

Tema 4. Métodos: definición, tipos y uso en Java

Duración: 3 sesiones

Índice:

1. Tipos de métodos Java

Sesión 1

- Definición (dinámicos y estáticos), métodos representativos (constructores y main) y diferencias a la hora de definirlos y usarlos
- Tipos de clases según el tipo de los métodos que contienen (Programa, Tipo de Datos y de Utilidades)
- Detalles sobre métodos estáticos (ejercicios usando la clase Math) y dinámicos (Objeto en Curso y referencia this)

2. Uso y diseño de métodos en base a su especificación

- Recordatorio: parámetros y argumentos de un método, su resultado y el tipo de este; invocación a un método según su tipo de resultado
- Documentación, o Especificación
- Ejercicios

3. Sobrecarga y Sobrescritura de un método Java

- Sobrecarga de métodos y variables de una clase (Ppio. de Máxima Proximidad). Ejercicios
- Sobrescritura de los métodos toString y equals. Ejercicios

4. Ejecución de (una llamada a) un método: paso de parámetros por valor; trazas; Registro de Activación y Pila de Llamadas

Nota

Para que puedas practicar con los conceptos que se introducen en esta sesión y responder a las cuestiones que se te formulan en adelante...



BlueJ: ejemplos-Tema4



- **Descarga** (desde la carpeta Tema 4 de la PoliformaT) y **descomprime** el proyecto **BlueJ ejemplos – Tema 4**
 - **Abre el proyecto** (clic en el icono de BlueJ) y prepárate para usarlo
- OJO:** la clase **PuntoR** de este proyecto es una versión de la usada en el tema anterior

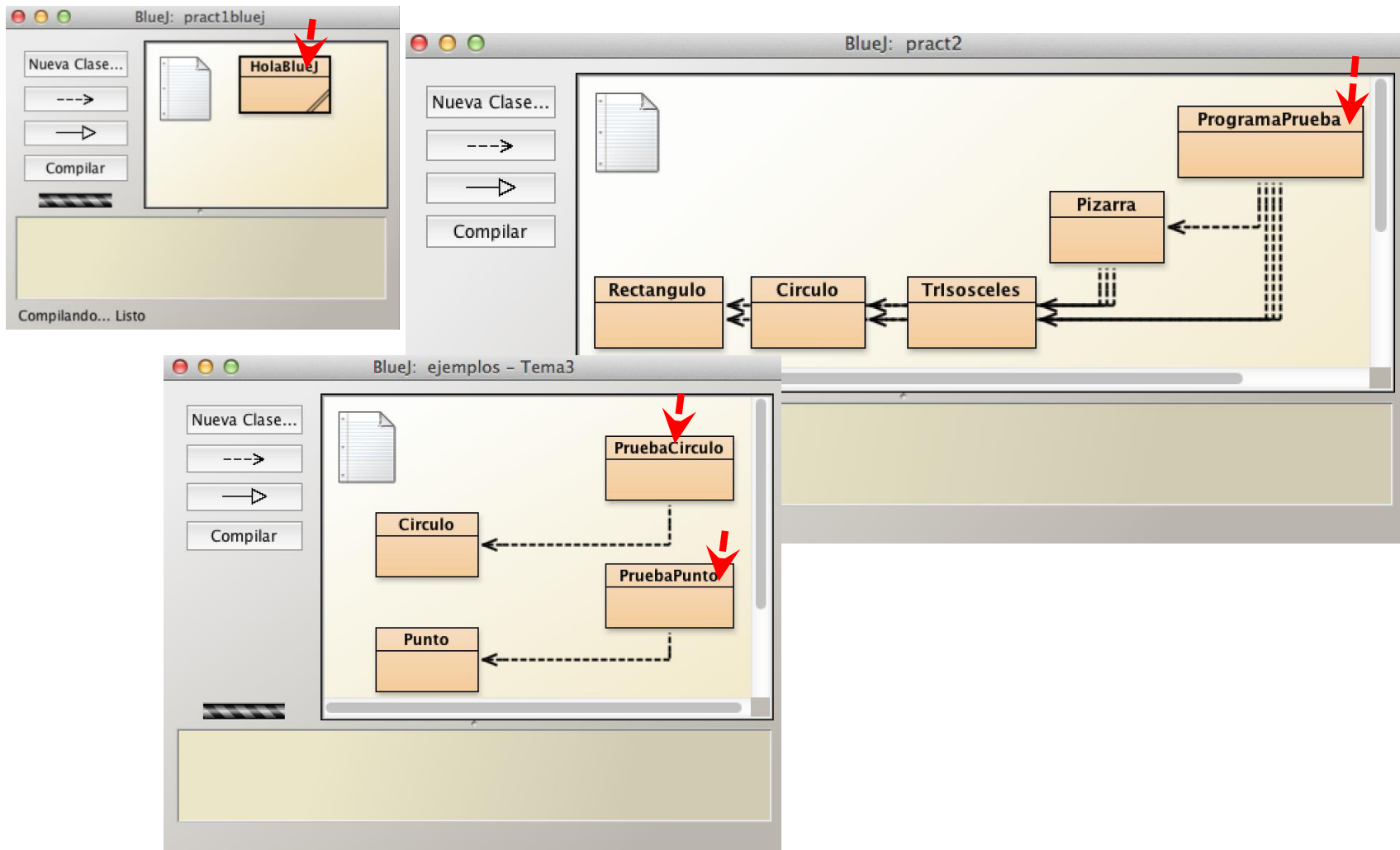
Tipos de métodos Java: definición y representantes (I)

Recuerda las aplicaciones Java que hemos visto. Entre sus **métodos**...

¿Cuál es imprescindible SIEMPRE? ¿En qué clase se define SIEMPRE?

main

Clase Programa



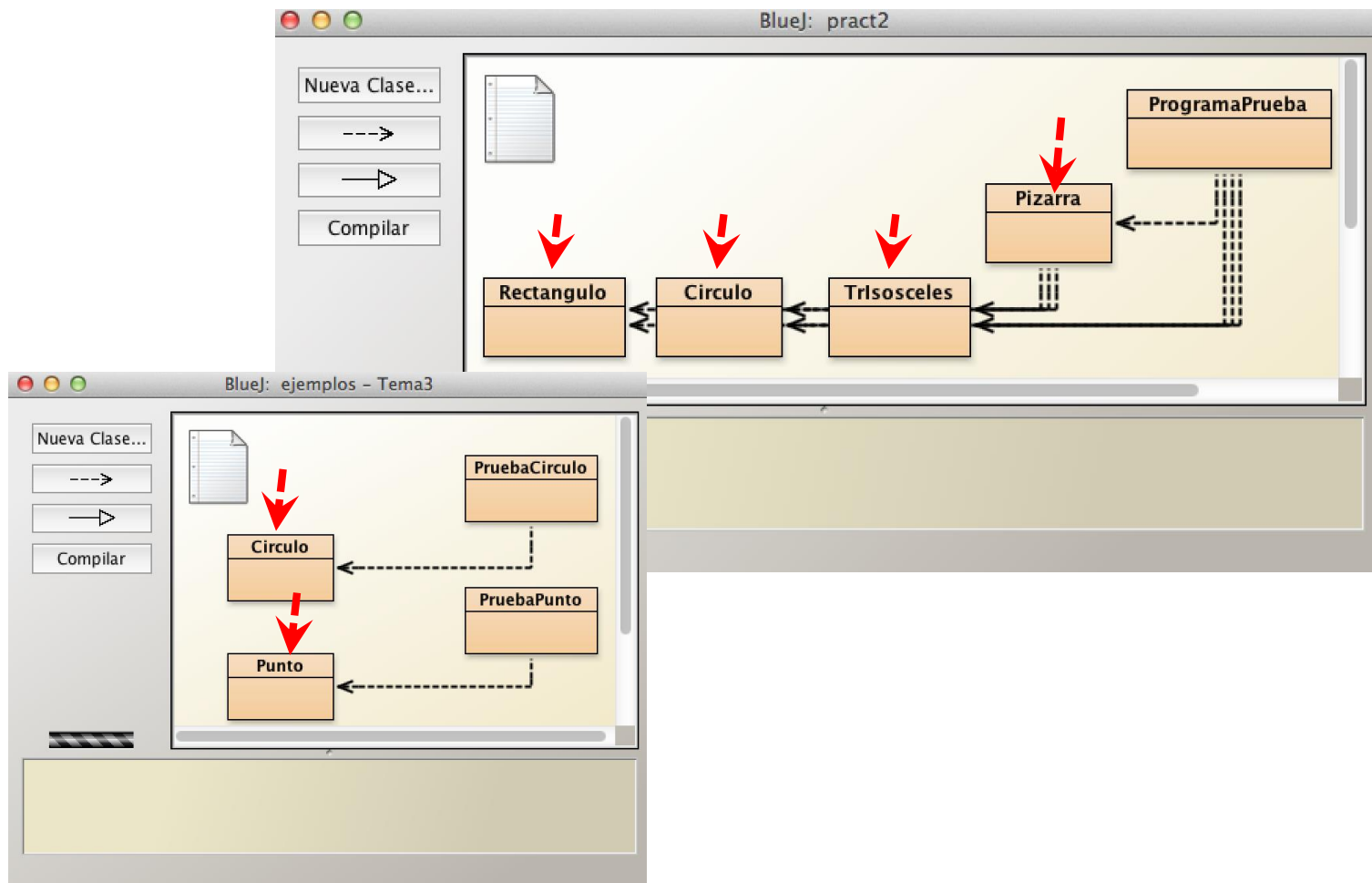
Tipos de métodos Java: definición y representantes (II)

Recuerda las aplicaciones **no triviales** que hemos visto. Entre los **métodos** que se **usan en el main** de su clase Programa...

