

Module 1

Classes,

Module 1

- Classes: Classes in Java; Declaring a class; Class name; Super classes; Constructors; Creating instances of class; Inner classes.



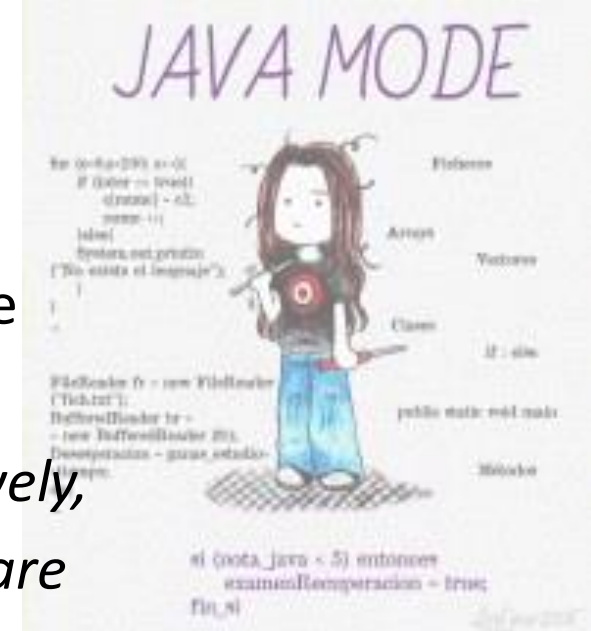
Class

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```



- a class is a *template* for an object,
- an object is an *instance* of a class.

- The data, or variables, defined within a class are called ***instance variables***.
 - *The code is contained within **methods**. Collectively, the methods and variables defined within a class are called **members of the class***
 - the general form of **a class does not specify a main() method.**
- Java classes do not need to have a main() method.
- You only specify one if that **class is the starting point for your program.**
 - Further, **applets don't require a main() method** at all.



JAVA MODE

```

for (var i=0; i<=10; i++)
{
    if (i%2 == 0)
    {
        console.log(i);
    }
    else
    {
        console.log("This is not a number");
    }
}

```

Variables

Arrays

Variables

Classes

if : else

public static void main

Methods

```

FileReader fr = new FileReader(
"tictac.txt");
BufferedReader br =
new BufferedReader(fr);
InputStreamReader isr = new InputStreamReader(
fr);

```

FileReader

BufferedReader

InputStreamReader

Class declaration

are stored in

```

// class declaration
class MyClass {
    // class body
}

```

entences

operation = true;

File s1

- ***C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately.***
- *This sometimes makes for very large .java files, since any class must be entirely defined in a single source file.*
- *This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain*

```
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        double vol;  
  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
  
        vol = mybox.width * mybox.height *  
            mybox.depth;  
  
        System.out.println("Volume is " + vol);  
    }  
}
```



Declaring Objects

Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object

`Box mybox = new Box();`

- *NOTE*
- *Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers.*
- ***correct. An object reference is similar to a memory pointer.***
- *The main difference—and the key to Java's **safety**—is that **you cannot manipulate references** as you can actual pointers.*
- *Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer*



