Icono *Training Consulting*

Formación

Curso Iniciación Programación Java

2025

Formador

Ana Isabel Vegas

INGENIERA INFORMÁTICA con Master Máster Universitario en Gestión y Análisis de Grandes Volúmenes de Datos: Big Data, tiene la certificación PCEP en Lenguaje de Programación Python y la certificación JSE en Javascript. Además de las certificaciones SCJP Sun Certified Programmer for the Java 2 Platform Standard Edition, SCWD Sun Certified Web Component Developer for J2EE 5, SCBCD Sun Certified Business Component Developer for J2EE 5, SCEA Sun Certified Enterprise Architect for J2EE 5.

Desarrolladora de Aplicaciones FULLSTACK, se dedica desde hace + de 20 años a la CONSULTORÍA y FORMACIÓN en tecnologías del área de DESARROLLO y PROGRAMACIÓN.

training@iconotc.com

Iniciación Programación Java

❑ **Duración:** 25 horas

❑ **Modalidad:** On-line

❑ **Fechas/Horario:**

- Días 17, 18, 19, 20 y 21 Noviembre 2025
- Horario 9:30 – 14:30 hs.

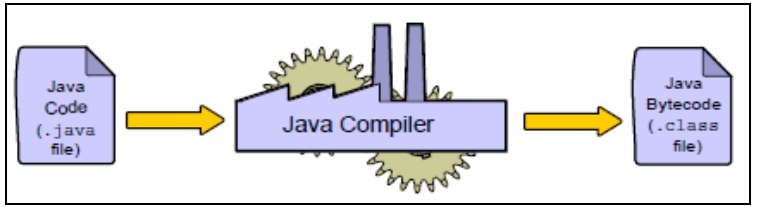
❑ **Contenidos:**

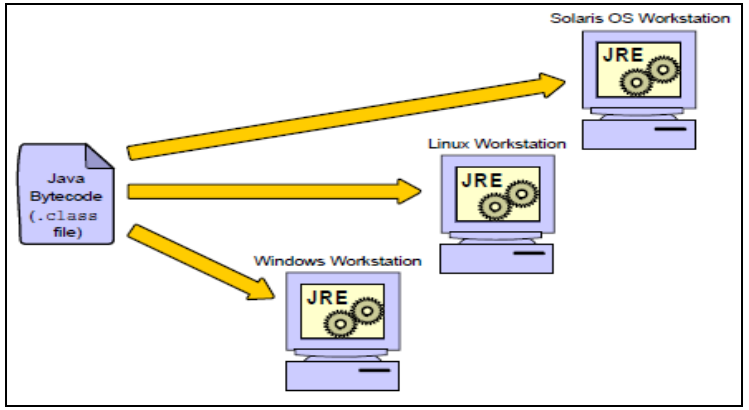
- Ciclo de vida de un programa
- Definición de clases, atributos y métodos
- Operadores y sentencias
- Definición y uso de paquetes
- Implementación de jerarquías de herencia
- Métodos constructores
- Definición e implementación de interfaces
- Redefinición polimórfica de métodos
- Gestión de errores mediante excepciones
- Anotaciones
- Genéricos e inferencia de tipos
- Colecciones de objetos
- Nociones de programación funcional

Introducción

Lenguaje Java

- El lenguaje de programación Java es un lenguaje orientado a objetos.
- Lenguaje independiente de la plataforma





JRE y JDK

- JRE (Java Runtime Environment); Sería necesario únicamente para ejecutar una aplicación. Esta sería la versión que se deben descargar los clientes, usuarios finales de nuestra aplicación. Incluye únicamente la JVM y un conjunto de librerías para poder ejecutar.
- JDK (Java Development Kit); Estos son los recursos que necesitamos los desarrolladores ya que incluye lo siguiente:
  - JRE
  - Compilador de java
  - Documentación del API (todas las librerías de Java)
  - Otras utilidades por ejemplo para generar archivos .jar, crear documentación, efectuar un debug.
  - También incluye ejemplos de programas.

Ediciones JAVA

- JSE (Java Standard Edition); Es la edición más básica de Java pero no menos importante. Recoge los fundamentos básicos del lenguaje pero solo nos permite desarrollar aplicaciones locales, aplicaciones escritorio y applets (aplicaciones que se ejecutan en el navegador del cliente).

- JEE (Java Enterprise Edition); Esta edición es la más completa de todas. Gracias a ella podremos desarrollar aplicaciones empresariales tales como aplicaciones web, aplicaciones eCommerce, aplicaciones eBusiness, ...etc.

- JME (Java Micro Edition); Con esta edición podremos desarrollar aplicaciones para micro dispositivos tales como teléfonos móviles, PDAs, sistemas de navegación, ...etc.

Sintaxis Java

Variables

- Una variable sirve para identificar mediante un nombre a un espacio de memoria en el que se sitúa un dato antes de ser utilizado por el procesador.

- En Java hay un conjunto de variables convencionales para gestionar datos numéricos, booleanos o de tipo carácter, a las que se denominan variables primitivas, y un conjunto especial de variables que identifican objetos, que se conocen como variables de referencia.

Tipos de variables

- **Numéricas**:
  - Enteros
    - byte; –128 a 127 (byte)
    - short; -32768 a 32767
    - int; -231 a 231-1 (int)
    - long; -263 a 263-1 L
  - Reales
    - float; +-3,4 x 1038 a +-1,4 x 10-45 F(float)
    - double; +-1,8 x 10308 a +-4,9 x 10-324

Tipos de variables

- **Booleanas**:
  - boolean; true o false (sin comillas)

- **Carácter**:
  - char; un solo carácter. (el dato se introduce entre comillas simples). Se usa también para caracteres de control.
  - String; un conjunto de caracteres. (la “S” debe ser mayúscula), (el dato se introduce entre comillas dobles)

Caracteres de control

‘\n’	Línea siguiente
‘\t’	Tabulador
‘\b’	Retroceso
‘\a’	Alarma (beep)
‘\r’	Retorno de carro
‘\’’	Comillas simples
‘\\’	Contrabarra (Backslash)
‘\”’	Dobles comillas
‘\xxx’	Carácter en octal
‘\0’	Carácter nulo (null)
‘\uxxxx’	Carácter en hexadecimal Unicode

Nombre de variables

- Para otorgar nombre a una variable hay que seguir una serie de normas:
  - El primer carácter del nombre debe ser una letra mayúscula o minúscula, “_” o “$”.
  - No pueden utilizarse como nombres de variables las palabras reservadas de JAVA.
  - Los nombres deben ser continuos, es decir, sin espacios en blanco.
  - Los identificadores de variables son sensibles a las mayúsculas y a las minúsculas.

Palabras reservadas

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

Crear una variable

- Declarar variables primitivas:

  tipo_de_primitiva nombre_variable;

  ejemplo: int numero;

- Declarar variables de referencia a objeto:

  Nombre_de_clase nombre_objeto;

  ejemplo: String nombre;

Crear una variable

- Podemos declarar más de una variable al mismo tiempo, siempre y cuando sean del mismo tipo

```
int num1, num2, num3, num4;
```

- También se puede declarar e inicializar más de una variable a la vez.

```
int num1 = 3, num2 = 5, num3 = 7, num4 = 8;
```

Comentarios

- Varias líneas: /* */
  - Ejemplo:

```
/* Este es un ejemplo
de un comentario
que ocupa varias líneas */
```

- Una sola línea: //
  - Ejemplo: // Este es de una sola línea

Operadores

- Aritméticos
- Relaciónales
- Lógicos
- Asignación
- Prioridad

Operadores Aritméticos

*	(multiplicación)
/	(división)
%	(resto de la división)
+	(suma)
-	(resta)
++	(incremento)
--	(decremento)
-	(cambio de signo)

Operadores Relacionales

>	(mayor que)
>=	(mayor o igual que)
<	(menor que)
<=	(menor o igual que)
==	(igual que)
!=	(distinto de)

Operadores Lógicos

&&	(AND)
||	(OR)
!	(negación)

Operadores de Asignación

=	(asignación normal)
+=	(con incremento)
-=	(con decremento)
/=	(con división)
%=	(con resto)

Prioridad de los operadores

Expresión:	() []
Unitarios:	- ! ++ --
Multiplicativos:	* / %
Aditivos:	+ -
Relacionales:	< <= > >= == !=
AND lógico:	&&
OR lógico:	||
Condicionales:	?:
Asignación:	= *= /= %= += -=

Fundamentos String

- Un objeto de la clase String en una cadena de caracteres inmutable, no se pueden modificar
- Los métodos que operan con String devuelven una copia de la cadena modificada
- Se pueden crear:

```
String n1=new String("mi cadena");
```

