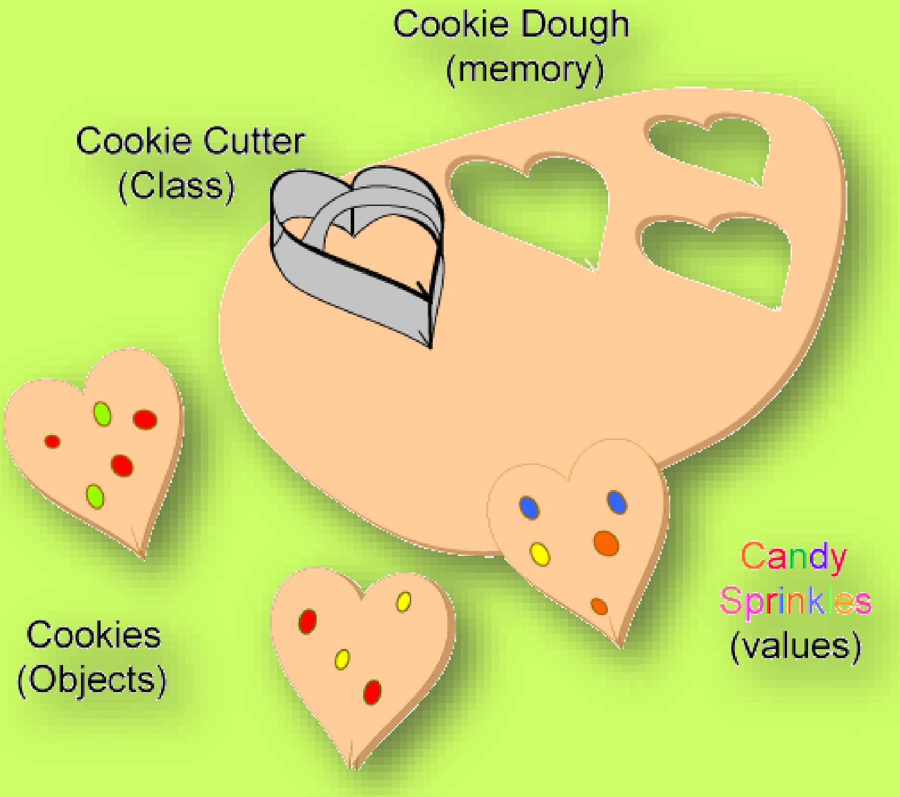
Creating an object is called **instantiation**.

Invoking the object’s method length:

```
len = str1.length();
```



OneClass, Many Objects



Cookies are objects in the real world, but Cookie Cutters are objects too. Do you think that a Java class has an object-like nature?

Yes. And a class has characteristics that are not shared with objects

STATIC

(no matter how many objects have been made, there is only one of these)

Example

Constructors

The **new** operator is used with a **constructor** **to create an object**

```
String str1 = new String("Random Jottings");
```

The **constructor** **`String`** is part of the definition for the class **`String`**:

- Is often used with **values** stored in the created object
- There are usually several **different constructors** in a class
  - Each constructor has **different parameters**
- All constructors create the **same type** of object

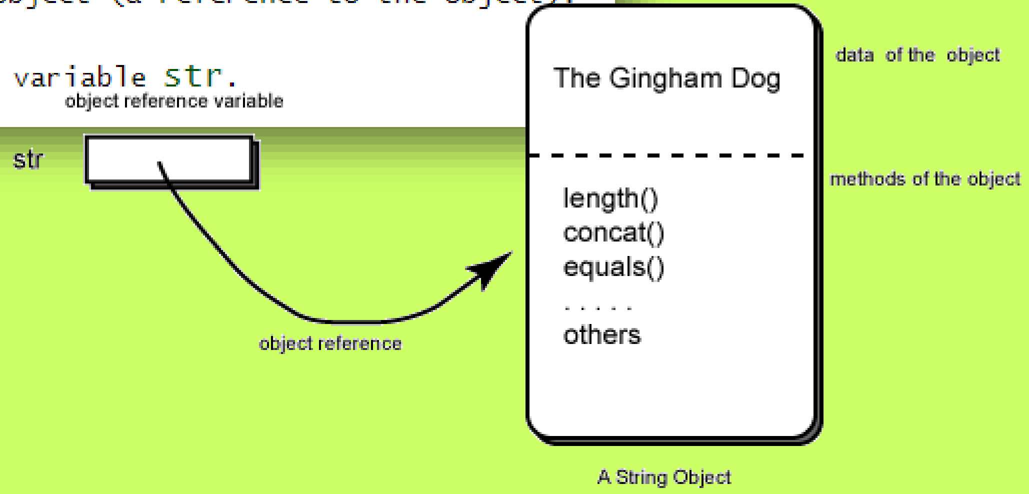
Could a constructor be used a second time to change the values of an object it created?

→ ***No***. *A constructor is used once per object. Once an object has been created the constructor is finished.*

Object Creation Steps

```
String str;    // place to hold an object reference

str =       new String( "The Gingham Dog" );
--+--       ----------------+--------------
  |                         |
  |
  |                         1.  An object is created using the constructor.
  |                             The Java system keeps track of
  |                             how to find the object (a reference to the object).

  2. A reference to the object is stored in the variable str.
```

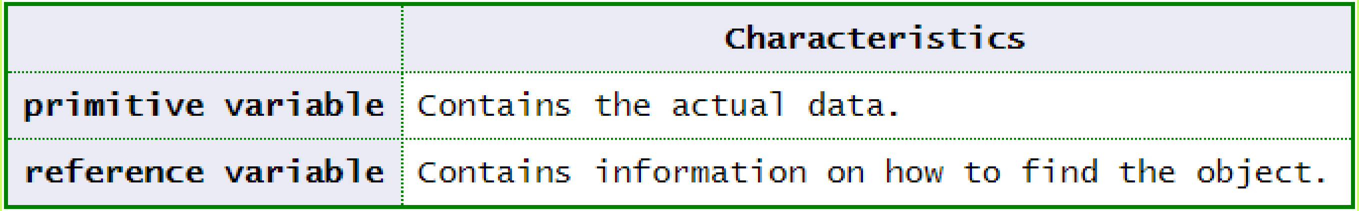
Important: A Java variable **never** contains an object



Object reference

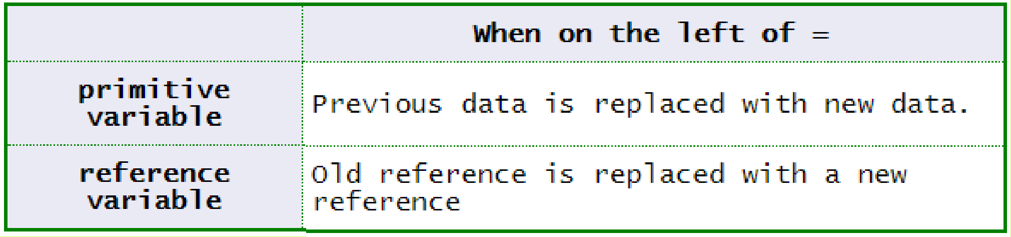
describes the location in memory of a particular object

Remembering Kinds of Variables



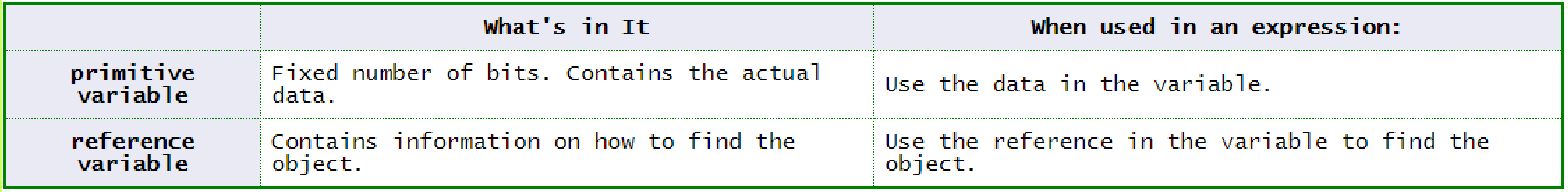
	Characteristics
primitive variable	Contains the actual data.
reference variable	Contains information on how to find the object.

Assignment



	When on the left of =
primitive variable	Previous data is replaced with new data.
reference variable	Old reference is replaced with a new reference

Usage:



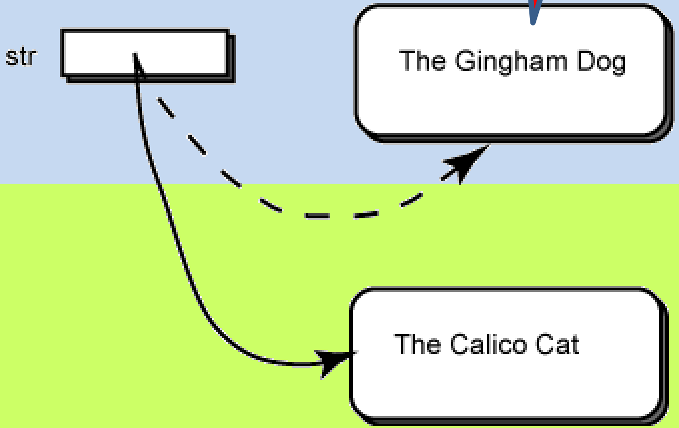
	What's in It	When used in an expression:
primitive variable	Fixed number of bits. Contains the actual data.	Use the data in the variable.
reference variable	Contains information on how to find the object.	Use the reference in the variable to find the object.

Object Creation Steps

Look at the following code:

```
public class EgString3
{
    public static void main ( String[] args )
    {
        String str; str = new String("The Gingham Dog");
        System.out.println(str); str = new String("The Calico Cat");
        System.out.println(str);
    }
}
```

Garbage!!

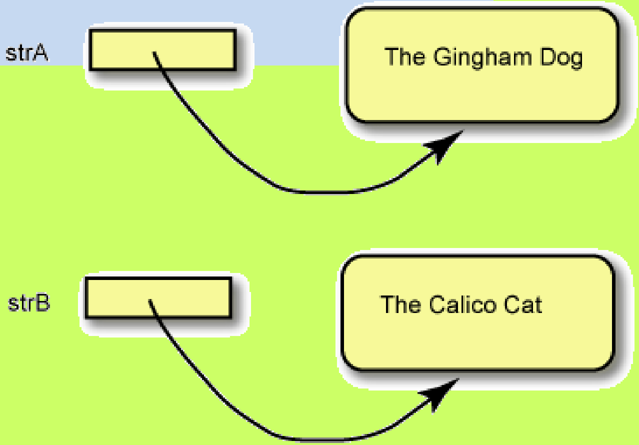


How many objects were created by the program?
How many reference variables does the program contain?

Equality of References

Look at the following code:

```
String strA; // reference to the first object
String strB; // reference to the second object

strA = new String( "The Gingham Dog" ); // create the first object and save its reference
System.out.println( strA );

strB = new String( "The Calico Cat" ); // create the second object and save its reference
System.out.println( strB );

if ( strA == strB ) System.out.println( "This will not print." );
```



The **==** operator *does **NOT** look at objects!* It only looks at references.

Another Example

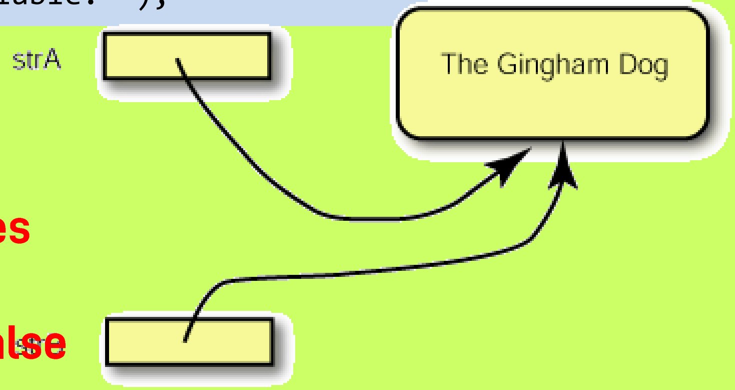
Look at the following code:

```
String strA; // reference to the first object
String strB; // reference to the second object

strA = new String( "The Gingham Dog" ); // create the only object
System.out.println( strA );

strB = strA; // Copy the reference into strB

System.out.println( strB );

if ( strA == strB ) System.out.println( “Same info in each variable." );
```

The = assignment *does **NOT** make a copy of the object!*

? Could ***two*** *different objects* contain **equivalent** data? **Yes**

In this case: What would `(strA == strB)` return? **False**



Tricky Question

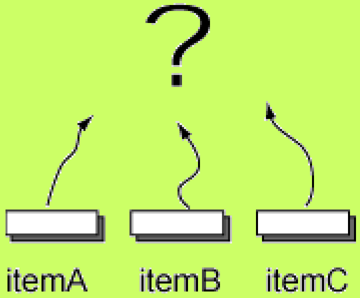
Imagine that there are three reference variables: `itemA, itemB, itemC`. And say that:

`itemA == itemB` returns **`true`**

and that

`itemB == itemC` returns **`true`**.

How many objects are there?



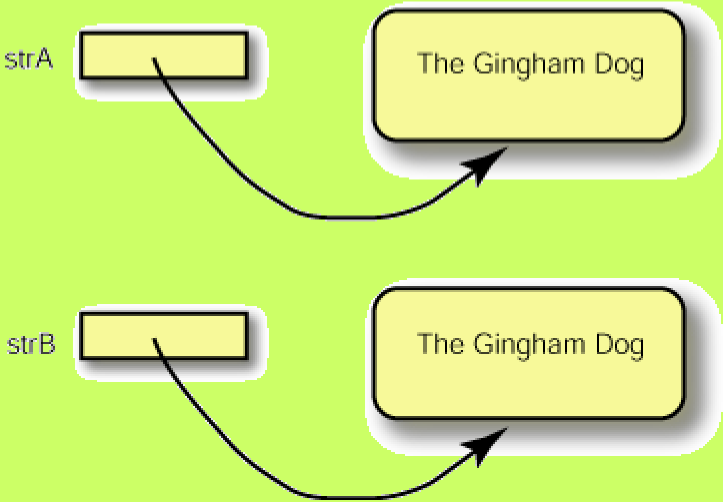
Solution:

Just one object!
(and three reference variables, each referring to it.)

? How can we detect if two objects are equivalent? ▶

The equals() method

```
String strA; // first object
String strB; // second object

strA = new String( "The Gingham Dog" );
strB = new String( "The Gingham Dog" );

// check for equivalence using strA's method
if ( strA.equals( strB ) )
     System.out.println( "This WILL print.");

// check for equivalence using strB's method
if ( strB.equals( strA ) )
    System.out.println( "This WILL print, also.");

// check for identity
if ( strA  == strB ) ç
    System.out.println( "This will NOT print.");
```



Strings that are == are always equal()

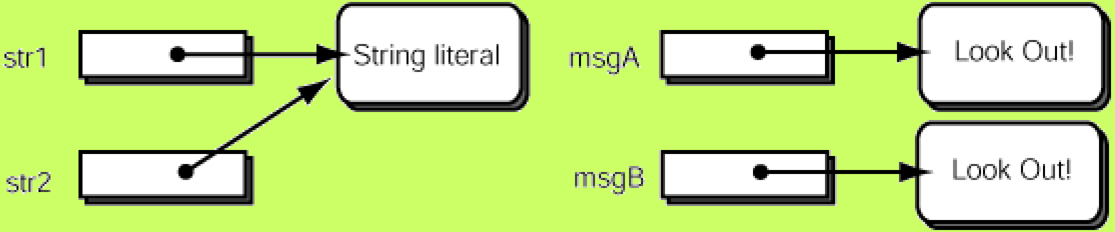
String Literals

Inspect the following code. How many objects are there? 2

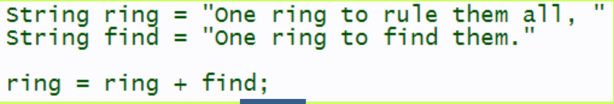
```
String msgA = new String("Look Out!");
String msgB = new String("Look Out!");
```

Inspect the following code. How many objects are there? 1

```
String str1 = "String Literal";
String str2 = "String Literal";
```



String objects are **immutable**. This means that after construction, a String object cannot be altered.