Declaring an object of type Box



Statement

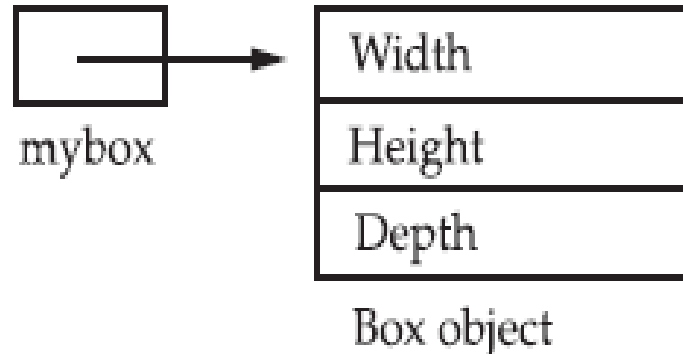
```
Box mybox;
```

Effect

Box
mybox

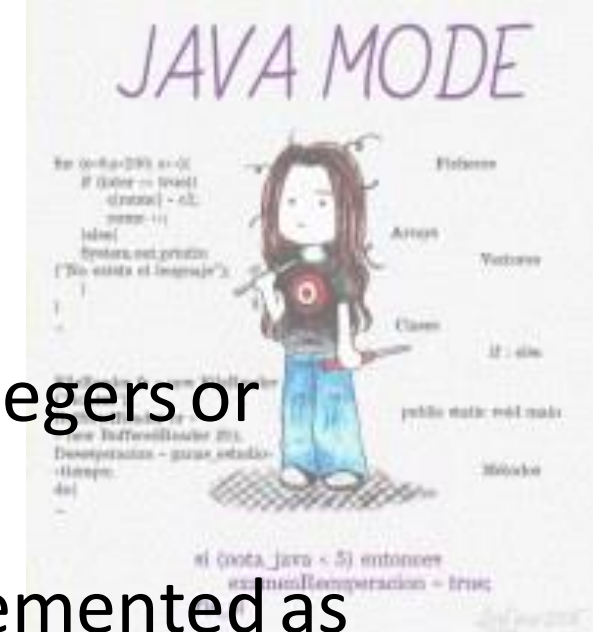
new allocates memory for an object during run time

```
mybox = new Box();
```




The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created.

- you do not need to use **new for** integers or characters.
- Java's primitive types are not implemented as objects. Rather, they are implemented as “normal” variables. This is done in the interest of efficiency.
- by not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently.



Java Mode

Java Mode is a programming language that can be used to create a logical relationship between



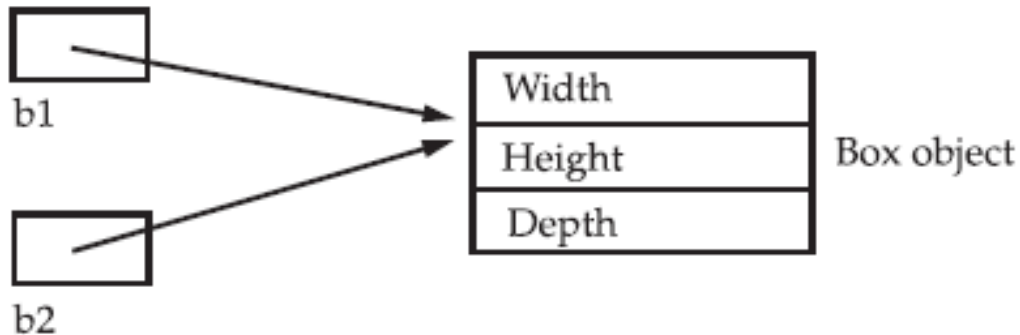
The background features a code editor with Java code snippets and various programming terms. The code includes comments like '/* Author: [Name] */', '/* Date: [Date] */', and '/* Version: [Version] */'. It also shows a class definition 'class Test {', a main method 'public static void main', and a try-catch block 'try {', 'catch (Exception e) {', '}'. The terms 'Variables', 'Arrays', 'Classes', 'If-else', and 'Methods' are scattered around the code.

- A class creates a new data type that can be used to create objects. That is, a class creates a **logical framework** that defines the relationship between its members.
- When you declare an object of a class, you are creating an **instance of that class**.
- Thus, a class is a **logical construct**. An object has **physical reality**. (That is, an object occupies space in memory.)

