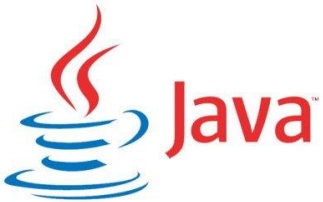
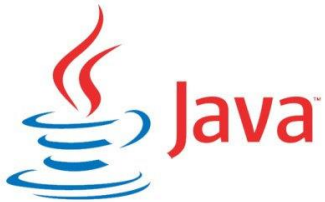
Paradigmas y Lenguajes III

UNIDAD II - Programación Orientada a Objetos



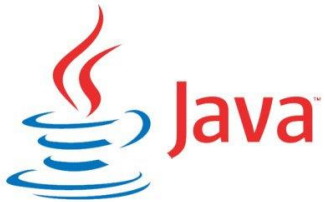
2.1 Introducción a JAVA

- JAVA originalmente fue creado por Sun Microsystems en 1995
- Sun Microsystems fue adquirido por Oracle en 2009
- JAVA tiene características de C y C++ que hacen que sea fácil de entender
- Los Programa en JAVA son independiente la plataforma

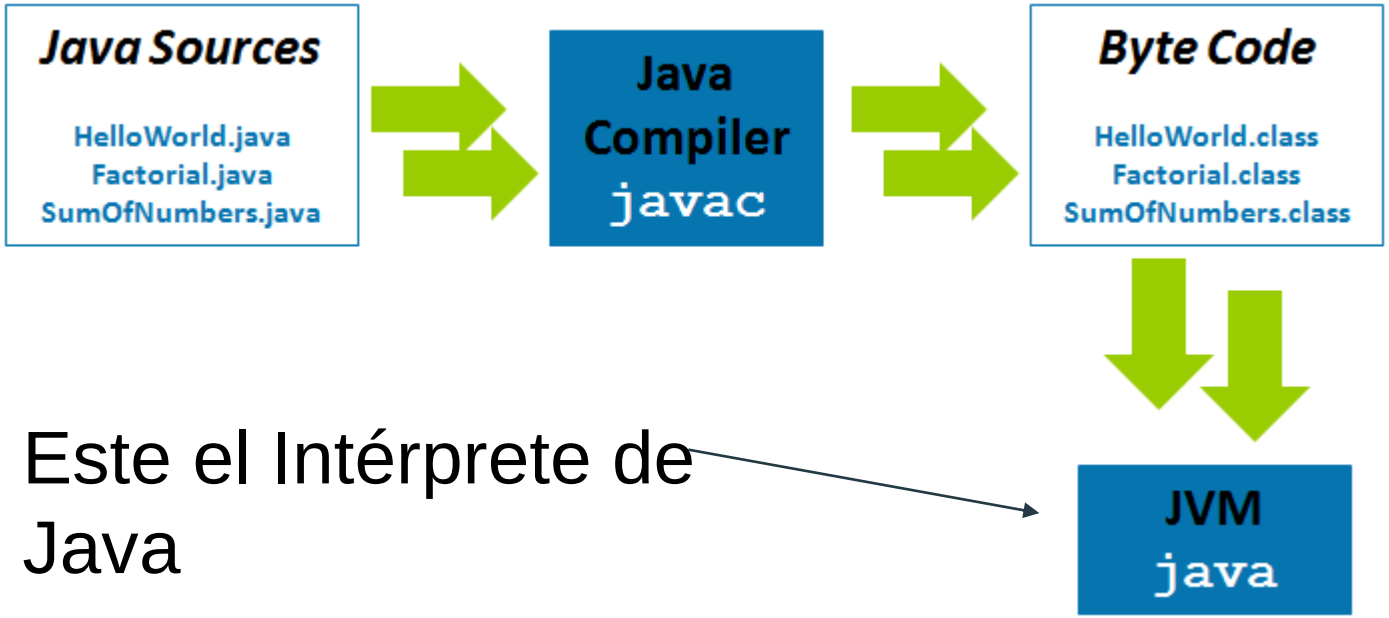


2.1 Java - Características

- Orientado a Objetos
- JAVA tiene características de C y C++
- Portable
- 3 Billones de Dispositivos están corriendo JAVA



2.1 INTRODUCCION JAVA



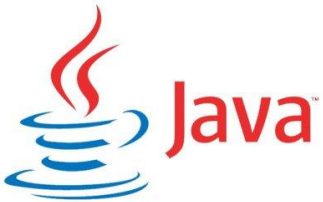
Este el Intérprete de Java

Java

2.1 IDEs

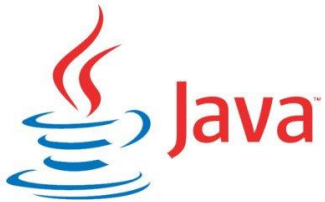
Son entornos de desarrollo integrado que ofrecen herramientas que ayudan en el proceso de desarrollo del software, como editores, debuggers para localizar errores lógicos (que pueden causar que el programa no se ejecute correctamente). Algunos de los IDEs más populares para Java son:

- Eclipse (www.eclipse.org)
- Netbeans (www.netbeans.org)
- IntelliJ IDEA (www.jetbrains.com)



2.1 Paquetes

Un paquete en JAVA es un mecanismo para proveer NAMESPACE, es un espacio dentro del cual los nombres son únicos pero fuera de este puede no serlo. Para calificar una construcción como única, se debe escribir o mencionar su NAMESPACE completo.

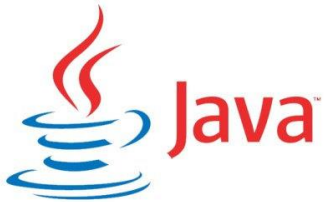


2.1 Paquetes

- Para definir una paquete se usa la palabra clave ***package***

```
package orgType.orgName.appName.compName;
```

  - ***orgType*** ies el tipo de organización, como com, org, or net.
  - ***orgName*** es el nombre del dominio de la organización, como oracle, o ibm.
  - ***appName*** es el nombre de la aplicación abreviado
  - ***compName*** es el nombre del componente

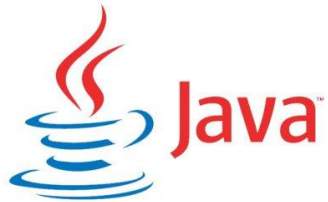


2.2 Mi Primer Programa en JAVA

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1
5  {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9        System.out.println("Welcome to Java Programming!");
10    } // end method main
11 } // end class Welcome1
```

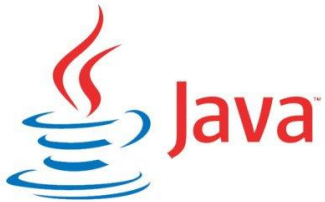
Salida del Programa

```
Welcome to Java Programming!
```



2.2 Otro Programa

```
1  // Fig. 2.7: Addition.java
2  // Addition program that inputs two numbers then displays their sum.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition
6  {
7     // main method begins execution of Java application
8     public static void main(String[] args)
9     {
10       // create a Scanner to obtain input from the command window
11       Scanner input = new Scanner(System.in);
12
13       int number1; // first number to add
14       int number2; // second number to add
15       int sum; // sum of number1 and number2
16
17       System.out.print("Enter first integer: "); // prompt
18       number1 = input.nextInt(); // read first number from user
19
20       System.out.print("Enter second integer: "); // prompt
21       number2 = input.nextInt(); // read second number from user
22
23       sum = number1 + number2; // add numbers, then store total in sum
24
25       System.out.printf("Sum is %d%n", sum); // display sum
26    } // end method main
27 } // end class Addition
```



2.3 Variables de Instancia - Métodos getter y setter

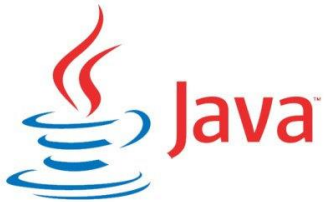
Recordar que Java es case-sensitive

Declaración de una Clase que se almacenará en una archivo con el mismo nombre

```
1  // Fig. 3.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account
6  {
7     private String name; // instance variable
8
9     // method to set the name in the object
10    public void setName(String name)
11    {
12       this.name = name; // store the name
13    }
14
15    // method to retrieve the name from the object
16    public String getName()
17    {
18       return name; // return value of name to caller
19    }
20 } // end class Account
```

Variables que pertenecen a la instancia de una Clase

Estos métodos me permiten obtener y establecer una valor a un atributo

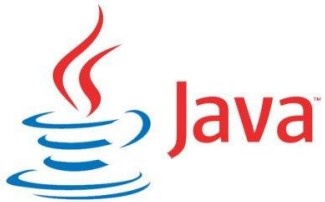


2.3 Tipos Primitivos VS Tipos de Referencia

Tipos Primitivos

Este tipo de variables son inicializadas por default.

Type	Size	Default value	Range of values
boolean	n/a	false	true or false
byte	8 bits	0	-128 to 127
char	16 bits	(unsigned)	\u0000' \u0000' to \uffff' or 0 to 65535
short	16 bits	0	-32768 to 32767
int	32 bits	0	-2147483648 to 2147483647
long	64 bits	0	-9223372036854775808 to 9223372036854775807
float	32 bits	0.0	1.17549435e-38 to 3.4028235e+38
double	64 bits	0.0	4.9e-324 to 1.7976931348623157e+308



2.3 Tipos Primitivos VS Tipos de Referencia

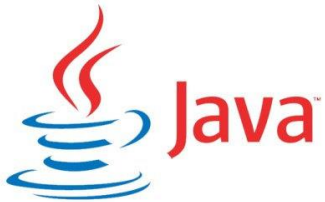
Tipos Referencias

- Todos los que no son tipo primitivos son tipo referencia.
- Son Clases, que especifican los tipos de objetos

Si no esta explicitamente inicializado, se inicializa por defecto con el valor ***null***

