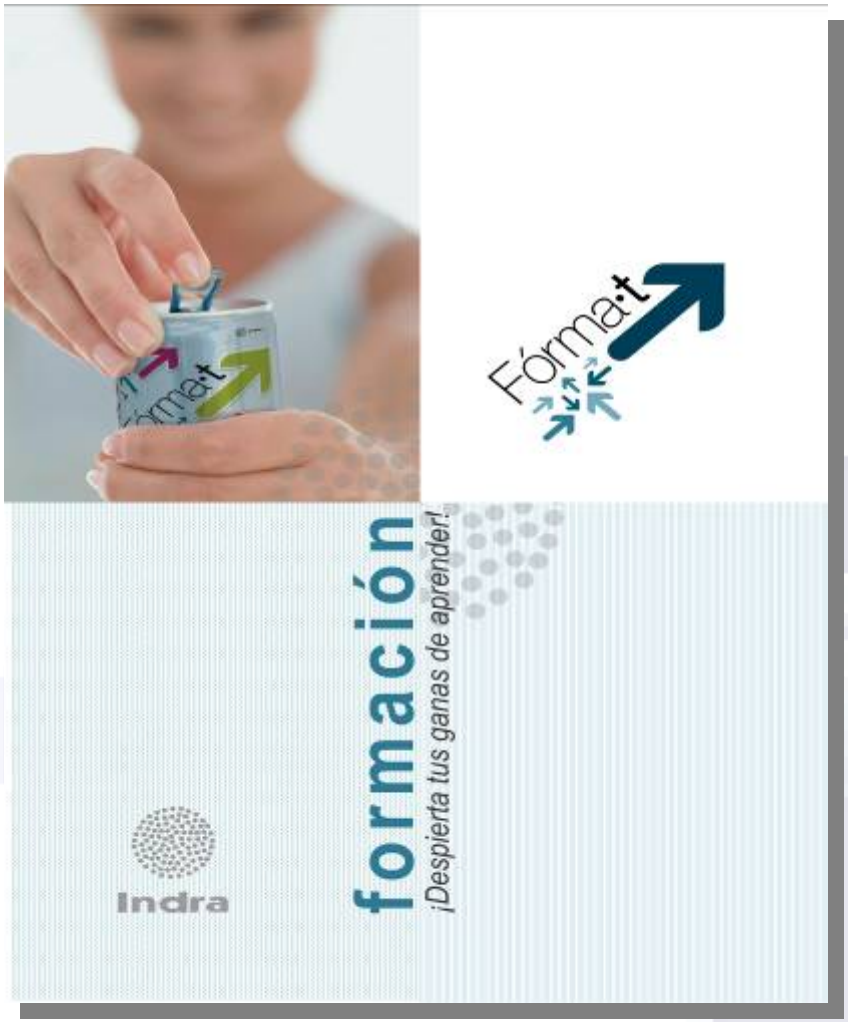


T- Programación

Programación Orientada a Objetos con Java.

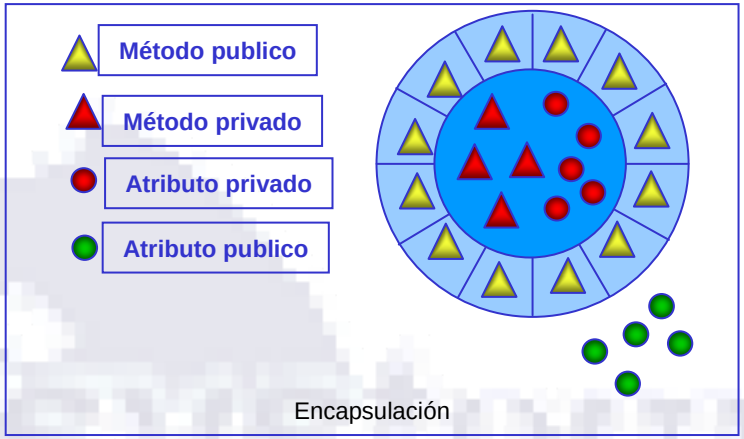
internet systems

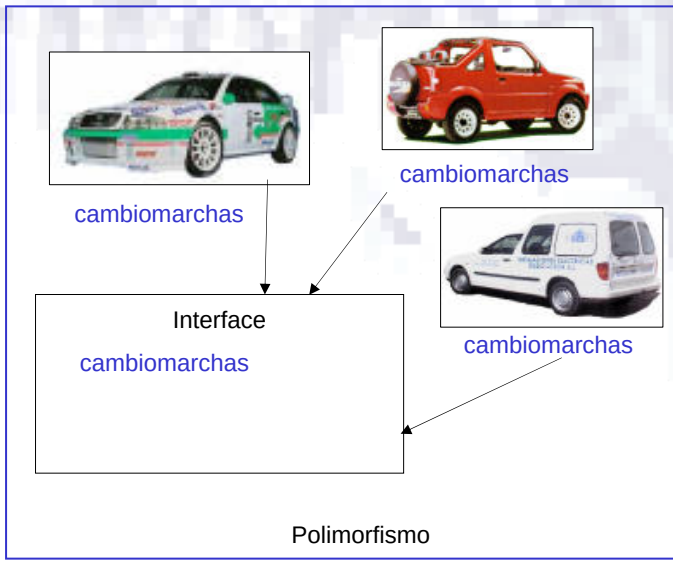
Indra

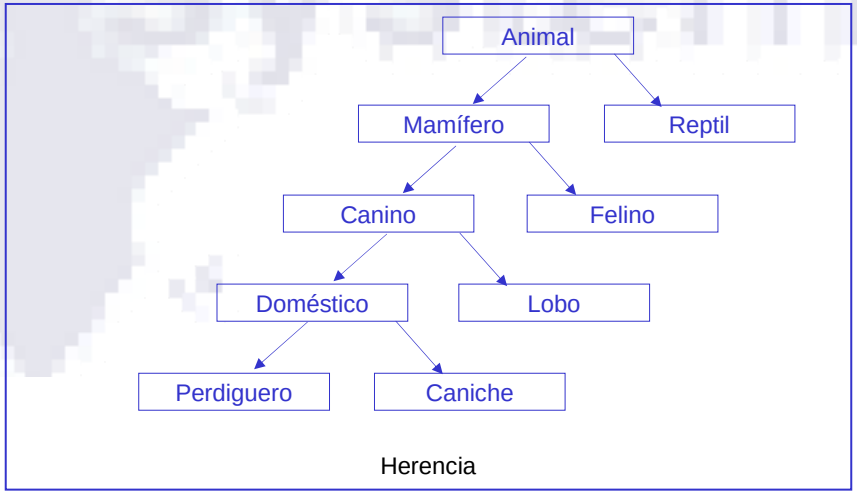
CONCEPTOS GENERALES

Introducción al lenguaje Java

- Lenguaje orientado a objetos
  - Encapsulación
  - Herencia
  - Polimorfismo

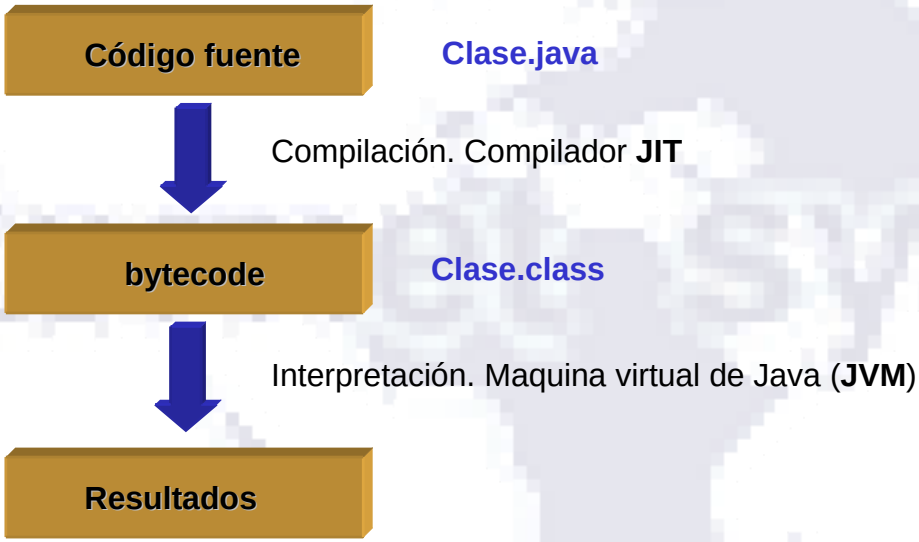






Introducción al lenguaje Java

- Lenguaje compilado e interpretado




internet systems

Variables

- Una **variable** es un **nombre** que contiene un valor que puede cambiar a lo largo del programa.
- Hay dos tipos principales de variables:
  - Variables de **tipos primitivos**. Están definidas mediante un valor único.
  - Variables **referencia**. Se refieren a una información más compleja: **arrays** u **objetos** de una determinada clase
- Desde el punto de vista de su papel en el programa, las variables pueden ser:
  - Variables **miembro** de una clase: Se definen en una clase, fuera de cualquier método. Pueden ser tipos primitivos o referencias.
  - Variables **locales**: Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves { }. Pueden ser también tipos primitivos o referencias.

