

UT4

Interacción de objetos. Estructura de control iterativa.

Módulo – Programación (1º)

Ciclos – Desarrollo de Aplicaciones Multiplataforma | Desarrollo de Aplicaciones Web

CI María Ana Sanz

Contenidos

- Abstracción y modularización
- Las clases como tipos
- Diagrama de clases y diagrama de objetos
- Tipos primitivos y tipos referencia
- Creación de nuevos objetos
- Múltiples constructores
- Llamadas a métodos
 - Llamadas internas
 - Llamadas externas
- `public / private`
- `this / null`
- Cómo usar Java fuera de BlueJ
 - el método *main()*
- Estructura de control iterativa
 - `while`
 - `for`
 - `do .. while`
- clases Scanner y Random
- Miembros de clase (*static*)
- Métodos recursivos

Objetivos

- Un programa orientado a objetos
 - conjunto de objetos que cooperan entre sí para realizar una tarea
 - los objetos se envían mensajes entre ellos para conseguir el objetivo común
- Aprenderemos a
 - construir pequeñas aplicaciones formadas por un conjunto de objetos de diferentes clases que
 - cooperan e interactúan entre ellos
- Utilizaremos la
 - estructura iterativa para realizar tareas repetitivas
- Cómo ejecutar Java fuera de Bluej?
 - Implementando el método *main()*

Abstracción y modularización

- Leer punto 4.1
 - Recordemos las características de la POO
 - **Abstracción** – ignorar detalles, capturar lo esencial
 - características relativas al observador.
 - enfatiza detalles con significado para el usuario, suprimiendo aquellos detalles que, por el momento, son irrelevantes o distraen de lo esencial.
 - Ejemplo: mapas
 - **Modularización**
 - división del problema en subproblemas
 - clases y métodos
- “Divide y vencerás”
- manejar la complejidad de un problema

La abstracción permite ver el bosque
y la modularización los árboles que hacen el
bosque

Abstracción y modularización

- **Encapsulación**
 - agrupar estado y comportamiento en la clase (hacer el estado no visible además)
 - separar la interface de una abstracción y su implementación
- **Jerarquía**
 - ordenación de las abstracciones

Las clases como tipos. Proyecto Reloj digital.

- Tipos primitivos
- Tipos referencia: clases. (String, Estudiante, Hora, VisorReloj,)
- **Reloj digital** – reloj de 24 horas

11:03 ¿Un visor de cuatro dígitos?

11 03 ¿O dos visores de dos dígitos?

horas
de 0 a 23

minutos
de 0 a 59

- un panel para horas, otro panel para minutos
- los dos paneles similares características (valor de 0 a límite)
- más reutilizable

Proyecto Reloj digital (cont...)

- Analicemos el código del proyecto. Qué hay nuevo?
 - clases como tipos
 - new()
 - 2 constructores
 - llamadas internas a métodos
 - llamada externas a métodos

Proyecto Reloj digital (cont...)

```
public class VisorNumero
{
    private int limite;
    private int valor;

    // Constructor
    // Métodos
}
```

```
public class VisorReloj
{
    private VisorNumero horas;
    private VisorNumero minutos;
}
```



el reloj como combinación de dos objetos VisorNumero

VisorNumero

```
public class VisorNumero
{
    private int limite;
    private int valor;

    public VisorNumero(int limiteMaximo)
    {
        limite = limiteMaximo;
        valor = 0;
    }

    public int getValor()
    {
        return valor;
    }
}
```

VisorNumero

```
public String getValorVisor()
```

```
{
```

```
    if (valor < 10)
```

```
    {
```

```
        return "0" + valor;
```

```
    }
```

```
    else
```

```
    {
```

```
        return "" + valor ;
```

```
    }
```

```
}
```

```
public void setValor(int nuevoValor)
```

```
{
```

```
    if ((nuevoValor >= 0) && (nuevoValor < limite))
```

```
    {
```

```
        valor = nuevoValor;
```

```
    }
```

```
}
```

```
public void incrementar()
```

```
{
```

```
    valor = (valor + 1) % limite;
```

```
}
```

```
}
```

con if

```
public void incrementar()
```

```
{
```

```
    valor ++;
```

```
    if (valor == limite)
```

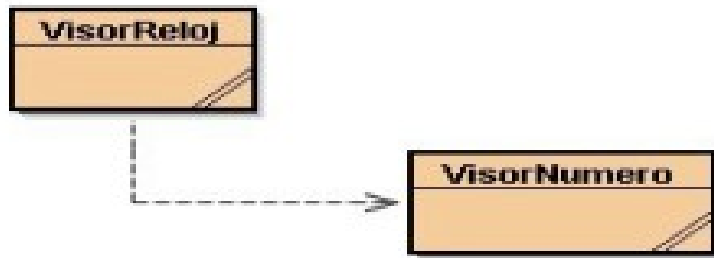
```
    {
```

```
        valor = 0;
```

```
    }
```

```
}
```

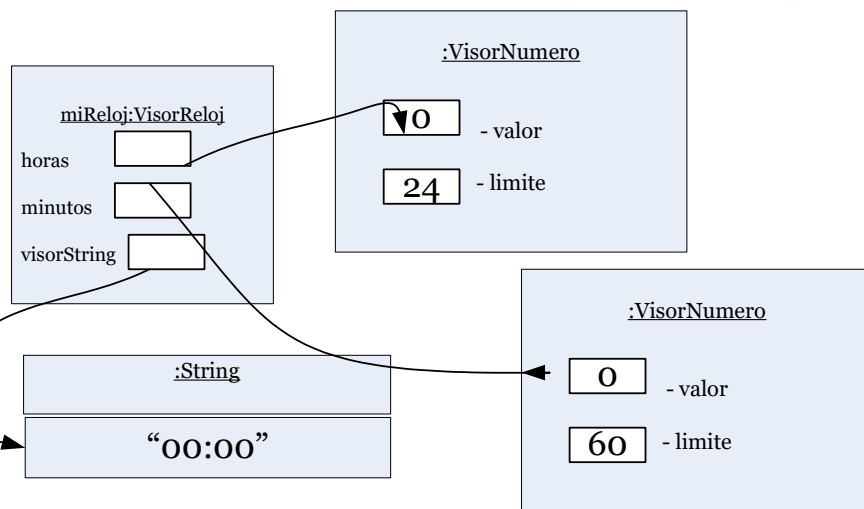
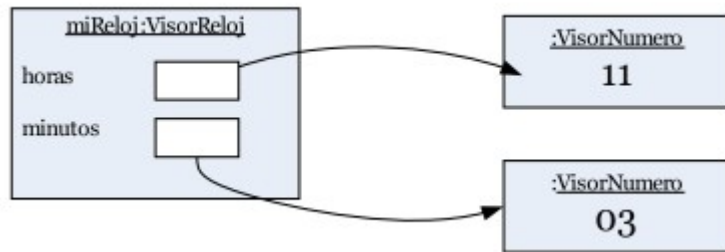
Diagrama de clases



