Programación Avanzada

Ing. Orlando Vasquez

PROGRAMACION ORIENTADA A OBJETOS

Pontificia Universidad
JAVERIANA
Colombia

Sesión

Sintaxis Básica
Conceptos de Objetos

- Clases
- Objetos, Atributos
- Métodos
- Métodos de acceso
- Paquetes de clases: creación y visibilidad
- Visibilidad de clases: pública
- Visibilidad Atributos: pública, privada
- Visibilidad de Métodos: pública, privada
- Encapsulamiento de Atributos

Método main

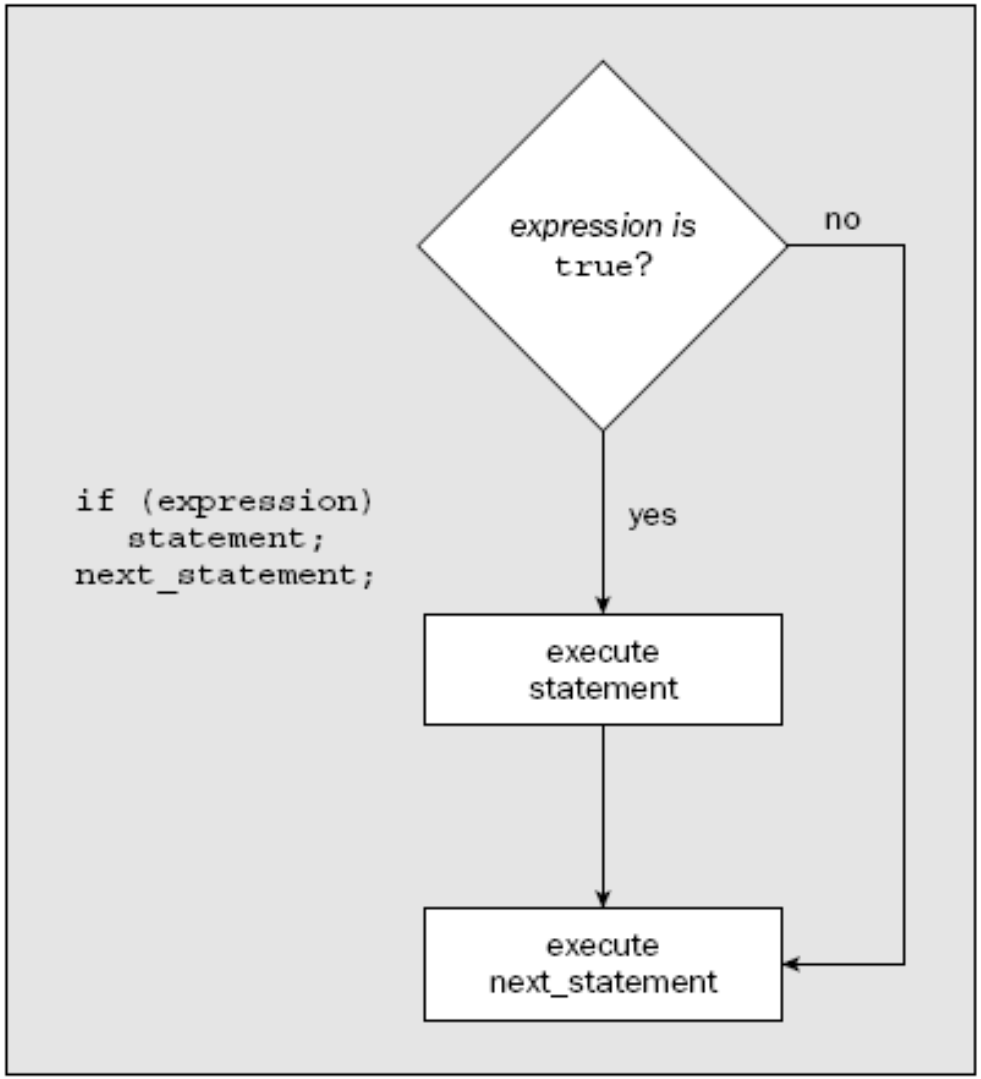
```
public static void main(String[] args) {
  byte value = 1;
  for (int i=0; i<8 ; i++) {
    value *= 2;
    System.out.println("Value is now " + value);
  }
}
```

Sentencias de control

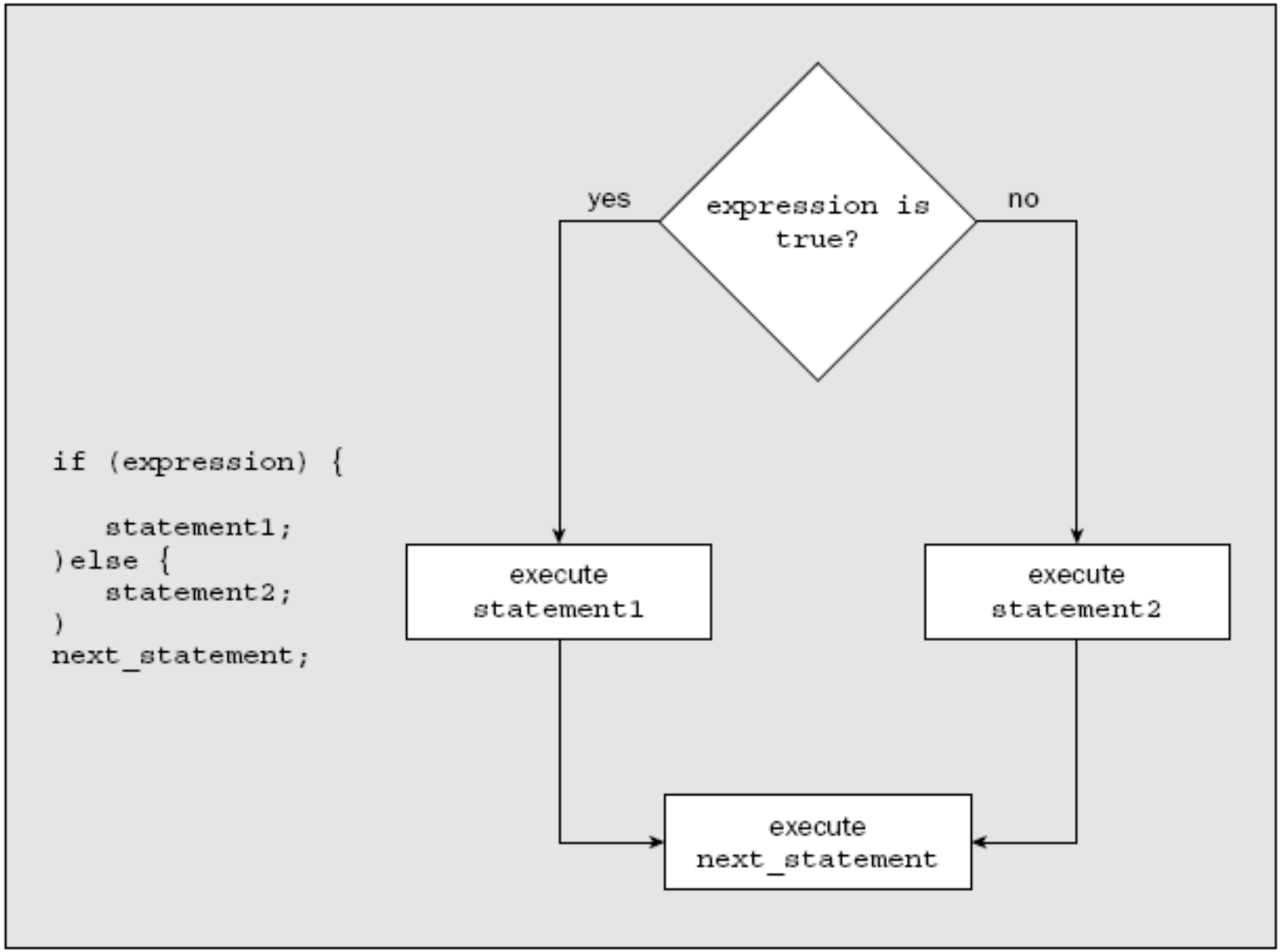
- Condicionales
  - Si se cumple una condición, es decir, si la condición es verdadera, entonces ejecute una serie de instrucciones.
- Ciclos
  - Repita una serie de instrucciones hasta que la condición sea falsa.

Condicionales

```
if(expression)
  statement;
```



Condicionales (else clause)



Condicionales

```
if(expr){
....
}
if(expr)
   sentencia;

if(expr){
....
} else {
....
}
```

Condicionales

```
if (x > y){
mayor = x;
}
else{
mayor = y;
}
```

```
if (x > y && x > z){
mayor = x;
}
else{
if (y > x && y > z){
    mayor = y;
}
else{
    mayor = z;
}
}
```

Condicionales

```
Switch (num){
   case num1:...break;
   case num2:...break;
   default:...break;
}
```

Condicionales

```
switch (opcion) {
case 1:
        result =  x + y;
        break;
case 2:
        result = x – y;
        break;
default:
        result = 0;
}
```

Ejemplo

```
public void edadEsCritica() {
  switch (edad) {
     case 0:
          System.out.println ("Acaba de nacer hace poco. No ha cumplido el año");
          break;
      case 18: System.out.println ("Está justo en la mayoría de edad"); break;
       case 65: System.out.println ("Está en la edad de jubilación"); break;
       default: System.out.println ("La edad no es crítica"); break;
        }
}
```

Ejemplo

```
public class TrySwitch {
  enum WashChoice {cotton, linen, wool, synthetic}     // Define enumeration type

  public static void main(String[] args) {
    WashChoice wash = WashChoice.cotton;   // Variable to define the choice of wash

    // The clothes variable specifies the clothes to be washed by an integer value
    // 1:shirts   2:sweaters  3:socks  4:sheets 5:pants
    int clothes = 3;

    switch(clothes) {
      case 1:
        System.out.println("Washing shirts.");
        wash = WashChoice.cotton;
        break;
      case 2:
        System.out.println("Washing sweaters.");
        wash = WashChoice.wool;
        break;
      case 3:
        System.out.println("Washing socks.");
        wash = WashChoice.wool;
```

Ciclos

```
1) while (expr){
....
}

2) do {
....
}while(expr);

3) for (exprInic; condCorte; pasoSig){
....
}
```

Ciclos

```
public static void main(String[] args) {
   byte value = 1;
   for (int i=0; i<8 ; i++) {
      value *= 2;
      System.out.println("Value is now " + value);
   }
}
```

Ejemplo

```
public class Factorial {
  public static void main(String[] args) {
    long limit = 20L;        // Calculate factorials of integers up to this value
    long factorial = 1L;     // A factorial will be stored in this variable

    // Loop from 1 to the value of limit
    for (long i = 1L; i <= limit; i++) {
      factorial = 1L;        // Initialize factorial

      for (long factor = 2; factor <= i; factor++) {
        factorial *= factor;
      }
      System.out.println(i + "! is " + factorial);
    }
  }
}
```

Ejemplo (Resultado)

```
1!   is  1
2!   is  2
3!   is  6
4!   is  24
5!   is  120
6!   is  720
7!   is  5040
8!   is  40320
9!   is  362880
10!   is  3628800
11!   is  39916800
12!   is  479001600
13!   is  6227020800
14!   is  87178291200
15!   is  1307674368000
16!   is  20922789888000
17!   is  355687428096000
18!   is  6402373705728000
19!   is  121645100408832000
20!   is  2432902008176640000
```

Ciclos

```
public class TryExample1_1 {
  public static void main(String[] args) {
    byte value = 1;
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
    value *= 2;
    System.out.println("Value is now "+value);
  }
}
```

Ciclos

```
public static void main(String[] args) {
   byte value = 1;
   for (int i=0; i<8 ; i++) {
      value *= 2;
      System.out.println("Value is now " + value);
   }
}
```

Ciclos (break)

```
public class Primes {
  public static void main(String[] args) {
    int nValues = 50;                  // The maximum value to be checked
    boolean isPrime = true;            // Is true if we find a prime

    // Check all values from 2 to nValues
    for(int i = 2; i <= nValues; i++) {
      isPrime=true;                    // Assume the current i is prime

      // Try dividing by all integers from 2 to i-1
      for(int j = 2; j < i; j++) {
        if(i % j == 0) {               // This is true if j divides exactly
          isPrime = false;             // If we got here, it was an exact division
          break;                       // so exit the loop
        }
      }
      // We can get here through the break, or through completing the loop
      if(isPrime)                      // So is it prime?
        System.out.println(i);         // Yes, so output the value
    }
  }
}
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

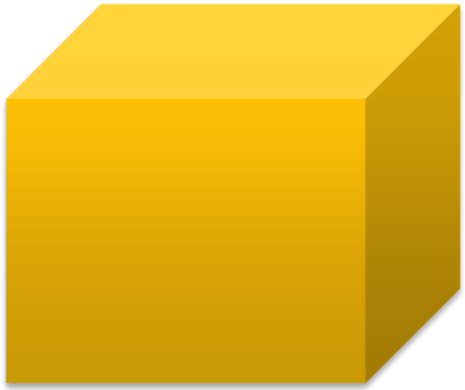
¿Qué es la programación orientada a objetos?

- Paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas.
- Entender el mundo como objetos.
- Se pueden descomponer en objetos más pequeños.
- Se relacionan entre sí
  - Un objeto puede pedir a otro que realice alguna acción por él.

¿Qué es un objeto?

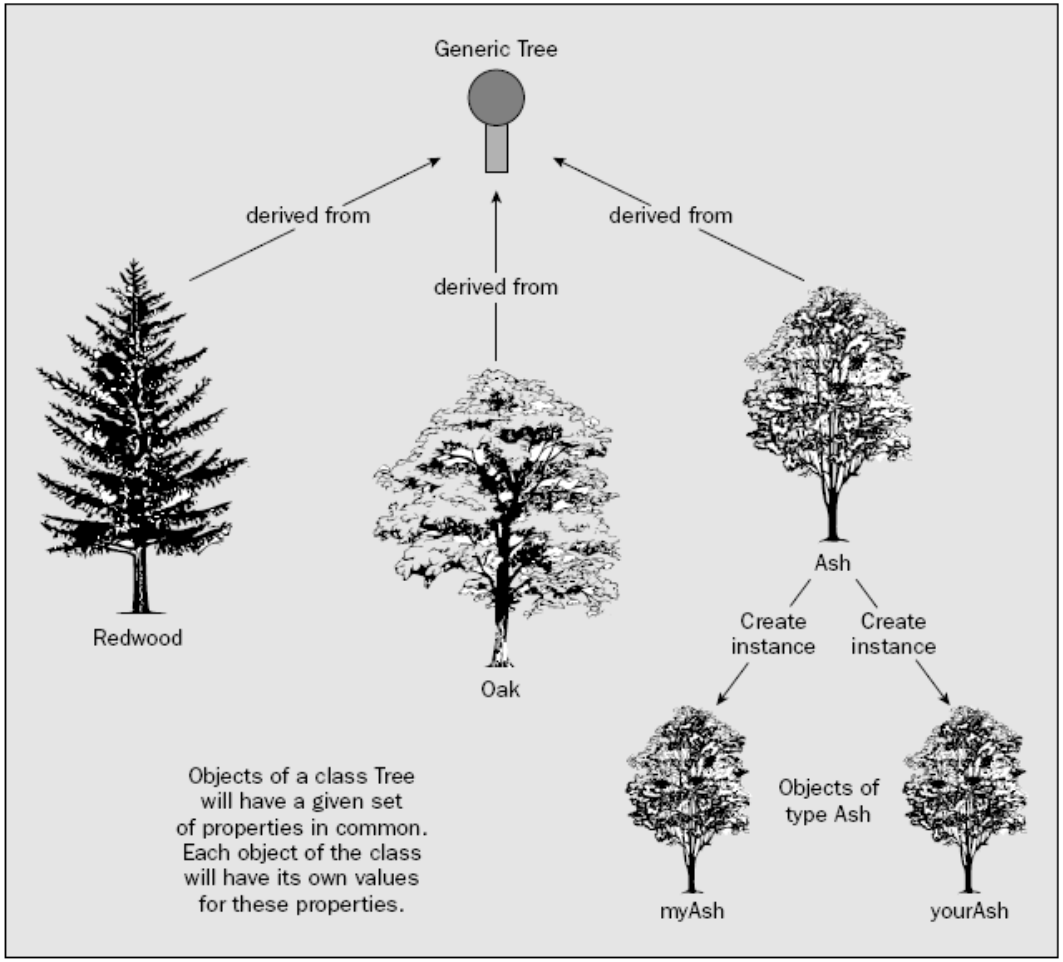
- Entidad lógica que representa una entidad del mundo real.
- Las entidades del mundo real se describen por medio de sus atributos y su conducta

Características?



Qué hace?
Para qué sirve?

Ejemplo



¿Qué es una clase?

- Plantilla que describe los tipos de estado y comportamiento que un objeto de su clase soporta

- ✓ Color
- ✓ Raza
- ✓ Edad
- ✓ Estatura

- ✓ Comer
- ✓ Mover la cola
- ✓ Jugar
- ✓ Ir al baño

¿A nivel de código qué es un objeto?

- Un objeto es una instancia de una clase
- Cada objeto tiene su propio estado y comportamiento (definidos por la clase)

- Una clase es una representación abstracta de un objeto
- Una instancia es una representación concreta de una clase

¿Qué es un atributo?

- Son las características que diferencian a un objeto de otro
- Determinan las cualidades del objeto

¿Qué es un atributo?

- Cada objeto tiene sus propias variables de instancia.
- El valor asignado a las variables de instancia hacen el estado de la clase.

- ✓ Color: blanco y café
- ✓ Raza: Beagle
- ✓ Edad: 1 año
- ✓ Estatura: 60cm

¿Qué es un método?

- Los métodos es donde se almacena la lógica de la clase: el algoritmo.

- ✓ Comer: ¿Cómo?
- ✓ Mover la cola: ¿Cómo?
- ✓ Jugar: ¿Cómo?
- ✓ Ir al baño: ¿Cómo?

¿Qué es una propiedad?

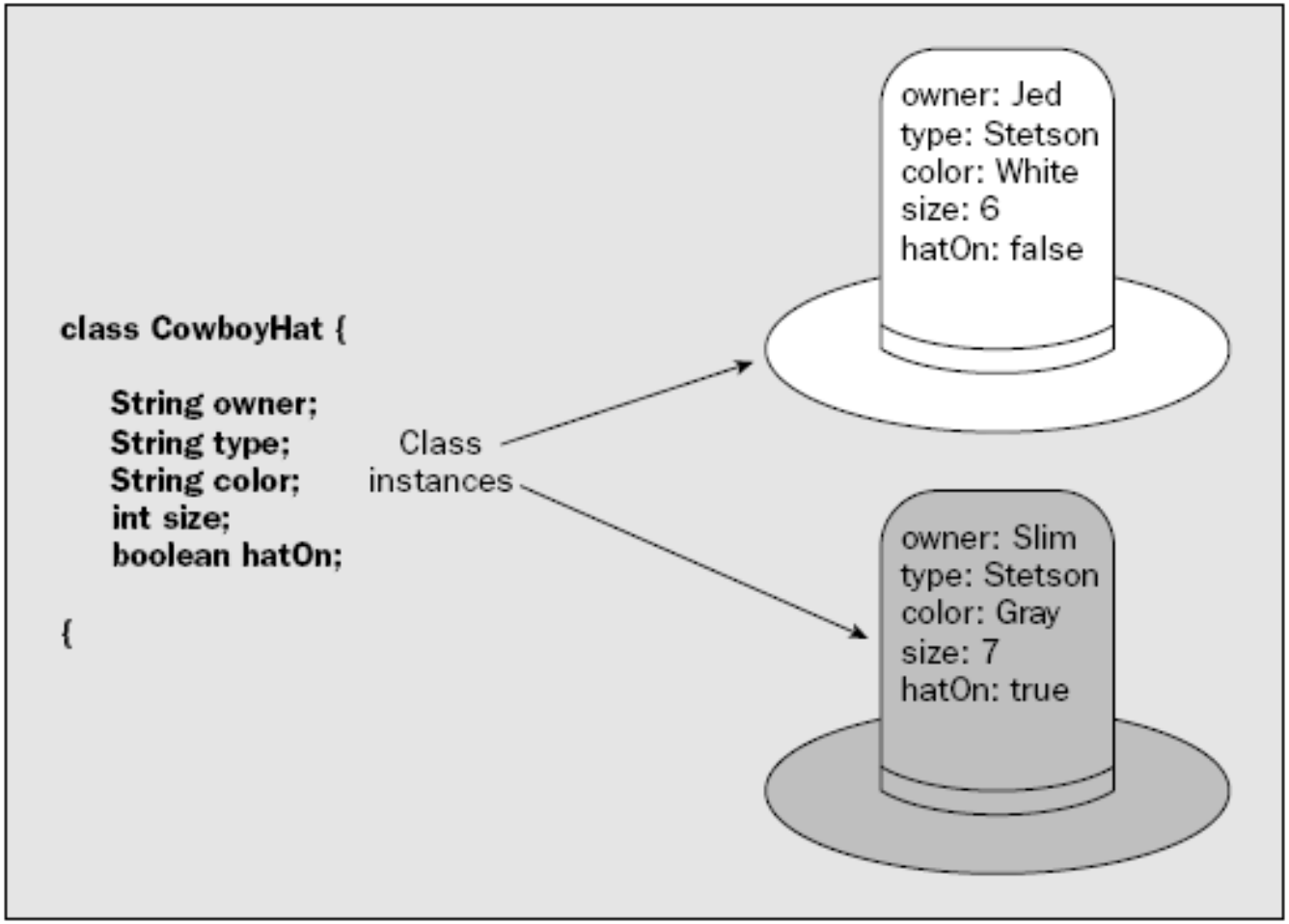
- Son métodos que permiten acceder a los atributos del objeto sin hacer referencia directa a ellos, generalmente de lectura y escritura

- ✓ Atributo: color
- ✓ Propiedad lectura: getColor
- ✓ Propiedad escritura: setColor

Ejemplo

- Clase animal
  - Atributos:
    - Color
    - Raza
    - Cola
    - …
  - Métodos:
    - Correr
    - Volar
    - Comer
    - …
  - Propiedad:
    - getColor
    - getRaza
    - getCola
    - setColor
    - setRaza
    - setCola

Resumen

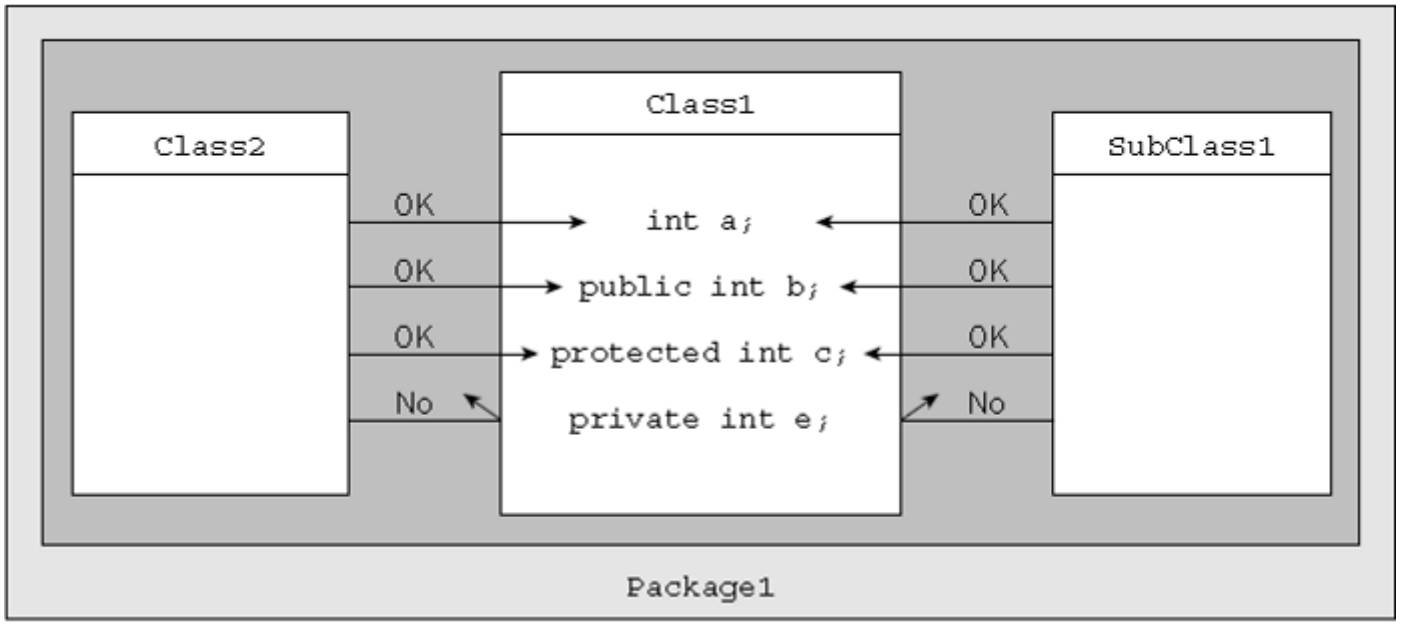


Visibilidad

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Visibilidad

- Modificadores de acceso
  - protected
  - public
  - private



Clase (Java)