```
Scanner input = new Scanner(System.in);
```

Se crea un objeto de la clase Scanner y se asigna a la variable ***input*** una referencia a un ***objeto Scanner***.

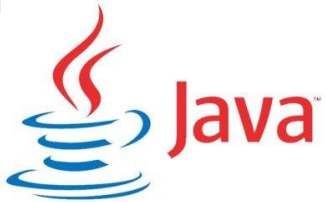


2.3 Métodos

- **Aquí tenemos la definición de un método**

```
public Integer calcularSuma(Integer numeroA, Integer numeroB)
{
    //Acá se define como hacer el cálculo
}
```

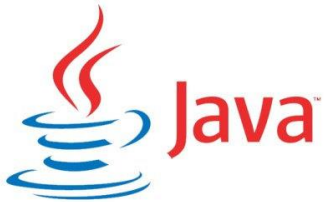
1. Modificadores — como `public`, `private`, y otros que veremos luego.
2. El tipo de retorno — el tipo de dato que el método debe retornar, o `void` si el método no retorna ningún valor.
3. La lista de parámetros va entre paréntesis, especificando el tipo de datos y el nombre del mismo.



2.3 Sobrecarga de Métodos

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Esto significa que los métodos dentro de una clase pueden tener el mismo nombre, pero si tienen diferentes listas de parámetros



2.3 Constructores

```
1  // Fig. 3.5: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account
5  {
6     private String name; // instance variable
7
8     // constructor initializes name with parameter name
9     public Account(String name) // constructor name is class name
10    {
11       this.name = name;
12    }
13
14    // method to set the name
15    public void setName(String name)
16    {
17       this.name = name;
18    }
19
20    // method to retrieve the name
21    public String getName()
22    {
23       return name;
24    }
25 } // end class Account
```

- Un constructor debería tener el mismo nombre de la clase
- Un constructor especifica la lista de parámetros que el constructor requiere para hacer esta tarea.
- Constructores no pueden retornar valores
- Los constructores no se pueden heredar pero si se pueden invocar desde una clase hija. (super)

Métodos Getters y Setters

- Cuando todos los campos de una clase son privados
- Un ***atributo público*** de una instancia puede ser leída y escrita por cualquier método que tenga la referencia al objeto
- Un ***atributo privado*** de una instancia, solo puede ser accedido mediante el ***método get*** correspondiente

Composición

- Una clase tiene ***referencias a objetos*** de otras clases como miembros
- Esto lo podemos encontrar de dos formas:
    - Composición o Agregación
    - Tiene una relación con
- Ejemplo
    - Auto tiene:
        - ruedas
        - pedal de frenos
        - pedal de acelerador

Ejemplo: Class Empleado (Manos a la obra)

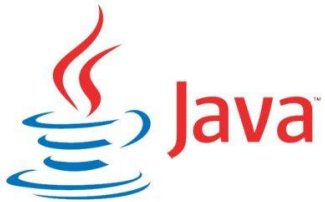
```
public class Empleado
{
 private String nombres;
 private String apellido;
 private Direccion direccion;
 private AreaTrabajo areaTrabajo;
}
```

2.3 Constantes

```
public class Account{

        private String name;
        public static final double NOMBRE_CONSTANTE =
20;

}
```

Indica que una variable, método o clase no se va a modificar.

El modificador static no sirve para crear constantes, sino para crear miembros que pertenecen a la clase, y no a una instancia de la clase. Esto implica, entre otras cosas, que no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos.



2.4 Casting de Clases

¿ Que vemos aquí ?

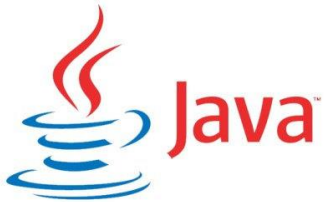
```
Object aSentenceObject = "This is just a regular sentence";
String aSentenceString = (String)aSentenceObject;
////////////////////////////////////////////////////
```

Desde el tipo ***Object*** que es un tipo muy amplio para una variable. Acá estamos haciendo **"downcasting"** para que la variable se convierta en tipo **String.**