Relación de composición en UML (un objeto VisorReloj se compone de 2 objetos VisorNumero)

- *Estructura estática* de la aplicación **MUY IMPORTANTE**
- Clases y relaciones entre ellas
- A partir de él escribimos código
- A partir del código se puede deducir el diagrama de clase (ingeniería inversa – extensión UML de BlueJ)
- En BlueJ muy simplificado

Diagrama de objetos



Al crear el objeto *miReloj*

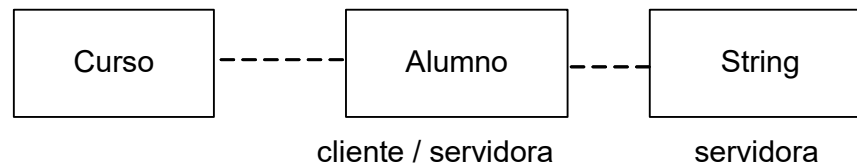
Esto es lo que ocurre al hacer botón Derecho / new VisorReloj() desde BlueJ. El nombre de la instancia es *miReloj*.

- Muestra los objetos y sus relaciones en un determinado momento de la ejecución del programa.
- Proporciona información de los objetos en tiempo de ejecución.
- Presenta una *vista dinámica* del programa.

Diagrama de objetos en detalle del VisorReloj al principio, cuando se crean los objetos (lo que interesa en el diagrama de objetos son los atributos)

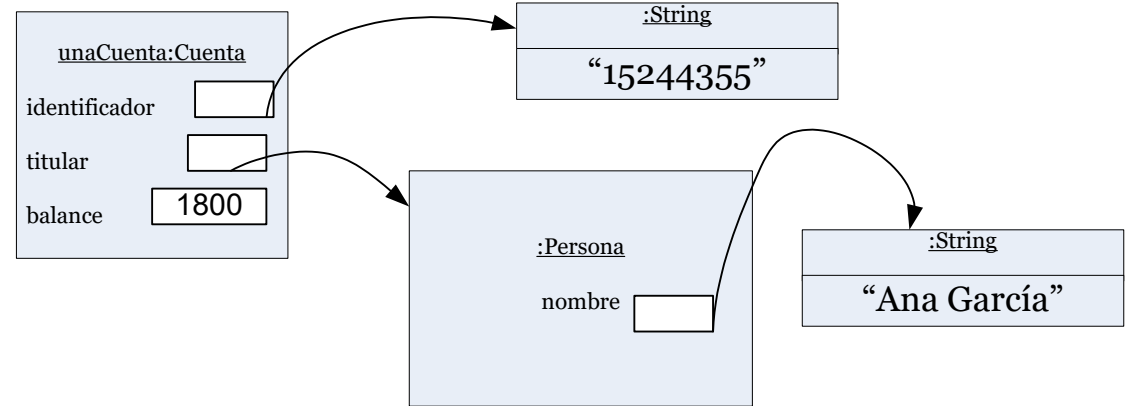
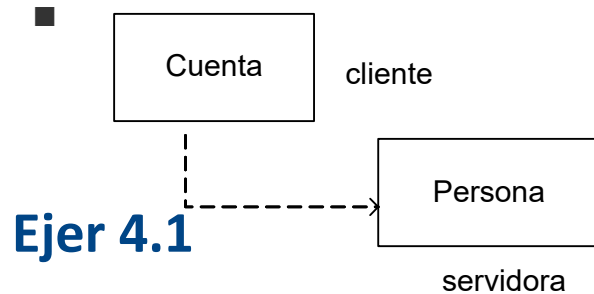
Diagrama clases y diagrama de objetos

- Relación Cliente / Servidor entre las clases
 - VisorNumero clase servidora de VisorReloj
 - VisorReloj utiliza los servicios de VisorNumero
 - VisorReloj actúa como clase cliente
- Una misma clase a veces servidora, a veces cliente

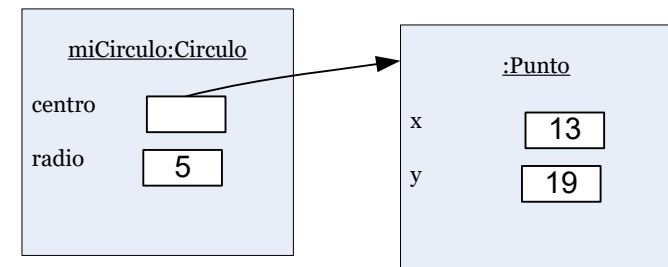
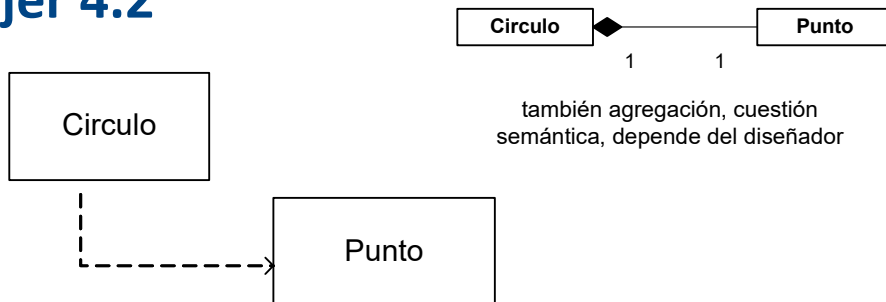


