VOXXED DAYS ISTANBUL

Java 8: New and Noteworthy

Nosheen Zaza

Università della Svizzera italiana

VOXXED DAYS ISTANBUL

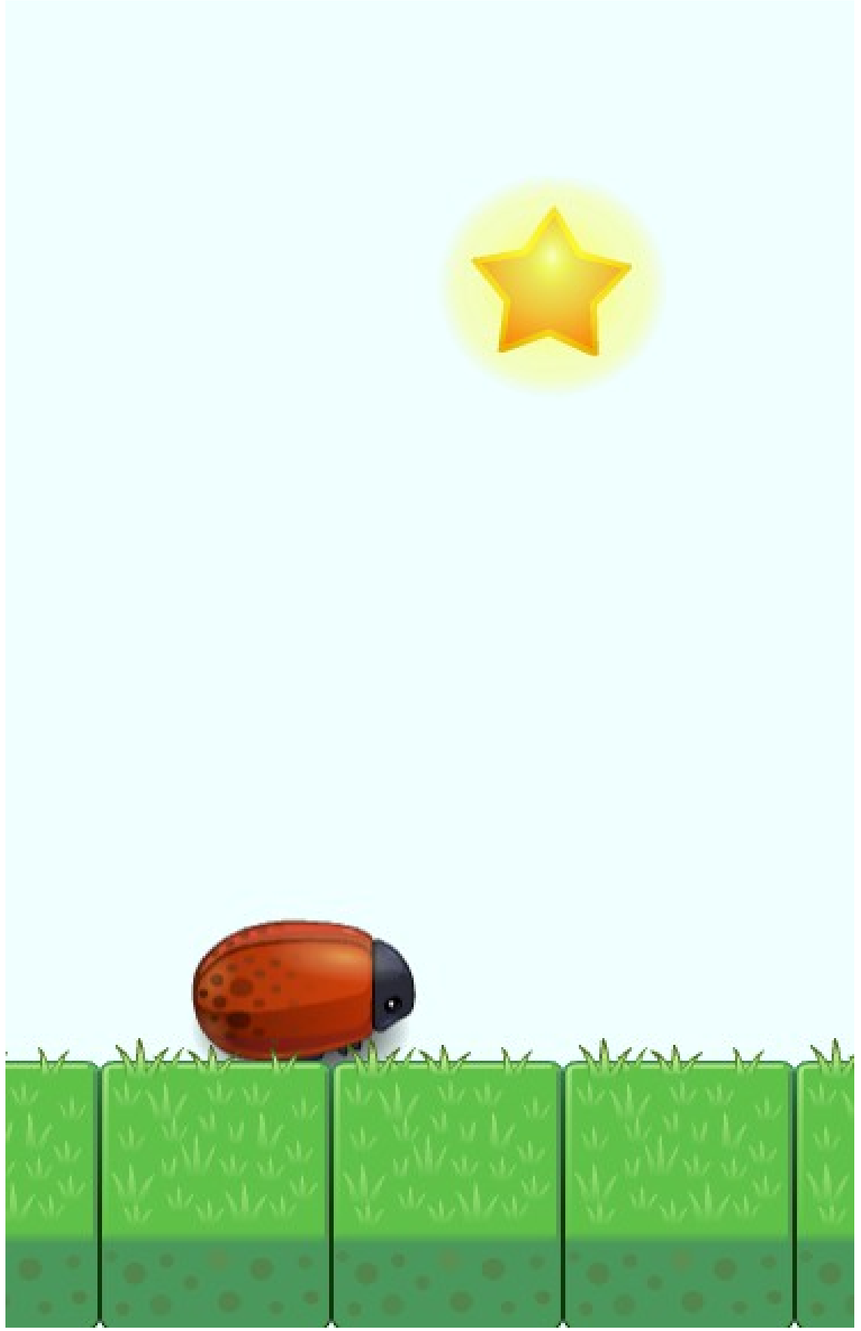
I am...

