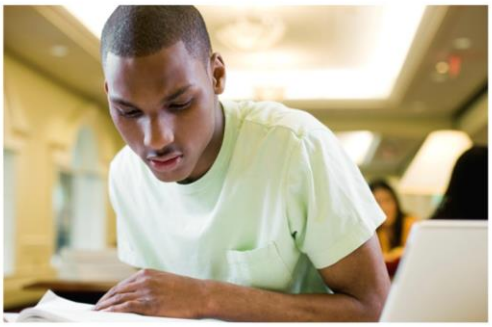
ORACLE

Academy

Java Foundations

7-4

Sobrecarga de métodos



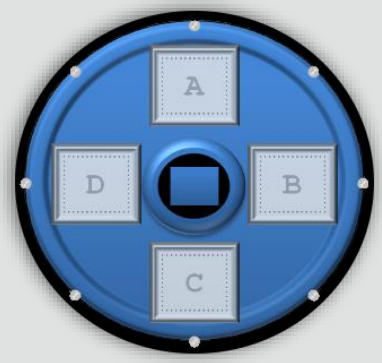
ORACLE
Academy

Objetivos

- En esta lección se abordan los siguientes objetivos:
  - Comprender los efectos de varios constructores en una clase
  - Definir la sobrecarga de un método
  - Explicar la firma de método
  - Comprender cuándo es posible la sobrecarga y cuándo no

Ejercicio 1

- Juegue al rompecabezas básico 8
  - https://objectstorage.uk-london-1.oraclecloud.com/n/lrvrlgaqj8dd/b/Games/o/JavaPuzzleBall/index.html
- Tenga en cuenta lo siguiente:
  - ¿Qué puede decir sobre las luces que rodean a cada rueda?



¿Por qué agregamos luces a las ruedas?

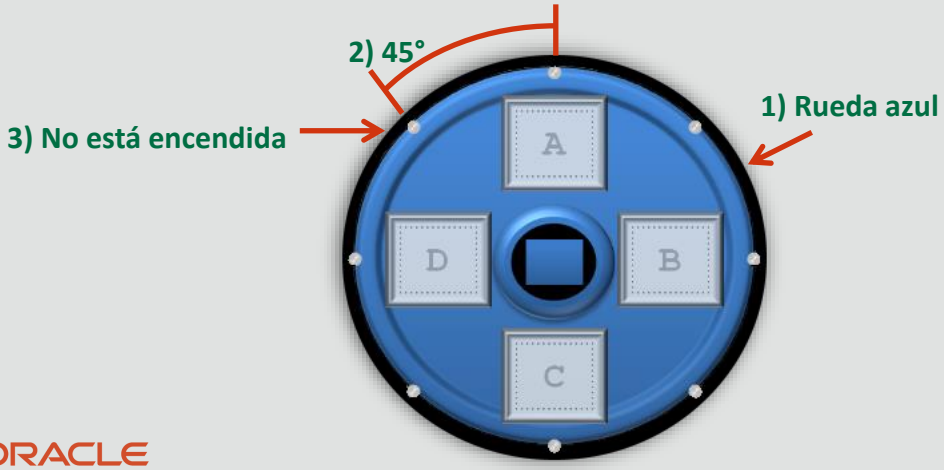
- Las compilaciones anteriores no incluían estas luces
  - No formaban parte del diseño original
  - ¿Por qué se han agregado?
- Se han agregado para evitar confusiones al jugador
  - Algunos jugadores no sabían que la rueda se ajustaría al ángulo de 45° más próximo
  - Algunos jugadores tenían que girar la rueda varias veces antes de que llegara al siguiente incremento de 45°
  - Esto provocaba confusión y frustración ya que los jugadores pensaban: **“La rueda no ha girado como yo quería”**

Plan para resolver estos problemas

- Agregar ocho luces a cada rueda
  - Las luces actúan de “marca”
  - Muestran cada incremento de 45° donde se debe ajustar la rueda
- Se puede iluminar una sola luz, lo que indica:
  - La rotación donde se ha obtenido la rueda
  - La rotación a la que se ajustará la rueda si se suelta

Propiedades de luz

- Una luz necesita las siguientes propiedades:
  - La rueda a la que pertenece
  - Su rotación alrededor de la rueda
  - Si se debe iluminar