Assigning Object Reference Variables

```
Box b1 = new Box();
```

```
Box b2 = b1;
```



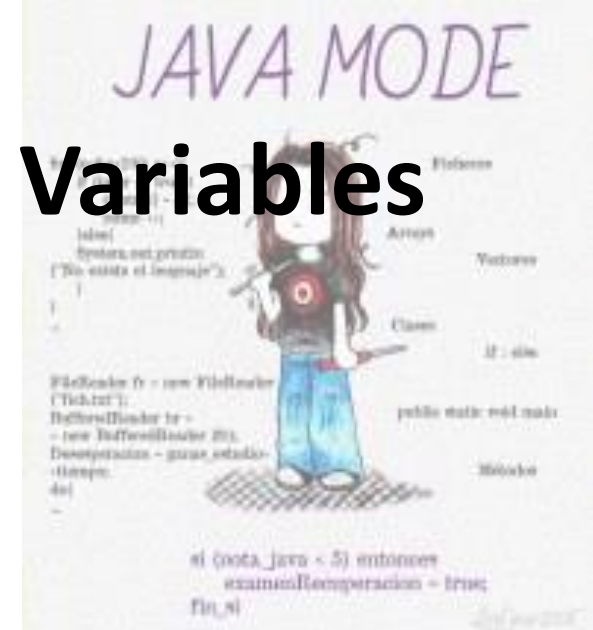
```
Box b1 = new Box();
```

```
Box b2 = b1;
```

```
// ...
```

```
b1 = null;
```

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.



Methods

- This is the general form of a method:

type name(parameter-list) {

```
// body of method
```

}

- Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

return value;



```
// display volume of a box
void volume(){
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
} }
```

```
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

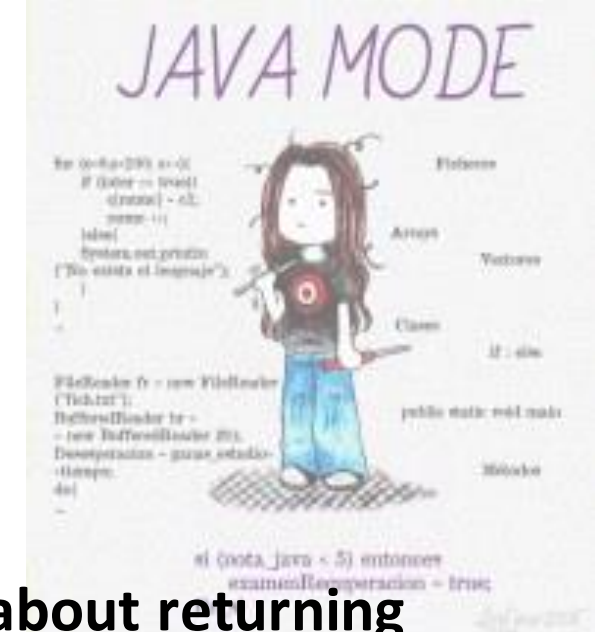
/* assign different values to mybox2's
```



```
// display volume of first box
mybox1.volume();

// display volume of second
box
mybox2.volume();
}
}
```

Returning Values

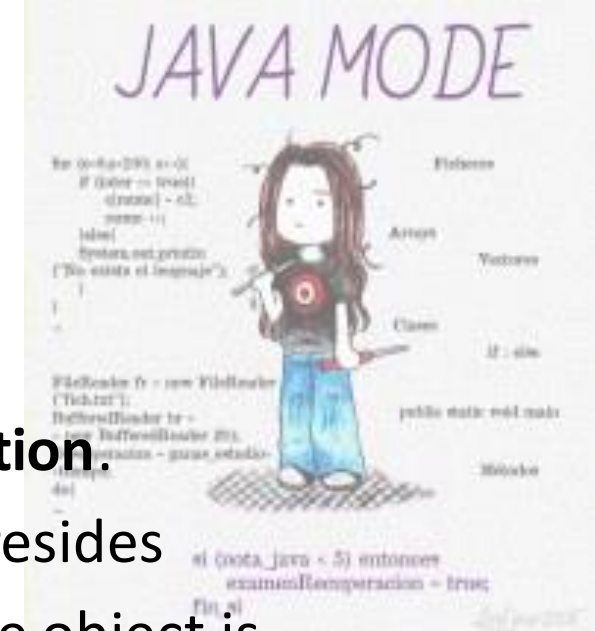


There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is boolean, you could not return an integer.
- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Constructors

