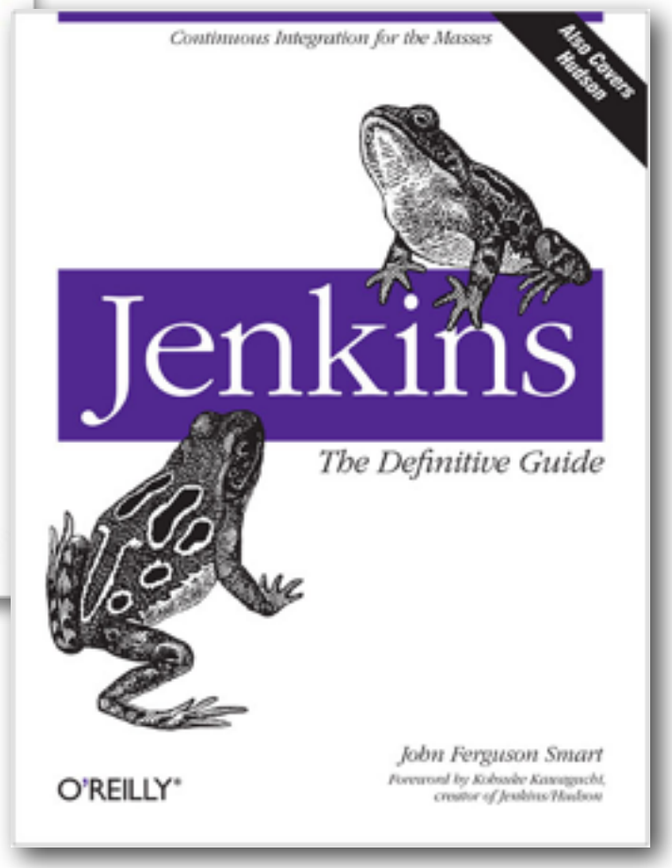
Clean Coding Practices for Java Developers

John Ferguson Smart
john.smart@wakaleo.com
http://www.wakaleo.com
Twitter: wakaleo

So who is this guy, anyway?



Who are you writing code for, anyway?

Who are you writing code for, anyway?

The computer?

Who are you writing code for, anyway?

Other developers?

Who are you writing code for, anyway?

Your next boss?

Why is clean code so important?

Easier to Understand

Easier to Change

Cheaper to Maintain

Choose your names well

"What's in a name? That which we call a rose
By any other name would smell as sweet."
Romeo and Juliet (II, ii, 1-2)

Really?

Use a common vocabulary

```
getCustomerInfo()
getClientDetails()
getCustomerRecord()
...
```

Which one do I use? Are they the same?

```
getCustomer()
```

Choose one and stick to it

Use meaningful names

```
int days;
```

What does this represent?

```
int daysSinceCreation;
```

```
int daysSinceLastModification;
```

```
int durationInDays;
```

These are all more meaningful choices

Don’t talk in code

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = “102”;
    ...
}
```


```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private final String recordId = “102”
  ...
}
```

But don’t state the obvious

```
List<Client> clientList;
```

Is ‘List’ significant or just noise?

```
List<Client> clients;
```

```
List<Client> regularClients;
```

```
List<Client> newClients;
```

In short: Don’t make me think!

What does this do?

```
private int calculate(List<TestStep> t) {
    int res = 0;
    for(TestStep s : t) {
        if (!s.getChildren().isEmpty()) {
            res += calculate(s.getChildren());
        } else {
            res++;
        }
    }
    return res;
}
```

Clearer parameter name

More explicit method name

```
private int countLeafStepsIn(List<TestStep> testSteps) {
    int leafCount = 0;
    for (TestStep step : testSteps) {
        if (step.isAGroup()) {
            leafCount += countLeafStepsIn(step.getChildren());
        } else {
            leafCount++;
        }
    }
    return leafCount;
}
```

What are we counting?

Method call rather than boolean expression

Make your code tell a story

Methods should be small

```
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();

    Map<String, Object> storyContext = new HashMap<String, Object>();
    storyContext.put("stories", storyResults);
    storyContext.put("storyContext", "All stories");
    addFormattersToContext(storyContext);
    writeReportToOutputDirectory("stories.html",
                                 mergeTemplate(STORIES_TEMPLATE_PATH).usingContext(storyContext));