¿Cuáles son imprescindibles SIEMPRE? ¿En qué clases se han definido SIEMPRE?

Constructores

Clases Tipo de Datos



Tipos de métodos Java: definición y representantes (III)

Observa las **cabeceras** del **main** y del resto de métodos de una aplicación (constructores, consultores y modificadores)

¿Cuál es el elemento que distingue la del **main de todas las demás?**

Modificador static

PISTA: sitúa cada elemento de cada una de las siguientes cabeceras de métodos en la columna que le corresponde (según la definición general de cabecera de un método Java), usando un guión ("-") para indicar que un hueco debe quedar vacío

```
public static void main(String[] args)
public PuntoR(double abs, double ord)
public double distOrigen()
public void setRadio(double nuevo)
```

¿Para qué hemos usado ya el modificador static?

Modificador de visibilidad	Otros modificadores	Tipo del resultado	identificador	Lista de Parámetros
public	static	void	main	String[] args
public	-	-	PuntoR	double abs, double ord
public	-	double	distOrigen	-
public	-	void	setRadio	double nuevo

Recuerda – Tema 2: estructura básica de un método

```
[mod. visibilidad][otros] tipo nomMetodo([listaParams]) {  
    [[final] tipo nomVarLocal1;  
    [final] tipo nomVarLocal2;  
    ...  
    [final] tipo nomVarN; ]  
    [instrucción1; ... ; instrucciónM;]  
    return [expresiónTipo];  
}
```

Cabecera (o perfil)

Cuerpo

Recuerda – Tema 2: modificadores de visibilidad

- **private**: acceso **solo dentro** de la clase
- **public**: acceso desde **cualquier otra** clase

Recuerda – Tema 3: “Constantes Java”

- Declaración del atributo **PI_APROX** de la clase **Circulo** como “Constante Java”
`public static final double PI_APROX = 3.14;`

Tipos de métodos Java: definición y representantes (IV)

- ① Recuerda **cómo has usado las “Constantes Java”** y los métodos de la clase `Math` en el Tema 3. Luego, responde...

¿Cómo se **USA** un método **static** (vía operador `.`) desde fuera de la clase donde se ha definido?

```
NombreDeLaClase.nombreDelMetodo(...)
```

- ② Identifica qué métodos se **USAN** (invocan) en el cuerpo de los siguientes, declarados en las clases `Punto` y `PruebaCírculo` del proyecto *ejemplos – Tema 3*

- El método `distOrigen` definido en la clase `Punto`

```
public double distOrigen() { return Math.sqrt(x * x + y * y); }
```

- El método `main` de la clase `PruebaCírculo`:

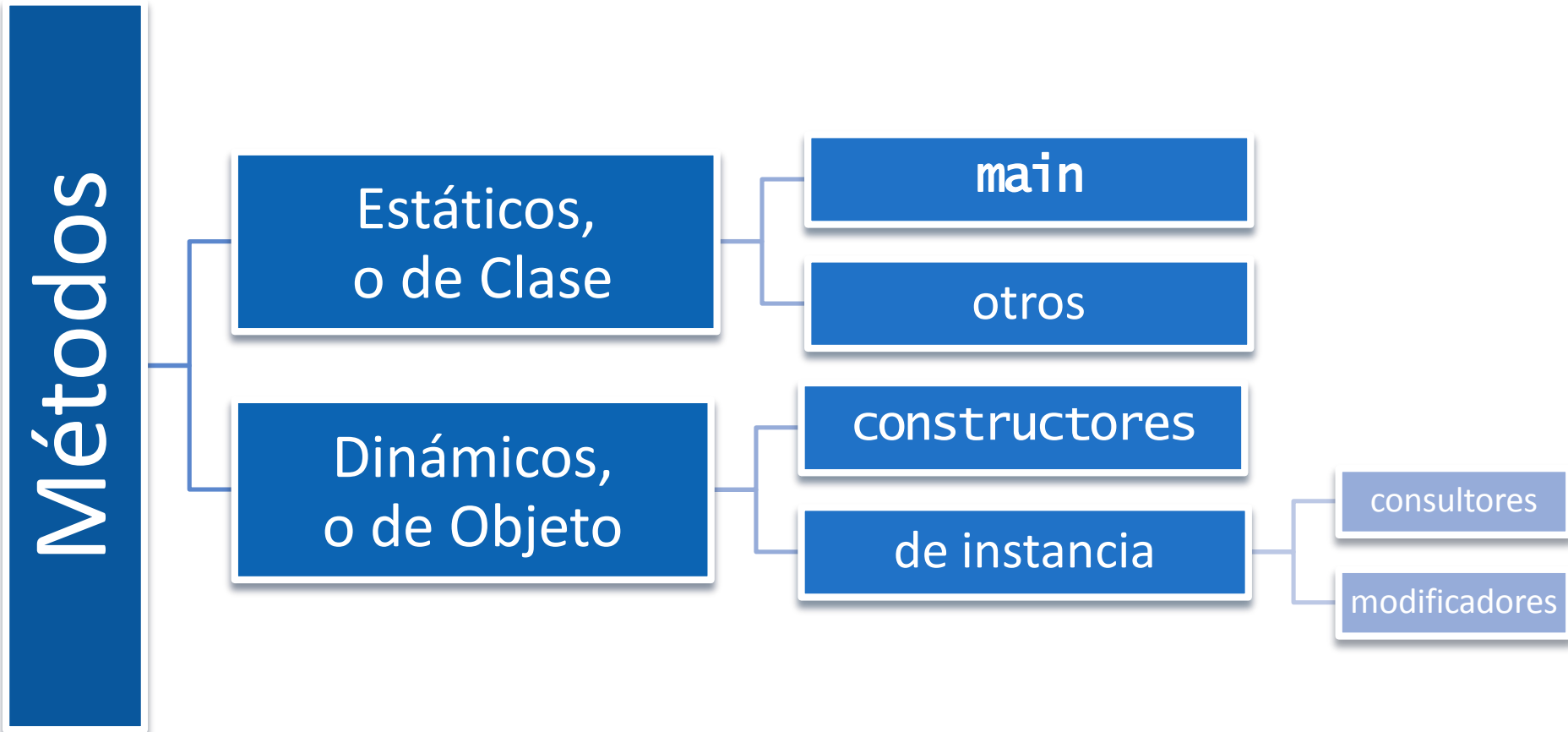
```
c1.setRadio(2 * c1.getRadio());
```

```
System.out.println(c2.toString());
```

Luego, por la forma en la que se invocan, indica cuáles son métodos **static** y cuáles no (i.e. cuáles son dinámicos)

Tipos de métodos Java: definición y representantes (V)

Observa que un método siempre se usa, i.e. se invoca o llama o aplica, **en el cuerpo** de la definición **de otro método**. Observa también que...



