- **Software Development Activities**
- **Understanding Objects in Java: Chapter 2**

Software Development Activities

- **Requirements Analysis**
  - Software requirements: what a program must accomplish
- **Design**
- **Implementation**
- **Testing**

Software Development Activities

- **Requirements Analysis**
  - Software requirements: what a program must perform, NOT how it will perform it
  - Functional requirements
  - Non-functional requirements
- **Output: Requirements document (also called functional specification)**

Software Development Activities

- **Design**
  - how a program will accomplish its requirements
  - O-O design involves:
    - **identify the classes and objects**
    - **define how they interact**
    - **relationship among the classes,** *inheritance, dependency….*

Software Development Activities

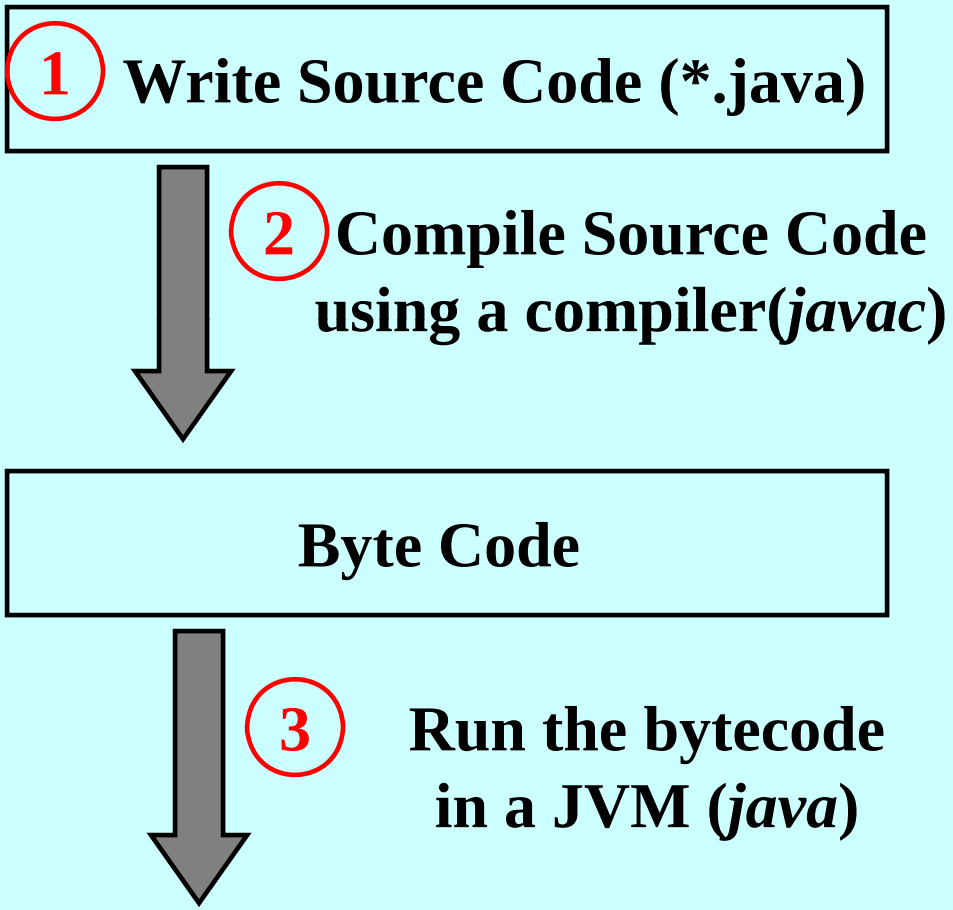
- **Implementation**
  - process of writing the source code
  - translating the design into a particular programming language
  - focus on coding details, coding guidelines and documentation

Software Development Activities

- **Testing**
  - ensure that a program does what it should do
  - *use JUnit framework…*

Creating a simple Java Application

Creating a simple Java application:



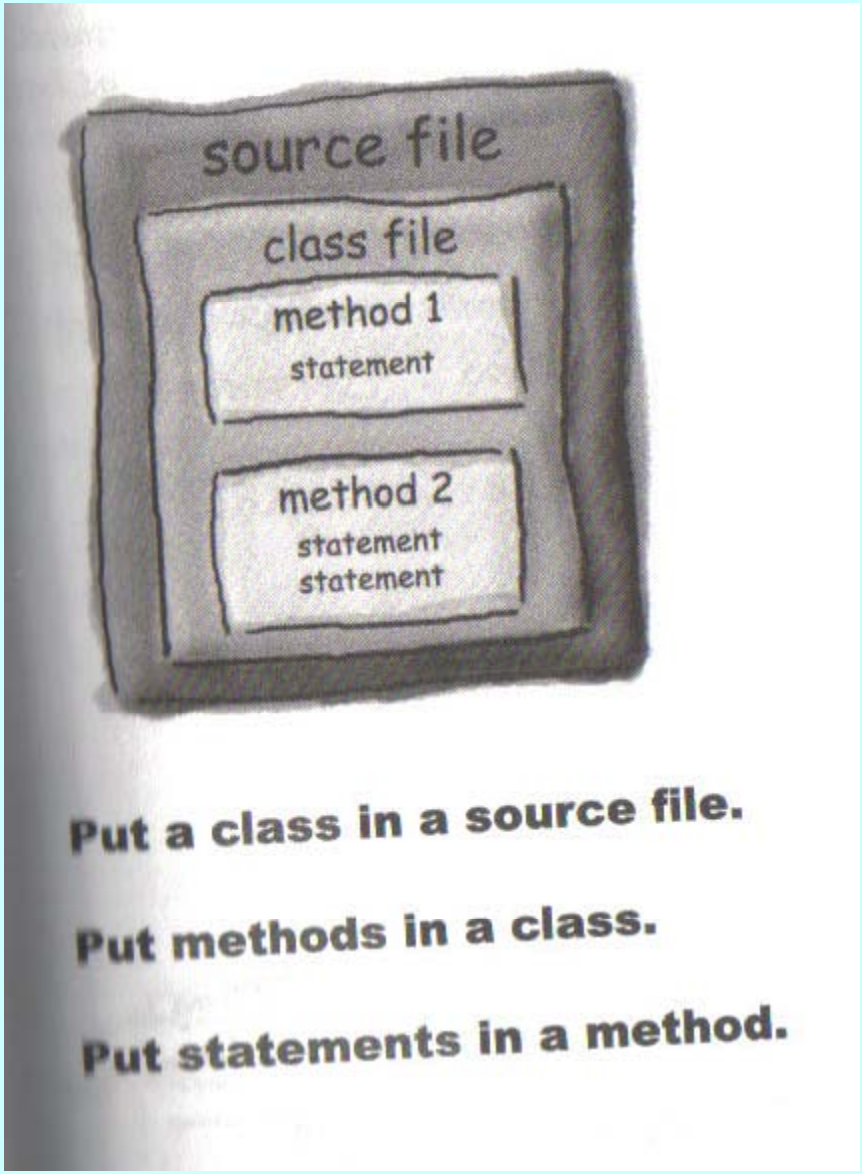
1. **Create a file, e.g. HelloWorld.java**
2. **Compile:**

   javac HelloWorld.java
3. **Run:**

   java HelloWorld

Code structure

in Java



When the JVM starts running,

- **it looks for the class you give it at the command line**
- **then it looks for the main method inside that class**
- **main method is executed until the end of main is reached**

main method looks like:

```
public static void main(String[] args)
{
    ……
}
```

What will happen when HelloWorld class runs:

java HelloWorld

```
public class HelloWorld
{
  ⇒public static void  main(String[] args)
     {
       ⇒System.out.println("Hello");
       ⇒System.out.print("World);
       ⇒System.out.println("Example");
     }
}
```

Output:

