Sesión 05: Terminología OO

Programación 2

1. Clases y Objetos

Ángel Herranz

2019-2020

Universidad Politécnica de Madrid

En capítulos anteriores. . .

- Clases y objetos (intro)
- Objetos, referencias y variables (y primitivos)
- Clases: plantilla para crear objetos

En capítulos anteriores. . .

- Clases y objetos (intro)
- Objetos, referencias y variables (y primitivos)
- Clases: plantilla para crear objetos

Encapsular

datos y comportamiento

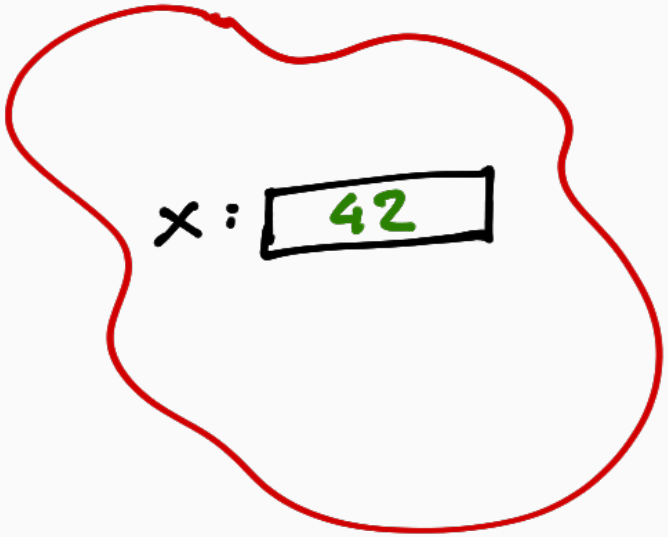
- Racionales

Objetos, referencias y variables

```
public class Sesion02 {
  public static void main(String args[]) {
    A a;
    a = new A();
  }
}
```

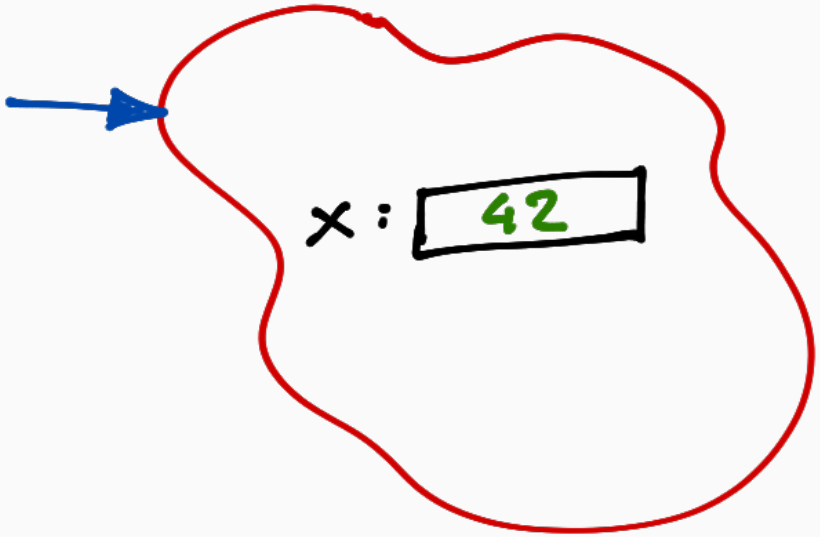
Objetos, referencias y variables

```
public class Sesion02 {
  public static void main(String args[]) {
    A a;
    a = new A();
  }
}
```