Tipos de métodos Java: tipos de clases según el tipo de los métodos que contienen

- Clase Programa: **SOLO** métodos **estáticos**, entre los que **SIEMPRE** está **main**
- Clase Tipo de Datos: **CUALQUIER** tipo de métodos, **MÍNIMO** uno dinámico y **NUNCA** **main**
- Clase de Utilidades: **SOLO** métodos **estáticos**, entre los que **NUNCA** está **main**

➡ Agrupan métodos de utilidad general sobre tipos previamente definidos, por lo que **SOLO** pueden tener atributos estáticos

Ejemplos: clase **Math**; clase **UtilPunto**, en la Figura 4.12 del libro de la asignatura

Tipos de métodos Java: detalles sobre métodos estáticos

1. Se DECLARAN como **static**...

- **main**, el representante de este tipo de métodos, **SOLO** en una clase Programa: su nombre y el resto de su cabecera están **predefinidos**
- **CUALQUIER** otro método, en **CUALQUIER** tipo de clase. Ejemplos: clases **Math** y **PuntoR**

2. Se USAN (vía operador **.**) sobre su clase: **NombreClase.nombreMetodo(...)**

- **NO** hay que crear (vía operador **new**) **NINGÚN** objeto para poder usarlos
- El método **main** es invocado por la JVM POR DEFECTO al ejecutar **java NombreClase**



BlueJ: ejemplos-Tema4

- **Recuerda** las características de la clase **Math** (en la documentación del API de Java 8 o en *Tema3 - La clase Math y las 'Constantes Java'*, el resumen del API que hay en mi carpeta de PoliformaT). Luego, **responde** a las siguientes cuestiones sobre la clase:
 - ¿Cuántas variables y métodos **estáticos** tiene?
 - **Calcula** en el *Code Pad*: el valor de π ; la raíz cuadrada de **4.5**; **4.5⁻¹¹**; los valores redondeados a **int** de **4.5** y **4.4**
- **Abre** la clase **PuntoR** del proyecto y **responde**:
 - ¿Qué atributos **estáticos** define? ¿Qué métodos **estáticos** define? ¿Con qué nombres?
 - **Escribe** en el *Code Pad* una expresión que, usando un método estático de la clase, cree un objeto de tipo **PuntoR** a partir de las coordenadas polares (**5**, **45**). **Sitúa** el objeto creado en el *Object Bench* e **inspecciónalo**

Tipos de métodos Java: detalles sobre métodos dinámicos (I)

1. **NUNCA** se **DECLARAN** como **static**...

- Los métodos constructores, **los representantes** de los métodos dinámicos: su nombre está **predeterminado**, proporcionando Java un constructor por defecto –sin parámetros– para cualquier clase. En principio, **SOLO pueden declararse** en una **clase Tipo de Datos**
- Los métodos consultores y modificadores de atributos NO estáticos. **SOLO pueden declararse** en una **clase Tipo de Datos**

2. Se **USAN** (vía operador **.**) **sobre un objeto de su clase**, el denominado **objeto en curso**. **OJO:** para evitar errores, **el objeto en curso debe crearse PREVIAMENTE** (vía operador **new**)



BlueJ: ejemplos-Tema4

- **Edita** la clase **PuntoR** del proyecto, busca el método **moverAleat** y **responde**...
 1. ¿**Por qué** es un método **dinámico**?
 2. ¿**Qué variables** de la clase son **accesibles** desde él? ¿Cuáles son sus **tipos y roles**?
 3. ¿**Qué tipo de método dinámico** es (**constructor, modificador o consultor**)? Justifícalo
 4. ¿**Qué método estático** usa? ¿**De qué clase** es? ¿Es **dinámico o estático**?
 5. **Escribe** en el *Code Pad* la secuencia de instrucciones que crea el punto nuevo (**2.0, 3.0**) y lo desplaza de forma aleatoria. Luego, **sitúa** nuevo en el *Object Bench* e **inspecciónalo**

Tipos de métodos Java: detalles sobre métodos dinámicos (II)

Objeto en curso



BlueJ: ejemplos-Tema4

Observa el cuerpo del método **main** de la clase **PruebaPuntoR** y **responde**...

1. ¿Sobre qué objeto se aplica (**Objeto en Curso**) el método **distOrigen** en su **4ª** instrucción?
2. ¿Cuál es el **Objeto en Curso** tras ejecutarse su primera instrucción?

PruebaPuntoR

Compilar Deshacer Cortar Copiar Pegar Encontrar...

```
8  
9 public static void main(String[] args) {  
10     PuntoR a = new PuntoR(1.5, 1.5);  
11     PuntoR b = new PuntoR();  
12     double dB = b.distOrigen();  
13     double dA = a.distOrigen();  
14 }
```

HEAP

361b8504
PuntoR

No HEAP

new PuntoR(1.5, 1.5)
<object reference> (PuntoR)
361b8504

a:
PuntoR

PuntoR a = new PuntoR(1.5, 1.5);
a
<object reference> (PuntoR)
361b8504

Tipos de métodos Java: detalles sobre métodos dinámicos (III)

Referencia this



BlueJ: ejemplos-Tema4

Observa el cuerpo del método **aleatorio** de la clase **PuntoR** y **responde**...

1. ¿Qué dos métodos usa?
2. ¿Cuál de ellos es dinámico? ¿De qué clase? ¿Sobre qué objeto se aplica (**Objeto en Curso**)?

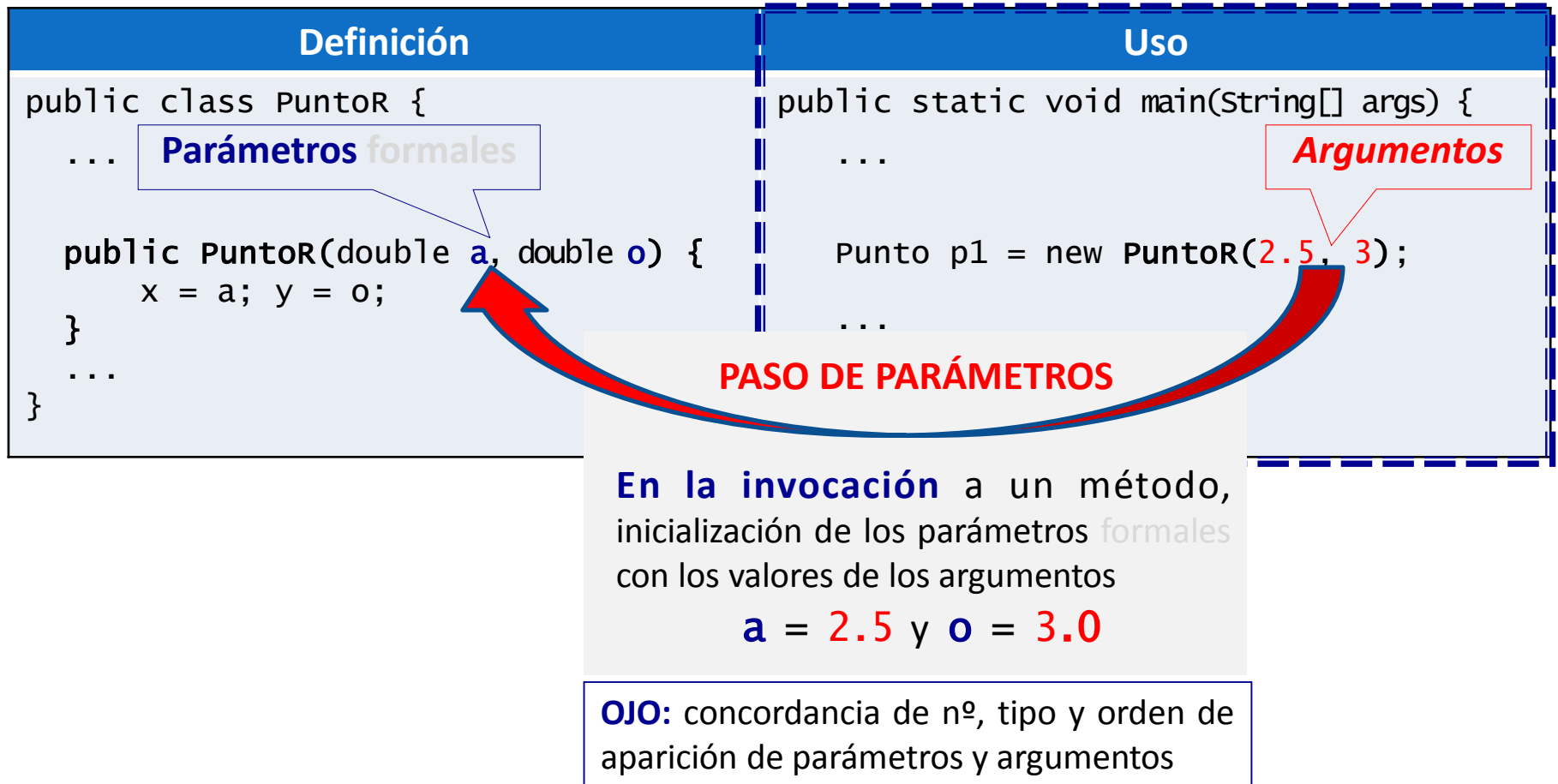
```
public class PuntoR {  
    private double x, y; PuntoR this;  
    ...  
    public double distOrigen() { return Math.sqrt(x * x + y * y); }  
    private double aleatorio() {  
        return Math.random() * (distOrigen() + 1);  
    }  
    ...  
}
```