- A PhD. student at the faculty of informatics, Università della Svizzera italiana in Lugano-Switzerland.
- I work in the Lugano Language Lab (λ3)
  - New programming languages, language extensions, compiler frameworks, code mining, language features for distributed applications programming.

This talk

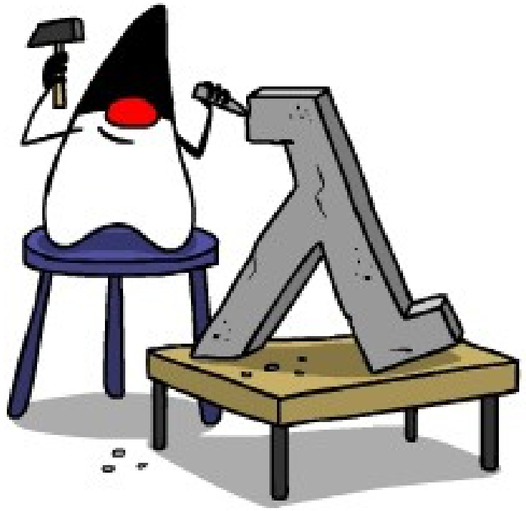
- New language and library features of Java 8.
- Focus on lambdas and streams.
- Sample application: a Bug's life.
  - ~~Stolen~~ adapted and extended from a version used to demonstrate the use of reactive streams in Scala.
  - github.com/Applied-Duality/RxGame
- Find game code at

  github.com/nosheenzaza/bug-life-java8



Java 8

- Major changes to the language, libraries and the framework as a whole.
- On language level, the impact of changes on the way you program as much or more than that since the introduction of generics.
- Facilitate functional-style programming through lambdas and streams.



Java 8

- Major changes to the language, libraries and the framework as a whole.
- On language level, the impact of changes on the way you program as much or more than that since the introduction of generics.
- Facilitate functional-style programming through lambdas and streams.

Functional Programming

- Has roots in **Lambda** Calculus
- Everything is a function (vs. everything is an object.)
- Apply functions on other functions or values.
- Avoid mutable state and side effects.
- A paradigm older than OOP.
- Trending with the promise of facilitating parallelism
  - Easier when shared state is not mutable and in absence of side effects.

Alonzo Church (1903-1995)

Functional Programming

- Has roots in **Lambda** Calculus
- Everything is a function (vs. everything is an object)
- Apply functions on other functions or values.
- Avoid mutable state and side effects.
- A paradigm older than OOP.
- Trending with the promise of facilitating parallelism
  - Easier when shared state is not mutable and in absence of side effects.

Alonzo Church (1903-1995)

Functions as Parameters Pre Java 8 (without admitting it)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
   new EventHandler<KeyEvent>() {
      public void handle(KeyEvent event) {
         if (event.getCode() == KeyCode.SPACE)
            bug.jump();
   }
});
```

Functions as Parameters Pre Java 8 (without admitting it)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
   new EventHandler<KeyEvent>() {
      public void handle(KeyEvent event) {
         if (event.getCode() == KeyCode.SPACE)
            bug.jump();
   }
});
```

```
public interface EventHandler<T extends Event> extends EventListener {
    void handle(T event);
}
```

Functions as Parameters Pre Java 8 (without admitting it)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
   new EventHandler<KeyEvent>() {
      public void handle(KeyEvent event) {
         if (event.getCode() == KeyCode.SPACE)
            bug.jump();
   }
});
```

```
public interface EventHandler<T extends Event> extends EventListener {
    void handle(T event);
}
```

Why can't we just pass the function?

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

Lambda Expression

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
   new EventHandler<KeyEvent>() {
      public void handle(KeyEvent event) {
         if (event.getCode() == KeyCode.SPACE)
            bug.jump();
   }
});
```

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
   new EventHandler<KeyEvent>() {
      public void handle(KeyEvent event) {
         if (event.getCode() == KeyCode.SPACE)
            bug.jump();
   }
});
```

The compiler knows that x is of type KeyEvent

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

```
public interface EventHandler<T extends Event> extends EventListener {
    void handle(T event);
}
```

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

```
public interface EventHandler<T extends Event> extends EventListener {
    void handle(T event);
}
```

The body of method 'handle'

What if the interface has more than one abstract method?

Functions as Parameters: Java 8 (embrace and exploit!)