```
public class Carro
{
public Carro(){
}
public Carro(String marca){
            }
public Carro(String marca, int km){
            }
String marca;
int kilometraje;
String color;
void encender(){
}
void acelerar(){
}
void apagar(){
}
void getSillas(){
}
}
```

Encapsulamiento de Atributos

El ***encapsulamiento*** *es* un principio del lenguaje **Java** cuyo objetivo es hacer que los atributos de las clases se puedan editar sólo a través de métodos declarados en la misma clase.
Para lograr esto De manera general, se hace teniendo los atributos como privados y métodos públicos que los controlan (propiedades). Estos métodos se denominan getters (se encargan de obtener el valor de la propiedad) y setters (se encargan de setearla).

Al definir de esta manera las clases, permite controlar que el cambio de valores de los atributos se realice de acuerdo a condiciones implementadas en la clase

Encapsulamiento de Atributos

Utilidad del encapsulamiento?

El encapsulamiento nos permite la mantenibilidad, estabilidad y seguridad de nuestra clase. Ejemplo en un proyecto en equipos, otro equipo va utilizar nuestras clases.

Al utilizar el encapsulamiento nuestra clase puede validar la información que le suministran ej: la edad del perro no puede ser menor que cero, la raza no puede ser un dato null, etc.

Encapsulamiento de Atributos

Sin encapsulamiento

```
public class Perro {
    public String nombre;
    public String raza
    public float edad;
}
```

```
Perro p = new Perro();
p.nombre = "Neron";
p.raza = “criollo”;
p.edad =3.5;
System.out.println("Mi perro se llama: " + p.nombre);
System.out.println("La raza es: " + p.raza);
System.out.println("Tiene: " + p.edad+" años");
```

Encapsulamiento de Atributos

Forma correcta con encapsulamiento

```
public class PerroValiente {
    private String nombre;
    private String raza
    private float edad;
    public String getNombre(){ return nombre;}
    public String getRaza(){ return this.raza;}
    public float getEdad(){ return edad;}
    public void setNombre(String pNombre){ nombre = pNombre;}
    public void setEdad(int pEdad){ edad = pEdad;}

   public void setRaza(String raza){
       this.raza = raza;
   }
}
```

```
Perro p = new PerroValiente();
p.setNombre ("Neron“);
p.setRaza (“criollo”);
p.setEdad (3.5);
System.out.println("Mi perro se llama: " + p.getNombre());
System.out.println("La raza es: " + p.getRaza());
System.out.println("Tiene: " + p.getEdad()+" años");
```

Programación Avanzada

Ing. Orlando Vasquez

Sesión

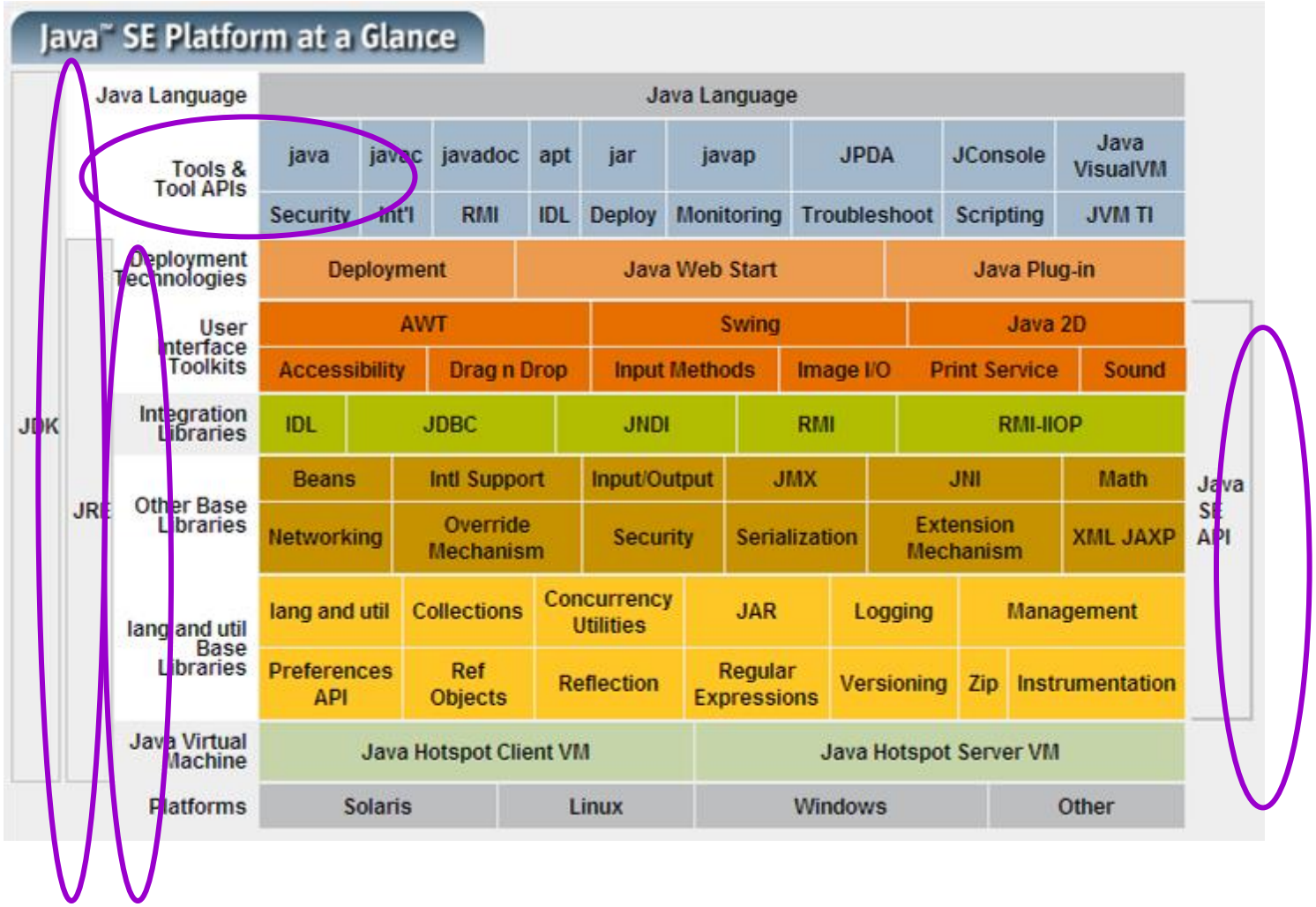
Proceso de Instanciación de objetos
- Constructores por defecto
- Constructores con parámetros
- Proceso de creación de un objeto

Compilación de programas
- Variables de ambiente
- Javac
- Java

¿Qué es Java?

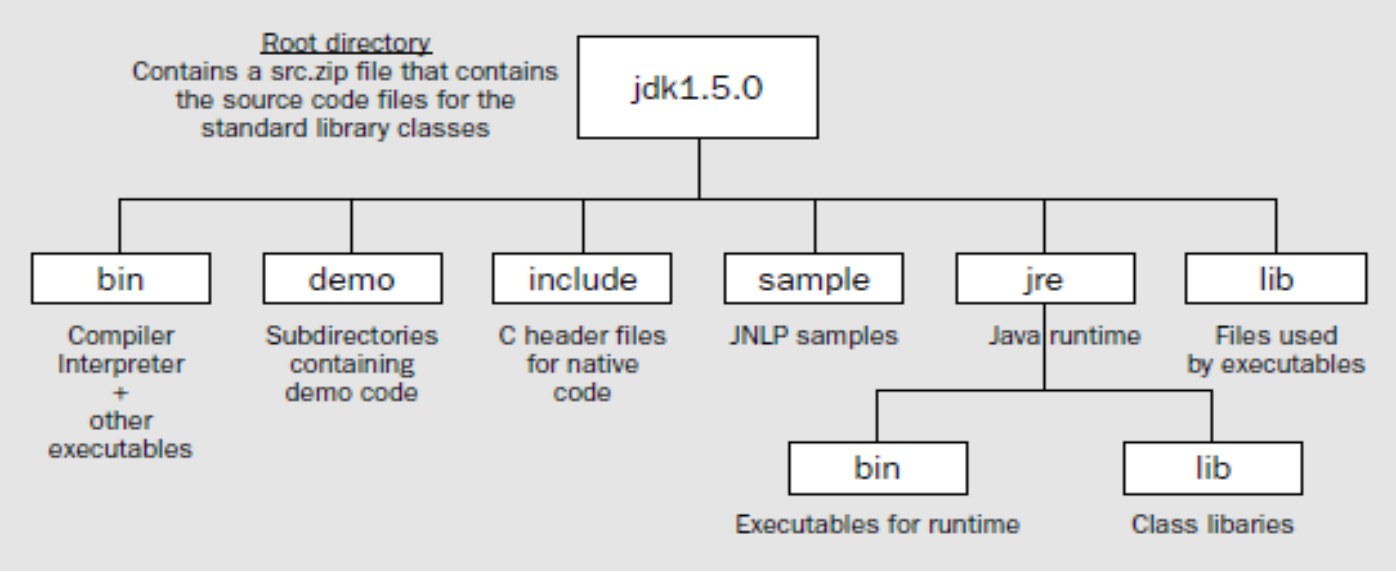
- Lenguaje de programación
- Desarrollado por Sun Microsystems hoy propiedad de Oracle
- Orientado a objetos
- Simple
- Portable
- Robusto
- Seguro

Ambiente Java



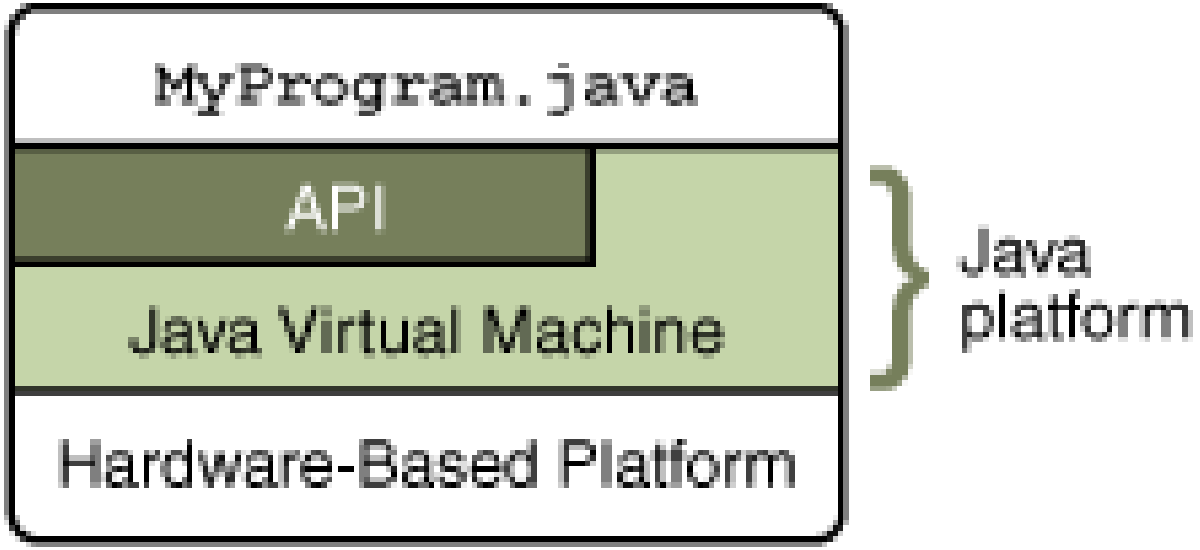
Ambiente Java

- JDK (Java Development Kit): necesario para desarrollar
- JRE (Java Runtime Environment): necesario para ejecutar



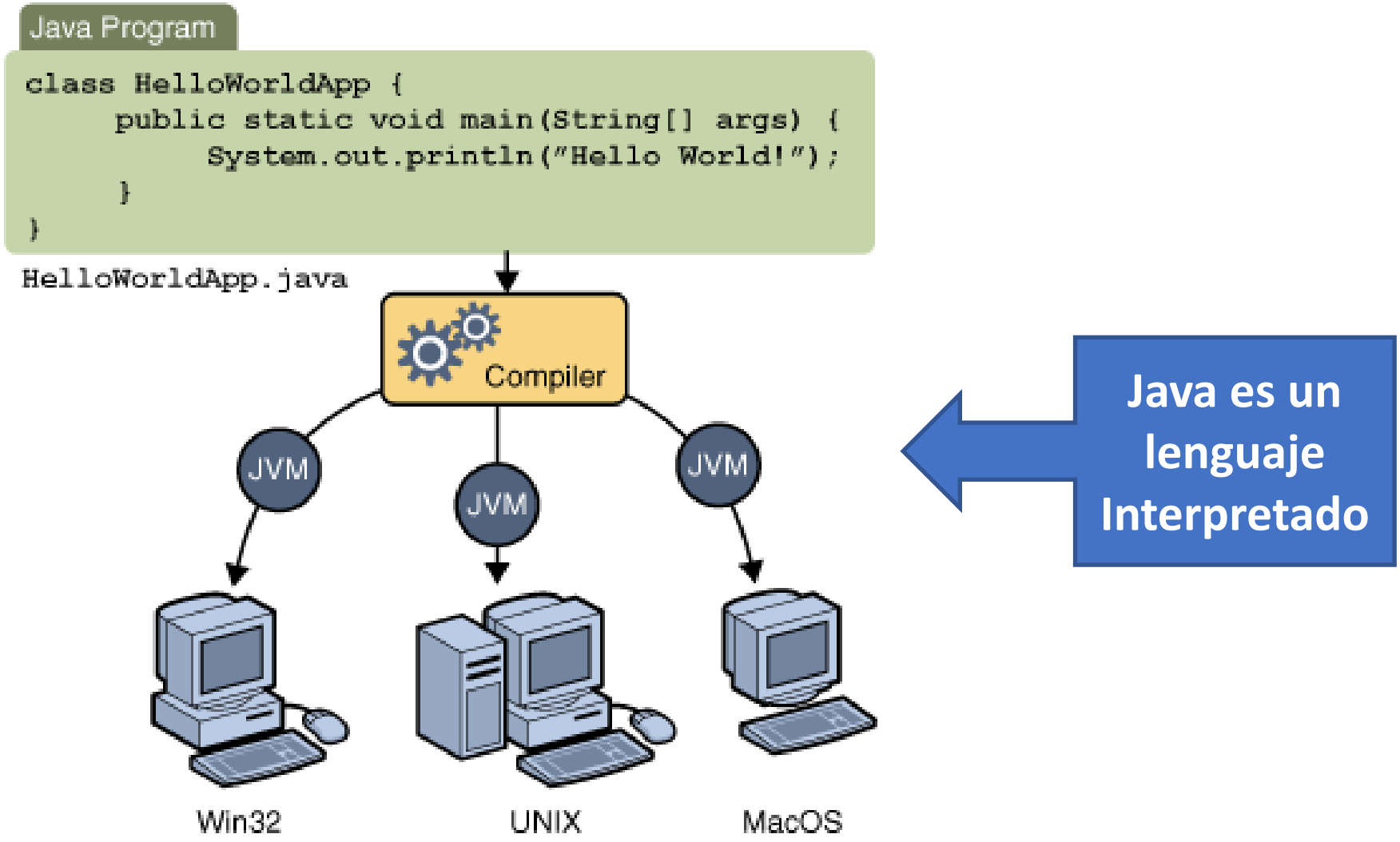
Plataforma Java

- El programa no corre directamente en el computador sino en una Plataforma Java.

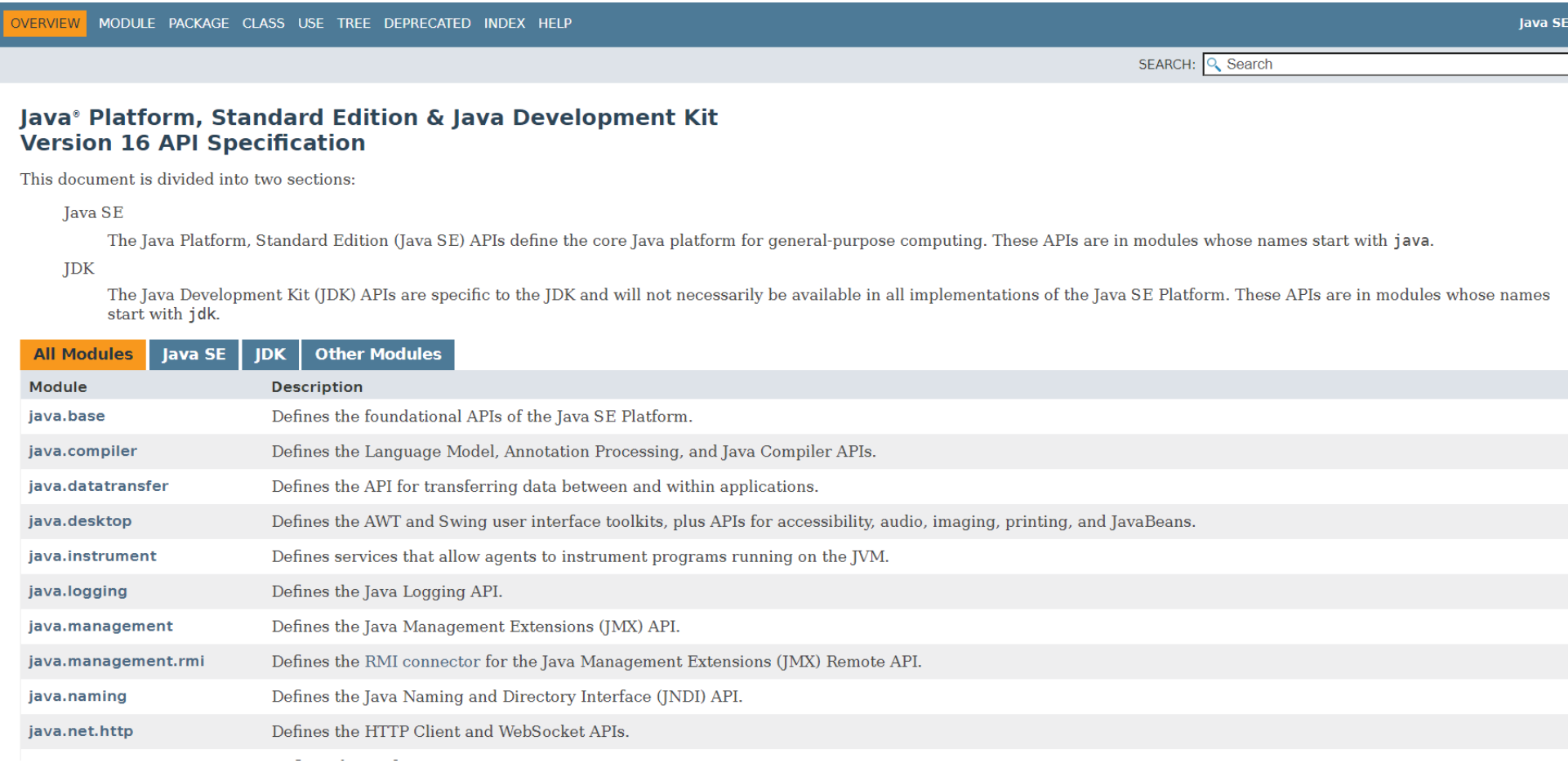


- Puede correr un programa Java en una variedad de computadores con diferentes sistemas operativos.

Ambiente Java



Java API (App Programming Interface)



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP Java SE

SEARCH: Search

Java® Platform, Standard Edition & Java Development Kit Version 16 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules | Java SE | JDK | Other Modules

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.net.http	Defines the HTTP Client and WebSocket APIs.

Comandos y Variables de Ambiente

- Javac: compilar una clase java (.java)
- Java: ejecutar una clase compilada (.class)

- Muy importante considerar las versiones de JDK y JRE (no se puede ejecutar una clase compilada con un JDK superior que el JRE)

- JAVA_HOME: variable de ambiente que indica la ruta donde se encuentra el JDK
- CLASSPATH: variable de ambiente que indica la ubicación de librerías (.class, .jar) para la compilación y ejecución

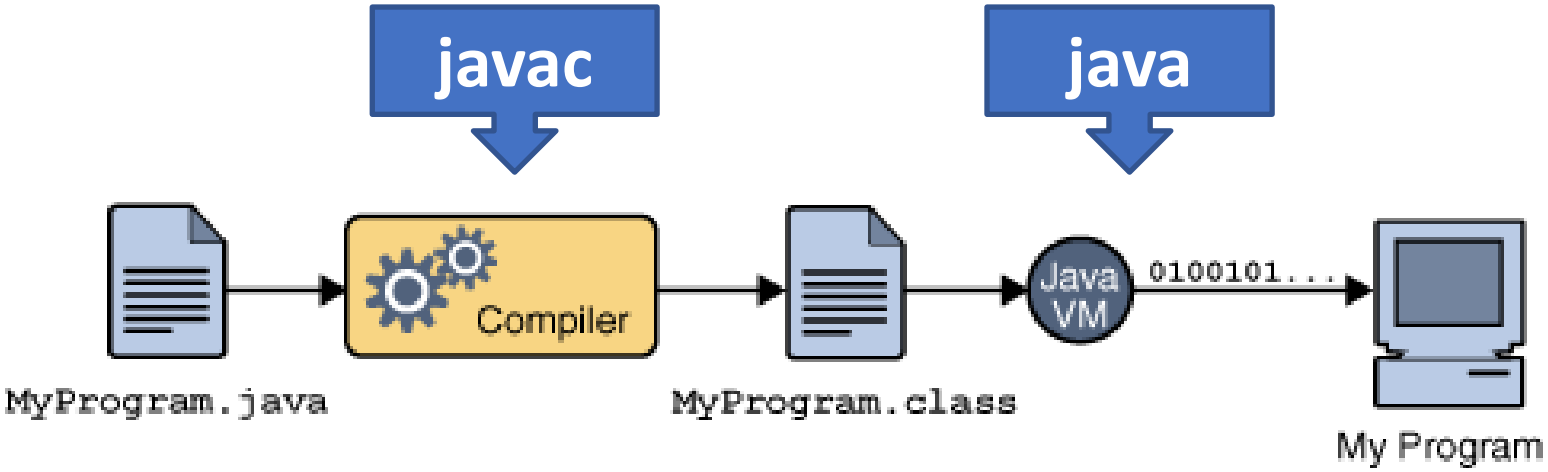
Comandos y Variables de Ambiente