- O también:

```
String n1="mi cadena";
```

En este caso puede reutilizar cadenas de un pool

Métodos String

- int length(). Devuelve la longitud de la cadena
- String toLowerCase(), toUpperCase(). Devuelven la cadena convertida a minúsculas y mayúsculas, respectivamente

```
String n1="cadena";
System.out.println(n1.toUpperCase()); //muestra: CADENA
System.out.println(n1); //muestra: cadena, no ha cambiado
```

- String substring(int a, int b). Devuelve un trozo de cadena comprendido entre las posiciones a y b-1

```
String n1="esto es un texto";
System.out.println(n1.substring(3,9)); //muestra: o es u
```

Métodos String

- char charAt(int pos). Devuelve el carácter que ocupa la posición indicada

```
String n1="esto es un texto";
System.out.println(n1.charAt(0)); //muestra: e
System.out.println(n1.charAt(20));  //StringIndexOutOfBoundsException
```

- int indexOf(String cad). Devuelve la posición de la cadena parámetro. Si no existe, devuelve -1

```
String n1="esto es un texto";
System.out.println(n1.indexOf("un")); //muestra: 8
```

- String replace(CharSequence c1, CharSequence c2). Devuelve la cadena resultante de reemplazar la subcadena c1 por c2.

```
String n1="esto es un texto";
System.out.println(n1.replace("es","de")); //muestra: deto de un texto
```

Métodos String

- boolean startsWith(String s), endsWith(String s). Indica si la cadena empieza o termina, respectivamente, por el texto recibido:

```
String n1="esto es un texto";
System.out.println(n1.endsWith("to")); //muestra: true
System.out.println(n1.startsWith("eso")); //muestra: false
```

- String trim(). Devuelve la cadena resultante de eliminar espacios al principio y al final de la misma

```
String n1="  cade prueba nueva  ";
System.out.println(n1.trim().length()); //muestra: 17
```

- String concat(String s). Mismo efecto que aplicar el operador +

- boolean isEmpty(). Devuelve true si es una cadena vacía. Equivale:

```
cad.equals("")
```

if ... else

- Permite tomar decisiones en función de unas determinadas condiciones que se impongan para ejecutar uno u otro código según lo que indiquen dichas condiciones. Su sintaxis es la siguiente:

```
if (condición) {

        código si cierto;

}

else {

        código si falso;

}
```

- Comprueba la validez o falsedad de una condición. Si esta condición es verdadera se ejecuta el código contenido entre las primeras llaves. Si es falsa, se interpreta el código tras las segundas llaves.

if ... else

- La cláusula else no es obligatoria. Si la condición fuese falsa no se llevaría a cabo ninguna acción. Además si no ponemos else y el código de las primeras llaves es tan solo de una línea podemos utilizar este código:

```
if (condición) sentencia;
```

if ... else abreviado

- Con los símbolos ? y : podemos construir un condicional de forma abreviada de la siguiente manera:

condición ? sentencia_v : sentencia_f;

ejemplo: a = (c < d) ? c*2 : 27 ;

es lo mismo que:

```
if (c<d)
{
        a=c*2;
}
else
{
        a=27;
}
```

if ... else anidados

```
if (condición1) {
        código si cierto;
}
else if (condición2) {
        código si cierto;
}
else if (condición3) {
        código si cierto;
}
else {
        código si falso;
}
```

switch ... case

- Durante la ejecución de un programa puede darse el caso de necesitar tomar una decisión a partir de múltiples posibilidades, pero donde sólo una de ellas va a ser posible en cada ocasión. Su sintaxis es:

```
switch (expresión) {
        case valor1_expresion:
                codigo1;
                break;
        case valor2 _expresion :
                codigo 2;
                break;
        default:
                codigo por omisión;
}
```

La expresión debe ser de tipo **char, byte, short, int o String.**

Bucle for

- Un bucle es aquella operación o conjunto de operaciones que se repite cierto número de veces hasta llegar a un estado que le obliga a detenerse, permitiendo que continúe la ejecución normal hacia delante del código. A las operaciones que se repiten en cada vuelta se las denomina conjuntamente Cuerpo del bucle.

- Cuando se sabe de antemano el número de veces que se va a repetir el cuerpo del bucle antes de detenerse, se dice que el bucle es determinado. Si no se pude saber de antemano cuándo se detendrá, se dice que es un bucle indeterminado.

Bucle for

- La sintaxis es la siguiente:

```
for (contador=valor_inicial; condición del bucle; incremento) {
        cuerpo del bucle;
}
```

- La condición de bucle es una expresión que indica si el cuerpo del bucle se debe ejecutar una vez más o no. Mientras esta condición sea verdadera se repetirá el bucle. En el momento en que sea false se detendrá. Si inicialmente ya es falsa, el cuerpo del bucle no se ejecutará nunca.

- Los bucles for se pueden anidar.

Bucle while

- Itera mientras la condición de final se cumpla. Comprueba la condición al principio. Si es falsa no se ejecuta

```
while (condición) {
        cuerpo del bucle;
}
```

Bucle do ... while

- Itera mientras la condición de final se cumpla. Comprueba después, al menos se ejecuta una vez.

```
do {
        cuerpo del bucle;
} while (condición);
```

Break y Continue

- Continue:
  - Hace que se termine la iteración actual del bucle y que se pase a la siguiente. Esta sentencia sólo se puede utilizar dentro de un bucle si no es así genera error de compilación.

- Break:
  - Interrumpe del bucle definitivamente. Se puede utilizar en un bucle o en un switch.

Arrays

Arrays o Matrices

- Un array es un conjunto de elementos colocados de forma adyacente en la memoria de manera que nos podemos referir a ellos con un solo nombre común mientras que, por otro lado, no se pierde la independencia de los mismos.

- Es un modo de agrupar datos para que se guarden ordenadamente y sean más cómodos de manejar y gestionar. Todo elemento pertenece a un array lleva asociado un índice de forma unívoca: siempre se puede acceder a un elemento determinado en cualquier array si se conoce su índice.

Declaración de arrays

- En Java todos los arrays deben ser declarados y construidos antes de ser usados.

```
int vector[];                  //una dimensión
int [] vector;
String alumnos[][];            //dos dimensiones
String [][] alumnos;
```

- Cuando se define un array de referencias a objetos, es decir, de objetos, estos toman por defecto null hasta que se le asigne una referencia a un objeto existente. En el caso de variables primitivas toman el valor cero si son numéricas y false, si son de tipo boolean.

Definición de arrays

- Un array una vez declarado debe ser definido, proceso que también se puede identificar como construcción, en el cual se reserva el espacio necesario para almacenar los datos o los objetos que se va a contener.

```
vector = new int[30];

nombres = new String[var_nombres];

alumnos = new String [3][2];
```

Definición de arrays

- Un array de dos índices, puede tener en cada fila componentes de distinta longitud, de tal forma que se pueden construir, por ejemplo, matrices triangulares:

```
int matriz [][];
matriz = new int [3][];
        matriz[0] = new int [2];
        matriz[1] = new int [4];
        matriz[2] = new int [5];
```

Declaración y definición

- Es posible declarar y definir el array en una misma instrucción.

```
int vector[] = new int[30];
boolean resultados[]=new boolean[10];
String nombres[]=new String[var_nombres];
String alumnos[][]=new String[3][2];
```

Declaración, definición e inicialización

- Es posible hacer las tres cosas en una misma sentencia:

```
int vector[] = {2,3,4,5,6};
String nombres = {“Juan”,”Lola”,”Luis”,”Ana”};
String alumnos[][]={{“alumn00”,”alumn01”,”alumn02”},
                    {“alumn10”,”alumn11”,”alumn12”}};
```

Longitud de un array

- Para investigar la longitud de un array utilizamos el atributo length del array.

- Con dicho atributo obtenemos el máximo de elementos posibles y no el número de elementos ocupados.