```
scene.addEventHandler(KeyEvent.KEY_PRESSED,
        x -> {if (x.getCode() == KeyCode.SPACE) bug.jump();});
```

```
@FunctionalInterface
public interface EventHandler<T extends Event> extends EventListener {
    void handle(T event);
}
```

Lambda Syntax

```
(int x, int y) -> x * y

(x, y) -> x * y

() -> 42

(String s) -> { System.out.println(s);}
```

Lambdas in Depth

```
public class Game extends javafx.application.Application {
   ....
   @Override
   public void start(final Stage stage) {
        ...

        Scene scene = new Scene(root);
        scene.addEventHandler(KeyEvent.KEY_PRESSED,
            event ->
             {if (event.getCode() == KeyCode.SPACE)
              bug.jump();});
        ...
   }
   ...
}
```

Lambdas in Depth

```
public class Game extends javafx.application.Application {
   ....
   @Override
   public void start(final Stage stage) {
        ...

        Scene scene = new Scene(root);
        scene.addEventHandler(KeyEvent.KEY_PRESSED,
            event ->
             {if (event.getCode() == KeyCode.SPACE)
              this.??? // what does 'this' here refer to?
              bug.jump();});
        ...
   }
   ...
}
```

Lambdas in Depth

```
public class Game extends javafx.application.Application {
   ....
   @Override
   public void start(final Stage stage) {
        ...

        Scene scene = new Scene(root);
        scene.addEventHandler(KeyEvent.KEY_PRESSED,
            event ->
             {if (event.getCode() == KeyCode.SPACE)
              Scene scene = new Scene(root);
              bug.jump();});
        ...
   }
   ...
}
```

Lambdas in Depth

```
public class Game extends javafx.application.Application {
   ....
   @Override
   public void start(final Stage stage) {
        ...

        Scene scene = new Scene(root);
        scene.addEventHandler(KeyEvent.KEY_PRESSED,
            event ->
             {if (event.getCode() == KeyCode.SPACE)
              scene = new Scene(root);
              bug.jump();});
        ...
   }
   ...
}
```

java.util.function package

- Provide target types for lambda expressions for commonly used function shapes.
- You can define your own as well when needed.

java.util.function package

- Provide target types for lambda expressions for commonly used function shapes.
- You can define your own as well when needed.

```
IntToDoubleFunction f1 = x -> x / 3.0;
f1.applyAsDouble(4);

BiConsumer<String, Integer> f2 = (x, y) ->
System.out.println(x + " " + y);
f2.accept("Hello", 3);

Predicate<String> f3 = String::isEmpty;
f3.test("String");
```

java.util.function package

- Provide target types for lambda expressions for commonly used function shapes.
- You can define your own as well when needed.

```
IntToDoubleFunction f1 = x -> x / 3.0;
f1.applyAsDouble(4);

BiConsumer<String, Integer> f2 = (x, y) ->
System.out.println(x + " " + y);
f2.accept("Hello", 3);

Predicate<String> f3 = String::isEmpty;
f3.test("String");
```

Method references

```
Math::sqrt                x -> Math.sqrt(x)

Object::toString          obj -> obj.toString()

Obj::toString             () -> obj.toString()

Object::new               () -> new Object()
```

Collections and Lambda Support

- With lambdas in hand, many powerful methods can be defined.
- Collections framework is extended to support lambdas.
- How does this extension not break already existing binaries?
- ***Default methods in interfaces.***

Default Methods in Iterable

```
public interface Iterable<T> {
    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
```

Default Methods in Collection

```
public interface Collection<E> extends Iterable<E> {
   default boolean removeIf(Predicate<? super E> filter) {
         Objects.requireNonNull(filter);
         boolean removed = false;
         final Iterator<E> each = iterator();
         while (each.hasNext()) {
             if (filter.test(each.next())) {
                 each.remove();
                 removed = true;
             }
         }
         return removed;
    }

   default Stream<E> stream() {
         return StreamSupport.stream(spliterator(), false);
    }

    default Stream<E> parallelStream() {
         return StreamSupport.stream(spliterator(), true);
    }
```

Default Methods: Multiple inheritance?