Variables

- Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, excepto algunos caracteres especiales como operadores o separadores (, . + - * /).
- Existe ***palabras reservadas*** las cuales tienen un significado especial para ***Java*** y que no se pueden utilizar como nombres de variables.

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	null
package	private	protected	public	return	short
static	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

Tipos de datos

- Es un lenguaje fuertemente tipado.
- Existen dos tipos de datos que se pueden asignar a las variables.
  - **Tipos Simples**: Definidos por el lenguaje
  - **Tipos Referencia**: Apuntan a objetos por medio de sus posiciones de memoria.

Tipos de datos simples

Tipo de variable	Tamaño	Descripción
Boolean.	1 byte	Valores true y false
Char.	2 bytes	Comprende el código ASCII
Byte.	1 byte	Valor entero entre -128 y 127
Short.	2 bytes	Valor entero entre -32768 y 32767
Int.	4 bytes	Valor entero entre -2.147.483.648 y 2.147.483.647
Long.	8 bytes	Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes	De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes	De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tipos de datos simples

- **boolean** no es un valor numérico: sólo admite los valores **true** o **false**. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser **boolean**.
- **char** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter.
- **byte**, **short**, **int** y **long** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos.
- Los tipos **float** y **double** son valores reales con 6-7 (float) y 15 (double) cifras decimales.

Tipos de datos simples

- **Conversiones de tipos**
  - **Implícitas**: Se realizan de modo automático **de un tipo a otro de más precisión**, por ejemplo de **int** a **long**, de **float** a **double**, etc.

```
byte a=40;
byte b=50;
byte c= 100;
int d=a * b / c;
```

a * b podría superar el rango definido para un byte. Java promociona automáticamente el resultado a int

  - **Explícitas**: Son las conversiones de un tipo de mayor a otro de menor precisión. A estas conversiones explícitas se les llama **cast**.

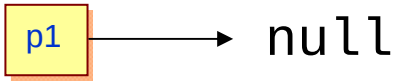
```
int a;
byte b;
b = (byte) a;
```

  - No se puede convertir un tipo numérico a **boolean**.

Tipos de datos referencia

- Una **referencia** es una variable que indica dónde está en la memoria del ordenador un objeto.
- Al declarar una referencia todavía no se encuentra “apuntando” a ningún objeto en particular, luego se le asigna el valor **null**.
- Si se desea que esta **referencia** apunte a un nuevo objeto es necesario utilizar el operador **new**. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones).
- También es posible igualar la **referencia** declarada a un objeto existente previamente.

```
Punto p1;
```



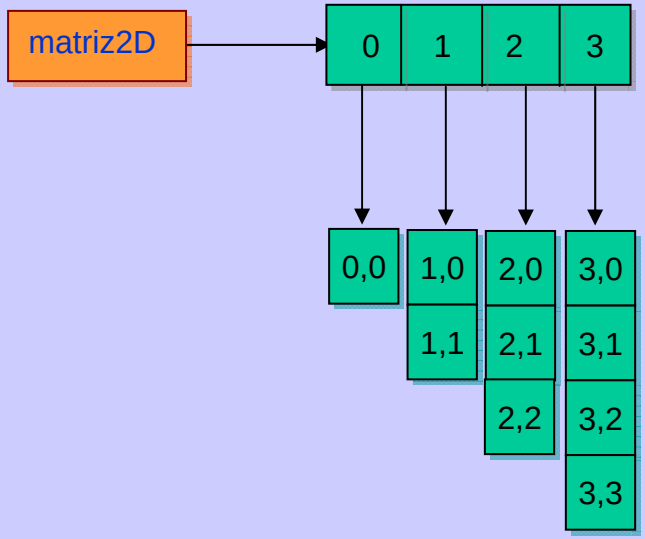

```
p1=new Punto();
```

Tipos de datos especiales

```
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

Tipos de datos especiales

```
class DosDimensiones {
  public static void main(String args[]) {
    int matriz2D[][] = new int[4][];
    matriz2D[0] = new int[1];
    matriz2D[1] = new int[2];
    matriz2D[2] = new int[3];
    matriz2D[3] = new int[4];
    int i, j, k = 0;
    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        matriz2D[i][j] = k;
        k++;
      }
    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(matriz2D[i][j] + " ");
      System.out.println();
    }
  }
}
```



Tipos de datos especiales

- Cadenas de caracteres.
  - **String** es el tipo cadena.
  - No es un tipo de datos simple.
  - Tampoco es un array de char.
  - Es un objeto con sus atributos, métodos y constructores.

```
String str = "esto es una cadena";
String str = new String ("esto es una cadena");
```

Visibilidad de las variables

- Se entiende por **visibilidad** o **scope** de una variable, la parte de la aplicación donde dicha variable es accesible.
- Todos las variables deben estar incluidas en una clase.
- Las variables declaradas dentro de unas llaves **{ }**, es decir dentro de un **bloque**, son visibles y existen dentro de estas llaves.
  - Las variables declaradas al principio de una función existen mientras se ejecute la función
  - Las variables declaradas dentro de un bloque **if** no serán válidas al finalizar las sentencias correspondientes a dicho **if**
  - Las variables miembro de una **clase** (es decir declaradas entre las llaves { } de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Operadores

- OPERADORES ARITMÉTICOS
  Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales .

Nombre	Representación
Suma	+
Resta	-
Producto	*
División	/
Resto de la div.	%

- OPERADORES DE ASIGNACIÓN
  Los operadores de asignación permiten asignar un valor a una variable .

Operador	uso	equivalente
+=	op1 += op2	op1= op1 + op2
-=	op1 -= op2	op1= op1 - op2
*=	op1 *= op2	op1= op1 * op2
/=	op1 /= op2	op1= op1 / op2
%=	op1 %= op2	op1= op1 % op2

Operadores

- OPERADORES INCREMENTALES incrementan o decrementan el valor de una variable en una unidad.

nombre	uso	equivalente
preincremento	++ i	i = i + 1
predecremento	-- i	i = i - 1
postincremento	i ++	i = i + 1
postdecremento	i --	i = i - 1

```
a = 5;
b = 2;
a = ++ b;
```

Después de la ejecución

a = 3 y b = 3

```
a = 5;
b = 2;
a = b ++;
```

Después de la ejecución

a = 2 y b = 3

Operadores

Operador	uso	nombre
>	op1 > op2	Mayor
>=	op1 >= op2	Mayor o igual
<	op1 < op2	Menor
<=	op1 <= op2	Menor o igual
==	op1 == op2	Igual
!=	op1 != op2	distinto

OPERADORES RELACIONALES realizan comparaciones de igualdad, desigualdad y relación de menor o mayor.

El resultado de estos operadores es siempre un valor boolean.

Operadores

Operador	Nombre	Acción
&	AND	True si los dos son true
|	OR	True si uno de los dos es true
!	NOT	Lo contrario
&&	AND en cortocircuito	True si los dos son true. Si el primero es false no se evalúa el segundo
||	OR en cortocircuito	True si uno de los dos es true. Si el primero es true no se evalúa el segundo

OPERADORES LÓGICOS.

Se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales.

Operadores

OPERADOR CONDICIONAL.

permite realizar bifurcaciones condicionales sencillas.

- **expresionBooleana ? res1 : res2**

Donde se evalúa expresionBooleana y se devuelve **res1** si el resultado es true y **res2** si el resultado es false.

```
x = 1;
y = 10;
z = (x < y) ? x+3 : y+8;
```

Asignaría a z el valor 4, es decir x+3

Conceptos generales

- SENTENCIAS O EXPRESIONES
  - Una **expresión** es un conjunto de variables unidas por operadores.
  - Una **sentencia** es una expresión que acaba en punto y coma (;). Se permite incluir varias sentencias en una línea

```
i = 0; j = 5; x = i + j;
```

- COMENTARIOS
  - Comentario de una línea //
  - Comentario de varias líneas /*

```
.....
..... */
```

Bifurcaciones

- if
  - Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor true).

```
if (ExpresiónBooleana) {
        sentencias;
}
```

Las llaves { } sirven para agrupar en un bloque las sentencias que se han de ejecutar. No son necesarias si sólo hay una sentencia dentro del if.

Bifurcaciones

- if else
  - Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el ***else*** se ejecutan en el caso de no cumplirse la expresión de comparación (***false***).