- El usuario de BlueJ actúa como cliente cuando interactúa con el Object Bench
- Ejer 4.1 y 4.2

Ejer 4.1 y 4.2



Ejer 4.2



Tipos primitivos y tipos referencia

■ Tipos **primitivos**

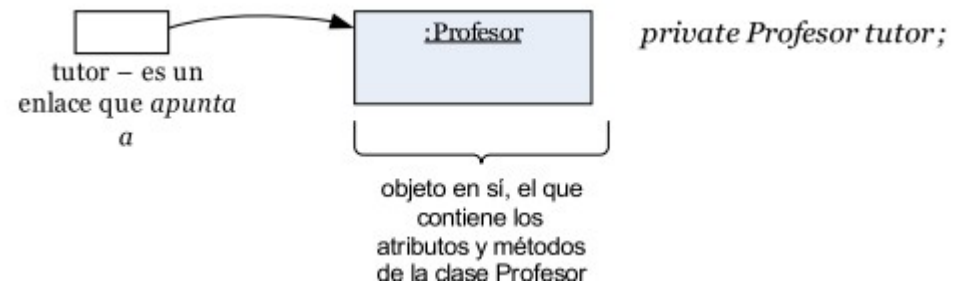
- byte, short, int, float, double, long, char, boolean
- las variables de un tipo primitivo almacenan directamente el valor

`int numero = 34;`

34

■ Tipos **referencia** o tipos **objeto**

- definidos por las clases
- Algunas clases son clases predefinidas de Java (String) otras las define el usuario
- las variables de un tipo objeto (aquellas cuyo tipo es una clase) almacenan una referencia al objeto, no el objeto en sí



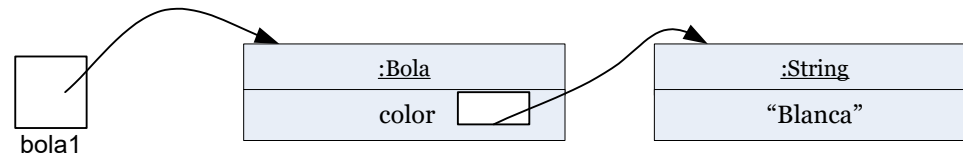
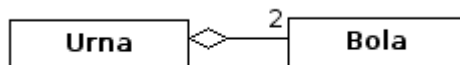
Tipos primitivos y tipos referencia

```
private VisorNumero horas;  
private Persona titular;
```

```
private Bola unaBola;  
private Profesor tutor;
```

```
public class Urna  
{  
    private Bola bola1;  
    private Bola bola2;  
}
```

```
public class Bola  
{  
    private String color;  
    public Bola(String queColor)  
    {  
        color = queColor;  
    }  
}
```



■ Ejer 4.3

Crear nuevos objetos

```
public class VisorReloj
{
    private VisorNumero horas;
    private VisorNumero minutos;
    private String visorString;    // simula el visor actual

    /**
     * Constructor de objetos VisorReloj. Este constructor crea un nuevo
     * reloj puesto en hora a las 00:00
     */

    public VisorReloj()
    {
        horas = new VisorNumero(24);
        minutos = new VisorNumero(50);
        actualizarReloj();
    }
    .....
}
```

llamada al constructor
a través de código

24 es el parámetro actual

Crear nuevos objetos

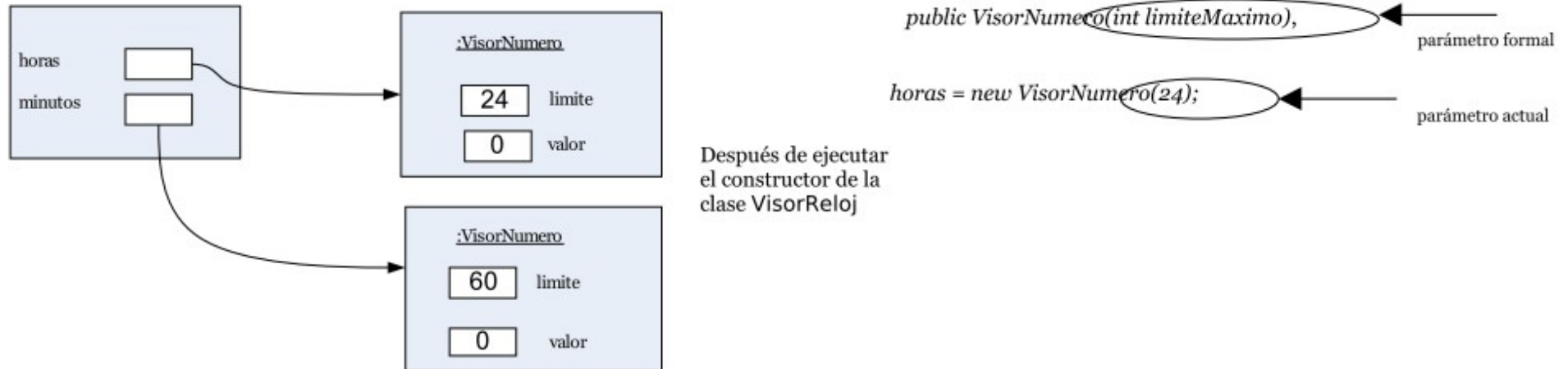
```
horas = new VisorNumero(24);
```



- **new()** es un operador – invoca al constructor para construir un objeto de la clase

- **new()**
 - crea un nuevo objeto de la clase especificada detrás de new y asigna una referencia a ese objeto a la variable especificada a la izquierda de la asignación
 - ejecuta el código del constructor (el constructor, en realidad, inicializa el objeto)

Crear nuevos objetos



- Si una clase no declara un constructor
 - se ejecuta uno sin argumentos y cuerpo vacío (constructor por defecto)
 - se proporciona solo si no hay uno definido explícitamente
- Siempre incluiremos un constructor
- Ejer 4.4 y 4.5

Múltiples constructores. Constructores sobrecargados.