    Map<String, Object> featureContext = new HashMap<String, Object>();
    addFormattersToContext(featureContext);
    featureContext.put("features", featureResults);
    writeReportToOutputDirectory("features.html",
                                 mergeTemplate(FEATURES_TEMPLATE_PATH).usingContext(featureContext));

    for(FeatureResults feature : featureResults) {
        generateStoryReportForFeature(feature);
    }

    generateReportHomePage(storyResults, featureResults);

    getTestHistory().updateData(featur

    generateHistoryReport();
}
```

Code hard to understand

Methods should be small

```
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();

    private void generateStoriesReport(final List<StoryTestResults> storyResults) throws IOException {
        Map<String, Object> context = new HashMap<String, Object>();
        context.put("stories", storyResults);
        context.put("storyContext", "All stories");
        addFormattersToContext(context);
        String htmlContents = mergeTemplate(STORIES_TEMPLATE_PATH).usingContext(context);
        writeReportToOutputDirectory("stories.html", htmlContents);
    }
    featureContext.put("features", featureResults);
    writeReportToOutputDirectory("features.html",
                                 mergeTemplate(FEATURES_TEMPLATE_PATH).usingContext(featureContext));

    for(FeatureResults feature : featureResults) {
        generateStoryReportForFeature(feature);
    }

    generateReportHomePage(storyResults, featureResults);

    getTestHistory().updateData(featureResults);

    generateHistoryReport();
}
```

Refactor into clear steps

Methods should be small

```
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();

    generateStoriesReportFor(storyResults);

    Map<String, Object> featureContext = new HashMap<String, Object>();
    addFormattersToContext(featureContext);
    featureContext.put("features", featureResults);
    writeReportToOutputDirectory("features.html",
                                 mergeTemplate(FEATURES_TEMPLATE_PATH).usingContext(featureContext));

    for(FeatureResults feature : featureResults) {
        generateStoryReportForFeature(feature);
    }

    generateReportHomePage(storyResults, featureResults);

    getTestHistory().updateData(featureResults);

    generate
}
```

```
private void updateHistoryFor(final List<FeatureResults> featureResults) {
    getTestHistory().updateData(featureResults);
}
```

Methods should be small

```
private void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                        final List<FeatureResults> featureResults)
    throws IOException {

        copyResourcesToOutputDirectory();

        generateStoriesReportFor(storyResults);
        generateFeatureReportFor(featureResults);
        generateReportHomePage(storyResults, featureResults);

        updateHistoryFor(featureResults);
        generateHistoryReport();
}
```

Methods should only do one thing

Too much going on here...

```
public String getReportName(String reportType, final String qualifier) {
    if (qualifier == null) {
        String testName = "";
        if (getUserStory() != null) {
            testName = NameConverter.underscore(getUserStory().getName());
        }
        String scenarioName = NameConverter.underscore(getMethodName());
        testName = withNoIssueNumbers(withNoArguments(appendToIfNotNull(testName, scenarioName)));
        return testName + "." + reportType;
    } else {
        String userStory = "";
        if (getUserStory() != null) {
            userStory = NameConverter.underscore(getUserStory().getName()) + "_";
        }
        String normalizedQualifier = qualifier.replaceAll(" ", "_");
        return userStory + withNoArguments(getMethodName()) + "_" + normalizedQualifier + "." + reportType;
    }
}
```

Mixing *what* and *how*

Methods should only do one thing

Chose *what* to do here

```
public String getReportName(final ReportType type, final String qualifier) {
    ReportNamer reportNamer = ReportNamer.forReportType(type);
    if (shouldAddQualifier(qualifier)) {
        return reportNamer.getQualifiedTestNameFor(this, qualifier);
    } else {
        return reportNamer.getNormalizedTestNameFor(this);
    }
}
```

The *how* is the responsibility of another class

Encapsulate boolean expressions

```
for (TestStep currentStep : testSteps) {
    if (!currentStep.isAGroup() && currentStep.getScreenshots() != null) {
        for (RecordedScreenshot screenshot : currentStep.getScreenshots()) {
            screenshots.add(new Screenshot(screenshot.getScreenshot().getName(),
                    currentStep.getDescription(),
                    widthOf(screenshot.getScreenshot()),
                    currentStep.getException()));
        }
    }
}
```

What does this boolean mean?