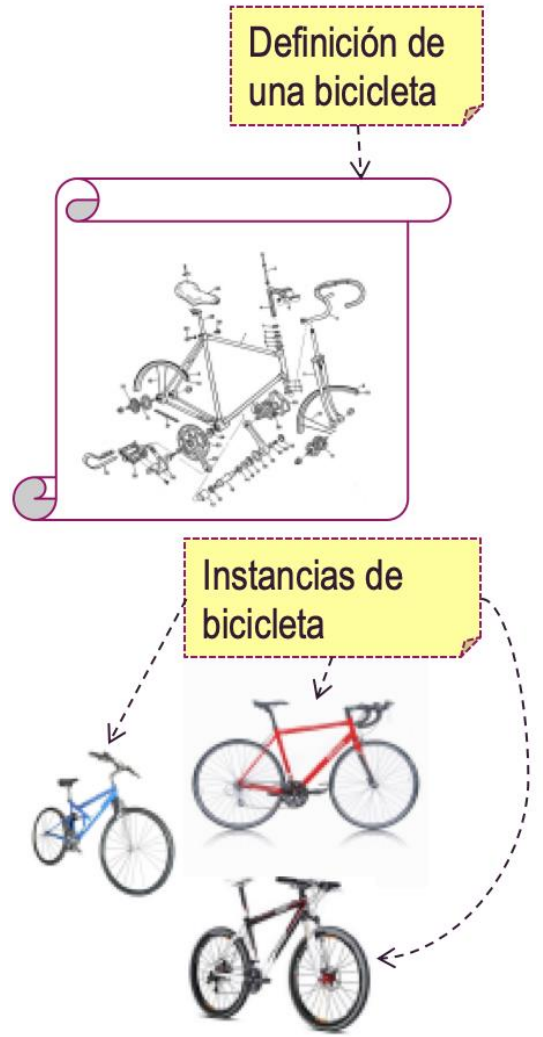
```
int nFilas = matriz.length;
int nColumnas1 = matriz[0].length;
```

Clases y Objetos

Clases y objetos

- Clase:
  - Una clase es la definición de las características concretas de una determinada tipología de objetos. Es decir, cuáles son los atributos y métodos de los que van a disponer todos los objetos de ese tipo. Equivale a un tipo de dato.

- Objeto:
  - Un objeto es una agregado de atributos (información) y métodos (comportamiento), creado según la definición de una clase.
  - A un objeto concreto que pertenece a una clase se le suele llamar instancia.



Declaración de clases

- El archivo en el cual declaramos la clase se debe seguir este orden.
  - Declaración del paquete; Un paquete cumple una función similar a una carpeta en Windows. El paquete nos sirve para organizar nuestros recursos y además poder implementar niveles de acceso como veremos más adelante.
  - Importaciones; En una clase podremos utilizar otras clases ya creadas. Estas pueden ser del propio API o también clases desarrolladas por terceras personas. Para poder acceder a otras clases es necesario importarlas previamente.
  - Declaración de la clase; Aquí se definirán los recursos de la clase.

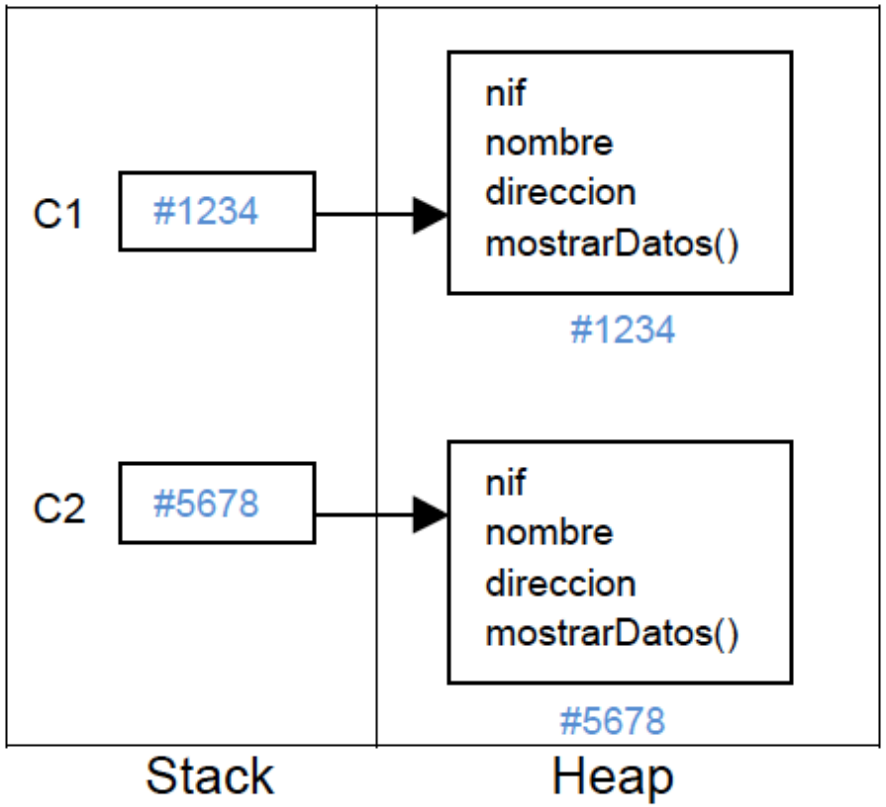
```java
// Declaracion del paquete
package ejemplo1clasesyobjetos;

// importaciones si fuesen necesarias

// Declaracion de la clase
public class Cliente {

    // propiedades
    public String nif;
    public String nombre;
    public String direccion;

    // constructores
    public Cliente() {
    }

    // metodos
    public String mostrarDatos() {
        return "nif=" + nif + " nombre=" + nombre + " direccion=" + direccion;
    }
}
```

Creación de objetos

- A la hora de crear un objeto utilizamos la palabra new seguida del constructor.

Ejemplo: new Cliente();

```
Cliente c1 = new Cliente();
Cliente c2 = new Cliente();
```



Programación Orientada a Objetos

Características de la P.O.O

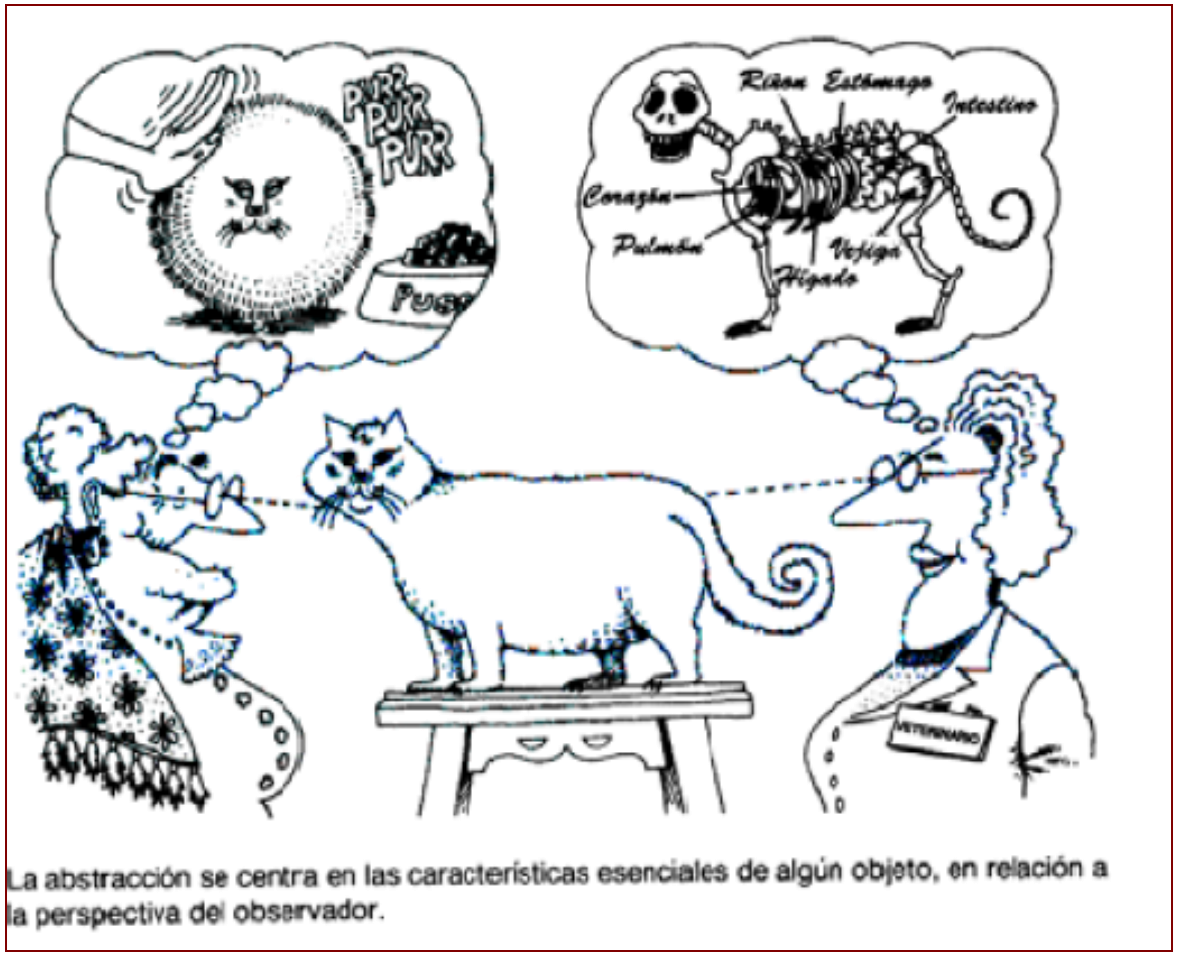
- Abstracción
- Encapsulamiento
- Herencia
- Polimorfismo

Abstracción

- Es la acción de separar las características esenciales de algo sin incluir detalles irrelevantes.

- Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

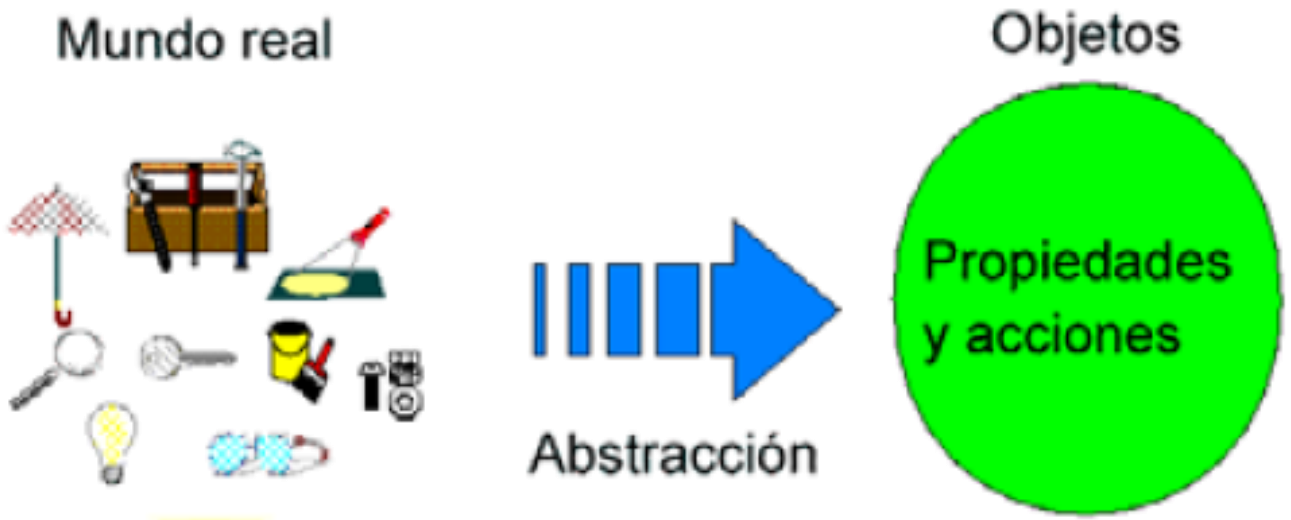
Abstracción



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Abstracción

- Nos fijaremos en el comportamiento de los objetos para definir las acciones u operaciones que son capaces de realizar.
- Ejemplo de objetos: una factura ,un usuario, un albarán…
- A partir de objetos que tienen unas propiedades y acciones comunes se deben abstraer las clases. En las clases se definen las propiedades y operaciones comunes de los objetos.



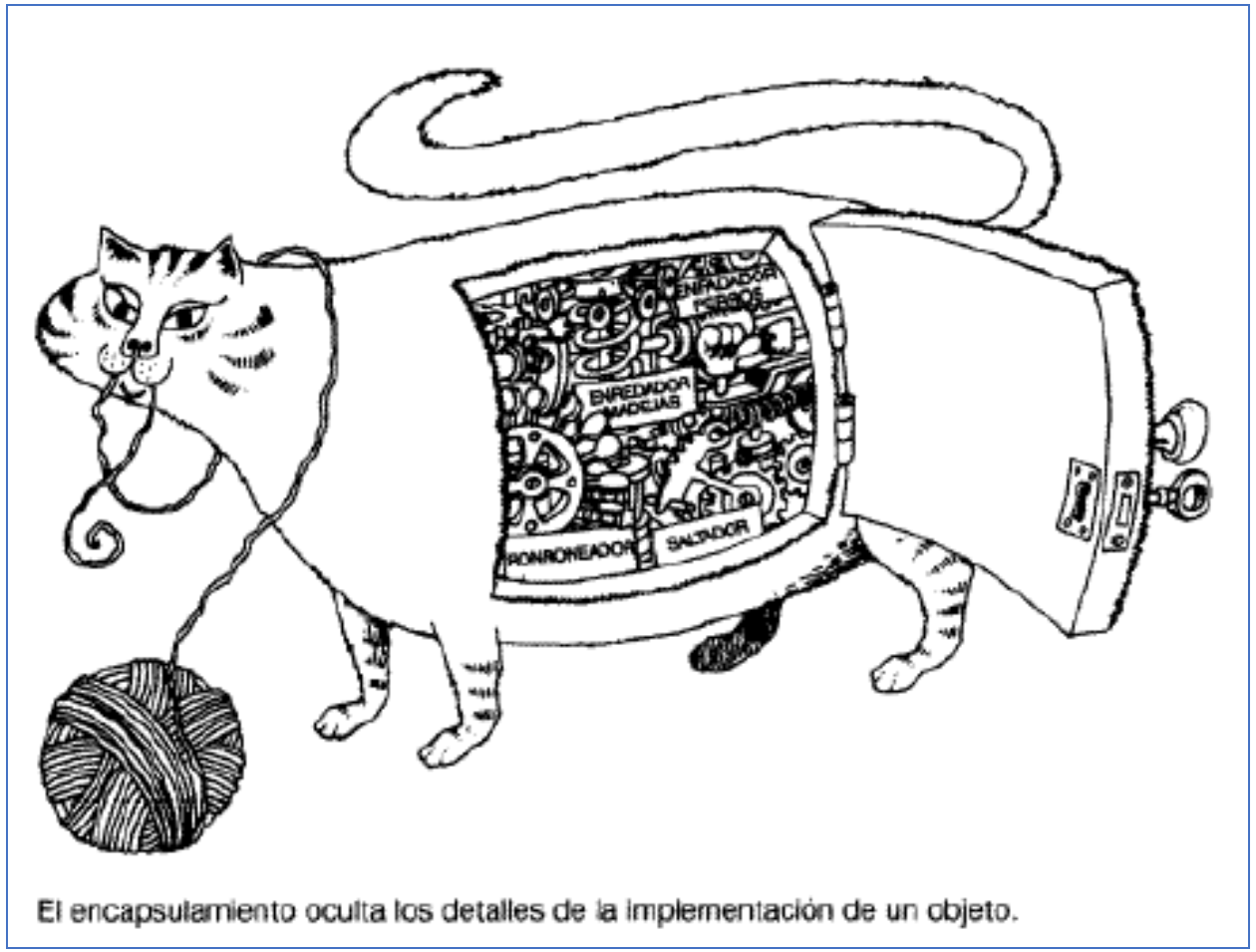
2. Encapsulamiento.-

- Ocultación de información

- Es la acción de incluir dentro de un objeto todo lo que necesita, de tal forma que ningún otro objeto necesita conocer nunca su estructura interna.

- El objeto se vería como una caja negra, en la que se ha metido de alguna manera toda la información relacionada con dicho objeto.

Encapsulamiento



El encapsulamiento oculta los detalles de la implementación de un objeto.

Encapsulación - Ocultación de datos

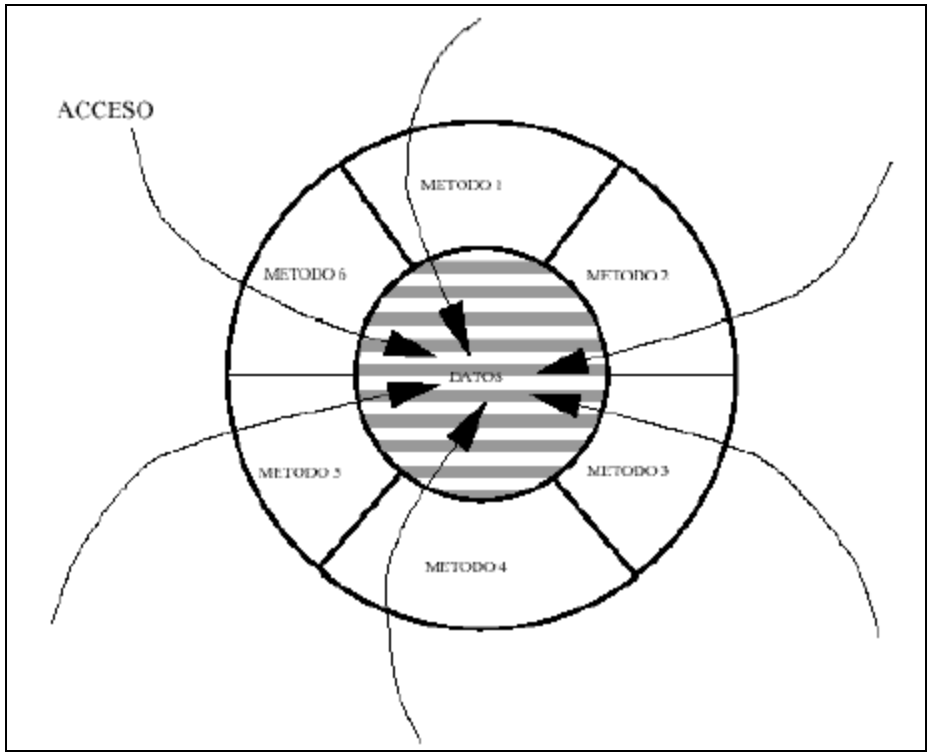
- La palabra reservada private permite una accesibilidad total desde cualquier método de la clase, pero no desde fuera de ésta.

```
public class MyDate
{
     private int day, month, year;

     public void tomorrow ()
     {
        this.day = this.day + 1;
     }
}
```

```
public class DataUser
 {
     public static void main (String args[])
     {
        MyDate mydate = new Date();
        mydate.day = 21; //Incorrecto
     }
}
```