```
Exception in thread "main" java.lang.NoClassDefFoundError: MyProgram/class
```

- CLASSPATH puede ser variable predefinida (en el sistema operativo)
- Java –cp <ruta de librerías> MyProgram

- Java –xms 128m –xmx 512m MyProgram
  - Define que el HEAP del programa Java va a utilizar de la memoria RAM 128MB iniciales hasta 512MB como máximo

Ambiente Java



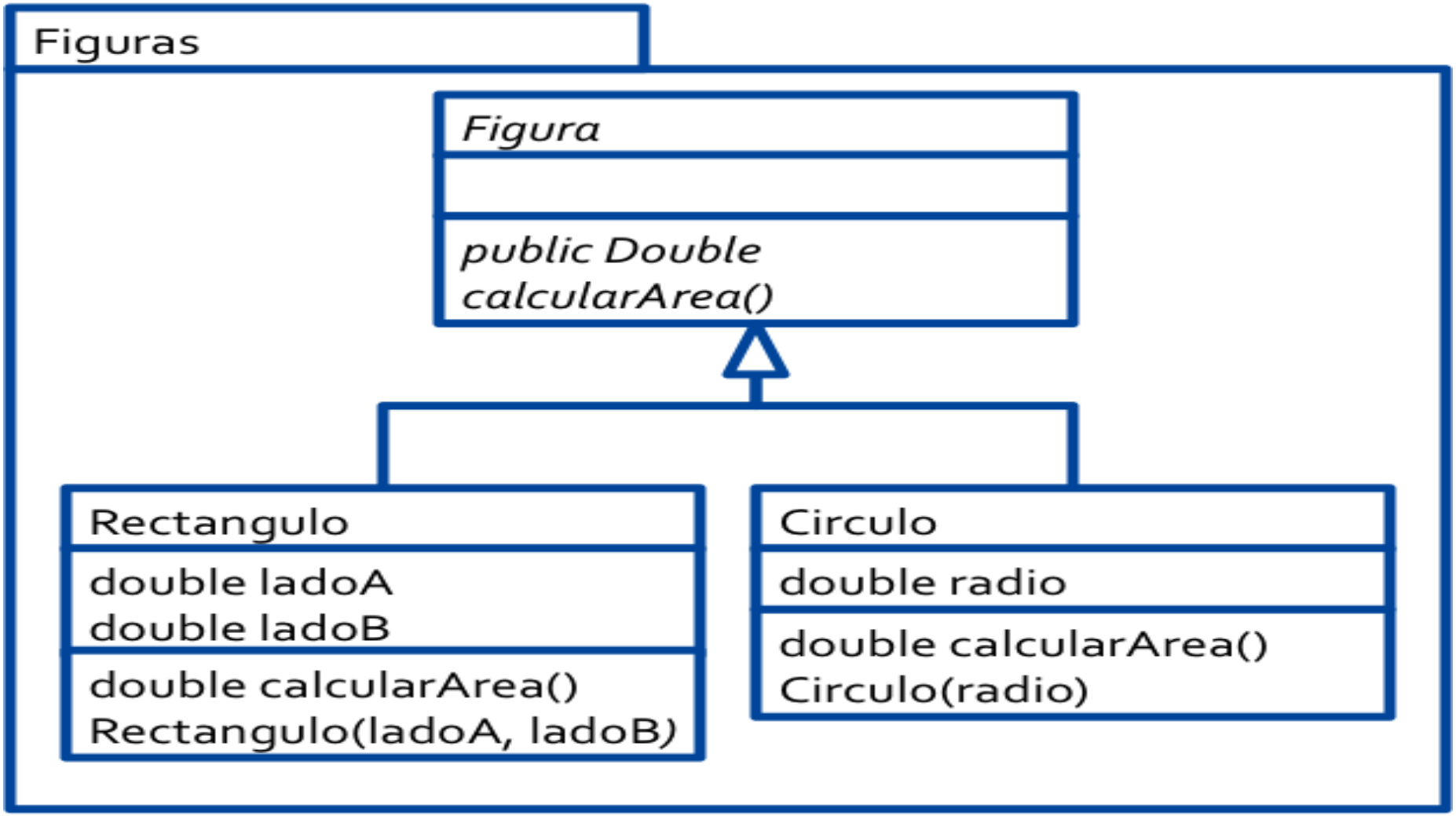
Paquetes

- Los paquetes son manera para diseñar y organizar a gran escala.
- Se utilizan para categorizar y agrupar clases.
- Equivale al concepto de librería.
- Una clase puede definirse como perteneciente a un **package** y puede usar otras clases definidas en ese o en otros packages.
- Los packages delimitan el espacio de nombres. El nombre de una clase debe ser único dentro del package donde se define.
- Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.
- Un paquete equivale a una ruta de carpetas o directorios en el disco.

Paquetes

- Sintaxis:
- **package** *nombre_package*;
  - **package** miPackage;
    . . .
    class miClase {
    . . .
- Si un enunciado aparece en un archivo fuente java, debe ser la primera línea del archivo, a excepción de los espacios en blanco y los comentarios.
- La claúsula package es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto sin nombre.

Paquetes


Figuras
Figura
public Double
calcularArea()
Rectangulo
double ladoA
double ladoB
double calcularArea()
Rectangulo(ladoA, ladoB)
Circulo
double radio
double calcularArea()
Circulo(radio)

Import

```
package myproject;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Vector;

public class Pruebas {
    public Pruebas() {
     Dog dog = new Dog();
    }
```

- Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package.
- En caso contrario es necesario hacer accesible el espacio de nombres donde está definida la clase que vamos a utilizar a nuestra nueva clase.
- La cláusula **import** simplemente indica al compilador donde debe buscar clases adicionales.

Nombres de los paquetes

- Se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las direcciones URL de Internet.
  - misPaquetes.Agenda
- De esta forma se pueden tener los packages ordenados según una jerarquía equivalente a un sistema de archivos jerárquico.
- El API de java está estructurado de esta forma
  - Primer calificador (java o javax) que indica la base, un segundo calificador (lang, util…) que indica el grupo funcional de clases y opcionalmente subpackages en un tercer nivel, dependiendo de la amplitud del grupo.

Diagrama de clases

- Un **diagrama de clases** es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos.

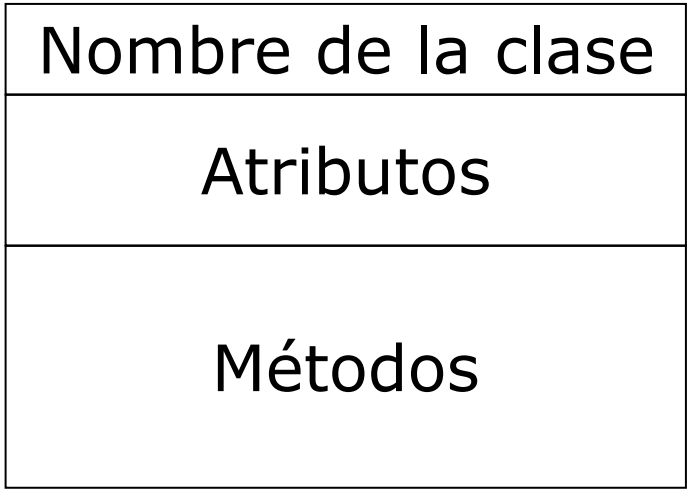


Diagrama de Clases

- public (+)
- protected (#)
- private (-)

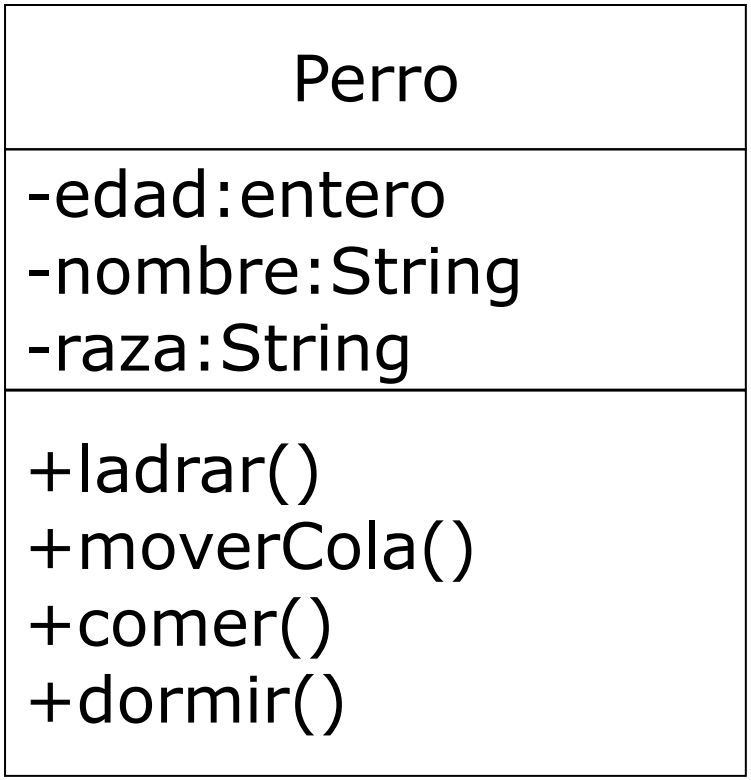


Diagrama de Clases

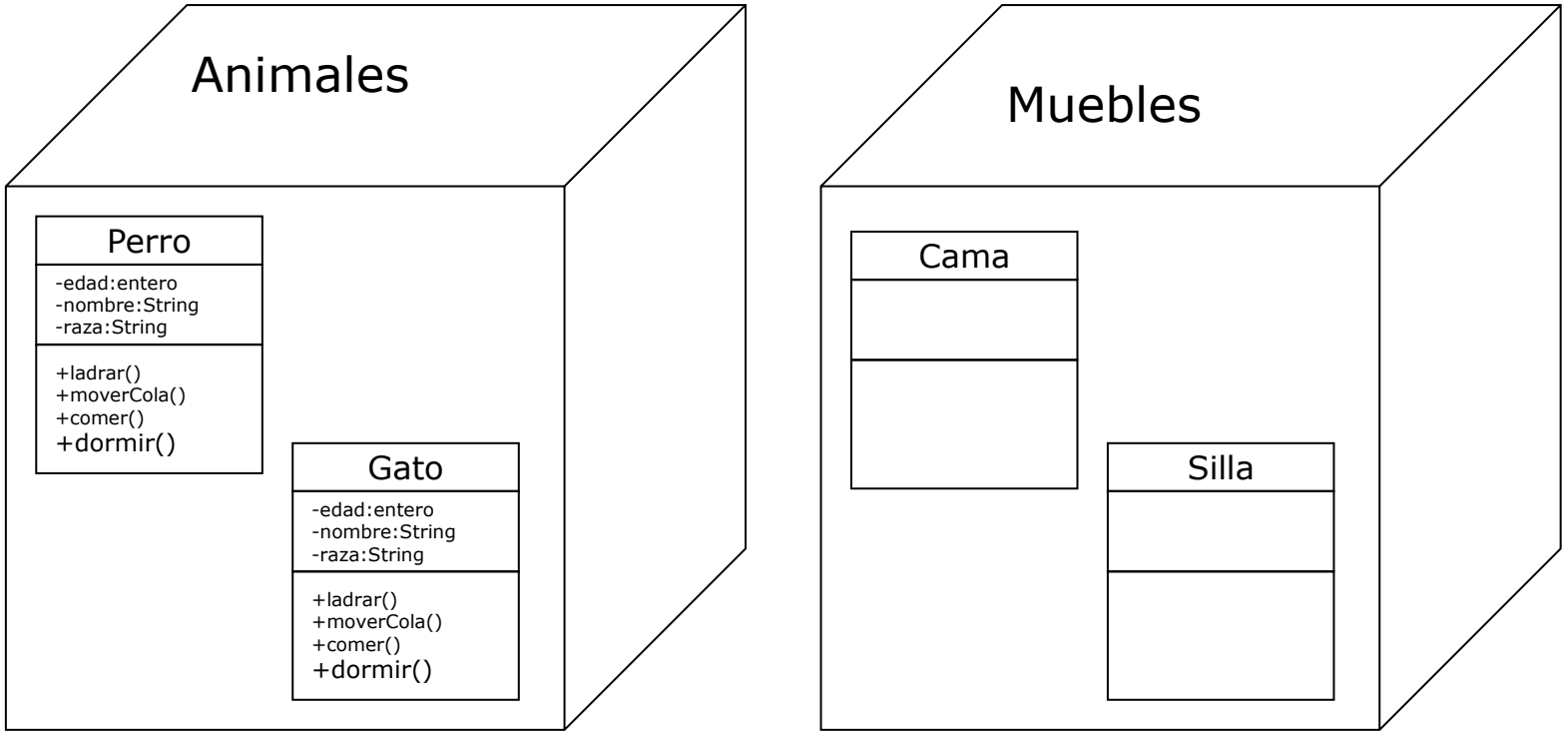
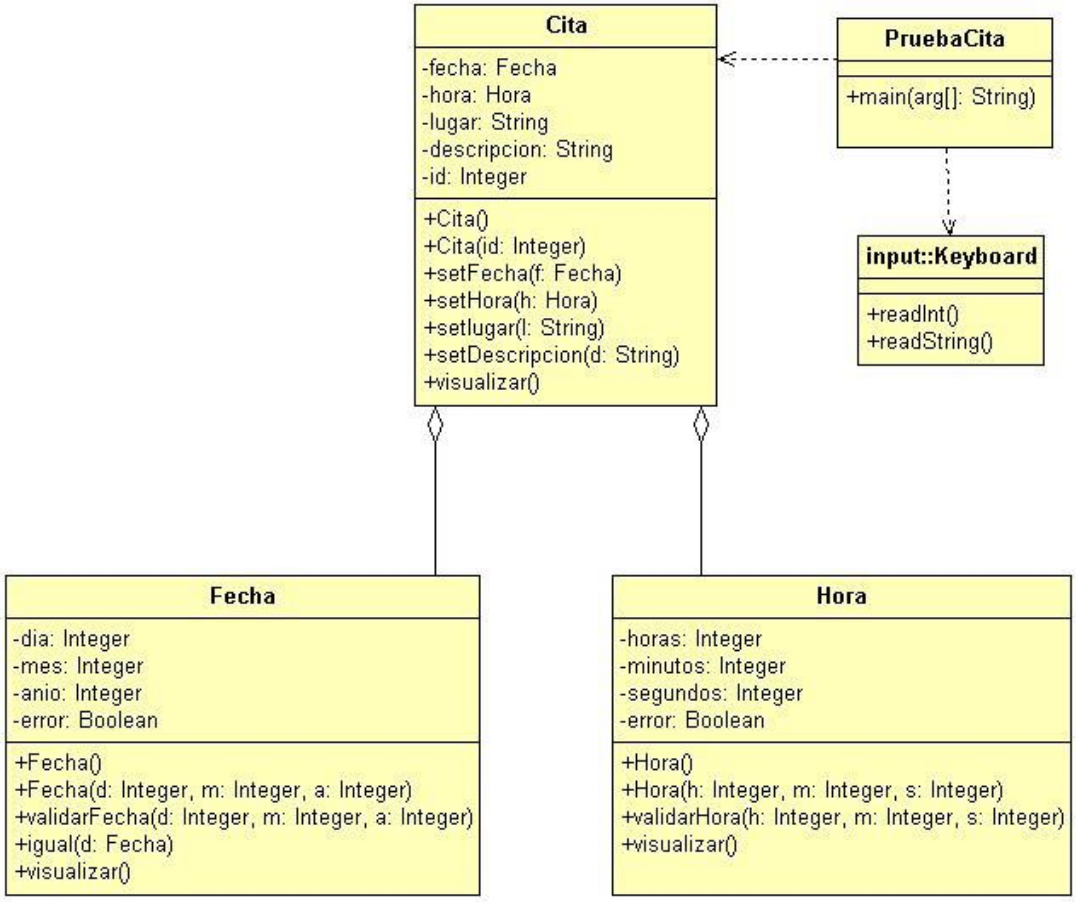


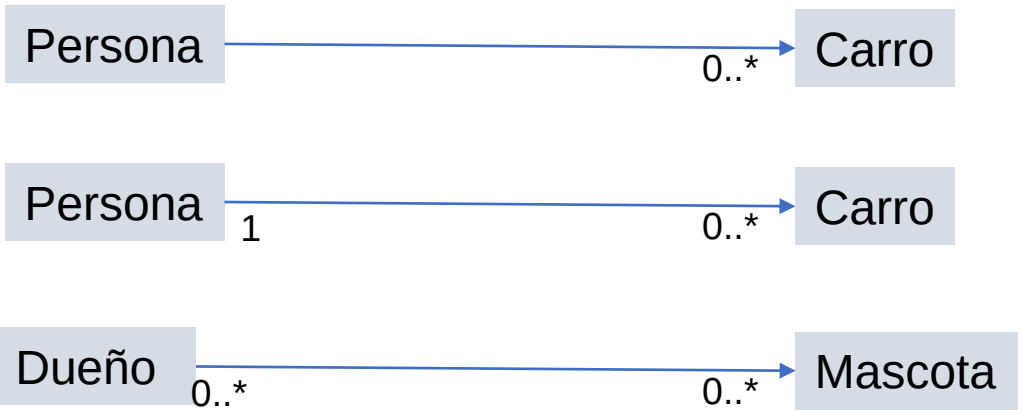
Diagrama de clases



Tipos de relaciones

1.Asociación

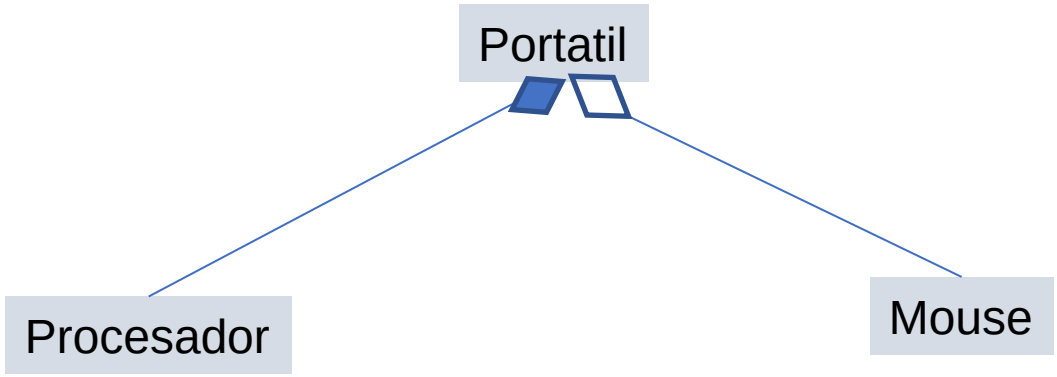
Indica como una clase se relaciona con otra. Una relación se dibuja con una línea solida entre dos clases. Por lo general son binarias y pueden ser unidireccionales o bidireccionales



1	Uno y solo uno
0..1	Cero o uno
0..*	Cero o mas
1..*	Al menos uno
N..M	Mínimo N, máximo M

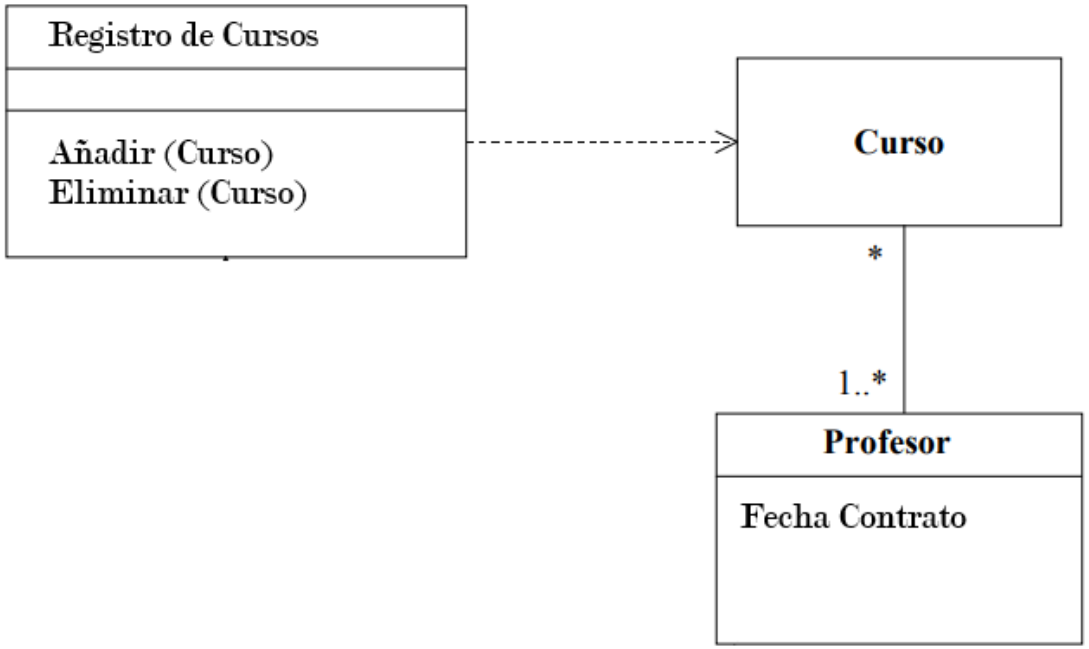
Tipos de relaciones

2. Agregaciones
Se utiliza para modelar una relación “todo o parte”, lo que significa que el objeto todo contiene al objeto parte
2.1 Agregación normal: Se dibuja con un rombo vacío del lado del todo. Esto indica que la clase “Todo” sigue funcionando a pesar de no tener alguna de las partes
2.2. Agregación de composición: Se dibuja con un rombo lleno del lado del todo e indica que el “todo” necesita a las “partes” para su funcionamiento



Tipos de relaciones

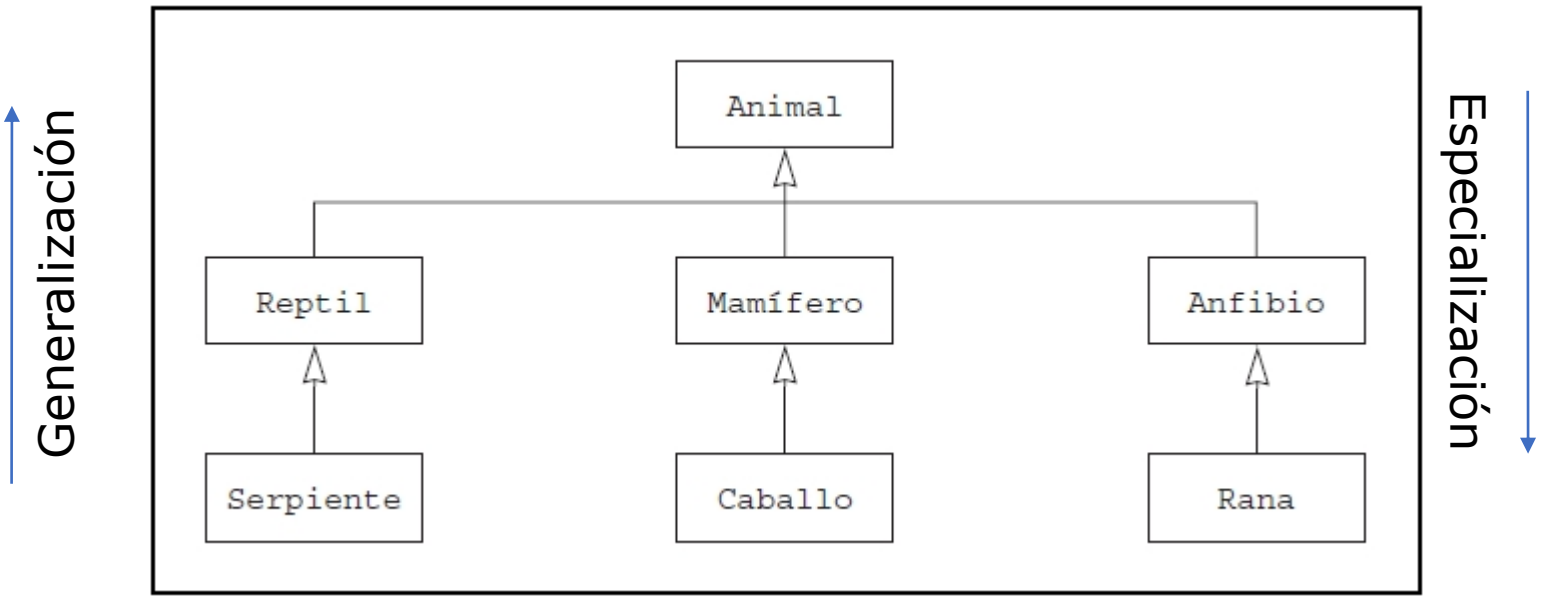
3. Dependencia: Es una relación en la cual una clase “depende” de otra para su funcionamiento, pero la clase de la cual se depende no hace parte estructural de la clase dependiente. La relación de dependencia se representa por una línea discontinua con una flecha en el lado del elemento independiente



Tipos de relaciones

4. generalización indica que una clase hereda los métodos y atributos definidos por una superclase.

5. especialización es la creación de subclases a partir de una superclase.
Se demarca con una línea continua y una flecha vacía del lado de la superclase



Declaración de una clase