```
for (TestStep currentStep : testSteps) {
    if (currentStep.needsScreenshots()) {
        ...
```

```
public boolean needsScreenshots() {
    return (!isAGroup() && getScreenshotsAndHtmlSources() != null);
}
```

Expresses the intent better

Avoid unclear/ambiguous class name

```
        for (TestStep currentStep : testSteps) {
            if (currentStep.needsScreenshots()) {
                for (RecordedScreenshot screenshot : currentStep.getScreenshots()) {
                    screenshots.add(new Screenshot(screenshot.getScreenshot().getName(),
                            currentStep.getDescription(),
                            widthOf(screenshot.getScreenshot()),
                            currentStep.getException()));
                }
            }
        }
```

Is this class name really accurate?

Too many screenshots!

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (weNeedAScreenshotFor(currentStep)) {
            for (ScreenshotAndHtmlSource screenshotAndHtml : currentStep.getScreenshotsAndHtmlSources()) {
                screenshots.add(new Screenshot(screenshotAndHtml.getScreenshotFile().getName(),
                        currentStep.getDescription(),
                        widthOf(screenshotAndHtml.getScreenshot()),
                        currentStep.getException()));
            }
        }
    }
    return ImmutableList.copyOf(screenshots);
```

Using a more revealing class name

And a clearer method name

Encapsulate overly-complex code

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (currentStep.needsScreenshots()) {
            for (ScreenshotAndHtmlSource screenshotAndHtml : currentStep.getScreenshotsAndHtmlSources()) {
                screenshots.add(new Screenshot(screenshotAndHtml.getScreenshot().getName(),
                        currentStep.getDescription(),
                        widthOf(screenshotAndHtml.getScreenshot()),
                        currentStep.getException()));
            }
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

What does all this do?

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (weNeedAScreenshotFor(currentStep)) {
            screenshots.addAll(
                convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep)));
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

Clearer intention

Encapsulate overly-complex code

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (currentStep.needsScreenshots()) {
            screenshots.addAll(
                convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep)));
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

What we are doing

```
private Converter<ScreenshotAndHtmlSource, Screenshot> toScreenshotsFor(final TestStep currentStep) {
    return new Converter<ScreenshotAndHtmlSource, Screenshot>() {
        @Override
        public Screenshot convert(ScreenshotAndHtmlSource from) {
            return new Screenshot(from.getScreenshotFile().getName(),
                                  currentStep.getDescription(),
                                  widthOf(from.getScreenshotFile()),
                                  currentStep.getException());
        }
    };
}
```

How we do it

Avoid deep nesting

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (currentStep.needsScreenshots()) {
            screenshots.addAll(
                convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep)));
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

Code doing too many things

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();

    List<TestStep> testStepsWithScreenshots = select(getFlattenedTestSteps(),
                                                     having(on(TestStep.class).needsScreenshots()));

    for (TestStep currentStep : testStepsWithScreenshots) {
        screenshots.addAll(convert(currentStep.getScreenshotsAndHtmlSources(),
                                   toScreenshotsFor(currentStep)));
    }

    return ImmutableList.copyOf(screenshots);
}
```

Break the code down into logical steps

Remove the nested condition

Keep each step simple!

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();

    List<TestStep> testStepsWithScreenshots = select(getFlattenedTestSteps(),
                                                     having(on(TestStep.class).needsScreenshots()));

    for (TestStep currentStep : testStepsWithScreenshots) {
        screenshots.addAll(convert(currentStep.getScreenshotsAndHtmlSources(),
                                   toScreenshotsFor(currentStep)));
    }

    return ImmutableList.copyOf(screenshots);
}
```