Hello

WorldExample

Objects and Classes

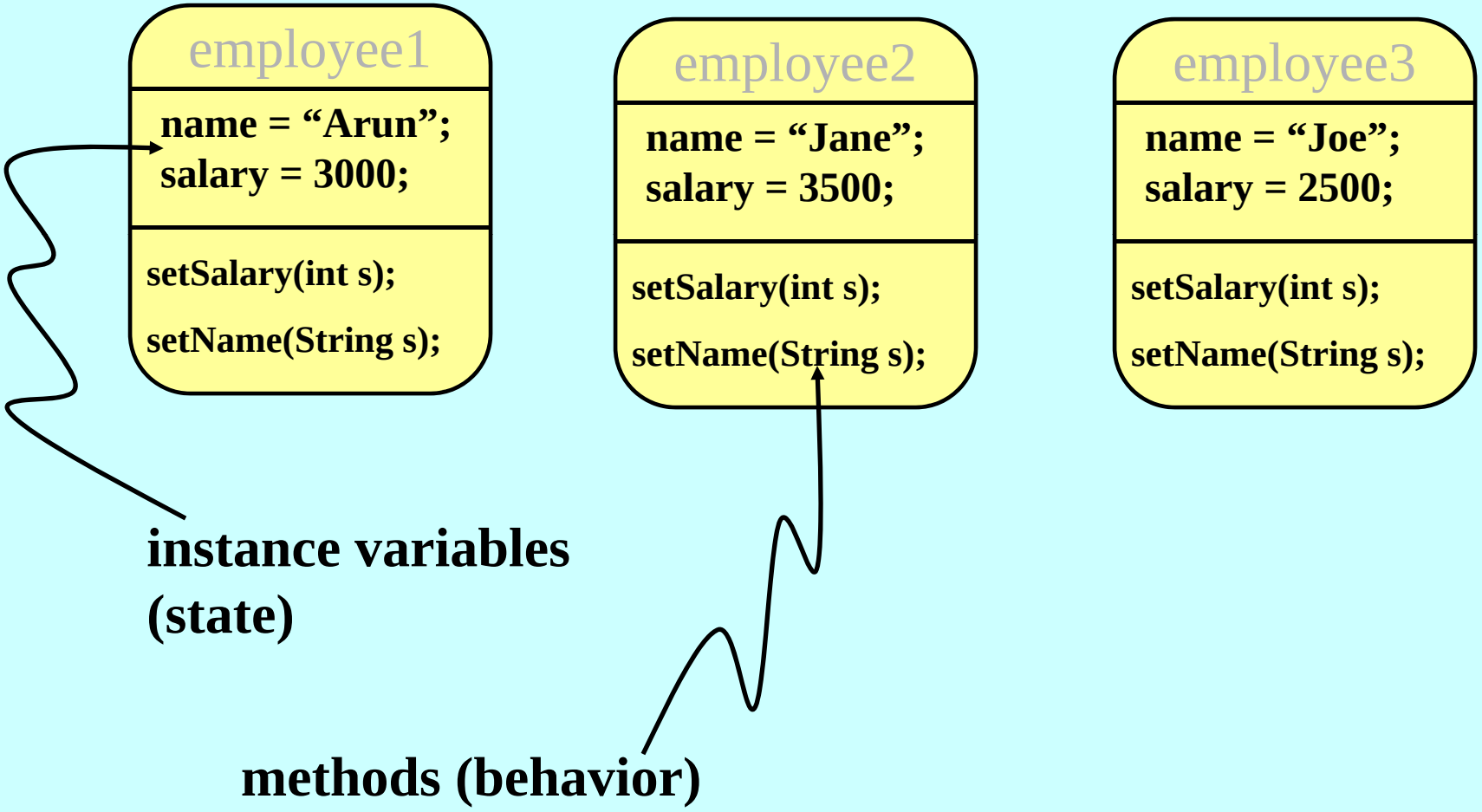
What is an Object ?

- **An object can simply represent any object or entity that you may think of in "real world".**

e.g. a Person, a Car, a table, a Bank Account, an Employee

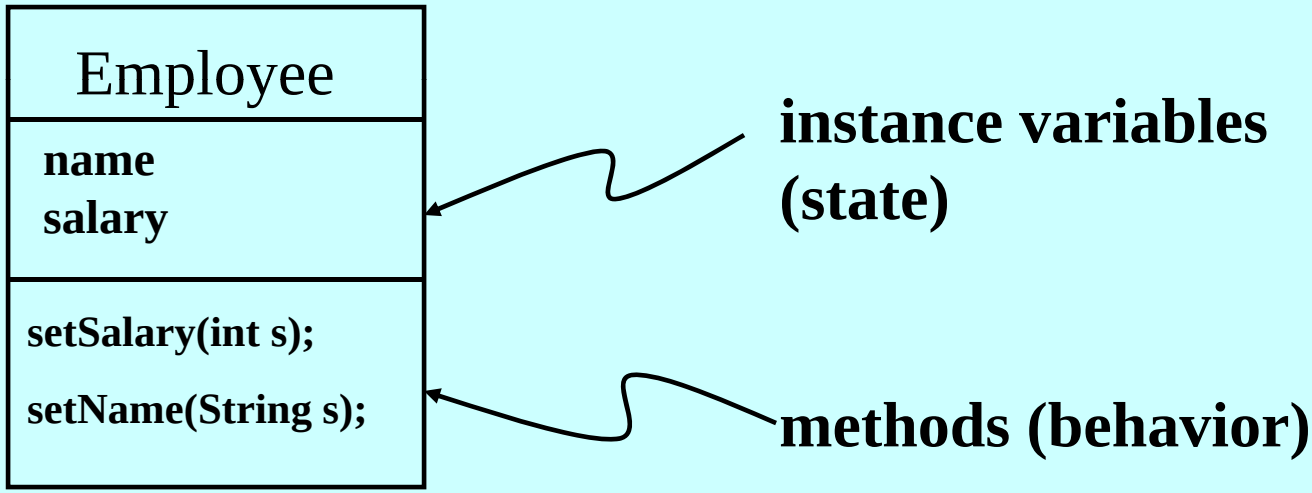
- **Every software object have state and behavior**
  - **state is defined by one or more *variables***
  - **behavior is defined by its *methods.***

Visual representation of software objects



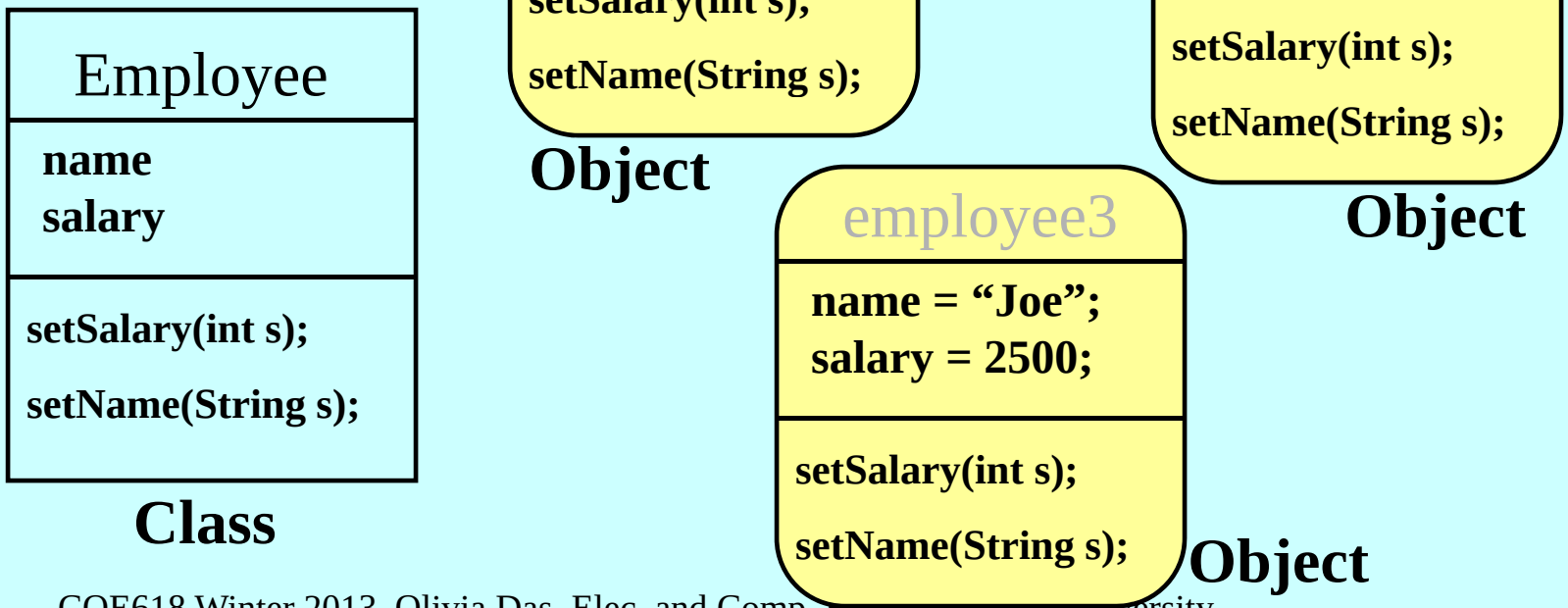
What is a Class ?

• A class is a blueprint that defines the variables and the methods common to all objects of a certain kind..



Difference between a class and an object ?