```
if (ExpresiónBooleana) {
        sentencias1;
}else{
        sentencias2;
}
```

Bifurcaciones

```
int numero = 61; // "numero" tiene dos dígitos
if(Math.abs(numero) < 10) // Math.abs() calcula el valor absoluto. (false)
    System.out.println("Numero tiene 1 digito ");
else if (Math.abs(numero) < 100)  // Si numero es 61, estamos en este caso
    System.out.println("Numero tiene 2 digitos ");
else {   // Resto de los casos
    System.out.println("Numero tiene mas de 2 digitos ");
    System.out.println("Se ha ejecutado la opcion por defecto ");
}
```

```
        }
```

Bifurcaciones

```
char c = (char)(Math.random()*26+'a'); // Generación aleatoria de letras
                                       // minúsculas
System.out.println("La letra " + c );
switch (c) {
        case 'a': // Se compara con la letra a
        case 'e': // Se compara con la letra e
        case 'i': // Se compara con la letra i
        case 'o': // Se compara con la letra o
        case 'u': // Se compara con la letra u
            System.out.println(" Es una vocal ");
            break;
        default:
            System.out.println(" Es una consonante ");
}
```

Bucles

- while
  - Las sentencias se ejecutan mientras la **Expresión booleana** sea **true**.

```
while (Expresión booleana ) {
    sentencias;
}
```

Bucles

```
for(int i = 1, j = i + 10; i < 5; i++, j = 2*i) {

        System.out.println(" i = " + i + " j = " + j);

}
```

Genera la salida

```
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

Bucles

- **do while**
  - Es similar al bucle **while** pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no).

```
do {
   sentencias
} while (Expresión booleana );
```

Sentencias especiales

- Sentencias **break** y **continue**
  - **break** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.
  - **continue** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración “i” que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

Sentencias especiales

- Sentencia **return**
  - Es otra forma de salir de un bucle o de un método. A diferencia de **continue** o **break**, la sentencia **return** sale también del método o función.
  - En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (return value;).
- Bloque **try** {...} **catch** {...} **finally** {...}
  - Se utilizan en la gestión de excepciones

CLASES

Conceptos básicos

- Una clase es una agrupación de **datos** (variables o campos) y de **funciones** (métodos) que operan sobre esos datos.

```
[public] class Classname {
        // definición de variables y métodos
        ...
}
```

- La palabra **public** es opcional: si no se pone, la clase sólo es visible para las demás clases del **package**. Todos los métodos y variables deben ser definidos dentro del **bloque** {...} de la clase.
- Un **objeto** (instancia) es un ejemplar concreto de una clase. Las **clases** son como tipos de variables, mientras que los **objetos** son como variables concretas de un tipo determinado.

Conceptos básicos

- Una clase proporciona una definición para un tipo particular de objeto: define sus datos y la funcionalidad sobre dichos datos.
- Una clase es una plantilla.
- Hay una copia de esa clase por programa pero muchos objetos de esa clase en el mismo programa.
- Los métodos definen las operaciones que se pueden realizar, qué es lo que hace la clase. no pueden existir fuera de las clases.
- Todas las variables y funciones de Java deben pertenecer a una clase. No hay variables y funciones globales.
- Java tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.

Conceptos básicos

- En Java no hay herencia múltiple. Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object.
- La clase Object es la base de toda la jerarquía de clases de Java.
- En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
- Las clases se pueden agrupar en paquetes, introduciendo una línea al comienzo del fichero (package nombrepaquete;).

Ejemplo de clase

```
public class Punto {
   public double x;
   public double y;
   public Punto(double a, double b) {
       x=a;
       y=b;
   }
   public double calcularDistancia() {
       double z;
       z=Math.sqrt((x*x)+(y*y));
       return z;
   }
}
```

atributos

constructor

método

Clase Punto.class

Elementos de una clase. Atributos

- Los atributos definen la estructura de los objetos que instancien una clase.
- Cada **objeto** que se crea de una clase tiene **su propia copia** de los atributos y de los métodos. Por ejemplo, cada objeto de la clase **Punto** tiene sus propias coordenadas **x** e **y**.
- Para acceder al valor de un atributo se escribe:

  Objeto.atributo;

```
p1.x=5;
p1.y=7;
System.out.println(p1.x+" , "+p1.y);
```

- Para aplicar un método a un objeto concreto :

  Objeto.método();

```
double d = p1.calcularDistancia();
System.out.println("distancia: "+d);
```

Elementos de una clase. Métodos

- Los **métodos** son funciones definidas dentro de una clase. Salvo los métodos **static** o de clase, se aplican siempre a un objeto de la clase por medio del **operador punto** (.).
- Dicho objeto es su **argumento implícito**. Esto implica que se puede hacer referencia a sus atributos dentro del método. De todas formas, también se puede acceder a ellas mediante la referencia **this**.
- Los métodos pueden además tener otros **argumentos explícitos** que van entre paréntesis, a continuación del nombre del método.
- La primera línea de la definición de un método se llama **declaración.**
- El **valor de retorno** puede ser un valor de un **tipo primitivo** o una **referencia**.

Elementos de una clase. Métodos

- No puede haber más que un único valor de retorno .
- Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.
- Los métodos pueden definir **variables locales**. Su visibilidad estará limitada al propio método. Las variables locales no se inicializan por defecto.

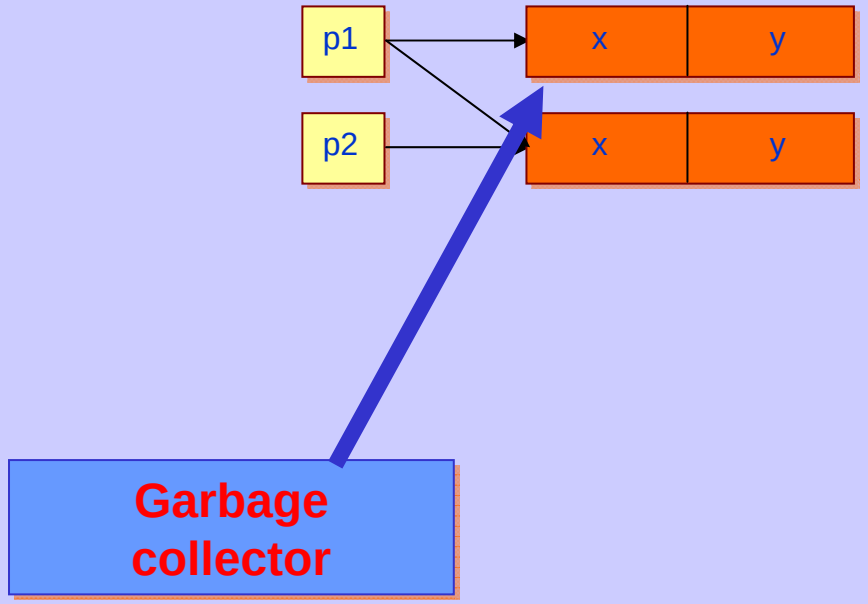
```
public Punto elPuntoMasLejano(Punto c) {
        int a;
        if (this.x >= c.x)
                return this;
        else
                a++;   //error: a no está inicializada
                return c;
}
```

Elementos de una clase. Constructores

- Un **constructor** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del **constructor** es reservar memoria e inicializar las variables miembro de la clase.
- Sirven para la inicialización correcta de objetos.
- Los **constructores** no tienen valor de retorno (ni siquiera **void**) y se tienen que llamar igual que la clase.
- Una clase puede tener **varios constructores**, que se diferencian por el tipo y número de sus argumentos (**sobrecargados**).
- Se llama **constructor por defecto** al constructor que no tiene argumentos. Cuando no hay constructor, el compilador crea uno por defecto.
- los **constructores** sólo pueden ser llamados por otros **constructores** o por métodos **static**. No pueden ser llamados por los **métodos de objeto** de la clase.

Elementos de una clase. Destructores

```
Punto p1=new Punto(2,3);

Punto p2=new Punto(5,7);

p1=p2;
```



Sobrecar

```
class Principal {
  public static void main(String args[]) {
    SobrecargaDemo ob = new SobrecargaDemo();
    double res;
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.2);
    System.out.println(res);
  }
}
```

```
class SobrecargaDemo {
  void test(){
    System.out.println("vacio");
  }

  void test(int a) {
    System.out.println("a: " + a);
  }

  void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }

  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
```

Sobrecarga de constructores

```
class Caja{
    double alto;
    double ancho;
    double largo;
    Caja(double a, double b, double c){
          alto = a;
          ancho = b;
          largo = c;
    }
    Caja() {
          alto = -1;
          ancho = -1;
          largo = -1;
    }
    Caja(double longitud) {
          alto = ancho = largo = longitud;
    }
}
```

```
class Principal{
    public static void main(String args[]){
          Caja c1=new Caja(10,20,15);
          Caja c2=new Caja();
          Caja c3=new Caja(5);
          .
          .
          .
    }
}
```

Paso de objetos como parámetros

- Los objetos también pueden pasar a un método como parámetros, ya que son variables del tipo definido por su clase.

```
class Caja{
    double alto;
    double ancho;
    double largo;
    Caja( double a, double b, double c ){
          alto = a;
          ancho = b;
          largo = c;
    }
    Caja(double ob ){
          alto = ob.alto;
          ancho = ob.ancho;
          largo = ob.largo;
    }
}
```

```
class Principal{
    public static void main(String args[]){
          Caja c1=new Caja(10,20,15);
          Caja cajaclonada=new Caja(c1);
          .
          .
          .
    }
}
```