Dentro del código de un método dinámico,
this es (la referencia a) el **Objeto en Curso**

```
private double aleatorio() {  
    return Math.random() * (this.distOrigen() + 1);  
}  
public double distOrigen() {  
    return Math.sqrt(this.x * this.x + this.y * this.y);  
}
```

Uso y diseño de métodos

RECUERDA: parámetros formales y argumentos



Uso y diseño de métodos

RECUERDA: invocación a un método según su tipo de resultado

- Si el **tipo de resultado** que devuelve un método es **distinto de void**, i.e. el método es **un constructor o un consultor**, su invocación (**en azulón**) puede realizarse en cualquier contexto (**en azul oscuro**) en el que se espera un resultado del mismo tipo, o compatible

Por ejemplo:

```
PuntoR p = new PuntoR();  
PuntoR q = PuntoR.dePolarARectangular(p.distOrigen(), 0.5);  
double d = p.distOrigen();  
System.out.println("La abscisa del punto es " + p.getX());
```

- Si el **tipo de resultado** que devuelve un método es **void**, i.e. el método es **un modificador**, su invocación es una instrucción más, sin más contexto que el punto y coma (;) que debe ir detrás de ella

Por ejemplo: suponiendo que los puntos **p** y **q** ya han sido creados...

```
p.mover(3.0, 4.0);  
q.moverAleat();  
p.setX(q.getX());
```

Uso y diseño de métodos: Documentación o Especificación

Definiciones

■ ¿Qué es?

- Especificación de todas las características de un método, tanto las de sus datos (parámetros y precondiciones) como las del resultado exacto que obtiene para cada entrada especificada –denominada de entrada-salida
- Evita referencias a los detalles de implementación, para no confundir qué hace el método con cómo lo hace

● ¿Para qué sirve?

- Para reutilizar el *software*, i.e. para saber cómo usarlo independientemente de su implementación: cuál es su perfil, qué condiciones especiales deben cumplir sus parámetros (precondiciones) y cuál es exactamente su resultado para cada entrada especificada
- Para producir un *software* de calidad, en el que la implementación siempre satisface la especificación, a modo de contrato

● Ejemplos:

- Documentación del API de Java
- Documentación de cualquier clase de cualquier proyecto BlueJ que hemos usado hasta la fecha

Uso y diseño de métodos: Documentación o Especificación Estándar Java

- La herramienta **javadoc** genera automáticamente el documento **html** con la documentación de una clase siguiendo el estilo estándar de Java

- Rasgos básicos:

```
/** Descripción del método, incluyendo precondiciones y
 * casos especiales de datos y resultados
 * @param parámetro1 tipo1
 * .....
 * @param parámetroN tipoN
 * @return tipoRetorno valor
 */
```

Descripción de sus parámetros

Descripción de su resultado, **salvo** cuando es **void**



BlueJ: ejemplos-Tema4

- Edita** la clase **PuntoR** y **obten** su documentación
- Abre** los ficheros **PuntoR.html** y **TanqueDeAgua.html** de la subcarpeta **doc** del proyecto
- Accede** desde BlueJ a la documentación de **String**, **compárala** con las de **PuntoR** y **TanqueDeAgua** e **indica** cuál de las dos últimas tiene un estilo más parecido al suyo
- Edita**, **SIN COMPILAR**, la clase **TanqueDeAgua**; observa que la especificación de sus métodos **Sí** contiene los rasgos básicos del estándar de Java.

Uso y diseño de métodos: Ejercicio



BlueJ: ejemplos-Tema4

Accede desde BlueJ a la Documentación de `String` y **busca** un método que te permita resolver el siguiente problema:

Convertir en un `String` un valor de tipo `double`, por ejemplo el resultado de $23.5 + 5.2$

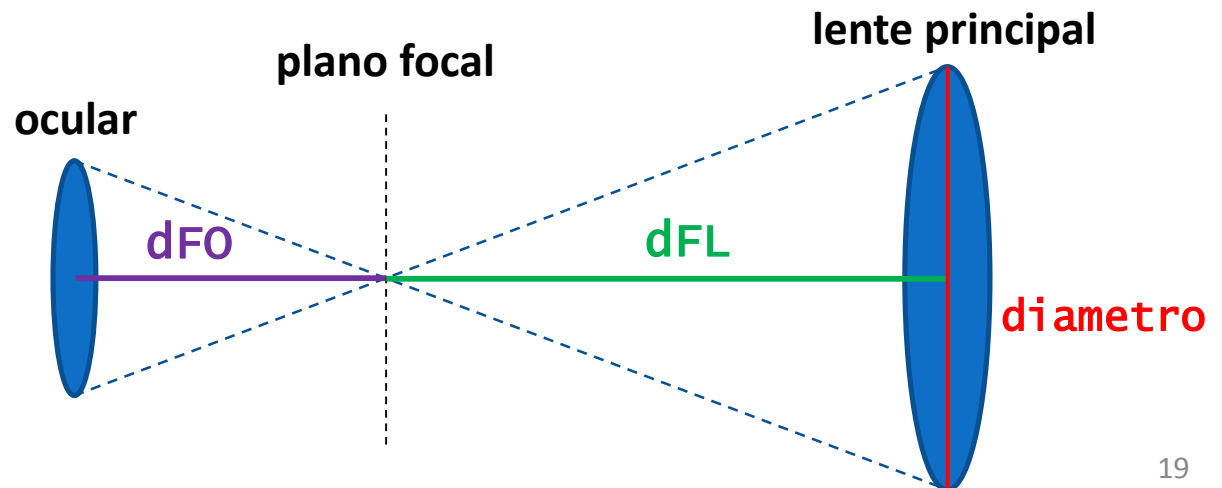
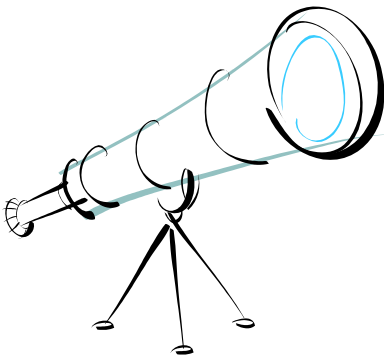
Cuando lo encuentres, **escribe en el *Code Pad*** del proyecto la expresión correspondiente, usando el método de `String` de acuerdo con su tipo y especificación

Uso y diseño de métodos: Ejercicio “La clase Telescopio”

Descripción y Representación Java de un telescopio

Para realizar este ejercicio, **descarga** (de la carpeta Tema 4 de PoliformaT) y **descomprime** el proyecto **BlueJ ejercicios – Tema 4**. También te puede resultar útil la siguiente descripción de un telescopio

- Un telescopio se puede caracterizar por medio de (**TIENE UN**)...
 - *diámetro* del objetivo, o lente principal (**díametro en mm**)
 - *distancia focal* de la *lente* principal (**dFL en mm**)
 - *distancia focal* del *ocular* (**dFO en mm**)
- A partir de los valores de estas componentes se pueden calcular, entre otros:
 - *aumentos*, o relación entre **dFL** y **dFO**
 - *relación focal*, o relación entre **dFL** y **díametro**



Uso y diseño de métodos: Ejercicio “La clase Telescopio”

Enunciados de las partes 1 y 2 del ejercicio



BlueJ: ejercicios – Tema 4

P1: usar una clase en base a su Especificación

Accede a la Documentación de la clase `Telescopio` del proyecto BlueJ *ejercicios – Tema 4* (en la subcarpeta `doc` del proyecto está el fichero `Telescopio.html` que la contiene) y **úsala** para diseñar, en el mismo proyecto, un programa `TestTelescopio` tal que...

