

The logo for Oracle Academy is centered on a light gray background. It features the word "ORACLE" in a bold, orange, sans-serif font. Below it, the word "Academy" is written in a smaller, dark gray, sans-serif font. The entire logo is framed by a thin black border, with dark gray horizontal bars at the top and bottom.

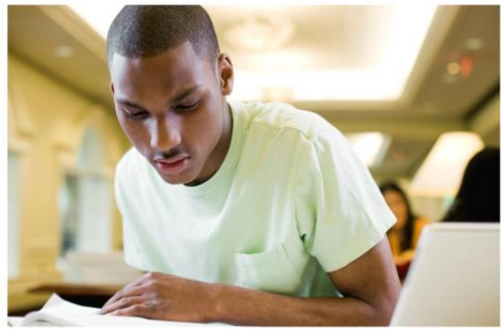
ORACLE

Academy

Java Foundations

7-4 Overloading Methods

ORACLE
Academy



Copyright © 2022, Oracle and/or its affiliates. Oracle, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Objectives

- This lesson covers the following objectives:
 - Understand the effects of multiple constructors in a class
 - Define overloading of a method
 - Explain the method signature
 - Understand when overloading is and isn't possible





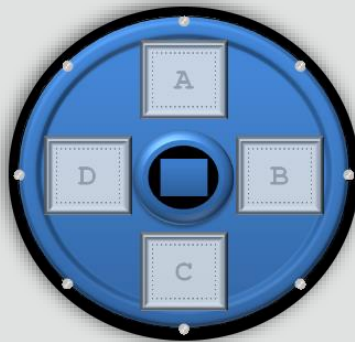
Exercise 1

- Play Basic Puzzle 8

- <https://objectstorage.ca-toronto-1.oraclecloud.com/n/yzr73ksbiwdp/b/Games/o/JavaPuzzleBall/index.html>

- Consider the following:

- What can you say about the lights surrounding each wheel?



Why Did We Add Lights to the Wheels?

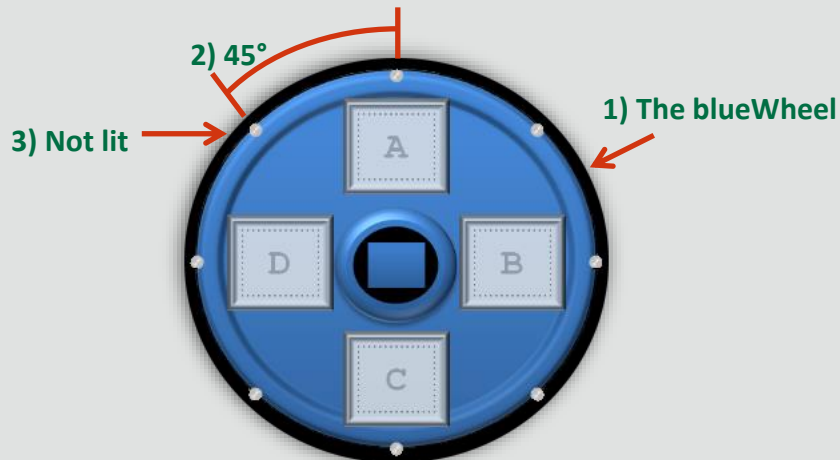
- Earlier builds didn't include these lights
 - They were never part of the original design
 - So why were they added?
- Lights were added to address player confusion
 - Some players didn't realize that the wheel would snap to the nearest 45° angle
 - Some players needed to rotate the wheel several times before they reached the next 45° increment
 - This caused confusion and frustration as players thought **"The wheel didn't rotate where I wanted it to"**

The Plan to Solve These Issues

- Add eight lights to each wheel
 - Lights act as a “tick” mark
 - They show each 45° increment where the wheel could snap
- A single light may brighten, which shows:
 - The rotation where the wheel was grabbed
 - The rotation where the wheel will snap if it’s released

Light Properties

- A light requires the following properties:
 - The wheel it belongs to
 - Its rotation around that wheel
 - If it should be lit



Programming the Light Class

- Here's a simplified version of this class:

```
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
```


Calling the UIWheelLight Constructor

- A constructor call would look something like this:

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

- But then we thought: **“I’m too lazy to type all that!”**
 - There’s a legitimate reason for this
 - It isn’t because we’re bad programmers
 - It isn’t because we’re stupid



Why It's Great to Be Lazy

- A little math told us ...
 - There are eight lights on a wheel
 - One additional light will appear lit
 - 8/9 (or 89%) of lights will be instantiated unlit
 - 89% is a substantial majority
- Therefore, the final argument is redundant and will complicate code 89% of the time
- Complicated code is bad and should be minimized

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

Redundant

Lights don't actually turn on and off. Instead, when a light should be "on," we instantiate a ninth light and position it on top of the corresponding "off" light.