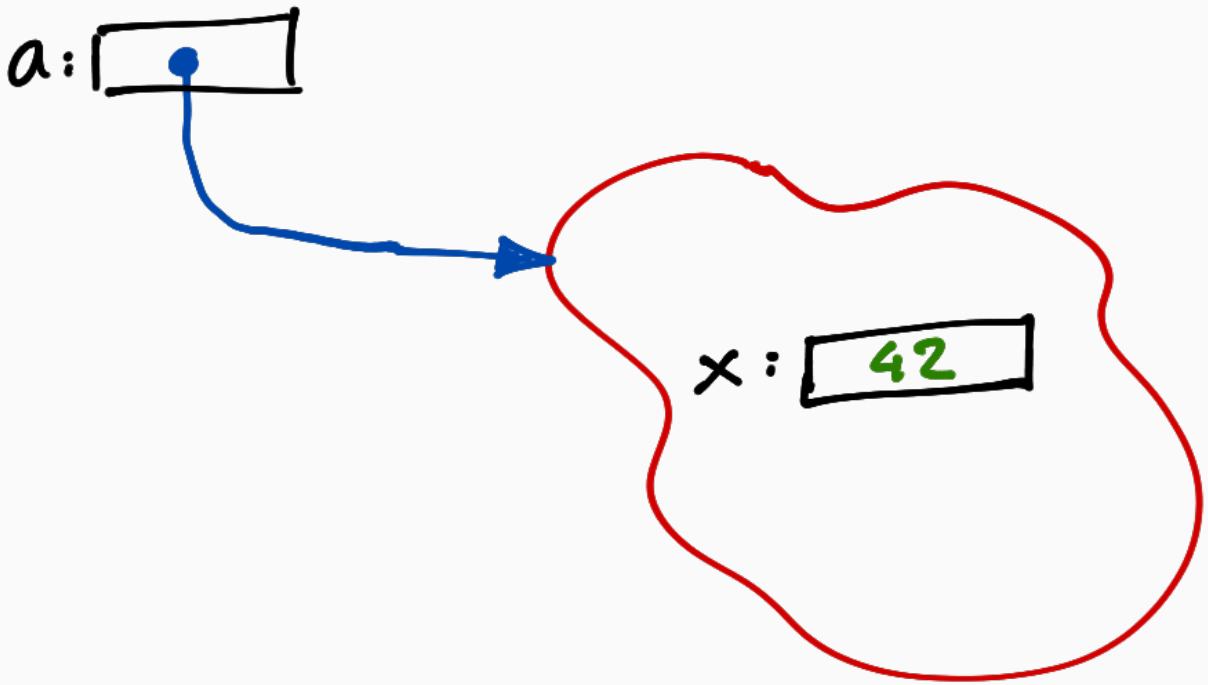
Objetos, referencias y variables

```
public class Sesion02 {
  public static void main(String args[]) {
    A a;
    a = new A();
  }
}
```



Objetos, referencias y variables

```
public class Sesion02 {
  public static void main(String args[]) {
    A a;
    a = new A();
  }
}
```



Clases: datos + comportamiento

```
class Cancion {
  String titulo;
  String interprete;
  int duracion;
  ...
  Cancion (String titulo,
           String interprete,
           int duracion) {
    this.titulo = titulo;
    this.interprete = interprete;
    this.duracion = duracion;
  }
  ...
  String duracion() {
    String minutos =
      duracion / 60;
    String segundos =
      duracion % 60;
    if (segundos.length() == 1)
      segundos
        = "0" + segundos;
    }
    return
      minutos + ":" + segundos;
  }
}
```

Clases: datos + comportamiento

```
class Cancion {
  String titulo;
  String interprete;
  int duracion;
  ...
  Cancion (String titulo,
           String interprete,
           int duracion) {
    this.titulo = titulo;
    this.interprete = interprete;
    this.duracion = duracion;
  }
  ...
  String duracion() {
    String minutos =
      duracion / 60;
    String segundos =
      duracion % 60;
    if (segundos.length() == 1)
      segundos
        = "0" + segundos;
    }
    return
      minutos + ":" + segundos;
  }
}
```