1. **Cree** un objeto `t` de la clase `Telescopio` con una lente principal de **76.2** mm de diámetro, una distancia focal de **165.1** mm y una distancia focal ocular de **20.32** mm
2. **Muestre por pantalla**, con 2 cifras decimales, los aumentos y la relación focal del telescopio `t`
3. **Actualice** a un nuevo valor leído desde teclado el diámetro de `t`
4. **Muestre por pantalla**, con 2 cifras decimales, la nueva relación focal de `t`

PISTA: para familiarizarte con la clase que usarás en tu programa, puedes crear un `Telescopio t` en el *Object Bench* del proyecto, inspeccionarlo y comprobar su funcionalidad

RECUERDA: debes comprobar tu estilo de programación con el *Checkstyle* de BlueJ



BlueJ: ejercicios – Tema 4

P2: diseñar una clase en base a su Especificación

En el proyecto BlueJ *ejercicios – Tema 4*

- **Diseña** la clase `Telescopio` en base a su Especificación
- **Comprueba** después que tu programa `TestTelescopio` sigue funcionando correctamente

RECUERDA: debes comprobar tu estilo de programación con el *Checkstyle* de BlueJ

Uso y diseño de métodos: Ejercicio CAP



La clase Cuadrado que usa Punto (clave CCDHK4ai)

Completa el código de la siguiente clase **Cuadrado**, que usa un objeto de tipo **Punto** para representar su centro, en lugar de las coordenadas **centroX** y **centroY** de tipo **int** que se han usado en una versión previa de esta clase. Para ello debes usar la Especificación de **Punto** que tienes disponible en Documentos Relacionados

Tema 4. Métodos: definición, tipos y uso en Java

Duración: 3 sesiones

Índice:

1. Tipos de métodos Java

- Definición (dinámicos y estáticos), métodos representativos (constructores y main) y diferencias a la hora de definirlos y usarlos
- Tipos de clases según el tipo de los métodos que contienen (Programa, Tipo de Datos y de Utilidades)
- Detalles sobre métodos estáticos (ejercicios usando la clase Math) y dinámicos (Objeto en Curso y referencia this)

2. Uso y diseño de métodos en base a su especificación

- Documentación, o Especificación
- Ejercicios
- Parámetros formales y argumentos de un método, su resultado y el tipo de este

Sesión 2

3. Sobrecarga y Sobrescritura de un método Java

- Sobrecarga de métodos y variables de una clase (Ppio. de Máxima Proximidad). Ejercicios
- Sobrescritura de los métodos toString y equals. Ejercicios

4. Ejecución de (una llamada a) un método: paso de parámetros por valor; trazas; Registro de Activación y Pila de Llamadas

Recuerda – Tema 3:

Bloques de instrucciones – Variables globales y locales

- Una variable Java se debe definir al principio del bloque más interno en el que se usa
- El **ámbito** de una variable es la parte del bloque en la que es conocida y se puede usar
- Una variable es (de ámbito) **local** en el bloque en el que se define y **global** en los bloques internos a este
 - ➔ Son **variables locales en/a un método**
 - las que se definen en su cuerpo
 - sus parámetros, que se inicializan a los valores que se le pasan como argumentos (paso de parámetros **por valor**)
 - si el método es dinámico, la referencia (final) **this**
 - ➔ Son **variables globales en/a un método** los atributos de su clase (variables de clase e instancia) e, incluso, los definidos como públicos en otras



BlueJ: ejemplos-Tema4

- **Edita** el programa `TestLocalGlobal` e **indica**, si existen, los nombres de sus **variables locales** y **globales**
- **Ejecuta** el programa `TestLocalGlobal`... ¿Por qué se “queja” el compilador? Explícalo
- **Comenta** la línea “conflictiva” y **ejecuta** el programa `TestLocalGlobal`... Explica el resultado que aparece en el terminal de BlueJ

```
public class TestLocalGlobal {  
  
    private static final double PI = Math.PI;  
  
    public static void main(String[] args) {  
        System.out.println("constante PI en main: " + PI);  
        double argumento = PI;  
        System.out.println("argumento en main: " + argumento + "\n");  
        aCero(argumento);  
        System.out.println("argumento en main: " + argumento);  
    }  
    public static void aCero(double parametro) {  
        System.out.println("argumento en aCero: " + argumento);  
        System.out.println("constante PI en aCero: " + PI);  
        System.out.println("parametro en aCero: " + parametro);  
  
        parametro = 0.0;  
        System.out.println("parametro en aCero: " + parametro + "\n");  
    }  
}
```

Sobrecarga y sobrescritura de métodos Java

Sobrecarga de variables - Principio de Máxima Proximidad (I)

Si dentro de un método una variable **local** y otra **global** tienen el **mismo identificador** (sobrecarga de variables), siempre se asocia dicho identificador a la **variable local**

-Principio de máxima proximidad-

Para entender las consecuencias de este principio, realiza los siguientes ejercicios.



BlueJ: ejemplos-Tema4

- **Observa cómo modifica** tu profes@r el método constructor con 2 parámetros de la clase **PuntoR**:

```
public PuntoR(double abs, double ord) {  
    double x = abs;  
    double y = ord;  
}
```

- **Cuando compila** la clase, ... ¿**Algún error**?
- **Cuando ejecuta Checkstyle**... ¿**Alguna sugerencia**?
- **Cuando crea** un **PuntoR** con el constructor en el *Object Bench* del proyecto e **inspecciona** su estado... ¿ **Cómo es el resultado**, correcto o incorrecto? ¿**Por qué**?

Moraleja - ¡Evita complicaciones!

Al diseñar una clase, **NUNCA** uses el mismo nombre para una variable local que para una global. En concreto, **NUNCA vuelvas a declarar un atributo privado de una clase en ninguno de sus métodos**

Sobrecarga y sobrescritura de métodos Java

Sobrecarga de métodos

Dos o más **métodos** de una clase están **sobrecargados** si...

- tienen el **mismo nombre**, incluso el mismo tipo de resultado
- sus **parámetros** son **diferentes** en número o tipo u orden



BlueJ: ejemplos-Tema4

- **Accede** desde BlueJ a la documentación de `String` y busca los métodos `indexOf`, sobrecargados. En base a su especificación, y haciendo las pruebas que consideres oportunas en el *Code Pad* del proyecto, **indica** cuál de ellos usarías si quieres obtener la posición de la primera aparición del carácter 'r' en el `String` "Sobrecarga" y cuál si quieres obtener la de su segunda (y última) aparición
- **Edita** la clase `PuntoR` e **indica** cuáles de sus métodos están sobrecargados

Sobrecarga y sobrescritura de métodos Java

Sobrecarga de métodos constructores – uso de `this`

Siempre que una clase define más de uno, los métodos **constructores** están, por definición, **sobrecargados** (mismo nombre, sin tipo de resultado y listas de parámetros distintas). En este caso, la forma más eficiente y legible de definirlos es:

- Por orden **decreciente** de nº de parámetros
- El primero, el de mayor número de parámetros, consta del cuerpo habitual
- Cada uno de los restantes consta de un cuerpo cuya primera instrucción es

`this(argumentos_primer_constructor);`

o **invocación explícita al constructor de la clase con mayor nº de parámetros**



Bluej: ejemplos-Tema4

En la clase **PuntoR**, **observa** el orden de sus métodos constructores y recuérdalo cada vez que diseñes una clase. Luego, comenta las instrucciones existentes en el cuerpo de su segundo y tercer constructor y substitúyelas por una del tipo...

`this(argumentos_primer_constructor);`

Sobrecarga y sobrescritura de métodos Java

Ejercicio de sobrecarga de métodos constructores – Ampliación de Telescopio



BlueJ: ejercicios – Tema 4

- **Añade** los siguientes métodos a la clase **Telescopio** del proyecto:
 1. **Un constructor** que cree un telescopio con una lente principal de diámetro **d**, una distancia focal **dF** y una distancia focal ocular **dFOcular**
 2. **Un constructor** que cree un telescopio con una lente principal de diámetro **d** y distancias focal y ocular **estándares**. **Usa las constantes de la clase y `this(...)`**
- Hecho lo anterior, **modifica** el constructor sin parámetros existente para que use **`this(...)`**

RECUERDA comprobar que los constructores figuran en orden decreciente de nº de parámetros