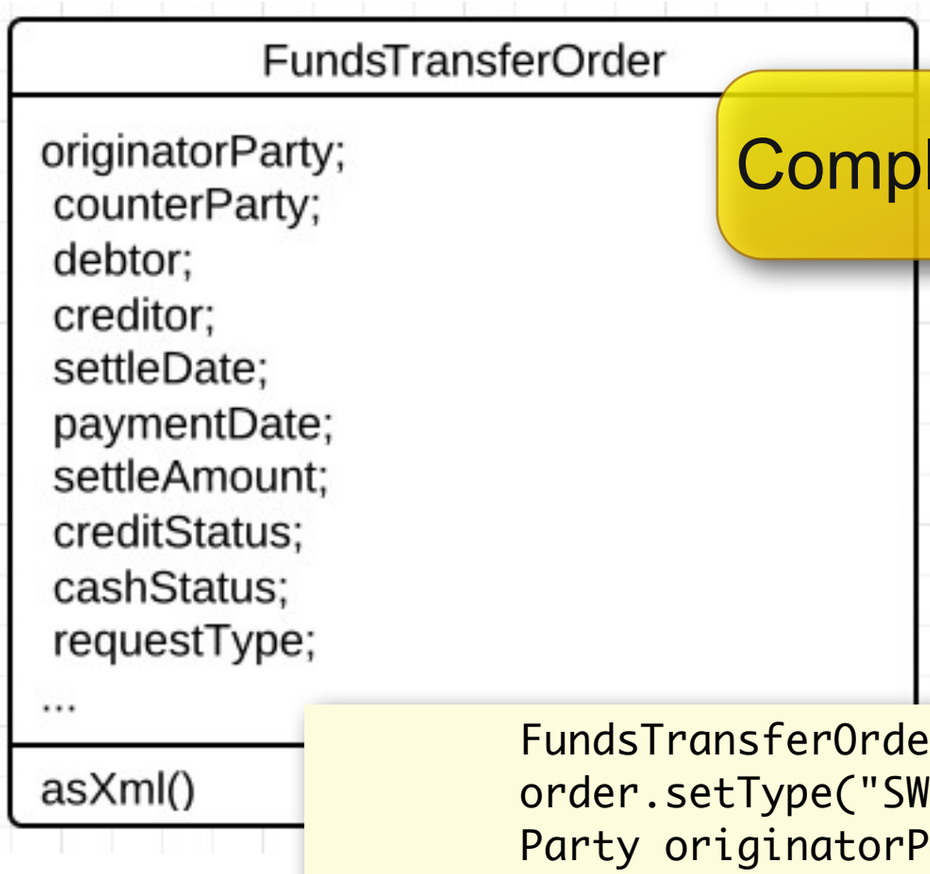
Too much happening here?

```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();

    List<TestStep> testStepsWithScreenshots = select(getFlattenedTestSteps(),
                                                     having(on(TestStep.class).needsScreenshots()));

    for (TestStep currentStep : testStepsWithScreenshots) {
        screenshots.addAll(screenshotsIn(currentStep));
    }

    return ImmutableList.copyOf(screenshots);
}

private List<Screenshot> screenshotsIn(TestStep currentStep) {
    return convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep));
}
```

This reads more smoothly

Code should communicate fluently

Use Fluent APIs



Complex domain object

Lots of variants

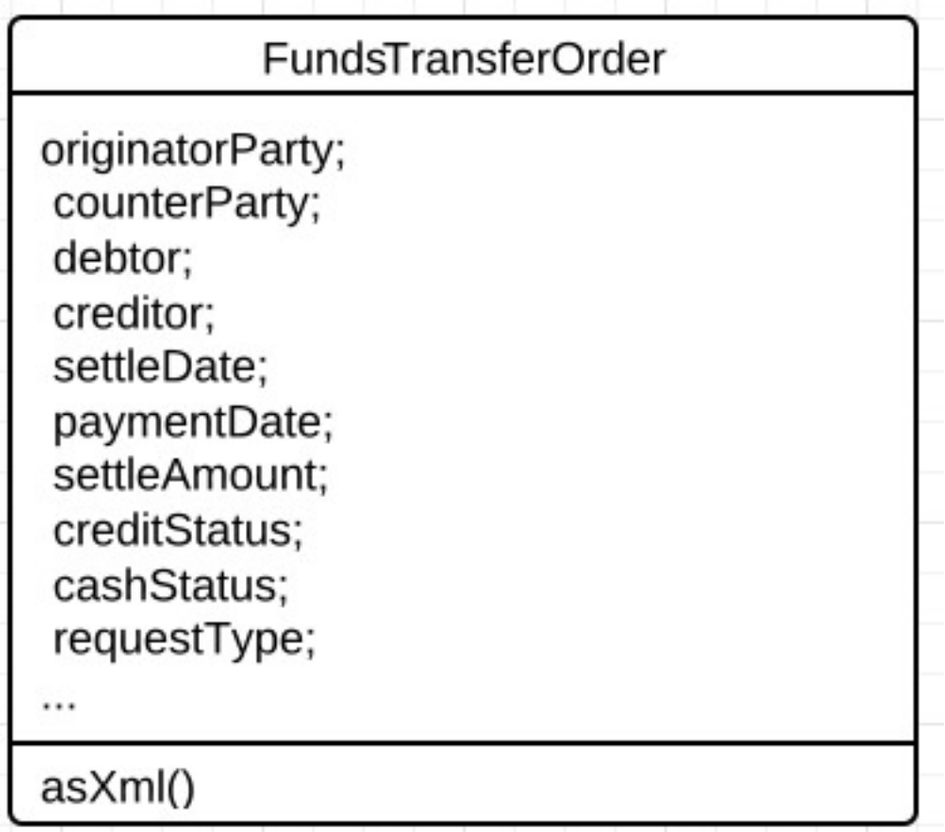
Object tree

```
FundsTransferOrder order = new FundsTransferOrder();
order.setType("SWIFT");
Party originatorParty = organizationServer.findPartyByCode("WPAC");
order.setOriginatorParty(originatorParty);
Party counterParty = organizationServer.findPartyByCode("CBAA");
order.setCounterParty(counterParty);
order.setDate(DateTime.parse("22/11/2011"));
Currency currency = currencyTable.findCurrency("USD");
Amount amount = new Amount(500, currency);
order.setAmount(amount);
```