- Varios constructores en la misma clase – **Constructores sobrecargados**

- todos con el mismo nombre (el de la clase)
- se diferencian por el nº y/o tipo de sus parámetros

```
public VisorReloj()  
public VisorReloj(int hora, int minuto)
```

- proporcionan alternativas a la hora de construir e inicializar los objetos
- la sobrecarga se puede aplicar no solo a los constructores sino a cualquier método
- los argumentos que se pasan en la llamada determinarán qué constructor o método se ejecutará

- Ejer 4.6 , 4.7 y 4.8

Constructores sobrecargados. Ejercicio.

```
public class Cuadrado
{
    private Punto centro;
    private double lado;
    .....
}

public class Punto
{
    private int x;
    private int y;
    public Punto()
    {
        x = 0;
        y = 0;
    }
    public Punto(int queX, int queY)
    {
        x = queX;
        y = queY;
    }
    .....
}
```

- Escribe tres constructores sobrecargados en la clase Cuadrado
 - a) sin parámetros. Crea un cuadrado de centro (0,0) y lado = 0
 - b) con tres parámetros, *queX*, *queY*, *queLado*
 - c) con dos parámetros, un punto ya creado, *quePunto* (es el centro) y *queLado*

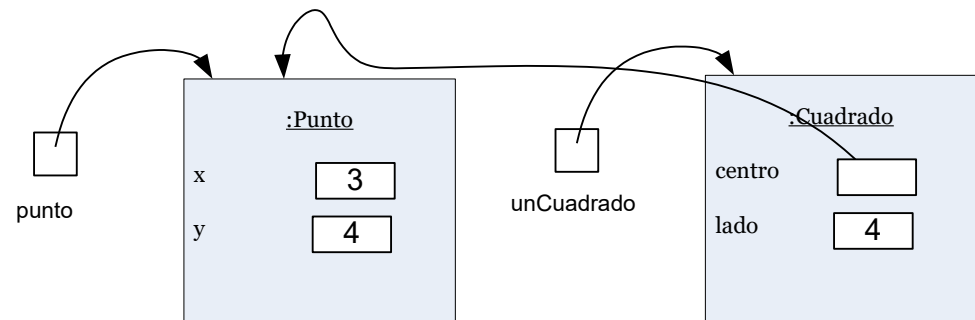
Constructores sobrecargados. Ejercicio (Sol.)

```
public class Cuadrado
{
    private Punto centro;
    private double lado;

    public Cuadrado()
    {
        centro = new Punto(0, 0);
        lado = 0;
    }
    public Cuadrado(int queX, int queY, double queLado)
    {
        centro = new Punto(queX, queY);
        lado = queLado;
    }
    public Cuadrado(Punto quePunto, double queLado)
    {
        centro = quePunto;
        lado = queLado;
    }
}
```

- En código aparte (fuera de la clase Cuadrado y Punto)

```
Punto punto = new Punto(3, 4);  
Cuadrado unCuadrado =  
new Cuadrado(punto, 7.3);
```



Constructores sobrecargados. Ejercicio

- Define una clase Urna con 2 bolas de tipo Bola.
- La urna tiene 3 constructores
 - uno sin parámetros que crea dos bolas negras
 - otro con un parámetro que es el color de las bolas
 - el tercer constructor recibe las dos bolas como parámetros
- La clase Bola
 - un atributo color (valor entero)
 - dos constantes para el color blanco y negro
 - un constructor sin parámetros que inicializa la bola con color negro
 - un constructor con un parámetro , el color de una bola
 - accesor para el color
 - `public boolean esNegra()`
 - `public String toString()`

Ejercicio Urna (Sol.)

```
public class Bola
{
    private final int NEGRA = 1;
    private final int BLANCA = 2;
    private int color;

    public Bola()
    {
        color = NEGRA;
    }

    public Bola(int c)
    {
        color = c;
    }

    public boolean esNegra()
    {
        return color == NEGRA;
    }

    public int getColor()
    {
        return color;
    }

    public String toString()
    {
        String resul = "Bola ";
        if (color == NEGRA)
        {
            resul += "negra";
        }
        else
        {
            resul += "blanca";
        }
        return resul;
    }
}
```


Ejercicio Urna (Sol.)

```
public class Urna
{
    private Bola bola1;
    private Bola bola2;

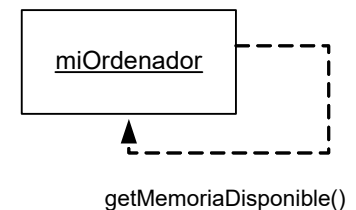
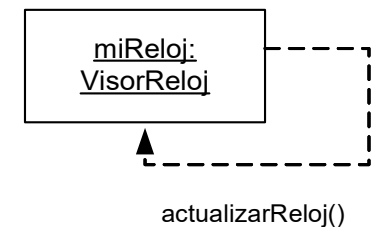
    public Urna()
    {
        bola1 = new Bola();
        bola2 = new Bola();
    }
    public Urna(int c)
    {
        bola1 = new Bola(c);
        bola2 = new Bola(c);
    }
    public Urna(Bola b1, Bola b2)
    {
        bola1 = b1;
        bola2 = b2;
    }
}
```

```
Urna urna = new Urna(2);
```

```
Bola bola1 = new Bola(1);
Bola bola2 = new Bola(2);
Urna urna = new Urna(bola1, bola2 );
```

Llamadas a métodos. Llamadas internas

- Internas / Externas
- Leer 4.7.
- Llamadas internas
 - un objeto se envía a sí mismo un mensaje
 - el receptor del mensaje es el propio objeto
 - emisor y receptor coinciden
 - *nombre_método(lista_parámetros);*
 - hay que tener en cuenta si el método devuelve o no un valor
 - actualizarReloj(); // no devuelve nada
 - double consumo = getConsumo(); //es una expresión que // devuelve un valor que he de utilizar
 - if (esBisiesto())
 - System.out.println("Memoria disponible " + getMemoriaDisponible());