Paso de variables por valor y por referencia

- Las variables de
  pasarán de difer
  dato de origen
  - Por valor:

```
class Principal{
    public static void main(String args[]){
        Test ob=new Test();
        int a=15;
        int b=20;
        S.o.p("a y b antes:"+a+" "+b);
        ob.metodo(a,b);
        S.o.p("a y b depues:"+a+" "+b);
    }
}
```

```
class Test{
   void metodo( int i, int j ){
        i=i*2;
        j=j/2;
   }
}
```

a y b antes: 15 20
a y b después: 15 20

Las operaciones que se realizan sobre tipos de datos simples no afectan a las variables

Paso de variables por valor y por referencia

- Por referenc

```
class Test{
   int a,b;
   Test( int i, int j ){
        a=i;
        b=j;
   }
   void metodo( Test ob ){
        ob.a *=2;
        ob.b /=2;
   }
}
```

```
class Principal{
    public static void main(String args[]){
        Test ob=new Test(15,20);
        S.o.p("ob.a y ob.b antes:"+ob.a+" "+ob.b);
        ob.metodo(ob);
        S.o.p("ob.a y ob.b despues:"+ob.a+" "+ob.b);
    }
}
```

ob.a y ob.b antes: 15 20
ob.a y ob.b después: 30 10

Las operaciones que se realizan sobre objetos si afectan a los atributos de estos

Control de acceso

```
class Test {
  int a; // acceso por defecto

  public int b; // acceso publico
  private int c; // acceso privado

  void setc(int i) {
        c = i;
  }
  int getc() {
        return c;
  }
}
```

pueden estar precedidos por

```
class Acceso {
  public static void main(String args[]) {
    Test ob = new Test();

    ob.a = 10;
    ob.b = 20;

    ob.c = 100; // Error!

    ob.setc(100); // OK

    System.out.println("a, b, y c: " +
         ob.a + " " +ob.b + " " + ob.getc());
  }
}
```


internet systems

static

```
class DemoStatic {
        int c;
        static int a = 42;
}

class Principal {
  public static void main(String args[]) {
        DemoStatic ob1=new DemoStatic();
        DemoStatic ob2=new DemoStatic();
        ob1.c =5;
        ob1.a=7;
        System.out.println("ob1.c = " + ob1.c);
        System.out.println("ob1.a = " + ob1.a);
        ob2.c =15;
        ob2.a=10;
        System.out.println("ob2.c = " + ob2.c);
        System.out.println("ob2.a = " + ob2.a);
        System.out.println("ob1.a = " + ob1.a);
        System.out.println("a = " + DemoStatic.a);
  }
}
```

static

```
class DemoStatic {
     static int a = 42;
     static int b = 99;

  static {
     System.out.println("cargando clase...");
     b = a * 4;
  }
  static void llamada() {
     System.out.println("a = " + a);
  }
}

class Principal {
  public static void main(String args[]) {
     System.out.println("antes de llamar al metodo");
     DemoStatic.llamada();
     System.out.println("b = " + DemoStatic.b);
  }
}
```

final

- Se utiliza para declarar variables de manera que su contenido no pueda ser modificado.
- Una variable **final** tiene que ser inicializada en su declaración.
- Una variable **final** se convierte en una constante.
- Por convenio, el identificador de una variable **final**, debe escribirse en mayúsculas.

```
final int NUEVO_ARCHIVO=3;
```

HERENCIA

```
class A {
  int i, j;
  void mostrarij() {
    S.o.p("i y j: " + i + " " + j);
  }
}

class B extends A {
  int k;

  void mostrark() {
    S.o.p("k: " + k);
  }
  void sum() {
    S.o.p("i+j+k: " + (i+j+k));
  }
}
```

. . . .

```
}
```

```
class Principal {
  public static void main(String args[]) {
    A superOb = new A();
    B subOb = new B();
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("superOb: ");
    superOb.mostrarij();
    System.out.println();

    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("subOb: ");
    subOb.mostrarij();
    subOb.mostrark();
    System.out.println();

    System.out.println("suma en subOb:");
    subOb.sum();
  }
}
```

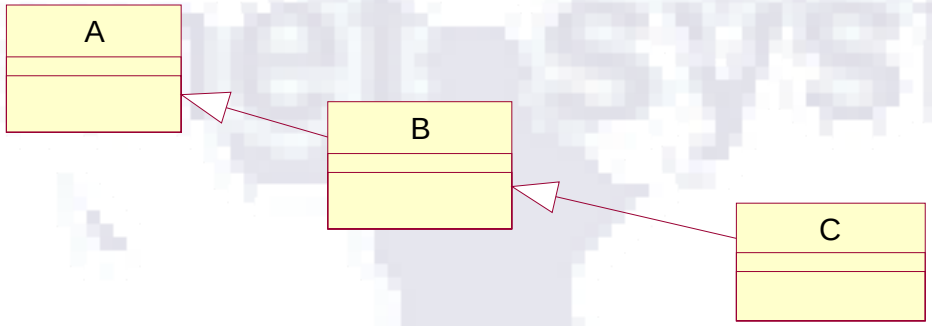

internet systems

```
class A {
  int i;
}

class B extends A {
  int i;

  B(int a, int b) {

    i = a;
    super.i = a;
    i = b;
}

  void mostrar() {
    System.out.println("i de la superclase: " + super.i);
    System.out.println("i de la subclase: " + i);
  }
}


class Principal {
  public static void main(String args[]) {
    B subOb = new B(1, 2);
    subOb.mostrar();
  }
}
```

¿a que i hace referencia?

Herencia

- Jerarquía multinivel:
  - La herencia se puede definir en varios niveles: Si B hereda de A y C hereda de B => C hereda de A y de B



Herencia. Orden de ejecución de los constructores

```
class A {
  A() {
    System.out.println(" constructor de A");
  }
}
class B extends A {
  B() {
    System.out.println(" constructor de B");
  }
}
class C extends B {
  C() {
    System.out.println("constructor de C");
  }
}
class Principal {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

La salida que genera es:

```
constructor de A
constructor de B
constructor de C
```


internet systems

Herencia. Sobreescritura de métodos

```
class A {
  int i, j;
  A(int a, int b) {
    i = a;j = b;
  }
  void mostrar() {
    System.out.println("i y j: " + i + " " + j);
  }
}
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void mostrar() {
    System.out.println("k: " + k);
}}
```

```
class Principal{
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.mostrar(); }
}
```

Se ejecuta el mostrar de B que sobreescribe al de A.

Para llamar al mostrar de A se usa la instrucción:

```
super.mostrar();
```

Herencia. Polimorfismo

```
class Figuras {
  double dim1;
  double dim2;
  Figuras(double a, double b) {
    dim1 = a;
    dim2 = b;
  }
  double area() {
    System.out.println("Area no
    definida.");
    return 0;
  }
}
class Rectangulo extends Figuras {
  Rectangulo(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Area del
    rectángulo.");
    return dim1 * dim2;
 }
}
```

```
class Triangulo extends Figuras {
  Triangulo(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Area del triángulo.");
    return dim1 * dim2 / 2;
  }
}

class Principal{
  public static void main(String args[]) {
    Figuras f = new Figuras(10, 10);
    Rectangulo r = new Rectangulo(9, 5);
    Triangulo t = new Triangulo(10, 8);
    Figuras fig;
    fig = r;
    System.out.println("Area :" + fig.area());
    fig = t;
    System.out.println("Area :" + fig.area());
    fig = f;
    System.out.println("Area :" + fig.area());
  }
}
```

Herencia. Clases abstractas

```
abstract class A {
  abstract void llamada() ;

  void noAbstracto() {
    System.out.println("Este es un método no abstracto.");
  }
}

class B extends A {
  void llamada() {
    System.out.println("Implementación del método abstracto");
  }
}

class Principal {
  public static void main(String args[]) {
    B b = new B();
    b.llamada() ;
    b.noAbstracto();
  }
}
```


internet systems

PAQUETES

Paquetes

- Son contenedores de clases que se utilizan para mantener el espacio de nombres de clases dividido en compartimentos.
- Definición:
  - Se pone antes de definir la clase, e indica el paquete al que pertenece la clase. Si ya está creado la añade, si no lo crea.