Complex code

Need to know how to create the child objects

Use Fluent APIs



More readable

No object creation

Easier to maintain

```
FundsTransferOrder.createNewSWIFTOrder()
                  .fromOriginatorParty("WPAC")
                  .toCounterParty("CBAA")
                  .forDebitor("WPAC")
                  .and().forCreditor("CBAA")
                  .settledOnThe("22/11/2011")
                  .forAnAmountOf(500, US_DOLLARS)
                  .asXML();
```

Use Fluent APIs

Readable parameter style

```
TestStatistics testStatistics = testStatisticsProvider.statisticsForTests(With.tag("Boat sales"));

double recentBoatSalePassRate = testStatistics.getPassRate().overTheLast(5).testRuns();
```

Fluent method call

Use Fluent APIs

```java
public Integer getSuccessCount() {
    return count(successfulSteps()).in(getLeafTestSteps());
}

public Integer getFailureCount() {
    return count(failingSteps()).in(getLeafTestSteps());
}

public Integer getIgnoredCount() {
    return count(ignoredSteps()).in(getLeafTestSteps());
}
```

Fluent style...

```java
StepCountBuilder count(StepFilter filter) {
    return new StepCountBuilder(filter);
}
```

A builder does the dirty work

```java
abstract class StepFilter {
    abstract boolean apply(TestStep step);

}
```

Represents how to select steps

```java
StepFilter successfulSteps() {
    return new StepFilter() {
        @Override
        boolean apply(TestStep step) {
            return step.isSuccessful();
        }
    };
}

StepFilter failingSteps() {
    return new StepFilter() {
        @Override
        boolean apply(TestStep step) {
            return step.isFailure();
        }
    };
}
```

Override to select different step types

Your code is organic

Help it grow

Replace Constructors with Creation Methods

TrainReservation

- TrainReservation(Station, Station, Date, Date, double)
- TrainReservation(Station, Station, Date, Date, double, Concession)
- TrainReservation(Station, Station, Date, Date, Discount, double)
- TrainReservation(Station, Station, Date, Date, Discount, Concession, double)

Too many constructors

Business knowledge hidden in the constructors

Which one should I use?

Replace Constructors with Creation Methods

TrainReservation

- TrainReservation(Station, Station, Date, Date, double)
- TrainReservation(Station, Station, Date, Date, double, Concession)
- TrainReservation(Station, Station, Date, Date, Discount, double)
- TrainReservation(Station, Station, Date, Date, Discount, Concession, double)

Private constructor

TrainReservation

- TrainReservation(Station, Station, Date, Date, Discount, Concession, double)

createStandardReservation(Station, Station, Date, Date, double)	TrainReservation
createReservationWithConcession(Station, Station, Date, Date, double, Concession)	TrainReservation
createDiscountReservation(Station, Station, Date, Date, double, Discount)	ation
createDiscountReservationWithConcession(Station, Station, Date, Date, double, Discount, Concession)	ation