- **initializes an object immediately upon creation.**
- **has the same name as the class** in which it resides
- is **automatically called immediately** after the object is created, before the **new operator completes**.
- they have **no return type**, not even void. This is because the implicit return type of a class' constructor is the class type itself.
- job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, **usable object immediately**.



```
class-var = new classname( );  
Box mybox1 = new Box();
```



- new Box() is calling the Box() constructor. When you do not explicitly define a constructor for a class, then **Java creates a default constructor for the class.**
- The default constructor **automatically initializes all instance variables to zero.**
- Once you define your own constructor, the default constructor is no longer used.


```
class Box {
    double width;
    double height;
    double depth;
```

```
// This is the constructor for Box.
```

```
Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
}
```

```
// compute and return volume
double volume() {
    return width * height * depth;
}
}
```



Parameterized Constructors

```
class Box {  
    double width;  
    double height;  
    double depth;
```

```
// This is the constructor for Box.  
Box(double w, double h, double d) {  
    width = w;  
    height = h;  
    depth = d;  
}
```

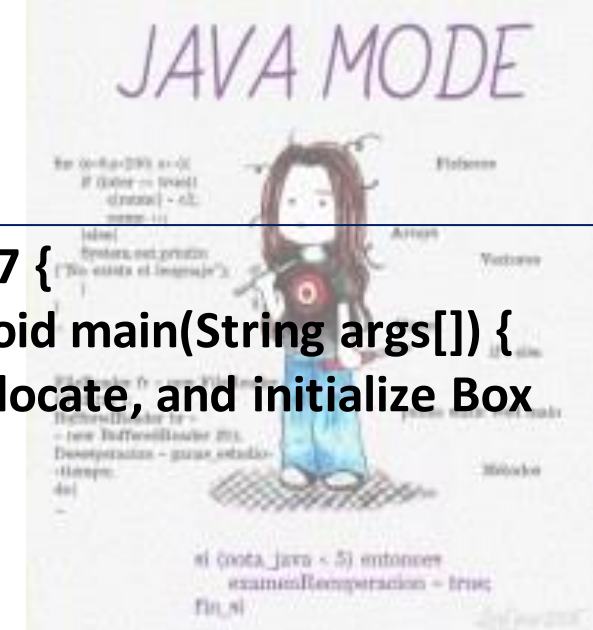
```
// compute and return volume  
double volume() {  
    return width * height * depth;  
} }
```

```
class BoxDemo7 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box  
        objects
```

```
Box mybox1 = new Box(10, 20, 15);  
Box mybox2 = new Box(3, 6, 9);
```

```
double vol;  
vol = mybox1.volume();  
System.out.println("Volume is " + vol);
```

```
vol = mybox2.volume();  
System.out.println("Volume is " + vol);  
} }
```