```
access_modifier class MyClass {
    // field, constructor, and
    // method declarations
}

access_modifier class MyClass extends MySuperClass implements YourInterface {
    // field, constructor, and
    // method declarations
}
```

Ejemplo:

```
public class MiPrimeraClase{
……
……
……

}
```

Declaración de variables

Las clases tienen diferentes tipos de variables

- Variables miembro de una clase: se denominan campos.
- Variables en un método o bloque de código: se denominan variables locales.
- Variables en las declaraciones de métodos: se denominan parámetros

- access_modifier dataType variableName

- Ej:
  - private int variable1;
  - private char variable2=‘x’;
  - Public String variable3=“cadena ejemplo”;

Declaración de métodos

Los métodos son la implementación del comportamiento de una clase.

```
acces_modifier dataTypeReturn methodName (parameter list){
……
……..
  return variable;
}
```

Nota: dataTypeReturn, puede ser de tipo *void*

Ejemplo

```
public double areaCirculo(double radio){
        double area=0;
        if (radio<=0){
            System.out.println("Valor de radio no valido");
            area=-1;
        }
……..
return área;
}
```

Método Constructor

- Método que permite instanciar una clase, es decir, es aquel que se invoca cuando se crea un objeto
- Define las características de inicialización del objeto


```
// Constructor to create a Hat object
public Hat(String person, int theSize)  {
    size = theSize;                 // Set the hat size
  owner = person;                 // Set the hat owner
}
```

This is a special method that creates **Hat** objects

Método Constructor

- **Constructor por defecto no recibe ningún parámetro**

Ej:

```
public class Deposito {
    //Campos de la clase
    private float diametro;
    private float altura;
    private String idDeposito;
    //Constructor sin parámetros auxiliar
    public Deposito () {    } //Cierre del constructor
```

Método Constructor

```
//Constructor de la clase que solicita parámetros
  public Deposito (
float valor_diametro,
float valor_altura,
String valor_idDeposito)
  {
      if (valor_diametro > 0 && valor_altura > 0) {
         diametro = valor_diametro;
         altura = valor_altura;
         idDeposito = valor_idDeposito;
      } else {
         diametro = 10;
         altura = 5;
         idDeposito = "000";
      }
  }
……
```

Múltiples Constructores

```
public class Persona {
   private String nombre;
   private int edad;
    public Persona (String nombrePersona) {   //CONSTRUCTOR 1
      nombre = nombrePersona;
      edad = 0;     }

   public Persona () {   //CONSTRUCTOR2
      nombre = "";
      edad = 0;     }

   public String getNombre () { return nombre; }
} //Cierre de la clase
```

Modificadores

- Establecen condiciones para acceder a los atributos, propiedades y métodos del objeto

- Modificadores de acceso:
  - public, protected, private.

- Otros (no acceso):
  - final: Ninguna clase puede heredar de ella
  - abstract: No puede ser instanciada. Su misión es ser extendida

Modificadores

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Vida de un Objeto

- Un objeto ocupa espacio de memoria, cuyo tamaño depende de la cantidad de atributos e información que contiene
- El tiempo de vida es determinado por la variable que contiene la referencia al objeto
- El garbage collector es el responsable de limpiar periódicamente objetos sin referencia en la memoria

Ej;

Deposito deposito=null;

deposito= new **Deposito** **(2,3.5, “01”) ;**

Consolidación de Conceptos

```
package persona;

import java.util.Vector;

public class Persona {

    private Vector companieros;

    public Persona() {
        this.companieros = new Vector();
    }

    public Persona(Vector amigos) {
        this.companieros = amigos;
    }

    public void addCompaniero(Persona unaPersona){
        this.companieros.add(unaPersona);
    }

    public Persona getPrimerCompaniero(){
        return (Persona)this.companieros.remove(0);
    }

}
```

Paquete

Imports

Declaración de la clase

Variables

Constructores

Métodos

Taller Conceptos de Objetos

Reglas:

1) No se permite código resuelto en internet
2) No esta permitido copiar código de otros grupos
3) Los programas que no estén correctamente indentados tendrán -1 punto
4) Se deben entregar los archivo fuente .java
5) Al inicio de cada programa colocar los datos de los integrantes del grupo y su respectivo número

EJ: /*Pedro Perez, Mariana Gomez Grupo 3*/

6) Enviar un pantallazo de la ejecución del programa dentro de archivo Word (indicar integrantes del grupo) ej:

```
Indique lado 2:
5
El area del cuadrilatero es: 25.0
----------------------------------
Menu de opciones
```

Taller Conceptos de objetos

- Crear un programa que permita implementar las siguientes funciones geométricas
  - Área de un circulo
  - Diámetro del circulo
  - Área de un triangulo
  - Área de un cuadrilátero
- Si los datos enviados no son validos, cada función debe mostrar un mensaje por pantalla y enviar como respuesta el valor de -1
- La clase que contiene el método main, se debe llamar Math. La clase que contiene los métodos de cálculos se debe llamar Geometria.
- Este taller es asistido
- Nota: para compilar y ejecutar manualmente

javac paht1\path2\path3\Clase.java

java paht1.path2.path3.Clase

System.out.println(“Dato:”+dato);

Sesión

Atributos
- de clase (estáticos), Final

Métodos
- Métodos de instancia
- Parámetros
- Sobrecarga de métodos
- Métodos static

Usando los objetos
- La propiedad this, La notación punto (.)

Variable

- Pueden **cambiar** de contenido a lo largo de la ejecución de un programa
- Corresponde a un área reservada en la memoria principal.
- Una declaración de variable siempre contiene dos componentes, el **tipo de la variable** y su **nombre**

Variable

<Tipo_dato> <identificador>

- int x;
- char caracter;
- boolean esta;
- int cont1;
- long cedula;
- float cuotaDeb;

Identificador

- Todos los componentes: clases, variables y métodos, necesitan un nombre.
- Nombres pueden empezar en $, _ ó cualquier letra.
- Después del primer carácter puede ir cualquier $,_,número o letra.
- Los identificadores son sensibles a mayúsculas
- Sin restricciones de número de caracteres.

Identificador

- Variables:

  La primera letra va en minúscula. Si el nombre está compuesto de varias palabras, la primera letra de cada una debe ir en mayúscula.

  – precioDolar
  – telefonoFijo

- Constantes:

  En mayúscula. Si el nombre está compuesto de varias palabras, estas deben ir separadas por _.

  – PESO_MINIMO. Viene precedida por la palabra reservada **final**
  – Ej: **final** double PI=3.14;

Identificador

- Variables:
  - int _a;
  - int :b;
  - int -d;
  - int $c;
  - int _____2_w;
  - int _$;
  - int 7g;
  - int este_es_un_nombre_muy_largo;
  - int e#;
  - int .f;

¿Correctos?

Identificador

- Clases:

  La primera letra debe estar en mayúscula.

  Si el nombre está compuesto de varias palabras, la primera de cada una debe ir en mayúscula.

  – Animal

  – CarroDeportivo

Identificador

- Métodos:

  La primera letra debe estar en minúscula.

  Si el nombre está compuesto de varias palabras, la primera de cada una debe ir en mayúscula.

  Normalmente el identificador es la combinación de verbo-sustantivo.

  – cambiarNombre()
  – setPrecio()

Palabras reservadas

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

Tipos de datos primitivos

- Numéricos
  - Enteros
    - byte, int, short, long
  - Reales
    - float
    - double
- Lógicos
  - boolean
- Caracteres
  - Char

Nota: Las variables alfanuméricas son objetos de la clase **String**

Tipos de datos primitivos

Identificador	Descripción	Rango	Valor por defecto
byte	Entero con signo, 8 bits	-128 a 127	0
short	Entero con signo, 16 bits	-32.768 a 32.767	0
Int	Entero con signo, 32 bits	-2.147.483.648 a 2.147.483.647	0
long	Entero con signo, 64 bits	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	0

Tipos de datos primitivos

Identificador	Descripción	Rango	Valor por defecto
float	Punto flotante, precision simple	Ver 32-bit IEEE 754 floating point	0.0
double	Punto flotante, precision doble	Ver 64-bit IEEE 754 floating point.	0.0
boolean	Valor booleano	true o false	false
char	Caracter simple, 16 bits	'\u0000' a '\uffff'	'\u0000'

Constantes

- Información que no cambia
- Deberían declararse como final y static

```
final int FEET_PER_YARD = 3;            // Constant values
final double MM_PER_INCH = 25.4;        // that cannot be changed
```

Operadores

- Aritméticos
- Asignación
- Relacionales
- Lógicos

Aritméticos

Suma	+	4 + 2
Resta	-	4 – 2
Multiplicación	*	8 * 5
División	/	10 / 2
Módulo	%	10 % 2

Asignación

- int x = 8;
- int variable = 0;
- char b = ‘a’;
- float f = 0.8f;
- double n = 0.8;
- boolean esPrimo = false;
- variable = x;

Asignación

int x=0, y=3;

- x = x + 1;
- x++;
- x += 1;
- x = y / x;
- c = x % y;
- c--;

Equivalentes

+=, -=, *=, /=

Relacionales

Igualdad	==	5 == 5
Diferencia	!=	5 != 5
Mayor que	>	5 > 4
Menor que	<	3 < 9
Mayor o igual	>=	2 >= 2
Menor o igual	<=	15 <= 20

Para comparar objetos se utiliza el metodo **equals.**

Ej: nombreObjeto.equals(objetoAComparar)

cadena.equals(“Contenido”)

Operadores

- Aritméticos
- Asignación
- Relacionales
- **Lógicos**

Lógicos

And	&&	b && c
Or	||	b || c
not	!	!b

Sobrecarga de metodos

La **sobrecarga de métodos** es la creación de varios **métodos** con el mismo nombre pero con diferente lista de ***tipos*** de parámetros. **Java** utiliza el ***número*** y ***tipo*** de parámetros para seleccionar cuál definición de **método** ejecutar. El ejemplo mas clásico de sobrecarga lo vemos en los métodos constructores

Ej:

```
public class Sobrecargado {
  public static void main(String[] args) {
      sobrecarga(1);
      sobrecarga(2,3);
      sobrecarga(3,"palabra");

  }
  public static void sobrecarga (int i){
System.out.println ("1 parámetro entero");
  }
  public static void sobrecarga (int i, int j){
     System.out.println ("2 parámetros enteros");
  }
  public static void sobrecarga (int i, String x){
     System.out.println ("1 parámetro entero y 1 parámetro String");
  }
}
```

Métodos de clase y métodos de instancia

Los métodos de clase, son aquellos métodos que pueden ser invocados directamente sin la necesidad instanciar un objeto de la respectiva clase. Los métodos de clase se definen con la palabra clave static. Ej:

Ej:

```
private static String copyright="Pontificia Universidad Javeriana";

public static void info(){
    System.out.println (copyright);
  }
```

Un método declarado estático, solo puede invocar variables de clase que hayan sido declaradas estáticas.

Métodos de clase y métodos de instancia

Los métodos de instancia, son aquellos métodos que para ser invocados, es necesario instanciar primero la clase correspondiente. Ejemplos de estos métodos son los setters y getters.

Nota:

Los Objetos instanciados no pueden invocar métodos de clase (static)

No se puede llamar un método de instancia sin “instanciar” primero un objeto

Propiedad this y notación punto(.)

La palabra clave **this** en java se utiliza para hacer referencia al objeto actual. Con la palabra **this**, podemos referenciar métodos y variables del objeto actual. Uno de los momentos en los cuales es común utilizar **this**, es en los métodos seters

Ej:

```
String marca;
String placa;
public void setMarca(String marca){
this.marca=marca
}
public void setPlaca(String placa){
  this.placa=placa
}
```

Error de uso

```
int nroPuestos
public void setNroPuestos(int nroPuestos){
        nroPuestos=nroPuestos;
    }
```

Sesión 19

- La Clase String

Pontificia Universidad
JAVERIANA
Colombia

Clase String

- Un objeto *String* representa una cadena alfanumérica, cuya posición de memoria no puede ser alterada durante la ejecución del programa. Es decir los objetos de la clase String son inmutables, cuando se reasigna un valor a un objeto String, lo que se obtiene como resultado es un objeto nuevo
- String cadena = “Hola Mundo”;
- cadena =“cadena cambia de valor”;

Cadenas de Caracteres (String)

```
System.out.println("This is \na string constant!");
```

```
This is
a string constant!
```

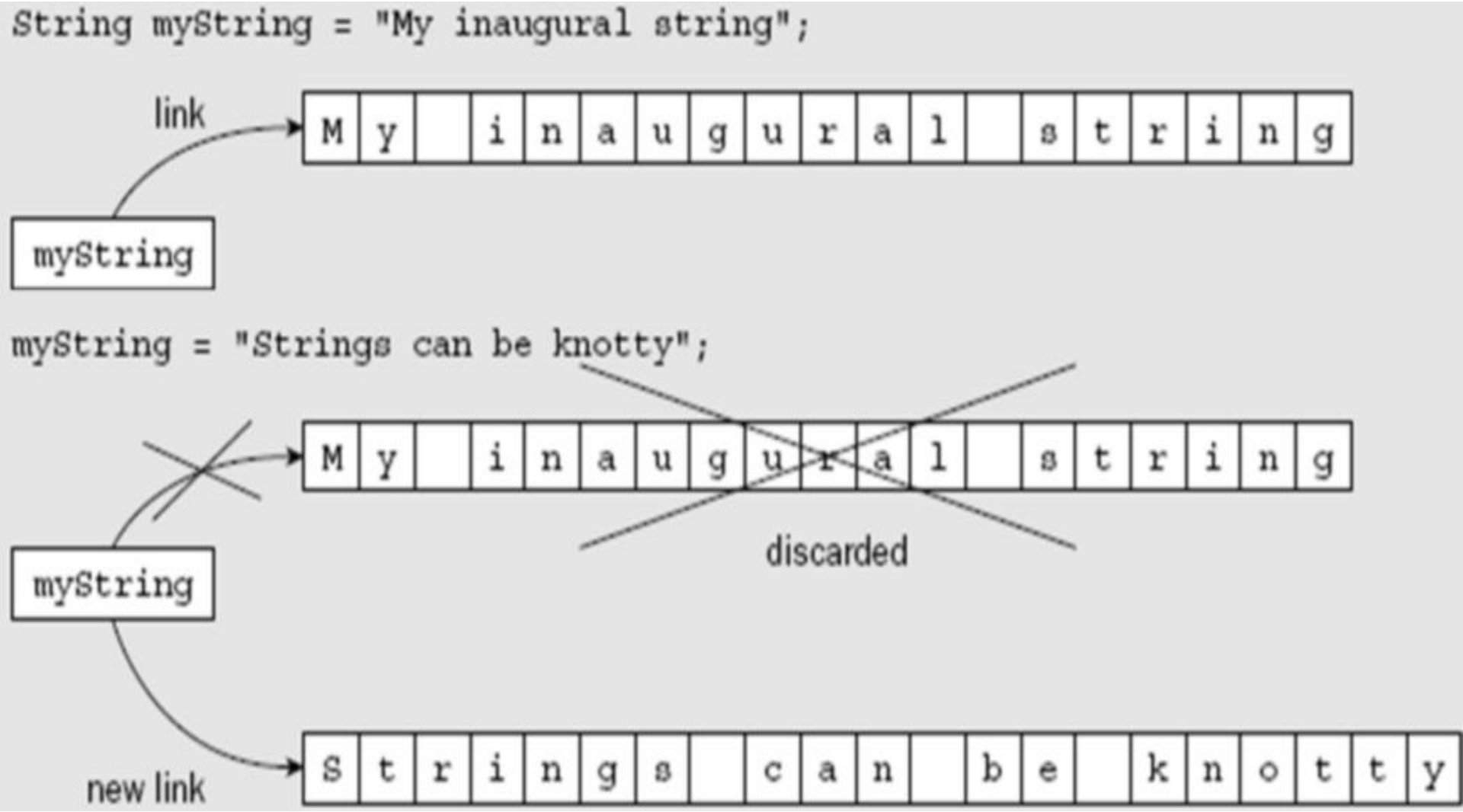
Arreglos de caracteres

```
char[] message = new char[50];
```

```
char[] vowels = { 'a', 'e', 'i', 'o', 'u'};
```

```
char[] sign = {'F', 'l', 'u', 'e', 'n', 't', ' ',
               'G', 'i', 'b', 'b', 'e', 'r', 'i', 's', 'h', ' ',
               's', 'p', 'o', 'k', 'e', 'n', ' ',
               'h', 'e', 'r', 'e'};
```

Cadenas de Caracteres (String)



Arreglos de Cadenas

```
String[] names = new String[5];
```

```
String[] colors = {'red', 'orange', 'yellow', 'green'};
```

El método main recibe un arreglo de cadenas con parametros de ejecución

```
public static void main(String[] args) {
   // Code for method...
}
```

Operaciones con cadenas

int indexOf(char c)	**Devuelve la posición de un carácter dentro de la cadena de texto. En el caso de que el carácter buscado no exista nos devolverá un -1. Si lo encuentra nos devuelve un número entero con la posición que ocupa en la cadena.**
int indexOf(char c, int fromIndex)	**Realiza la misma operación que el anterior método, pero en vez de hacerlo a lo largo de toda la cadena lo hace desde el índice (fromIndex) que le indiquemos.**
int lastIndexOf(char c)	**Nos indica cual es la última posición que ocupa un carácter dentro de una cadena. Si el carácter no está en la cadena devuelve un -1.**
int lastIndexOf(char c, int fromIndex)	**Lo mismo que el anterior, pero a partir de una posición indicada como argumento.**
int indexOf(String str)	**Busca una cadena dentro de la cadena origen. Devuelve un entero con el índice a partir del cual está la cadena localizada. Si no encuentra la cadena devuelve un -1**
int indexOf(String str, int fromIndex)	**Misma funcionalidad que indexOf(String str), pero a partir de un índice indicado como argumento del método.**
int lastIndexOf(String str)	**Si la cadena que buscamos se repite varias veces en la cadena origen podemos utilizar este método que nos indicará el índice donde empieza la última repetición de la cadena buscada**

Eckel, B. (2007). *Piensa en Java*. Pearson Educación. Capitulo 13. Cadenas de caracteres

Operaciones con cadenas

lastIndexOf(String str, int fromIndex)	**Lo mismo que el anterior, pero a partir de un índice pasado como argumento.**
String substring(int beginIndex)	**Este método nos devolverá la cadena que se encuentra entre el índice pasado como argumento (beginIndex) hasta el final de la cadena origen.**
String substring(int beginIndex, int endIndex)	**Este método devuelve la subcadena indicando el índice inicial y final del cual queremos obtener la cadena.**
String toLowerCase()	**Convierte todos los caracteres en minúsculas.**
String toUpperCase()	**Convierte todos los caracteres a mayúsculas**
String trim()	**Elimina los espacios en blanco de los extremos de la cadena**
String replace(char oldC, char newC)	**Este método lo utilizaremos cuando lo que queramos hacer sea el remplazar un carácter por otro. Se reemplazarán todos los caracteres encontrados.**

Eckel, B. (2007). *Piensa en Java*. Pearson Educación. Capitulo 13. Cadenas de caracteres

Operaciones con cadenas

String valueOf(boolean b); String valueOf(int i); String valueOf(long l); String valueOf(float f); String valueOf(double d); String valueOf(Object obj);	**Convierte un tipo de dato primitivo a String**
length()	**Numero de caracteres de la cadena**
charAt(int c)	**Devuelve el caracter indicado en la posicion int**
toCharArray ()	**Devuelve un arreglo de caracteres**
equals(string str)	**Devuelve falso o verdadero si los String son iguales (su contenido)**
compareTo (string str)	**El resultado es negativo, cero o positivo dependiendo de la comparacion lexicografica. Las mayusculas y minusculas no son iguales**
contains(string str)	**Devuelve falso o verdadero si un string contiene a otro**

Eckel, B. (2007). *Piensa en Java*. Pearson Educación. Capitulo 13. Cadenas de caracteres

Formateo de la salida

System.out.format: Similar al uso del printf en lenguaje C++

Ej:
System.out.format(("El 1er punto de la recta es %d, %d y el 2do punto es %d, %d y la pendiente es : %f\n"), x1, y1, x2, y2, pendiente);

String.format para alineación y redondeo

Ej:
System.out.format("%***N***s\n",variable); //Donde **N** es el numero de espacios a rellenar por la izquierda. Si N es negativo, rellena espacios por la derecha.

Ej:
System.out.format("%***N***.***M***f\n", variable); // Donde N es él numero máximo de enteros a mostrar y M es el numero de máximo de decimales a mostrar.

Convenciones ms comunes con **formatter**
%d: enteros y entero largo
%s: cadenas
%f: flotante y double

Equipo	Pos	Ptos	J	G	E	P	GF	GC	DG
America	1	4	2	1	1	0	5	4	1
Cali	2	1	1	0	1	0	2	2	0
Millonarios	3	0	1	0	0	1	2	3	-1

Separar cadenas y sustitución

split: Divide una cadena de caracteres en una matriz de objetos de String utilizando como delimitador la expresión regular.
Ej:

```
int n=0;
String palabras[];
String cadena= "Esta es una cadena de palabras";
palabras= cadena.split(" ");
n=palabras.length;
for (int i=0; i<n;i++){
    System.out.format ("Palabra [%d]:%s \n",i,palabras[i]);
}
```

Separar cadenas y sustitución

replaceFirst: Devuelve un string en el cual se reemplaza la primera aparición de una cadena de un string por otra.

replace y replaceAll: Devuelve un string en el cual se reemplazan todas las apariciones de una cadena de un string por otra.

Manejo de Fechas en Java

Las clases encargadas del manejo de fechas en Java son **java.util.date** y **java.util.Calendar** para versiones anteriores al JDK 8, posterior a esta version de JDK, se utiliza **LocalDate** y **LocalDateTime.**

Ver Ejemplo

Taller

Reglas:

1) No se permite código resuelto en internet
2) No esta permitido copiar código de otros grupos
3) Los programas que no estén correctamente indentados tendrán -1 punto
4) El plazo de entrega es 2 de Abril
5) Se deben entregar los archivo fuente .java
6) Al inicio de cada programa colocar los datos de los integrantes del grupo y su respectivo número

EJ: /*Pedro Perez, Mariana Gomez Grupo 3*/

7) Enviar un pantallazo de la ejecución del programa dentro de archivo Word (indicar integrantes del grupo) de lo contrario se resta un punto

Taller

1. Escribir un programa que lea una cadena de caracteres por teclado, luego lea una palabra por teclado, al final indique cuantas veces se encuentra la palabra en la cadena solicitada inicialmente

Ej:

Cadena=Pontificia Universidad Javeriana

Palabra= ia

Respuesta: 2 veces

Taller

2. Escriba un programa en java que lea una frase y a continuación muestre cada palabra de la frase, seguida del número de letras que tiene cada palabra.

Ej: Cadena=Pontificia Universidad Javeriana

Pontificia: 10

Universidad: 11

Javeriana:9

Sesión 20

- Excepciones

Excepciones

- Una excepción es un evento que ocurre durante la **ejecución** de un programa
- Detiene el flujo normal de la secuencia de instrucciones del programa
  - Es una condición anormal que surge en la ejecución de un programa.
- No hay errores de compilación.
- Las excepciones están diseñadas para la detección y corrección de errores y en general para el control de los eventos inusuales que deseemos controlar.

Excepciones

- Las aplicaciones deberían ser capaz de controlar errores y ejecutar una medida para continuar con su ejecución.
- Se debe manejar independientemente la lógica del programa del control de eventos y errores inesperados.
- No necesariamente todos los errores deban ser manejados con excepciones, las excepciones deben reservarse para las situaciones inusuales y catastróficas que puedan generarse.
  - Errores de input de usuario es un evento normal que se puede controlar sin recurrir a excepciones.

Excepciones

Situaciones que causan una excepción:

Errores de código y datos	Cast inválido a un objeto. Intentar acceder a una posición inválida de un arreglo. División por cero.
Excepciones estándar	Excepciones que pueden arrojar las clases de la biblioteca de Java.
Arrojar excepciones propias	Crear excepciones propias
Errores de Java	Pueden surgir errores ejecutando la JVM, quien corre nuestros programas compilados, usualmente generados por un error en el programa

Excepciones

```
try {
      //código susceptible de lanzar una excepcion
}
catch (tipo_exception e)
{
      //código de tratamiento de la excepcion
}
catch (tipo_exception e)
{
      //código de tratamiento de la excepcion
}
catch (Exception e){
 //código de tratamiento general de la excepcion
}
finally
{
      //código que se ejecuta siempre
}
```

Excepciones

try

- Es el bloque de código donde se espera se produzca una excepción.
- El bloque **try** tiene que ir seguido, al menos, por una cláusula **catch** o una cláusula **finally**.

catch

- Es el código que se ejecuta cuando se genera la excepción. En este bloque controlamos cualquier excepción que coincida con el tipo de excepción manejada
- Para evitar la interrupción del programa, este bloque no debe generar errores.
- Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No es una buena practica intentar capturar todas las excepciones con una sola cláusula, como esta:

  catch(Exception e) {
- El uso de la clase Exception debe ser la ultima opción en el manejo de errores.
- El uso indiscriminado de la clase Exception presentaría un uso demasiado general, y podría ocasionar muchas más excepciones de las esperadas

Excepciones

finally

- El bloque finally sirve para colocar en él instrucciones que se desea que se realicen siempre que se sale de un bloque try.

- Es opcional, pero pasa a ser obligatorio cuando el bloque try no va seguido por ninguna cláusula catch.

Excepciones

```
int arreglo[] =  new int [10];
…
for (int i=0 ; i<=arreglo.length ;i++ )  {
System.out.println(arreglo[i]);
}
```

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 10
at Paquete1.Class3.main(Class3.java:19)
```

Excepciones

- En Java, una excepción es un objeto que es creado cuando sucede una situación inusual en el programa.
- Class Exception

  java.lang.Object

  java.lang.Throwable

  java.lang.Exception
- Este objeto contiene campos que almacenan información sobre la naturaleza del problema.
- https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Exception.html

Excepciones

Class Name	Exception Condition Represented
`ArithmeticException`	An invalid arithmetic condition has arisen, such as an attempt to divide an integer value by zero.
`IndexOutOfBoundsException`	You've attempted to use an index that is outside the bounds of the object it is applied to. This may be an array, a `String` object, or a `Vector` object. The `Vector` class is defined in the standard package `java.util`. You will be looking into the `Vector` class in Chapter 14.
`NegativeArraySizeException`	You tried to define an array with a negative dimension.
`NullPointerException`	You used an object variable containing `null`, when it should refer to an object for proper operation — for example, calling a method or accessing a data member.
`ArrayStoreException`	You've attempted to store an object in an array that isn't permitted for the array type.
`ClassCastException`	You've tried to cast an object to an invalid type — the object isn't of the class specified, nor is it a subclass or a superclass of the class specified.
`IllegalArgumentException`	You've passed an argument to a method that doesn't correspond with the parameter type.

Excepción

En resumen

- Una excepción es lanzada y debe ser capturada.
- Cuando sucede el evento o circunstancia el objeto (que identifica la excepción) es lanzado como un argumento a una pieza de código que ha sido escrita específicamente para capturar y manejar este tipo de problema.
- Lanzar throw
- Capturar catch

Manejo de excepciones

```
int x=2, y=2;
int z= x/(x-y);
System.out.println(z);
```

Exception in thread "main"
java.lang.ArithmeticException: / by zero

Manejo de Excepciones

```
int x=2, y=2, z;
try{
   z= x/(x-y);
   System.out.println(z);
}
catch(Exception e){
   System.out.println("Esta intentando dividir por cero");
}
finally{
   //Código adicional que desee incluir
   System.out.println("Esto se muestra siempre");
}
```

Esta intentando dividir por cero
Esto se muestra siempre

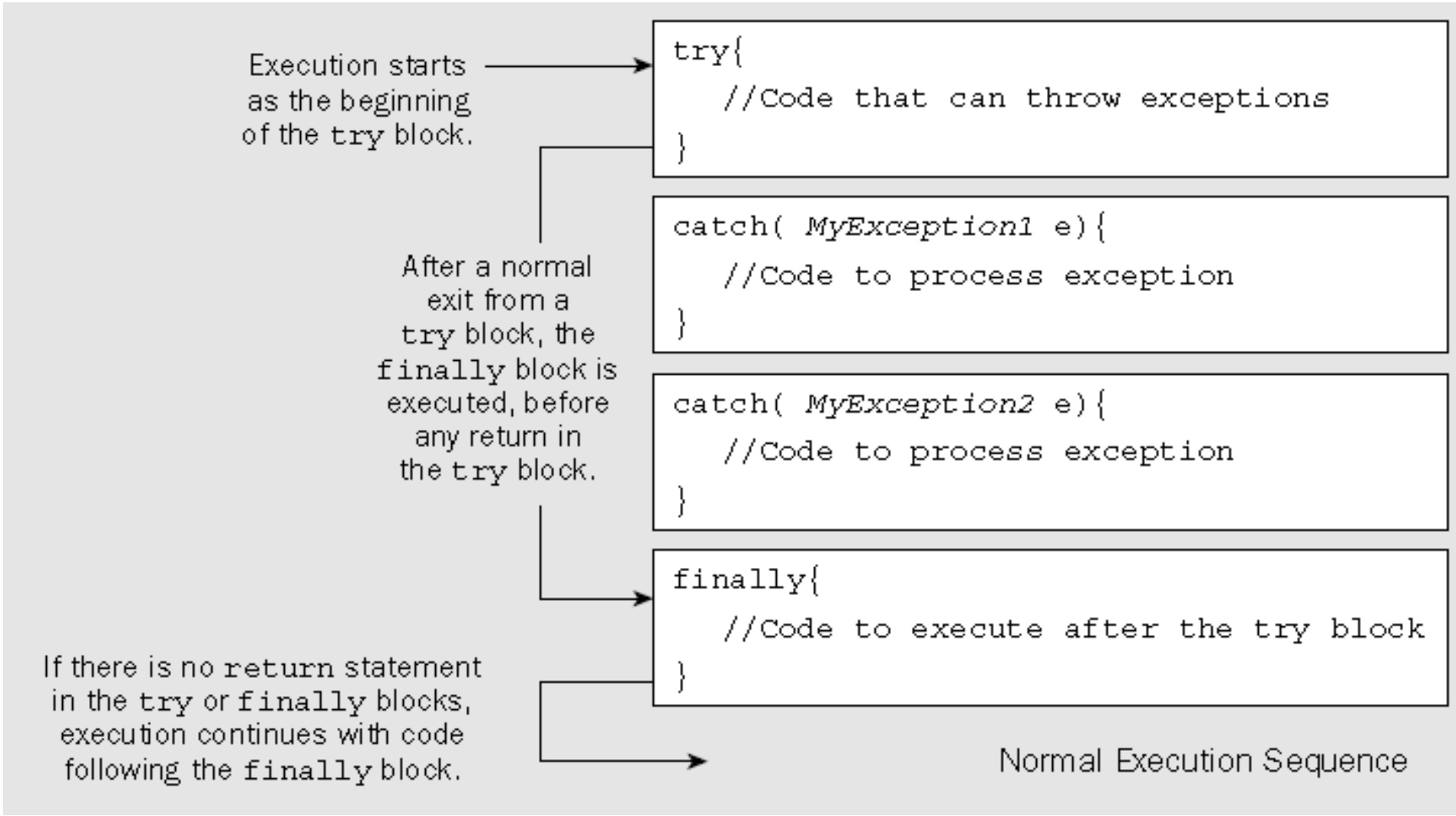
Manejo de Excepciones

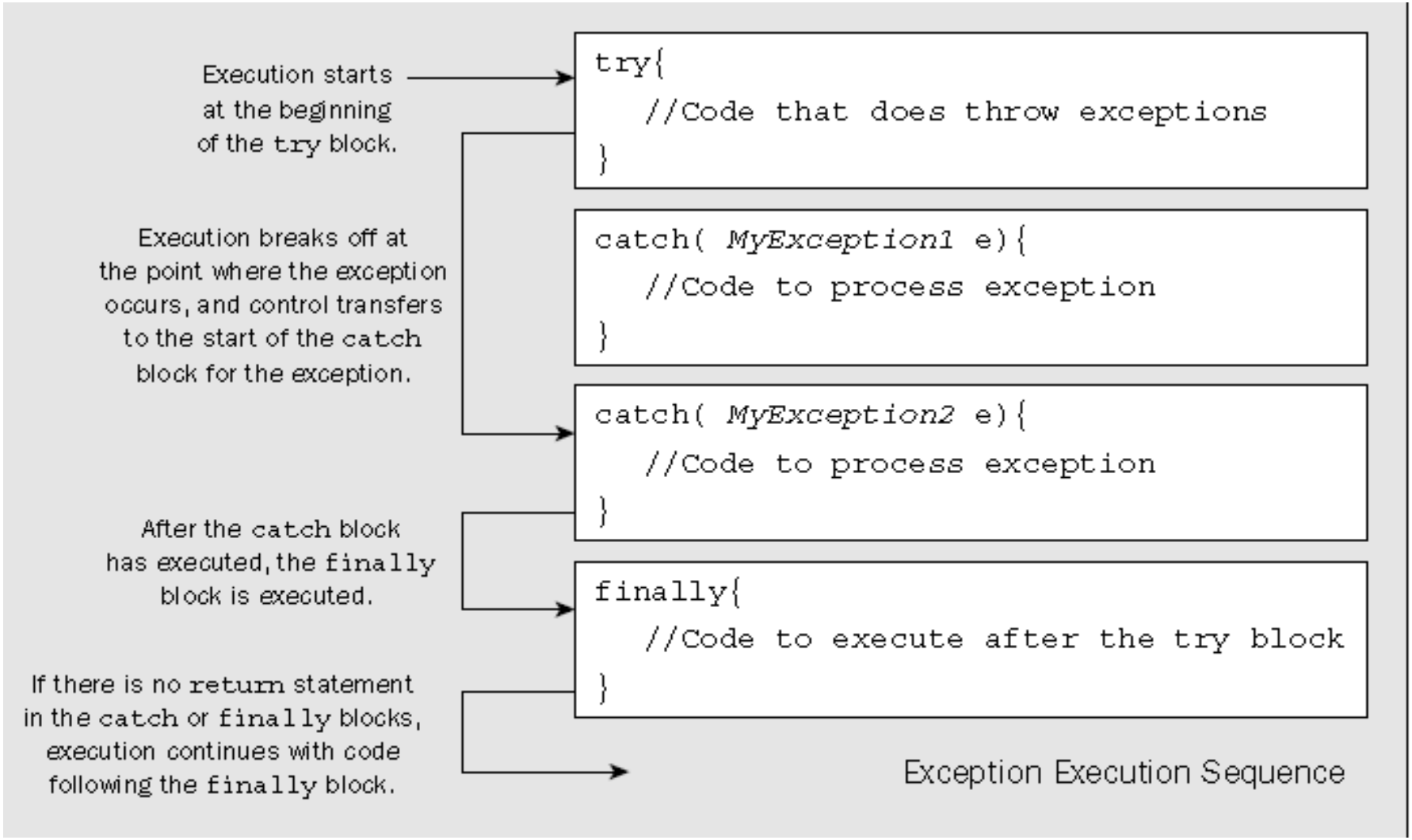
```
int x=2, y=2, z;
try{
    z= x/(x-y);
    System.out.println(z);
}
catch(ArithmeticException e){
    System.out.println("Esta intentando dividir por cero");
}
catch(Exception e){
    System.out.println("Ocurrio un error realizando al
     división");
}
finally{
    //Código adicional que desee incluir
    System.out.println("Esto se muestra siempre");
}
```

Particular

General

Manejo de Excepciones




Execution starts at the beginning of the try block.
try{
//Code that does throw exceptions
}
Execution breaks off at the point where the exception occurs, and control transfers to the start of the catch block for the exception.
catch(MyException1 e){
//Code to process exception
}
catch(MyException2 e){
//Code to process exception
}
After the catch block has executed, the finally block is executed.
finally{
//Code to execute after the try block
}
If there is no return statement in the catch or finally blocks, execution continues with code following the finally block.
Exception Execution Sequence

Ejercicio

- ¿Qué pasa si se ejecuta el siguiente código?

```
String cadena = "123a";
Integer entero = Integer.parseInt(cadena);
System.out.println("El numero es "+entero);
```

- ¿Cómo lo puedo controlar?

Ejercicio

- ¿Qué pasa si se ejecuta el siguiente código?
  - Se genera un error:

```
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "123a"
    at
    java.lang.NumberFormatException.forInputString(NumberFo
    rmatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:456)
    at java.lang.Integer.parseInt(Integer.java:497)
    at Paquete1.Class3.main(Class3.java:41)
```

Ejercicio

- ¿Cómo lo puedo controlar?

```
String cadena = "123a";
try{
    Integer entero = Integer.parseInt(cadena);
}
catch(NumberFormatException e){
    …
}
```

Manejo de excepciones

Relanzar excepciones:

```
try {
// Code that originates an arithmetic exception
} catch(ArithmeticException e) {
throw e;
}
```

Manejo de excepciones

Desde nuestros métodos podemos lanzar excepciones para que otro objeto se encargue de controlarla.

```
public static void main(String args[]){
    metodo2();
}
public static void metodo2()  {
    try{
        metodo1();
    }
    catch(NumberFormatException e){
        System.out.println("Excepcion al invocar al metodo1 :" + e.getMessage());

    }
 }

public static void metodo1() throws NumberFormatException {
    String cadena = "123a";
    try {
        Integer entero = Integer.parseInt(cadena);
    } catch (NumberFormatException e) {
        System.out.println("Aqui metodo1 maneja el error");
        throw e;
    }
 }
```

Excepciones comprobadas y no comprobadas

•**Excepciones comprobadas:** Son excepciones que el compilador de Java te obliga a manejar explícitamente (bloque try catch). Si el código debe lanzar una excepción comprobada, debes definirla en la clausula **throws**

•Ejemplos:

- IOException: Ocurre al trabajar con archivos o flujos de entrada/salida, como al intentar leer un archivo que no existe.
- SQLException: Se produce al interactuar con bases de datos, como al intentar ejecutar una consulta SQL inválida.
- ClassNotFoundException: Surge cuando intentas cargar una clase que no se encuentra en el classpath.

•**Excepciones no comprobadas:** No esta obligado a manejarlas. Suelen ser errores de programación, como acceder a un elemento de un array que no existe.

- La mayoría de las excepciones no comprobadas son subclases de RuntimeException. Incluyen errores de programación comunes, como:
- NullPointerException: Se lanza cuando intenta acceder a un miembro de un objeto que es null.
- ArrayIndexOutOfBoundsException: Ocurre cuando intenta acceder a un índice de un array que está fuera de sus límites.
- IllegalArgumentException: Se produce cuando pasa un argumento inválido a un método.

La clase Throwable

- La clase provee una serie de métodos que permiten acceder a los mensajes de error y al *stack trace*.
- El *execution stack* (pila de ejecución) sigue la pista de todos los métodos que están en ejecución en un momento determinado.
- El record del *execution stack* es almacenado en el objeto de excepción que consiste del número de línea en el código fuente donde se originó la excepción un rastro seguido de las llamadas a los métodos hasta el punto en el que se produjo la excepción.
- Esto le ayudará a entender dónde y cómo se originó la excepción.

La clase Throwable

Method	Description
`getMessage()`	This returns the contents of the message, describing the current exception. This will typically be the fully qualified name of the exception class (it will be a subclass of `Throwable`) and a brief description of the exception.
`printStackTrace()`	This will output the message and the stack trace to the standard error output stream — which is the screen in the case of a console program.
`printStackTrace(PrintStream s)`	This is the same as the previous method except that you specify the output stream as an argument. Calling the previous method for an exception object `e` is equivalent to: `e.printStackTrace(System.err);`

Ejercicio

```
public void metodo2() {
      try{
         metodo1();
      }
      catch(NumberFormatException e){
         System.out.println("Excepcion al invocar al metodo1");
         System.out.println("Llamando al metodo getMessage: "+e. getMessage());
         System.out.println("La pila de ejecucion es: ");
         e.printStackTrace();

      }
  }
```

Ejercicio

Excepcion al invocar al metodo1
Llamando al metodo getMessage: For input string: "123a“
La pila de ejecucion es:
java.lang.NumberFormatException: For input string: "123a"
at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:456)
at java.lang.Integer.parseInt(Integer.java:497)
at Paquete1.Class3.metodo1(Class3.java:10)
at Paquete1.Class3.metodo2(Class3.java:19)
at Paquete1.Class3.main(Class3.java:33)

Excepciones comprobadas y no comprobadas

•**Excepciones comprobadas:** Son excepciones que el compilador de Java te obliga a manejar explícitamente (bloque try catch). Si el código debe lanzar una excepción comprobada, debes definirla en la clausula **throws**

•Ejemplos:

- IOException: Ocurre al trabajar con archivos o flujos de entrada/salida, como al intentar leer un archivo que no existe.
- SQLException: Se produce al interactuar con bases de datos, como al intentar ejecutar una consulta SQL inválida.
- ClassNotFoundException: Surge cuando intentas cargar una clase que no se encuentra en el classpath.

•**Excepciones no comprobadas:** No esta obligado a manejarlas. Suelen ser errores de programación, como acceder a un elemento de un array que no existe.

- La mayoría de las excepciones no comprobadas son subclases de RuntimeException. Incluyen errores de programación comunes, como:
- NullPointerException: Se lanza cuando intenta acceder a un miembro de un objeto que es null.
- ArrayIndexOutOfBoundsException: Ocurre cuando intenta acceder a un índice de un array que está fuera de sus límites.
- IllegalArgumentException: Se produce cuando pasa un argumento inválido a un método.