BlueJ: ejercicios – Tema 4

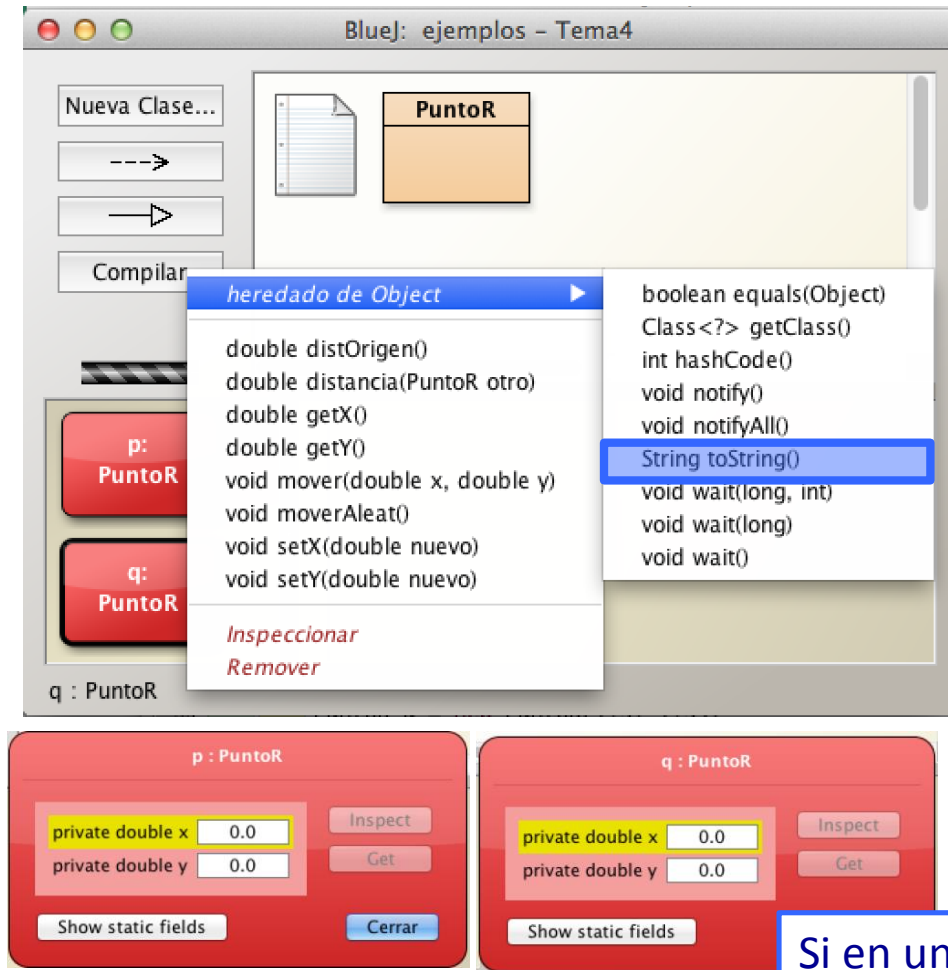
Modifica el diseño de la clase **TestTelescopio** del proyecto **ejercicios – Tema 4** de forma que...

1. **Cree** tres telescopios: **t1**, con diámetro **76.2** mm, distancia focal **165.1** mm y distancia focal ocular **20.32** mm; **t2**, con diámetro **76.2** mm y distancias focal y ocular **estándares**; **t3**, un telescopio **estándar**
2. **Muestre por pantalla**, con 2 cifras decimales, los aumentos y la relación focal de cada telescopio creado

RECUERDA comprobar con **Checkstyle** de BlueJ la corrección de tu estilo de programación

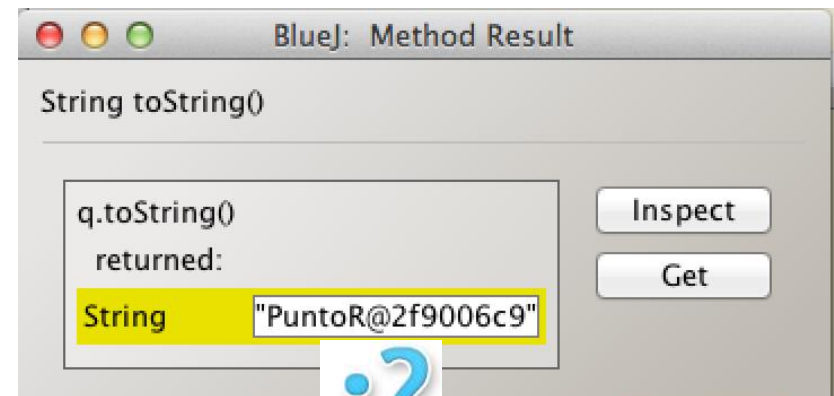
Sobrecarga y sobrescritura de métodos Java

Sobrescritura de los métodos equals y toString de Object()



Cualquier clase Java **ES UN** caso particular de Object → **HEREDA** todos sus métodos

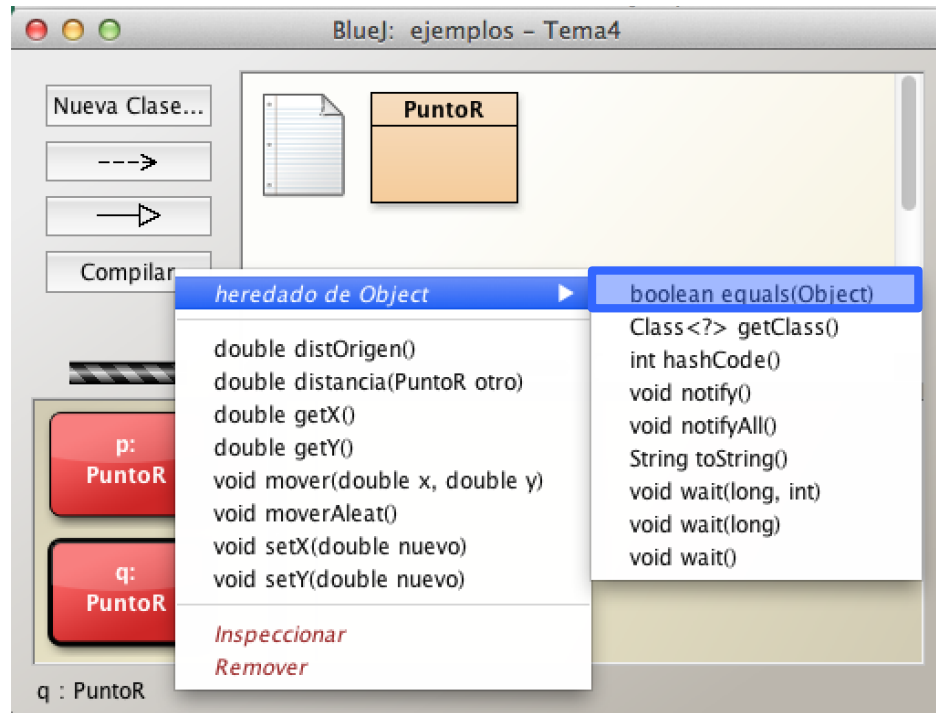
¿Y SI NO SOBREScribe toString?



Si en una clase **C** la JVM **NO** encuentra sobrescrito el método **m** de **Object**, ejecuta el código del método **m** definido en **Object**

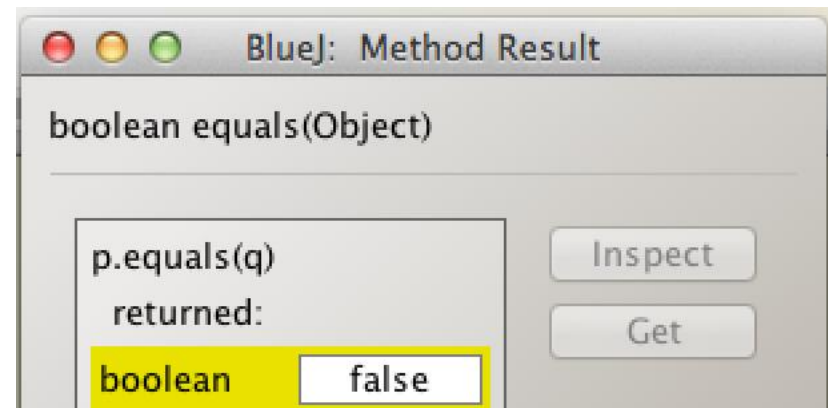
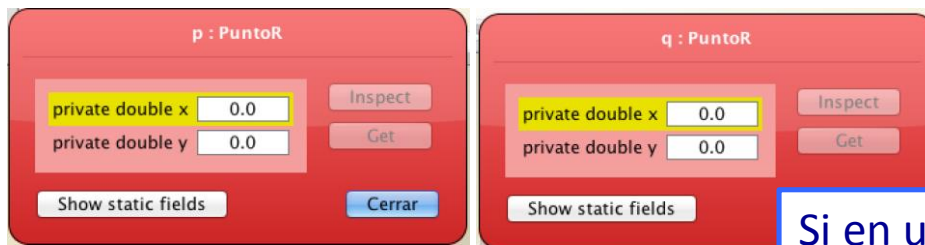
Sobrecarga y sobrescritura de métodos Java

Sobrescritura de los métodos equals y toString de Object(II)



Cualquier clase Java **ES UN** caso particular de Object → **HEREDA** todos sus métodos

¿Y SI NO SOBRESCRIBE equals?