```
String aSentenceString = "This is just another regular sentence";
Object aSentenceObject = (Object)aSentenceString;
```

¿ Que vemos aquí ?

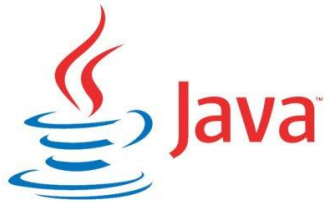
Aquí tomamos una variable de un tipo más específico(**String**) y la casteamos a un tipo de variable más angenerico



2.5 Operadores - El operador instanceof

```
public class Test
{
 public static void main(String[] args)
    {
     Test t= new Test();
     System.out.println(t instanceof Test);
      }
}
```

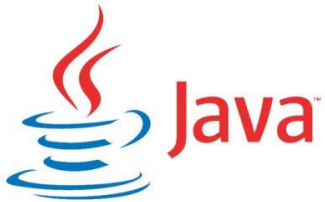
En JAVA el operador ***instanceof*** es usado para chequear el tipo de un objeto en tiempo de ejecución. Este es un recurso a través del cual en nuestro programa podemos obtener información acerca de objeto



2.6 Pasando información a un método o un constructor.

```
public double computePayment(
                double loanAmt,
                double rate,
                double futureValue,
                int numPeriods) {

    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
                    - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator) - ((futureValue * partial1) / denomina
    return answer;
}
```

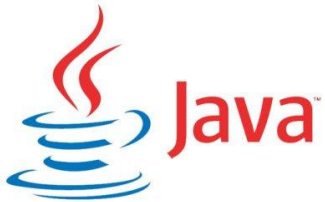
Parámetros se refiere a la lista de variables en la declaración de un método..



2.6 Objetos

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declare and create a point object and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        // display rectOne's width, height, and area
        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        // set rectTwo's position
        rectTwo.origin = originOne;

        // display rectTwo's position
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

        // move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}
```

Aqui un pequeño programa, llamado ***CreateObjectDemo***, que crea 3 objetos: un objeto Point y dos objetos Rectangle.

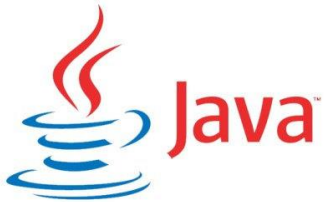


2.6 Objetos

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

- Lo que está en negrita, son las ***declaraciones*** de todas las variables, que es el tipo de Objeto a crear mas el nombre de la variable.
- ***Inicialización*** la palabra clave ***new*** en JAVA es un operador que crea un objeto, que es seguido por la llamada a un constructor, que inicializa un objeto-

```
//Aquí tenemos la clase Punto
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

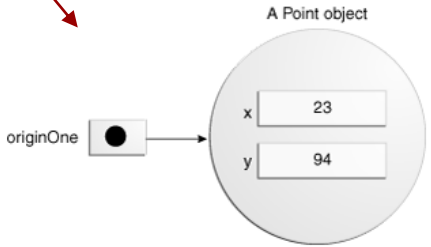


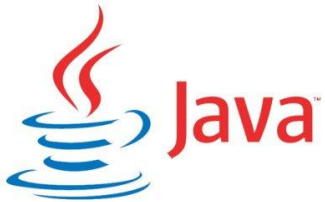
2.6 Objetos

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

```
//Aquí tenemos la clase Punto
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

- Lo que está en negrita, son las ***declaraciones*** de todas las variables, que es el tipo de Objeto a crear mas el nombre de la variable.
- ***Inicialización*** la palabra clave ***new*** en JAVA es un operador que crea un objeto, que es seguido por la llamada a un constructor, que inicializa un objeto-



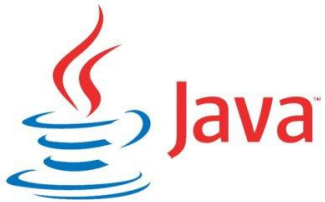


2.6 Objetos

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
```

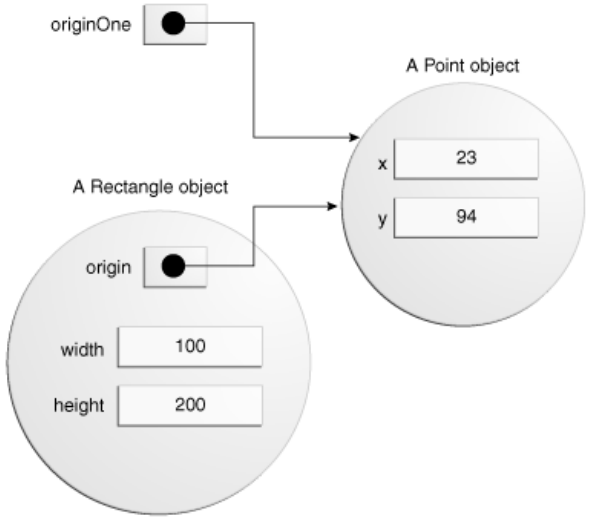
- Aquí tenemos las clase **Rectangle,** que nos estaba faltando

¿Que tenemos aqui ?



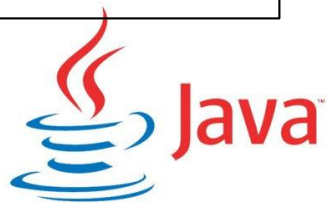
2.6 Objetos