- A class is a blueprint for an object
- A class is used to construct objects of that kind



Creating and Using Objects

Lets write two classes:

- **One class for the type of the object we want to use**

  e.g. Circle

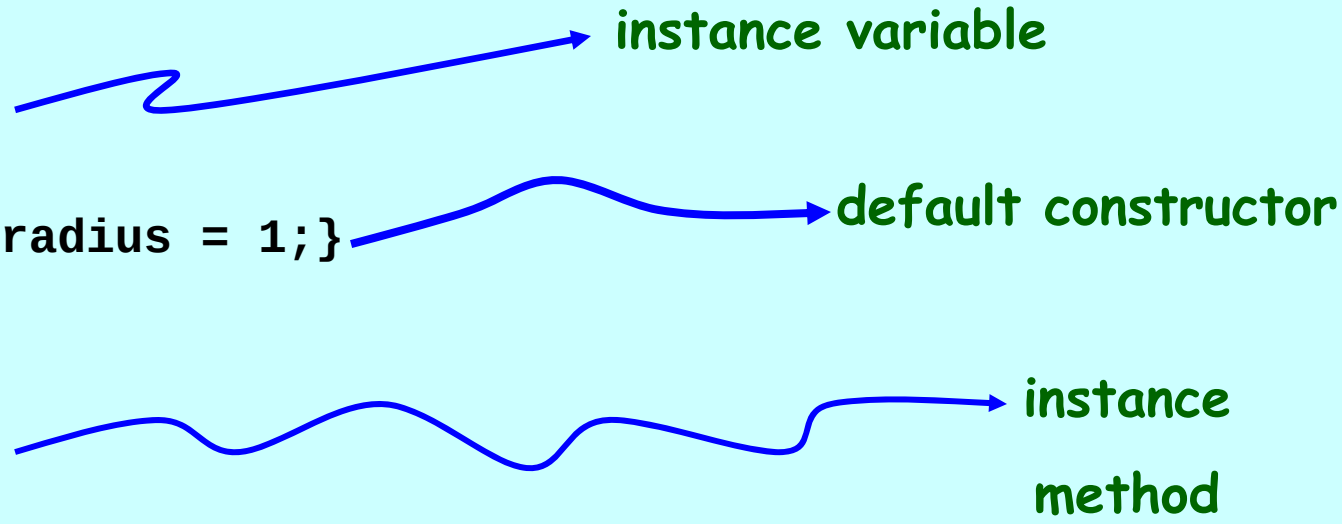
- **Another class to *test* our new class that tests the methods and variables of the newly created object. Let the name of the tester class be CircleTest**

  - **In this tester class, we put the main method.**

  - **Inside main method, we create and access objects of our new class type.**

Class Declaration

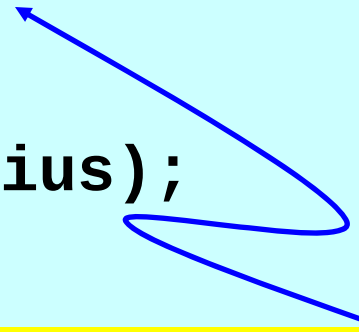
```
public class Circle
{
    int radius;

    Circle() { radius = 1;}


    void dump()
    {
        System.out.println("A Circle.");
    }
}
```



CircleTest class (the client): uses Circle class

The dot (.) operator gives you access to an object's state and behavior.

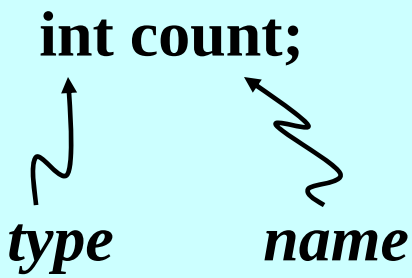
```
public class CircleTest
{
   public static void main(String[] args)
   {
      Circle c = new Circle();
      c.radius = 5;
      System.out.println(c.radius);
      c.dump();
   }
}
```



create a new Circle object (by calling a constructor)

Variables and Data Types

Variables must have a type and a name.



Two kinds of variables:

- **Primitive type**
- **Object Reference type**

Primitive Types

Type	Bit Depth	Value Range
boolean	**(JVM specific)**	***true* or *false***
char	**16 bits**	**0 to 65535**
byte	**8 bits**	**-128 to 127**
short	**16 bits**	**-32268 to 32767**
int	**32 bits**	**-2147483648 to 2147483647**
long	**64 bits**	**-huge to huge**
float	**32 bits**	**varies**
double	**64 bits**	**varies**

Primitive Declarations with assignments

```
int x;
x = 234;
byte b = 89;
double d = 34.98;
char c = ‘f’;
int z = x;
boolean found = true;
long b = 3456789;
float f = 32.5f;
```

Which conversions are legal ??

int x = 24;

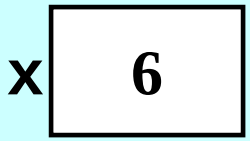
byte b = x;

byte b = 127;

int j = b;

• **Primitive: byte, short, int, long, float, double, char, boolean.**

```
int x;
 x = 6;
```

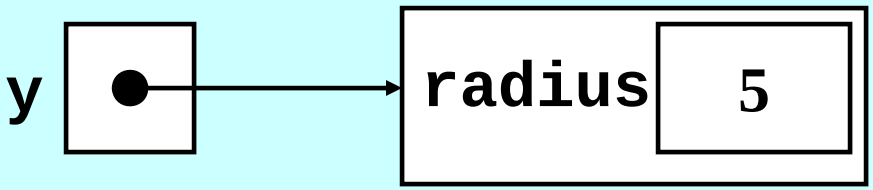


```
public class Circle {
    int radius;
    Circle(int j) {
        radius = j;
    }
    int getRadius() {
        return radus;
    }
    ……
}
```

- **Object Reference (Non-primitive)**

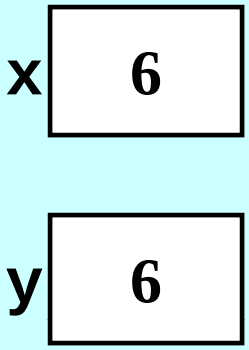
```
Circle y;
y = new Circle(5);
```

a Circle object



• **Primitive: byte, short, int, long, float, double, char, boolean.**

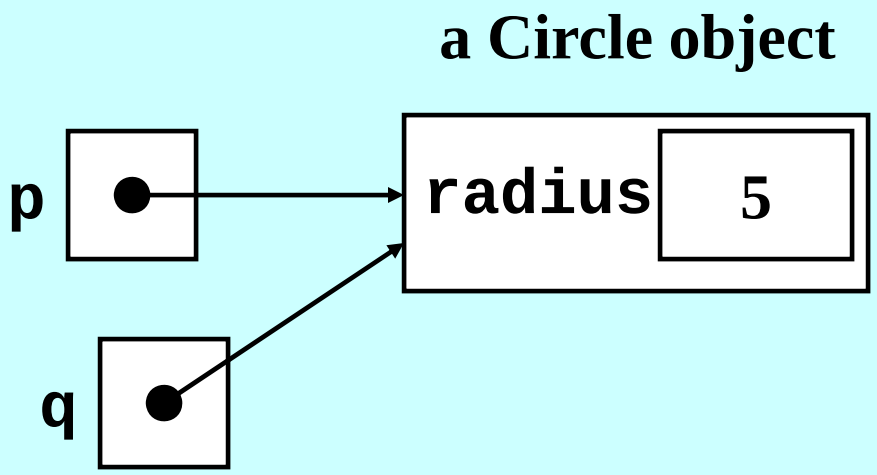
```
int x;
x = 6;
int y = x;
```



p and q are aliases

• **Reference (Non-primitive)**

```
Circle p;
p = new Circle(5);
Circle q = p;
```



Arrays

Arrays

- **objects that help us organize large amount of information**
- **An array is a structure that can hold multiple values of the same type.**
- **After creation, an array is a fixed-length structure.**
- **is an object in Java**

Arrays: Example

```
public class ArrayTest {
    public static void main(String[] args) {
        int[] ia = new int[4];
        for (int i = 0; i < ia.length; i++) {
            ia[i] = i;
        }
        for (int i = 0; i < ia.length; i++) {
            System.out.println( ia[i] );
        }
    }
}
```

Answer:

0

1

2

3

Alternate Array Syntax

- Therefore the following two declarations are equivalent:

```
float[] prices;
float prices[];
```

- The first format generally is more readable and should be used

Initializer lists: Example

```
public class Primes {

  public static void main (String[] args) {

    int[] primeNums = {2, 3, 5, 7, 11, 13, 17, 19};

    System.out.println ("Array length: " +
                              primeNums.length);

    System.out.println (“First few prime numbers are:");

    for (int prime : primeNums)

        System.out.print (prime + "  ");

  }

}
```

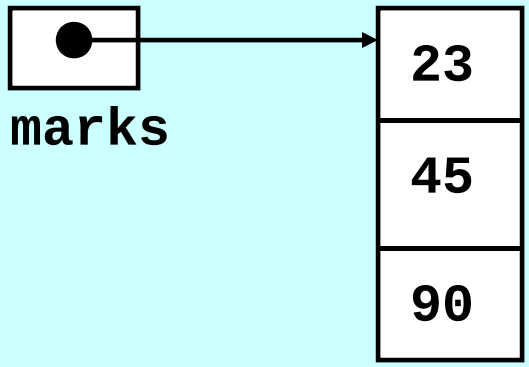
Answer:

Array length: 8

First few prime numbers are:

2 3 5 7 11 13 17 19

Exercise




```
int[] marks;

marks = new int[3];

marks[0] = 23; marks[1] = 45; marks[2] = 90;
```

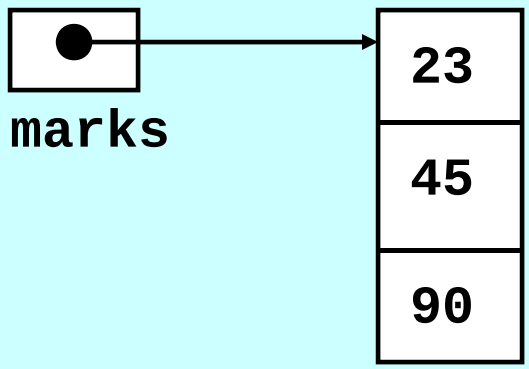
Output ??

```
System.out.println( marks[0] );

System.out.println( marks[1] );

System.out.println( marks[2] );
```

Answer:

```
23

45

90
```

Exercise

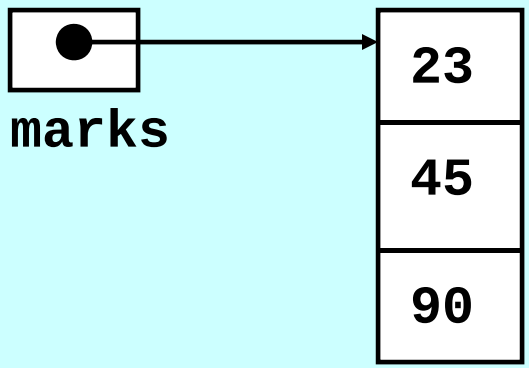



```
System.out.println( marks[1] );

System.out.println( marks[2] );

System.out.println( marks[3] );
```

Answer

Which statement will generate ArrayIndexOutOfBoundsException ??

Example




```
int[] marks = {23, 45, 90};
```

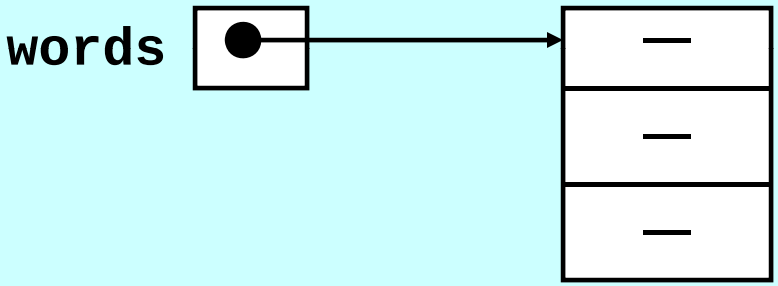
Compilation error will occur if:

```
int[] marks;

marks = {23, 45, 90};
```

Arrays of Objects

- **The elements of an array can be object references**

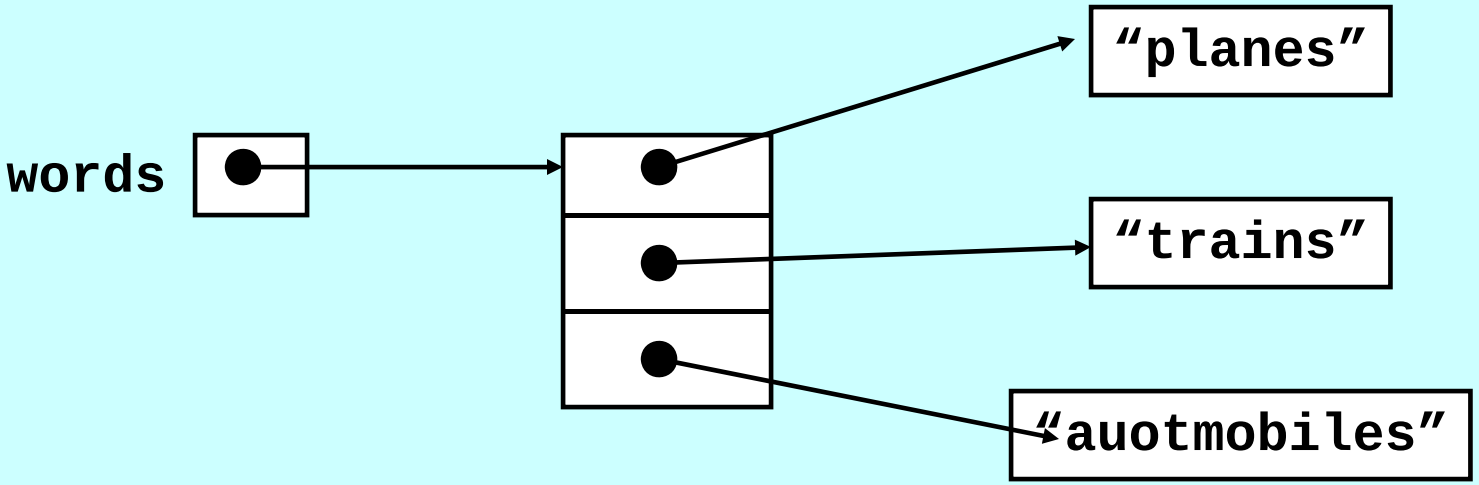
```
String[] words = new String[3];
```



- **reserves space to store 3 references to String objects**
- **DOES NOT create the String objects**
- **each object must be instantiated separately**

Arrays of Objects

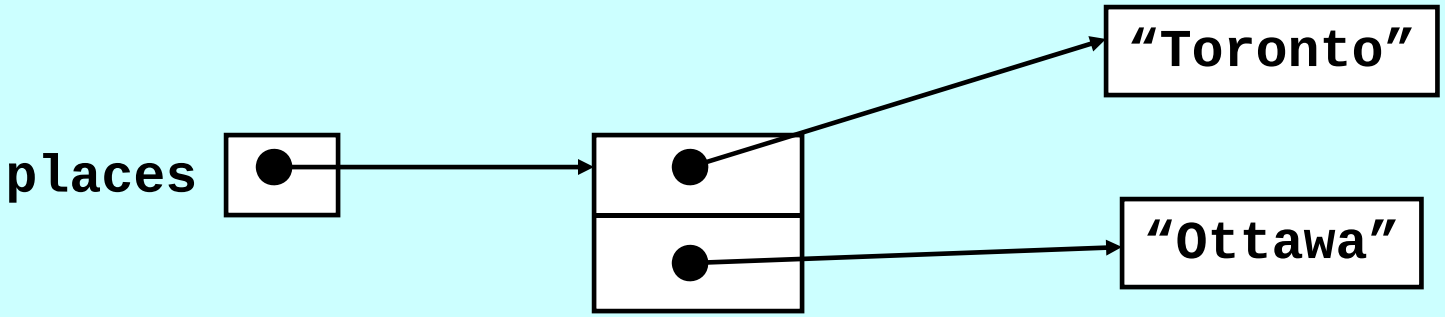
```
words[0] = new String(“planes”);
words[1] = new String(“trains”);
words[2] = new String(“automobiles”);
```



Arrays of Objects: Use of Initializer list

```
String[] places = { “Toronto”, “Ottawa”};
```

```
String[] places = { new String(“Toronto”),
                    new String(“Ottawa”) };
```



Example-1

```
public class Grade {
    private String level;

    public Grade(String level) {
        this.level = level;
    }

    public void setLevel(String l) { level = l; }
    public String getLevel() { return level; }

    public String toString() { return level; }
}
```

```
public class GradeTest {
    public static void main(String[] args) {
        Grade[] grades = {
                            new Grade("A+"),
                            new Grade("A"),
                            new Grade("A-")
                          };
        for( Grade g : grades )
            System.out.println( g );
    }
}
```

Output ??