```
package p1;
package p1.p2;
```

- Acceso:

```
import p1.*;
import p1.p2.*;
import p1.Clase;
```

Paquetes

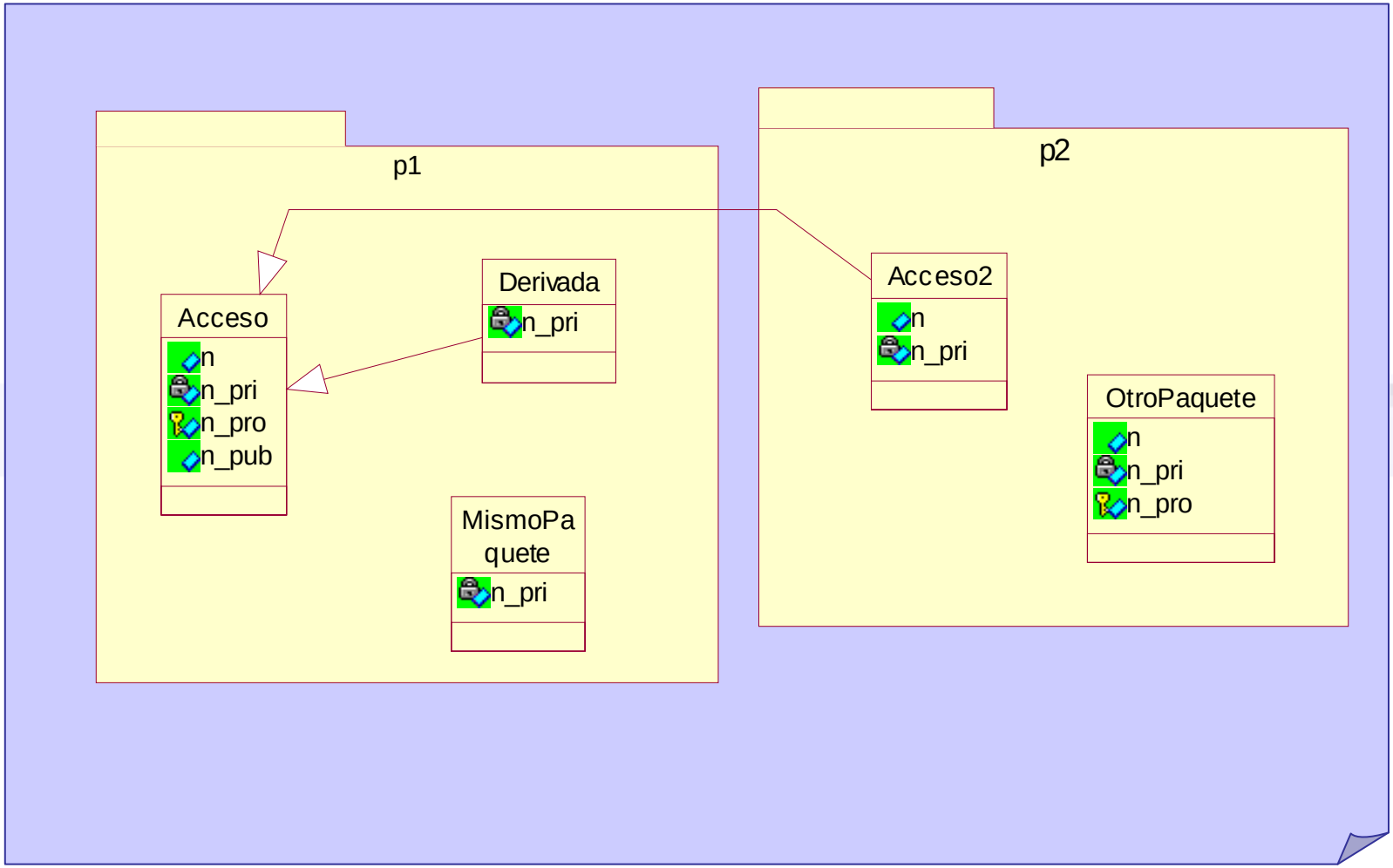
- Los paquetes tienen importancia para la accesibilidad a una clase.
- El identificador de visibilidad protected permite mostrar elementos de una clase fuera del paquete actual pero sólo para las subclases directas de su clase.

Nivel de acceso	private	Sin modificador	protected	public
Misma clase	Si	Si	Si	Si
Subclase del mismo paquete	No	Si	Si	Si
No subclase del mismo paquete	No	Si	Si	Si
Subclase de diferente paquete	No	No	Si	Si
No subclase de diferente paquete	No	No	No	Si


internet systems


internet systems

Paquetes. Ejemplo



INTERFACES

Interfaces

- Un interface son estructuras de código similares a las clases que contienen la definición de un conjunto de funcionalidades, pero no su implementación.
- Las clases que implementen un interface están obligadas a implementar todos los métodos de interface.
- Un interface no puede contener métodos implementados.
- Un interface no de puede instanciar.

```
Interface Llamada{
    void llamar(int param);
}
```

```
class Cliente implements Llamada{
    public void llamar(int p){
                S.o.p("Llamando a llamar"+p);
    }
}
```

Interfaces

- Una clase puede implementar varios interfaces

  class Cliente implements Llamada, Llamada2{....}

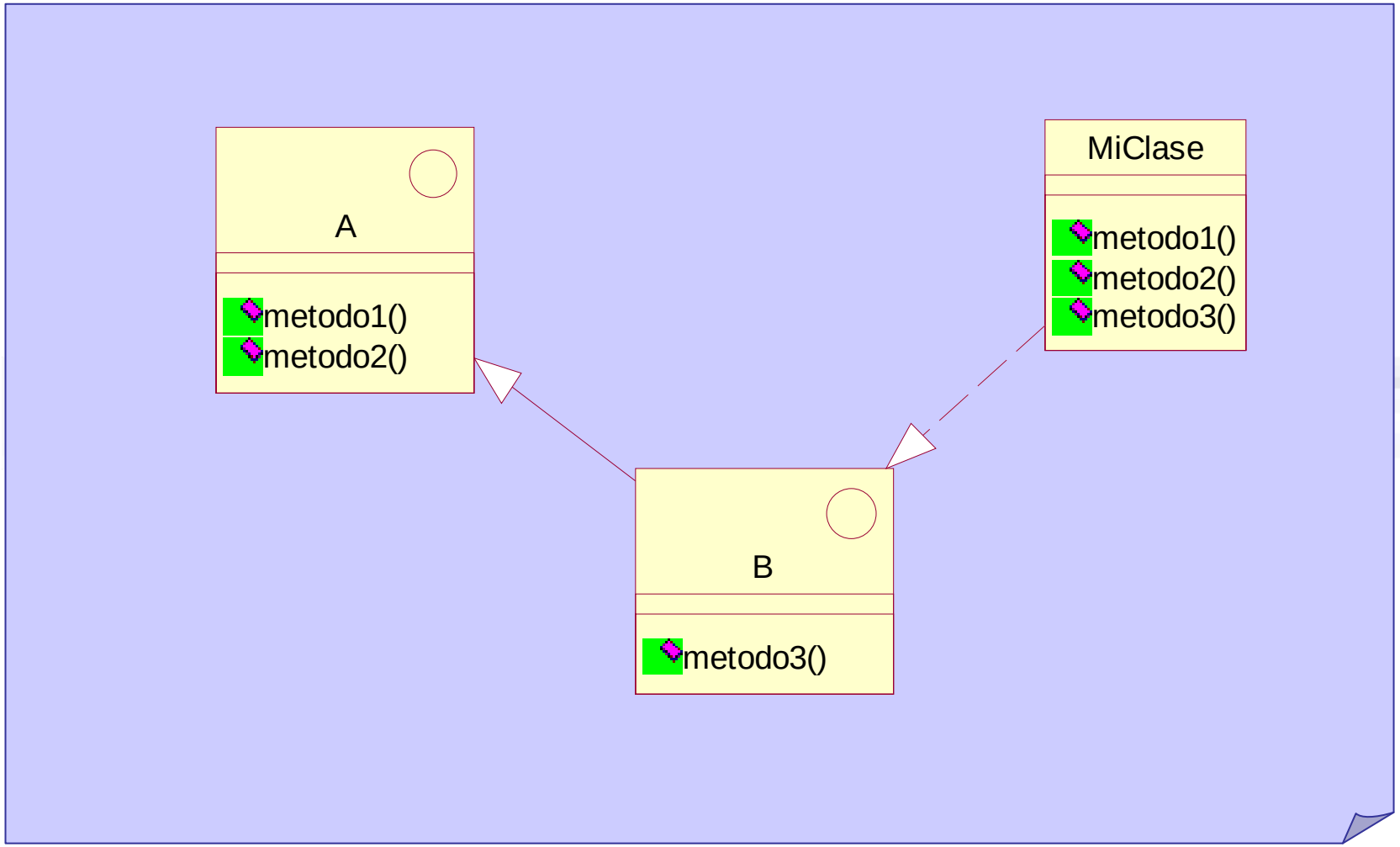
- Los métodos de las clases que implementan un interface deben ser públicos.
- Si se hereda de m
  método será únic
- Una clase que imp
  propios.
- Se pueden declarar objetos a través de una variable de referencia a un interface.*
- Las variables de un interface son constantes dentro de la clase que implementa al interface.
- Las interfaces se pueden heredar.