```
Point originOne = new Point(23, 94);

Rectangle rectOne = new Rectangle(originOne, 100,
200);
```



- Si miramos el constructor ¿QUE ENCONTRAMOS ?

```
Rectangle rectTwo = new Rectangle(50, 100);

Rectangle rect = new Rectangle();
```



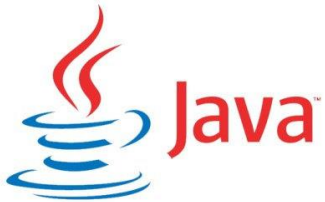
2.7 The Garbage Collector

- Algunos lenguajes de programación requieren que vayas haciendo el trackeo, de alguna manera acerca de los objetos que se van creando y que vos explícitamente deberías destruir.
- La plataforma de JAVA te permite crear los objetos que tu quieras (con un límite por supuesto xD).
- No tienes que preocuparte por destruir estos objetos
- El entorno de ejecución de JAVA borra los objetos cuando esté determina que ya no están siendo usados. Este procedimiento es el denominado ***garbage collection***

2.8 Uso de la palabra clave ***this***

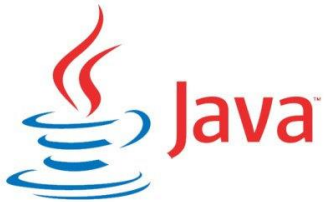
```
//Aquí tenemos la clase Punto
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
//Aquí tenemos la clase Punto
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```



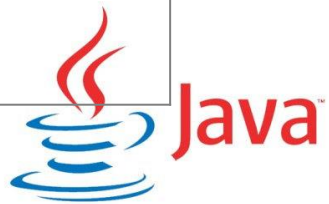
2.8 Uso de ***this*** con el ***constructor***

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

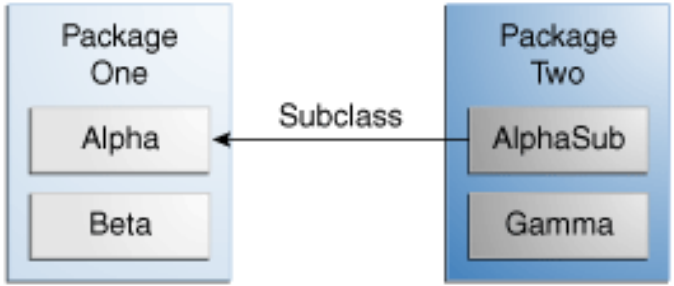


2.9 El acceso a los miembros de la clase

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

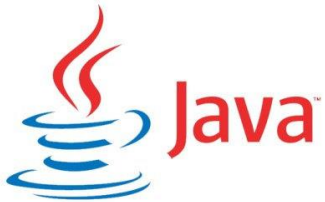


2.9 El acceso a los miembros de la clase



VISIBILIDAD

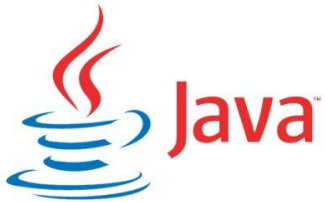
Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N



2.10 Interfaces

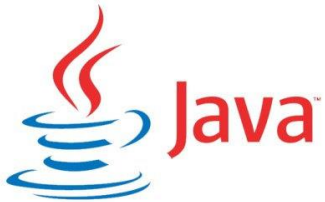
- En JAVA una interfaz es un tipo de referencia, similar a las clases, que puede solo contener constantes, firmas de métodos y métodos estáticos.

- El cuerpo de métodos, únicamente existe en los métodos estáticos.



