JavaFX

Tutorial 9 (14th April 2021)

Lambda expressions

- Interface: set of abstract methods
- Interface is also a (reference) *type*.
- Implementation of an interface: concrete class implementing (all) abstract methods
- Lambda expression: implementation of an interface + instantation
  - the interface should contain only one abstract method: a functional interface
- Like any other (reference) value, a lambda expression has a type, namely the interface it implements

Lambda expressions: examples

```
interface Callable<V> {
      V call();
}
```

```
interface Function<T,R> {
      R apply(T t);
}
```

```
Callable<Double> doubleFun = () -> 3.0;
Function<String,Integer> strToInt = s -> Integer.valueOf(s);
```

Lambda expressions: examples (2)

```
Function<String,Integer> strToInt = s -> Integer.valueOf(s);
```

is short for

```
public class StrToInt implements Function<String, Integer> {
    @Override
    public Integer apply( String str ) {
        return Integer.valueOf(str);
    }
}
```

```
Function<String,Integer> strToInt = new StrToInt ();
```

Lambda expressions: examples (3)

```
Function<String,Integer> strToInt = s -> Integer.valueOf(s);
```

also short for

```
Function<String,Integer> strToInt = new Function<>() {
    @Override
    public Integer apply(String s) {
        return Integer.valueOf(s);
    }
};
```

Lambda expressions: examples (4)

```
Function<String,Integer> strToInt = s -> Integer.valueOf(s);
```

this can be abbreviated to

```
Function<String,Integer> strToInt = Integer::valueOf;
```

Using lambda expressions

```
Callable<Double> doubleFun = () -> 3.0;
Function<String,Integer> strToInt = s -> Integer.valueOf(s);
```

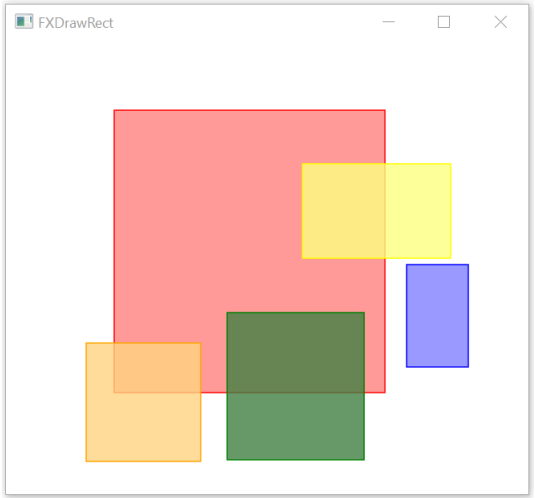
```
System.out.println( doubleFun.call() );
System.out.println( strToInt.apply("43") );
```

lambda expressions: capturing variables

```
Double d = 2.0;
Callable<Double> doubleFun = () -> 3.0 + d;
```

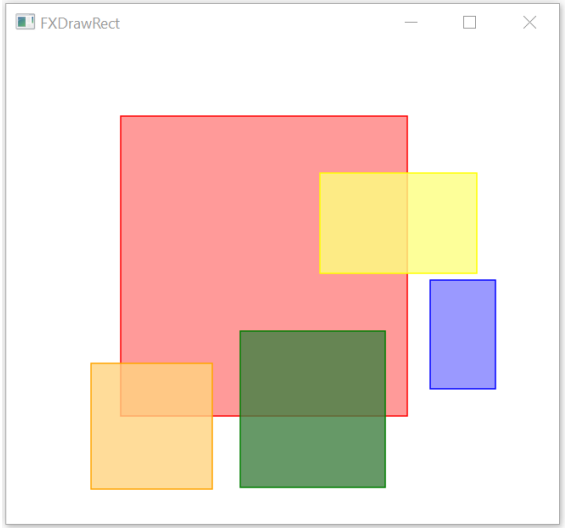
```
Double d = 2.0;
Callable<Double> doubleFun = () -> 3.0 + d;
d = d + 2;
System.out.println( doubleFun.call() );
```

mouse events: drawing rectangles




```
public class FXDrawRect extends Application {
  private static final double MIN_SIZE = 5;
  private static final Color[] colors = {Color.RED,…};
  private int currentColorIx;
  private Rectangle currentRect;
  private double currentXOffset, currentYOffset;
  @Override
  public void start(Stage stage){
    Pane pane = new Pane();
    pane.setOnMousePressed(e -> newRect(pane, e));
    pane.setOnMouseDragged(e -> dragRect(e));
    pane.setOnMouseReleased(e -> finishRect(e));
    Scene scene = new Scene(pane, 300, 250);
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(scene);
    stage.show();
  }
```