Encapsulamiento

- Los datos o propiedades son privados y se accede a ellos a través de los métodos



Encapsulamiento

- Como los datos son inaccesibles, la única manera de leerlos o escribirlos es a través de los métodos de la clase. Todo atributo privado va a tener asociado dos métodos públicos (get y set) a través de los cuales vamos a poder modificar el valor del atributo (set) o recuperar su valor (get).

	Sintaxis	Sintaxis

GET	public (tipo de retorno)
	getAtributo()
	{
	return expresión;
	}
		public int **get**Day()
		{
		return day;
		}

SET	public void **set**Atributo(argumento)
	{
	atributo=argumento;
	}
		public void **set**Day(int hoy)
		{
		//aquí lógica de verificación
		day=hoy;
		}

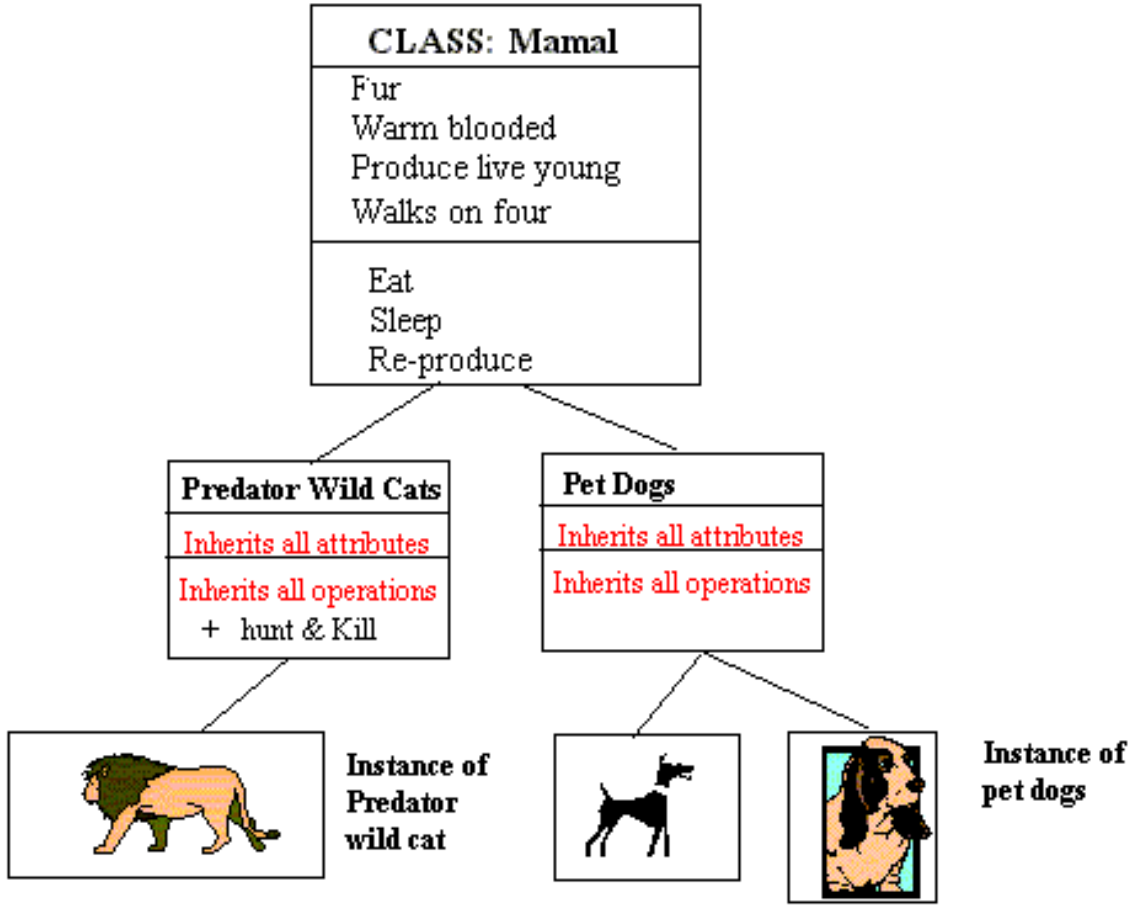
Encapsulamiento

- Esto proporciona consistencia y calidad.
- Si creamos una clase que permite acceso libre podríamos tener este tipo de errores:

```
MyDate d = new MyDate();
d.day = 32;                    Incluir días no válidos
d.month = 2;
d.day = 30;                    Posible pero incorrecto
d.month = d.month +1;          Salir del rango de meses.
```

Herencia

- La herencia es un mecanismo para compartir automáticamente métodos y atributos entre clases y subclases.
- Esta característica nos permite la reutilización del código.
- Una clase derivada puede añadir nuevos atributos y métodos y/o redefinir las variables y métodos heredados



Polimorfismo

- Es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos que son instancias de una misma jerarquía de clases.
- El polimorfismo significa “ muchas formas ”

- No se debe confundir polimorfismo con sobrecarga:
  - Sobrecarga se resuelve en tiempo de compilación, dado que los métodos se deben diferenciar en el tipo o en el número de parámetros.
  - Polimorfismo se resuelve en tiempo de ejecución, todos los métodos tienen los mismos parámetros, las acciones cambian en función del objeto al que se le aplica.

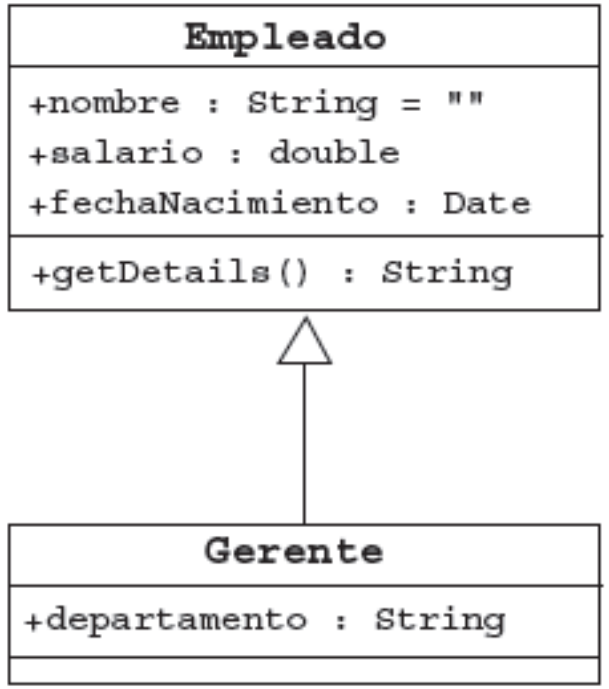
Polimorfismo

- El polimorfismo es una habilidad de tener varias formas; por ejemplo, la clase Jefe tiene acceso a los métodos de la clase Empleado.
- Un objeto tiene sólo un forma.
- Una variable tiene muchas formas, puede apuntar a un objeto de diferentes maneras.
- En Java hay una clase que es la clase padre de todas las demás: java.lang.Object.
- Un método de esta clase (por ejemplo: toString() que convierte cualquier elemento de Java a cadena de caracteres), puede ser utilizada por todos.