Clases: datos + comportamiento

```
class Cancion {
  String titulo;
  String interprete;
  int duracion;
  ...
  Cancion (String titulo,
           String interprete,
           int duracion) {
    this.titulo = titulo;
    this.interprete = interprete;
    this.duracion = duracion;
  }
  ...
  String duracion() {
    String minutos =
      duracion / 60;
    String segundos =
      duracion % 60;
    if (segundos.length() == 1)
      segundos
        = "0" + segundos;
    }
    return
      minutos + ":" + segundos;
  }
}
```

Clases: datos + comportamiento

```
class Cancion {
  String titulo;
  String interprete;
  int duracion;
  ...
  Cancion (String titulo,
           String interprete,
           int duracion) {
    this.titulo = titulo;
    this.interprete = interprete;
    this.duracion = duracion;
  }
  ...
  String duracion() {
    String minutos =
      duracion / 60;
    String segundos =
      duracion % 60;
    if (segundos.length() == 1)
      segundos
        = "0" + segundos;
    }
    return
      minutos + ":" + segundos;
  }
}
```

Clases: datos + comportamiento

```
class Cancion {
  String titulo;
  String interprete;
  int duracion;
  ...
  Cancion (String titulo,
           String interprete,
           int duracion) {
    this.titulo = titulo;
    this.interprete = interprete;
    this.duracion = duracion;
  }
  ...
  String duracion() {
    String minutos =
      duracion / 60;
    String segundos =
      duracion % 60;
    if (segundos.length() == 1)
      segundos
        = "0" + segundos;
    }
    return
      minutos + ":" + segundos;
  }
}
```

Terminología OO y Java

atributo, clase, constructor,
declaración, inicialización, instancia,
invocación, mensaje, miembro,
método, modificador, objeto,
observador, parámetro, puntero,
primitivo, referencia, tipo, variable

Terminología OO y Java

attribute, class, constructor,
declaration, initialization, instance,
invocation, message, member,
method, modifier, object,
observer, parameter, pointer,
primitive, reference, type, variable

Atributos y métodos

- Nadie usa los términos *datos* y *comportamiento*
- Datos: atributos (*attribute*)
- Comportamiento: métodos (*method*)
  - Constructor: mismo nombre que la clase
  - Observador: observa, devuelve algo, no modifica
  - Modificador modifica, no devuelve nada (**`void`**)

- Ámbos (datos y métodos): miembros (*member*)[2]

[2] Terminología muy Java, en concreto, *instance members*

Atributos y métodos

- Nadie usa los términos *datos* y *comportamiento*
- Datos: atributos (*attribute*)
- Comportamiento: métodos (*method*)
  - Constructor: mismo nombre que la clase
  - Observador: observa, devuelve algo, no modifica
  - Modificador modifica, no devuelve nada (**`void`**)
  - ¿Puede haber métodos *observamodificadores*?[1]
- Ámbos (datos y métodos): miembros (*member*)[2]

[1] Sí, pero hagamos caso a Bertrand Meyer y a su principio CQS (*Command Query Separation Principle*) y evitémoslos

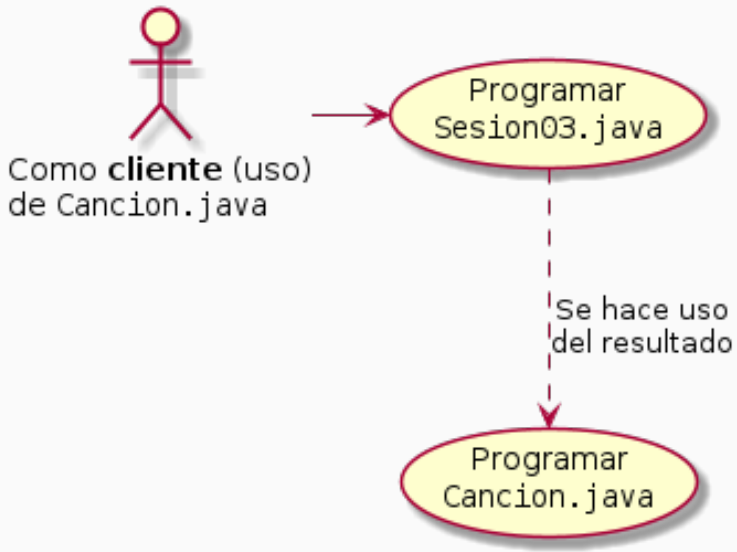
[2] Terminología muy Java, en concreto, *instance members*

