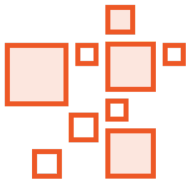
# Mastering Object-oriented Programming in Java

## TURNING EVERYTHING INTO OBJECTS

**Zoran Horvat**
CEO AT CODING HELMET

@zoranh http://codinghelmet.com

# Motivation to Develop Object-oriented Code



**In this module**

A few sketches of code with and without objects

**In the following modules**

Many techniques to incorporate object-oriented design

**Why objects and not procedural design?**

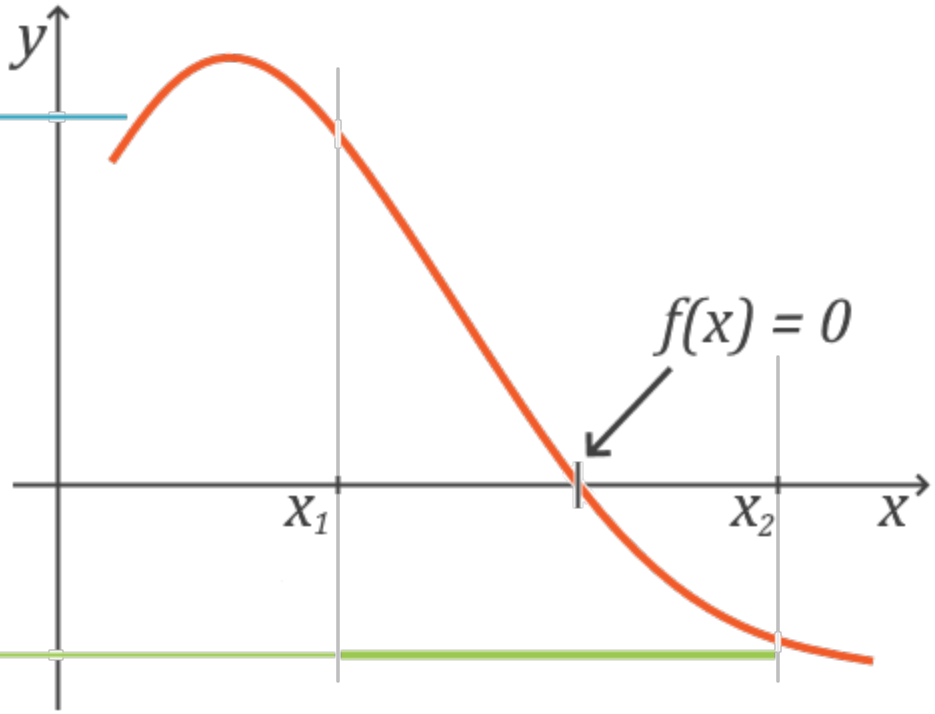
# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;

  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;

    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }

  return (lower + upper) / 2;
}
```



# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;
  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;
    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }
  return (lower + upper) / 2;
}
```

**Programmatic function**

Takes inputs and returns an output

Effectively computable

Completes in finitely many steps

# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;
  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;
    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }
  return (lower + upper) / 2;
}
```

Algorithm convergence parameter

`tolerance = 1e-10` fine

`tolerance = 1e-15` just fine

`tolerance = 1e-16` never terminates

# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;
  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;
    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }
  return (lower + upper) / 2;
}
```

```
double findZeroObjects(
    Function<Double, Double> f,
    double x1, double x2) {
  Segment range = new Segment(x1, x2);
  BisectionAlgorithm algorithm =
    range.bisect(f);
  double zero = algorithm.convergeTo(0);
  return zero;
}
```

# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;
  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;
    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }
  return (lower + upper) / 2;
}
```

```
double findZeroObjects(
    Function<Double, Double> f,
    double x1, double x2) {
  Segment range = new Segment(x1, x2);
  BisectionAlgorithm algorithm =
    range.bisect(f);
  double zero = algorithm.convergeTo(0);
  return zero;
}
```

Encapsulates tolerance (or any other mechanism)

# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;
  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;
    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }
  return (lower + upper) / 2;
}
```

```
double findZeroObjects(
    Function<Double, Double> f,
    double x1, double x2) {
  Segment range = new Segment(x1, x2);
  BisectionAlgorithm algorithm =
    range.bisect(f);
  double zero = algorithm.convergeTo(0);
  return zero;
}
```

