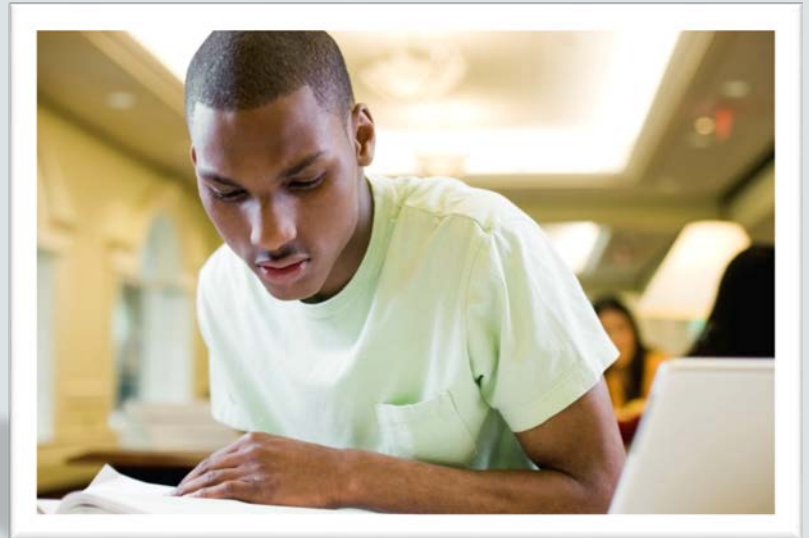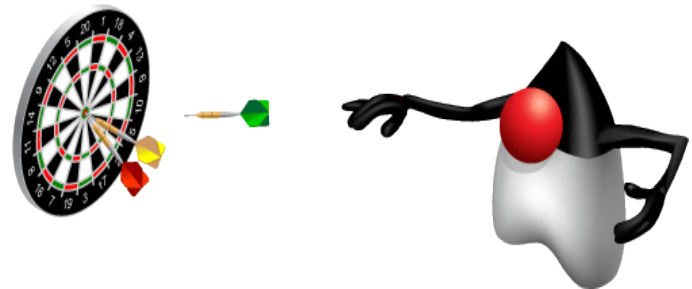# Java Foundations

7-4
Overloading Methods

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

Topics

- **Overloading Constructors**
- Overloading Methods
- When Overloading Isn't Possible

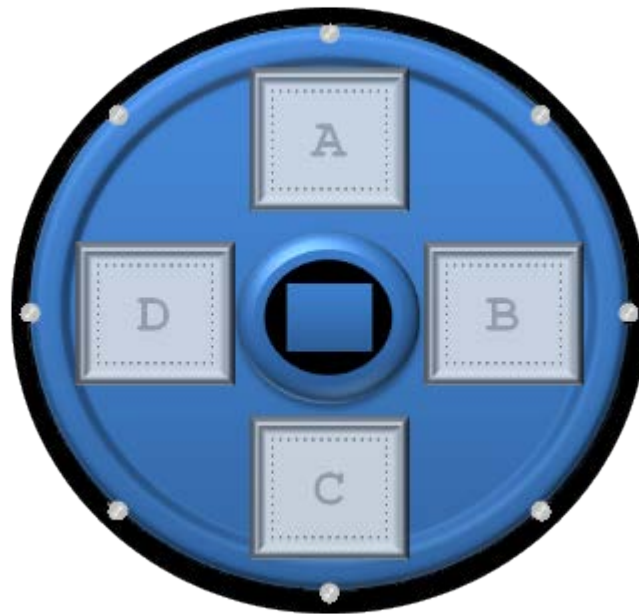| Creating a Class | Instantiating Objects | Constructors | Overloading Methods | Object Interaction and Encapsulation | Static Variables and Methods |

Section 7

Exercise 1

- Play **Basic Puzzle 8.**

- 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:

1. The wheel it belongs to
2. Its rotation around that wheel
3. If it should be lit

2) 45°

3) Not lit

1) The blueWheel
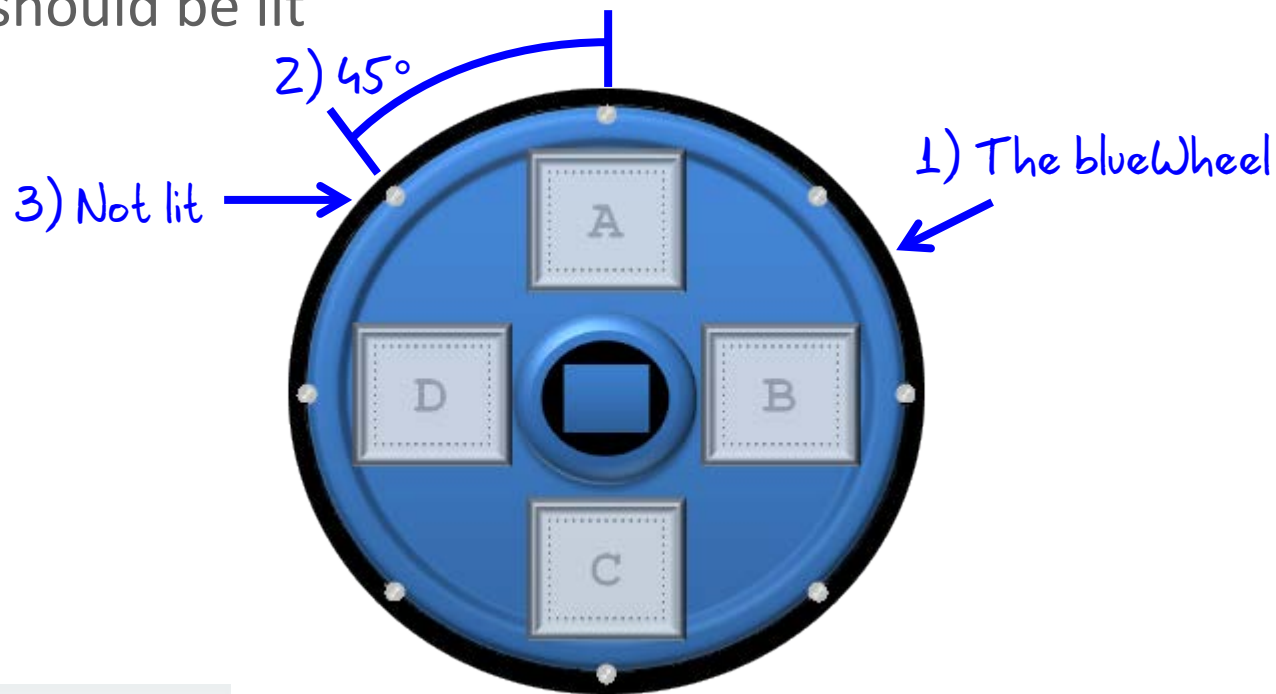
Programming the Light Class

Here's a simplified version of this class:

```java
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;
    }
}
```

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.

ORACLE® ACADEMY

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

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:

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

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

2 parameters (annotation pointing to `UIWheel w, double r`)

3 parameters (annotation pointing to `UIWheel w, double r, boolean l`)

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.
- 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
```
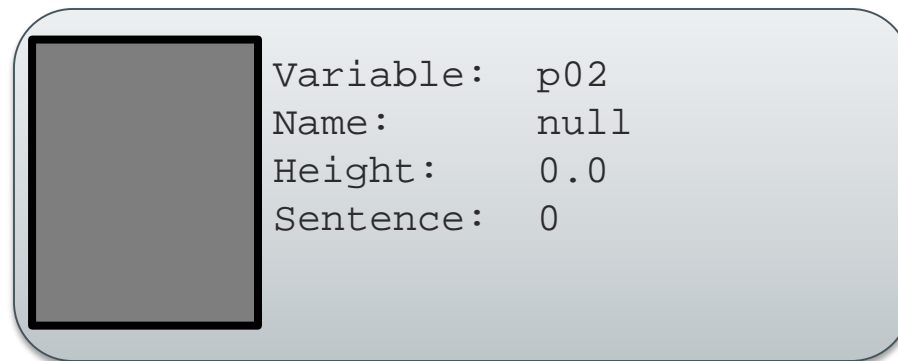
Recognizing Redundancy in Constructors

- Very similar code is repeated in these constructors.

- It's possible to minimize this redundancy.

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

    public UIWheelLight(UIWheel w, double r, boolean l){
        wheel = w;
        rotation = r;           Repeated
        isLit = l;
    }
}
```

Constructors Can Call Other Constructors

By using the `this` keyword, one constructor may call another.