Herencia

```
public class Empleado {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;

  public String getDetails() {...}
}
```

```
public class Gerente extends Empleado {
  public String departamento;
}
```



Sobreescritura de métodos

```
public class Empleado {
  protected String nombre;
  protected double salario;
  protected Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
           + "Salario: " + salario;
  }
}

public class Gerente extends Empleado {
  protected String departamento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
           + "Salario: " + salario + "\n"
           + "Gerente de: " + departamento;
  }
}
```

Invocación a métodos sobreescritos

```
public class Empleado {
  private String nombre;
  private double salario;
  private Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\nSalario: " + salario;
  }
}

public class Gerente extends Empleado {
  private String departamento;

  public String getDetails() {
    // llama al método de la superclase
    return super.getDetails()
           + "\nDepartamento: " + departamento;
  }
}
```

Sobrecarga de métodos

```
public void println(int i)
public void println(float f)
public void println(String s)
```

Sobrecarga de constructores

```
public class Empleado {
  private static final double SALARIO_BASE = 15000.00;
  private String nombre;
  private double salario;
  private Date fechaNacimiento;

  public Empleado(String nombre, double salario, Date FdeNac) {
    this.nombre = nombre;
    this.salario = salario;
    this.fechaNacimiento = FdeNac;
  }
  public Empleado(String nombre, double salario) {
    this(nombre, salario, null);
  }
  public Empleado(String nombre, Date FdeNac) {
    this(nombre, SALARIO_BASE, FdeNac);
  }
  public Empleado(String nombre) {
    this(nombre, SALARIO_BASE);
  }
  // más código de Empleado...
}
```

Interfaces

- Es una colección de:
  - declaraciones de métodos (sin definirlos)
  - declaraciones de constantes.

- Las clases que implementen (implements) el interface han de definir obligatoriamente las funciones declaradas en él.

- Una clase puede implementar uno ó más interfaces.

Interfaces (cont.)

- Una interfaz se puede comparar a una clase abstracta que tiene todos sus métodos abstractos, ya que en una interfaz no puede existir ningún método con código.

- Una interfaz puede heredar de otra.

- Una clase puede implementar varias interfaces.

Interfaces (cont.)

- Sintaxis para crearla:

```
public interface nombreInterfaz [extends interface1]
```

- Sintaxis para implementarla:

```
public class nombreClase implements interf1, interf2, ...
```

Ejemplo de Interface

EL INTERFACE

```
import java.awt.Graphics;
public interface Geometria
{
    public double area();
    public void dibujar(Graphics g);
}
```

LAS CLASE QUE IMPLEMENTAN EL INTERFACE

```
public class Circulo extends Figura implements Geometria
{
    //...
}
```

```
public class Rectangulo extends Figura implements Geometria
{
    //..
}
```

Ejemplo de Interface

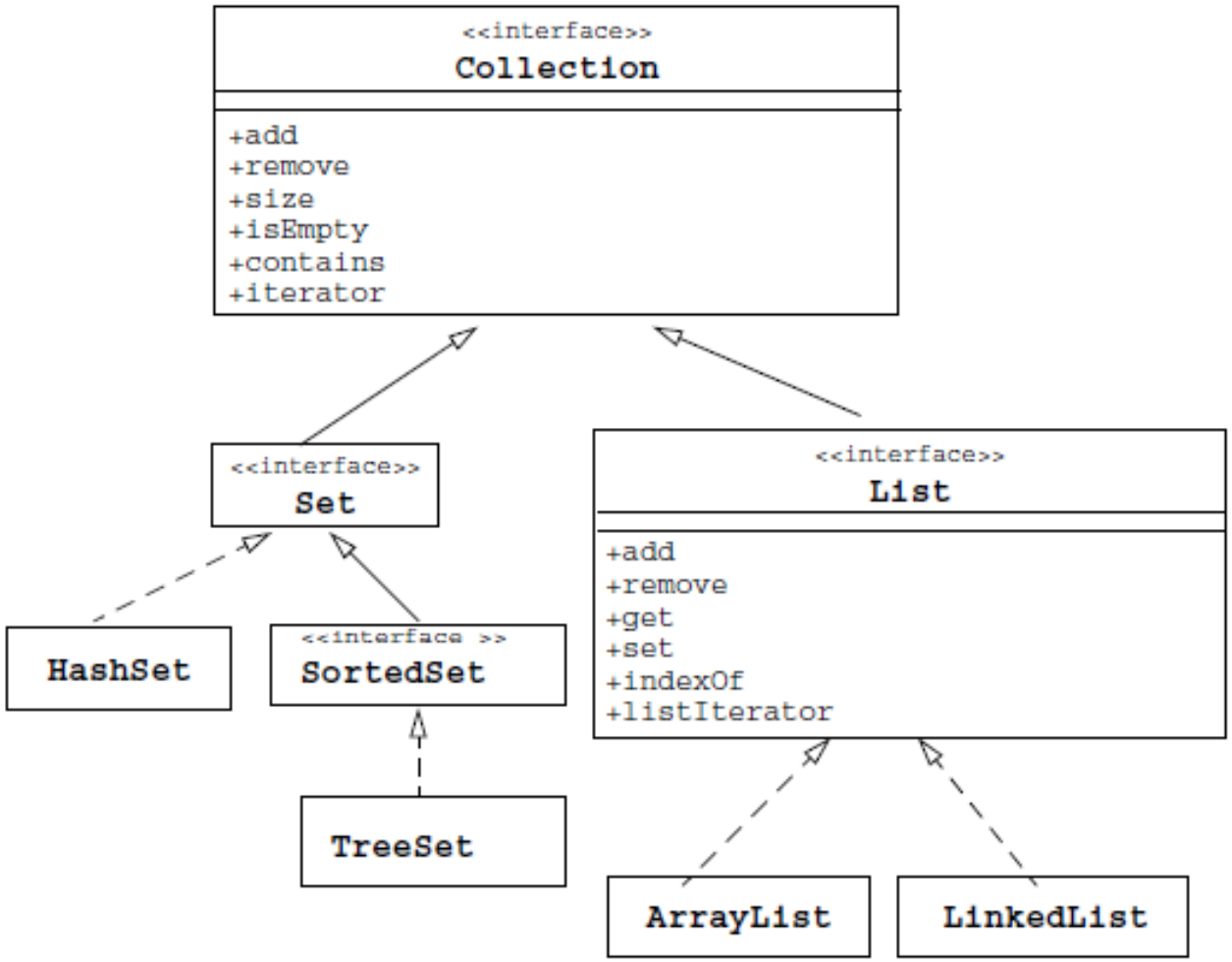
```
import java.awt.Graphics;
public class Circulo extends Figura implements Geometria
{
    //VARIABLES
    private float radio;
    public final static double  PI = 3.14;
    //CONSTRUCTORES
    // ...
    //METODOS
    //....

    public double area(){
        double a;
        a = 2*PI* radio * radio;
        return a;
    }
    public  void dibujar(Graphics g){
        g.fillOval(this.getCoordX(),this.getCoordY(),(int)radio, (int)radio);
    }

}
```

Colecciones y Genéricos

API Collections

- El API Collections contiene interfaces que permiten agrupar objetos en una de las siguientes colecciones:
- Collection; Un grupo de objetos denominados elementos, la implementación determina si guardan un orden específico y si se permiten elementos duplicados.
- Set; Es un tipo de colección donde no se garantiza que se conserve el orden de entrada de los elementos. Tampoco permite elementos duplicados.
- List – En esta colección si se garantiza el orden de entrada y si que se permiten los elementos duplicados.

API Collections



Principales Colecciones

	Hash Table	Array redimensionable	Árbol balanceado	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Set

- Recordamos que una colección de tipo Set no garantiza el orden de entrada de los elementos y tampoco permite introducir elementos duplicados.

```
// crear una coleccion de tipo Set
Set coleccionSet = new HashSet();
coleccionSet.add("uno");
coleccionSet.add("segundo");
coleccionSet.add(new Integer(4));
coleccionSet.add(new Float(3.15));
coleccionSet.add("segundo");  // los repetidos no se añaden
System.out.println(coleccionSet);
```

```
[4, 3.15, segundo, uno]
```

List

- Una colección de tipo List si que conserva el orden de entrada de los elementos y además permite introducir elementos duplicados.

```
// crear una coleccion de tipo List
List coleccionList = new ArrayList();
coleccionList.add("uno");
coleccionList.add("segundo");
coleccionList.add(new Integer(4));
coleccionList.add(new Float(3.15));
coleccionList.add("segundo");  // los repetidos SI se añaden
System.out.println(coleccionList);
```