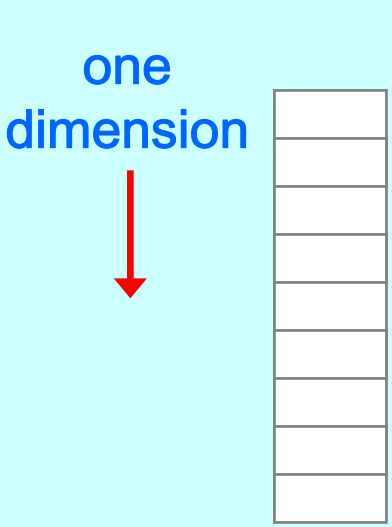
Answer:

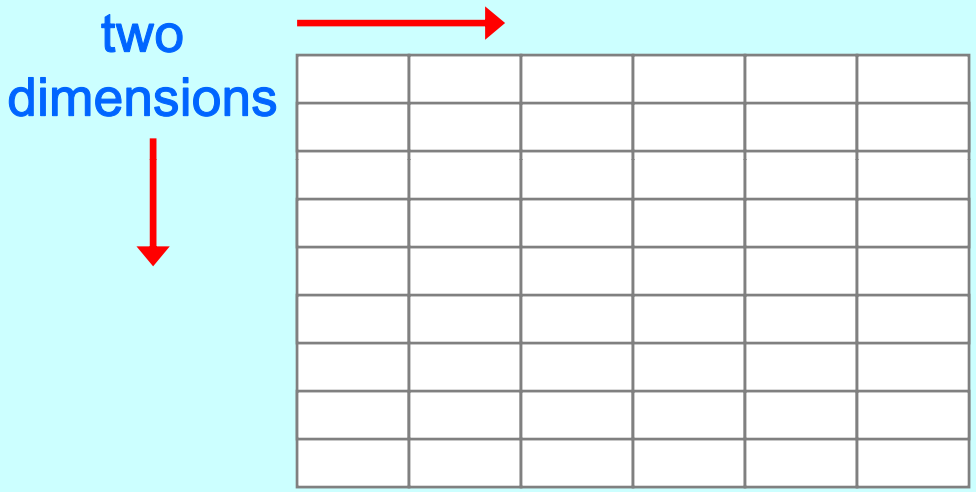
A+

A

A-

Two-dimensional Arrays





Two-dimensional Arrays

- **two-dimensional array is an array of arrays in Java**

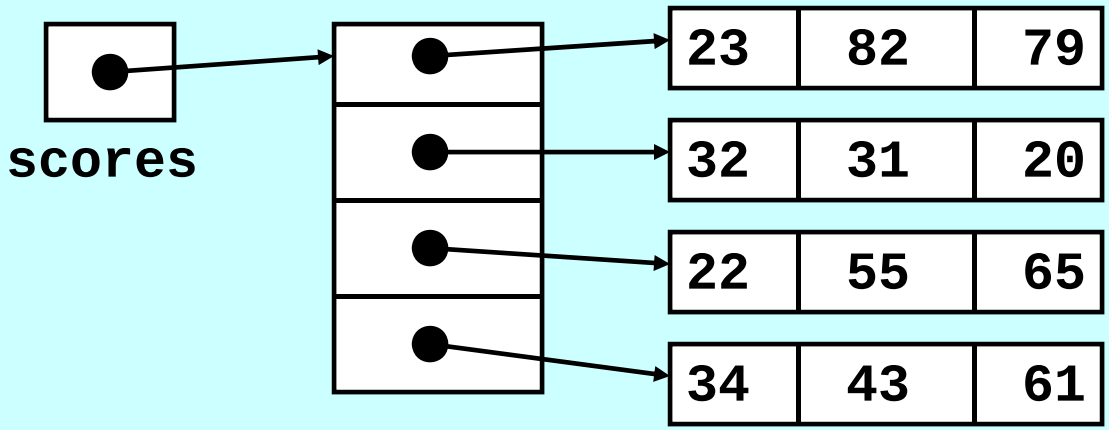
```
int[][] scores = new int[4][3];
System.out.println( scores[0][2] );
```

Output ??

Answer:

79

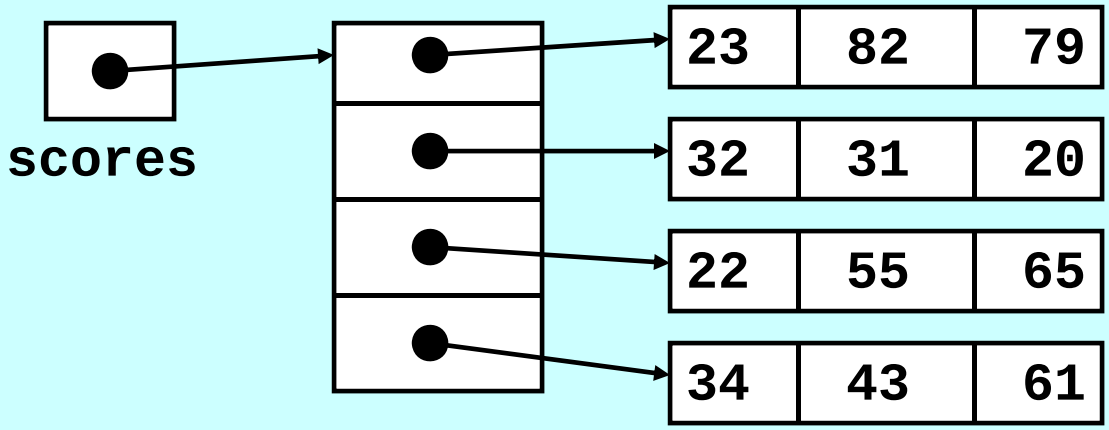
23	82	79
32	31	20
22	55	65
34	43	61



Two-dimensional Arrays

- **initializer list can be used to instantiate two-dimensional arrays**

```
int[][] scores = { {23, 82, 79},
                   {32, 31, 20},
                   {22, 55, 65},
                   {34, 43, 61} };
```



```
public static void main (String[] args) {
  int[][] table = new int[2][3];
  for (int row=0; row < table.length; row++)
  {
     for (int col=0; col < table[row].length; col++)
     {
        table[row][col] = row + col;
     }
  }
```

```
for (int row=0; row < table.length; row++)
{
   for (int col=0; col < table[row].length; col++)
   {
      System.out.print(table[row][col] + "\t");
   }
   System.out.println();
}
```

```
  for (int row=0; row < table.length; row++)
  {
     for( int value : table[row] )
     {
        System.out.print( value + "\t" );
     }
     System.out.println();
  }
}
```

```
int[][] table = new int[4][5];
```

Expression	Type	Description
`table`	`int[][]`	array of integer arrays
`table[2]`	`int[]`	array of integers
`table[1][2]`	`int`	integer

Multi-dimensional Arrays

- **Any array with more than one dimension is called *multi-dimensional array***

- **Each dimension has its own *length* constant**

- **Since each dimension is an *array of array references*, the arrays within one dimension can be of different lengths**
  - **these are sometimes called *ragged arrays***

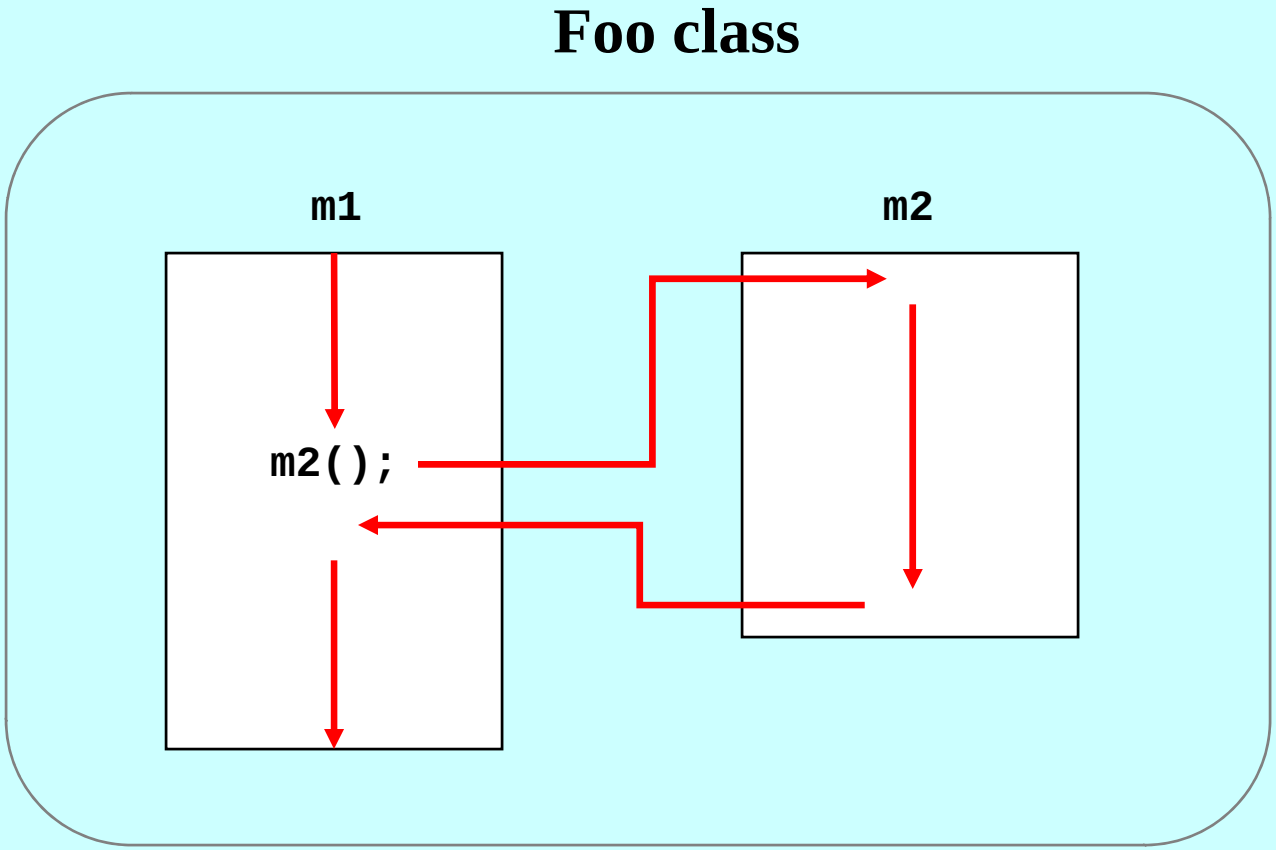
Methods and Encapsulation

Method Declarations

- A *method declaration* specifies the code that will be executed when the method is *invoked*
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