Static creator methods

One implementing class

Replace Constructors with Creation Methods

TrainReservation

- TrainReservation(Station, Station, Date, Date, double)
- TrainReservation(Station, Station, Date, Date, double, Concession)
- TrainReservation(Station, Station, Date, Date, Discount, double)
- TrainReservation(Station, Station, Date, Date, Discount, Concession, double)

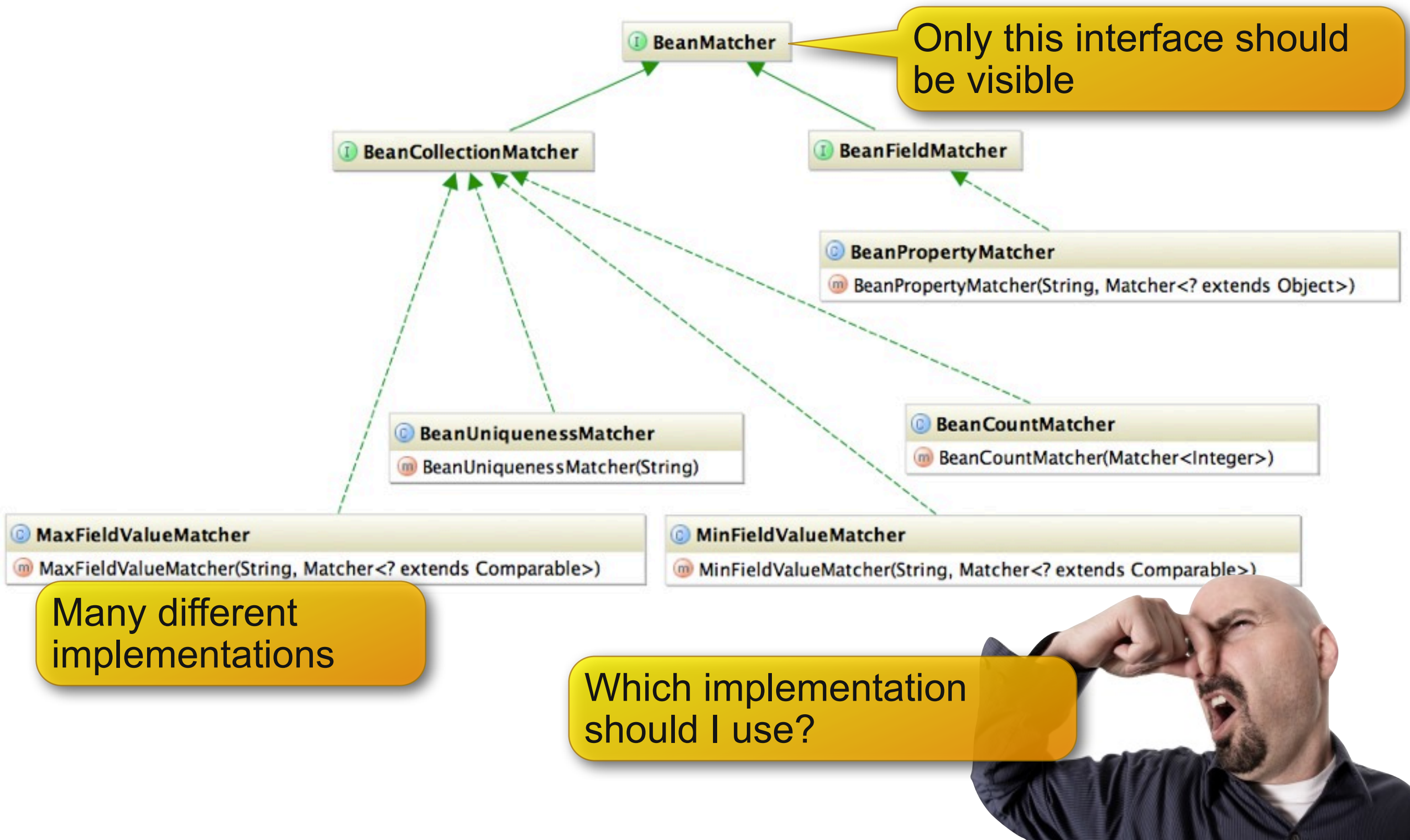
TrainReservation

- TrainReservation(Station, Station, Date, Date, Discount, Concession, double)