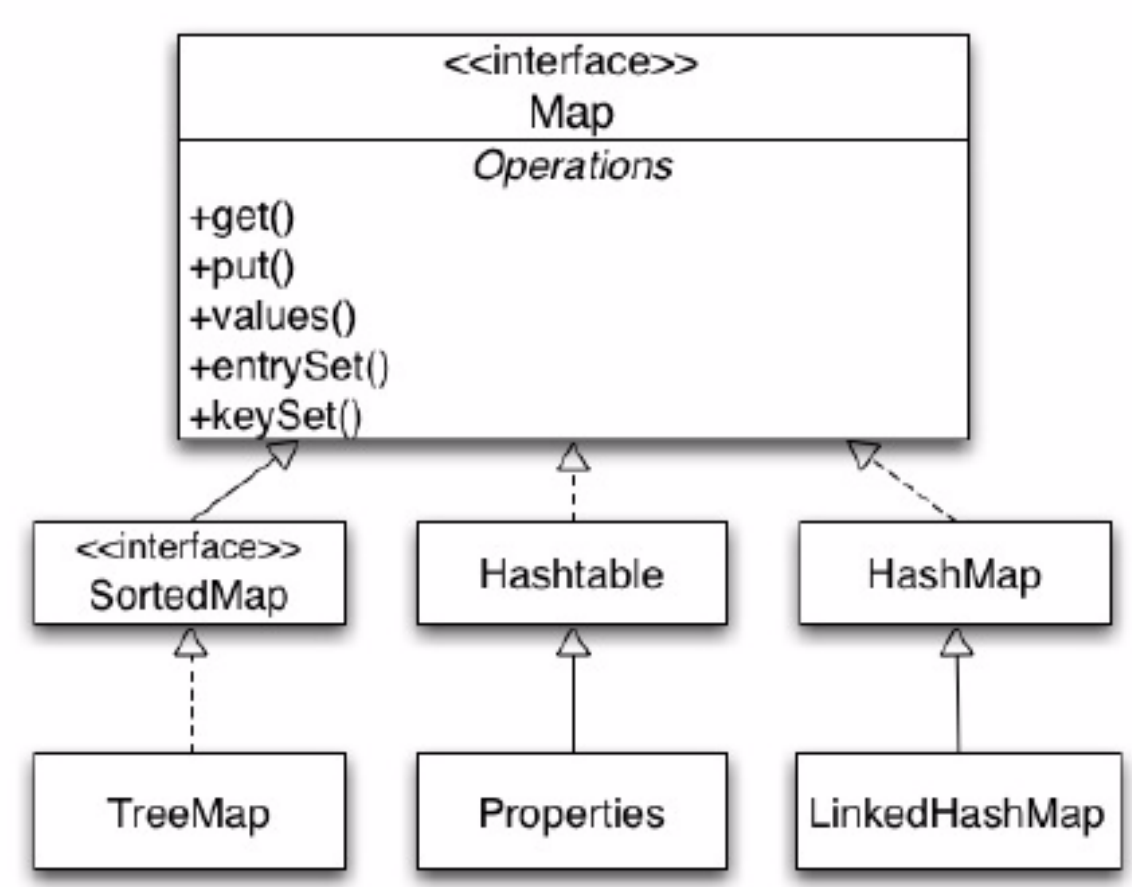
```
[uno, segundo, 4, 3.15, segundo]
```

Map

- Un mapa no se considera una colección ya que no hereda de la interface Collection.
- Los elementos de un mapa se forman como clave-valor. Además tienen las siguientes restricciones:
    - Las claves duplicadas no están permitidas.
    - Una clave solo puede referenciar un valor, no varios.

- La interface Map define una serie de métodos para manipular el mapa, los más interesantes son:
    - entrySet; Devuelve una colección de tipo Set con todos los elementos (clavevalor).
    - keySet; Devuelve una colección de tipo Set con todas las claves del mapa.
    - values; Devuelve un objeto de tipo Collection con todos los valores del mapa.

Map


<<interface>>
Map
Operations
+get()
+put()
+values()
+entrySet()
+keySet()
<<interface>>
SortedMap
Hashtable
HashMap
TreeMap
Properties
LinkedHashMap

Map

```
// crear un Mapa
// un mapa no permite claves duplicadas, valores duplicados si.
Map mapa = new HashMap();
mapa.put("1", "uno");
mapa.put("2", "dos");
mapa.put("1", "tres"); // sobreescribe el elemento

// mostrar todas las claves (keys)
System.out.println(mapa.keySet());

// mostrar todos los valores (values)
System.out.println(mapa.values());

// mostrar todos los elementos como pares key-value
System.out.println(mapa.entrySet());
```

```
[2, 1]
[dos, tres]
[2=dos, 1=tres]
```

Genéricos

- Utilizar un tipo genérico a la hora de crear la colección nos permite:
- Eliminar la necesidad de casting
- Controlar en tiempo de compilación que todos los elementos son del tipo genérico.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Genéricos

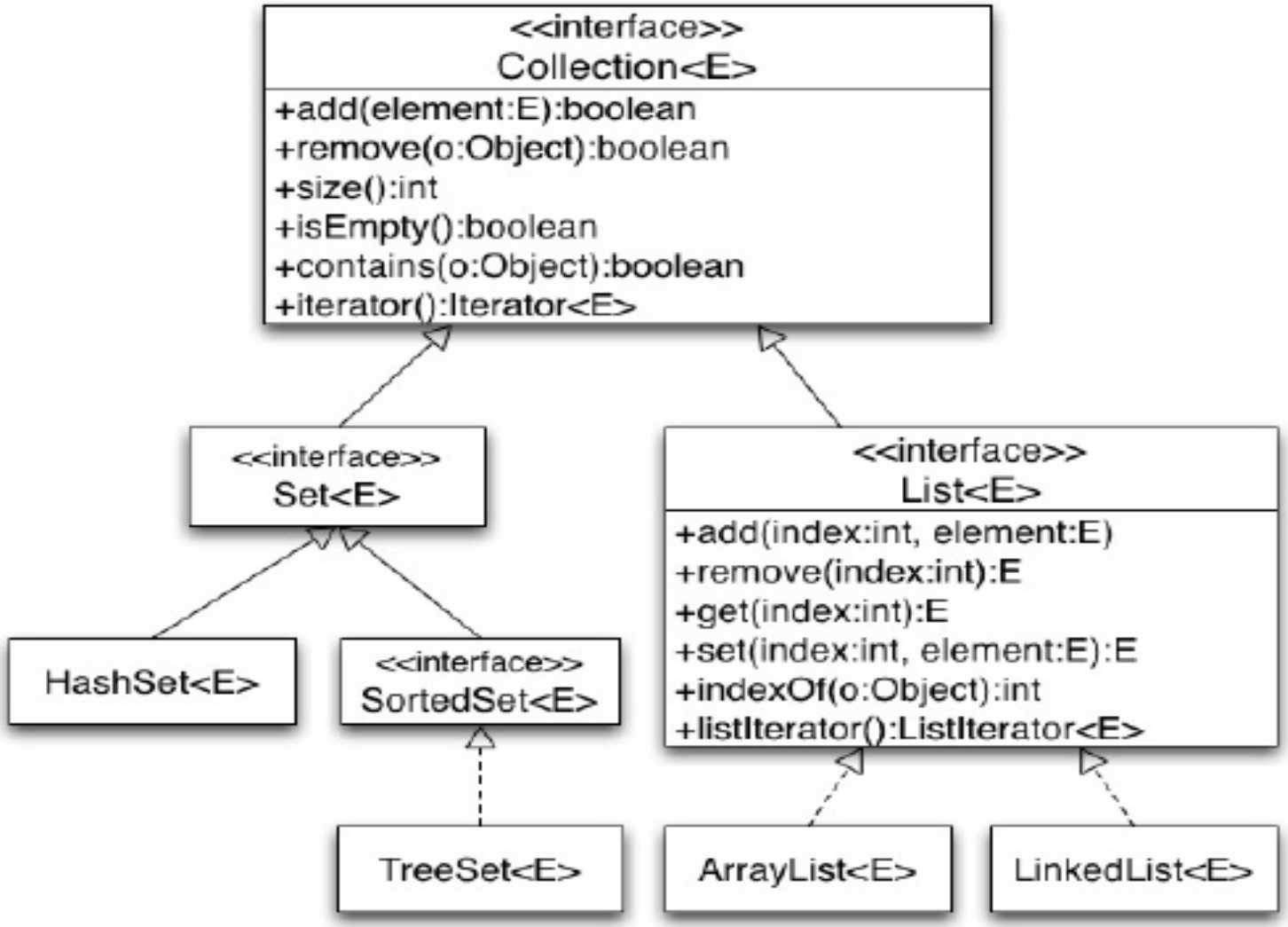
Category	Non Generic Class	Generic Class
Class declaration	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Constructor declaration	`public ArrayList (int capacity);`	`public ArrayList<E> (int capacity);`