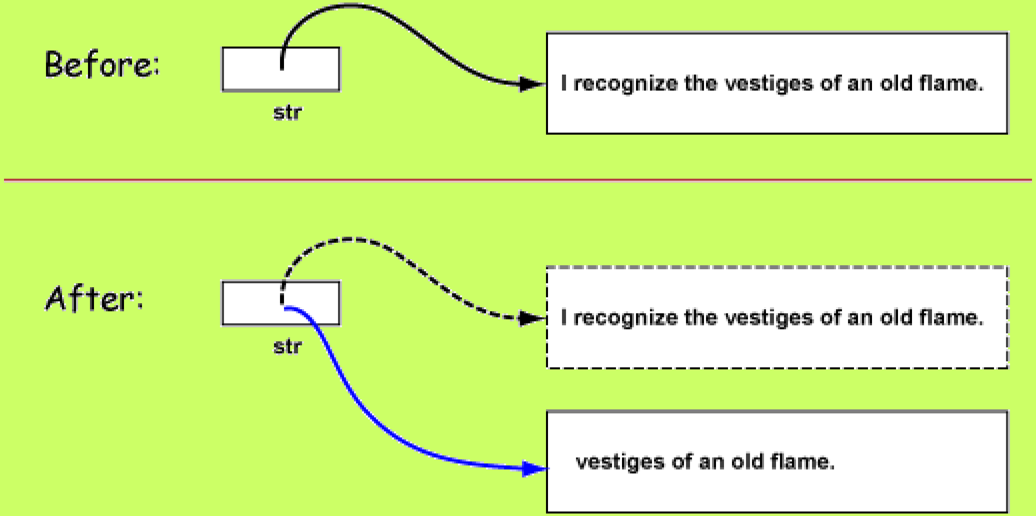
```
String ring = "One ring to rule them all, "
String find = "One ring to find them."

ring = ring + find;
```

The reference variable ring is changed in the third statement to refer to a different String than originally. (Its original String becomes garbage, which is collected by the garbage collector.)

“Changing” a String

? What can we do to “change a String”?

1. Compute a new String object.
2. Assign the reference to the new String to a reference variable.


Before:
str
I recognize the vestiges of an old flame.
After:
str
I recognize the vestiges of an old flame.
vestiges of an old flame.

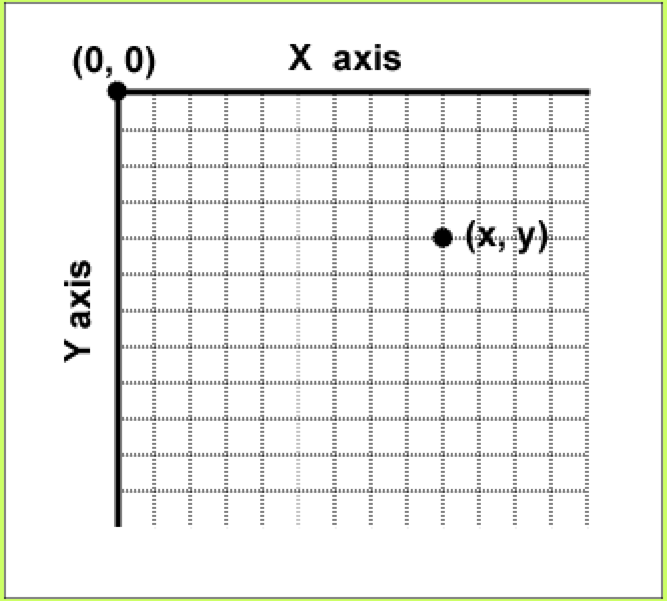
Cascaded String Operations

```
String burns = "My love is like a red, red rose.";
. . . . . .
if ( burns.toLowerCase().startsWith(" MY LOVE".trim().toLowerCase()))
    System.out.println( "Both start with the same letters." );
else
    System.out.println( "Prefix fails." );
```

What is printed?

```
Both start with the same letters.
```

The Class Point



Q: What two variables will a Point object have?
A: A pair of numbers (x, y)

Look at the constructor, parameters and methods here

Q: What is the difference between the constructors?
A: They require different parameters

To use the point class, we must import:

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

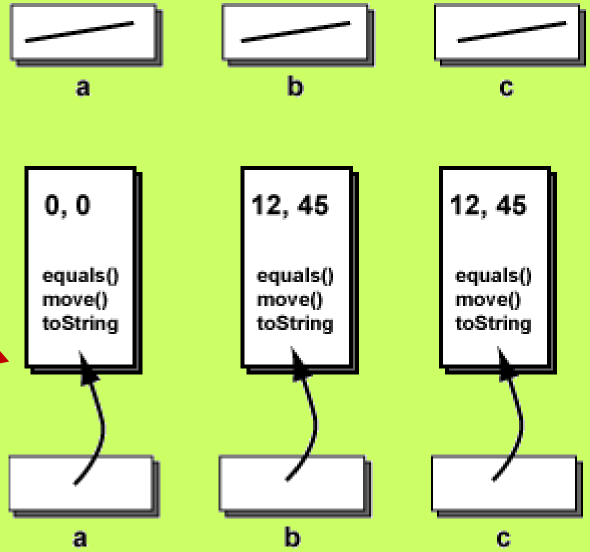
Instantiating Point objects

```
import java.awt.*;

class PointEg1 {

  public static void main ( String arg[] ) {
    Point a, b, c;
    a = new Point();
    b = new Point( 12, 45 );
    c = new Point( b );
  }
}
```

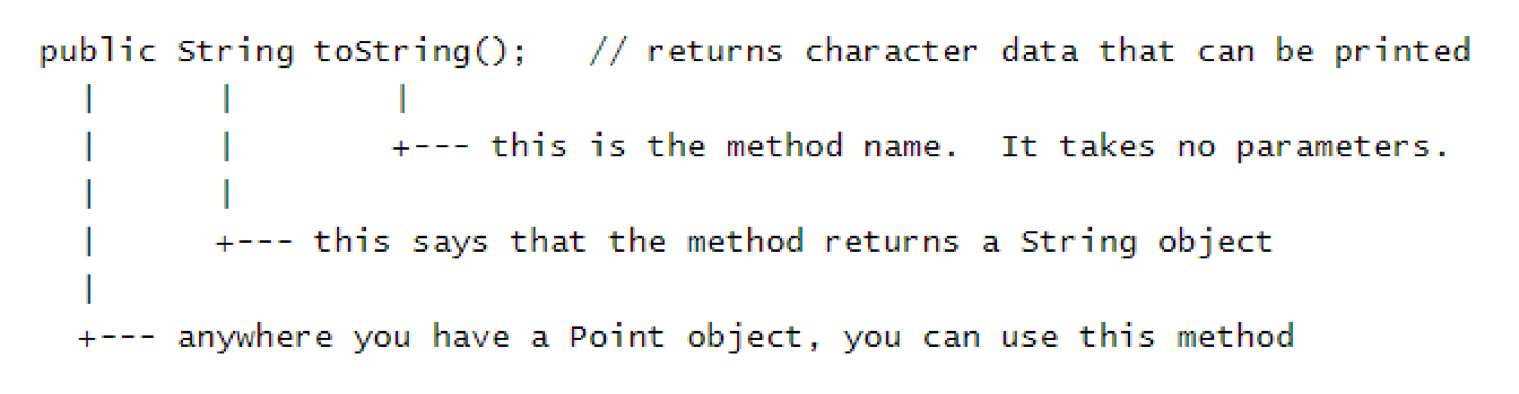
Number of:

Reference variables? **3**

Objects? **0**



The toString() Method



```
public String toString();   // returns character data that can be printed
  |      |        |
  |      |        +--- this is the method name.  It takes no parameters.
  |      |
  |      +--- this says that the method returns a String object
  |
  +--- anywhere you have a Point object, you can use this method
```

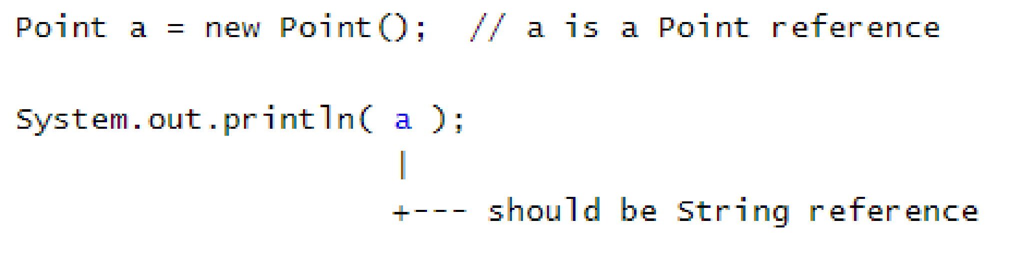
All objects have their own toString() method

```
Point a;
a = new Point();
String strA = a.toString();
System.out.println( strA );
```

The Point object has not been altered: it still exists and is referred to by a.

```
java.awt.Point[x=0,y=0]
```

Automatic toString() Call



```
Point a = new Point();  // a is a Point reference

System.out.println( a );
                    |
                    +--- should be String reference
```

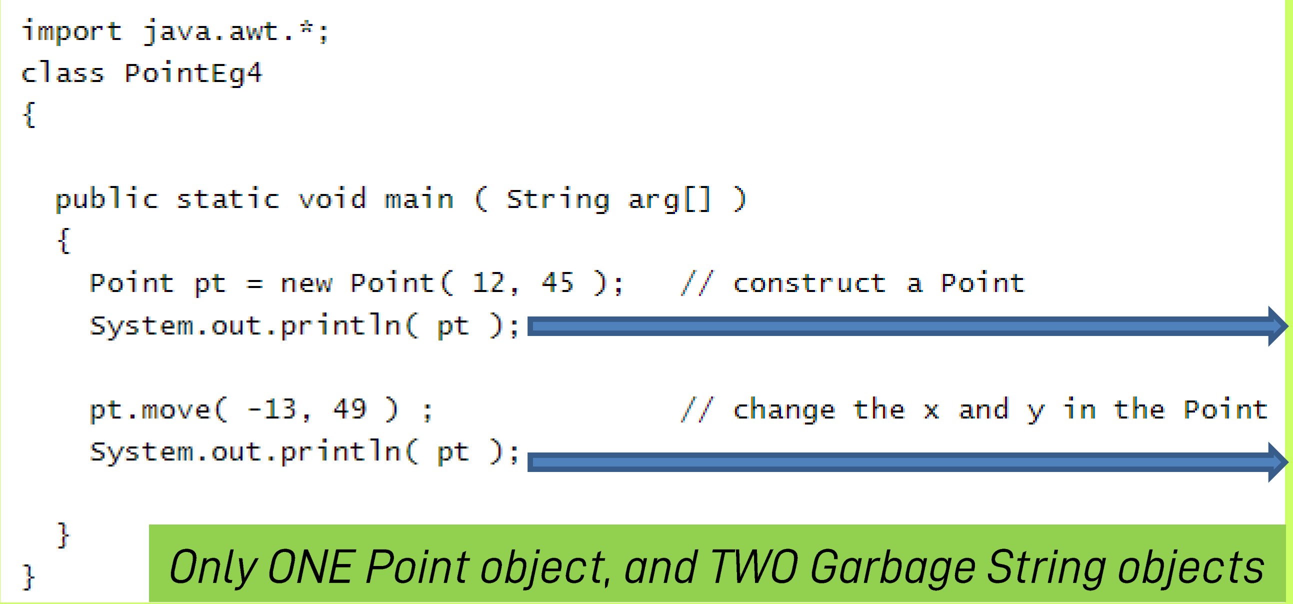
When a parameter should be a String reference, but is a reference to another type of object, Java calls the object's toString() method to create a String and then uses the resulting String reference.

```
Point a;
a = new Point();
System.out.println( a );
```

```
java.awt.Point[x=0,y=0]
```

There are 2 objects, but the String one is ***GARBAGE***

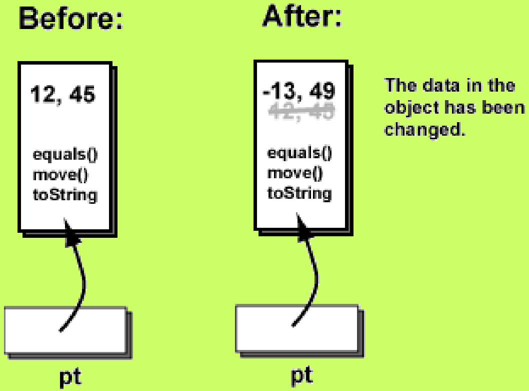
Changing Data Inside a Point



```
import java.awt.*;
class PointEg4
{

  public static void main ( String arg[] )
  {
    Point pt = new Point( 12, 45 );   // construct a Point
    System.out.println( pt );

    pt.move( -13, 49 ) ;              // change the x and y in the Point
    System.out.println( pt );

  }
}
```

```
java.awt.Point[x=12,y=45]
java.awt.Point[x=-13,y=49]
```

Only ONE Point object, and TWO Garbage String objects

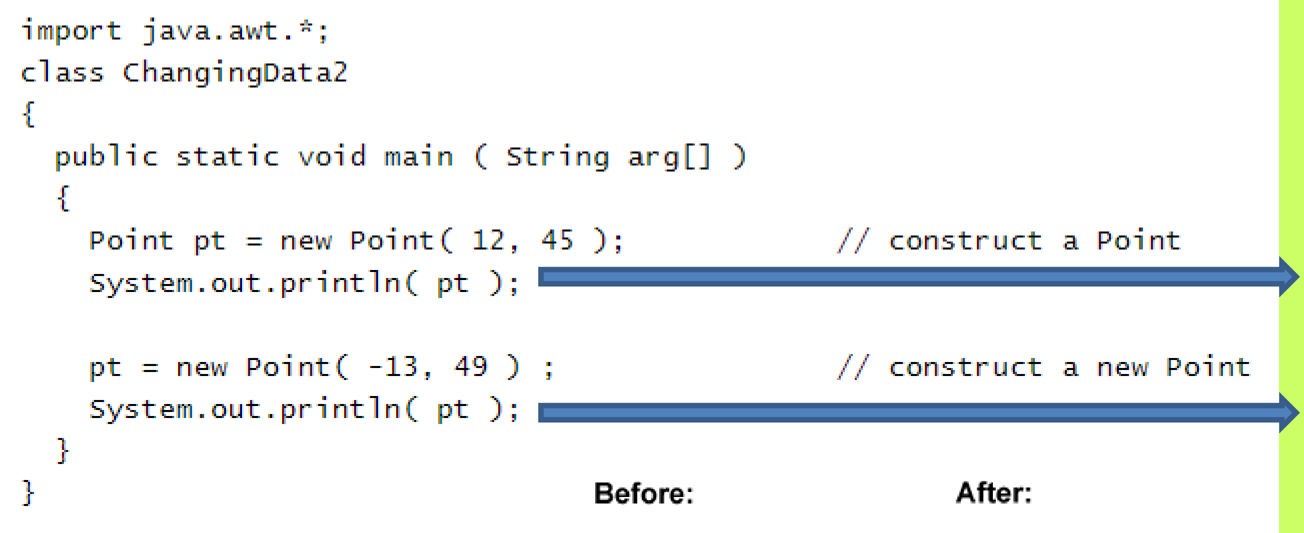


The data in the object has been changed.

? Can a ***constructor*** be used to change the data inside an object?

No. Constructors always create new objects.

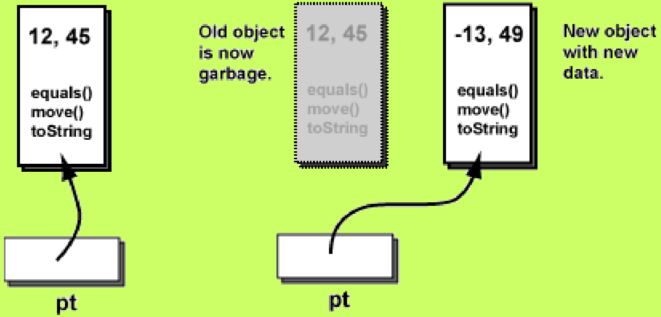
Dangerously Similar Program



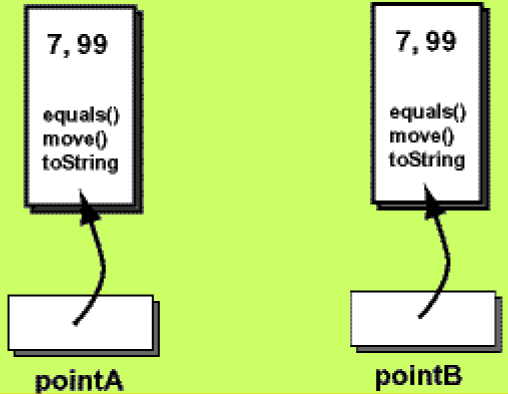
```
import java.awt.*;
class ChangingData2
{
  public static void main ( String arg[] )
  {
    Point pt = new Point( 12, 45 );              // construct a Point
    System.out.println( pt );

    pt = new Point( -13, 49 ) ;                  // construct a new Point
    System.out.println( pt );
  }
}
```

java.awt.Point[x=12,y=45]

java.awt.Point[x=-13,y=49]



Last Point Example



Would the == operator do the same as the equals() method?