2.10 Interfaces

- Las interfaces no pueden ser instanciadas, solo pueden ser implementadas por otras clases o extendidas por otras interfaces

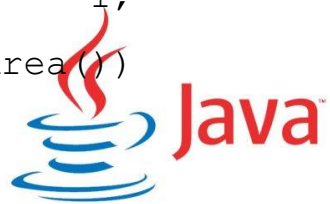


2.10 Interfaces - Definir una Interfaz

```
public interface Relatable {
  // this (object calling isLargerThan)
     // and other must be instances of
     // the same class returns 1, 0, -1
     // if this is greater than,
     // equal to, or less than other
     public int isLargerThan(Relatable other);
}
```

2.10 Interfaces - Implementar una Interfaz

```
public class RectanglePlus
    implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1; }}
```

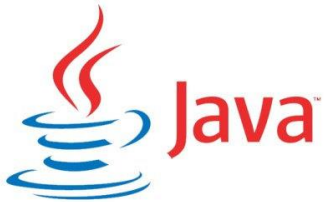


2.11 Herencia

- Una clases derivada o desde cualquier otra clase es llamada sub-clase
- La clase desde que la sub-clase es derivada es llamada super-clase (clase padre o clase base).
- Exceptuando ***Object,*** todas las demás clases derivan de esta o de alguna otra que está bajo esta misma jerarquía.

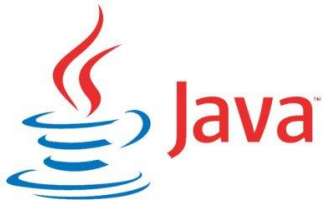
2.11 Herencia

- JAVA solamente soporta herencia simple, que significa que sólo puede heredar de una sola **clase**
- JAVA no soporta herencia múltiple
- Los constructores de la clase no se heredan
- Los constructores de la clase padre se deben llamar desde la clase hija



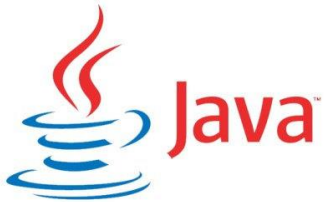
2.11 Herencia

- El constructor de la clase Object, no necesita ser llamado
- La palabra clave **super()** permite llamar al constructor de la clase padre

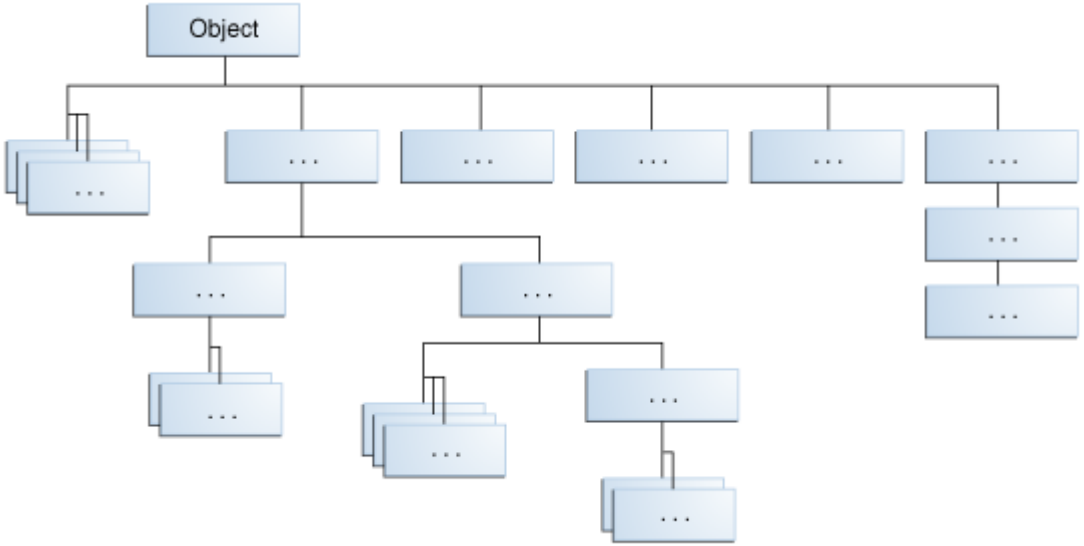


2.12 Sobreescritura

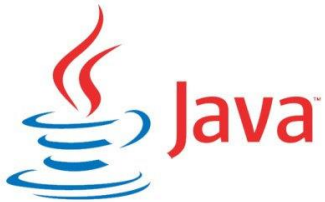
- **@Override** Annotation, significa que la declaración del método que le sigue debe sobreescribir el método existente de la clase padre
- La sobreescritura debe tener la misma firma que el metodo de la clase padre



2.13 Jerarquía de Clases en JAVA



La clase **Object**, definida en el package **java.lang**, define e implementa el comportamiento común a todas las clases, incluyendo las que nosotros definimos.