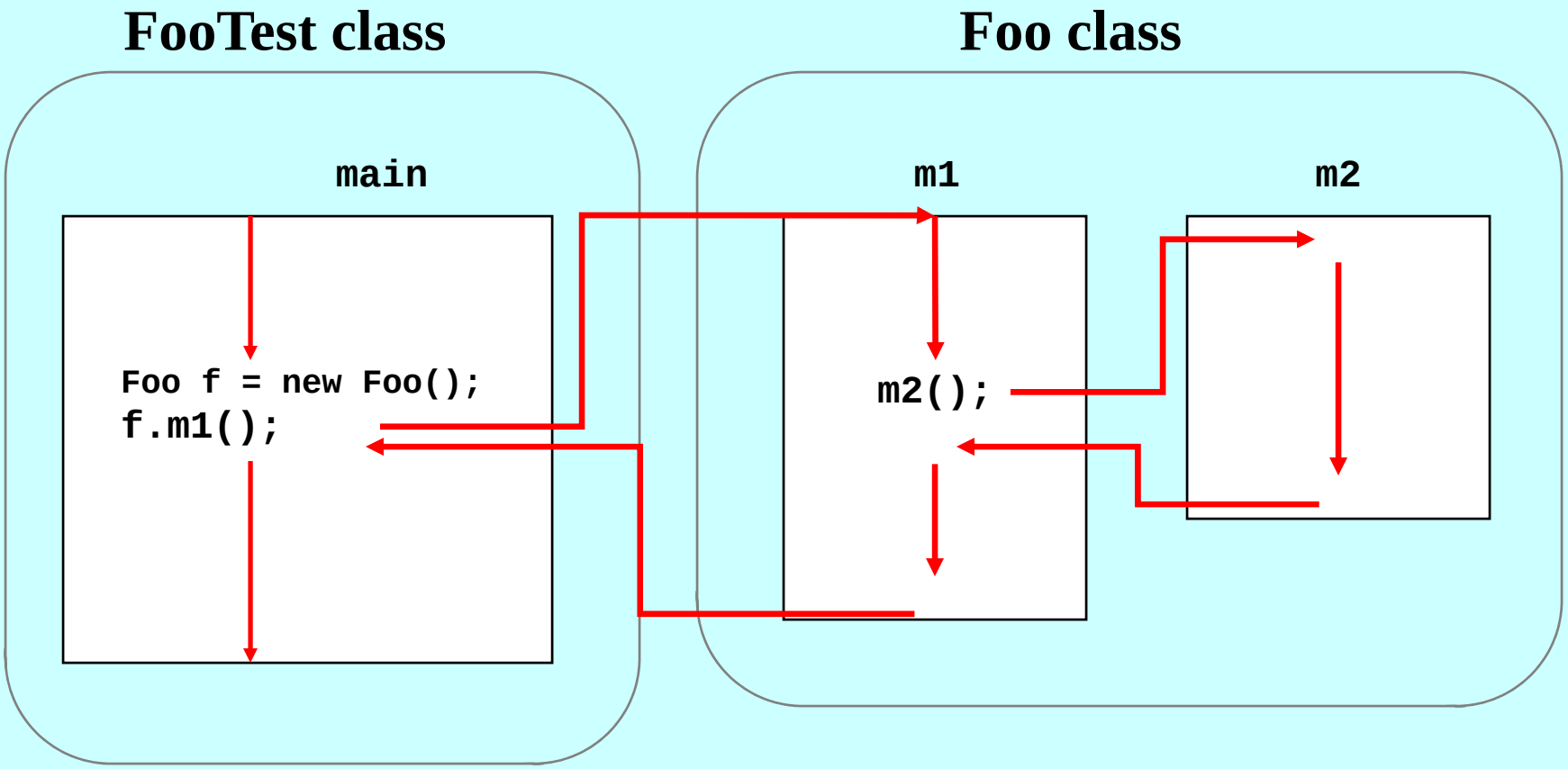
What happens when a method is invoked ?

If the called method is in the *same class*, only the method name is needed



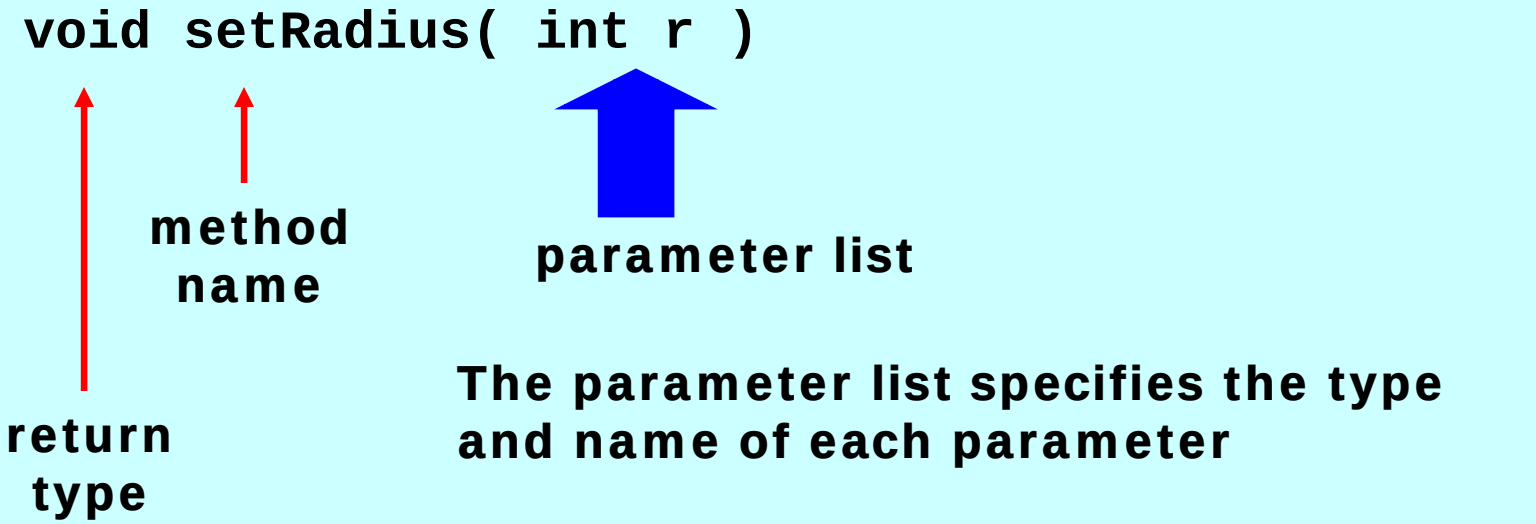
What happens when a method is invoked ?