No. *The == operator tests if two reference variables refer to the same object. (**Alias-detector**)*

pointA == pointB ?

No

pointA.equals(pointB)?

Yes

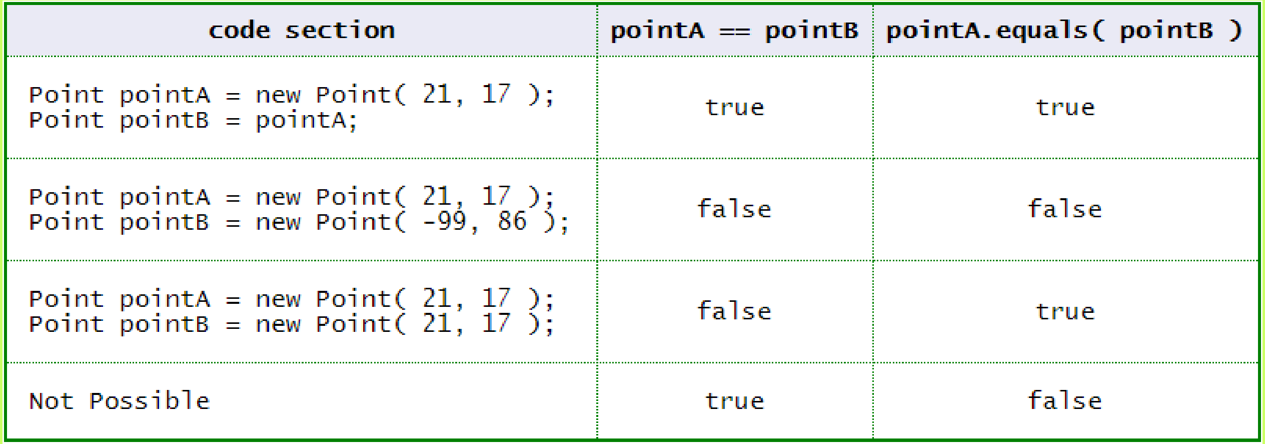
Practice

More Practice

?

In the case that `pointA.equals(pointB)` is `true`:

- The two variables refer to the same object?
- There are two objects with equivalent data?



code section	pointA == pointB	pointA.equals(pointB)

`Point pointA = new Point( 21, 17 );`
`Point pointB = pointA;`
	true	true

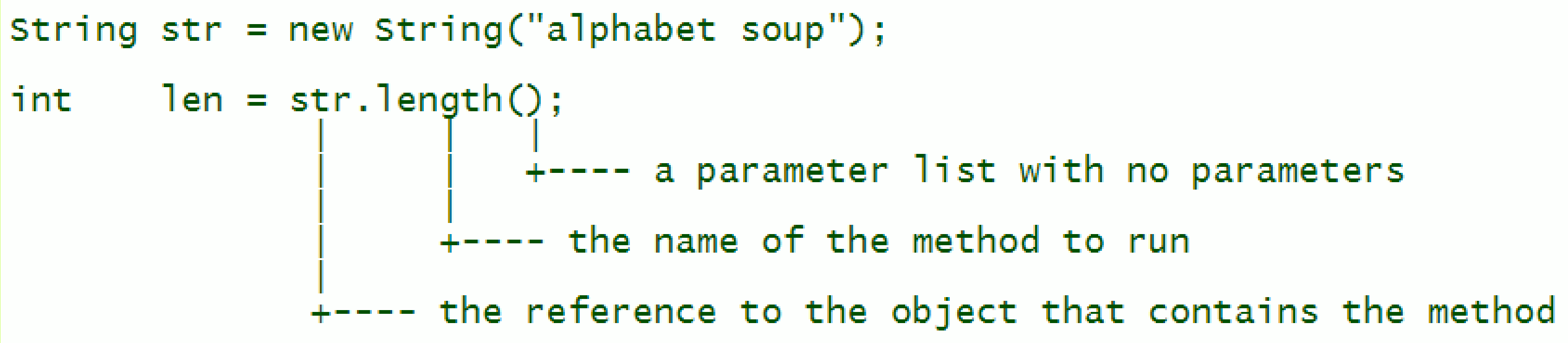
`Point pointA = new Point( 21, 17 );`
`Point pointB = new Point( -99, 86 );`
	false	false

`Point pointA = new Point( 21, 17 );`
`Point pointB = new Point( 21, 17 );`
	false	true

Not Possible	true	false

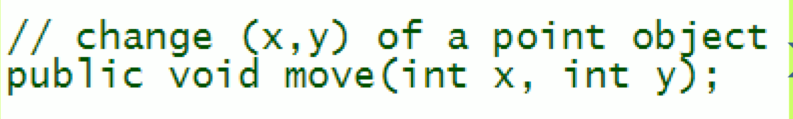
Method Parameters

There can be methods which **does not** require any parameters:

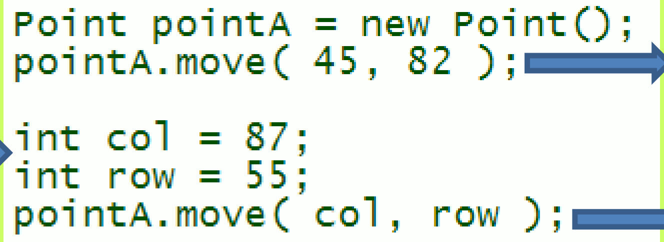


```
String str = new String("alphabet soup");

int    len = str.length();
                |      |   |
                |      |   +---- a parameter list with no parameters
                |      |
                |      +---- the name of the method to run
                |
                |
                +---- the reference to the object that contains the method
```

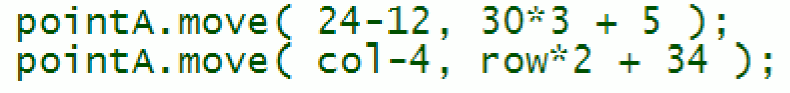
Others, on the contrary, need information about what it is to do (**parameters**):



```
// change (x,y) of a point object
public void move(int x, int y);
```



```
Point pointA = new Point();
pointA.move( 45, 82 );

int col = 87;
int row = 55;
pointA.move( col, row );
```



```
pointA.move( 24-12, 30*3 + 5 );
pointA.move( col-4, row*2 + 34 );
```

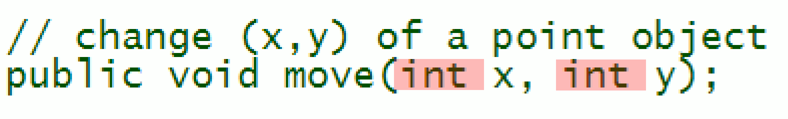
values

variables

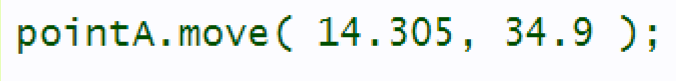
expressions

Parameter Types

Parameters must be the correct type:



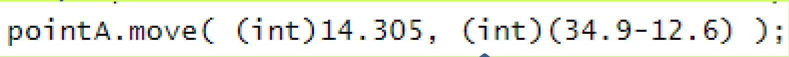
```
// change (x,y) of a point object
public void move(int x, int y);
```



```
pointA.move( 14.305, 34.9 );
```

→ **Error!** →

Type cast:
(requiredType)(expression)

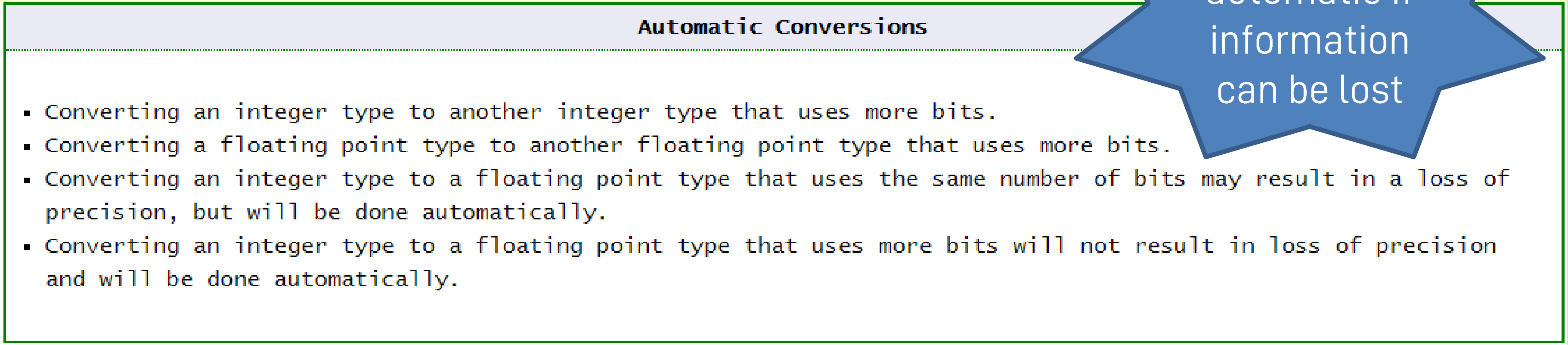


```
pointA.move( (int)14.305, (int)(34.9-12.6) );
```

Parameters can be converted:

- **Explicitly** → Type cast
- **Implicitly** → Compiler makes the conversion automatically

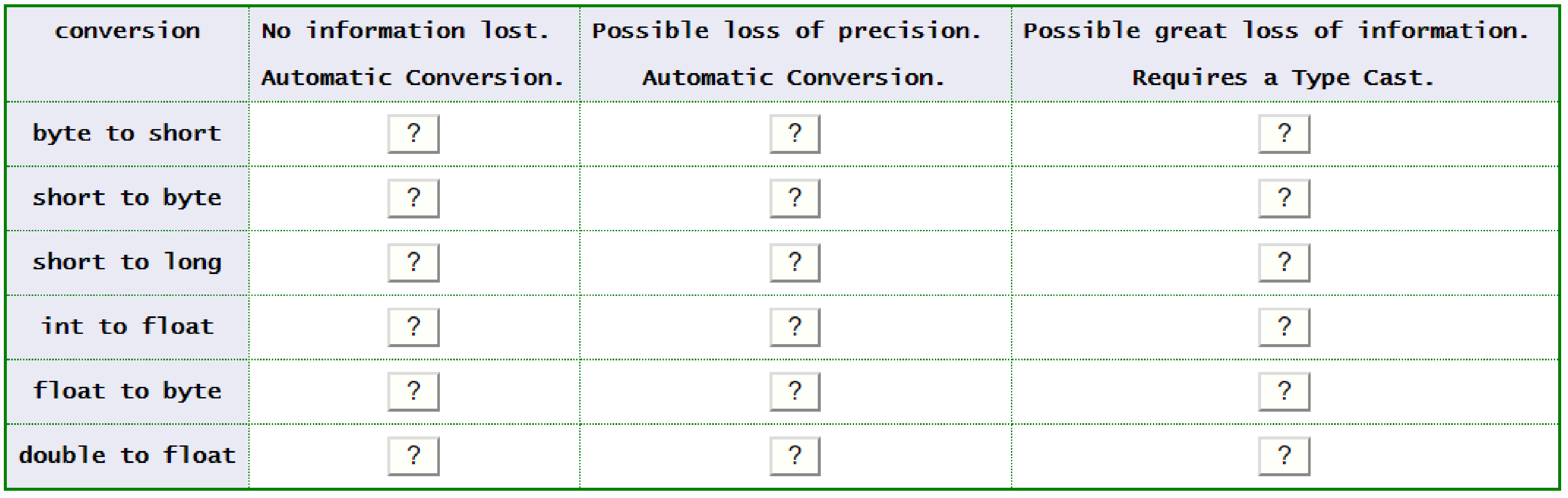
Not automatic if information can be lost



Automatic Conversions

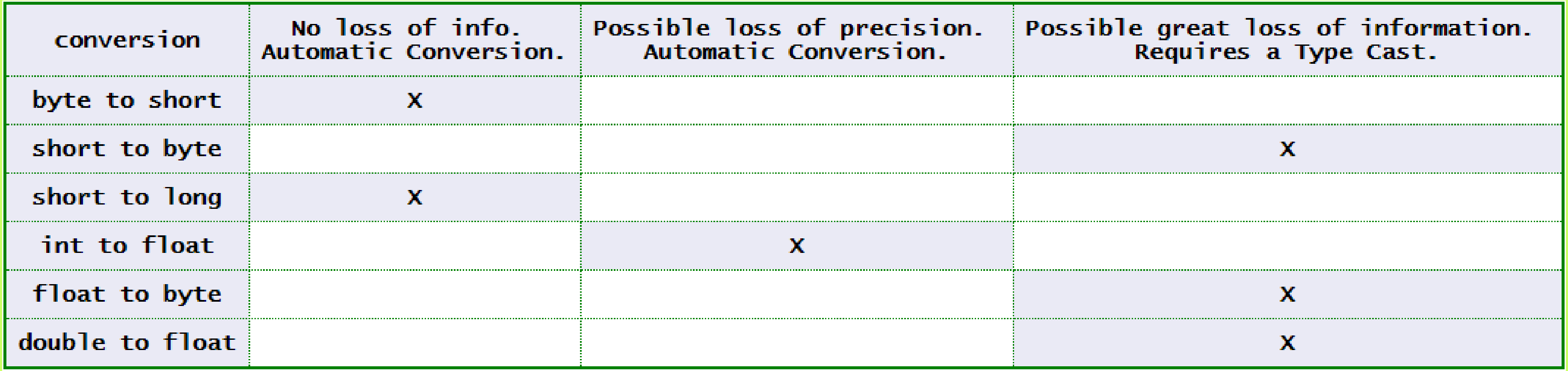
- Converting an integer type to another integer type that uses more bits.
- Converting a floating point type to another floating point type that uses more bits.
- Converting an integer type to a floating point type that uses the same number of bits may result in a loss of precision, but will be done automatically.
- Converting an integer type to a floating point type that uses more bits will not result in loss of precision and will be done automatically.

Question



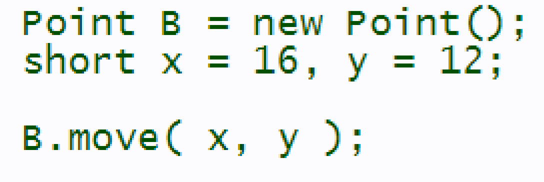
conversion	No information lost. Automatic Conversion.	Possible loss of precision. Automatic Conversion.	Possible great loss of information. Requires a Type Cast.
byte to short	?	?	?
short to byte	?	?	?
short to long	?	?	?
int to float	?	?	?
float to byte	?	?	?
double to float	?	?	?

Solution



conversion	No loss of info. Automatic Conversion.	Possible loss of precision. Automatic Conversion.	Possible great loss of information. Requires a Type Cast.
byte to short	x		
short to byte			x
short to long	x		
int to float		x	
float to byte			x
double to float			x

The parameters x and y contain short values which are converted into int values for the method. Are the contents of x and y altered by this conversion?



```
Point B = new Point();
short x = 16, y = 12;

B.move( x, y );
```

No. When a *primitive variable* is used as a parameter for any method at all, the method will not change the value in the variable.

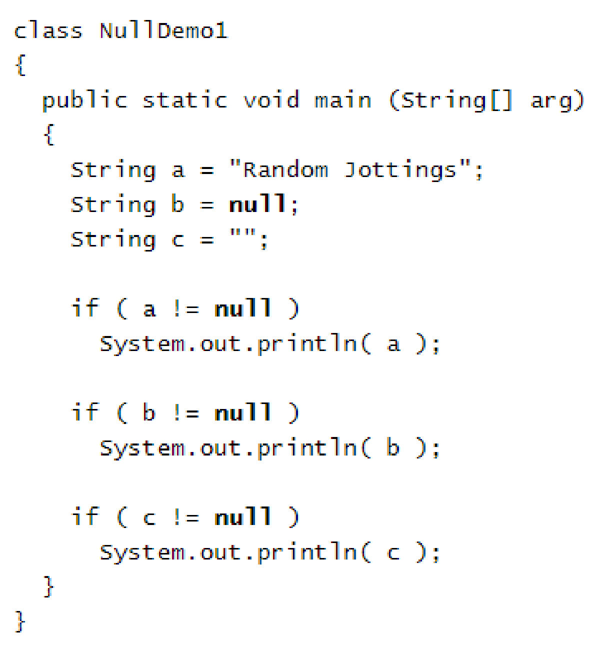
The null Value

null is a special value that means "**no object**."