Method declaration	`public void add((Object o)`
	
	`public Object get(int index)`
		`public void add(E o)`
		
		`public E get(int index)`

Variable declaration examples	`ArrayList list1;`
	
	`ArrayList list2;`
		`ArrayList <String> list1;`
		
		`ArrayList <Date> list2;`

Instance declaration examples	`list1 = new ArrayList(10);`
	
	`list2 = new ArrayList(10);`
		`list1= new ArrayList<String> (10);`
		
		`list2= new ArrayList<Date> (10);`

Genéricos


<<interface>>
Collection<E>
+add(element:E):boolean
+remove(o:Object):boolean
+size():int
+isEmpty():boolean
+contains(o:Object):boolean
+iterator():Iterator<E>
<<interface>>
Set<E>
<<interface>>
List<E>
+add(index:int, element:E)
+remove(index:int):E
+get(index:int):E
+set(index:int, element:E):E
+indexOf(o:Object):int
+listIterator():ListIterator<E>
HashSet<E>
<<interface>>
SortedSet<E>
TreeSet<E>
ArrayList<E>
LinkedList<E>

Lambdas

Visión general de Lambdas

- Una interface funcional es una interfaz que proporciona un único método abstracto

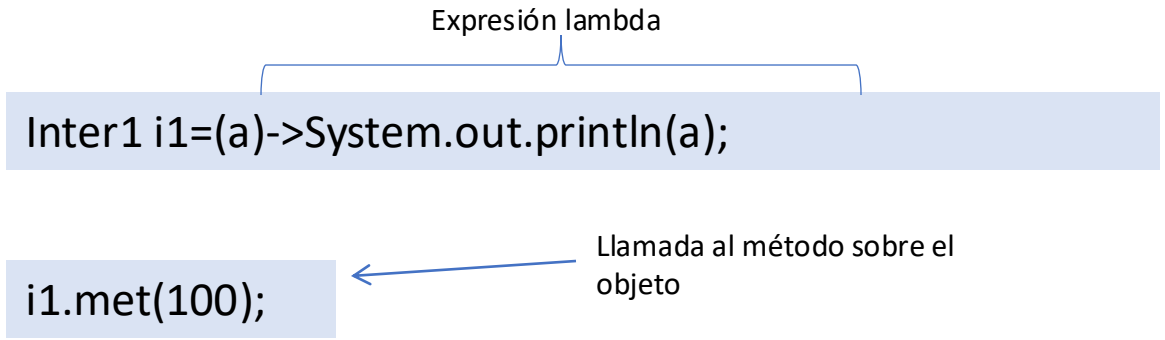
```
public interface Runnable{
   void run();
}
```

```
public interface Inter2{
   boolean process(int n, String pt);
   static void print(){}
}
```

```
public interface Inter1{
   void met(int data);
   default int res(){return 1;}
}
```

Visión general de Lambdas

- Que es una expresión lambda?
- Implementación de una interfaz funcional
- Proporciona el código del único método abstracto de la interfaz, a la vez que genera un objeto que implementa la misma



Visión general de Lambdas

- Una expresión lambda tiene dos partes, la lista de parámetros del método y la implementación:

```
parametros->implementación
```

- Los parámetros pueden indicar o no el tipo
- La lista de parámetros se puede indicar o no entre paréntesis (obligatorio si hay dos o más) y también si se indica el tipo
- En caso de devolver un resultado, la implementación puede omitir la palabra return si consta de una sola instrucción

Streams

- Creación de un stream
  - A partir de una colección:

```
ArrayList<Integer> nums=new ArrayList<>();
nums.add(20);nums.add(100);nums.add(8);
Stream<Integer> st=nums.stream();
```

  - A partir de un array:

```
String[] cads={"a","xy","jk","mv"};
Stream<String>st= Arrays.stream(cads);
```

  - A partir de una serie discreta de datos:

```
Stream<Double> st=Stream.of(2.4, 7.4, 9.1);
```

  - A partir de un rango de datos:

```
IntStream stint=IntStream.range(1,10);
IntStream stint2=IntStream.rangeClosed(1,10);
```

Stream de tipos primitivos

Streams

- Tipos de métodos de Stream

  - Métodos intermedios. El resultado de su ejecución es un nuevo Stream. Ejemplos: filtrado y transformación de datos, ordenación, etc.

  - Métodos finales. Generan un resultado. Pueden ser void o devolver un valor resultado de alguna operación. Ejemplos: calculo (suma, mayor, menor, …), búsquedas, reducción, etc.

Control de Excepciones

Introducción

- El mecanismo implementado por Java para indicar que sucedió algo fuera de lo normal es la gestión de excepciones.

- Cuando ocurre un situación que es preciso gestionar como excepcional, el código genera una excepción.

- Una excepción es cualquier situación que interrumpe el flujo natural de la ejecución de un programa.

Excepciones de uso frecuente

- ArithmeticException; Excepcion en una operación aritmética, p.e. división por cero.
- ArrayStoreException;Excepción en una operación en el uso de un array.
- NegativeArraySizeException; Excepción en una operación en el uso de un array.
- NullPointerException; Excepción producida por una referencia a un objeto nulo.

Excepciones de uso frecuente (cont.)

- SecurityException; Excepción en el sistema de seguridad.
- EOFException; Excepción producida por alcanzar el final de un fichero.
- IOException; Excepción en un proceso de entrada y salida.
- FileNotFoundException; Excepción por no encontrar un fichero.
- NumberFormatException; Excepción en la conversión de una cadena de caracteres a un valor numérico.

Captura de excepciones

- Sintaxis:

```
try {
        //Bloque de sentencias que podrían generar una excepción.
} catch (clase_de_excepcion_1 e){
        //sentencias que se ejecutan si se ha producido una
        //excepción de la clase clase_de_excepcion_1.
} catch (clase_de_excepcion_2 e){
        // sentencias que se ejecutan si se ha producido una
        //excepción de la clase clase_de_excepcion_2.
} catch (Exception e){
        // sentencias que se ejecutan si se ha producido una
        // excepción no capturada anteriormente.
} finally {
        //Bloque de sentencias que se ejecutan siempre.
}
```

Captura de excepciones

- Si la excepción producida no es de la clase del primer catch se investiga el siguiente, y así hasta que algún bloque catch lo capture. El último bloque captura cualquier clase, ya que todas las clases de excepciones derivan de la clase Exception.

```java
import java.io.*;
public class excepciones {
    public static void main(String args[]) throws IOException{
                int resultado;
                int num1=7;
                int num2=0;
                try{
                        resultado=(num1/num2);
                        System.out.println(resultado);
                }
                catch(ArithmeticException e){
                        System.out.println("No se puede dividir por 0");
                }
                catch(Exception e){
                        System.out.println("Es otro tipo de error");
                }
        }
}
```

Métodos útiles

- getMessage(); pertenece a la clase Exception y muestra información de la excepción que se ha producido.
- Ejemplo: System.out.println(e.getMessage());

- printStackTrace(); imprime en la consola del sistema los nombres de los métodos llamados hasta el método que ha producido la excepción.
- Ejemplo: e.printStackTrace();

Generación de excepciones, sentencia throw

- Para producir la generación de una excepción en nuestro código se utiliza una instrucción como esta:

```
throw new objetoException();
```

- El objeto excepción es de una clase determinada y siempre es una clase derivada de la clase Throwable definida en el paquete java.lang.

```java
public class Persona2 {
    private int edad;

    public void ponEdad(int edadNueva) throws Exception {
        if (edadNueva < 0 || edadNueva > 100) throw (new Exception("Edad no valida"));
            edad = edadNueva;

    }
    public String toString(){
        return "Edad " + edad;
    }
}
```

```java
public class VerPersona2 {

   public static void main(String args[]){
        Persona2 miobjeto = new Persona2();
        try{
            int edad=15;
            miobjeto.ponEdad(edad);
            System.out.println(miobjeto.toString());
        }catch(Exception e){
            System.out.println(e.getMessage());
        }
    }

}
```

Excepciones personalizadas

- Java nos permite especializar las excepciones creando una clase derivada de la clase Exception o de alguna de sus clases hijas que no sean finales.

- Entre otras cosas podemos reescribir el método getMessage() para incluir un mensaje personalizado.

```
public class MiExcepcion extends Exception{
    public MiExcepcion(){
        super("No se puede dividir por cero");
    }
}
```

```
import java.io.*;
public class excepciones {
   public static void main(String args[]) throws MiExcepcion{
                int resultado;
                int num1=7;
                int num2=0;
                try{
                    if (num2==0){
                        throw new MiExcepcion();
                    }
                    else{
                      resultado=(num1/num2);
                    }
                }
                catch(MiExcepcion e){
                        System.out.println(e.getMessage());
                }
    }
  }
```

Icono *Training Consulting*

Iniciación Programación Java

Completa nuestra encuesta de satisfacción a través del QR

GRACIAS

2025