The called method is often part of *another class*



Method Declarations

- A method declaration starts with a **method header**



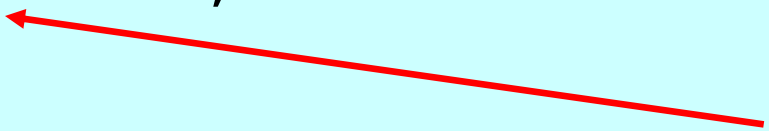
The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Declarations

The method header is followed by the *method body*

```
void setRadius( double r )
{
    radius = r;
}

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



The return expression must be consistent with the return type

The **return** statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a `void` return type
- A *return statement* specifies the value that will be returned

```
return <expression>;
```

- Its expression must conform to the return type

Method Parameters

- When a method is invoked, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header

```
circle1.setRadius( 25 );
```

```
void setRadius( double r )
{
    radius = r;
}
```

r is the formal parameter

25 is the actual parameter

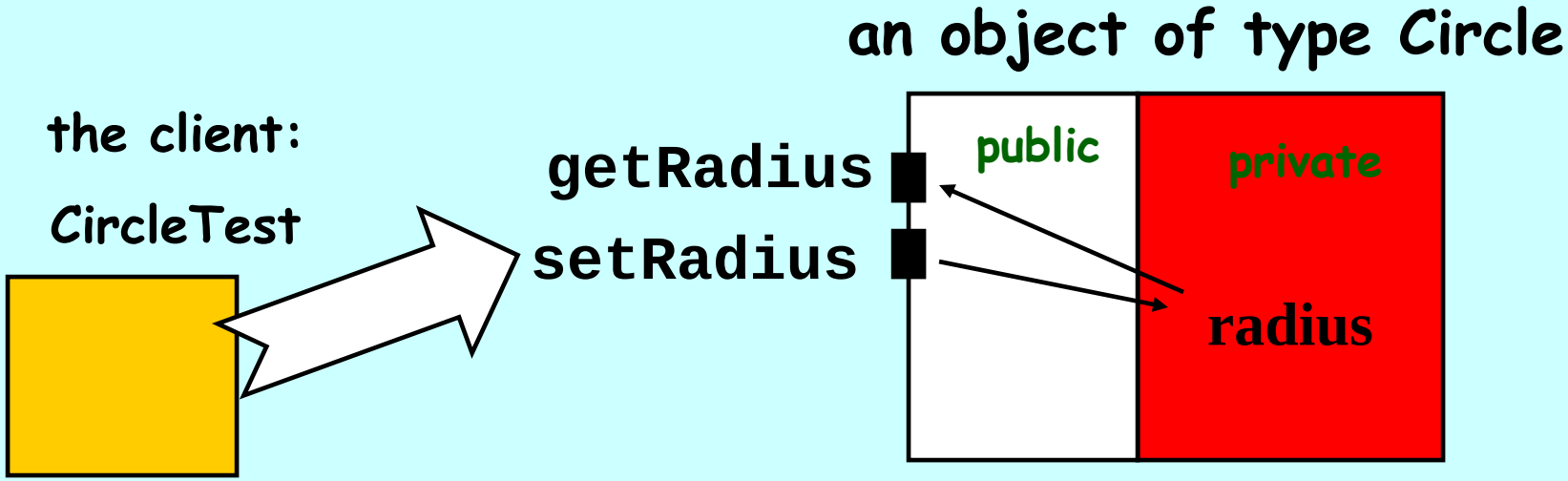
What happens when we run: *java CircleTest*

```
public class CircleTest
{
public static void main(String[] args)
{
    Circle c1 = new Circle();
    c1.setRadius( 5 );
    c1.dump();
   }
}
```

```
public class Circle
{
    private int radius;

public Circle() {
    radius = 1;
}

public void dump() {
     System.out.println("" + radius);
}
public void setRadius(int r) {
     radius = r;
}
    public int getRadius() {
        return radius;
    }
}
```

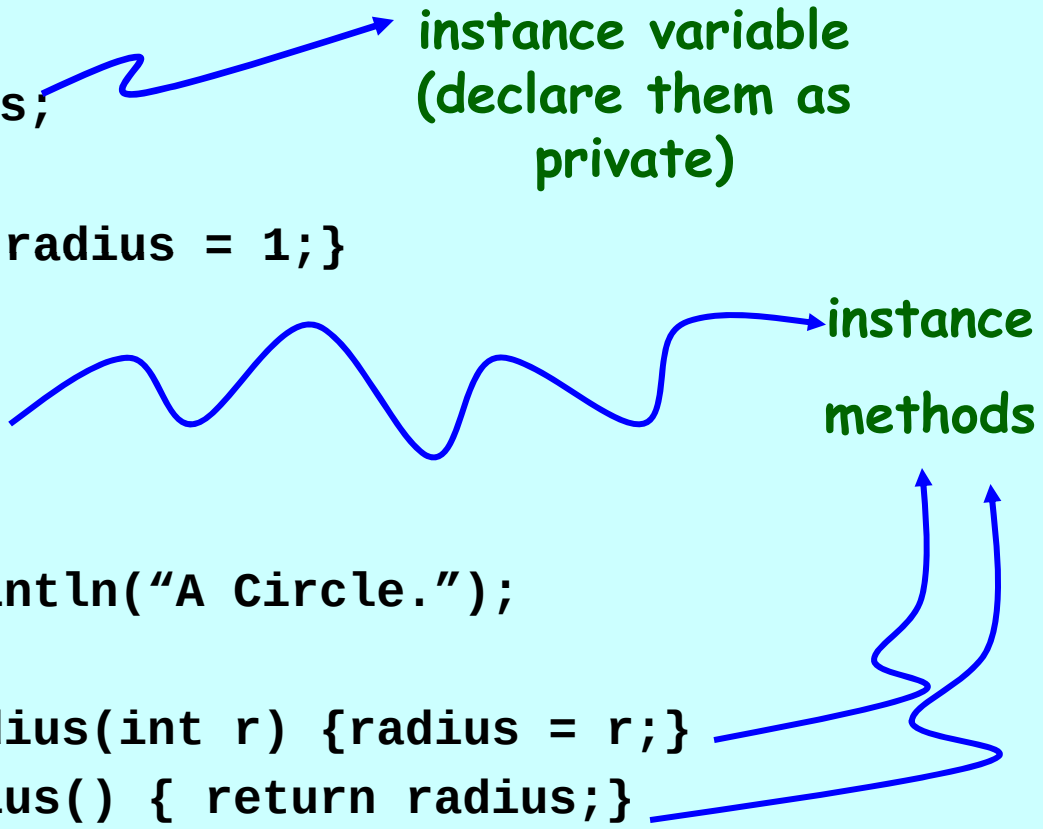
Encapsulation: data hiding

- a client class does not need to know how the object is actually represented, just need to know how it is to be used



private instance variables: enforces encapsulation

```
public class Circle
{
    private int radius;

    public Circle() { radius = 1;}


    public void dump()
    {
        System.out.println("A Circle.");
    }
    public void setRadius(int r) {radius = r;}
    public int getRadius() { return radius;}
}
```



CircleTest class (the client): uses Circle class

```
public class CircleTest
{
   public static void main(String[] args)
   {
      Circle c = new Circle();
      c.radius = 5;
      System.out.println(c.radius);
      c.setRadius( 5 );
      System.out.println(c.getRadius());

      c.dump();
   }
}
```

Packages

Packages

- Classes can be grouped into packages.
- two purpose:
  - supports encapsulation
  - naming

The Scanner class

```
import java.util.Scanner;
```

fully qualified name

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

        Scanner scan = new Scanner(System.in);
        int j = scan.nextInt();
        System.out.println( j );
    }
}
```

next(), nextLine()…

More on visibility modifiers

```
package games;


public class FreeCell
{
    int position;
    public FreeCell() {
        position = 9;
    }
}
```

fully qualified name:
games.FreeCell

```
package games;


public class Solitaire
{
    public static void main(String[] args)
    {
        FreeCell f = new FreeCell();
        System.out.println( f.position );
    }
}
```

Compile ??

position is a package protected variable in FreeCell class. Since TicTacToe class is not in games package, this code will not compile.

```
import games.*;

public class TicTacToe
{
    public static void main(String[] args)
    {
        FreeCell f = new FreeCell();
        System.out.println( f.position );
    }
}
```

Exercise

Do we need an import statement ?

```
public class Test
{
    public static void main(String[] args)
    {
        String s = new String( “Ottawa” );
    }
}
```

Answer:

No. This is because the String class is in java.lang package which is automatically imported in every Java program.

Static variables and Static Methods

Static class members

- **a static method is invoked through its class name**

  double d = Math.sqrt(49);

- **variables can be static as well**
- **static methods - class methods**

  static variables – class variables

Static class members

- **declared using static modifier**

```
private static int count;

public static int getCount() {…}
```

Static variables

```
public class Circle
{
    private double radius;
    private static int count;

    public Circle() {
        radius = 1.0;
        count++;
    }
    ……….
}
```

Static methods

- **static methods never use instance variable values.**

- **do not need to instantiate any objects in order to invoke static methods**

- **a static method means "behavior not dependent on an instance variable, so no instance is required. Just the class."**

- **can be invoked using class name**

```
e.g. int x = Math.round(42.2);
```

Difference between Static and Non-static methods

```
Song s = new Song("foo");
s.play();
```

```
public class Song {
  private String title;

  public Song(String t) {
    title = t;
  }

  public void play() {
   System.out.println(title);
  }
}
```

```
int x = Math.abs(345);
```

```
public class Math {
  public static int abs(int a) {
    //Returns the absolute
    //value of an int value
  }
}
```

A class can have both static and non-static methods.