Variables are often set to null when they are declared.

null can be assigned to any reference variable.

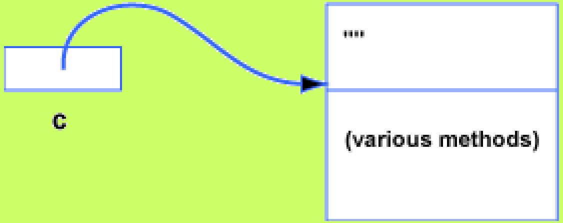
```
String a = null;
Point  b = null;
```



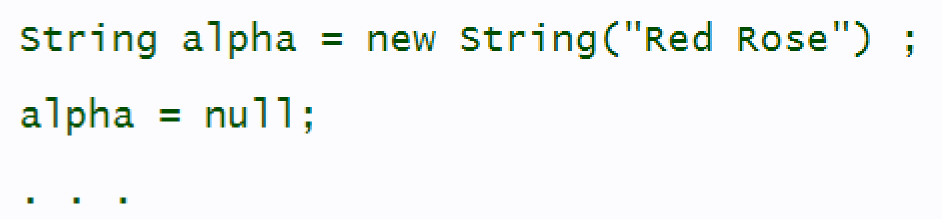
```
class NullDemo1
{
  public static void main (String[] arg)
  {
    String a = "Random Jottings";
    String b = null;
    String c = "";

    if ( a != null )
      System.out.println( a );

    if ( b != null )
      System.out.println( b );

    if ( c != null )
      System.out.println( c );
  }
}
```

? What exactly is variable c initialized to?

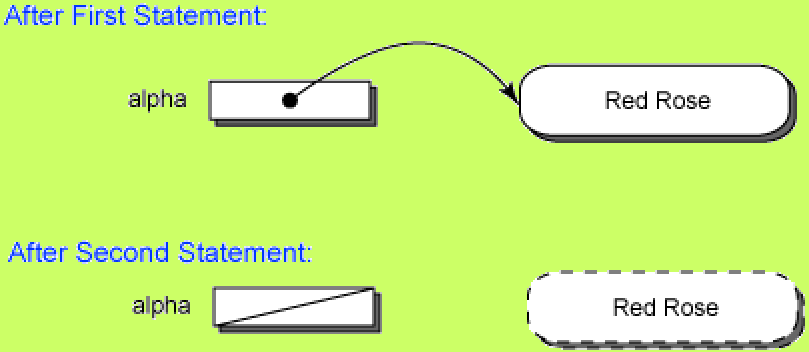
The reference variable c is initialized to a reference to a String object with **no characters (empty string)**. This is most certainly a different value than **null**.

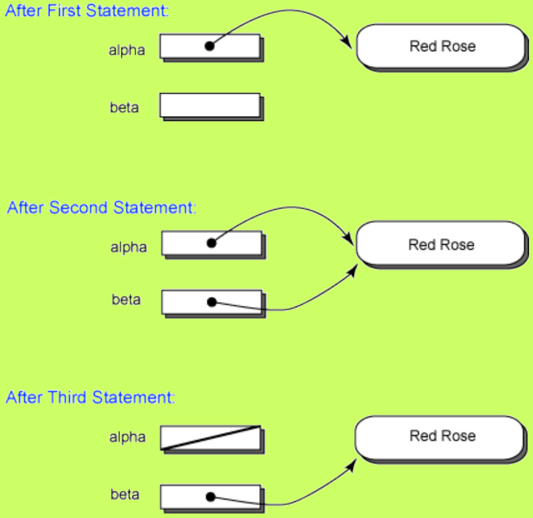


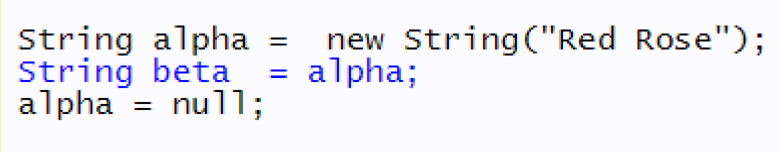
Garbage



```
String alpha = new String("Red Rose") ;

alpha = null;

. . .
```

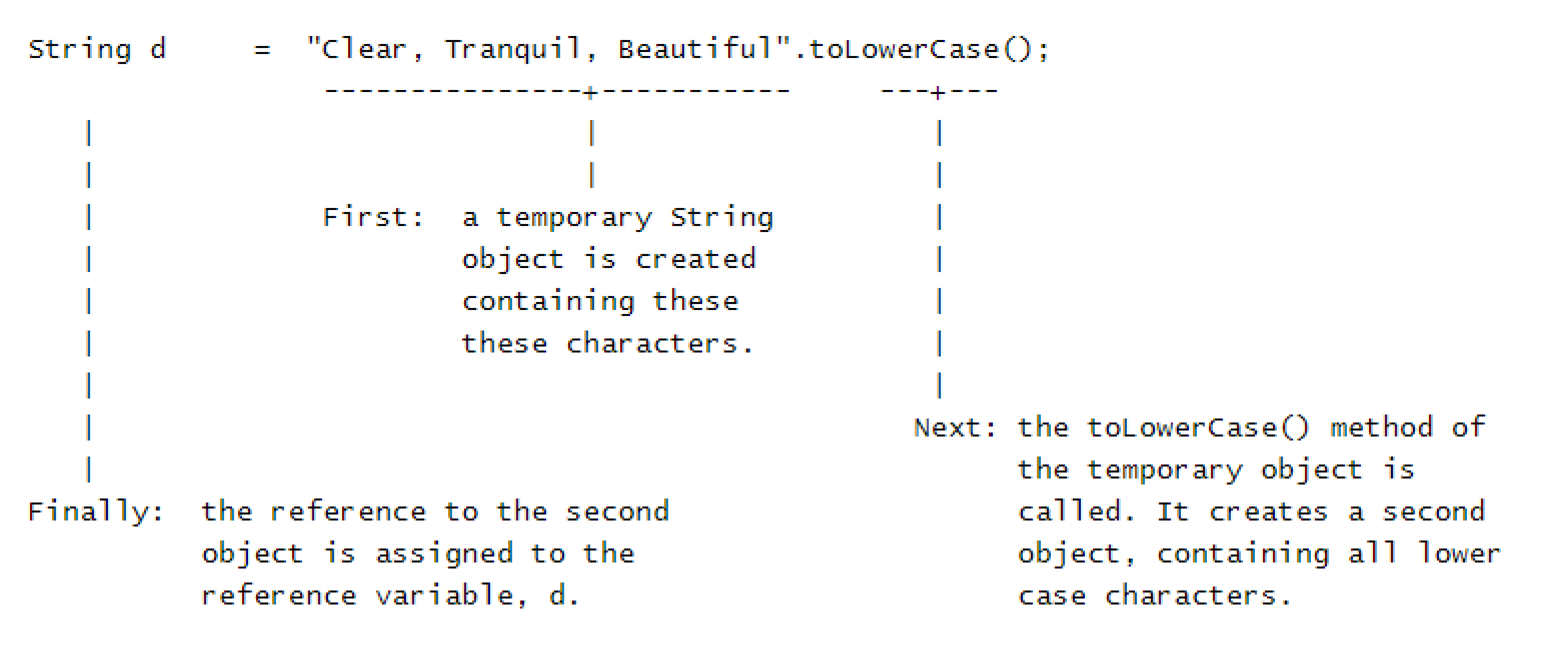






```
String alpha =  new String("Red Rose");
String beta  = alpha;
alpha = null;
```

Temporary Objects



```
String d     =  "Clear, Tranquil, Beautiful".toLowerCase();
                 ---------------+------------    ---+---
   |                            |                   |
   |                            |                   |
   |             First:  a temporary String         |
   |                     object is created          |
   |                     containing these           |
   |                     these characters.          |
   |                                                |
   |                                               Next: the toLowerCase() method of
   |                                                     the temporary object is
Finally:  the reference to the second                    called. It creates a second
          object is assigned to the                      object, containing all lower
          reference variable, d.                         case characters.
```

?

What happens in this case?
`System.out.println("Dark, forlorn...".toLowerCase());`

A String is constructed. Then a second String is constructed (by the toLowerCase() method). A reference to the second String is used a parameter for println(). Both String objects are temporary. After println() finishes, both Strings are garbage.

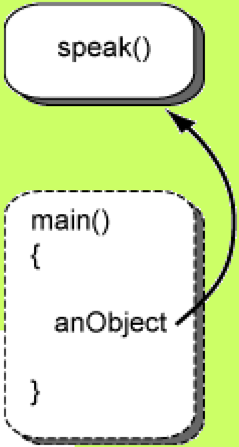
Defining Your Own Classes

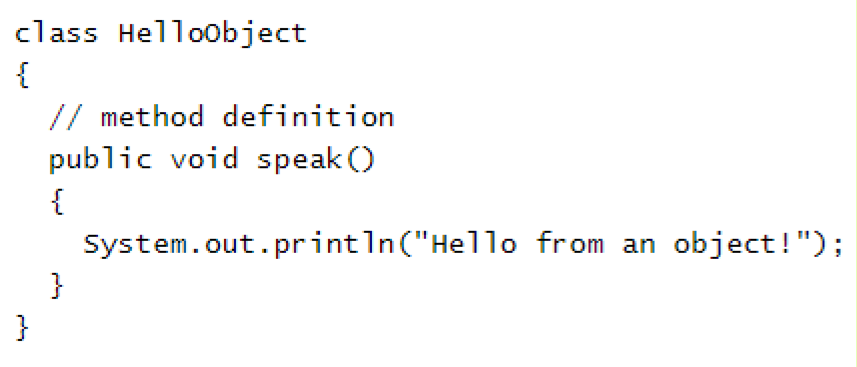
Class definition looks like this:

modifiers[†] class *ClassName*

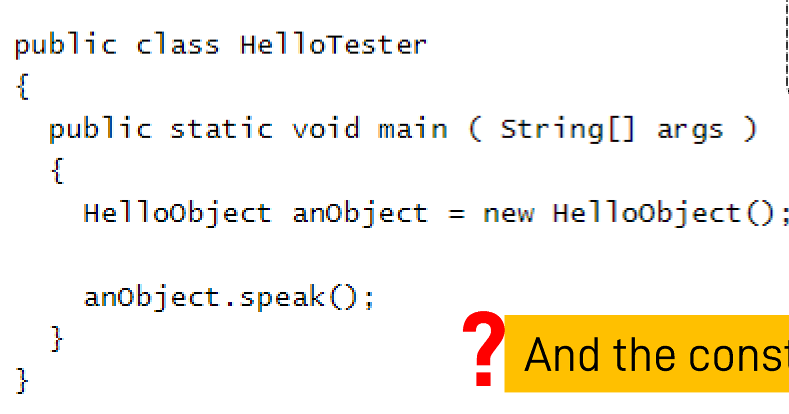
```
 {
    // Description of the instance variables
    // Description of the constructors
    // Description of the methods
}
```

[†] *For now, replace modifiers with public in the class that contains main and don't include it in other classes in the same file.*





```
class HelloObject
{
  // method definition
  public void speak()
  {
    System.out.println("Hello from an object!");
  }
}
```



```
public class HelloTester
{
  public static void main ( String[] args )
  {
    HelloObject anObject = new HelloObject();

    anObject.speak();
  }
}
```

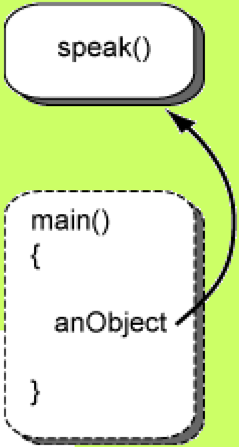
? And the constructor?

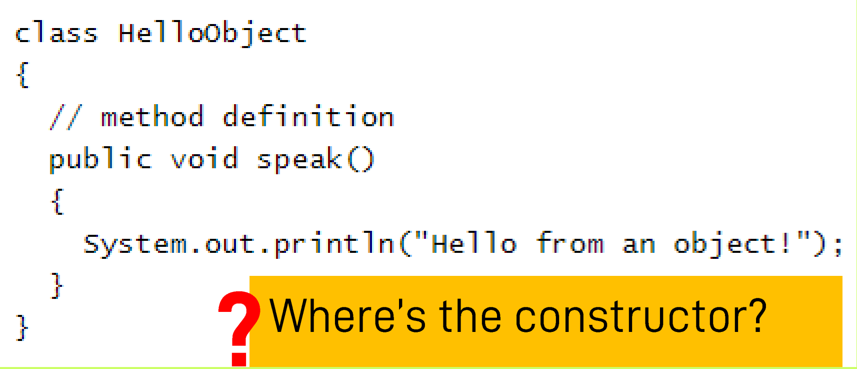
Defining Your Own Classes

Class definition looks like this:

[†] *For now, replace modifiers with public in the class that contains main and don't include it in other classes in the same file.*

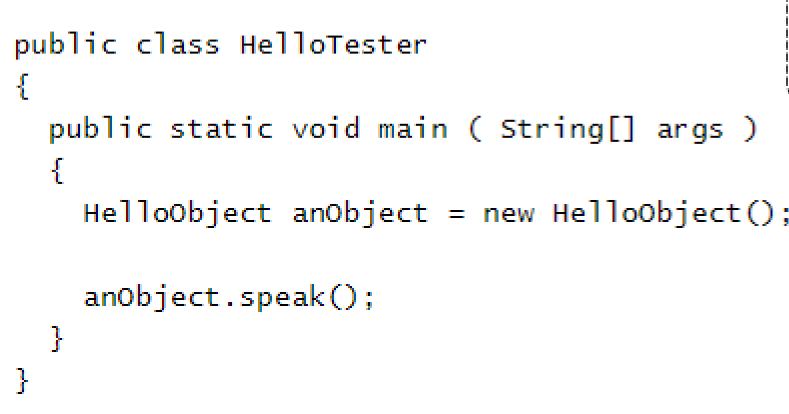
```
modifiers† class ClassName
 {
    // Description of the instance variables
    // Description of the constructors
    // Description of the methods
}
```





```
class HelloObject
{
  // method definition
  public void speak()
  {
    System.out.println("Hello from an object!");
  }
}
```

? Where's the constructor?



```
public class HelloTester
{
  public static void main ( String[] args )
  {
    HelloObject anObject = new HelloObject();

    anObject.speak();
  }
}
```

Default Constructor

To construct an object, there **must** be a constructor.

If a class definition does not include a constructor a **default constructor** is automatically supplied by the Java compiler:

- It works with the Java virtual machine to **find** main memory for the object.
- Sets up that **memory** as an object.
- Puts in the **variables** and **methods** specified in the class definition.
- Returns an **object reference** to your program.

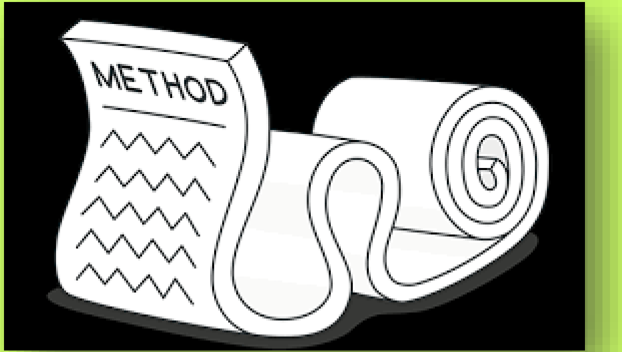
If you need to **initialize variables** → **Create a constructor**

Syntax Rule: If you define one or more constructors for a class, then those are the only constructors that the class has. The default constructor is supplied automatically only if you define no constructors.