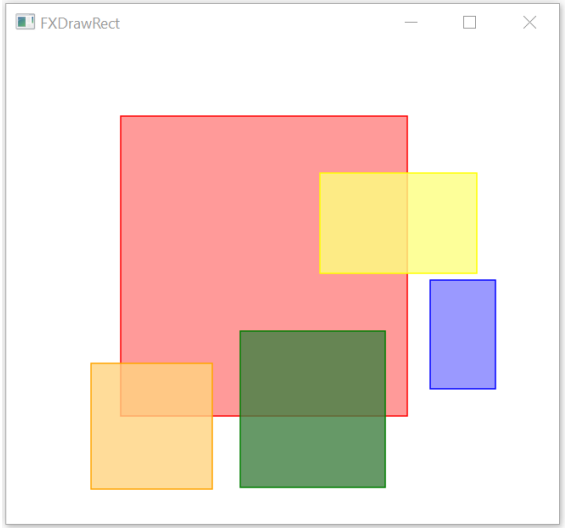
dragging a rectangle



resize the newly created rectangle

```
private void dragRect(MouseEvent e) {
  double newWidth  = max(e.getX() - currentRect.getX(), MIN_SIZE);
  double newHeight = max(e.getY() - currentRect.getY(), MIN_SIZE);
  currentRect.setWidth(newWidth);
  currentRect.setHeight(newHeight);
}
```

finish a rectangle




```
private void finishRect (MouseEvent e) {
  Rectangle thisRect = currentRect;
   thisRect.setOnMousePressed(e2 -> {
     if (e2.isShiftDown()) {
         thisRect.toFront();
     }
     currentXOffset = e2.getX() - thisRect.getX();
     currentYOffset = e2.getY() - thisRect.getY();
     e2.consume(); });
  thisRect.setOnMouseDragged(e2 -> {
    thisRect.setX(e2.getX() - currentXOffset);
    thisRect.setY(e2.getY() - currentYOffset);
    e2.consume(); });
}
```

Finish a rectangle (with anonymous inner classes)

```
private void finishRectAlt( MouseEvent e ) {
    Rectangle rect = currentRect;
    currentRect.setOnMousePressed( new EventHandler<MouseEvent>() {
        @Override
        public void handle( MouseEvent e2 ) {
            if (e2.isShiftDown()) {
                r.toFront();
            }
            currentXOffset = e2.getX() - rect.getX();
            currentYOffset = e2.getY() - rect.getY();
            e2.consume(); // stop propagation up
        }
    });

    currentRect.setOnMouseDragged(new EventHandler<MouseEvent>() {
        @Override
        public void handle( MouseEvent e2 ) {
            rect.setX(e2.getX() - currentXOffset);
            rect.setY(e2.getY() - currentYOffset);
            e2.consume(); // stop propagation up
        }
    });

}
```

Finish a rectangle (with inner classes)

```
private void finishRectAlt2(MouseEvent e) {
    currentRect.setOnMousePressed( new StartDraggingHandler( currentRect ) );
    currentRect.setOnMouseDragged( new ContinueDraggingHandler(currentRect ) );
}

private class StartDraggingHandler implements EventHandler<MouseEvent> {
    Rectangle myRect;

    public StartDraggingHandler(Rectangle myRect) {
        this.myRect = myRect;
    }

    @Override
    public void handle(MouseEvent e2) {
        if (e2.isShiftDown()) {
            myRect.toFront();
        }
        currentXOffset = e2.getX() - myRect.getX();
        currentYOffset = e2.getY() - myRect.getY();
        e2.consume();
    }
}
```

