Java Puzzle Ball
Nick Ristuccia

Lesson 4-2
Lambda Expressions
Exercise 4

• Play **Lambda Puzzles 1 through 7.**
  – Destroy Blue Bumpers
  – Preserve Red Bumpers

• Consider the following:
  – Can you identify use-cases for lambda expressions?
  – Can you figure out how the logic operators work?

You're welcome to play beyond puzzle 7
Lambda Use-Case 1

• Lambda expressions handle mouse and keyboard input.

• `blade` is a field in the Ball class. It's also a special object type.
  – Yes, an object can be used as a field.

• `.setOnMousePressed()` is a method this object type is capable of.
  – It tells the instance to listen in the Event the mouse is pressed.

• An Event is special class in Java. One is created by the action of clicking the mouse on `blade`. The Event is represented by `e`.

```java
blade.setOnMousePressed(
e -> setDirection(Dir.NE));
```
Lambda Use-Case 1 continued

• `setDirection()` is a method defined in the Ball class.
  – Instead of accepting a numeric value, this method accepts a special `Dir` value.

• `Dir` is a direction **enumerator**. The 8 possible `Dir` values are defined elsewhere and were specially written for this game.

• Put this all together, and you get your observation: Clicking the blade changes the direction it's moving to the one you set in the lambda expression. This is presented as a single line of code.
Does this Sound Complicated?

• The previous slides are heavy on technical detail.
• The details may be more helpful later as reference material as you start playing with lambda expressions in NetBeans.
• The important thing for now is that you understand the use case conceptually.
One Other Observation

• You can't change the `Dir` value while the ball is moving across the screen.
  – This rule was implemented for the sake of accuracy.

• Values that are explicitly written in code are considered **hard coded**.
  – You can't change hard coded values in NetBeans while your program is running.
  – You have to restart the program for your changes to take effect.
  – Give this a try in Lab 4.
Lambda Use Case 2

• Lambda expressions allow you to easily work with a collection of objects.
• As you play, you perform the same logic as the lambda code:
  – Take a collection of many bumpers.
  – Identify bumpers based on certain properties (color).
  – For each bumper matching that criteria, perform an action on it (Destroy/Preserve).
What is the Java Syntax?

• The collection is called `BumperList`.

• **Filter** through objects in the collection based on their properties:
  – Shape, color, number
  – If the criteria matches, the object passes through the filter.

• Call a method **for each** instance that matches the filter criteria.
  – `b` represents any given bumper instance in the collection.

```java
BumperList.stream()
  .filter(b -> b.getShape() == Shape.STAR)
  .forEach(b -> b.setShape(Shape.RECT));
```
Other Aspects You May Have Noticed, Part 1

• Lambda.05: A forEach statement can contain multiple methods.

• To do this...
  – Enclose all the methods in a set of curly braces {    }
  – Put a semicolon ; at the end of each method call.
Other Aspects You May Have Noticed, Part 2

• Lambda.06: Compound logic (&&, ||) can be applied to filters.
  – When using an &&, both criteria must be true for an object to pass through the filter.
  – When using an ||, either criteria can be true for an object to pass through the filter.

```java
.filter(b ->

  b.getNum() == 1
  ||
  b.getColor() == Color.RED
)
```
Other Aspects You May Have Noticed, Part 3

• Lambda.07: Multiple filters can be chained together.
  – This is the same as using 

• == checks to see if a property is equal to a value.

• != checks to see if a property is not equal to the value.
Summary of Use Case 2

- **Lambda.05**: A `forEach` block can contain multiple methods.

- **Lambda.06**: Compound logic (`&&`, `||`) can be applied to filters.

- **Lambda.07**: Multiple filters can be chained together.
How Useful is this?

• I pull up the game whenever I need to remember how to write lambdas.

• But the game could convey **functional programming** better:
  – Take a lambda expression.
  – Save it as a variable.
    • Reference the variable in methods like you would any field.
    • Pass the variable or logic between methods like you would any number.
  – Functional programming is about storing and passing around functionality and logic.
Example Using Lambda Logic as a Field Variable

• The lambda expression is stored as the field variable `lambdaExample`.  
  – This type of lambda expression is called a Consumer.  
  – There are many more types.

```java
public class SomeClass {
    // Example using lambda logic as a field variable
    private Consumer<ImageView> lambdaExample = (e -> setDirection(Dir.NE));
    ImageView blade;

    // Method using lambdaExample
    public void someMethod() {
        blade.setOnMousePressed(lambdaExample);
    }

    // Method to set direction
    public void setDirection(Dir dir) {
        // ... (implementation details)
    }
}
```
Example Passing Lambda Logic to a Method

Although using a field variable is probably better, you can pass logic directly.

```java
public class SomeClass {
    ImageView blade;
    
    public void someMethod() {
        someOtherMethod(e -> setDirection(Dir.NE));
    }

    public void someOtherMethod(Consumer x) {
        blade.setOnMousePressed(x);
    }

    public void setDirection(Dir dir) {
        ...
    }
}
```

Although using a field variable is probably better, you can pass logic directly.
How is Functional Programming Useful?

• It makes your programming more flexible.
• It makes your programming easier to maintain
• If you make a mistake and need to change your logic, you can change it in one place (the variable) instead of searching for each situation where the same logic is repeated.
  – You might miss one.
  – It's also tedious.

```java
Consumer<ImageView> lambdaExample = (e -> setDirection(Dir.NE));
```