Method Definition

Method definitions looks like this:

```
modifiers returnType methodName( parameterList )
{
    // Java statements
    return returnValue;
}
```



If it does not return anything:

```
public void speak()
{
    System.out.println("Hello from an object!");
}
```

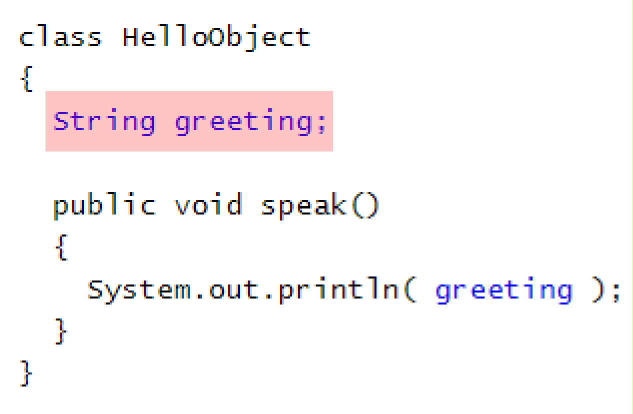
Instance Variables and Constructors

The **instance variables** are the variables that each object has as part of itself.

Usually each instance variable is marked `private`.

Object's methods use that object's instance variables.

They are usually initialized using a **constructor**.



```
class HelloObject
{
  String greeting;

  public void speak()
  {
    System.out.println( greeting );
  }
}
```

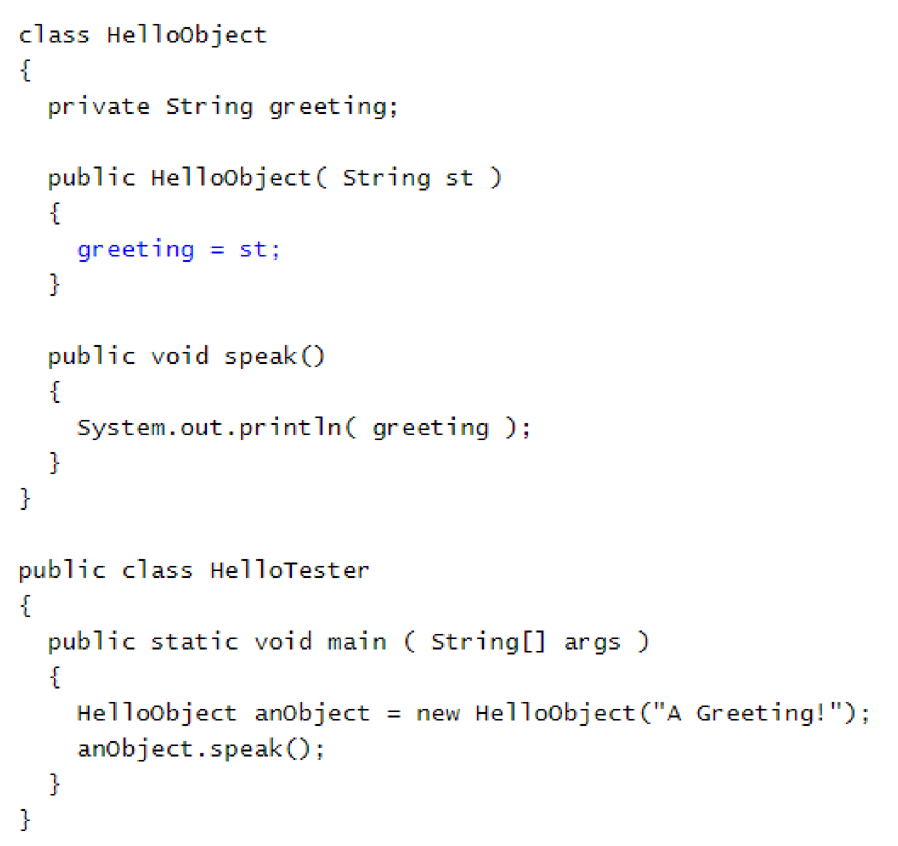
```
public className( parameterList )
{
    Statements involving the instance variables of the class and
the parameters in the parameterList.
}
```

Returns a **reference** to the object it constructs

Has the same **name** as the **class**

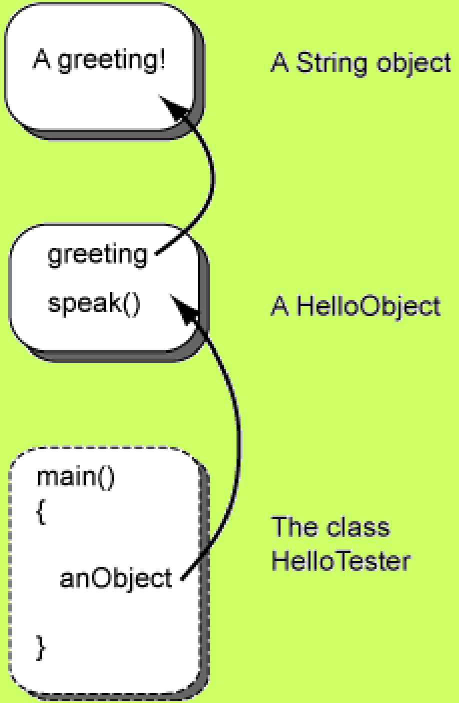
It's OK to have an **empty** parameter list.

Example of Completed Constructor



```
class HelloObject
{
  private String greeting;

  public HelloObject( String st )
  {
    greeting = st;
  }

  public void speak()
  {
    System.out.println( greeting );
  }
}

public class HelloTester
{
  public static void main ( String[] args )
  {
    HelloObject anObject = new HelloObject("A Greeting!");
    anObject.speak();
  }
}
```

? How many objects exist just before this program stops running?

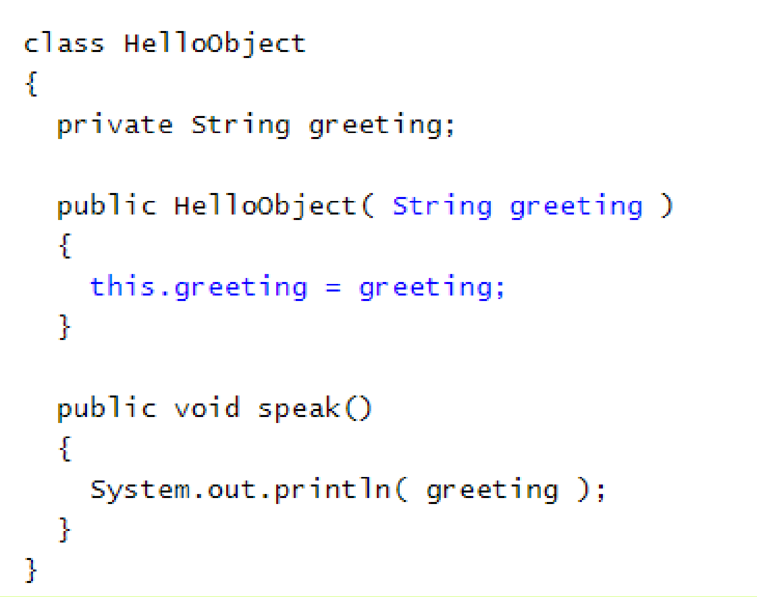


Variables and Parameters

What names may be used for the constructor parameters?

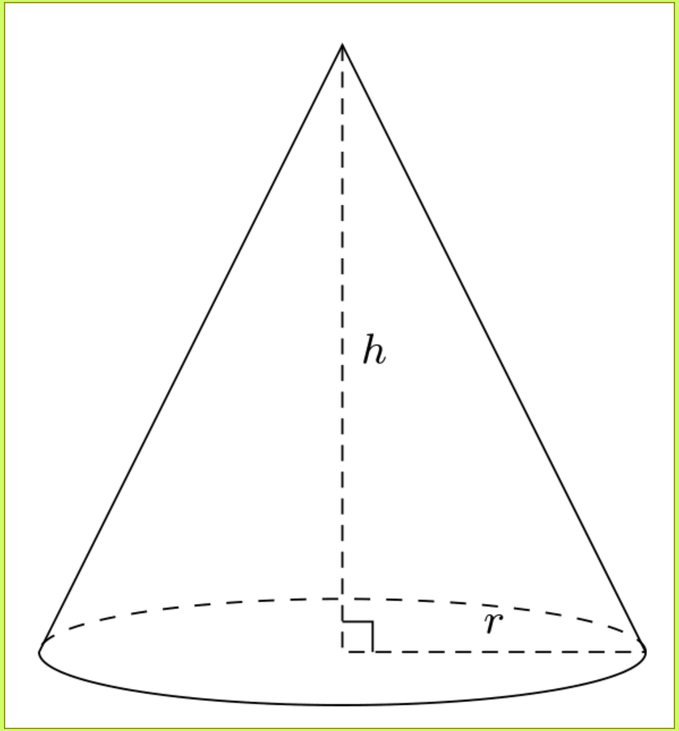
We can use the same names as can be used for instance variables

To **avoid confusion**, use the reserved word **this** to show when an identifier refers to an object's instance variable.



```
class HelloObject
{
  private String greeting;

  public HelloObject( String greeting )
  {
    this.greeting = greeting;
  }

  public void speak()
  {
    System.out.println( greeting );
  }
}
```

Designing a Class (cone)



What **variables**, **constructors** and **methods** is this class going to need?

```
private double height;
private double radius;
```

***Private variables**, how can we set them?*

```
public Cone( double radius, double height )
```

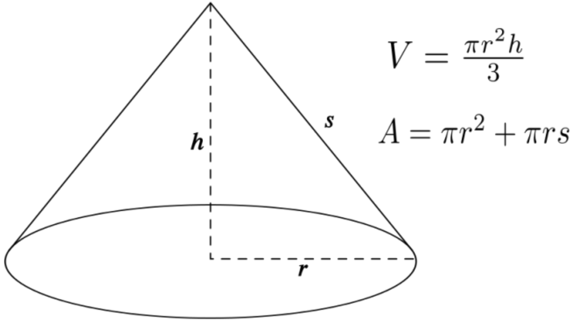
What methods could this class need?

```
public double area()
```

```
public double volume()
```

Testing a Class (cone)

```
import java.util.Scanner ;

public class TestCone
{
  public static void main( String[] args )
  {
    Scanner scan = new Scanner(System.in);

    double radius, height;

    System.out.print("Enter radius: " );
    radius = scan.nextDouble();

    System.out.print("Enter height: " );
    height = scan.nextDouble();

    Cone cone = new Cone( radius, height );

    System.out.println( "Area "  + cone.area() + " Volume: " + cone.volume() );
  }
}
```



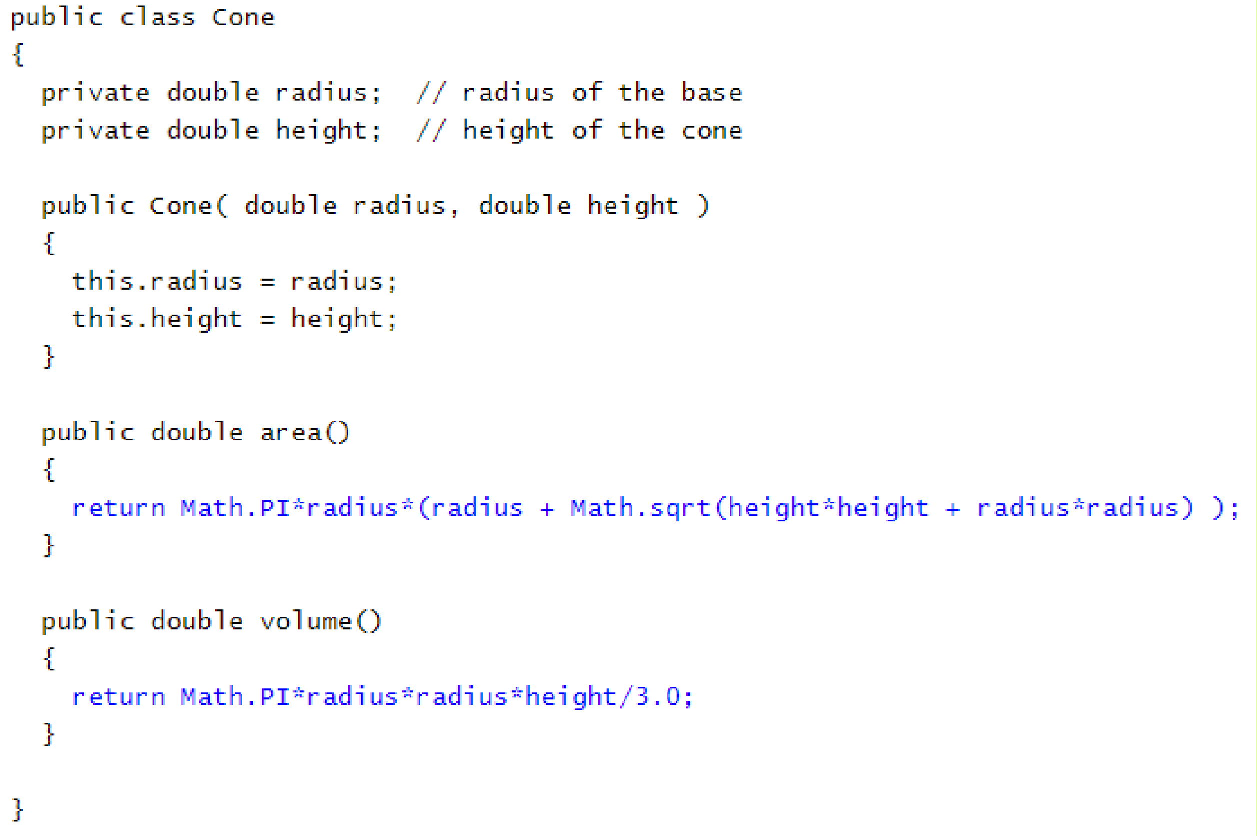
$$V = \frac{\pi r^2 h}{3}$$

$$A = \pi r^2 + \pi r s$$

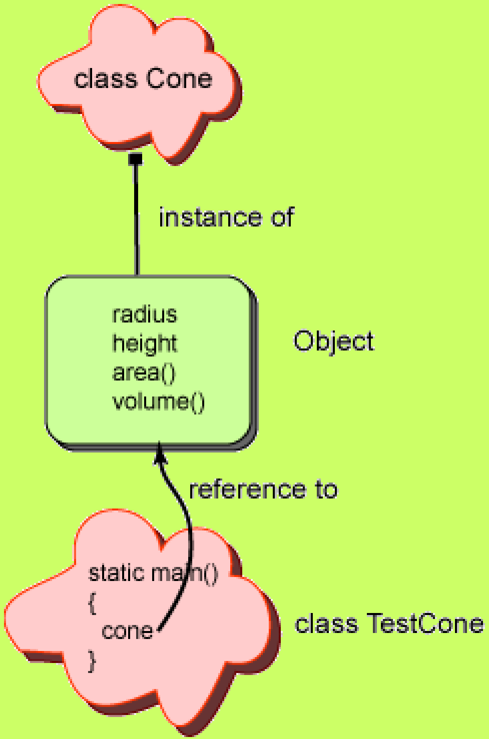
This is the program to test the cone. Write the code for the class Cone using this skeleton:

```
public class Cone
{
  // instance variables


  // constructor


  // methods

}
```

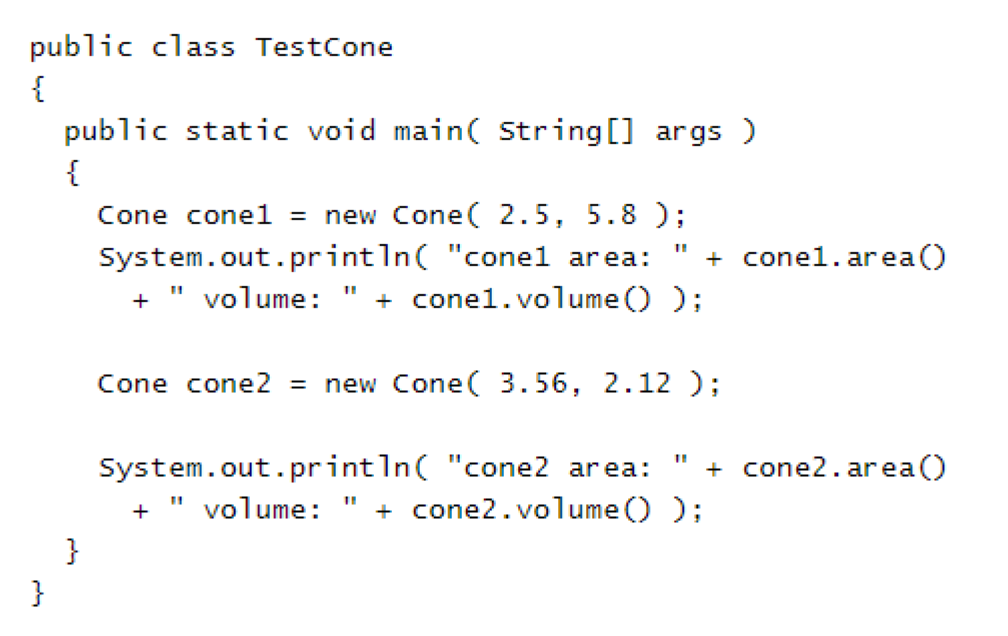
Defining a Class (cone)