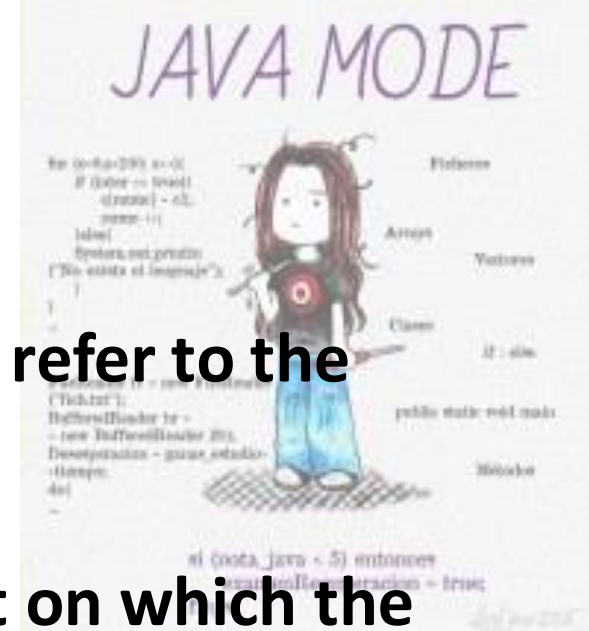


The this Keyword

- this can be used inside any method to refer to the *current object*.
- this is always a reference to the object on which the method was invoked

```
// A redundant use of this.
```

```
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```



JAVA MODE

for (int i=0; i<=10; i++)
 {
 if (i%2 == 0) {
 System.out.println("Even");
 } else {
 System.out.println("Odd");
 }
 }
 }

Variables
 Arrays
 Vectors
 Classes
 if : else
 public static void main
 Methods
 si (costa_java < 5) entonces
 sistemaReservacion = true;

Normal with the names

e as an instance

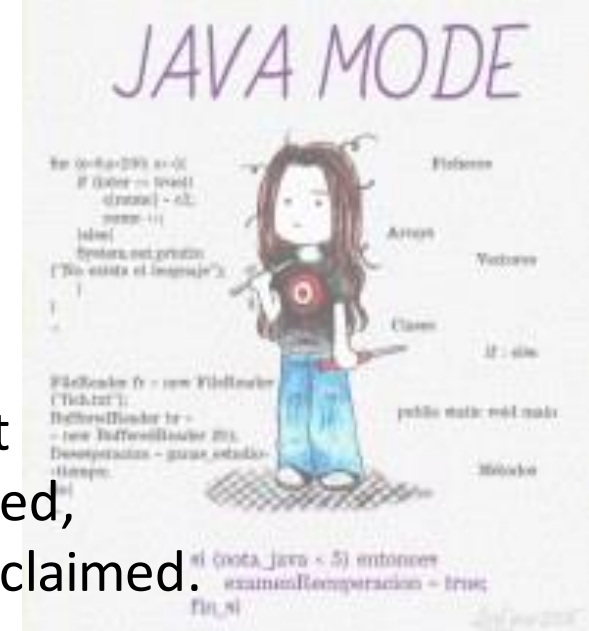
- `//` Use this to resolve name-space collisions.

```
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

Garbage Collection

handles deallocation for you automatically

- The technique that accomplishes this is called *garbage collection*.
- *It works like this: **when no references** to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.*
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.
- different Java run-time implementations will take varying approaches to garbage collection



The finalize() Method

- an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle then you might want to make sure these resources are freed before an object is destroyed
- Java provides a mechanism called *finalization*.
- *By using finalization, you can* define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.



- **Java run time** calls finalize method whenever it is about to recycle an object of that class.
- Inside the **finalize()** method, you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. **Right before an asset is freed**, the Java run time **calls the finalize() method on the object**.

The finalize() method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- the keyword **protected** is a specifier that prevents access to `finalize()` by code defined outside its class.
- It is important to understand that `finalize()` is only called just prior to garbage collection.



- It is not called when an object goes out-of-scope
- This means that you cannot know when—or even if—`finalize()` will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on `finalize()` for normal program operation.

Lessons Learned

JAVA MODE

```
for (i=0;i<100;i++) {
    if (i%10 == 10) {
        strnum() + "L";
        num = i + 1;
    } else {
        System.out.println(
            "No estado el lenguaje");
    }
}
```

Filesystem

Arrows

Variables

Class

if : else

public static void main

Block

```
FileReader fr = new FileReader(
    "Ticket.txt");
BufferedReader br =
    new BufferedReader(fr);
Desempeñador = game.start(
    0);
do {
    ...
} while (true);

si (nota_java < 5) entonces
    examenRecomperacion = true;
fin_si
```

Andrés J. 2018