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


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

ORACLE® ACADEMY

Topics

- Overloading Constructors

- Overloading Methods

- When Overloading Isn't Possible

| Creating a Class | Instantiating Objects | Constructors | Overloading Methods | Object Interaction and Encapsulation | Static Variables and Methods |

Section 7

JFo 7-4
Overloading Methods

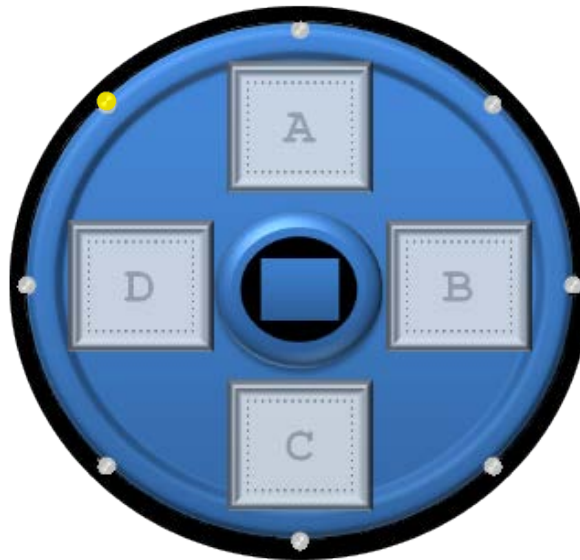Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

18

Light Behavior

Depending on where you click, the yellow light behaves slightly differently.

1. If you click the wheel, the light is positioned based on the location of the mouse cursor.

2. If you click slot A, B, C, or D, the light is positioned based on the center of that slot.

ORACLE® ACADEMY

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:

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

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

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.

```java
public class Calculator {

    public double sum(double num1){
        return num1;
    }
    public double sum(double num1, double num2){
        return num1 + num2;
    }
    public double sum(double num1, double num2, double num3){
        return num1 + num2 + num3;
    }
}
```

JFo 7-4
Overloading Methods

Type of Parameters

Each overloaded method below has parameters of different types.

```java
public class Calculator {

    public double sum(double num1, double num2){
        return num1 + num2;
    }
    public double sum(int num1, int num2){
        return num1 + num2;
    }
}
```

Order of Parameters

Each overloaded method has parameters in a different order.

```java
public class Calculator {

    public double sum(int num1, double num2){
        return num1 + num2;
    }
    public double sum(double num1, int num2){
        return num1 + num2;
    }
}
```

Calling Overloaded Methods

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

```java
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);
    }
}
```

ORACLE® ACADEMY

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.

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

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);
    }


    public double calcY(double m, double x, double b){
        double y = 0;
        y = mx + b;
        return y;
    }
}
```

Exercise 4

- Continue editing the PrisonerTest project.

- Identify and minimize any repeated code in the constructor and `display()` methods.

- Run the program to make sure the program still works properly.

ORACLE **ACADEMY**

Topics

- Overloading Constructors

- Overloading Methods
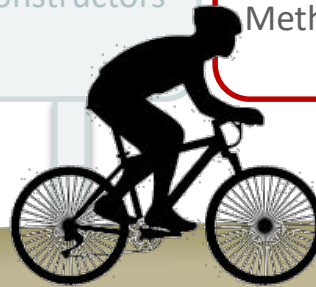
- When Overloading Isn't Possible

| Creating a Class | Instantiating Objects | Constructors | Overloading Methods | Object Interaction and Encapsulation | Static Variables and Methods |

Section 7

ORACLE ACADEMY

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
}
```

ORACLE ACADEMY

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
}
```

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);
```

```java
 public class Calculator {

    public double sum(double num1, double num2){
       return num1 + num2;
    }
    public double sum(double num1, double num2, double num3){
       return num1 + num2 + num3;
    }
}
```

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);
```

```java
public class Calculator {

    public double sum(double num1, double num2){
        return num1 + num2;
    }
    public double sum(double x, double y){
        return x + y;
    }
}
```

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);
```

```java
public class Calculator {

    public double sum(double num1, double num2){
        return num1 + num2;
    }
    public int sum(double num1, double num2){
        return num1 + num2;
    }
}
```

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);
```

```java
public class Calculator {

    public double sum(double num1, double num2){
        return num1 + num2;
    }
    public int sum(double num1, double num2, double num3){
        return num1 + num2 +num3;
    }
}
```

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

ORACLE **ACADEMY**

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