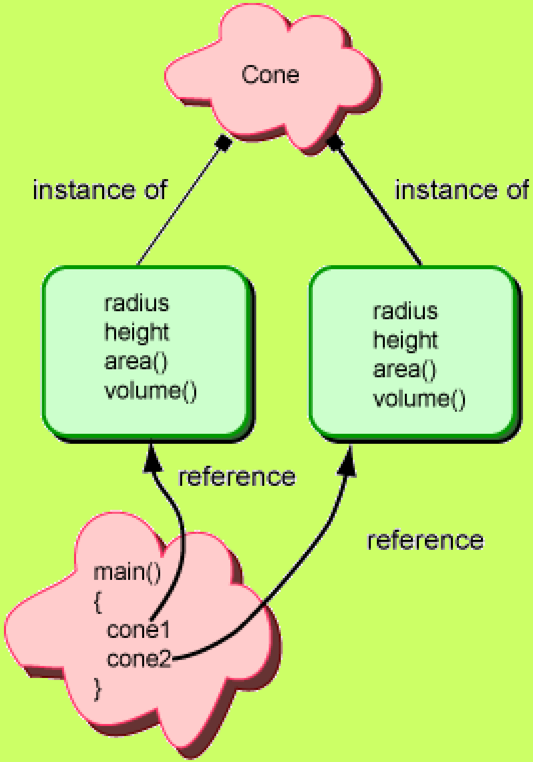
```
public class Cone
{
  private double radius;  // radius of the base
  private double height;  // height of the cone

  public Cone( double radius, double height )
  {
    this.radius = radius;
    this.height = height;
  }

  public double area()
  {
    return Math.PI*radius*(radius + Math.sqrt(height*height + radius*radius) );
  }

  public double volume()
  {
    return Math.PI*radius*radius*height/3.0;
  }

}
```



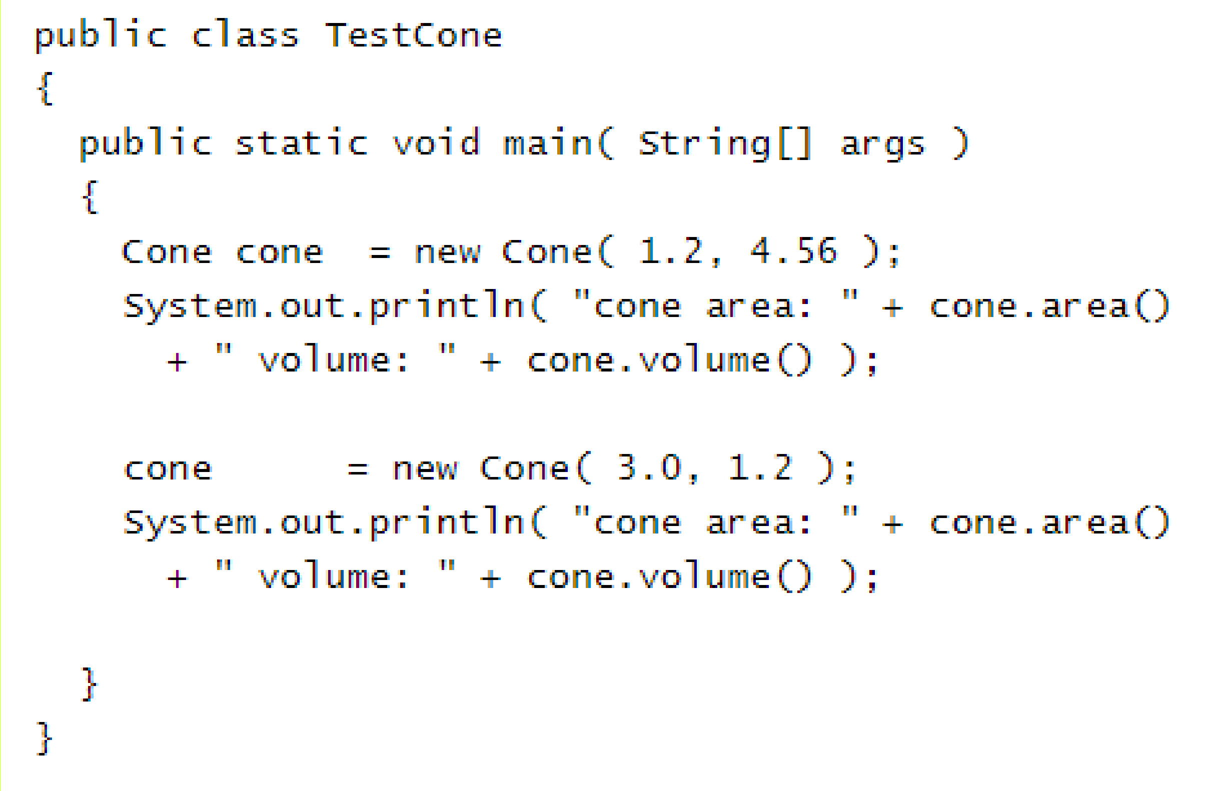
Two Cones



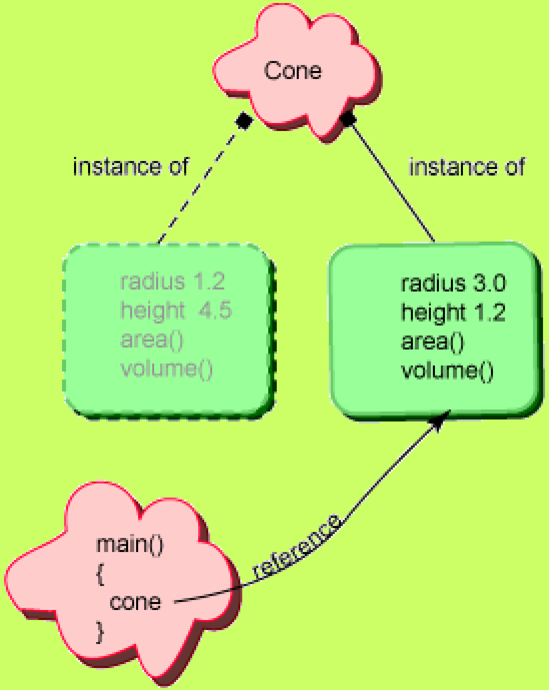
```java
public class TestCone
{
  public static void main( String[] args )
  {
    Cone cone1 = new Cone( 2.5, 5.8 );
    System.out.println( "cone1 area: " + cone1.area()
      + " volume: " + cone1.volume() );

    Cone cone2 = new Cone( 3.56, 2.12 );

    System.out.println( "cone2 area: " + cone2.area()
      + " volume: " + cone2.volume() );
  }
}
```



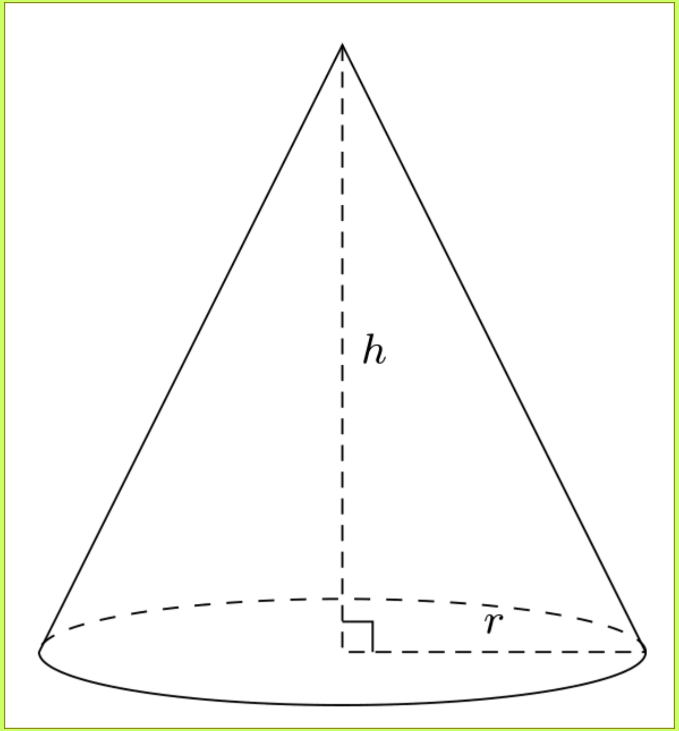
What Will Happen Here?



```
public class TestCone
{
  public static void main( String[] args )
  {
    Cone cone  = new Cone( 1.2, 4.56 );
    System.out.println( "cone area: " + cone.area()
      + " volume: " + cone.volume() );

    cone      = new Cone( 3.0, 1.2 );
    System.out.println( "cone area: " + cone.area()
      + " volume: " + cone.volume() );

  }
}
```



Designing a Class (cone)



? What **variables**, **constructors** and **methods** is this class going to need?

```
private double height;
private double radius;
```

***Private variables**, how can we set them?*

```
public Cone( double radius, double height )
```

```
public double area()
```

```
public double volume()
```

***Private variables**, how can we modify them?*

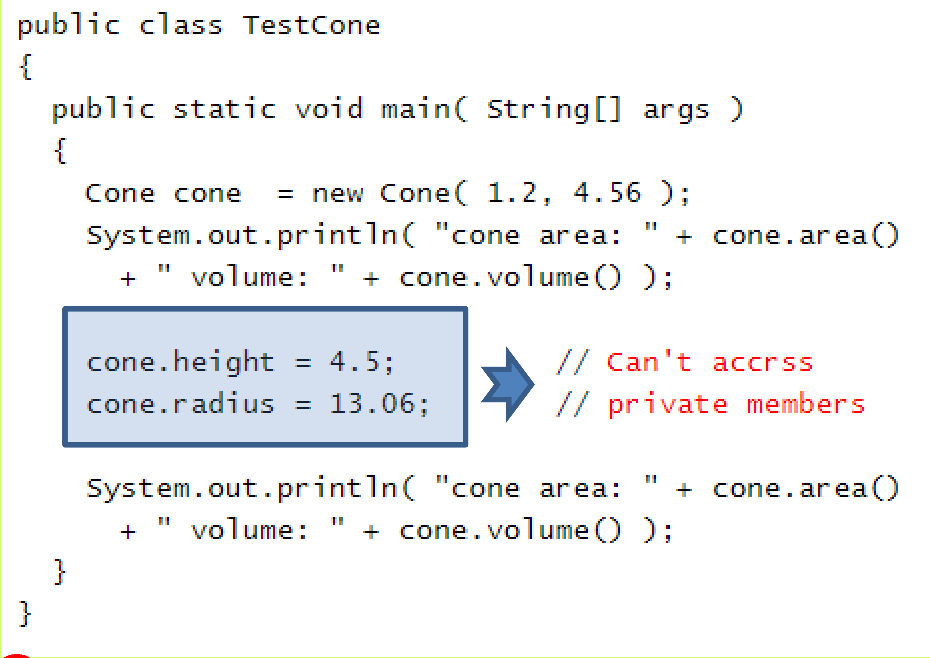
```
public double getHeight()
public double getRadius()
```

```
public void setHeight()
public void setRadius()
```

Getters

Setters

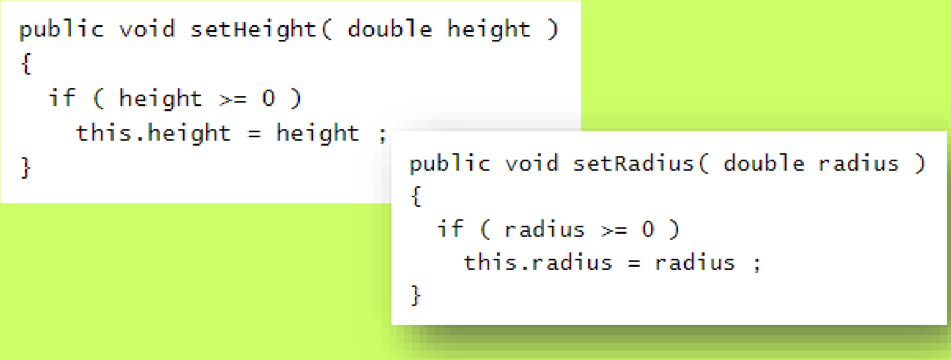
Getters & Setters



```
public class TestCone
{
  public static void main( String[] args )
  {
    Cone cone  = new Cone( 1.2, 4.56 );
    System.out.println( "cone area: " + cone.area()
      + " volume: " + cone.volume() );

    cone.height = 4.5;          // Can't accrss
    cone.radius = 13.06;        // private members

    System.out.println( "cone area: " + cone.area()
      + " volume: " + cone.volume() );
  }
}
```

? How would you define the mutators and access methods in the class Cone?

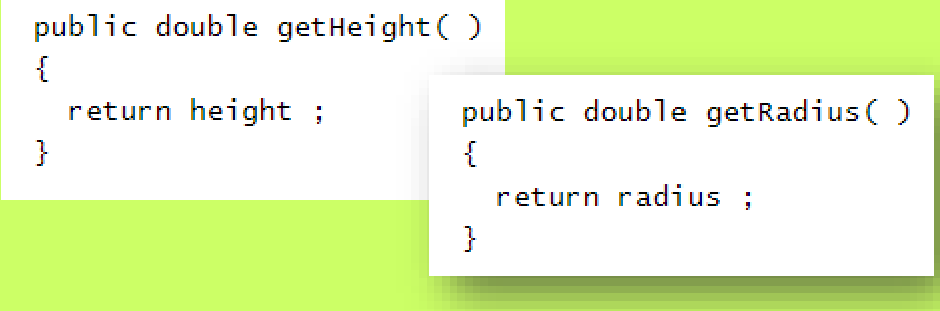
Setter → Mutator



```
public void setHeight( double height )
{
  if ( height >= 0 )
    this.height = height ;
}
```

```
public void setRadius( double radius )
{
  if ( radius >= 0 )
    this.radius = radius ;
}
```

Getter → Access method

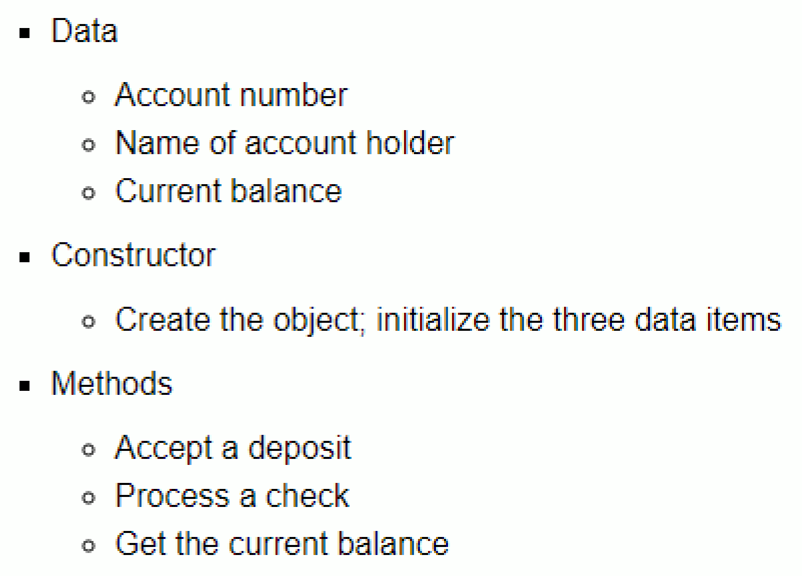


```
public double getHeight( )
{
  return height ;
}
```

```
public double getRadius( )
{
  return radius ;
}
```

Designing a Class (Checking Account)

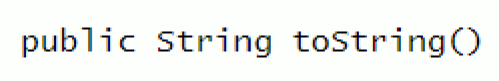
Requirements:



- Data
  - Account number
  - Name of account holder
  - Current balance
- Constructor
  - Create the object; initialize the three data items
- Methods
  - Accept a deposit
  - Process a check
  - Get the current balance

More requirements:

- Current balance can be **negative** or **positive**
- When processing a check, if the balance is less than $1000.00, €0.15 is **charged** for each check.
- Methods do **not check** for data **errors** (we assume that all data is correct)
- Implement (or override)* the `toString()` method, in order to show all the Account data.



```
public String toString()
```

**All objects automatically have a toString() method which they inherit from the class Object. It returns a String with the name of the class and the memory address of the object.*

If you put your own toString() method in a class, that one will be used instead of the inherited method.