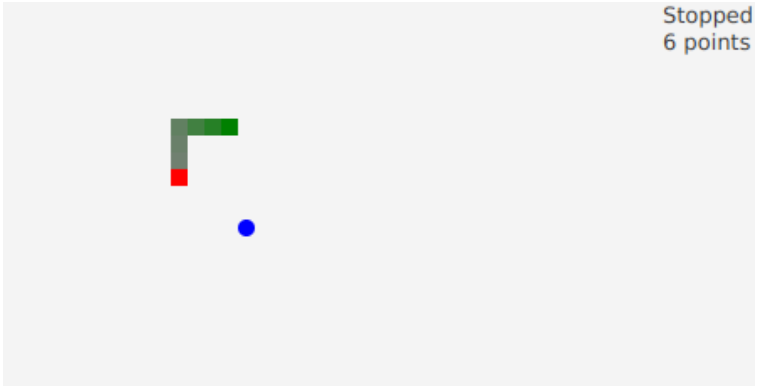
Finish a rectangle (with static nested classes)

```
private void finishRectAlt2(MouseEvent e) {
    rect.setOnMousePressed( new StaticStartDraggingHandler( rect, this ) );
    rect.setOnMouseDragged( new ContinueDraggingHandler( rect ) );
}

private static class StaticStartDraggingHandler implements EventHandler<MouseEvent> {
    Rectangle myRect;
    JavaFXDrawRect myOuterClass;

    public StartDraggingHandler(Rectangle myRect, JavaFXDrawRect myOuterClass) {
        this.myRect      = myRect;
        this.myOuterClass = myOuterClass;
    }

    @Override
    public void handle(MouseEvent e2) {
        if (e2.isShiftDown()) {
            myRect.toFront();
        }
        myOuterClass.currentXOffset = e2.getX() - myRect.getX();
        myOuterClass.currentYOffset = e2.getY() - myRect.getY();
        e2.consume();
    }
}
```

This week’s assigment: Snake

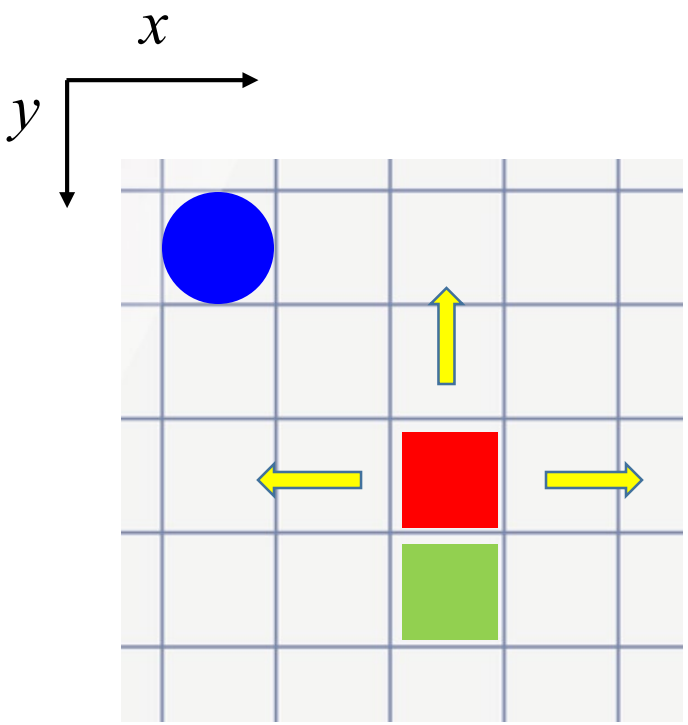
Provided code

- `Direction`
- `Food`
- `Snake`
- `World`
- `Main`



Direction

```
public enum Direction {
    Up(0,-1), Right(1,0), Down(0,1), Left(-1,0);

    private final int dX, dY;
    private Direction( int dX, int dY ) {
        this.dX = dX;
        this.dY = dY;
    }
    public int getdX() {
        return dX;
    }
    public int getdY() {
        return dY;
    }
}
```