Programación de la clase Light

- A continuación encontrará una versión simplificada de esta clase:

```
public class UIWheelLight {
   //Fields
   public UIWheel wheel;
   public double rotation;
   public boolean isLit;

   //Constructor
   public UIWheelLight(UIWheel w, double r, boolean l){
      wheel = w;
      rotation = r;
      isLit = l;
   } //end Constructor
} //end class UIWheelLight
```

Llamada al constructor UIWheelLight

- Una llamada al constructor podría parecerse a la siguiente:

```
UIWheelLight light1 = new UIWheelLight(blueWheel, 45, false);
```

- Pero luego pensamos: **“Me da mucha pereza escribir todo eso”**
  - Hay un motivo legítimo para ello
  - No se debe a que no seamos buenos programadores
  - No se debe a que no seamos inteligentes

Por qué es estupendo ser perezoso

- Con unos pocos cálculos matemáticos sabemos que...
  - Hay ocho luces en una rueda
  - Aparecerá una luz adicional encendida
  - 8/9 (o el 89%) de las luces se instanciarán apagadas
  - El 89% es una mayoría importante
- Por lo tanto, el argumento final es redundante y complicará el código el 89% del tiempo
- El código complicado no es positivo y se debe reducir

```
UIWheelLight light1 = new UIWheelLight(blueWheel, 45, false);
```

Redundante

Las luces realmente no se encienden y apagan. En su lugar, cuando una luz debe estar "encendida", instanciamos una novena luz y la colocamos encima de la correspondiente luz "apagada".

Sobrecarga de constructores

- Puede escribir más de un constructor en una clase
  - Esto se conoce como sobrecargar un constructor
  - Una clase puede tener un número ilimitado de constructores
- Cada constructor sobrecargado tiene el mismo nombre
- Pero difieren en alguna de las formas siguientes:
  - Número de parámetros
  - Tipos de parámetros
  - Orden de parámetros

Constructores sobrecargados: Ejemplo

- La implantación de esta estrategia en la clase UIWheelLight se parece a la siguiente:

```
public class UIWheelLight {
    …
   //Constructors
   public UIWheelLight(UIWheel w, double r){
       wheel = w;
       rotation = r;
       isLit = false;
   } //end Constructor

   public UIWheelLight(UIWheel w, double r, boolean l){
      wheel = w;
      rotation = r;
      isLit = l;
   } //end Constructor
} //end class UIWheelLight
```

2 parámetros

3 parámetros

Llamada a constructores sobrecargados

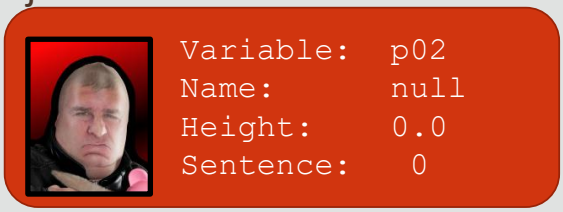
- Se puede instanciar un objeto llamando a cualquiera de sus constructores de clase
- Proporcione los argumentos y Java encontrará el constructor más adecuado

```
UIWheelLight light1 = new UIWheelLight(blueWheel, 45);
```

```
UIWheelLight light1 = new UIWheelLight(blueWheel, 45, false);
```

Ejercicio 2

- Continúe con la edición del proyecto `PrisonTest`
  - Se proporciona una versión de este programa en los archivos `PrisonTest_Student_7_4.java` y `Prisoner_Student_7_4.java`
- Sobrecargue el constructor existente
  - Cree su propio constructor sin argumentos
  - Al llamar a este constructor se deben inicializar los campos con los valores siguientes
  - Instancie un objeto con este constructor



Reconocimiento de redundancia en los constructores

- En estos constructores se repite un código muy parecido
- Es posible minimizar esta redundancia

```
public class UIWheelLight {
   //Constructors
   public UIWheelLight(UIWheel w, double r){
      wheel = w;
      rotation = r;
      isLit = false;
   }//end constructor

   public UIWheelLight(UIWheel w, double r, boolean l){
      wheel = w;
      rotation = r;
      isLit = l;
   }//end constructor
}//end class UIWheelLight
```