Solution

```
// instance variables
private String accountNumber;
private String accountHolder;
private int    balance;
```

```
//constructors
public CheckingAccount( String accNumber, String holder, int start )
{
  accountNumber = accNumber ;
  accountHolder = holder ;
  balance       = start ;
}
```

```
// methods
public int getBalance()
{
  return balance ;
}

public void  processDeposit( int amount )
{
  balance = balance + amount ;
}
```

CheckingAccount class

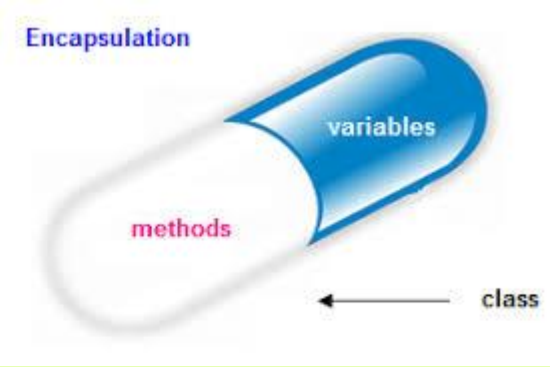
Now test it!

```
public void processCheck( int amount )
{
  int charge;
  if ( balance < 100000 )
    charge = 15;
  else
    charge = 0;

  balance =  balance - amount - charge  ;
}
```

```
public String toString()
{
   return "Account: " + accountNumber + ";\tOwner: " + accountHolder + ";\tBalance: " + balance ;
}
```

Encapsulation

Make instance variables only visible to the object's own methods.



Apply `private` modifier to the instance variables!

Write **getters** and **setters** for these variables.

? Can a method be private?

! **Yes!** But it can only be used by the other methods of the object. (`main()` can't use a private method)

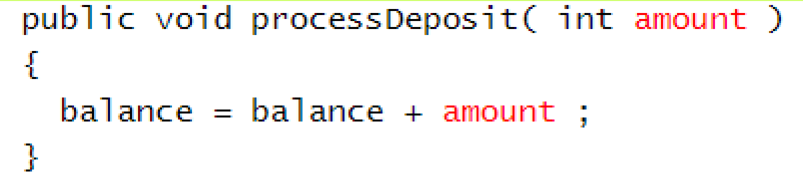
? Should a constructor be made public or private?

! **Almost always** it will be public. So that objects can be constructed by "outsiders" such as the main() method of a testing class.

Formal and Actual Parameters

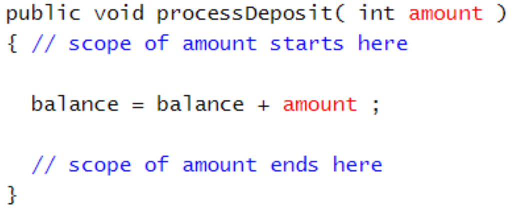
Formal parameter

- **Identifier** used in a method to stand for the value that is passed to the method



```
public void processDeposit( int amount )
{
  balance = balance + amount ;
}
```

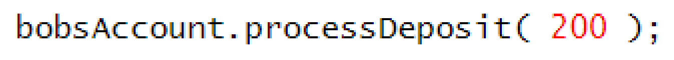
*The formal parameters of a method can be seen **only** by the statements of their own method.*



```
public void processDeposit( int amount )
{ // scope of amount starts here

  balance = balance + amount ;

  // scope of amount ends here
}
```

Actual parameter (argument)

- The actual **value** that is passed to the method



```
bobsAccount.processDeposit( 200 );
```

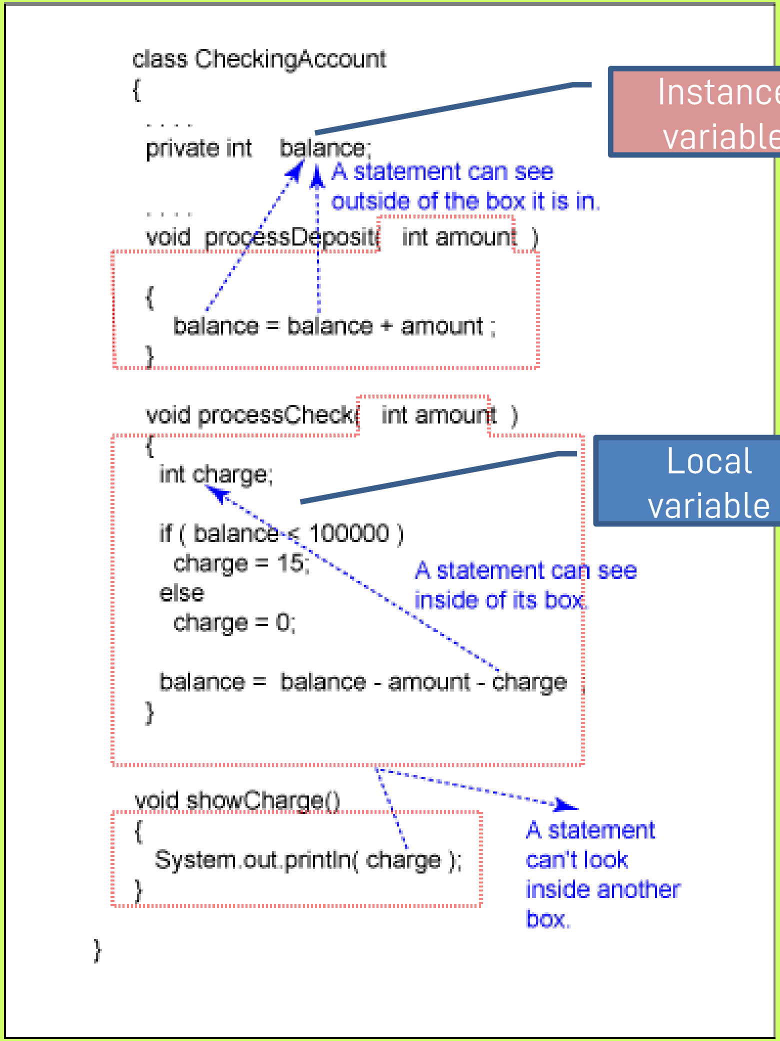
Formal parameters are bound to an actual value only as long as their method is active.

Call by value: Changes to the Formal Parameter do not affect the Caller

Example

Parameter Scopes

```
public void  processDeposit( int amount )
{ // scope of amount starts here
  balance = balance + amount ;
  // scope of amount ends here
}

public void processCheck( int amount )
{ // scope of amount starts here
  int charge;

  incrementUse();
  if ( balance < 100000 )
    charge = 15;
  else
    charge = 0;

  balance =  balance - amount - charge  ;
  // scope of amount ends here
}
```



```
class CheckingAccount
{
 . . . .
 private int    balance;

 . . . .
 void  processDeposit(  int amount  )
 {
    balance = balance + amount ;
 }

 void processCheck(  int amount  )
 {
   int charge;

   if ( balance < 100000 )
     charge = 15;
   else
     charge = 0;

   balance =  balance - amount - charge  ;
 }

void showCharge()
{
  System.out.println( charge );
}

}
```

A statement can see outside of the box it is in.

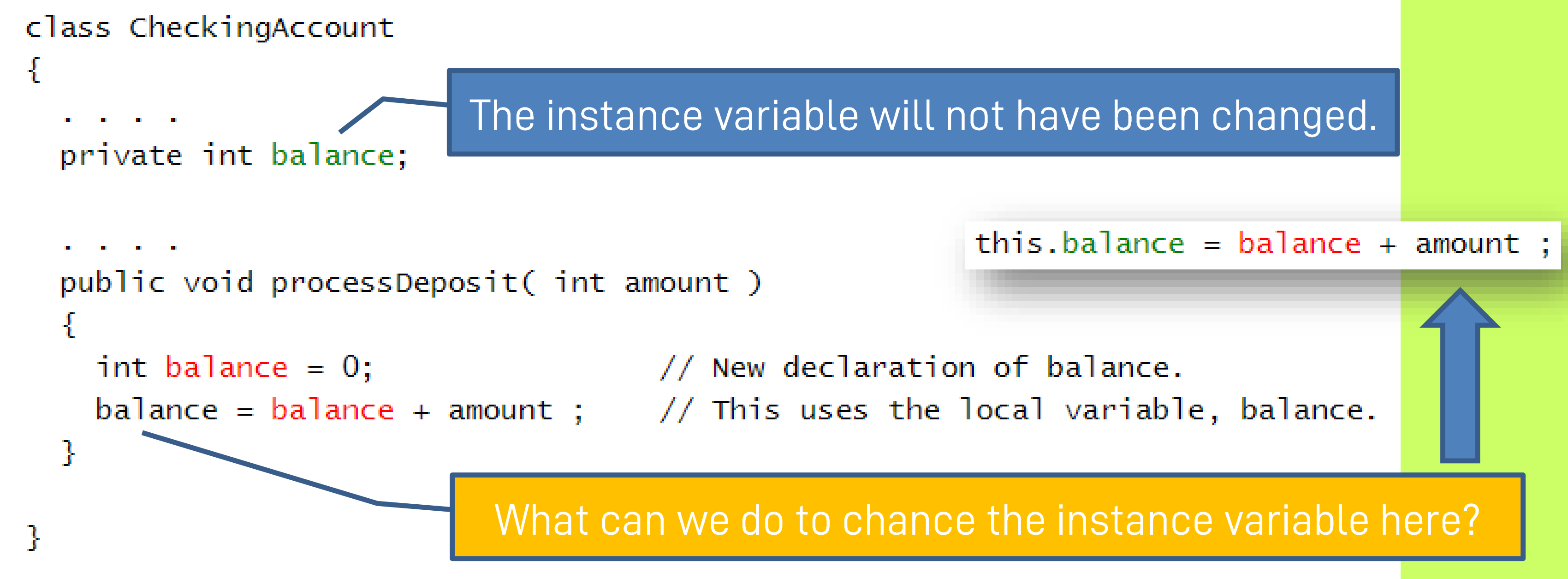
A statement can see inside of its box.

A statement can't look inside another box.

Instance variable

Local variable

Instance & Local var. with Same Name



```
class CheckingAccount
{
  . . . .
  private int balance;

  . . . .
  public void processDeposit( int amount )
  {
    int balance = 0;                 // New declaration of balance.
    balance = balance + amount ;     // This uses the local variable, balance.
  }

}
```

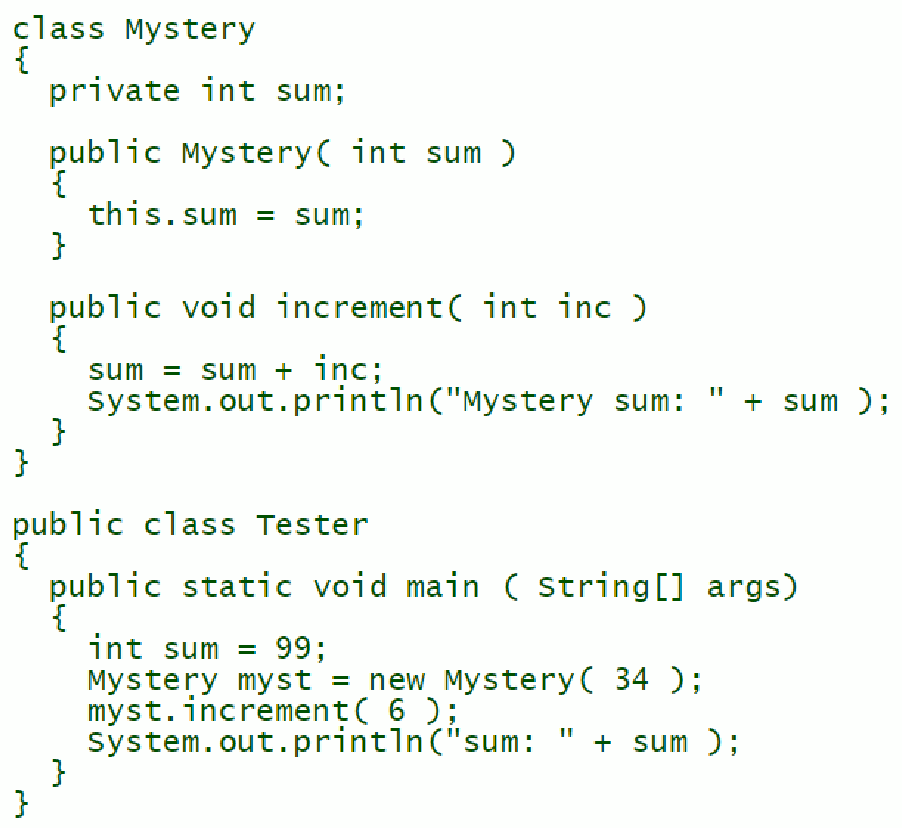
The instance variable will not have been changed.

```
this.balance = balance + amount ;
```

What can we do to chance the instance variable here?

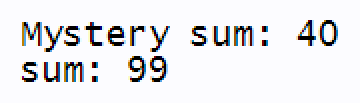
Think of statements as looking "upward" from their own location inside their "glass box" to find a variable. If they find a variable inside their box (scope), that is the one they use. An instance variable of the same name will have been shadowed.

Mistery of the Many Scopes



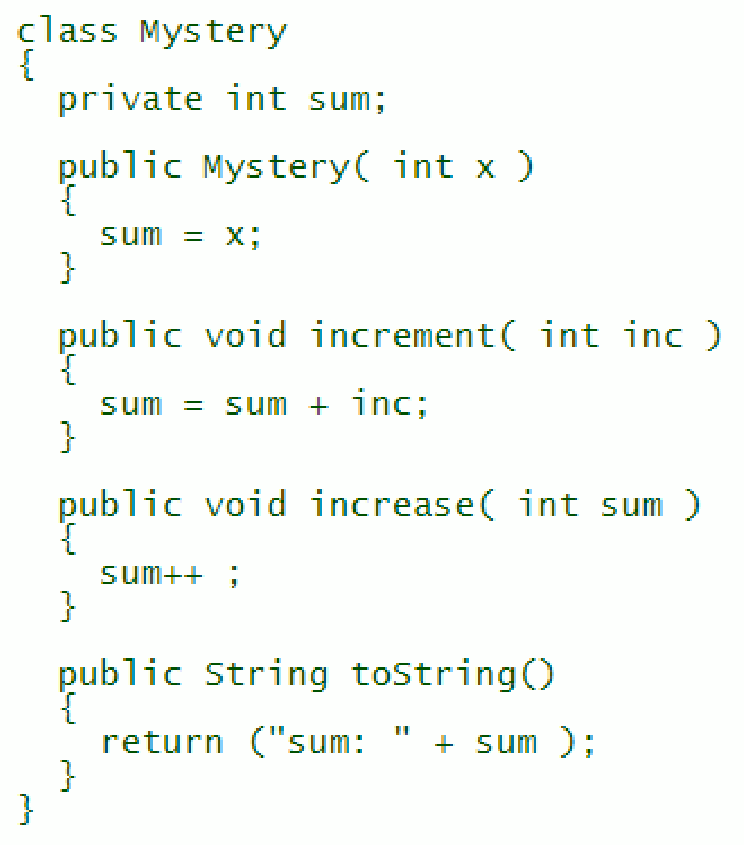
```
class Mystery
{
  private int sum;

  public Mystery( int sum )
  {
    this.sum = sum;
  }

  public void increment( int inc )
  {
    sum = sum + inc;
    System.out.println("Mystery sum: " + sum );
  }
}

public class Tester
{
  public static void main ( String[] args)
  {
    int sum = 99;
    Mystery myst = new Mystery( 34 );
    myst.increment( 6 );
    System.out.println("sum: " + sum );
  }
}
```

What does this program print?