Llamadas a métodos internos

```
public class Circulo
{
    .....
    public double calcularArea()
    {
        return PI * radio * radio;
    }
    public void printArea()
    {
        double area = calcularArea();
        System.out.println("Area " + area);
    }
}

public class Fecha
{
    .....
    public boolean esBisiesto()
    {
    }
    public int diasMes()
    {
        if (esBisiesto()) ...
    }
}
```

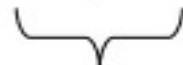
- los métodos internos permiten descomponer una tarea en subtareas
- evitan duplicar código
- cortos y con tareas concretas y precisas

Llamadas a métodos externos

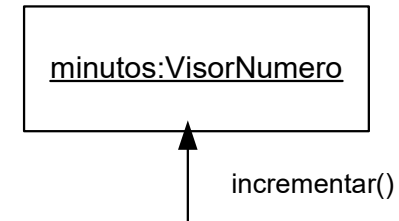
- Leer 4.7.2

-

objeto.nombre_método(lista de parámetros);



referencia al objeto



- paso de mensajes entre objetos

```
VisorReloj miReloj = new VisorReloj();  
miReloj.emitirTic();  
miReloj.ponerEnHora(4, 15);  
String horaString = miReloj.getHora();
```



desde fuera del objeto

Visibilidad public / private

- **public**
 - método público
 - visible desde fuera de la clase
 - puede ser invocado desde otros objetos
 - **minutos.incrementar();**
- **private**
 - método que solo puede ser llamado desde dentro de la propia clase (llamadas internas)
 - **actualizarReloj();**

Llamadas a métodos - Resumen

- La clase (dentro del objeto)
 - se conocen los atributos
 - se conocen todos los métodos, public y private
 - en el método se trabaja sobre o con el objeto receptor
 - para pasar otro mensaje al objeto receptor basta invocar al correspondiente método (sin poner receptor ni punto – interna)
 - el método puede utilizar también objetos locales y objetos como parámetros
- Desde fuera del objeto
 - no se conocen los atributos
 - no se conocen los métodos privados
 - solo se conocen los servicios que proporciona (métodos públicos)
 - se utiliza pasándole mensajes (*objeto.mensaje()*)
- Ejer 4.9 y 4.10

Ejer 4.9 y 4.10

■ Ejer 4.9

```
impre.imprimir("texto.txt", true);  
int estado = impre.getEstado();
```

■ Ejer 4.10

```
public class Bicicleta  
{  
    private Rueda delantera;  
    private Rueda trasera;  
    public Bicicleta()  
    {  
        delantera = new Rueda(0.0);  
        trasera = new Rueda(0.0);  
    }  
}
```

```
public void verificar()  
{  
    if (delantera.estaDesinflada())  
    {  
        delantera.inflar();  
        delantera.inflar();  
        delantera.inflar();  
    }  
    if (trasera.estaDesinflada())  
    {  
        delantera.inflar();  
        delantera.inflar();  
        delantera.inflar();  
    }  
}
```

Uso de this

- leer 4.9.1
- **this** en el código fuente
 - alude al receptor, al objeto actual
 - **Rueda miRueda = new Rueda(0.5);**
 - *miRueda* es el objeto actual (receptor)

```
public class Rueda
{
    private double presion;
    public Rueda( double presion)
    {
        presion = presion;
    }
    .....
}
```

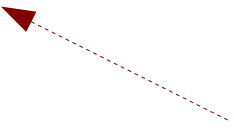
this.presion = presion;



Uso de this

```
public class Coche
{
    .....
    public void aparcar(Parking p)
    {
        p.aparcarCoche(this); // paso el objeto actual como parámetro
    }
    .....
}

public class Parking
{
    private Coche c;
    .....
    public void aparcarCoche(Coche c)
    {
        this.c = c;
    }
    .....
}
```



para pasar el objeto actual
como parámetro a otro
método.

■ Ejer 4.11

Uso de null

- leer 4.9.2
- Cuando se declara una variable de tipo objeto
 - la variable inicialmente no apunta a nada
 - tiene el valor **null**
 - al invocar al constructor se le asigna un valor (una referencia al objeto)
- Si una variable tiene el valor *null*
 - no se pueden invocar a métodos (null.metodo()) lanza un error - excepción)
 - hay que evitar excepciones (NullPointerException)
- **Garbage Collector**
 - mecanismo Java para destruir objetos que no son referenciados por ninguna variable

Uso de null

```
public boolean masBaratoQue(Articulo a)
{
    if (a == null) { // este test permite verificar si la variable apunta o no a un objeto
        return false; // mejor lanzar una excepción
    }
    return (a.getPrecio() > this.precio);
}
```

- Ejer 4.12
- Ejer 4.13 (de momento no)

→ Guarda el proyecto Reloj Digital con un nuevo nombre Reloj Digital con segundos
→ Añade a la clase VisorReloj de este nuevo proyecto un nuevo atributo *segundos* y modifícala adecuadamente

Ejercicios

EJAD01_02 Reloj digital con alarma (clase de TestReloj)

EJAD03_04 Circulo / Punto (con clase TestPunto)

EJAD05 Linea / Punto

EJAD06 Fracción

EJAD07 CuentaBancaria

Repaso Math.random()

- Clase Math
 - clase de utilidades
 - contiene constantes y métodos estáticos (métodos de clase) PI E
 - en el paquete *java.lang* que se importa automáticamente
 - `Math.pow()` `Math.random()` `Math.sqrt()`
- `Math.random()`
 - `0.0 <= Math.random() < 1.0` (genera un aleatorio de tipo double)
 - `(int) (Math.random() * b) + a` - devuelve un entero entre [a, a + b)
 - `(int) (Math.random() * 10);` //entre 0 y 10 exclusive
 - `(int) (Math.random() * 50);` // entre 0 y 49
 - `(int) (Math.random() * 50) + 50;` // entre 50 y 99
 - `(int) (Math.random() * 10) + 1;` //entre 1 y 10
 - `(int) (Math.random() * 2);` //entre 0 y 1
 - `(int) (Math.random() * 2) + 1;` //entre 1 y 2

Repaso Math.random()