```
public interface A {
    default void foo(){
        System.out.println("Calling A.foo()");
    }
}
public interface B {
    default void foo(){
        System.out.println("Calling B.foo()");
    }
}
public class C implements A, B {
}
```

Default Methods

```
public interface A {
    default void foo(){
        System.out.println("Calling A.foo()");
    }
}
public interface B {
    default void foo(){
        System.out.println("Calling B.foo()");
    }
}
public class C implements A, B {
}
```

```
Compiler error: duplicate default methods named foo with
the parameters () and () are inherited from the types
Game.B and Game.A
```

Default Methods

```
public interface A {
    default void foo(){
        System.out.println("Calling A.foo()");
    }
}
public interface B {
    default void foo(){
        System.out.println("Calling B.foo()");
    }
}
public class C implements A, B {
     public void foo(){
        A.super.foo();
     }
   }
```

Streams

- A new abstraction of a data collection.
- Declarative : What to Do vs. How to Do.
- Lazy: stream items are created on demand.
- Internal Iteration.
- Direct parallelization.

Creating a stream

- From any collection:

```
tiles.stream();
```

- From known items:

```
Stream.of(new Grass(root), sun, bug)
```

- Numeric Range:

```
IntStream.range(0, 8);
```

- From a generator function:

```
Stream.generate(Math::random);
```

- Based on previous items in the stream:

```
Stream.iterate(1, x -> x + 1);
```

Operations on Streams

```
Stream.iterate(1 , x -> x + 1)
    .map(x -> x * 2)
    .filter(x -> x % 3 == 0)
    .limit(1000)
    .collect(Collectors.groupingBy(x -> x > 30))
    .forEach((x, y) -> System.out.println(x + ": " + y));
```

Operations on Streams

```
Stream.iterate(1 , x -> x + 1)
    .map(x -> x * 2)
    .filter(x -> x % 3 == 0)
    .limit(1000)
    .collect(Collectors.groupingBy(x -> x > 30))
    .forEach((x, y) -> System.out.println(x + ": " + y));
```

```
false: [6, 12, 18, 24, 30]