Definición de excepciones propias

Para crear sus propias clases de manejo de excepciones se debe:

1. Crear una clase con el nombre de su excepción, la cual extiende de la clase exception.
2. Utilizar la clase de excepción creada en un bloque try catch

```
public class YEsCero extends Exception {
    public YEsCero(String msg) {
        super(msg);
    }
}
```

```
public class LanzandoExcepciones  {
    public float calcular1(int x, int y)throws YEsCero{
        float r=0;
        if (y==0){
            throw new YEsCero (" 'Y' no puede ser igual a cero \n");
        }
        else{
            r=x/y;
        }
        return r;
    }
}
```

```
int x=3, y=0;
float r=0f;
LanzandoExcepciones lz= new LanzandoExcepciones();
try{
      r=lz.calcular1(x, y);
 }
 catch(YEsCero ex){
     System.out.println( ex.getMessage());
 }
```

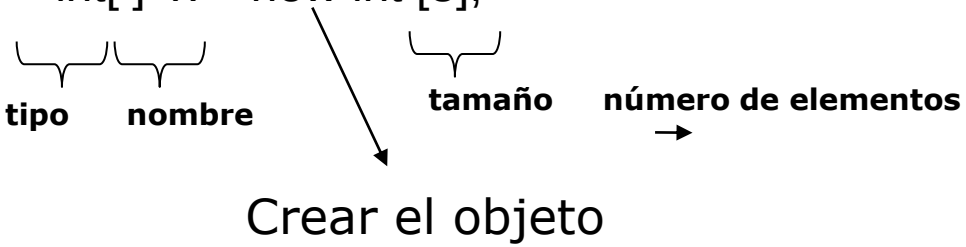
Arreglos

Los arreglos tienen la misma funcionalidad vista durante las sesiones de lenguaje C++ y su declaración es muy similar a la declaración con memoria dinámica

tipoDato (o clase)[ ] variable = new tipoDato (o clase)[tamaño];

Ej:

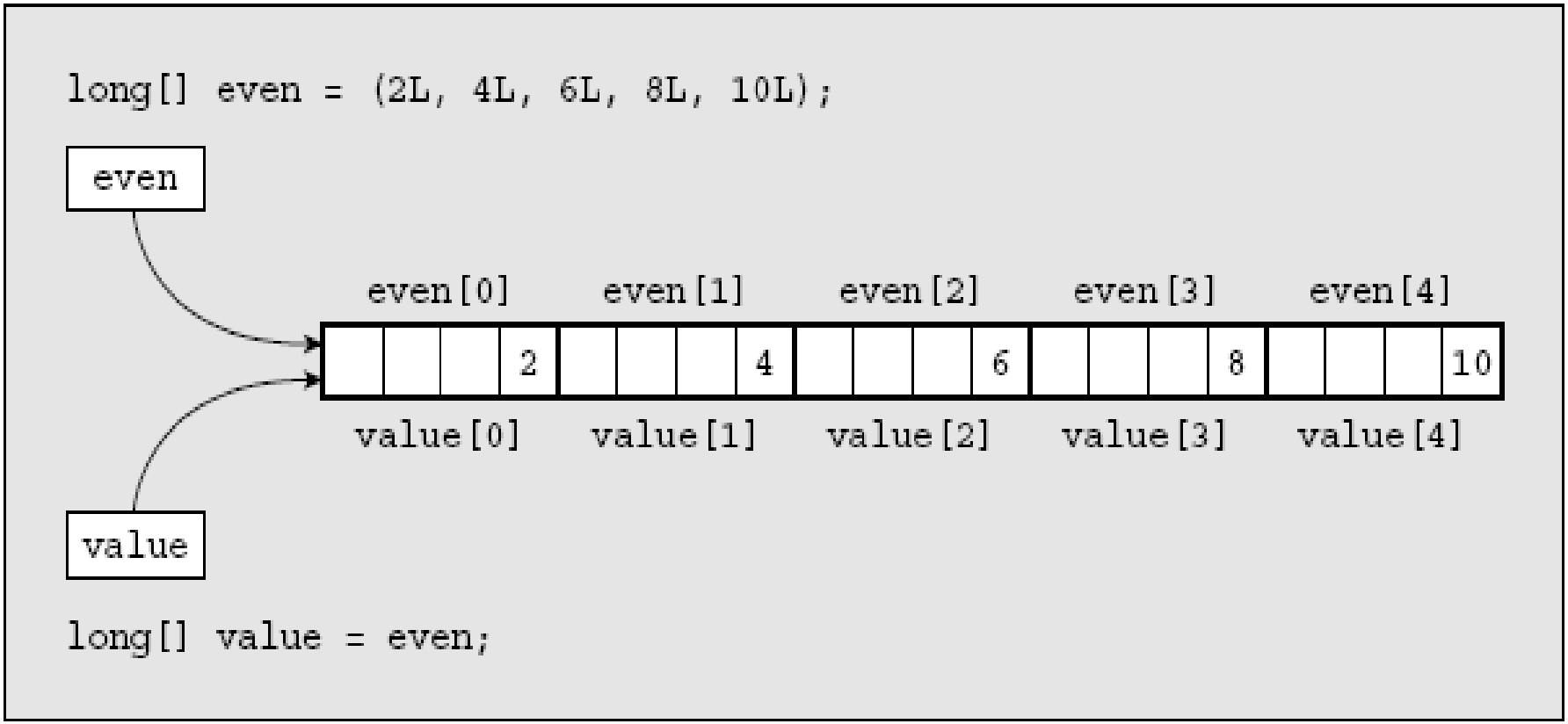
int[ ] x = new int [8];



2	100	13	25	0	-15	39	485
0	1	2	3	4	5	6	7

Arreglos (Inicialización)

```
long[] even = {2L, 4L, 6L, 8L, 10L};
long[] value = even;
```



Recomendación

Capitulo 12 del libro **Piensa en Java** de Bruce Eckel – **Tratamiento de excepciones**

Taller Excepciones y Arreglos

Reglas:

1) No se permite código resuelto en internet
2) No esta permitido copiar código de otros grupos
3) Los programas que no estén correctamente indentados tendrán -1 punto
4) Se deben entregar los archivo fuente .java
5) Al inicio de cada programa colocar los datos de los integrantes del grupo y su respectivo número

EJ: /*Pedro Perez, Mariana Gomez Grupo 3*/

7) Enviar un pantallazo de la ejecución del programa dentro de archivo Word (indicar integrantes del grupo) ej:

```
Indique lado 2:
5
El area del cuadrilatero es: 25.0
-----------------------------------
Menu de opciones
```

Ejercicio

- Implementar una calculadora que permita sumar, restar, multiplicar, dividir, hallar el factorial de un número y potencia (x^y).
- El programa debe controlar las excepciones que se puedan presentar por ingreso de datos errados (ej. Letras)

Ejercicio

- Elaborar una agenda que permita almacenar nombres de contactos con sus respectivos teléfonos y correos electrónicos utilizando arreglos.
- Utilice un arreglo y una clase de tipo Persona para almacenar Nombre, Telefono y correo.
- Realice la lectura hasta que el nombre sea “fin”
- El numero máximo de registros de la agenda es de 100.
- Imprima por pantalla los datos almacenados; tener en cuenta que no se deben mostrar los registros del arreglo que no tienen información. Es decir, el arreglo es de 100 posiciones, si hay 5 registros solo debe mostrar 5 registros.

Sesión

- Archivos
- Wrappers

Pontificia Universidad
JAVERIANA
Colombia

Archivos y Directorios

- El primer objeto con el cual interactuamos para manejo de archivos corresponde a la clase File. La clase File no representa (según su nombre) un archivo, sino que también pueden referenciar un directorio
- Un objeto File no representa al archivo físico, sino una referencia a él
- Crear un objeto File no determina que el archivo o directorio realmente exista
- Ejemplos:

```
File myDir = new File("C:/jdk1.5.0/src/java/io");
```

```
File myDir = new File("C:\\jdk1.5.0\\src\\java\\io");
```

```
File myFile = new File("C:/jdk1.5.0/src/java/io/File.java");
```

- Si el path es incorrecto, sólo hasta ejecución se detecta

Archivos y Directorios

Determinar el directorio de trabajo: Para esto utilizamos la clase Path. Necesitamos importar las siguientes librerías:

```
import java.nio.file.Path;
import java.nio.file.Paths;
Ej:
public static void main (String args[]){
        Path path = Paths.get("");
        String directoryName = path.toAbsolutePath().toString();
        System.out.println("Directorio actual = " +directoryName);
}
```

Archivos y Directorios

Crear una carpeta o directorio.

La creación de una carpeta se realiza con el método mkdir.

Ej:

```
public static void main (String args[]){
        Path path = Paths.get("");
        String directoryName = path.toAbsolutePath().toString();
                System.out.println("Directorio actual = " +directoryName);

        File carpeta = new File(directoryName+File.separator+"datos");
        if (!carpeta.exists()) {
            if (carpeta.mkdirs()) {
                System.out.println("Carpeta creada");
            } else {
                System.out.println("Error al crear carpeta " + carpeta);
            }
        }else{
            System.out.println("Carpeta ya existe: "+carpeta);
        }
    }
```

Archivos y Directorios

- Creación de objeto File con parent directory (directorio donde está el archivo)

```
File myDir = new File("C:/jdk1.5.0/src/java/io");    // Parent directory
File myFile = new File(myDir, "File.java");          // The path to the file
```

- Mismo resultado:

```
File myFile = new File("C:/jdk1.5.0/src/java/io", "File.java");
```

Universal Naming Convention (UNC)

- Se puede evitar la dependencia del prefijo en Windows usando UNC
- Windows: \\server_name\directory_path\filename
- Otros que soporten UNC:
  - //server_name/directory_path/filename

```
File myFile = new File("//myPC/shared/jdk1.5.0/src/java/io", "File.java");
```

```
File myFile = new File(File.separator + File.separator + "myPC" +
                       File.separator + "shared" + File.separator +
                       "jdk1.4" + File.separator + "src" + File.separator +
                       "java" + File.separator + "io", "File.java");
```

Propiedades del Sistema

- Puede ser usado para obtener información sobre directorios de usuario donde sea conveniente almacenar archivos

- User.dir: directorio base para paths relativos

```
String currentDir = System.getProperty("user.dir");
```

```
File dataFile = new File(currentDir, "output.txt");
```

- User.home: directorio home del usuario (p.e. Documents and Settings\Ana Lozano\...)

```
File dataFile = new File(System.getProperty("user.home"), "output.txt");
```

Propiedades del Sistema

```
public class TryProperties {
  public static void main(String[] args) {
    java.util.Properties properties = System.getProperties();
    properties.list(System.out);
  }
}
```

- Es posible cambiar las propiedades para la ejecución del programa

```
String oldValue = System.clearProperty("java.class.path");
```

```
System.setProperty("user.dir", "C:/MyProg");
```

Métodos de la clase File

Method	Description
`getName()`	Returns a `String` object containing the name of the file without the path — in other words, the last name in the path stored in the object. For a `File` object representing a directory, just the directory name is returned.
`getPath()`	Returns a `String` object containing the path for the `File` object — including the file or directory name.
`isAbsolute()`	Returns `true` if the `File` object refers to an absolute pathname, and `false` otherwise. Under MS Windows, an absolute pathname begins with either a drive letter followed by a colon and then a backslash or a double backslash. Under Unix, an absolute path is specified from the root directory down.
`getParent()`	Returns a `String` object containing the name of the parent directory of the file or directory represented by the current `File` object. This will be the original path without the last name. The method returns `null` if there is no parent specified. This will be the case if the `File` object was created for a file in the current directory by just using a file name.

Métodos de la clase File

`getParentFile()`	Returns the parent directory as a `File` object, or `null` if this File object does not have a parent.
`toString()`	Returns a `String` representation of the current `File` object and is called automatically when a `File` object is concatenated with a `String` object. You have used this method implicitly in output statements. The string that is returned is the same as that returned by the `getPath()` method.
`hashCode()`	Returns a hashcode value for the current `File` object.
`equals()`	You use this method for comparing two `File` objects for equality. If the `File` object passed as an argument to the method has the same path as the current object, the method returns `true`. Otherwise, it returns `false`.

Examinar archivo o directorio

Method	Description
`exists()`	Returns `true` if the file or directory referred to by the `File` object exists and `false` otherwise.
`isDirectory()`	Returns `true` if the `File` object refers to an existing directory and `false` otherwise.
`isFile()`	Returns `true` if the `File` object refers to an existing file and `false` otherwise.
`isHidden()`	Returns `true` if the `File` object refers to a file that is hidden and `false` otherwise. How a file is hidden is system-dependent. Under Unix a hidden file has a name that begins with a dot. Under Windows a file is hidden if it is marked as such within the file system.
`canRead()`	Returns `true` if you are permitted to read the file referred to by the `File` object and `false` otherwise. This method can throw a `SecurityException` if read access to the file is not permitted.

Examinar archivo o directorio

`canWrite()`	Returns `true` if you are permitted to write to the file referred to by the `File` object and `false` otherwise. This method may also throw a `SecurityException` if you are not allowed to write to the file.
`getAbsolutePath()`	Returns the absolute path for the directory or file referenced by the current `File` object. If the object contains an absolute path, then the string returned by `getPath()` is returned. Otherwise, under MS Windows the path is resolved against the current directory for the drive identified by the pathname, or against the current user directory if no drive letter appears in the pathname, and against the current user directory under Unix.
`getAbsoluteFile()`	Returns a `File` object containing the absolute path for the directory or file referenced by the current `File` object.

Examinar archivo o directorio

Method	Description
`list()`	If the current `File` object represents a directory, a `String` array is returned containing the names of the members of the directory. If the directory is empty, the array will be empty. If the current `File` object is a file, `null` is returned. The method will throw an exception of type `SecurityException` if access to the directory is not authorized.
`listFiles()`	If the object for which this method is called is a directory, it returns an array of `File` objects corresponding to the files and directories in that directory. If the directory is empty, then the array that is returned will be empty. The method will return `null` if the object is not a directory, or if an I/O error occurs. The method will throw an exception of type `SecurityException` if access to the directory is not authorized.
`length()`	Returns a value of type `long` that is the length, in bytes, of the file represented by the current `File` object. If the pathname for the current object references a file that does not exist, then the method will return zero. If the pathname refers to a directory, then the value returned is undefined.
`lastModified()`	Returns a value of type `long` that represents the time that the directory or file represented by the current `File` object was last modified. This time is the number of milliseconds since midnight on 1st January 1970 GMT. It returns zero if the file does not exist.

Creación de un archivo texto

```
File myFile= new File (carpeta,"datos.txt");
System.out.println("Archivo existe: "+ myFile.exists());
if (!myFile.exists()){
    try{
        myFile.createNewFile();
     } catch (IOException e) {
         System.out.println("No se logro crear el archivo");
         e.printStackTrace();
     }
}
```

Ejemplo

```
System.out.println("getPath()           " + myFile.getPath());
try{
   System.out.println("getCanonicalPath()  " + myFile.getCanonicalPath() );
} catch (IOException e) {
        System.out.println("No se logro crear el archivo");
        e.printStackTrace();
}

System.out.println("Es archivo? " + myFile.isFile());
System.out.println("Esta oculto? " + myFile.isHidden());
System.out.println("Se puede leer? " + myFile.canRead() );
System.out.println("Se puede escribir? " + myFile.canWrite() );
System.out.println("Tamaño:" + myFile.length() + " bytes");
```

Crear y modificar archivos o directorios

Method	Description
`renameTo(File path)`	The file represented by the current object will be renamed to the path represented by the `File` object passed as an argument to the method. Note that this does *not* change the current `File` object in your program — it alters the physical file. Thus, the file that the `File` object represents will no longer exist after executing this method, because the file will have a new name and possibly will be located in a different directory. If the file's directory in the new path is different from the original, the file will be moved. The method will fail if the directory in the new path for the file does not exist, or if you don't have write access to it. If the operation is successful, the method will return `true`. Otherwise, it will return `false`.
`setReadOnly()`	Sets the file represented by the current object as read-only and returns `true` if the operation is successful.

Crear y modificar archivos o directorios

`mkdir()`	Creates a directory with the path specified by the current `File` object. The method will fail if the parent directory of the directory to be created does not already exist. The method returns `true` if it is successful and `false` otherwise.
`mkdirs()`	Creates the directory represented by the current `File` object, including any parent directories that are required. It returns `true` if the new directory is created successfully and `false` otherwise. Note that even if the method fails, some of the directories in the path may have been created.
`delete()`	This will delete the file or directory represented by the current `File` object and return `true` if the delete was successful. It won't delete directories that are not empty. To delete a directory, you must first delete the files it contains.
`deleteOnExit()`	Causes the file or directory represented by the current `File` object to be deleted when the program ends. This method does not return a value. The deletion will be attempted only if the JVM terminates normally. Once you call the method for a `File` object, the delete operation is irrevocable, so you will need to be cautious with this method.

Escribiendo texto en un archivo

Existen diferentes clases para escribir texto en archivos algunas de estas son:

- **FileWriter** : Clase para escribir archivos de caracteres. Los constructores de esta clase asumen que la codificación de caracteres predeterminada y el tamaño de búfer de bytes predeterminado son aceptables.
- **BufferedWriter** :Escribe texto en una secuencia de salida de caracteres, almacenando caracteres en búfer para proporcionar la escritura eficiente de caracteres individuales, matrices y cadenas. Se puede especificar el tamaño del búfer o se puede aceptar el tamaño predeterminado. El valor predeterminado es lo suficientemente grande para la mayoría de los propósitos.

```
FileWriter outFile = new FileWriter(carpeta.getAbsolutePath()+File.separator+"Data1.txt",true);
BufferedWriter bWriter = new BufferedWriter(outFile);
do{
        s= sc.nextLine();
        if (!s.toLowerCase().equals("fin"))
                bWriter.write(s+"\n");
}while (!s.toLowerCase().equals("fin"));

bWriter.close();
outFile.close();
```

Leyendo texto de un archivo

Existen diferentes clases para escribir texto en archivos algunas de estas son:

- **FileReader** : Clase para leer archivos de caracteres. Los constructores de esta clase asumen que la codificación de caracteres predeterminada y el tamaño de búfer de bytes predeterminado son aceptables.
- **BufferedReader** :Lee texto de una secuencia de salida de caracteres, almacenando caracteres en búfer. El valor predeterminado es lo suficientemente grande para la mayoría de los propósitos.

```
FileReader freader = new FileReader(carpeta.getAbsolutePath()+File.separator+"Data1.txt");
BufferedReader br = new BufferedReader(freader);
String s;
System.out.println("Contenido del archivo Data1.txt" );
while((s = br.readLine()) != null) {
   System.out.println(s);
}
br.close();
freader.close();
```

Escribiendo Objetos en un archivo

Para escribir en un archivo binario que contiene objetos, es necesario crear una clase que extienda de la clase **ObjectOutputStream** y sobreescribir el método **writeStreamHeader**

```
public class AppObjectOutputStream extends ObjectOutputStream {
    public AppObjectOutputStream(OutputStream out) throws IOException{
            super(out);

    }
  protected void writeStreamHeader() throws IOException {
    // do not write a header
  }

}
```

```
if (!archivoExiste)
    obj1=new ObjectOutputStream(archivo);
 else
     obj1=new AppObjectOutputStream(archivo);
```

Escribiendo Objetos en un archivo

Existen diferentes clases para escribir (serializar) objetos en archivos algunas de estas son:

- **FileOutputStream** : Un flujo de salida de archivo es un flujo de salida para escribir datos en un archivo. Si un archivo está disponible para escritura o puede crearse dependiendo de la plataforma algunas de ellas en particular, permiten que un archivo se abra para escritura por solo un FileOutputStream (u otro objeto de escritura de archivos) a la vez. En tales situaciones, los constructores de esta clase fallarán si el archivo involucrado ya está abierto.

  FileOutputStream está diseñado para escribir flujos de bytes sin procesar, como datos de imagen u objetos.

- **ObjectOutputStream** : Un ObjectOutputStream escribe tipos de datos primitivos y gráficos de objetos Java en un OutputStream. Los objetos se pueden leer (reconstituir) utilizando un ObjectInputStream. El almacenamiento persistente de objetos se puede lograr utilizando un archivo para la transmisión.

  Solo los objetos que admiten la interfaz java.io.Serializable se pueden escribir en utilizando **ObjectOutputStream**.

```
FileOutputStream  archivo = new
FileOutputStream(carpeta.getAbsolutePath()+File.separator+"Animales.bin");
ArrayList<Animales> animales = new ArrayList<Animales>();
animales.add(new Animales ("insecto","saltamontes",false,0.5f,0.3f));
animales.add(new Animales ("mamifero","perro",false,20,15));
animales.add(new Animales ("reptil","cocodrilo",false,140,15));
ObjectOutputStream obj1 = new ObjectOutputStream(archivo );
Animales a =null;
Iterator it = animales.iterator();

while(it.hasNext()){
   a=(Animales)it.next();
   obj1.writeObject(a);
}
```

Leyendo Objetos de un archivo

Existen diferentes clases para leer (deserializar) objetos en archivos algunas de estas son:

- **FileInputStream** : Un FileInputStream obtiene bytes de entrada de un archivo en un sistema de archivos. Los archivos disponibles dependen del entorno del host. FileInputStream está diseñado para leer flujos de bytes sin procesar, como datos de imagen u objetos. Para leer secuencias de caracteres, considere usar FileReader.

- **ObjectInputStream** : Un ObjectInputStream deserializa datos primitivos y objetos escritos previamente usando un ObjectOutputStream. ObjectOutputStream y ObjectInputStream pueden proporcionar una aplicación con almacenamiento persistente para gráficos de objetos cuando se utilizan con FileOutputStream y FileInputStream respectivamente. ObjectInputStream se utiliza para recuperar aquellos objetos previamente serializados.

Leyendo Objetos de un archivo

```
FileInputStream  archivo = new FileInputStream(carpeta.getAbsolutePath()+File.separator+"Animales.bin");
ObjectInputStream obj1= new ObjectInputStream(archivo);
a=(Animales)obj1.readObject();
while (a!=null){
    animales.add(a);
    try{
        a=(Animales)obj1.readObject();
    }
    catch (EOFException e){
        a=null;
    }
}
a =null;
Iterator it = animales.iterator();
while(it.hasNext()){
    a=(Animales)it.next();
    System.out.println (a.toString());
}
```

Clases Wrapp

Los **Wrappers java** (envoltorios) son clases complemento de los tipos primitivos ya que estos son los únicos elementos de Java que no son objetos. Los wrap se utilizan frecuentemente para pasar de un tipo de dato a otro.

Tipo primitivo	**Wrapper Class**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Clase Integer

- Cada tipo numérico tiene su propia clase de objetos .
- int -> Integer
- Provee métodos para convertir de int a String y viceversa.
- Métodos útiles para manejar enteros.
  - Integer ()
  - public double doubleValue()
  - public float floatValue()
  - … intValue
  - … longValue
  - …shortValue
  - compareTo()
  - toString()
  - equals

Clase Long

- Cada tipo numérico tiene su propia clase de objetos .
- long -> Long
- Provee métodos para convertir de Long a String y viceversa.
- Métodos útiles para manejar Long.
  - Long ()
  - public double doubleValue()
  - public float floatValue()
  - … intValue
  - … longValue
  - …shortValue
  - compareTo()
  - toString()
  - equals

Clase Float

- Cada tipo numérico tiene su propia clase de objetos .
- float -> Float
- Provee métodos para convertir de float a String y viceversa.
- Métodos útiles para manejar enteros.
  - Float ()
  - public double doubleValue()
  - public float floatValue()
  - … intValue
  - … longValue
  - …shortValue
  - compareTo()
  - toString()
  - equals

Clase Double

- Cada tipo numérico tiene su propia clase de objetos .
- double -> Double
- Provee métodos para convertir de double a String y viceversa.
- Métodos útiles para manejar enteros.
  - Double ()
  - public double doubleValue()
  - public float floatValue()
  - … intValue
  - … longValue
  - …shortValue
  - compareTo()
  - toString()
  - equals

Clase Boolean

Method Summary	
boolean	booleanValue() Returns the value of this Boolean object as a boolean primitive.
int	compareTo(Boolean b) Compares this Boolean instance with another.
boolean	equals(Object obj) Returns true if and only if the argument is not null and is a Boolean object that represents the same boolean value as this object.
static boolean	getBoolean(String name) Returns true if and only if the system property named by the argument exists and is equal to the string "true".
int	hashCode() Returns a hash code for this Boolean object.
static boolean	parseBoolean(String s) Parses the string argument as a boolean.
String	toString() Returns a String object representing this Boolean's value.
static String	toString(boolean b) Returns a String object representing the specified boolean.
static Boolean	valueOf(boolean b) Returns a Boolean instance representing the specified boolean value.
static Boolean	valueOf(String s) Returns a Boolean with a value represented by the specified String.

Sesión

- Arreglos
- Colecciones: Collection, ArrayList y List, LinkedList , Map, HashSet, TreeSet, LinkedHashSet.