- Simular el lanzamiento de un dado
 - `(int) (Math.random() * 6) + 1;`
 - `Math.random()` - entre 0.0 y < 1.0
 - `Math.random() * 6` - entre 0.0 y < 6.0
- Simular una nota entre 1 y 10
 - `int nota = (int) (Math.random() * 10) + 1;`
- Temperatura entre 10 y 30
 - `Math.random() * 21 + 10`

Repaso Math.random()

- **Math.random()** – genera un aleatorio $0 \leq n^{\circ} < 1.0$
- **Math.random() * 10** – genera un aleatorio $0 \leq n^{\circ} < 10.0$
- **(int) (Math.random() * 10)** – genera un aleatorio $0 \leq n^{\circ} \leq 9$
- **(int) (Math.random() * 10) + 1** – genera un aleatorio $1 \leq n^{\circ} \leq 10$
- **(int) (Math.random() * 50) + 50** –
 - un aleatorio $50 \leq n^{\circ} \leq 99$ ($50 \leq n^{\circ} < 100$)

- **(int) (Math.random() * (max - min)) + min** – aleatorio entre [min, max)
- **(int) (Math.random() * (max - min + 1)) + min** – aleatorio entre [min, max]

```
import static java.lang.Math.pow;  
así puedo poner en el código pow(2, 3) y no Math.pow(2, 3)
```

Clase Random

- clase de la API
 - para generar números aleatorios
 - en el paquete *java.util* (*hay que importarla*)
 - Java organiza las clases en paquetes que son agrupaciones de clases lógicamente relacionadas
 - Cuando se utiliza una clase (nuestra o de la API) Java tiene que saber en qué paquete se encuentra
 - la sentencia **import** le dice al compilador dónde localizar la clase
- ¿Cómo utilizar Random?
 - crear una instancia de la clase Random
 - llamar a un método de esa instancia para obtener el nº aleatorio, por ej, **nextInt()**
 - importarla incluyendo la sentencia **import java.util.Random** al comienzo de la declaración de la clase que utilizará a Random

Clase Random

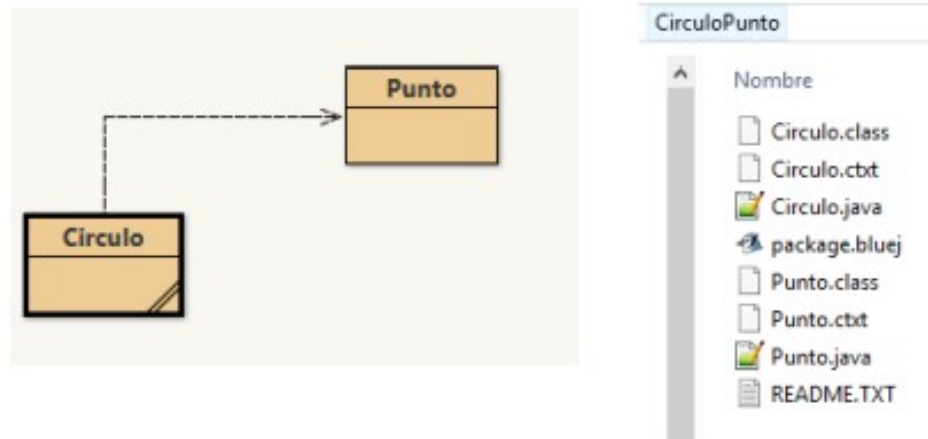
```
import java.util.Random; //la primera sentencia de una clase
public class DemoRandom
{
    private Random generador; // si no importo java.util.Random generador
    public DemoRandom()
    {
        generador = new Random();
    }
    public void testRandom()
    {
        int valor = generador.nextInt(10) + 1;
        int otroValor = generador.nextInt(21) + 10; //entre 10 y 30
        System.out.println("Valor " + valor);
        System.out.println("Otro valor " + otroValor);
    }
}
```

como atributo para
no crear un objeto
Random cada vez que
generemos un aleatorio

■ Ejer 4.17

```
public void tirarDado()
{
    cara = generador.nextInt(6) + 1;
    // cara = (int) (Math.random() * 6) + 1;
}
```

Estructura de un proyecto BlueJ



- | | |
|----------------------|--|
| package.bluej | Hay uno por paquete. Guarda detalles del proyecto. |
| *.java | Un fichero fuente java por cada clase del proyecto |
| *.class | Fichero de código standard java compilado. Hay uno por clase |
| *.ctxt | Fichero BlueJ con información adicional. Hay uno por clase |

Desarrollando Java fuera de BlueJ

- Edición y compilación fuera de BlueJ
 - editar - con cualquier editor de textos (NotePad++) **.java**
 - compilar desde línea de comandos
 ->**javac** Circulo.java se crea el **.class**
 - ejecutar llamando a la JVM
 ->**java** Circulo
 - error,
- Método **main()**
 - El sistema Java siempre inicia la ejecución de una aplicación buscando un método llamado **main()**
 - Este método debe existir y debe tener la siguiente signatura:
public static void main(String[] args)