Si en una clase **C** la JVM **NO** encuentra sobrescrito el método **m** de **Object**, ejecuta el código del método **m** definido en **Object**

Sobrecarga y sobrescritura de métodos Java

Sobrescritura de los métodos equals y toString de Object(III)

```
public class PuntoR {  
    ...  
    /** comprueba si un punto (this) es igual  
     * a otro, i.e. si son de la misma clase  
     * y si sus coordenadas coinciden */  
    public boolean equals(Object otro) {  
        return otro instanceof PuntoR  
            && this.x == ((PuntoR) otro).x  
            && this.y == ((PuntoR) otro).y;  
    }  
  
    /** devuelve un String que representa un  
     * punto en el formato matemático (x, y) */  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
    ...  
}
```

El estándar de Java recomienda comprobar que...

- otro es de la misma clase que **this** (vía instanceof)
- otro y **this** coinciden atributo a atributo

Sobrecarga y sobrescritura de métodos Java

Sobrescritura de los métodos equals y toString de Object (IV)



BlueJ: ejemplos-Tema4

Edita la clase **PuntoR** del proyecto, **añade** los métodos **equals** y **toString** definidos en la transparencia anterior, **compila** la clase y...

1. **Crea** dos puntos estándar **p** y **q** en el *Object Bench* del proyecto
2. **Ejecuta** el método **toString** sobre ellos y **observa** el resultado
¿Qué método **toString** se ha ejecutado, el definido en **Object** o en **PuntoR**?
3. **Ejecuta** el método **equals** para comprobar si **p** y **q** son iguales y **observa** el resultado

¿Qué método **equals** se ha ejecutado, el definido en **Object** o en **PuntoR**?

Si en una clase **C** la JVM **SÍ** encuentra sobrescrito el método **m** de **Object**, ejecuta el código del método **m** definido en **C**

Sobrecarga y sobrescritura de métodos Java

Ejercicio de sobrescritura de equals y toString – Segunda Ampliación de Telescopio



BlueJ: ejercicios – Tema 4

Edita la clase **Telescopio** del proyecto y **añade** los métodos **equals** y **toString**. Además, **redondea a dos decimales** los valores **double** del cuerpo del método **toString**

RECUERDA comprobar con **Checkstyle** de BlueJ la corrección de tu estilo de programación



BlueJ: ejercicios – Tema 4

Modifica el diseño de la clase **TestTelescopio** del proyecto de forma que...

1. **Muestre por pantalla** los tres telescopios creados (**toString**)
2. **Muestre por pantalla** el resultado de comprobar si **t1** es igual a **t2** o **t3**

RECUERDA comprobar con **Checkstyle** de BlueJ la corrección de tu estilo de programación

Tema 4. Métodos: definición, tipos y uso en Java

Duración: 3 sesiones

Índice:

1. Tipos de métodos Java

- Definición (dinámicos y estáticos), métodos representativos (constructores y main) y diferencias a la hora de definirlos y usarlos
- Tipos de clases según el tipo de los métodos que contienen (Programa, Tipo de Datos y de Utilidades)
- Detalles sobre métodos estáticos (ejercicios usando la clase Math) y dinámicos (Objeto en Curso y referencia this)

2. Uso y diseño de métodos en base a su especificación

- Documentación, o Especificación
- Ejercicios
- Parámetros **formales** y argumentos de un método, su resultado y el tipo de este

3. Sobrecarga y Sobrescritura de un método Java

- Sobrecarga de métodos y variables de una clase (Ppio. de Máxima Proximidad). Ejercicios
- Sobrescritura de los métodos toString y equals. Ejercicios

- 4. **Ejecución de (una llamada a) un método:** paso de parámetros por valor; trazas; Registro de Activación y Pila de Llamadas

Sesión 3

Ejecución de (una llamada/invocación a) un método

Paso de parámetros en Java

POR VALOR



BlueJ: ejemplos-Tema4

Abre el programa **TestPilaRA** y sigue las indicaciones del profes@r para: saber cómo mostrar por pantalla los valores de las variables de un método en cada instante de su ejecución (**traza**) y entender por qué son los que son (**paso de parámetros en Java**); entender los conceptos de **método activo**, Registro de Activación (**RA**) y **Pila** de Llamadas

```
13 public class TestPilaRA {
14     public static void main(String[] args) {
15         System.out.println("-----Metodo activo: main-----");
16         double a = 5.65; // PuntoR a = new PuntoR(1.5, 1.5);
17         double b = 6.87; // PuntoR b = new PuntoR();
18         System.out.println("-->ANTES DE ejecutar intercambiar, a = " + a
19             + " y b = " + b);
20
21         intercambiar(a, b);
22
23         // Mira los valores actuales de a y b en el RA del metodo activo:
24         // ¿Han cambiado al ejecutar intercambiar(a, b)? ¿Por que?
25         System.out.println("\n-->TRAS ejecutar intercambiar, a = " + a
26             + " y b = " + b);
27     }
28
29     // Encapsula el codigo de intercambio de (los valores) de a y b
30     private static void intercambiar(double /*PuntoR*/a, double /*PuntoR*/ b) {
31         System.out.println("\n\t----Metodo activo: intercambiar----");
32         double /*PuntoR*/copiaDeA = a;
33         // Mira los valores actuales de a, b y copiaDeA en el RA del metodo activo
34         System.out.println("\n\t-->INICIALMENTE, a = " + a + ", b = " + b
35             + " y copiaDeA = " + copiaDeA);
36
37         a = b;
38         b = copiaDeA;
39
40         // Mira los valores actuales de a y b en el RA del metodo activo
41         // ¿Han cambiado? ¿Por que?
42         System.out.println("\n\t-->FINALMENTE, a = " + a + ", b = " + b
43             + " y copiaDeA = " + copiaDeA);
44     }
45 }
```

BlueJ: BlueJ: Ventana de Terminal - ejemplos - Tema4

```
-----Metodo activo: main-----
-->ANTES DE ejecutar intercambiar, a = 5.65 y b = 6.87

----Metodo activo: intercambiar----

-->INICIALMENTE, a = 5.65, b = 6.87 y copiaDeA = 5.65

-->FINALMENTE, a = 6.87, b = 5.65 y copiaDeA = 5.65

-->TRAS ejecutar intercambiar, a = 5.65 y b = 6.87
```

BlueJ: BlueJ: Ventana de Terminal - ejemplos - Tema4

```
-----Metodo activo: main-----
-->ANTES DE ejecutar intercambiar, a = (1.5, 1.5) y b = (0.0, 0.0)

----Metodo activo: intercambiar----

-->INICIALMENTE, a = (1.5, 1.5), b = (0.0, 0.0) y copiaDeA = (1.5, 1.5)

-->FINALMENTE, a = (0.0, 0.0), b = (1.5, 1.5) y copiaDeA = (1.5, 1.5)

-->TRAS ejecutar intercambiar, a = (1.5, 1.5) y b = (0.0, 0.0)
```

Ejecución de (una llamada/invocación a) un método

Paso de parámetros por valor – Ejercicio



Paso de parámetros: Traza del programa Ejemplo1 (clave CCCDHG4ai)
(ejercicio nº 2-a del Capítulo 5 del libro)

Realizar este ejercicio te ayudará a terminar de entender lo que es...

- Invocar la ejecución de un método, desde el cuerpo de otro
- Un argumento y un parámetro de un método
- El paso de parámetros en Java (paso por valor) y sus consecuencias
para variables de tipos primitivos y variables referencia

Ejecución de (una llamada/invocación a) un método

- Recuerda -

Paso de parámetros en Java

POR VALOR

El **RA** asociado a la ejecución de un método **NO CONTIENE** sus variables locales y sus parámetros (inicializados a los argumentos usados al invocarlo) **SINO COPIAS DE ESTOS**

Paso de parámetros por valor

CONSECUENCIAS

Cualquier modificación de los valores de las **COPIAS** de las variables locales y parámetros de un método en su **RA** asociado **NO SON PERMANENTES SINO LOCALES**, i.e. “duran” en memoria lo mismo que su RA

Ejecución de (una llamada/invocación a) un método

Paso de parámetros en Java



BlueJ: ejemplos-Tema4

POR VALOR

Abre el programa **TestPilaRAObjetos** y **ejecútalo**. Recuerda en todo momento que...