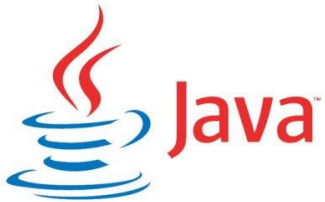
2.13 Jerarquía de Clases en JAVA

Desde la clase **Object**, directa o indirectamente se heredan 11 metodos:

equals	compara si dos objetos son iguales
hashCode	son valores enteros usados para acelerar el almacenamiento y recuperar la información almacenada
toString	devuelve un string que representa el objeto
wait,notify,notifyAll	estos métodos están relacionados con hilos múltiples

2.13 Jerarquía de Clases en JAVA

getClass	En JAVA cada objeto conoce su tipo en tiempo de ejecución. Este método retorna un objeto de la clase **Class** (paquete java.lang) que contiene información acerca del tipo objeto
finalize	Es un método llamado por el garbage collector cuando el collector determinar que no hay más referencias al objeto
clone	crea y retorna una copia del objeto que lo llama



2.14 Abstracción en Java

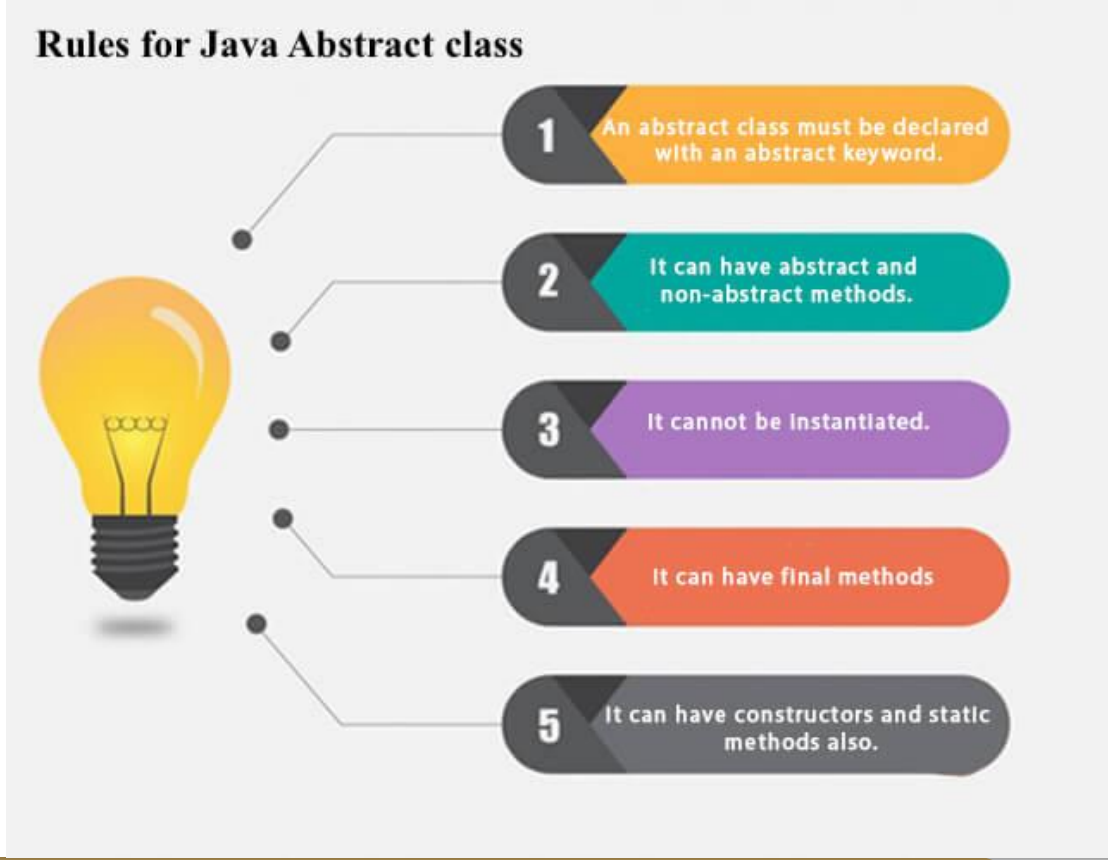
Abstracción es un proceso que oculta los detalles de implementación y muestra únicamente la funcionalidad al usuario.

2.14 Maneras de lograr la Abstracción

Hay dos maneras de lograr la abstracción en JAVA:

- **Clases Abstractas**
- **Interfaces**

2.14 Maneras de lograr la Abstracción



2.14 Maneras de lograr la Abstracción

Declaración de una Clase Abstracta

```
public abstract class Persona {

//body

}
```

2.14 Maneras de lograr la Abstracción

Un método que es declarado como abstracto y no tiene implementación es conocido o llamado **método abstracto**

```
public    abstract void imprimirEstado();
```

2.15 Ejemplo

```
public abstract class Bike{

                    abstract void run();

}
```

2.15 Ejemplo

```
class Honda4 extends Bike{
      void run(){
    System.out.println("Andando a 80 KM");
    }
     }
```