Primera incidencia

Repetición

Los constructores pueden llamar a otros constructores

- Con la palabra clave this, un constructor puede llamar a otro

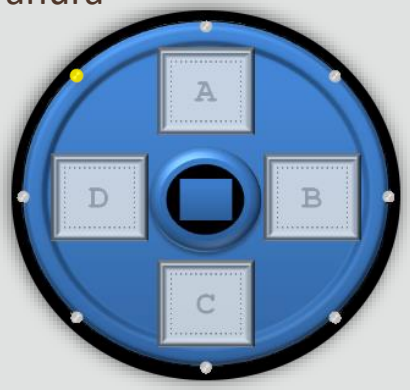
```
public class UIWheelLight {
   //Constructors
   public UIWheelLight(UIWheel w, double r){
      this(w, r, false);
   }//end constructor

   public UIWheelLight(UIWheel w, double r, boolean l){
      wheel = w;
      rotation = r;
      isLit = l;
   }//end constructor
}//end class UIWheelLight
```


Esto resulta útil porque, si se debe cambiar la lógica de un constructor, solo es necesario cambiarla en un único lugar.

Comportamiento de la luz

- Según el lugar donde se haga clic, la luz amarilla se comporta de un modo ligeramente distinto
  - Si hace clic en la rueda, la luz se posiciona según la ubicación del cursor del mouse
  - Si hace clic en la ranura A, B, C o D, la luz se posiciona según el centro de esa ranura



¿Cómo hemos programado esta diferencia sutil en el comportamiento?

- Hemos sobrecargado el método responsable de posicionar la luz amarilla
- El código se parece al siguiente:

```
public class UIWheelLight {
   …
   public void setPosition(double x, double y){
      //Do math
   }//end method setPosition

   public void setPosition(double x, double y, UISlot s){
      //Do slightly different math
   }//end method setPosition
}//end class UIWheelLight
```


x e y son las posiciones x e y donde se ha hecho clic con el mouse.

Sobrecarga de métodos

- Se puede sobrecargar cualquier método, incluidos...
  - Constructores
  - Métodos que modelan comportamientos de objetos
  - Métodos que realizan cálculos
- Todas las versiones de un método sobrecargado tienen el mismo nombre
- Pero difieren en alguna de las formas siguientes:
  - Número de parámetros
  - Tipos de parámetros
  - Orden de parámetros

Número de parámetros

- Cada uno de los métodos sobrecargados siguientes tiene un número de parámetros distinto

```
public class Calculator {

   public double sum(double num1){
      return num1;
   }//end method sum

   public double sum(double num1, double num2){
      return num1 + num2;
   }//end method sum

   public double sum(double num1, double num2, double num3){
      return num1 + num2 + num3;
   }//end method sum

}//end class Calculator
```

Tipo de parámetros

- Cada uno de los siguientes métodos sobrecargados tiene parámetros de tipos distintos

```
public class Calculator {

   public double sum(double num1, double num2){
      return num1 + num2;
   }//end method sum

   public double sum(int num1, int num2){
      return num1 + num2;
   }//end method sum

}//end class Calculator
```

Orden de parámetros

- Cada método sobrecargado tiene parámetros en un orden diferente

```
public class Calculator {

   public double sum(int num1, double num2){
      return num1 + num2;
   }//end method sum

   public double sum(double num1, int num2){
      return num1 + num2;
   }//end method sum

}//end class Calculator
```

Llamada a métodos sobrecargados

- Proporcione los argumentos y Java encontrará el método más adecuado

```
public class CalculatorTest{

   public static void main(String[] args){
      Calculator calc = new Calculator();

      calc.sum(1, 2);
      calc.sum(1, 2, 3);
      calc.sum(1.5, 4.5);
   }//end method main

}//end class CalculatorTest
```

Ejercicio 3

- Continúe con la edición del proyecto `PrisonTest`
- Escriba un método que imprima cada campo de Prisoner
  - Debe ser un método sin argumentos
- Sobrecargue este método para aceptar un argumento booleano
  - Si el valor booleano es true, este método debe llamar al método think()
- Llame ambas versiones de este método en un objeto

Reconocimiento de la redundancia en los métodos

- En estos métodos se repite un código muy parecido
- Es posible minimizar esta redundancia

```
public class Calculator{
   …
   public double calcY(double m, double x){
      double y = 0;
      y = mx;
      return y; ;
   }//end method calcY
   public double calcY(double m, double x, double b){
      double y = 0;
      y = mx + b;
      return y;
   }//end method calcY
}//end class Calculator
```

Primera incidencia

Repetición

Nota para los instructores: El código de estas dos diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Los métodos pueden llamar a otros métodos de la misma clase

- En este ejemplo, un método devuelve un valor al otro

```
public class Calculator{
   …
   public double calcY(double m, double x){
      return calcY(m,x,0);
   }//end method calcY

   public double calcY(double m, double x, double b){
      double y = 0;
      y = mx + b;
      return y;
   }//end method calcY
}//end class Calculator
```


Esto resulta útil porque, si los cálculos son erróneos o se deben ajustar, el código solo se tiene que cambiar una vez.