```
class Principal{
    public static void main(String args[]){
        llamada c=new Cliente( );
        c.llamar(5);
    }
}
```

*


internet systems

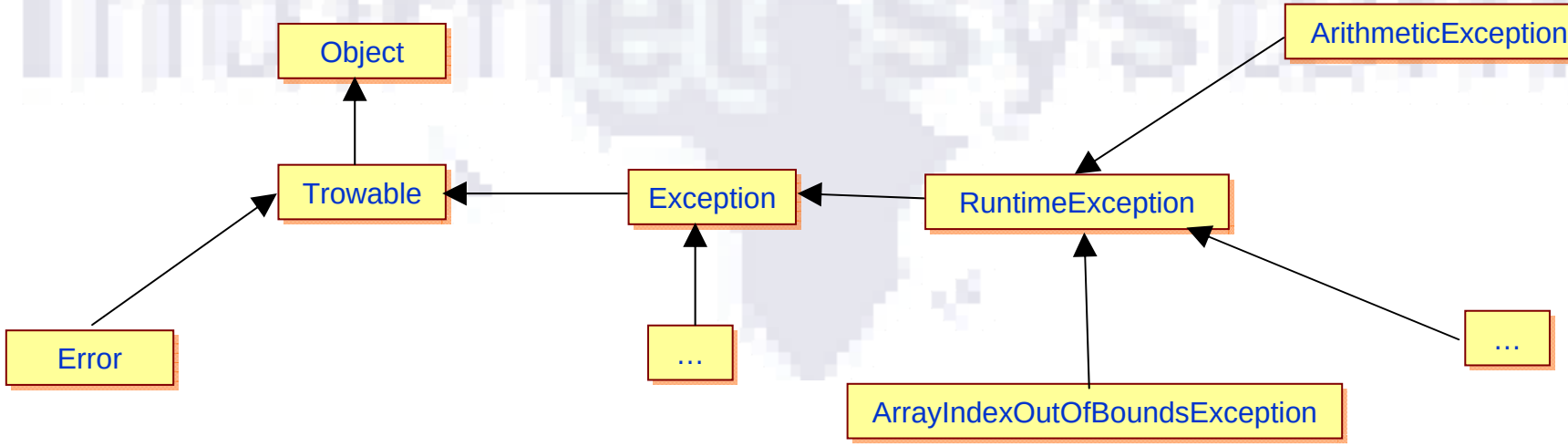
Interfaces


A
metodo1()
metodo2()
MiClase
metodo1()
metodo2()
metodo3()
B
metodo3()

EXCEPCIONES

Excepciones

- Una excepción es un error que se produce en tiempo de ejecución.
- Una excepción es un objeto. Todos estos objetos son subclases de la clase **Throwable,** y se ajustan a una estructura jerárquica.



Excepciones

- El control de de código de error se realiza con los bloques **try – catch**
  - Las zonas de código susceptibles de generar errores deben estar acotadas en un bloque **try**
  - Cuando se produce un error en el bloque **try** se genera un objeto del tipo de la excepción que se ha producido
  - Esta excepción se puede *recoger* en un bloque **catch**
- El ámbito del bloque **catch** está restringido a las sentencias del bloque **try** que le precede inmediatamente.

Excepciones

```
class Excepcion {
  public static void main(String args[]) {
    int a,d;
    d = 0;
    a = 42 / d;
    System.out.println("Mensaje");
  }
}
```

¡¡ERROR!!

SOLUCION

```
class Excepcion {
  public static void main(String args[]) {
    int d, a;

    try {
      d = 0;
      a = 42 / d;
      System.out.println("Mensaje.");
    } catch (ArithmeticException e) {
        System.out.println("Division por cero.");
    }
    System.out.println("Despues de catch.");
  }
}
```


internet systems

Excepciones

- Si no se conoce el tipo de excepción que se puede producir, es aconsejable utilizar el objeto genérico Exception.
- Existen métodos para obtener información sobre el objeto Exception (e). Estos y otros métodos pertenecen a la clase Throwable.

toString() Devuelve una descripción de la excepción

printStackTrace() Devuelve una traza

```
e.printStackTrace();
```


internet systems

Excepciones. Try anidados

```
class Anidados {
  public static void main(String args[]) {
    try {
      int a = args.length;
      int b = 42 / a;
      System.out.println("a = " + a);
      try {
        if(a==1) a = a/(a-a);
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99;
        }
      }
      catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Desbordamiento: " + e);
      }
    }
      catch(ArithmeticException e) {
      System.out.println("Division por 0: " + e);
    }
  }
}
```

- Si **a=0**, se genera una división por 0 y lo captura el catch externo
- Si **a=1**, genera división por 0. Como no puede capturarlo el catch interno, lo captura el externo.
- Si **a=2**, se produce un desbordamiento que lo captura el catch interno

Excepciones

```
class Demo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    }
    catch(NullPointerException e) {
      System.out.println("captura dentro de demo.");
      throw e;
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    }
    catch(NullPointerException e) {
      System.out.println("Nueva captura: " + e);
    }
  }
}
```

❶ ❷ ❸


internet systems

PROGRAMACIÓN MULTIHILO

MULTIHILO. INTRODUCCIÓN

- Programación multihilo es un estilo de ejecución en el que se conmuta entre distintas partes del código de un mismo programa durante la ejecución.
- Cada una de estas partes individuales se llama Thread (hilos, hebras o contextos de ejecución)
- La JVM los ejecuta de forma simultanea en una maquina multiprocesador y de forma conmutada en maquina de un solo procesador ,esto es, la ejecución de los threads se realizará asignando un determinado quantum o espacio de tiempo a cada uno de los threads.
- Permiten ejecutar dos tareas al mismo tiempo por el mismo
- Los hilos son procesos ligeros
- La conmutación entre hilos es mucho mas rápida que la conmutación entre procesos

MULTIHILO. INTRODUCCIÓN

- En la actualidad casi todos los sistemas operativos proporcionan servicios para la programación multihilo.
- Las JVM de todas las plataforma proporcionan hilos de ejecución aunque el sistema operativo subyacente no los proporcione, en este caso la JVM simula ese comportamiento.
- A partir de la versión 1.3 de Java, la JVM para entornos Windows proporciona servicios de hilos nativos (esto es proporcionados por la API Win32 y no simulados por el sistema). A parte de estos hilos nativos la JVM en todo caso sigue proporcionando hilos simulados por ella misma (no nativos) conocidos como hilos verdes (**green threads**).

MULTIHILO. INTRODUCCIÓN

- Hay dos formas de crear hilos en Java. Un hilo es una clase que desciende de la clase java.lang.Thread o bien que extiende a la interfaz Runnable (util en los casos de que la clase ya forme parte de una jerarquía de clases).
- La forma de construcción del hilo (derivación o implementación) es independiente de su utilización.
- En Java un hilo puede encontrarse en cualquiera de los siguiente estados:
  - *nace:* El hilo se declarado pero todavía no se ha dado la orden de puesta en ejecución. (método start()).
  - *listo:* El hilo esta preparado para entrar en ejecución pero el planificador aun no ha decidido su puesta en marcha
  - *ejecutándose:* El hilo esta ejecutándose en la CPU.

MULTIHILO. INTRODUCCIÓN

- *dormido:* El hilo se ha detenido durante un instante de tiempo definido mediante la utilización del método sleep().
- *bloqueado:* El hilo está pendiente de una operación de I/O y no volverá al estado listo hasta que esta termine.
- *suspendido:* El hilo ha detenido temporalmente su ejecución mediante la utilización del método suspend() y no la reanudará hasta que se llame a resume(). *Antiguo, 1.2.No debe usarse.*
- *esperando:* El hilo ha detenido su ejecución debido a una condición externa o interna mediante la llamada a wait() y no reanudará esta hasta la llamada al método notify() o notifyAll().
- *muerto:* El hilo ha terminado su ejecución bien porque termino de realizar su trabajo o bien porque se llamó al método stop(). *Antiguo 1.2.No debe usarse.*


internet systems

CREACIÓN DE HILOS

- Un hilo puede crearse extendiendo a la clase **Thread** o implementando a la interfaz **Runnable**. En ambos casos la clase debe proporcionar una definición del método run() que contiene el código a ejecutarse una vez que el hilo entre en estado de ejecución

```
public class HiloThread extends Thread{
   public void run(){
         while(true){
            System.out.println("Hola mundo,soy el hilo HiloTread");
         }
   }
}
```

```
public class HiloRunnable implements Runnable{
   public void run(){
      while(true){
         System.out.println("Hola mundo,soy el hilo HiloRunnable");
      }
   }
}
```

¿Como los pongo en marcha y como interactúo con ellos?.

Ejecución de hilos

```
public class Test{
   public static void main(String[] args){
        //Creamos un hilo del primer tipo
        //y lo ponemos en marcha

        HiloThread ht = new HiloTread();
        ht.start();
        //Creamos un hilo del segundo tipo
        //y lo ponemos en marcha

        Thread hr = new Thread(new HiloRunnable());
        hr.start();
   }
}
```

Ambos hilos se construyen de manera distinta: mientras que el que desciende de Thread se crea como un objeto más, el que implementa Runnable,se crea y se pasa al constructor de la clase genérica Thread.

Constructores de Thread

- La clase Thread tiene los siguientes constructores:
  - Thread()
  - Thread(Runnable target)
  - Thread(Runnable target, String name)
  - Thread(String name)
  - Thread(ThreadGroup group, Runnable target)
  - Thread(ThreadGroup group, Runnable target, String name)
  - Thread(ThreadGroup group, String name)

String name se usa para dar un nombre el thread de manera que luego pueda referenciarse de forma externa mediante los metodos String **getName()** y void **setName(String)**

String name se usa para dar un nombre el thread de manera que luego pueda referenciarse de forma externa mediante los metodos String **getName()** y void **setName(String)**

Algunos métodos de Thread

- **isAlive()** determina si un hilo está vivo o si se ha terminado de ejecutar.
  - final boolean isAlive() throws InterruptedException{ }
    - Devuelve true si el hilo que llama al método está vivo.
- **join()** espera hasta que termine de ejecutarse el hilo que llama al método.
  - final void join() throws InterruptedException{ }

Algunos métodos de Thread

- suspend() suspende la ejecución de un hilo hasta que este se vuelva a reanudar (con resume)
  - final void suspend();
- resume() reanuda un hilo que previamente ha sido suspendido.
  - final void resume();

Ejemplo