2.15 Ejemplo

```
class Honda4Test{

     public static void main(String
args[]){

   Bike obj = new Honda4();

   obj.run();

     }

     }
```

2.15 Otro Ejemplo

```
public abstract class Banco{

        public abstract double
tasaDeInteres();

}

}
```

2.15 Otro Ejemplo

```
    public class BancoPatagonia extends
Banco{

        public double tasaDeInteres(){

            return 0.45;

      }

        }

    }
```

2.15 Otro Ejemplo

```
public class BancoNacion extends Banco{

    public double tasaDeInteres(){

        return 0.35;

    }

  }

}
```

2.15 Otro Ejemplo

```
public class BancoTest{

        Banco b;

    b = new BancoPatagonia();

System.out.println("Tasa de
Interes:"+b.tasaDeInteres()+" %");
```

2.15 Otro Ejemplo

```
        b = new BancoNacion();

System.out.println("Tasa de
Interes:"+b.tasaDeInteres()+" %");


    }
```

2.15 Clases Abstractas vs Interfaces

Potencialmente deberíamos usar una clase abstracta en estas situaciones:

- Compartir codigo entre clases estrechamente relacionadas
- Cuando se espera que las clases que extiendan su clase abstracta tengan muchos métodos o campos comunes

2.15 Clases Abstractas vs Interfaces

Potencialmente deberíamos usar una interfaz cuando:

- Cuando se espera que clases que no están relacionadas implementen estos métodos
- Desea especificar el comportamiento de un tipo de datos en particular, pero no le preocupa quién implementa su comportamiento.

2.15 Clases Abstractas vs Interfaces

Potencialmente deberíamos usar una interfaz cuando:

- Desea potenciar la característica de herencia múltiple

2.16 Métodos Finales

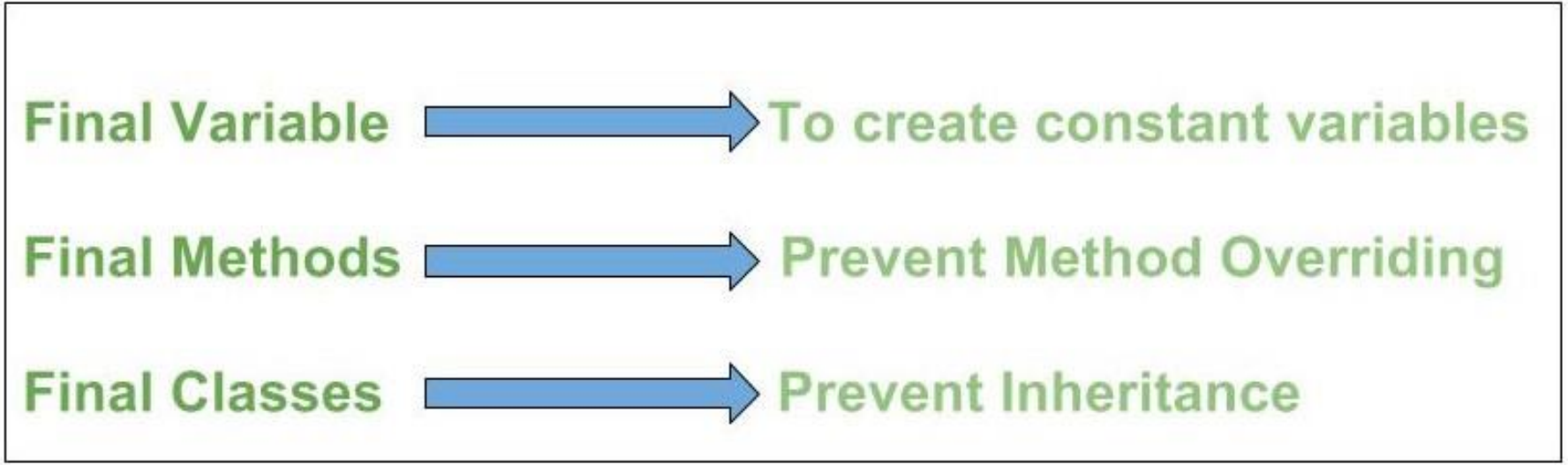
Un método final en un clase padre no puede ser sobreescrito en una clase hija

```
public final void metodo1()
{
    System.out.print("Hola");
}
```

2.16 Clases Finales

Una clase final es una clase que no puede ser extendida, es decir ninguna otra clase puede ser hija de ésta, será entonces una clase única y que no tendrá herencia o herederos al no poder ser extendida.

```
public final class ClaseMadre

{}
```


Final Variable
To create constant variables
Final Methods
Prevent Method Overriding
Final Classes
Prevent Inheritance

2.17 Colecciones Genéricas