- existe una diferencia entre **objeto** y **variable** que lo **Referencia**
- el **RA** asociado a la ejecución de un método se encuentra en la **PILA** y los **objetos** en el **HEAP**, dos zonas distintas de memoria

```
9 public class TestPilaRAObjetos {
10     public static void main(String[] args) {
11         System.out.println("-----Metodo activo: main-----");
12         PuntoR a = new PuntoR(1.5, 1.5);
13         PuntoR b = new PuntoR();
14         // Mira los valores actuales de (los OBJETOS) a y b en el RA del metodo activo
15         System.out.println("-->ANTES DE ejecutar intercambiar, a = " + a
16             + " y b = " + b);
17
18         intercambiar(a, b);
19
20         // Mira los valores actuales de (los OBJETOS) a y b en el RA del metodo activo
21         // ¿Han cambiado al ejecutar intercambiar(a, b)? ¿Por que?
22         System.out.println("\n-->TRAS ejecutar intercambiar, a = " + a
23             + " y b = " + b);
24     }
25
26     // Encapsula el codigo de intercambio de (los OBJETOS) a y b
27     private static void intercambiar(PuntoR a, PuntoR b) {
28         System.out.println("\n\t----Metodo activo: intercambiar----");
29         PuntoR copiaDeA = new PuntoR(a.getX(), a.getY());
30         // Mira los valores actuales de (los OBJETOS) a, b y copiaDeA en el RA del metodo activo
31         System.out.println("\n\t-->INICIALMENTE, a = " + a + ", b = " + b
32             + " y copiaDeA = " + copiaDeA);
33
34         a.setX(b.getX()); a.setY(b.getY());
35         b.setX(copiaDeA.getX()); b.setY(copiaDeA.getY());
36
37         // Mira los valores actuales de (los OBJETOS) a, b y copiaDeA en el RA del metodo activo
38         // ¿Han cambiado? ¿Por que?
39         System.out.println("\n\t-->FINALMENTE, a = " + a + ", b = " + b
40             + " y copiaDeA = " + copiaDeA);
41     }
42 }
```

BlueJ: BlueJ: Ventana de Terminal – ejemplos – Tema4

```
-----Metodo activo: main-----
-->ANTES DE ejecutar intercambiar, a = (1.5, 1.5) y b = (0.0, 0.0)

----Metodo activo: intercambiar----

-->INICIALMENTE, a = (1.5, 1.5), b = (0.0, 0.0) y copiaDeA = (1.5, 1.5)

-->FINALMENTE, a = (0.0, 0.0), b = (1.5, 1.5) y copiaDeA = (1.5, 1.5)

-->TRAS ejecutar intercambiar, a = (0.0, 0.0) y b = (1.5, 1.5)
```

Ejecución de (una llamada/invocación a) un método

Paso de parámetros por valor – Ejercicio



Paso de parámetros: Traza del programa Ejemplo2 (clave CCDHH4ai)
(ligera modificación del ejercicio nº 2-b del Capítulo 5 del libro)

Realizar este ejercicio te ayudará a entender lo que es...

- Invocar la ejecución de un método, desde el cuerpo de otro
- Un objeto y la variable Referencia que lo apunta
- El objeto en curso y la variable final `this`
- El paso de parámetros en Java (paso por valor) y sus consecuencias
PARA OBJETOS

Ejecución de (una llamada/invocación a) un método

- Recuerda -

Paso de parámetros en Java

POR VALOR

El **RA** asociado a la ejecución de un método **NO CONTIENE** sus variables locales y sus parámetros (inicializados a los argumentos usados al invocarlo) **SINO COPIAS DE ESTOS**

Paso de parámetros por valor

CONSECUENCIAS

Cualquier modificación de los valores de las **COPIAS** de las variables locales y parámetros de un método en su **RA** asociado **NO SON PERMANENTES SINO LOCALES**, i.e. “duran” en memoria lo mismo que su RA

PERO, como un objeto está en el Heap y no en la Pila, ...

Cualquier modificación del **OBJETO AL QUE APUNTA** (la copia de) una variable Referencia de un método en su **RA** asociado **NO ES LOCAL SINO PERMANENTE**, i.e. “dura” en (el Heap de la) memoria hasta que el objeto sea modificado de nuevo o sea desreferenciado

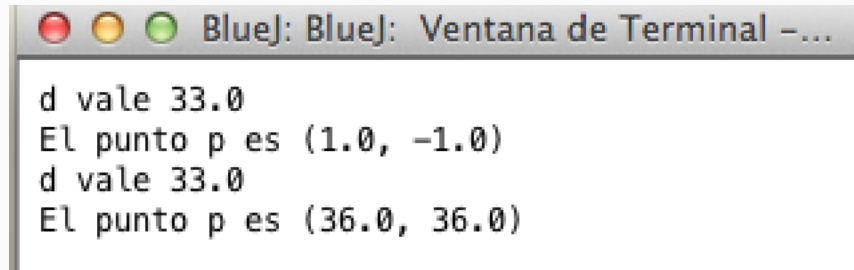
Ejecución de (una llamada/invocación a) un método

Ejercicio sobre paso de parámetros (Primer Parcial 2014-15)



BlueJ: ejercicios - Tema 4

- **Compila y ejecuta** `TestPasoDeParametros` del proyecto. **Comprueba** que en el terminal del proyecto aparece lo siguiente:



```
BlueJ: BlueJ: Ventana de Terminal -...  
d vale 33.0  
El punto p es (1.0, -1.0)  
d vale 33.0  
El punto p es (36.0, 36.0)
```

- A la vista del código de `TestPasoDeParametros` y del método `mover` de `Puntor`, **explica** el resultado de la ejecución de `TestPasoDeParametros`



Paso de parámetros: La clase Ejercicio 3 (clave CCDLGIabj)
(ejercicio nº 3 Primer Parcial del curso 2017-2018)

Ejecución de (una llamada/invocación a) un método

Registro de Activación y Pila de Llamadas (I)



Video1-S3:TestRAMActivo

- La JVM solo ejecuta **un método en cada momento: método Activo**
- Además de tener acceso a las zonas de memoria donde se sitúan los elementos accesibles para el código del método activo – en la zona NO-HEAP: su clase, otras de la aplicación que usa y sus variables de clase (**static**); en zona HEAP: los objetos que aparecen en su código– la JVM reserva una **zona de memoria extra** para almacenar los datos y resultados asociados a la ejecución de dicho código: **Registro de Activación**
- El Registro de Activación **contiene**:
 - Una variable **por cada variable local y por cada parámetro** que aparece en el código del método activo, obviamente de su mismo tipo ➡ Si el método es **dinámico**, contiene la variable **this**
 - Dos variables que **controla automáticamente** la JVM
 - a. Si el método devuelve un tipo de resultado distinto de **void**, la variable que almacena el **valor que devuelve (Valor de Retorno)** el método activo como resultado de su ejecución (**VR**), obviamente de su mismo tipo ➡ Si el método es un constructor, **VR** es **this**
 - b. La que almacena el **punto al que debe volver el control de la ejecución (Dirección de Retorno)** cuando acabe la ejecución del método activo (**DR**)
- El Registro de Activación **se libera al finalizar la ejecución del método activo al que está asociado**, tras recuperar de DR el punto al que retorna el control de la ejecución

Ejecución de (una llamada/invocación a) un método

Registro de Activación y Pila de Llamadas (II)



Video2-S3:TestPilaRA

¿En qué zona de la memoria se sitúa el RA asociado a un método activo?

Ni en el **HEAP** ni en el **NO-HEAP**, sino en una zona cuyo nombre determina la forma en que la JVM gestiona los diferentes RA asociados a los distintos métodos que se pueden invocar desde dentro de otro, de los cuales solo uno puede ser el método activo

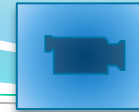
Cuando un método **mA** invoca a un método **mB**...

- La ejecución de **mA** queda en suspenso y se inicia la de **mB**, que pasa a ser el método activo
- El estado de **mA** se preserva en su Registro de Activación, que no se destruye hasta que se reanude y acabe su ejecución
- En memoria coexisten los RA de **mA** y **mB**, el del método activo **mB** (o RA activo) y el de **mA** que ha quedado en suspenso

¿Cómo gestiona la JVM los distintos RA que coexisten en memoria?

La Pila (*Stack*) de Llamadas

La Pila de Llamadas



Video2-S3:TestPilaRA

La JVM realiza una gestión LIFO (*Last In, First Out*) de los RA

Trata siempre **primero el último** RA que se ha creado, el asociado al método activo

- Cuando el método **mA** invoca al método **mB**, la JVM **"APILA"** el RA de **mB** sobre el de **mA** **(1)**, por lo que el RA asociado a **mB** ocupa el **"TOPE"** (o cima) de la Pila
- Cuando el método **mB** finaliza su ejecución, la JVM **"DESAPILA"** (libera) el RA de **mB** **(3)**, por lo que **mA** vuelve a ser el método activo y su RA ocupa el **"tope"** (o cima) de la Pila
- El método activo tiene acceso siempre al RA que ocupa el **"tope"** (o cima) de la Pila **(2)**