“Exception in thread main”
java.lang. No Such Method Error: main

Desarrollando Java fuera de BlueJ

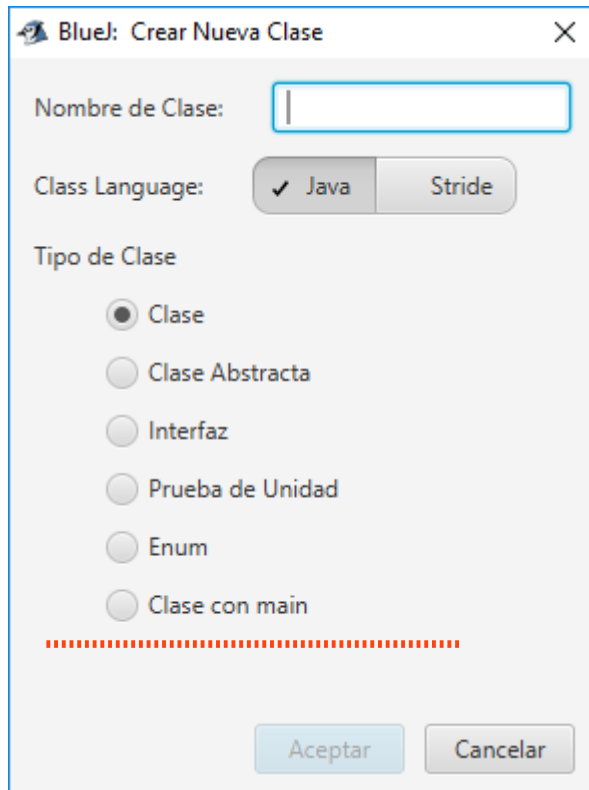
- El método `main()`
 - debe existir en alguna clase del proyecto
 - ha de ser público (para que pueda ser invocado desde fuera de la clase)
 - debe ser **static**
 - (método de clase – quiere decir que no se necesitan instancias de la clase que contiene a `main()` para invocarlo)
 - debe tener un array de parámetros `String` (esto permitirá pasar argumentos a la aplicación en el momento de su inicio)
 - sólo se puede invocar a `main()` para ejecutar una aplicación (punto de entrada a la aplicación)
- Incluiremos el `main()` en una clase aparte

Desarrollando Java fuera de BlueJ

```
public class AppEjemplo
{
    public static void main(String[] args)
    {
        Interfaz interfaz = new Interfaz();
        interfaz.ejecutar();
    }
}

public class TestCirculo
{
    public static void main(String[] args)
    {
        Circulo miCirculo;
        Punto unPunto;
        double x, y, area;
        unPunto = new Punto(3,6);
        miCirculo = new Circulo(unPunto, 6);
        x = unPunto.getX();
        y = unPunto.getY();
        area = miCirculo.calcularArea();
        System.out.println(" Area =" + area + "\n");
    }
}
```

Crear plantilla en BlueJ para la clase con main()



■ Abre una plantilla y guárdala con el nombre **main.tmpl**

- C:\BlueJ\lib\spanish\templates\newclass
- haz los cambios para que solo contenga el main()

Disco local (C:) > Archivos de programa (x86) > BlueJ > lib > spanish > templates > newclass			
Nombre	Fecha de modifica...	Tipo	Tamaño
abstract.tmpl	22/06/2017 15:21	Archivo TMPL	1 KB
enum.tmpl	22/06/2017 15:21	Archivo TMPL	1 KB
interface.tmpl	22/06/2017 15:21	Archivo TMPL	1 KB
main.tmpl	10/09/2017 19:43	Archivo TMPL	1 KB
README	22/06/2017 15:21	Archivo	2 KB
stdclass.tmpl	28/08/2017 20:12	Archivo TMPL	1 KB
unittest.tmpl	22/06/2017 15:21	Archivo TMPL	1 KB

Crear plantilla en BlueJ para la clase con main()

```
$PKGLINE
/**
 *
 * @author
 * @version
 */
public class $CLASSNAME
{

    /**
     *
     *
     */
    public static void main(String[] args)
    {

    }

}
```

Crear plantilla en BlueJ para la clase con main()

- En C:\BlueJ\lib\spanish\labels – hacia la línea 430
 - pkgmgr.newClass.main=Clase con main

```
428 pkgmgr.newClass.unittest=Prueba de Unidad
429 pkgmgr.newClass.enum=Enum
430 pkgmgr.newClass.main=Clase con main
431
```

- En C:\BlueJ\lib\bluej.defs – hacia la línea 315
bluej.classTemplates = stdclass abstract interface unittest enum main

```
314
315 bluej.classTemplates.java = stdclass abstract interface unittest enum main
316 bluej.classTemplates.stride = stdclass abstract interface
317 #bluej.templatePath = /home/mik/bluej/lib/english/templates/newclass
318 #bluej.templatePath = F:\shared\bluej\templates
319
```

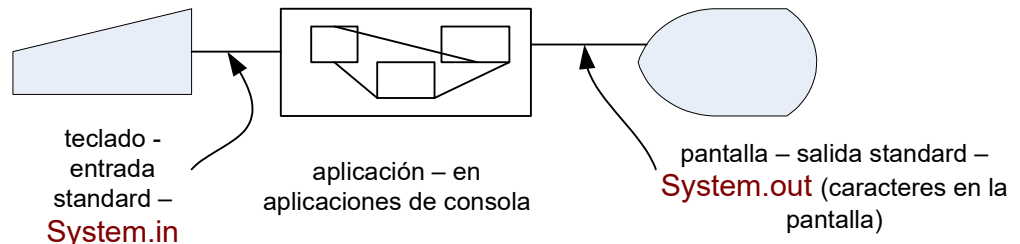
- Cerrar BlueJ y abrir

Desarrollando Java fuera de BlueJ

- En aplicaciones con interfaz de texto o gráficas
 - en el `main()` se crean los objetos y se llama a algún método que tome el control de la aplicación
- En toda aplicación Java al menos ha de haber una clase que contenga el `main()`
- Si estamos creando un conjunto de clases o un paquete Java que será utilizado por otros programadores (como la API) no se necesita el `main()`
- Desde BlueJ se puede llamar al `main()` sin crear instancias

La clase Scanner

- clase incorporada a partir de la versión 1.5 de Java
- en el paquete java.util (hay que importarla)
 - `import java.util.Scanner`
- permite leer valores desde el dispositivo de entrada standard, el teclado
 - lee caracteres desde un flujo de entrada (un input stream - el teclado - por ejemplo)
 - el flujo puede ser también un String, un fichero, ...
 - el flujo de caracteres se especifica cuando se crea el objeto Scanner)