```
public class NuevoHilo implements Runnable {
      String name;
      Thread t;
      public NuevoHilo(String nombre) {
            name = nombre;
            t = new Thread(this, nombre);
            System.out.println("Nuevo hilo: " + t);
            t.start();
      }
      public void run() {
            try {
                for (int i = 5; i > 0; i--) {
                    System.out.println(name + ":" + i);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                System.out.println("interrupción en el hilo: " + name);
      }}
}
```

Ejemplo

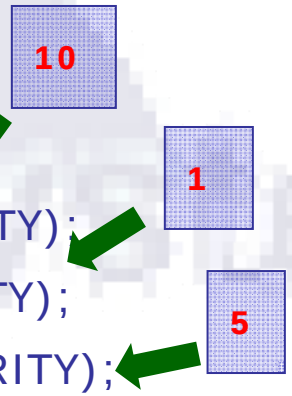
```
public class Principal {
      public static void main(String[] args) {
            NuevoHilo ob1=new NuevoHilo("Uno");
            NuevoHilo ob2=new NuevoHilo("Dos");
            NuevoHilo ob3=new NuevoHilo("Tres");
            System.out.println("¿El hilo Uno esta vivo?: "+ob1.t.isAlive());
            System.out.println("¿El hilo Dos esta vivo?: "+ob2.t.isAlive());
            System.out.println("¿El hilo Tres esta vivo?: "+ob3.t.isAlive());
      try {
            System.out.println("esperando a que terminen los otros");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
      } catch (Exception e) {
            System.out.println("Interrupcion hilo principal");}
      System.out.println("¿El hilo Uno esta vivo?: "+ob1.t.isAlive());
      System.out.println("¿El hilo Dos esta vivo?: "+ob2.t.isAlive());
      System.out.println("¿El hilo Tres esta vivo?: "+ob3.t.isAlive());
      System.out.println("Sale del principal");
      }
}
```

Prioridades de los hilos

- Cada hilo tiene asociada una prioridad representada por un número entero.
- La prioridad determina cuando se debe ejecutar un hilo frente a otro, y no la velocidad de ejecución.
- Cuando un hilo deja de ejecutarse para ejecutarse otro, se produce un **cambio de contexto** => al detenerse un hilo, se analizan los siguientes hilos. El que tiene prioridad más alta, se ejecuta.
- Las prioridades de un hilo pueden oscilar entre 1 y 10. La prioridad por defecto es 5.

Prioridades de los hilos

- **setPriority()** asigna prioridad a un hilo
  - final void setPriority(int nivel){}
    - Hilo.setPriority(2);
    - Hilo.serPriority(Thread.MAX_PRIORITY);
    - Hilo.serPriority(Thread.MIN_PRIORITY);
    - Hilo.serPriority(Thread.NORM_PRIORITY);



- **getPriority()** devuelve la prioridad actual de un hilo
  - final int getPriority(){}

¡¡OJO CON LAS PRIORIDADES!!

- En el siguiente ejemplo se lanzan dos hilos que cuentan el número de ciclos que realiza un bucle durante un segundo. Los dos hilos tienen distintas prioridades. Al final se muestra en pantalla el resultado. ¿Cuál será?.
- El resultado va a ser diferente dependiendo de la plataforma en que se ejecute.

```
public class Contador implements Runnable{
   int click=0;
   Thread t;
   private boolean running=true;
          public Contador(int p){
          t=new Thread(this);
          t.setPriority(p);
   }
   public void run(){
          while (running){
               click++;
          }
   }
   public void stop(){
          running=false;
   }
   public void start(){
          t.start();
   }
}
```

¡¡OJO CON LAS PRIORIDADES!!

```
public class Principal {
        public static void main(String[] args) {
                Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
                Contador hi=new Contador(Thread.NORM_PRIORITY+2);
                Contador lo=new Contador(Thread.NORM_PRIORITY-2);
                lo.start();hi.start();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        System.out.println("interrupcion");}
                lo.stop();hi.stop();
                try {
                        hi.t.join();
                        lo.t.join();
                } catch (InterruptedException e1) {
                        System.out.println("captura");}
                System.out.println("hilo con prioridad baja: "+lo.click);
                System.out.println("hilo con prioridad alta: "+hi.click);
        }
}
```

```
hilo con prioridad baja: 0
hilo con prioridad alta: 138989761
```

INTRODUCCIÓN A LAS APLICACIONES WEB

INTRODUCCIÓN A INTERNET

- **Red**: Una red es una agrupación de computadoras mediante la cual se posibilita el intercambio de información de un modo eficiente y transparente.
  - LAN: (Local ares network) Red de área local. Se reduce a los límites de un edificio.
  - MAN: (Metropolitan area network) Metropolitana. Varios edificios dentro de una misma área urbana.
  - WAN: (Wide area network) Une centros dispersos en áreas geográficas muy amplias. Suele estar integrada por varias redes mas pequeñas. Internet se puede considerar la WAN mas grande del mundo.
- **W.W.W**.: (World Wide Web) Denominación que se da a Internet – La gran telaraña mundial – Internet es una gran red de redes.

INTRODUCCIÓN A INTERNET

- TCP/IP
  - Para que los ordenadores que componen una red puedan comunicarse es necesario que todos hablen en el mismo "idioma". A esto se denomina un **protocolo**.
  - Un **protocolo** es un conjunto reglas e instrucciones comunes a todos los ordenadores de la misma red.
  - Internet utiliza el protocolo TCP/IP.
  - TCP (Transport Control Protocol)
  - IP (Internet protocol)
  - Es un sistema de comunicaciones basado en ensamblado y desensamblado de paquetes.

INTRODUCCIÓN A INTERNET

- Servicios
  - Sobre la base de comunicación del protocolo TCP/IP existen otros protocolos más específicos que permiten realizar tareas particulares:
  - **SMTP**: correo electrónico.
  - **TELNET**: conexiones y ejecución de comandos con máquinas remotas.
  - **FTP**: transmisión de ficheros
  - **NNTP**: acceso a foros de discusión y news
  - **HTTP**: conexión a servidores web

INTRODUCCIÓN A INTERNET

- HTTP
  - Permite solicitar datos a un servidor remoto para visualizarlos localmente (en un navegador) utilizando TCP/IP.
  - El protocolo **HTTP** no es permanente y es necesario mantener la conexión desde el servidor mediante el estado de la sesion.
  - Normalmente, en una aplicación cliente-servidor, el cliente va a hacer peticiones a páginas web. Estas peticiones se suelen hacer por medio de una **URL**.

INTRODUCCIÓN A INTERNET

- URL
  - Una URL es el nombre completo de un recurso.
  - Todo ordenador que use internet tiene su propia dirección electrónica (**IP**).
  - Una IP está compuesta por una secuencia de cuatro números de un máximo de tres cifras cada uno y separados por puntos:

    194.224.71.2
  - Para que resulte más fácil recordar los identificadores no se usa la IP, sino un sistema de nomenclatura paralelo denominado **DNS**.
  - Una URL hace referencia a un recurso (archivo, directorio de correo, ...) ubicado en una máquina remota identificada mediante su DNS.


internet systems

INTRODUCCIÓN A INTERNET

- URL's para HTTP
  - La sintaxis completa de una URL para HTTP es:

http://servidor:puerto/ruta del fichero

  - Ejemplo:

http://localhost:8100/euro/conversor.html

  - En este caso estamos haciendo una petición a una aplicación ubicada en un servidor web cuyo puerto de escucha es el 8100.


internet systems

ARQUITECTURA CLIENTE-SERVIDOR

- Tendencias actuales en aplicaciones C-S
  - Añadir inteligencia al cliente
    - JavaScript
    - Applets
  - Añadir inteligencia al servidor
    - Servlets
    - Jsp

APPLETS

APPLETS

- Un applet es un pequeño programa que se transmite junto con una aplicación web y que se instalan y ejecutan en el cliente.
- Un applet es una clase compilada que se envía al cliente junto con el documento HTML solicitado en una petición.
- El applet se ejecuta (interpreta) en el cliente (navegador) por medio de su JVM.
- Permiten dotar al cliente de funcionalidad para interactuar con la aplicación.
- Las aplicaciones web que contienen applets son lentas.

APPLETS

- Una applet debe heredar de la clase Applet.
- La clase Applet está contenida en el paquete

  java.applet
- java.applet define tres interfaces

  AppletContext

  AppletStub

  AudioClip
- Una clase que define un applet debe importar el paquete

  import java.applet.*;

APPLETS

- Para permitir que una applet tenga una interface gráfica donde visualizarse debe importar el paquete.