Arreglos (Ejemplos)

```
int primos[] ={2,3,5,7,11,13,17};

int [] primos = new int [100];
primos [0]=2;
primos [1]=3;

double [] data = new double[50];
for (int i=0; i<data.length();i++){
        data[i]=1.0;
}
```

Arreglos de dos dimensiones

float[][] temperatura = new float[10][365];

		temperatura	temperatura	temperatura	temperatura
temperatura	[0][0]	[1][0]	[2][0]	[3][0]	[.....][0]
temperatura	[0][1]	[1][1]	[2][1]	[3][1]	[.....][1]
temperatura	[0][3]	.....	.....	.....	.....
temperatura	[0][4]	.....	.....	.....	.....
temperatura	[0][.....]	.....	.....	.....	.....

Arreglos de objetos

- Al igual que podemos crear arreglos de datos primitivos o de la clase String, se pueden crear arreglos de objetos. Para esto es necesario crear primero el objeto tipo arreglo y luego asignar a cada posición del arreglo una instancia de la clase determinada.
- Para acceder a los métodos u atributos del objeto se realiza de la siguiente manera

  nombreArreglo[posición].nombreMetodo();

  nombreArreglo[posición].atributo

```
Ej
public static void main(String args[]){
      Pagos[ ] arrayPagos =  new Pagos[10];
      for (int i=0; i<10; i++){
         arrayPagos[i]= new Pagos();
      }
}
```

Ordenamiento de Arreglos

Para ordenar un arreglo de tipos de datos (Integer, Long,Double,etc) o String, se puede usar el método sort de la clase Arrays.
Ejemplo:

```
Long[] even1 ={10L,8L,6L,4L,2L};
Arrays.sort(even1);

Long[] even2 ={10L,2L,6L,8L,2L};
Arrays.sort(even2, Collections.reverseOrder());
```

Ordenamiento de Arreglos

Para ordenar un arreglo por sus atributos, esta es la forma mas eficiente

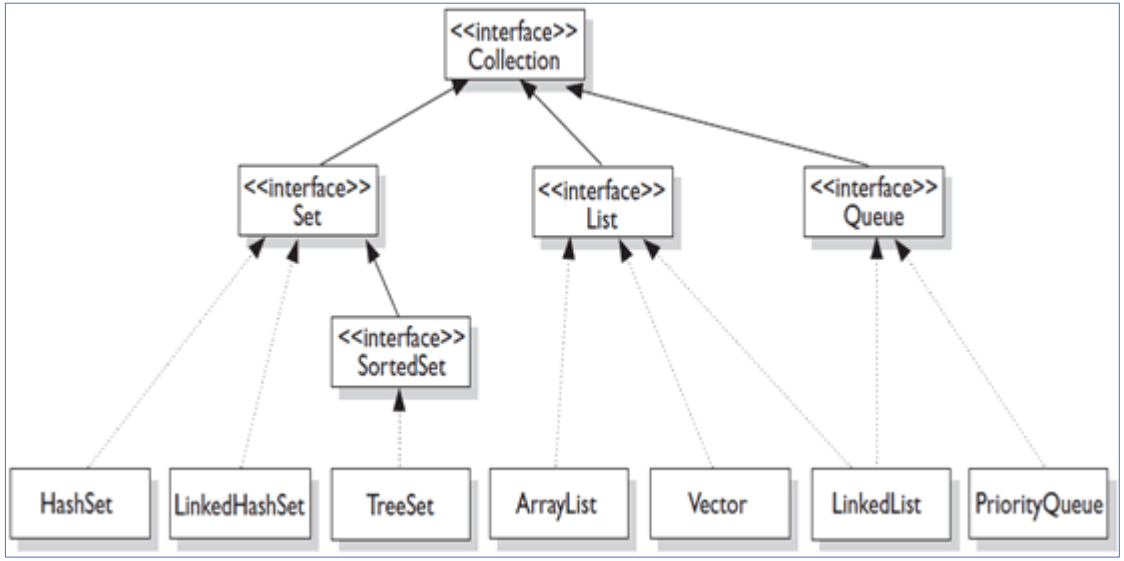
```
Arrays.sort(nombreArreglo, Comparator.comparing(NombreClase::Propiedad));
```

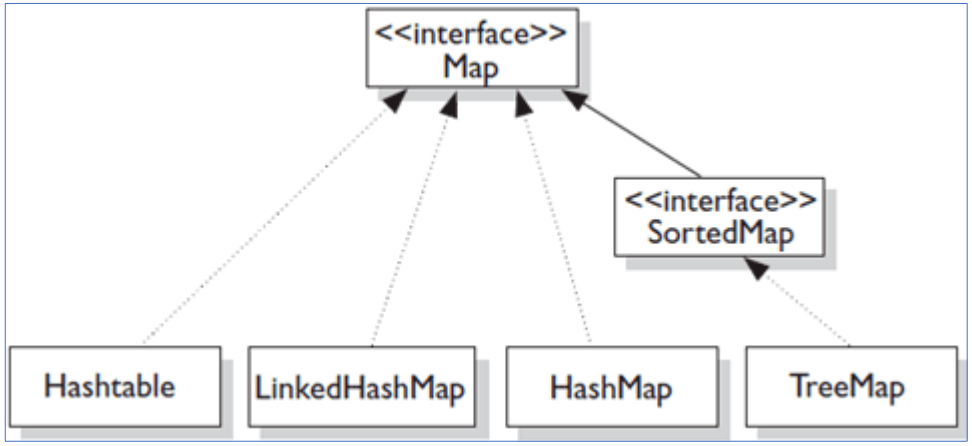
Ejemplo

```
Persona[] pp = new Persona[5];
Arrays.sort(pp, Comparator.comparing(Persona::getNombre));
```

Nota: import java.util.Comparator;

Árbol de colecciones





ArrayList, List y Vectores

La clase ArrayList es un arreglo de tamaño variable, La diferencia entre un arreglo y una ArrayList es que el tamaño de un arreglo no se puede modificar (si desea agregar o eliminar elementos a un arreglo, se debe crear uno nuevo). Mientras que los elementos se pueden agregar y eliminar de una ArrayList cuando lo desee.
Sintaxis:
ArrayList <TipoDato> nombreArray = new ArrayList<TipoDato>;

Metodos útiles de los ArrayList

- add: Para adicionar un objeto
- get: Para obtener un objeto
- set: cambiar un objeto por otro
- remove: Para eliminar un objeto
- iterator: para obtener un iterador y recorrer el ArrayList
- clear: Remover todos los objetos del ArrayList
- size: Nro. de elementos del ArrayList
- isEmpty: Devuelve falso o verdadero si el ArrayList esta vacio
- sort: Se utilizar para ordenar. Si el ArrayList, es de tipos de datos primitivos se utiliza Collections.sort(variableArrayList) , de lo contrario Collections.sort(Comparator.comparing(Clase::Propiedad));

- **Nota1**: La diferencia entre ArrayList y List, es que List es una interface y ArrayList es una clase
- **Nota2**: Los ArrayList y los Vectores, funcionan de forma similar, pero en memoria ocupa mas espacio un vector que un ArrayList, se recomienda mas el uso de ArrayList

LinkedList

La clase **LinkedList** es casi idéntica a **ArrayList**. La clase **LinkedList** es una colección que puede contener muchos objetos del mismo tipo, al igual que **ArrayList**. Sin embargo, aunque la clase ArrayList y la clase LinkedList se pueden usar de la misma manera, se construyen de manera muy diferente:

ArrayList
La clase ArrayList tiene una matriz regular dentro de ella. Cuando se agrega un elemento, se coloca en la matriz. Si la matriz no es lo suficientemente grande, se crea una nueva matriz más grande para reemplazar la anterior y se elimina la anterior.

LinkedList
Almacena sus elementos en "contenedores". La lista tiene un enlace al primer contenedor y cada contenedor tiene un enlace al siguiente contenedor de la lista. Para agregar un elemento a la lista, el elemento se coloca en un nuevo contenedor y ese contenedor se vincula a uno de los otros contenedores de la lista.

Cuándo usar
Utilice **ArrayList** para almacenar y acceder a datos, y **LinkedList** para manipular datos.

Métodos útiles de LinkedList
En muchos casos, ArrayList es más eficiente ya que es común necesitar acceso a elementos aleatorios en la lista, pero LinkedList proporciona varios métodos para realizar ciertas operaciones de manera más eficiente:

Método	**Descripción**
addFirst()	Agrega un elemento al principio de la lista.
addLast()	Agrega un elemento al final de la lista.
removeFirst()	Remueve el primer ítem de de la lista
removeLast()	Remueve el ultimo ítem de de la lista
getFirst()	Obtiene el primer Elemento de la lista
getLast()	Obtiene el ultimo elemento de la lista

HashSet y LinkedHashSet

Un HashSet es una colección de elementos donde cada elemento es único. El orden de almacenamiento de un HashSet es diferente al del ArrayList o LinkedList, por ende, cuando se muestra la información no necesariamente es el mismo orden en el cual se registró.

HashSet **no maneja algunos métodos** que si tienen ArrayList, LinkedList y Vector por la misma forma en que se almacena la información. Estos métodos son:

- sort
- get
- set

Un objeto LinkedHashSet funciona igual que un HashSet, la única diferencia es que los elementos del conjunto se encuentran en el orden que se insertan, similar a una lista pero sin dejar ingresar valores repetidos.

TreeSet

La clase Java TreeSet implementa la interfaz Set que usa un árbol para el almacenamiento. Hereda la clase AbstractSet e implementa la interfaz NavigableSet. Los objetos de la clase TreeSet se almacenan en orden ascendente.

Sintaxis: TreeSet<TipoDato> nombreTree = new TreeSet<TipoDato>();
Si el TipoDato es un String o un tipo de dato primitivo, el objeto siempre se muestra ordenado.

Si el tipo de Dato es un objeto, la clase del objeto debe implementar la interface **Comparable** y sobreescribir el método **compareTo obligatoriamente**

Ej:

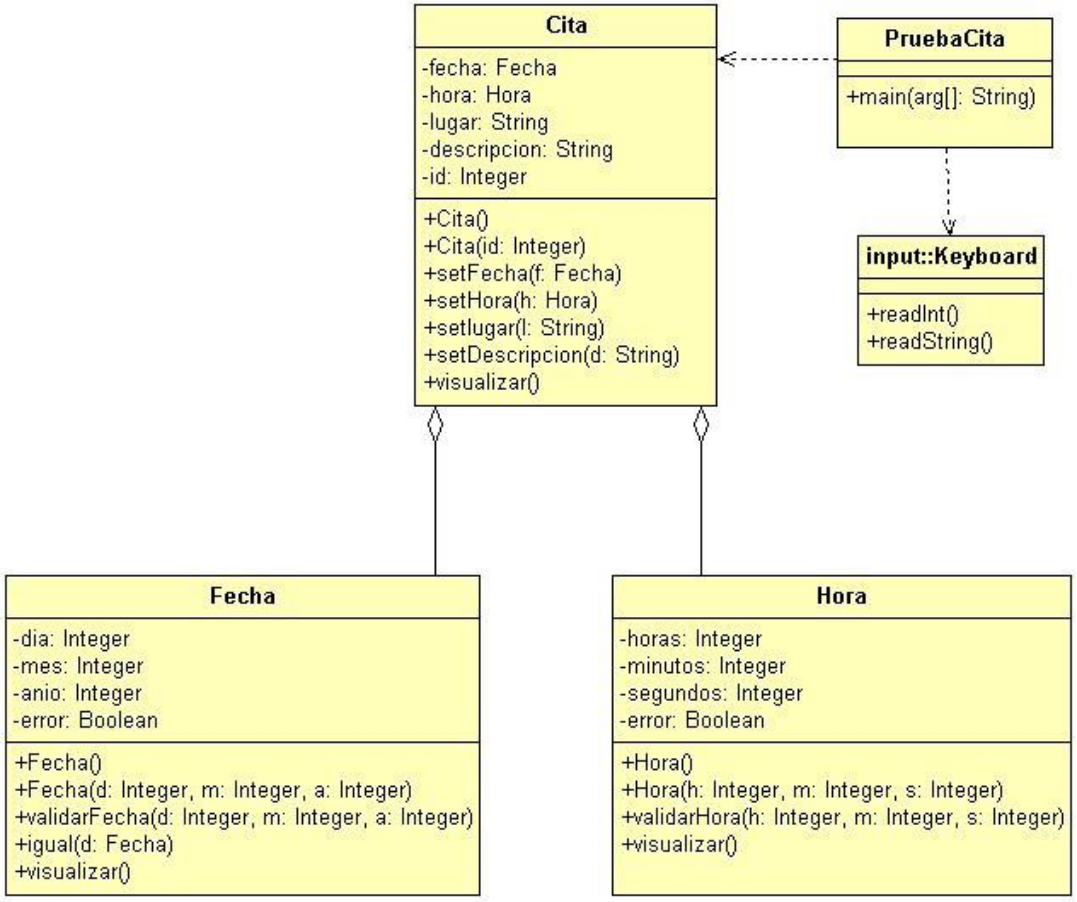
```
public class PersonaTreeSet  implements Comparable<PersonaTreeSet>{
    public int compareTo(PersonaTreeSet p) {
        int result=0;
        if(this.edad>p.getEdad()){
            result= 1;
        }else if(this.edad<p.getEdad()){
            result= -1;
        }else{
            result= 0;
        }
        return result;
    }
}
```

Sesión

Relación de Generalización

- Concepto de Herencia
- Beneficios de la Herencia
- Reconociendo la herencia
- Métodos final
- Superclases y Subclases
- Visibilidad protegida
- Reglas de Herencia
- Herencia con clases concretas
- Constructores en la herencia
- Uso de this y super

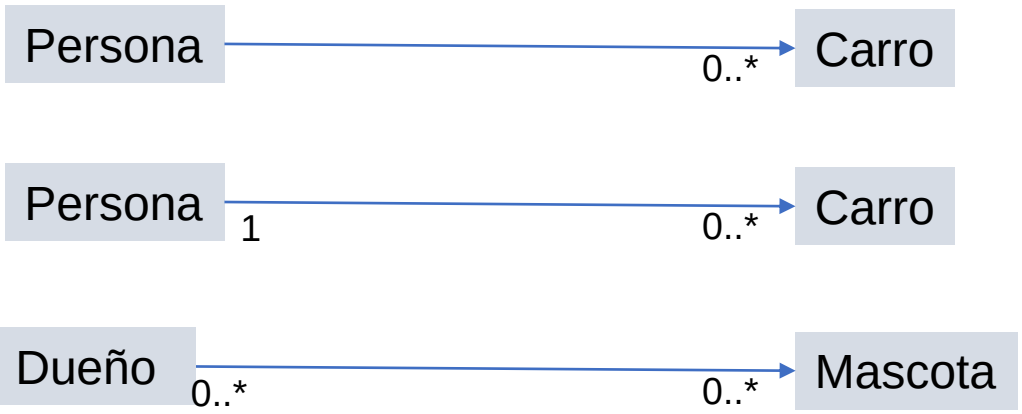
Diagrama de clases



Tipos de relaciones

1.Asociación

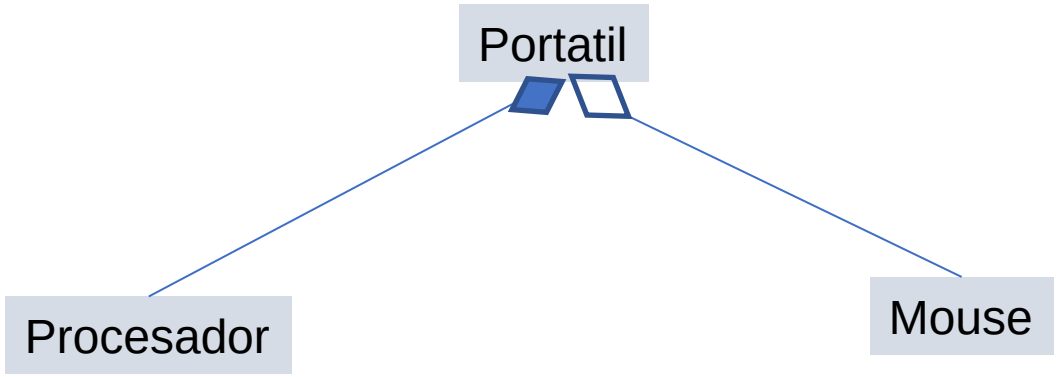
Indica como una clase se relaciona con otra. Una relación se dibuja con una línea solida entre dos clases. Por lo general son binarias y pueden ser unidireccionales o bidireccionales



1	Uno y solo uno
0..1	Cero o uno
0..*	Cero o mas
1..*	Al menos uno
N..M	Mínimo N, máximo M

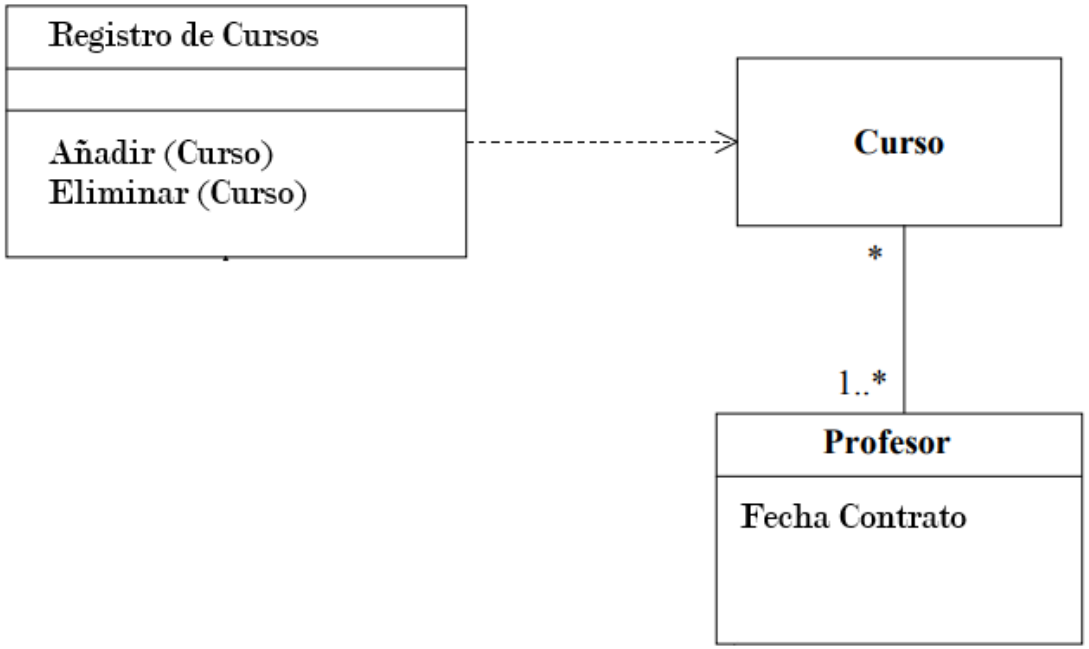
Tipos de relaciones

2. Agregaciones
Se utiliza para modelar una relación “todo o parte”, lo que significa que el objeto todo contiene al objeto parte
2.1 Agregación normal: Se dibuja con un rombo vacío del lado del todo. Esto indica que la clase “Todo” sigue funcionando a pesar de no tener alguna de las partes
2.2. Agregación de composición: Se dibuja con un rombo lleno del lado del todo e indica que el “todo” necesita a las “partes” para su funcionamiento



Tipos de relaciones

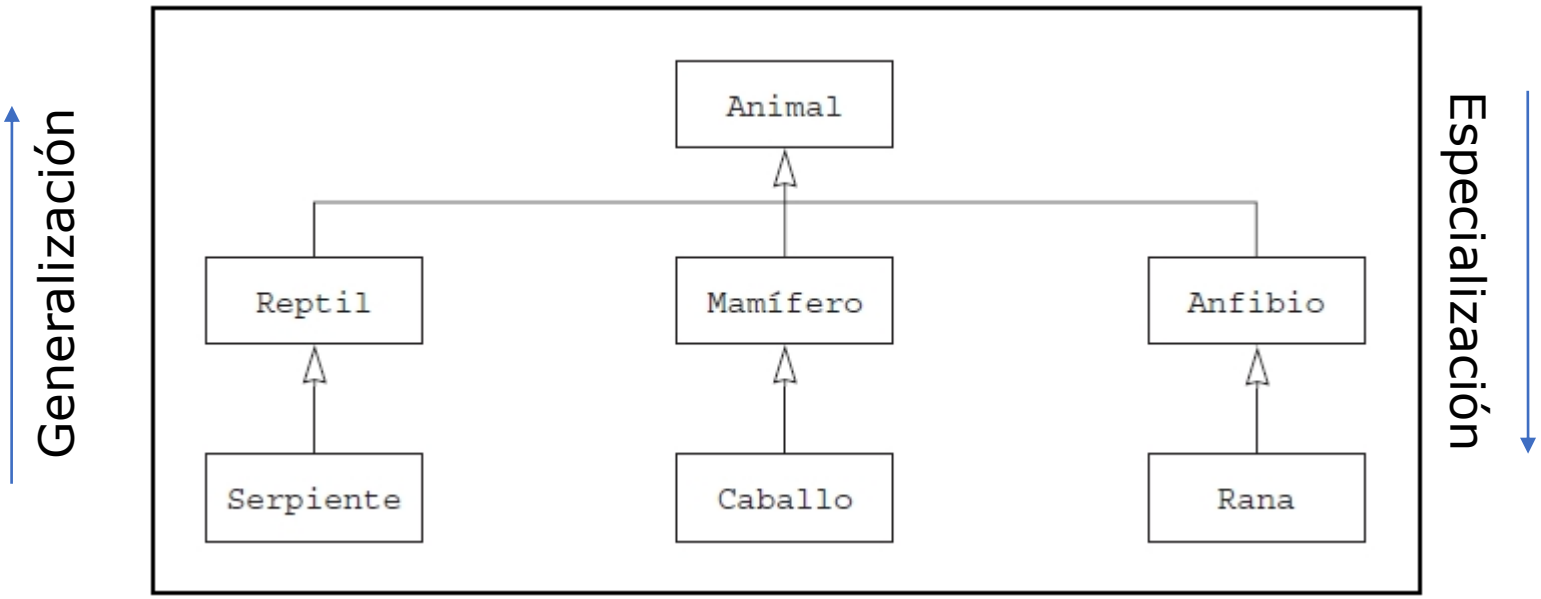
3. Dependencia: Es una relación en la cual una clase “depende” de otra para su funcionamiento, pero la clase de la cual se depende no hace parte estructural de la clase dependiente. La relación de dependencia se representa por una línea discontinua con una flecha en el lado del elemento independiente



Tipos de relaciones

4. generalización indica que una clase hereda los métodos y atributos definidos por una superclase.