Nota para los instructores: El código de estas dos diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Ejercicio 4

- Continúe la edición del proyecto `PrisonerTest`
- Identifique y minimice el código repetido en el constructor y los métodos display()
- Ejecute el programa para asegurarse de que sigue funcionando correctamente

Firma de método

- Una firma de método se crea a partir de...
  - Nombre del método
  - Número de parámetros
  - Tipo de parámetros
  - Orden de parámetros
- Siempre que uno de estos elementos difiera, una firma de método será única

Se trata de la firma de método

```
public void setPosition(double x, double y){
   //Do math
}//end method setPosition
```


Nota para los instructores: El código de estas dos diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Elementos que no incluye la firma de método

- La firma de método no incluye…
  - Nombre de parámetros
  - Tipo de retorno de método
- Cambiar cualquier de estos elementos no es suficiente para sobrecargar un método

No forman parte de la firma de método

```
public void setPosition(double x, double y){
   //Do math
} //end method setPosition
```


Nota para los instructores: El código de estas dos diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Correspondencia de llamadas de método y firmas

- En este ejemplo el recuento permite saber a qué versión de sum() se debe llamar
- La llamada de método tiene tres argumentos
- ¿Qué firma de método tiene tres parámetros?

```
    sum(1, 2, 3);
```

```
public class Calculator {

   public double sum(double num1, double num2){
      return num1 + num2;
   }//end method sum
   public double sum(double num1, double num2, double num3){
      return num1 + num2 + num3;
   }//end method sum
}//end class Calculator
```


Nota para los instructores: El código de estas cuatro diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Nombres de parámetro no coincidentes

- ¿Puede saber a qué versión de sum() se debe llamar si los nombres de parámetro son distintos?
- No se puede saber
- Java tampoco puede saberlo

```
sum(1, 2);
```

```
public class Calculator {

   public double sum(double num1, double num2){
      return num1 + num2;
   }//end method sum
   public double sum(double x, double y){
      return x + y;
   }//end method sum
}//end class Calculator
```


Nota para los instructores: El código de estas cuatro diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Tipos de retorno no coincidentes

- ¿Puede saber a qué versión de sum() se debe llamar si los tipos de retorno son distintos?
- No
- Java tampoco puede saberlo

```
     sum(1, 2);
```

```
public class Calculator {

   public double sum(double num1, double num2){
      return num1 + num2;
   }//end method sum
    public int sum(double num1, double num2){
      return num1 + num2;
   }//end method sum
}//end class Calculator
```


Nota para los instructores: El código de estas cuatro diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Sobrecarga en primer lugar

- Los métodos no están sobrecargados correctamente hasta que sus firmas difieren
- Si se cumple esta condición, puede modificar el tipo de retorno y los nombres de parámetro

```
    sum(1, 2);
```

```
public class Calculator {

   public double sum(double num1, double num2){
      return num1 + num2;
   }//end method sum
   public int sum(double num1, double num2, double num3){
      return num1 + num2 +num3;
   }//end method sum
}//end class Calculator
```


Nota para los instructores: El código de estas cuatro diapositivas debe estar en la misma posición para que sus diferencias se hagan más evidentes.

Resumen de sobrecarga de métodos

- Tienen el mismo nombre
- Tienen firmas diferentes:
  - El número de parámetros
  - Los tipos de parámetros
  - El orden de parámetros
- Pueden tener funcionalidades diferentes o similares

Resumen

- En esta lección, debe haber aprendido lo siguiente:
  - Comprender los efectos de varios constructores en una clase
  - Definir la sobrecarga de un método
  - Explicar la firma de método
  - Comprender cuándo es posible la sobrecarga y cuándo no


ORACLE
Academy