true: [36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96, 102,
108, 114, 120, 126, 132, 138, 144, 150, 156, 162, 168,
174, 180, 186, 192, 198, 204, 210, 216, 222, 228, 234,
240, 246, 252, 258, 264, 270, 276, 282, 288, 294, 300,
306, 312, 318, 324, 330, 336, 342, 348, 354, 360, 366,
372, 378, 384, 390, 396, 402, 408, 414, 420, 426, 432,
438, 444, 450, 456, 462, 468, 474, 480, 486, 492, 498,
504, 510, 516, 522, 528, 534, 540, 546, 552, 558, 564,
570, 576, 582, 588, 594, 600]
```

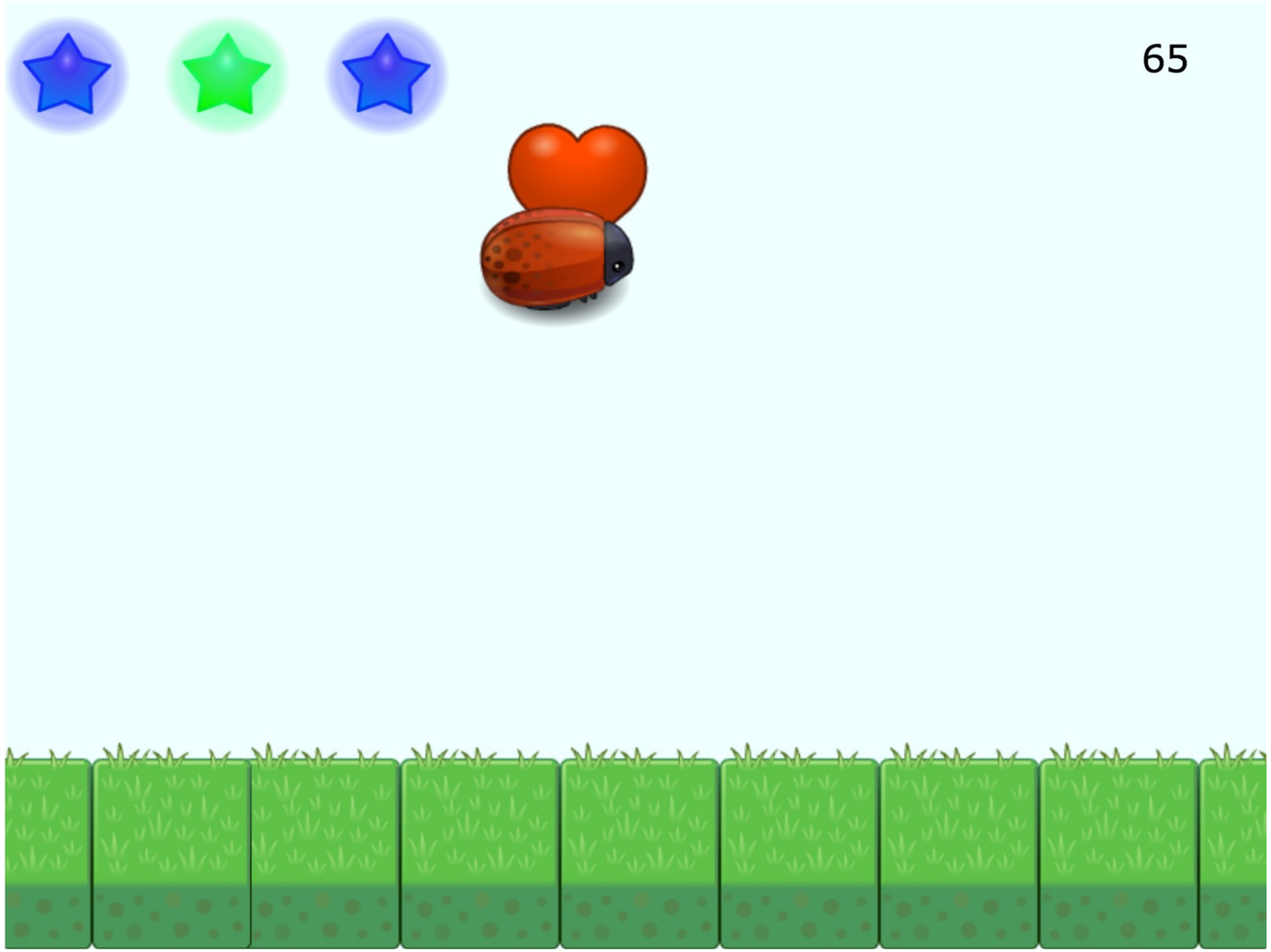
Parallel Stream Processing

```
Stream.iterate(1 , x -> x + 1)
     .parallel()
     .map(x -> x * 2)
     .filter(x -> x % 3 == 0)
     .limit(1000)
     .collect(Collectors.groupingByConcurrent(x -> x > 30))
     .forEach((x, y) -> System.out.println(x + ": " + y));
```

Parallel Streams

- Watch out!
  - Side effects
  - Even though lambdas operate only on final variables, when the variable is a reference to an object with mutable fields, you are out of luck if you are not careful.
  - We show this in the demo.
- Aim for reducing side effects and mutability.

Case Study, a Bug's Life



General Overview

```
public class Game extends javafx.application.Application {
public static void main(final String[] args) {
   javafx.application.Application.launch(args);
}
@Override
public void start(final Stage stage) {
   gameLoop(() -> {
      Platform.runLater(() -> {
         entities.forEach(e -> e.update());
            t.setText("" + points);
      });
      checkCollision(sun, bug, root);
   });
}
static interface Entity {
   public void update();
}
static class Grass implements Entity{
   final List<ImageView> tiles;
}
static class Collectible extends ImageView implements Entity{}
static class Bug extends ImageView implements Entity {}
}
```

gameLoop