Static methods: cannot use instance variables

```
public class Duck {
    private int size;

    public static void main(String[] args) {
      System.out.println("Size is: " + size);
    }
    public void setSize(int s) {size = s;}
    public int getSize() {return size;}
}
```

compile ??

Static methods: cannot use instance methods

```
public class Duck {
    private int size;

    public static void main(String[] args) {
      System.out.println("Size is: " + getSize());
    }
    public void setSize(int s) {size = s;}
    public int getSize() {return size;}
}
```

compile ??

Static variable: value is same for ALL instances of the class

```
public class Circle
{
    private double radius;
    private static int count = 0;
    public Circle() {
        radius = 1.0; count++;
    }
    public static int getCount() {return count;}
    public void setRadius(double d) {radius = d;}
    public double getRadius() {return radius;}
}
```

Static methods

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

        System.out.println(Circle.getCount());

        Circle c1 = new Circle();

        System.out.println(c1.getCount());
    }
}
```

Output:

0

1

Static methods

```
public class CircleTester {
    public static void main(String[] args){
        Circle c1 = new Circle();
        Circle c2 = new Circle();
        Circle c3 = new Circle();
        System.out.println(Circle.getCount());
    }
}
```

Output:

3

Instance variables: 1 per instance

Static variables: 1 per class

Static variable: initilization

- **static variables are initialized when a class is loaded.**

- **static variables in a class are initialized before any object of that class can be created**

- **static variables in a class are initialized before any static method of the class runs.**

- **Static variables get default values just like instance variables**
- **Static variables are accessed using class name**

Life and Death of an Object

- **Stack and Heap**

Stack and Heap

Stack

Method invocations and local variables live

Heap (Garbage collectible heap)

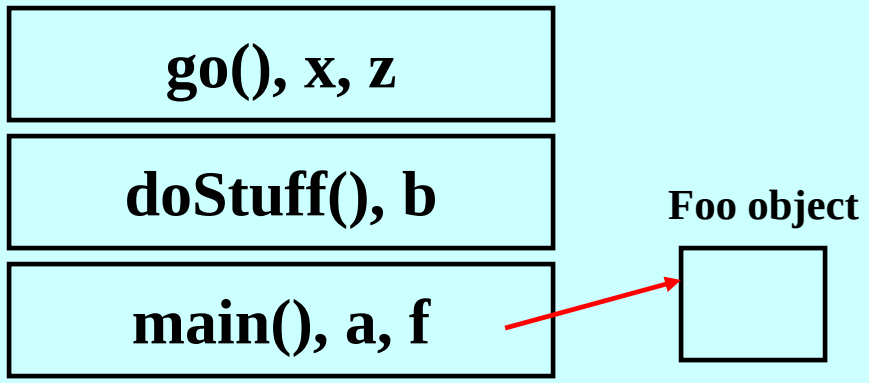
Where ALL objects live

Methods are stacked

```
public class Foo {
   public void doStuff() {
      boolean b = true;
      go( 4 );
   }
   public void go(int x) {
      int z = x + 24;
   }
}
```

```
public class Test {
  public static void main(String[] a) {
     Foo f = new Foo();
     f.doStuff();
}
```

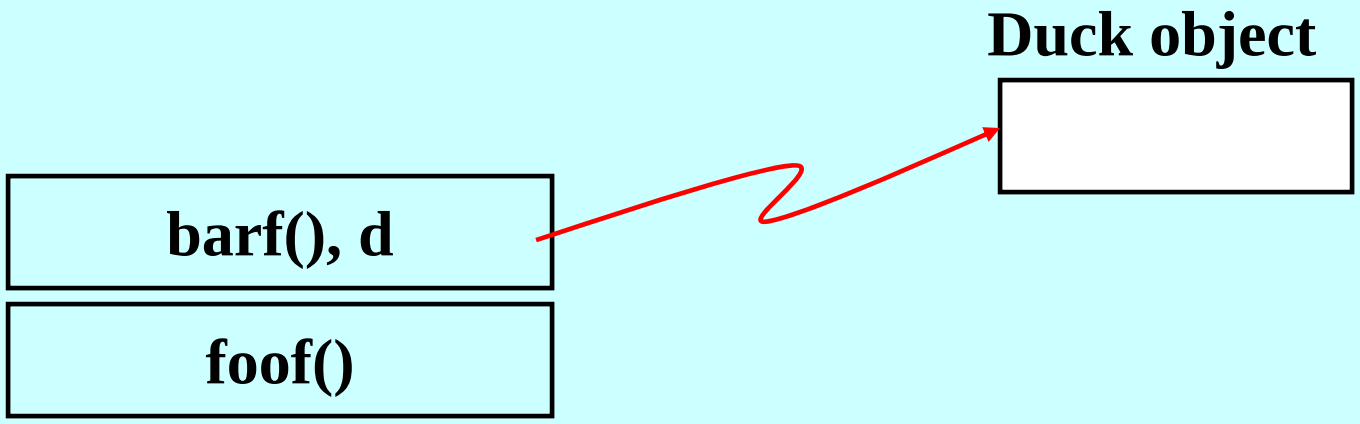
Stack frame: holds the state of the method including which line of code is executing and values of all local variables



Method at top of stack is currently running method

What about local variables that are objects ?

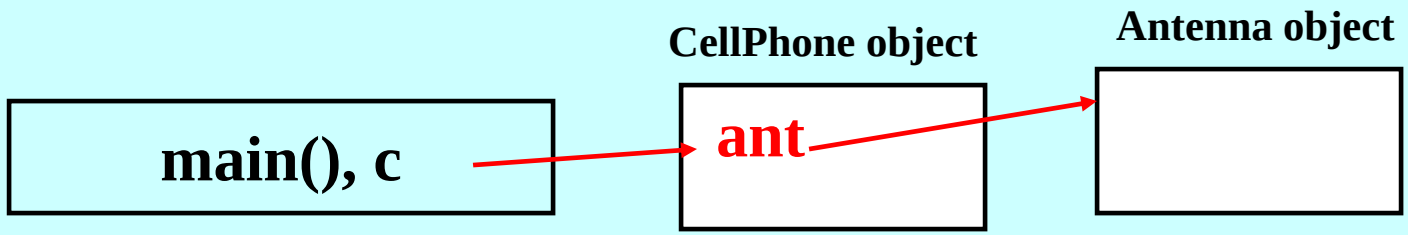
```
public class StackRef {
    public void foof() {
        barf();
    }
    public void barf() {
        Duck d = new Duck();
    }
}
```



Where do instance variables live ?

```
public class CellPhone {
  private Antenna ant;

  public CellPhone() {
    ant = new Antenna();
  }
}
```

```
public class Test {

 public static void main(String[] a) {

    CellPhone c = new CellPhone();
}
```



Comparing Variables

Comparing Characters

- **Follows Unicode Character set that defines an ordering of all possible characters**

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

```
boolean b = (‘a’ > ‘A’) ? true : false;
```

Comparing Strings

```
public class EqualityTest
{
  public static void main(String[] args)
   {
       String s1 = new String( "Maria" );
       String s2 = new String( "Maria" );
       if( s1 == s2 )
       {
          System.out.println( "Aliases of each other");
       }
   }
}
```

boolean expression evaluates to ??

Answer:

false

Not a way to check if two strings contain same characters !

Comparing Strings

Answer:

true

boolean expression evaluates to ??

```
public class EqualityTest
{
    public static void main(String[] args)
    {
        String s1 = new String( "Maria" );
        String s2 = new String( "Maria" );
        if( s1.equals(s2) )
        {
            System.out.println( "Both strings are same");
        }
    }
}
```

RIGHT way to check if two strings contain same characters

Comparing Strings

Equality test !

```
public class LexicographicComparison
{
   public static void main(String[] args)
   {
       String s1 = new String( "baker" );
       String s2 = new String( "baker" );
       if(  s1.compareTo(s2) == 0  )
       {
          System.out.println( "Both strings are same");
       }
   }
}
```

Comparing Strings

What is the boolean expression ?

```
public class LexicographicComparison
{
   public static void main(String[] args)
   {
       String s1 = new String( "baker" );
       String s2 = new String( "Baker" );
       if( s1.compareTo(s2) > 0 )
       {
          System.out.println( "s1 refers to a string that " +
                              "is lexicographically greater " +
                              "than the one s2 refers to." );
       }
   }
}
```

Comparing Strings

What is the boolean expression ?

```
public class LexicographicComparison
{
   public static void main(String[] args)
   {
       String s1 = new String( "Hello" );
       String s2 = new String( "hello" );
       if( s1.compareTo(s2) < 0 )
       {
          System.out.println( "s1 refers to a string that " +
                              "is lexicographically smaller " +
                              "than the one s2 refers to." );
       }
   }
}
```