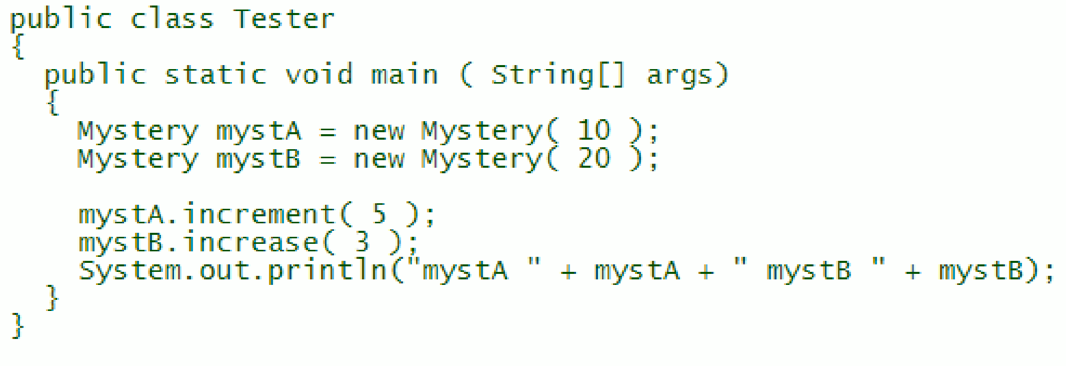


```
Mystery sum: 40
sum: 99
```

Another more mistery

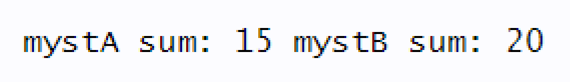


```
class Mystery
{
  private int sum;

  public Mystery( int x )
  {
    sum = x;
  }

  public void increment( int inc )
  {
    sum = sum + inc;
  }

  public void increase( int sum )
  {
    sum++ ;
  }

  public String toString()
  {
    return ("sum: " + sum );
  }
}
```



```
public class Tester
{
  public static void main ( String[] args)
  {
    Mystery mystA = new Mystery( 10 );
    Mystery mystB = new Mystery( 20 );

    mystA.increment( 5 );
    mystB.increase( 3 );
    System.out.println("mystA " + mystA + " mystB " + mystB);
  }
}
```

What does this program print?

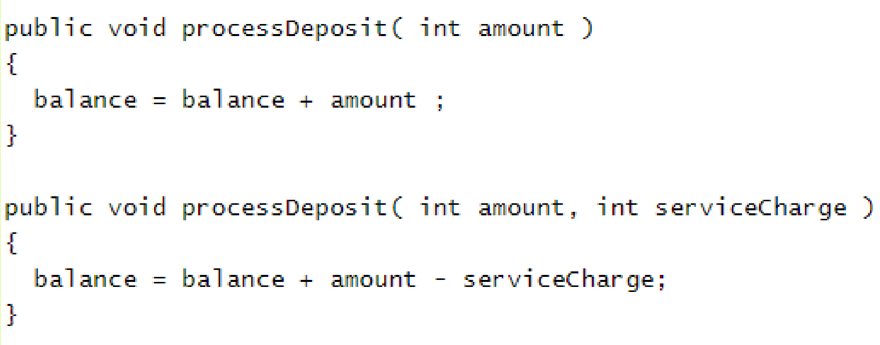


```
mystA sum: 15 mystB sum: 20
```

The instance variable in mystB did not change. The trick: the parameter sum of the second method shadowed the instance variable.

Method Overloading

Overloading is when two or more methods of a class have the **same name** but have **different parameter lists**.

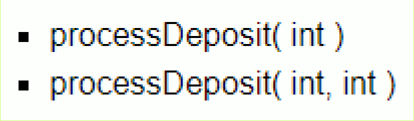


```
public void processDeposit( int amount )
{
  balance = balance + amount ;
}

public void processDeposit( int amount, int serviceCharge )
{
  balance = balance + amount - serviceCharge;
}
```

Method Signature

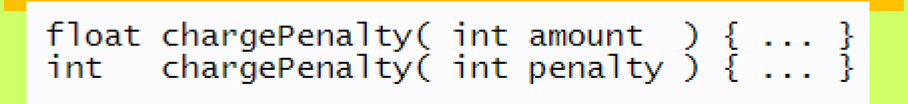
The **signature** of a method is:

- Its **name**
- The **number** and **types** of its parameters, in order



- processDeposit(int)
- processDeposit(int, int)

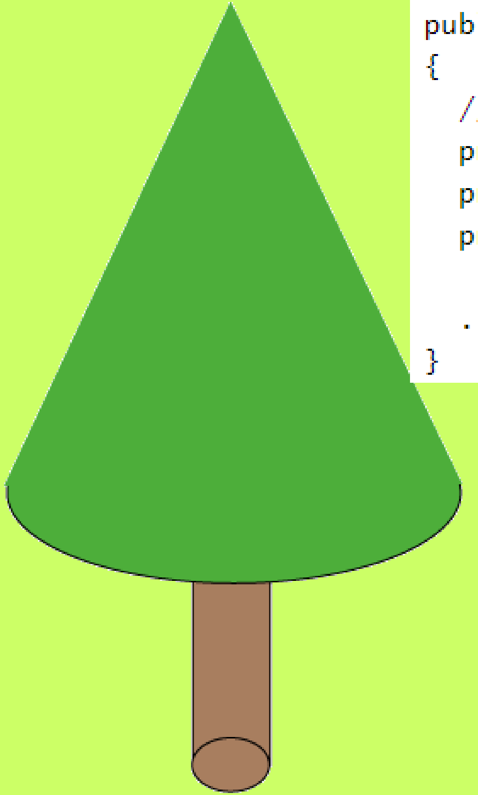
? Do these methods have unique signatures?



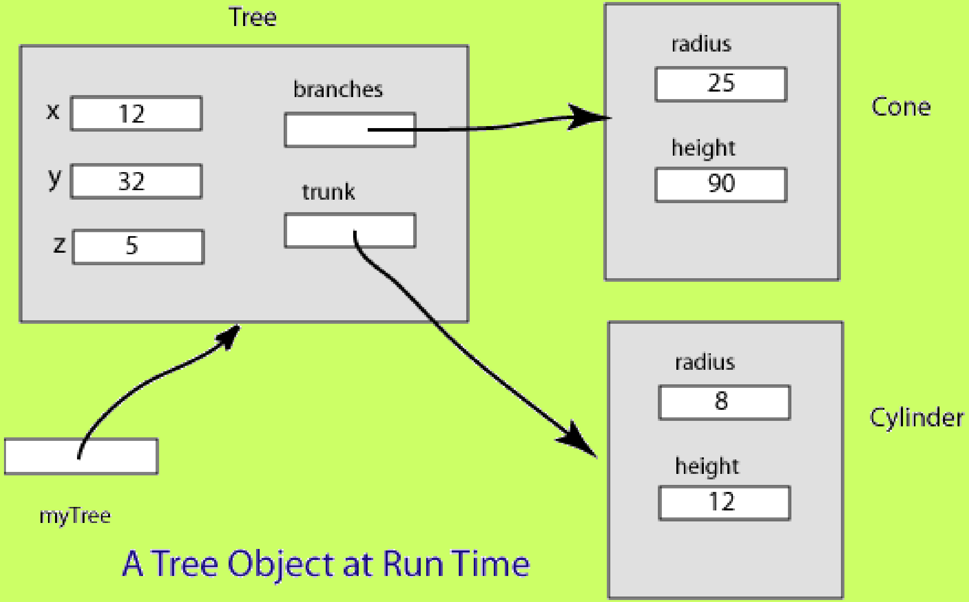
```
float chargePenalty( int amount  ) { ... }
int   chargePenalty( int penalty ) { ... }
```

No. *The names of the formal parameters are not part of the signature, nor is the return type. Both have the same signature*

Objects that Contain Objects



```
public class Tree
{
  // instance variables
  private double x, y, z;
  private Cone branches;
  private Cylinder trunk;

  ...
}
```



A Tree Object at Run Time

Constructor that uses Constructors

```
public class Tree
{
  // instance variables
  private double x, y, z;
  private Cone branches;
  private Cylinder trunk;


  ...
}
```

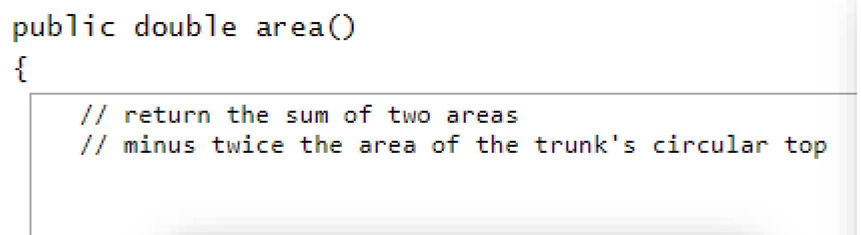
? What is the volume of the tree?

```
public double volume()
{
  // return the sum of two volumes
  return trunk.volume() + branches.volume();
}
```

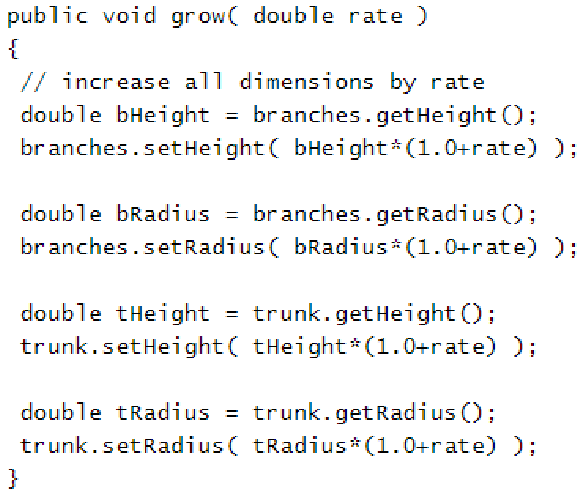
```
// constructor
public Tree( double trRad, double trHeight, double brRad, double brHeight, double x, double y, double z)
{
  trunk = new Cylinder( trRad, trHeight );
  branches = new Cone( brRad, brHeight );
  this.x = x; this.y = y; this.z = z;
}
```

Think of this

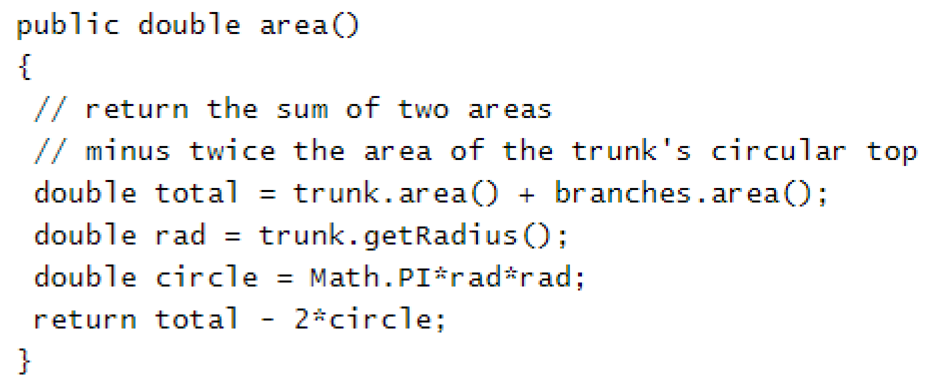
Program these two methods:



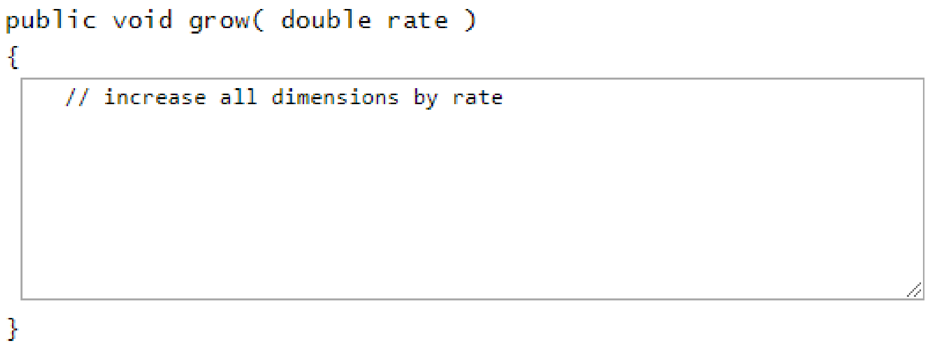
```
public double area()
{
    // return the sum of two areas
    // minus twice the area of the trunk's circular top



}
```



```
public void grow( double rate )
{
 // increase all dimensions by rate
 double bHeight = branches.getHeight();
 branches.setHeight( bHeight*(1.0+rate) );

 double bRadius = branches.getRadius();
 branches.setRadius( bRadius*(1.0+rate) );

 double tHeight = trunk.getHeight();
 trunk.setHeight( tHeight*(1.0+rate) );

 double tRadius = trunk.getRadius();
 trunk.setRadius( tRadius*(1.0+rate) );
}
```

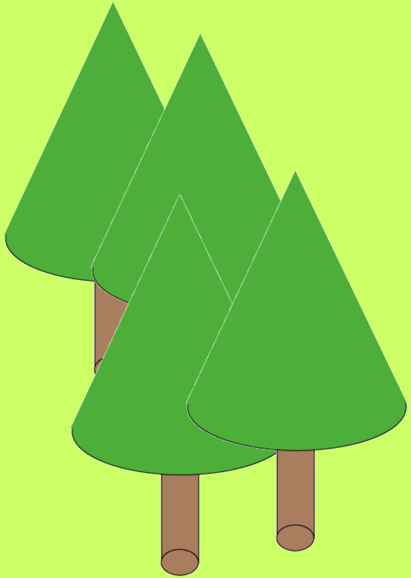


```
public double area()
{
 // return the sum of two areas
 // minus twice the area of the trunk's circular top
 double total = trunk.area() + branches.area();
 double rad = trunk.getRadius();
 double circle = Math.PI*rad*rad;
 return total - 2*circle;
}
```

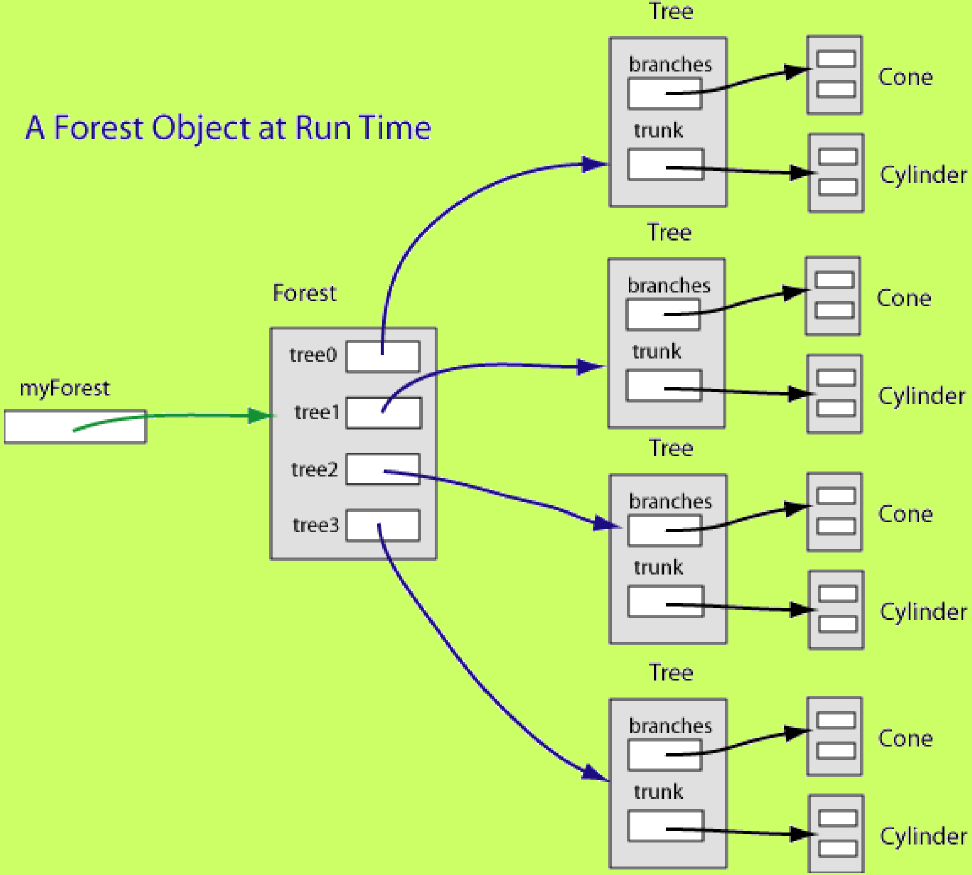


```
public void grow( double rate )
{
    // increase all dimensions by rate



}
```

Can't see the Forest for the Trees



A Forest Object at Run Time



myForest

Forest

tree0

tree1

tree2

tree3

Tree

branches

trunk

Cone

Cylinder

Tree

branches

trunk

Cone

Cylinder

Tree

branches

trunk

Cone

Cylinder

Tree

branches

trunk

Cone

Cylinder