Algunos sinónimos

$$\begin{aligned} \text{Objeto} &= \text{Instancia} \\ \text{Puntero} &= \text{Referencia} \\ \text{Método} &= \text{Mensaje} \\ \text{Invocar} &= \text{Enviar mensaje} \\ \text{Clase} &\subseteq \text{Tipo} \end{aligned}$$

Cuando usas código *de otro*

- No quieres saber cómo está hecho
- Sólo quieres saber qué hace
- No quieres repetir trabajo



Cuando *el otro* cambia el código

- Estás usando `Racional`, por ejemplo en `main`:

```
Racional r = new Raciona(1,3);
System.out.println(
  "El valor decimal es " + r.num / r.den
);
```

- El desarrollador de `Racional` renombra atributos `num` y `den` por `numerador` y `denominador`

Cuando *el otro* cambia el código

¿Qué va a pasar?

- Estás usando `Racional`, por ejemplo en `main`:

```
Racional r = new Raciona(1,3);
System.out.println(
  "El valor decimal es " + r.num / r.den
);
```

- El desarrollador de `Racional` renombra atributos `num` y `den` por `numerador` y `denominador`

Cuando tú eres el *el otro*

- Has desarrollado una clase para representar puntos en 2D

```
class Punto2D {
  double x;
  double y;
  ...
  void rotar(double rad) { ... // icomplejo! }
}
```

- Y ahora sabes que el método `rotar` va a ser usado masivamente por *código cliente*

Cuando tú eres el *el otro*

¿Qué vas a hacer?

- Has desarrollado una clase para representar puntos en 2D

```
class Punto2D {
  double x;
  double y;
  ...
  void rotar(double rad) { ... // icomplejo! }
}
```

- Y ahora sabes que el método `rotar` va a ser usado masivamente por *código cliente*

Cuando tú eres el *el otro*

¿Qué vas a hacer?

¿Qué va a pasar?

- Has desarrollado una clase para representar puntos en 2D

```
class Punto2D {
  double x;
  double y;
  ...
  void rotar(double rad) { ... // ¡complejo! }
}
```

- Y ahora sabes que el método `rotar` va a ser usado masivamente por *código cliente*

¿Ideas?

¿Qué vas a hacer?

1. Utilizar coordenadas polares para representar los puntos

```
class Punto2D {
  double radio;
  double angulo;
```

2. Implementar rotar con muy poco esfuerzo:

```
  void rotar(double rad) {
    angulo = angulo + rad;
  }
}
```

¿Qué va a pasar?

- Cualquier otro método implementado en la clase dejará de funcionar 👍

¿Qué va a pasar?

- Cualquier otro método implementado en la clase dejará de funcionar 👍
- Al modificar los atributos, cualquier código cliente de `Punto2D` dejará de compilar

¿Qué va a pasar?

- Cualquier otro método implementado en la clase dejará de funcionar 👍
- Al modificar los atributos, cualquier código cliente de `Punto2D` dejará de compilar

¡Vamos a sufrirlo!