# Procedural vs. Object-oriented

```
double findZeroProcedural(
    Function<Double, Double> f,
    double x1, double x2) {
  double lower = x1;
  double upper = x2;
  double tolerance = 1e-10;
  while (upper - lower > tolerance) {
    double middle = (lower + upper) / 2;
    if (Math.signum(f.apply(middle)) ==
        Math.signum(f.apply(lower))) {
      lower = middle;
    }
    else {
      upper = middle;
    }
  }
  return (lower + upper) / 2;
}
```

```
double findZeroObjects(
    Function<Double, Double> f,
    double x1, double x2) {
  return Segment.between(x1, x2)
      .bisect(f)
      .convergeTo(0);
}
```

# What Follows in This Course

## Making Your Java Code More Object-oriented

**Removing Booleans and branching instructions**

- Using proper polymorphic calls instead

**Using immutable objects and value objects**

**Removing nulls**

- Applying Null Object and Special Case patterns
- Using optional objects

# What Follows in This Course

Making Your Java Code More Object-oriented

Removing Booleans and branching instructions

Using immutable objects and value objects

Removing nulls

**Mastering Object-oriented Programming in Java**

**Turning loops implicit**

- Using streams to encapsulate loops
- Using domain-specific streams
- Applying composite objects

# What Follows in This Course

## Making Your Java Code More Object-oriented

Removing Booleans and branching instructions

Using immutable objects and value objects

Removing nulls

## Mastering Object-oriented Programming in Java

**Turning loops implicit**

**Inventing domain-specific languages**

- Making each object operate on one level
- Making object's interface readable
- Choosing names from the problem domain
- Chaining method calls to form "sentences"

# What Follows in This Course

## Making Your Java Code More Object-oriented

Removing Booleans and branching instructions

Using immutable objects and value objects

Removing nulls

## Mastering Object-oriented Programming in Java

**Turning loops implicit**

**Inventing domain-specific languages**

**Removing multiway branching**

- Using proper polymorphic calls instead
- Turning enums into classes

# What Follows in This Course

## Making Your Java Code More Object-oriented

Removing Booleans and branching instructions

Using immutable objects and value objects

Removing nulls

## Mastering Object-oriented Programming in Java

**Turning loops implicit**

**Inventing domain-specific languages**

**Removing multiway branching**

**Implementing the Rules pattern**

- Describing domain rules with Rule objects
- Implementing composite rules
- Using strategies for business logic

# What Follows in This Course

**Making Your Java Code More Object-oriented**

- Removing Booleans and branching instructions
- Using immutable objects and value objects
- Removing nulls

**Mastering Object-oriented Programming in Java**

- Turning loops implicit
- Inventing domain-specific languages
- Removing multiway branching
- Implementing the Rules pattern

# Dealing with Multitudes of Objects

```
Painter findCheapest(double sqMeters, List<Painter> painters) {
  Money lowestCost = Money.ZERO;
  Painter winner = null;
  for (Painter candidate: painters) {
    if (candidate.isAvailable()) {
      Money cost = candidate.estimateCompensation(sqMeters);
      if (winner == null ||
          cost.compareTo(lowestCost) <= 0) {
        winner = candidate;
        lowestCost = cost;
      }
    }
  }
  return winner;
}
```

Selects a painter

who is available

and asks for least money

# Dealing with Multitudes of Objects

```
Painter findCheapest(double sqMeters, List<Painter> painters) {
  Money lowestCost = Money.ZERO;
  Painter winner = null;
  for (Painter candidate: painters) {
    if (candidate.isAvailable()) {
      Money cost = candidate.estimateCompensation(sqMeters);
      if (winner == null ||
          cost.compareTo(lowestCost) <= 0) {
        winner = candidate;
        lowestCost = cost;
      }
    }
  }
  return winner;
}
```

Special values and null

used to drive edge cases

making the result hard to predict

# Dealing with Multitudes of Objects

```
Painter findCheapest(double sqMeters, List<Painter> painters) {
  return painters.stream()
      .filter(Painter::isAvailable)
      .min(Comparator.comparing(painter -> painter.estimateCompensation(sqMeters)));
}
```

# Dealing with Multitudes of Objects

```
Painter findCheapest(double sqMeters, List<Painter> painters) {
    return painters.stream()
            .filter(Painter::isAvailable)
            .min(Comparator.comparing(painter -> painter.estimateCompensation(sqMeters)));
}
```