5. especialización es la creación de subclases a partir de una superclase.
Se demarca con una línea continua y una flecha vacía del lado de la superclase



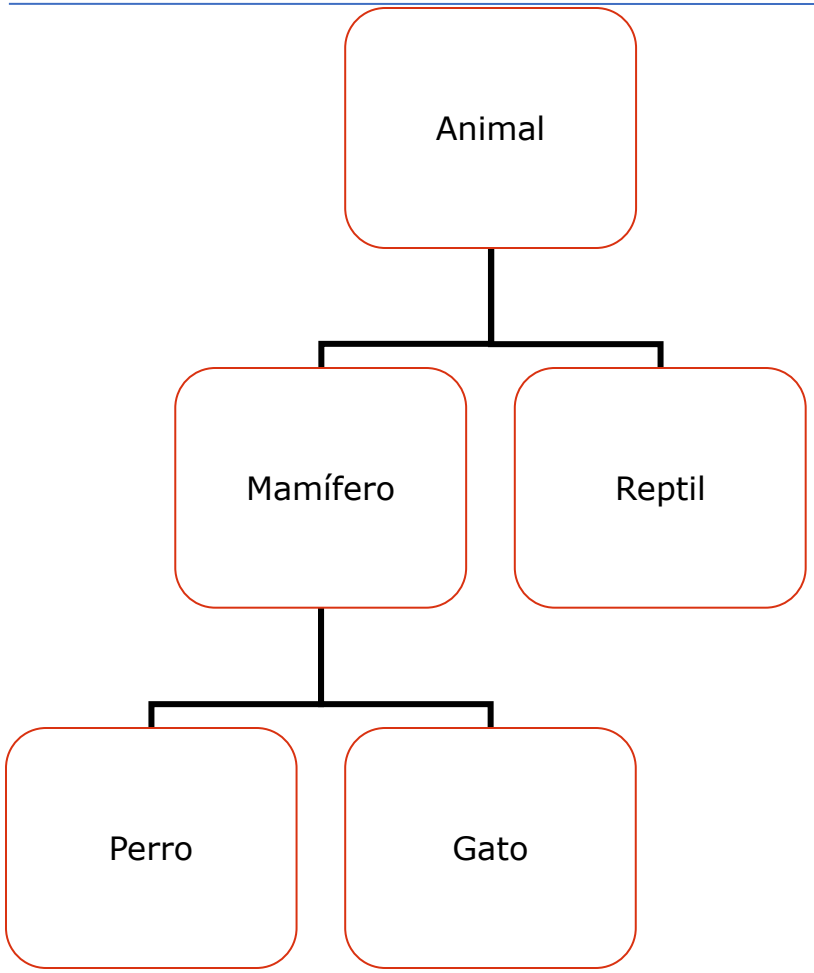
Resumen

- **Herencia**: Característica importante de la programación orientada a objetos es que permite crear nuevas clase a partir de clases ya definidas.
- **Polimorfismo**: Característica que permite que un objeto Java tome la forma de alguna de sus superclases.
- **Superclase**: Clase (Clase, Abstracta, Interface) que permite ser heredada por otras clases
- **Método abstracto**: Método que solo contiene la firma de método, mas no la implementación
- **Clase abstracta**: Clase que contiene uno o mas métodos abstractos, puede tener métodos con su respectiva implementación.
- **Interface**: Clase en la cual todos sus métodos son implícitamente abstractos y públicos.
- **final**: Aplica para indicar que una Clase que no puede ser heredada o un método no puede ser sobreescrito.

Herencia

- Característica importante de la programación orientada a objetos es que permite crear nuevas clase a partir de clases ya definidas.
- Es un mecanismo que permite derivar una clase de otra, de manera que extienda su funcionalidad.

Herencia



- El mamífero hereda los atributos y métodos del animal y además puede tener más atributos y métodos propios.
- El perro hereda los atributos y métodos del mamífero, es decir, los propios del mamífero más los que este heredó del animal

Herencia

- Se especifica agregando la cláusula **extends** después del nombre de la clase

```
public class Mamífero extends Animal{

}
```

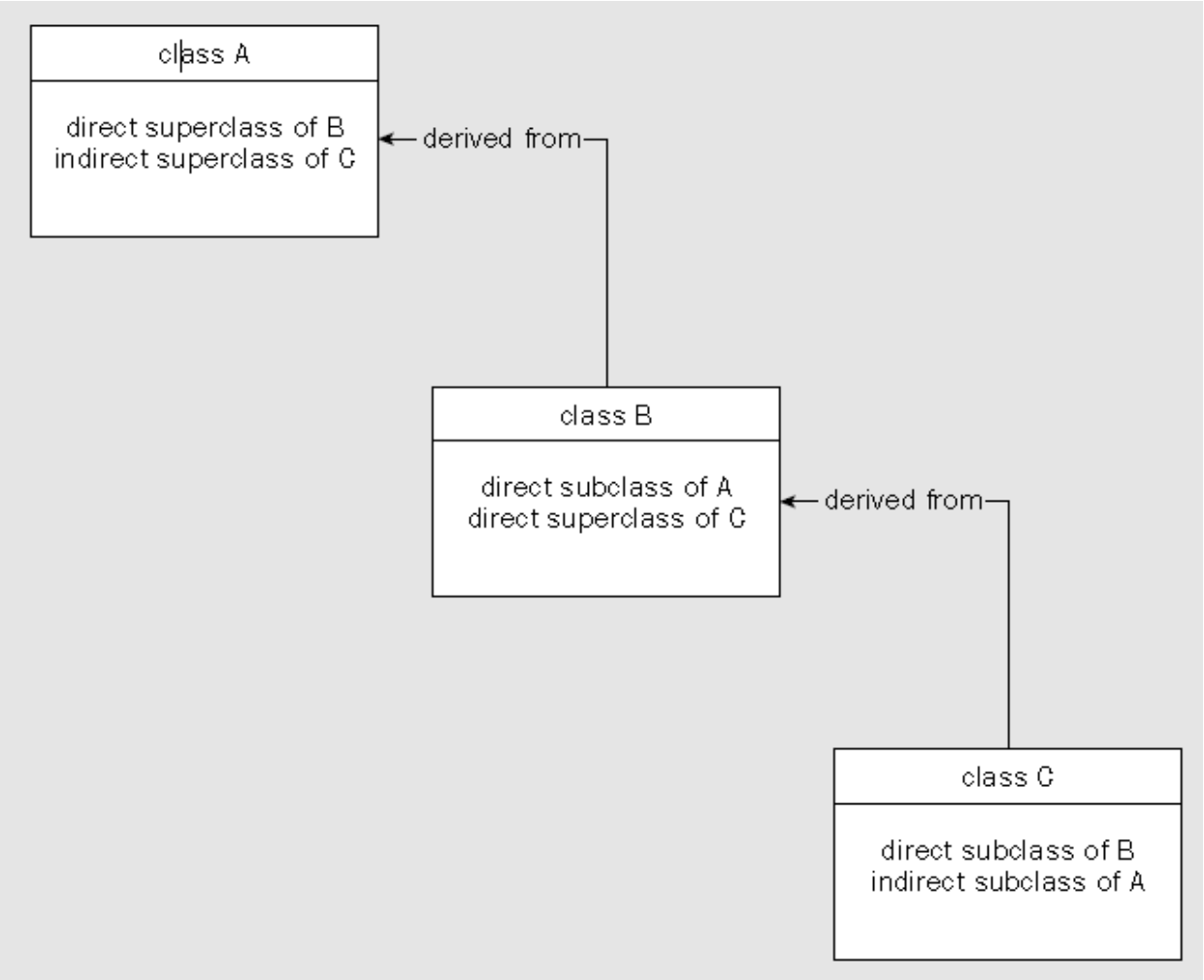
- Lo que quiere decir que el mamífero va a extender el comportamiento de la clase Animal.
- A través de la herencia podemos agregar nuevos campos y podemos agregar o sobre escribir métodos (override).

Herencia

- La clase base es llamada super clase o clase padre.
- La clase que hereda es llamada subclase, clase derivada, o clase hija.
  - Superclase Animal
  - Subclase Mamífero

- La clase de la que se va a heredar puede ser una clase construida por uno mismo, una clase estándar de Java o una clase desarrollado por alguien más.

Herencia


class A
direct superclass of B
indirect superclass of C
derived from
class B
direct subclass of A
direct superclass of C
derived from
class C
direct subclass of B
indirect subclass of A

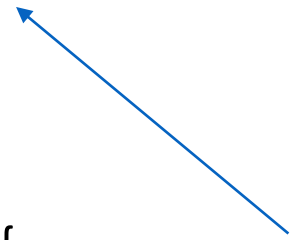
```
public class Animal {
  public Animal(String aType) {
    type = new String(aType);
  }

  public String toString() {
    return 'This is a ' + type;
  }

  private String type;
}
```

```
public class Dog extends Animal {
  // constructors for a Dog object

  private String name;                    // Name of a Dog
  private String breed;                   // Dog breed
}
```

Herencia

```
public class Animal {
   protected String raza;
   public Animal() {
   }

   protected void setRaza(String
     raza) {
      this.raza = raza;
   }

   protected String getRaza() {
      return raza;
   }
}
```

```
public class Cat extends Animal{

   public Cat(){
            super();
   }

   public static void main (String
     []args) {
      Cat c = new Cat();
      c.setRaza("x");
   System.out.println(c.raza);
   }

}
```



Qué imprime?

Herencia

```
public class Animal {
    protected String raza;
    public Animal() {
    }

    protected void setRaza(String raza) {
        this.raza = raza;
    }

    protected void imprimir (){
        System.out.println("soy la superclase");
    }

    protected String getRaza() {
        return raza;
    }
}
```

```
public class Cat extends Animal{
    private String raza="z";
     public Cat(){

     }

     public void imprimir (){
         System.out.println(this.raza);
         System.out.println(super.raza);
         System.out.println("soy el hijo");
     }
     public static void main (String []args) {
         Cat c = new Cat();
         c.setRaza("x");
         c.imprimir();
     }
}
```

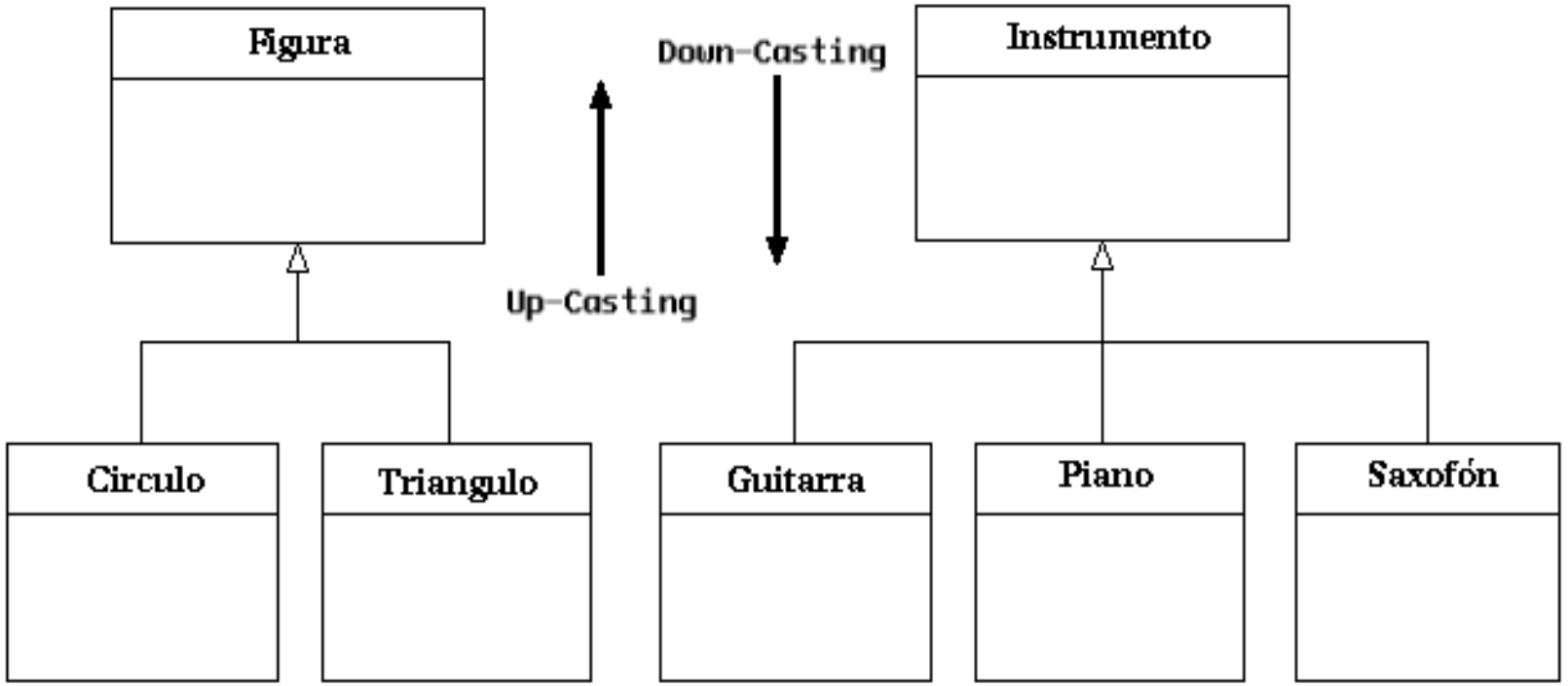
Qué imprime?

Sobreescritura y sobrecarga

- Animal
  - protected void imprimir (){
    System.out.println("soy la superclase");
    }
- Cat
  - public void imprimir (){
    System.out.println(this.raza);
    System.out.println(super.raza);
    System.out.println("soy la subclase");
    }
  - public void imprimir (String z){
    System.out.println("soy el hijo "+z);
    }

Polimorfismo

- Un mismo Objeto puede tomar diversas formas
- Superclase varClase = new Subclase();
- Figura a = new Circulo();
- Figura b = new Triangulo();
- Manipular un Objeto como si éste fuera de un tipo genérico.



Polimorfismo

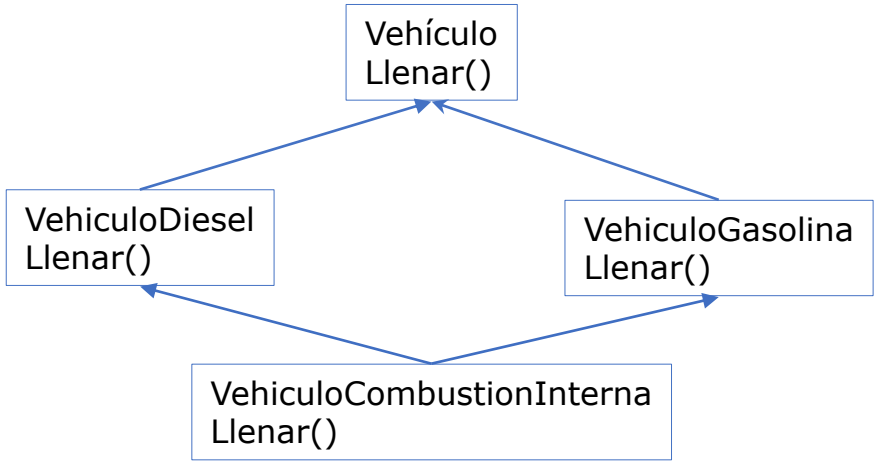
- Figura a = new Circulo();
- Figura b = new Triangulo();
- a y b sólo pueden acceder a las partes del objeto que pertenecen a Figura.
- Al ser a y b de la clase Figura, se puede realizar la operación a=b;
- if(a instanceof Circulo){

  …

  }

Herencia Múltiple

- Java no permite la herencia múltiple por se podrían generan ambigüedades (comúnmente llamado problema del diamante), si dos subclases de la misma clase (posteriormente superclases de otra clase) implementan el mismo método.




```
public static void main (String args[]){
VehiculoCombustionInterna vci = new VehiculoCombustionInterna();
vci.llenar (); //A cual de los métodos llenar se invoca???
}
```

Clases abstractas

- Una clase abstracta es en la que uno o más métodos están declarados pero no definidos. Estos se convierten en métodos abstractos.
- Se adiciona la palabra reservada **abstract.**

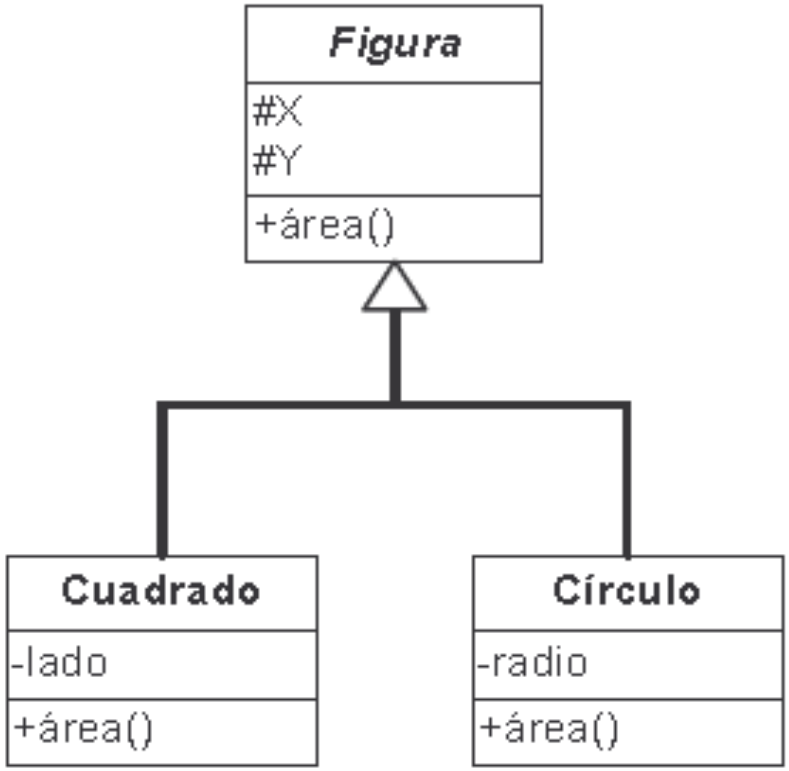
```
public abstract class Animal {
    private String type;
    public abstract void sound(); // Abstract method
    public Animal(String aType) {
      type = new String(aType);
    }
    public String toString() {
      return “This is a “ + type;
    }
}
```

  - public class Cat extends Animal

Clases abstractas

- Si un método es declarado como abstracto, la clase debe ser abstracta.
- Una clase abstractas puede tener sólo algunos métodos abstractos.
- Una clase abstracta no puede ser instanciada (no se puede hacer new de la clase).

Clases abstractas



Sesión

- Herencia Avanzada
  - Interfaces
  - Herencia múltiple
  - Anidamiento de clases

Interfaces

- Define los métodos que la subclase debe soportar, pero no el cómo.
  - La clase Animal define que tiene un método llamado comer(), pero no provee la lógica de éste. Cada animal en particular implementa el comportamiento del método comer.
    - Canario, serpiente, zancudo… Todos comen pero de forma diferente.
- Es una clase completamente abstracta.
- La subclase se debe encargar de implementar todos los métodos de la interface.
  - La subclase no extiende el comportamiento de la clase padre, sino que lo implementa.

Interfaces

- A la definición de la clase se le añade la palabra reservada **interface.**
- public **interface** Animal…
- public class Cat **implements** Animal…

- public abstract **interface** Rollable { } //válido pero redundante
- public **interface** Rollable { }
- Todos los métodos de una interfaz son implícitamente públicos y abstractos, no es necesario poner el modificador.

Interfaces

- Las interfaces no encapsulan datos, sólo definen los métodos.
- Las interfaces no tienen método constructor.
- Las interfaces no puede ser instanciadas.
- Las interfaces pueden extender otras interfaces. Únicamente interfaces.
- Las interfaces no pueden implementar otras clases.

Herencia Multiple - Interfaces

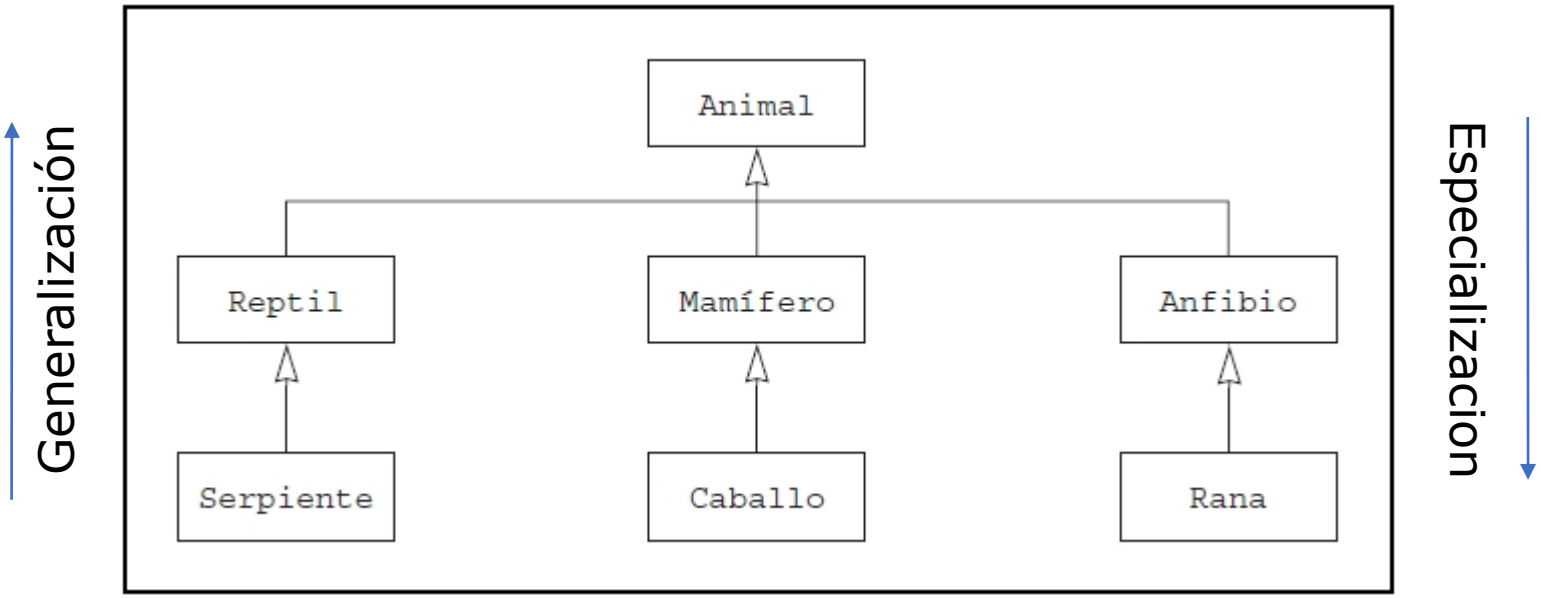
- Una clase puede extender a otra y a la vez **implementar** a una o mas clases. Esto permite tener subclases mas completas y con mayores funcionalidades.
- Se debe recordar que una clase solo puede extender una superclase, pero puede implementar mas de una interface
- Una clase que extiende e implementa a la vez se define así:

```
public class NombreClase extends Clase1 implements Clase2, Clase3 {……}
```

Generalización / Especialización

La **generalización** indica que una clase hereda los métodos y atributos definidos por una superclase.

La **especialización** es la creación de subclases a partir de una superclase



Anidamiento de clases

En java es posible crear clases dentro de otras clases.
Las clases anidadas ayudan a agrupar lógicamente clases que solo se utilizan en sitio especifico y por lo tanto aumenta el uso de la **encapsulación**

```
public class Anidamiento1 {

   public  class A1{
      public void metodo1(){
     }
   }
   private  class A2{
      public void metodo2(){
      }
   }

   public void llamar(){
      A1 a1 =  new A1();
      a1.metodo1();
      A2 a2 = new A2();
      a2.metodo2();


   }

}
```

```
class A1{
      public void metodo1(){
     }
   }
class A2{
      public void metodo2(){
      }
   }

 public class Anidamiento1 {

   public void llamar(){
      A1 a1 =  new A1();
      a1.metodo1();
      A2 a2 = new A2();
      a2.metodo2();


   }

}
```