Overloading Constructors

- You can write more than one constructor in a class
 - This is known as overloading a constructor
 - A class may have an unlimited number of constructors
- Each overloaded constructor is named the same
- But they differ in any of the following ways:
 - Number of parameters
 - Types of parameters
 - Ordering of parameters

Overloaded Constructors: Example

- Implementing this strategy in the UIWheelLight class looks something like this:

```
public class UIWheelLight { 2 parameters
    ...
    //Constructors
    public UIWheelLight(UIWheel w, double r){
        wheel = w;
        rotation = r;
        isLit = false;
    } //end Constructor 3 parameters

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

Calling Overloaded Constructors

- An object may be instantiated by calling any of its class constructors
- You supply the arguments, and Java finds the most appropriate constructor

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

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

Exercise 2

- Continue editing the `PrisonTest` project
 - A version of this program is provided for you in the files `PrisonTest_Student_7_4.java` and `Prisoner_Student_7_4.java`
- Overload the existing constructor
 - Create your own zero-argument constructor
 - Calling this constructor should initialize fields with the following values
 - Instantiate an object with this constructor



Variable: p02
Name: null
Height: 0.0
Sentence: 0

Recognizing Redundancy in Constructors

- Very similar code is repeated in these constructors
- It's possible to minimize this redundancy

```
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
```

First occurrence

Repeated

Constructors Can Call Other Constructors

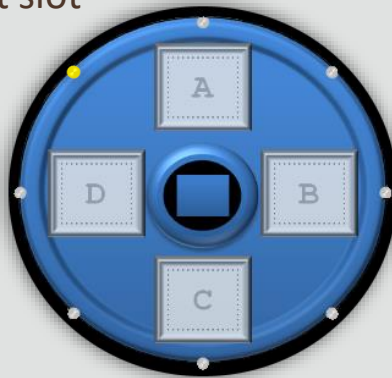
- By using the **this** keyword, one constructor may call another

```
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
```

This is helpful because, if the logic in a constructor needs to change, then the code needs to be changed only in one place.

Light Behavior

- Depending on where you click, the yellow light behaves slightly differently
 - If you click the wheel, the light is positioned based on the location of the mouse cursor
 - If you click slot A, B, C, or D, the light is positioned based on the center of that slot



How Did We Program This Subtle Difference in Behavior?

- We overloaded the method responsible for positioning the yellow light
- The code looks similar to this:

```
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 and y are the x-position and y-position where the mouse was clicked.

Overloading Methods

- Any method can be overloaded, including ...
 - Constructors
 - Methods that model object behaviors
 - Methods that perform calculations
- All versions of an overloaded method are named the same
- But differ in any of the following ways:
 - Number of parameters
 - Types of parameters
 - Ordering of parameters

Number of Parameters

- Each overloaded method below has a different number of parameters

```
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
```

Type of Parameters

- Each overloaded method below has parameters of different types

```
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
```

Order of Parameters

- Each overloaded method has parameters in a different order

```
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
```

Calling Overloaded Methods

- You supply the arguments, and Java finds the most appropriate method

```
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
```

Exercise 3

- Continue editing the `PrisonTest` project
- Write a method that prints every `Prisoner` field
 - This should be a zero-argument method
- Overload this method to accept a boolean argument
 - If the boolean is true, this method should call the `think()` method
- Call both versions of this method on an object

Recognizing Redundancy in Methods

- Very similar code is repeated in these methods
- It's possible to minimize this redundancy

```
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
```

First occurrence

Repeated

Methods Can Call Other Methods in the Same Class

- In this example, one method returns a value to the other

```
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
```

This is helpful because, if the calculations are wrong or need to be adjusted, then the code needs to be changed only once.

Exercise 4

- Continue editing the `PrisonTest` project
- Identify and minimize any repeated code in the constructor and `display()` methods
- Run the program to make sure the program still works properly

The Method Signature

- A method signature is created from the ...
 - Name of the method
 - Number of parameters
 - Type of parameters
 - Order of parameters
- As long as one of these differ, a method's signature will be unique

This is the method signature

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

Not the Method Signature

- The method signature does not include ...
 - Name of parameters
 - Method return type
- Changing either of these isn't enough to overload a method

These aren't part of the method signature

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

Matching Method Calls to Signatures

- In this example, counting makes it easy to see which version of `sum()` should be called
- The method call has three arguments
- Which method signature has three parameters?

```
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
```

Not Matching Parameter Names

- Can you tell which version of `sum()` should be called if the parameter names differ?
- You can't
- And neither can Java

```
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
```

Not Matching Return Types

- Can you tell which version of `sum()` should be called if the return types differ?
- No
- And neither can Java

```
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
```


Overload First

- Methods aren't properly overloaded until their signatures differ
- When this is true, then you're welcome to modify the return type and parameter names

```
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
```

Overloading Methods Summary

- Have the same name
- Have different signatures:
 - The number of parameters
 - The types of parameters
 - The order of parameters
- May have different functionality or similar functionality

Summary

- In this lesson, you should have learned how to:
 - Understand the effects of multiple constructors in a class
 - Define overloading of a method
 - Explain the method signature
 - Understand when overloading is and isn't possible