Returns Optional<Painter>

Cannot return Optional<Painter> as Painter

# Dealing with Multitudes of Objects

```
Optional<Painter> findCheapest(double sqMeters, List<Painter> painters) {
    return painters.stream()
        .filter(Painter::isAvailable)
        .min(Comparator.comparing(painter -> painter.estimateCompensation(sqMeters)));
}
```

Returns Optional<Painter>

Syntactically correct

Makes assumptions explicit

# Addressing Code Readability Issues

```
Money getTotalCost(double sqMeters, List<Painter> painters) {
  return painters.stream()
      .filter(Painter::isAvailable)
      .map(painter -> painter.estimateCompensation(sqMeters))
      .reduce(Money::add)
      .orElse(Money.ZERO);
}
```

# Addressing Code Readability Issues

```
Money getTotalCost(double sqMeters, List<Painter> painters) {
  return painters.stream()
      .filter(Painter::isAvailable)
      .map(painter -> painter.estimateCompensation(sqMeters))
      .reduce(Money::add)
      .orElse(Money.ZERO);
}
```

Idioms of the business domain

Idioms of the programing language

# Addressing Code Readability Issues

```
Money getTotalCost(double sqMeters, PaintersStream painters) {
  return painters.costs(sqMeters).sum();
}
```

# Addressing Code Readability Issues

```
Money getTotalCost(double sqMeters, PaintersStream painters) {
  return painters.costs(sqMeters).sum();
}
```

Choose names wisely

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

Encapsulated bisection algorithm

Hides a very complicated algorithm

Solving the work scheduling problem

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

*bisection* *convergence* *intervals*

**Alternate definition of design**

Makes code for a complex
business domain look simple
(e.g. by designing a Domain-specific Language)

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it by calculating the total area they paint

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it by calculating the total area they paint, let the algorithm converge

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it by calculating the total area they paint, let the algorithm converge to the total area from the input

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it by calculating the total area they paint, let the algorithm converge to the total area from the input, with scheduling precision of one millisecond

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it by calculating the total area they paint, let the algorithm converge to the total area from the input, with scheduling precision of one millisecond, and then take the middle of the interval as the overall result

# Inventing a Domain-specific Language

```
private Duration totalTime(List<Painter> painters, double sqMeters) {
  return DurationRange.zeroTo(this.minimumIndividualTime(painters, sqMeters))
      .bisect(time -> this.totalSqMeters(painters, time))
      .convergeTo(sqMeters, Duration.ofMillis(1))
      .middle();
}
```

To find the total time in which many painters will finish painting the area together, is to take a duration from zero to minimum time spent if a single painter did all the work, then bisect it by calculating the total area they paint, let the algorithm converge to the total area from the input, with scheduling precision of one millisecond, and then take the middle of the interval as the overall result.

# Removing Multiway Branching

```
private void claimWarranty(
    Article article, DeviceStatus status, Optional<LocalDate> sensorFailureDate) {

  LocalDate today = LocalDate.now();

  if (status.equals(DeviceStatus.allFine())) {
    this.claimMoneyBack(article, today);

  } else if (status.equals(DeviceStatus.notOperational())) {
    this.claimMoneyBack(article, today);
    this.claimExpress(article, today);

  } else if (status.equals(DeviceStatus.visiblyDamaged())) {

  } else if (status.equals(DeviceStatus.sensorFailed())) {
    this.claimMoneyBack(article, today);
    this.claimExtended(article, today, sensorFailureDate);

  } else if (status.equals(DeviceStatus.notOperational().andVisiblyDamaged()))) {
    this.claimExpress(article, today);
  }
  ...
}
```

# Removing Multiway Branching

```
private void claimWarranty(
    Article article, DeviceStatus status) {

  ClaimingRulesBuilder.handle(article, LocalDate.now())
    .withMoneyBackAction(this::offerMoneyBack)
    .withRepairAction(this::offerRepair)
    .withSensorRepairAction(this::offerSensorRepair)
    .buildAll()
    .applicableTo(status)
    .apply();
}
```

# Summary

**Samples of object-oriented code**

- Better than procedural code
- Enables state encapsulation
- Enables behavior encapsulation

# Summary

**Other benefits in object-oriented code**

- Objects are substitutable
- Concrete calls are resolved at run time

# Summary

**Advanced issues**

- Importance of naming objects and methods
- Makes code easier to read
- Closer to language of the business

# Summary

**Next module:**
Staying Focused on
Domain Logic with Streams