```
import java.awt.*;
```

- La ejecución de un applet **no** comienza con main().
- La salida de información de un applet **no** se realiza con el método System.out.print().
- El applet es una clase que hay que compilar. Una vez generado el archivo XXX.class, hay que hacer una referencia a este desde un documento HTML con la etiqueta

```
<APPLET>
```

APPLETS

- Referencia a un applet:

```
<HTML>
  <HEAD>
  </HEAD>
  <BODY>
    Esto es una referencia a un applet
    <APPLET CODE="ejemploapplet.Miapplet" WIDTH=200
    HEIGHT=60>
    </APPLET>
  </BODY>
</HTML>
```


internet systems

ALGUNOS METODOS GRAFICOS

public void drawString(String mensaje, int x, int y)

- Permite mostrar cadenas de caracteres en un applet
- Pertenece a Graphics
- mensaje: La cadena que se va a mostrar
- x,y : coordenadas de la posición donde se va a mostrar

public void setBackground(Color nuevocolor)

- Asigna color de fondo a la ventana del applet
- nuevocolor: Objeto de tipo Color que define los colores mediante constantes de la clase
  - Color.black, Color.blue, Color.orange, ...

ALGUNOS METODOS GRAFICOS

public void setForeground(Color nuevocolor)

- Color de primer plano.

public Color getBackground()

- Devuelve el color de fondo actual.

public Color getForeground()

- Devuelve el color de primer plano actual.

EJEMPLO

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
 Esto es una referencia a un applet <br>
 <APPLET CODE="ejemploapplet.Miapplet" WIDTH=500 HEIGHT=300>
 </APPLET>
 </BODY>
</HTML>
```

```
package ejemploapplet;

import java.awt.*;
import java.applet.*;
public class  Miapplet extends Applet{
          String msg;

          public void init() {
                    setBackground(Color.orange);
                    setForeground(Color.blue);
                    msg="dentro de init() --  ";
          }

          public void paint(Graphics g) {
                    msg+="Dentro de paint()  --  ";
                    g.drawString(msg,10,30);
          }

          public void start() {
                    msg+="dentro de start() --  ";
          }
}
```

- crear la clase Miapplet
- compilar y exportarla a una carpeta llamada **practicaapplet**

- Crear el documento HTML y guardarlo con el nombre **principal.htm** en la carpeta **practicaaplet.**
- **Ejecutar el documento principal.htm**

RESULTADO


Atrás
Búsqueda
Favoritos
Multimedia
Dirección
C:\curro\Java\codigo ejemplos\curso\ejemploapplet\principal.htm
Search Web
...attempting to retrieve buttons from Yahoo!...
Esto es una referencia a un applet
dentro de init() -- dentro de start() -- Dentro de paint() --

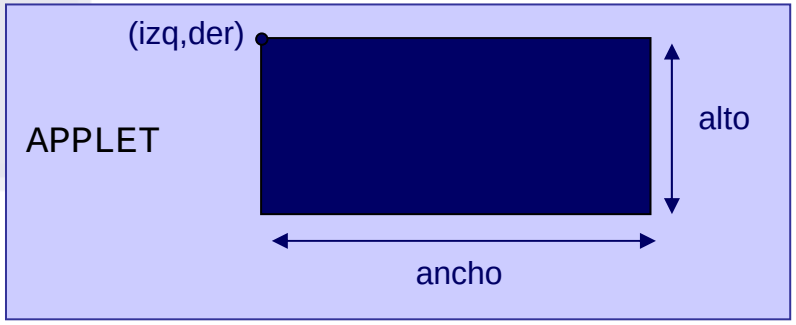
METODOS

- El método repaint

  public void repaint()

  public void repaint(int izq, int sup, int ancho, int alto)

  - Realiza una llamada al método paint() y así poder actualizar el contenido del applet.
  - izq, sup: coordenadas de origen de la zona que se quiere actualizar
  - ancho, alto: dimensiones de esa zona.



EJERCICIO

- Hacer un applet que sea capaz de mostrar un texto desplazándose horizontalmente (banner)
- Necesitas saber:
  - String.charAt(int n) → Devuelve el carácter que esta en la posición n de un String
  - String.substring(int n, int n2) → Devuelve un String con n2 caracteres cogidos a partir de la posición n del String que llama al método.
  - Para producir un retardo en un bucle:
    - Thread.sleep(250)
    - Para poder utilizar la clase Thread la clase debe implementar la interface Runnable.

SOLUCIÓN

```
package banner;
import java.applet.*;
import java.awt.*;
public class Banner extends Applet
          implements Runnable {
   String msg=" - un cartel en movimiento - ";
   Thread t=null;
   public void destroy(){
          if(t !=null){
             t.stop();
             t=null;
          }
   }
   public void init(){
          setBackground(Color.orange);
          setForeground(Color.blue);
          t=new Thread(this);
          t.start();
          t.suspend();
   }
```

```
   public void paint (Graphics g){
      g.drawString(msg,50,30);
   }
   public void run(){
      char ch;
      for(int i=1;i<200;i++ ){
          try{
             repaint();
             t.sleep(250);
             ch=msg.charAt(0);
             msg=msg.substring(1,msg.length());
             msg+=ch;
          }
          catch(InterruptedException e){}
      }
   }
   public void start(){
      t.resume();
   }
   public void stop(){
      t.suspend();
   }
}
```

VENTANA DE ESTADO

showStatus(String cadena)

- Sirve para mostrar un mensaje en la barra de estado

```
package appletestado;

import java.awt.*;
import java.applet.*;

public class VentanaEstado extends Applet{

  public void init() {
          setBackground(Color.cyan);
  }

public void paint(Graphics g) {
          g.drawString("Esto es un applet.", 10, 20);
          showStatus("Esto es un mensaje en la barra de estado.");
  }
}
```

GESTION DE EVENTOS

- Todos los eventos que se generan en un applet están encapsulados dentro de un objeto **Event**.
- Event pertenece a AWT.
- Event define varias variables que describen el evento:
  - x, y : posición del ratón cuando se produce el evento de ratón.
  - key: almacena la tecla pulsada.
- Event contiene métodos que procesan los eventos más habituales.
- Al sobreescribir en un applet algunos de estos métodos, se está indicando que hará el applet cuando se produzca dicho evento.

METODOS PARA EVENTOS DE RATON

- boolean mouseDown (Event objEvento, int x, int y)
  - Se ejecuta al pulsar el ratón
  - Event : Es el objeto que ha producido el evento
  - x, y : coordenadas
- boolean mouseDrag (Event objEvento, int x, int y)
  - Al mover al ratón con el botón pulsado (Arrastre)
- boolean mouseEnter (Event objEvento, int x, int y)
  - Al entrar en la ventana del applet
- boolean mouseExit (Event objEvento, int x, int y)
  - Al salir
- boolean mouseMove (Event objEvento, int x, int y)
  - Al mover el ratón sin puslar
- boolean mouseUp (Event objEvento, int x, int y)
  - Cuando se deja de pulsar el botón del ratón

EJEMPLO

C:\curro\Java\codigo ejemplos\curso\naves\naves

```
package naves;
import java.awt.*,java.applet.*;
public class Naves extends Applet {
          private Image normalUfo,ufo,bonusUfo;
          int rx=0,ry=0;
          public void init(){
                    normalUfo = getImage(getCodeBase(), "Images/Ufo1.gif");
                    bonusUfo = getImage(getCodeBase(), "Images/Ufo3.gif");
                    setBackground(Color.orange);
                    setForeground(Color.blue);}
          public boolean mouseDown (Event evento, int x, int y){
                    rx=x;
                    ry=y;
                    ufo=bonusUfo;
                    repaint();
                    return true;}
          public boolean mouseDrag (Event evento, int x, int y){
                    showStatus("Arrastrando. Posición del ratón: "+x+", "+y);
                    rx=x;
                    ry=y;
                    ufo=bonusUfo;
                    repaint();
                    return true;}
```

EJEMPLO continuación

```
        public boolean mouseEnter (Event evento, int x, int y){
                showStatus("Entrando. Posición del ratón: "+x+", "+y);
                rx=x;ry=y;ufo=normalUfo;
                repaint();
                return true;}
        public boolean mouseExit (Event evento, int x, int y){
                showStatus("Saliendo. Posición del ratón: "+x+", "+y);
                rx=x;ry=y;ufo=normalUfo;
                repaint();
                return true;}
        public boolean mouseMove (Event evento, int x, int y){
                showStatus("Posición del ratón: "+x+", "+y);
                rx=x;ry=y;ufo=normalUfo;
                repaint();
                return true;}
        public boolean mouseUp (Event evento, int x, int y){
                rx=x;ry=y;ufo=normalUfo;
                repaint();
                return true;}
        public void paint (Graphics g){
                g.drawImage( ufo, rx-15, ry-30, this);}
}
```