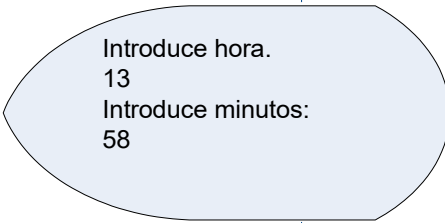
La clase Scanner

- se crea una instancia de la clase Scanner
 - `Scanner teclado = new Scanner(System.in);`
System.in se refiere a la entrada standard (System.out es la salida standard)
- se llama a un método de la clase para leer un entero, o un real, o un String, ...
 - `nextInt()` – lee un entero del teclado
 - `nextDouble()` – lee un real
 - `nextLine()` – lee un string completo hasta el fin de línea incluyendo los separadores (espacios, tabuladores,).
 - `next()` – lee un string
 - `nextBoolean()` – lee un valor booleano

flujo de
caracteres

La clase Scanner

```
import java.util.Scanner;
public class AppReloj
{
    public static void main(String[] args)
    {
        Scanner teclado = new Scanner(System.in);
        System.out.println("Introduce hora:");
        int hora = teclado.nextInt();
        System.out.println("Dame minutos:");
        int minutos = teclado.nextInt();
        RelojAlarma miReloj = new RelojAlarma(hora, minutos);
        miReloj.emitirTic();
        System.out.println("La hora es " + miReloj.getHora());
    }
}
```



Introduce hora.
13
Introduce minutos:
58

La clase Scanner

```
import java.util.Scanner;
public class TestPersona
{
    public static void main(String[] args)
    {
        Persona p = new Persona();
        Scanner teclado = new Scanner(System.in);
        String nombre;
        System.out.print(" Nombre de la persona  =");
        nombre = teclado.nextLine();
        p.setNombre(nombre);
        System.out.print(" Edad de la persona  =");
        p.setEdad(teclado.nextInt());
        p.printDatosPersona();
    }
}
```

Consideraciones sobre Scanner

- Scanner considera el flujo de entrada como una serie de tokens separados por espacios en blanco o tabuladores (delimitadores) o salto de línea
 - Ej. Si el usuario teclea `23 34 5ab 5bc Intro`
 - 23 es un token, 34 es un token, ...
 - `Scanner teclado = new Scanner(System.in);`
`int numero1 = teclado.nextInt(); // numero1 = 23`
`int numero2 = teclado.nextInt(); // numero2 = 34`
`String uno = teclado.next(); // uno = "5ab" hasta encontrar`
`separador`
`String dos = teclado.nextLine(); // dos = "5bc" lee el resto de la`
`línea hasta el final y se sitúa en la siguiente`

Consideraciones sobre Scanner

- Si no hay tokens en el flujo de entrada se espera a que el usuario teclee algo
- Para leer un carácter
 - `teclado.next().charAt(0)` o
 - `teclado.nextLine().charAt(0)`
- Recomendable acompañar cada lectura con un mensaje previo y leer los datos de uno en uno

```
System.out.print("Teclea nombre: ");
String nombre = teclado.nextLine();
System.out.print("Teclea edad: ");
int edad = teclado.nextInt();
```

Consideraciones sobre Scanner

- ¿Qué pasa si leemos antes edad y luego nombre?

```
System.out.print("Teclea edad: ");
```

```
int edad = teclado.nextInt();
```

```
teclado.nextLine(); // ignorar Intro
```

```
System.out.print("Teclea nombre: ");
```

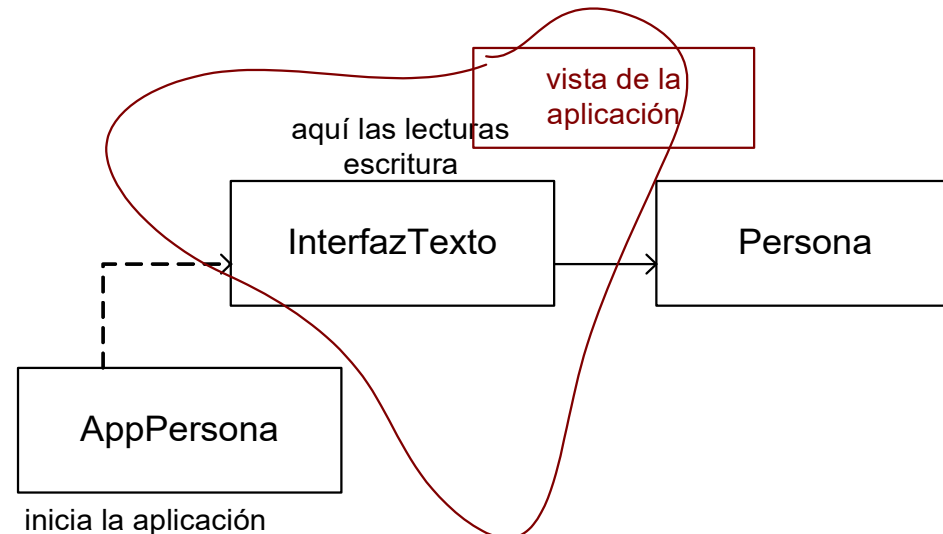
```
String nombre = teclado.nextLine();
```

- No pide nombre, lee el salto de línea introducido después de la edad
- Para leer (saltar) ese salto de línea

Organizando el código con Scanner

- ¿Cómo utilizaremos habitualmente Scanner?
 - en aquellas clases que interactúen con el usuario (interfaz de texto)
 - en esas clases leeremos desde el teclado y escribiremos en la pantalla

Persona
-nombre -edad
+Persona() +setNombre() +setEdad() +toString()



Organizando el código con Scanner

```
import java.util.Scanner;
public class InterfazTexto
{
    private Scanner teclado;
    private Persona p;
    public InterfazTexto ()
    {
        Scanner teclado = new Scanner(System.in);
        p = new Persona();
    }
    public void leerDatosPersona ()
    {
        String nombre;
        System.out.println("Nombre");
        nombre = teclado.nextLine();
        System.out.println("Edad");
        int edad = teclado.nextInt();
        p.setNombre(nombre);
        p.setEdad(edad);
    }
}
```

```
public void mostrarDatosPersona ()
{
    System.out.println(p.toString());
}
```

En una clase AppPersona, en el método main():

```
InterfazTexto interfaz = new InterfazTexto ();
interfaz.leerDatosPersona();
interfaz.mostrarDatosPersona();
```

Añadir atributos sueldo y estadoCivil a la clase Persona y modificar adecuadamente la clase InterfazTexto