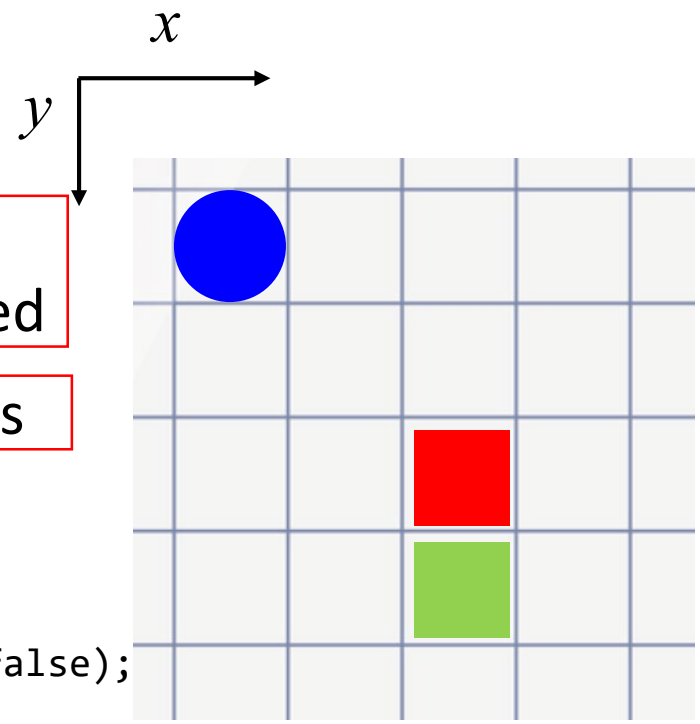


World

```
public class World {
    public final static int DELAY = 200;

    private final int size;

    private final Snake snake;
    private final Food food;

    private final BooleanProperty running = new SimpleBooleanProperty(false);
    private final IntegerProperty score = new SimpleIntegerProperty(0);

    public World( int size ) {
        this.size = size;

        snake = new Snake(size / 2, size / 2, this);
        food = new Food();

        // TODO: Implement timeline

        moveFoodRandomly();
    }
```

determines movement speed

number of cells



Segment

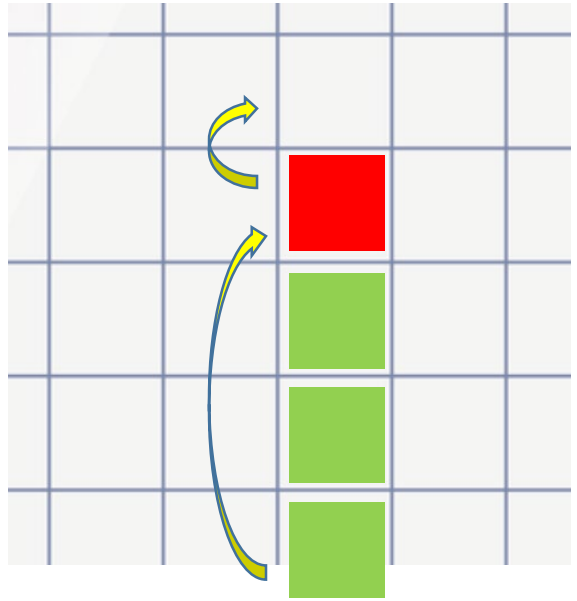
```
private class Segment {
    private final IntegerProperty x, y;

    public Segment(int x, int y) {
        this.x = new SimpleIntegerProperty(x);
        this.y = new SimpleIntegerProperty(y);
    }

    public void setPosition(int x, int y) {
        this.x.setValue(x);
        this.y.setValue(y);
    }
```

Snake

```
public class Snake extends Segment {
  private Direction direction = Direction.RIGHT;
  private final World world;

  private final List<Segment> body = new LinkedList<>();
  private final List<SnakeSegmentListener> listeners = new LinkedList<>();
```

Snake: movement

```
public class Snake extends Segment {
    <…>
    public void move() {
        int newX = getX() + direction.getDX();
        int newY = getY() + direction.getDY();
        // Bitten itself or hit the border => game over
        // Food eaten => new segment
        // Normal movement

            } else {
                if (! body.isEmpty() ) {
                    Segment tail = body.removeFirst();
                    body.addLast(tail);
                    tail.setPosition(getX(), getY());
                }
            }
```



Finally