```
void gameLoop(FuncVoidVoid gameLogic) {
new Thread( () -> {
     while(true) {
        try {
           TimeUnit.MILLISECONDS.sleep(refreshRate);
        } catch (Exception e) {e.printStackTrace();}
        gameLogic.run();
     }
   }).start();
}
```

```
@FunctionalInterface
static interface FuncVoidVoid {
public void run();
}
```

Streams Everywhere

```
final StackPane root = new StackPane();


final Canvas sky = new Canvas(screenWidth, screenHeight);
root.getChildren().add(sky);

final Collectible sun = new Collectible(root);
final Bug bug = new Bug(root);


List<Entity> entities =
   Stream.of(new Grass(root), sun, bug)
      .collect(Collectors.toList());
```

Streams Everywhere

```
public Grass(final StackPane root) {
   tiles = IntStream
         .range(0, nrTiles)
         .mapToObj( i -> new ImageView(){
             { setImage(tile);
               setTranslateX(i*getImage().getWidth());
              }
           })
         .collect(Collectors.toList());

    root.getChildren().addAll(tiles);
}

       @Override
       public void update() {
        tiles.forEach(tile -> {
           tile.setTranslateX(
              tile.getTranslateX() < -(tile.getImage().getWidth())?
                   screenWidth-grassSpeed:
                   tile.getTranslateX()-grassSpeed);
         } );
       }
```

Stream Operations

```
public int getBonusPoints() {
   return bonus.stream()
      .map(x -> x.getEffect() == null?5:10)
      .reduce(0, (x,y) -> x+y);
}
```

Stream Operations

```
public int getBonusPoints() {
   return bonus.stream()
      .map(x -> x.getEffect() == null?5:10)
      .reduce(0, (x,y) -> x+y);
}
```

```
List<Node> remove = root.getChildren().stream()
   .filter(x -> x instanceof Bonus.BImage)
   .collect(Collectors.toList());

root.getChildren().removeAll(remove);
```

Infinite Streams

```
static class Bug extends ImageView implements Entity {
     final double homeY = (-Grass.height/2)-5;
     final Iterator<Double> jumpPosition;
     boolean isJumping = false;
     boolean collided = false;
     final StackPane root;
     public Bug(final StackPane root) {
          this.root = root;
          jumpPosition = Stream
            .iterate(
            (double)jumpSpeed, x ->
               (getTranslateY()==homeY)?jumpSpeed:x - gravity)
            .map( speed -> {
               double dY  = ((1.0 * speed)) ;
               if (getTranslateY() < homeY + dY)
                  return getTranslateY() - dY;
               else return homeY;
            }).iterator();

          root.getChildren().add(this);
     }
```

Infinite Streams

```
public void jump() {
   isJumping = true;
}

@Override
public void update() {
   if(isJumping) {
      double position = jumpPosition.next();
      setTranslateY(position);
      if(position == homeY) {
         isJumping = false;
         collided = false;
         List<Node> remove = root.getChildren().stream()
            .filter(x -> x instanceof Bonus.BImage)
            .collect(Collectors.toList());
         root.getChildren().removeAll(remove);

      }
   }
}
```

Bonus: Optional

Goal: NullPointerException no more.

"***I call it my billion-dollar mistake.*** *It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that* ***all use of references should be absolutely safe, with checking performed automatically by the compiler.*** *But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes,* ***which have probably caused a billion dollars of pain and damage in the last forty years.***"

Sir Charles Antony Richard Hoare

Optional

- Absence of value should be checked statically.
- Interfaces should be honest about it.
- Optional provides a neat way to express and handle the absence of value.

Optional

```
public static Optional<Color> nextColor() {
  Color[] colors = new Color[]{Color.BLUE, Color.YELLOWGREEN};
  Random r = new Random();
  int index = r.nextInt(3);
  if(index == 2) return Optional.empty();
  else return Optional.of(colors[index]);
}
```

```
String colorName = nextColor()
  .map(x ->  x.toString())
  .orElse("No color");
```

Thank You!

Questions?