Punto2D.java

```
class Punto2D {
  double x;
  double y;
  void rotar(double rad) {
    double x2, y2;
    x2 = x * Math.cos(rad) - y * Math.sin(rad);
    y2 = y * Math.cos(rad) - x * Math.sin(rad);
    x = x2;
    y = y2;
  }
}
```

Punto2D.java (usa la terminología)

clase, atributo, método (modificador), parámetro, variable

```
class Punto2D {
  double x;
  double y;
  void rotar(double rad) {
    double x2, y2;
    x2 = x * Math.cos(rad) - y * Math.sin(rad);
    y2 = y * Math.cos(rad) - x * Math.sin(rad);
    x = x2;
    y = y2;
  }
}
```

Sesion05.java

```
class Sesion05 {
  public static void main(String[] args) {
    Punto2D p = new Punto2D();
    p.x = 1.0;
    p.y = 0.0;
    System.out.format("( %.2f, %.2f)\n", p.x, p.y);
    p.rotar(Math.PI / 2.0);
    System.out.format("( %.2f, %.2f)\n", p.x, p.y);
  }
}
```

Sesion05.java (usa la terminología)

variable, objeto (instancia), método (constructor), atributo, método (modificador), parámetro

```
class Sesion05 {
  public static void main(String[] args) {
    Punto2D p = new Punto2D();
    p.x = 1.0;
    p.y = 0.0;
    System.out.format("(%.2f, %.2f)\n", p.x, p.y);
    p.rotar(Math.PI / 2.0);
    System.out.format("(%.2f, %.2f)\n", p.x, p.y);
  }
}
```

`Punto2D` en coordenadas polares

1. Reescribir `Punto2D` usando coordenadas polares
2. Compilar y ejecutar

`private i`

Para evitar que una modificación en la representación interna de una clase afecta a los *códigos cliente*, es muy importante ocultar dicha representación

private i

Para evitar que una modificación en la representación interna de una clase afecta a los *códigos cliente*, es muy importante ocultar dicha representación

```
class Punto2D {
          double x;
          double y;
  ...
}
```

```
public static void
  main(String[] args) {
  Punto2D p =
    new Punto2D();
  p.x = 1.0;
  p.y = 0.0;
}
```

private i

Para evitar que una modificación en la representación interna de una clase afecta a los *códigos cliente*, es muy importante ocultar dicha representación

```
class Punto2D {
  private double x;
  private double y;
  ...
}
```

```
public static void
  main(String[] args) {
  Punto2D p =
    new Punto2D();
  p.x = 1.0; 👎 No compila
  p.y = 0.0; 👎 No compila
}
```

`private` ii

- Aparece antes del tipo en atributos y métodos
- Impide que el código cliente tenga acceso al atributo o al método
- Hay otros *modificadores* que iremos viendo:

  `private`, **`public`**, **`protected`**, o nada

Modificar `Sesion05`

- Crear dos instancias de `Punto2D` a y b:

$$a = (0, 0)$$
$$b = (1, 0)$$

- Imprimir las dos puntos tras cada cambio
- Modificar la ordenada (eje Y) de a: ponerla a 1
- Imprimir la ordenada de a
- Rotar 45º el punto a
- Imprimir la distancia entre a y b

Buenas prácticas i

En lugar de exponer los atributos se define un API[3] de acceso a los objetos

- `Punto2D()`: crea un punto $(0, 0)$ en coordenadas cartesianas
- `Punto2D(x,y)`: crea un punto (x, y) en coordenadas cartesianas (abscisa y ordenada)
- `x()`: devuelve la abscisa
- `y()`: devuelve la ordenada

[3] *Application Public Interface*

Buenas prácticas ii

- `cambiarX(double x)`: modifica la abscisa
- `cambiarY(double y)`: modifica la ordenada
- `distancia(Punto2D b)`: devuelve un double que representa la distancia desde **`this`** a `b`
- `rotar(double a)`: rota el punto alrededor del $(0, 0)$ la cantidad de `a` radianes

Buenas prácticas ii

- `cambiarX(double x)`: modifica la abscisa
- `cambiarY(double y)`: modifica la ordenada
- `distancia(Punto2D b)`: devuelve un double que representa la distancia desde **`this`** a `b`
- `rotar(double a)`: rota el punto alrededor del $(0, 0)$ la cantidad de `a` radianes

¡Implementar la buena práctica!