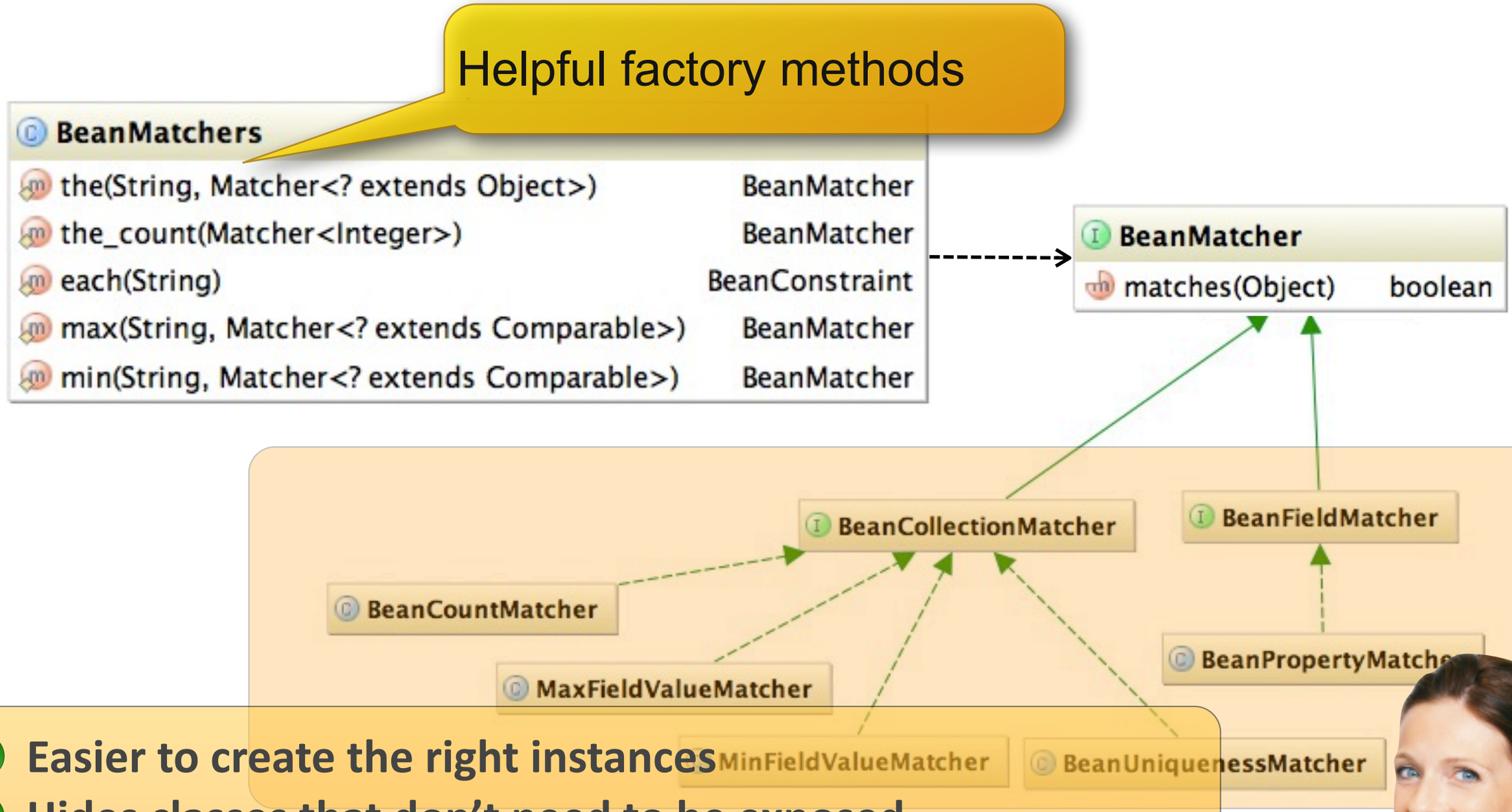
createStandardReservation(Station, Station, Date, Date, double)	TrainReservation
createReservationWithConcession(Station, Station, Date, Date, double, Concession)	TrainReservation
createDiscountReservation(Station, Station, Date, Date, double, Discount)	ation
createDiscountReservationWithConcession(Station, Station, Date, Date, double, Discount, Concession)	ation

- ✅ **Communicates the intended use better**
- ✅ **Overcomes technical limits with constructors**
- ❌ **Inconsistent object creation patterns**

Encapsulate Classes with a Factory

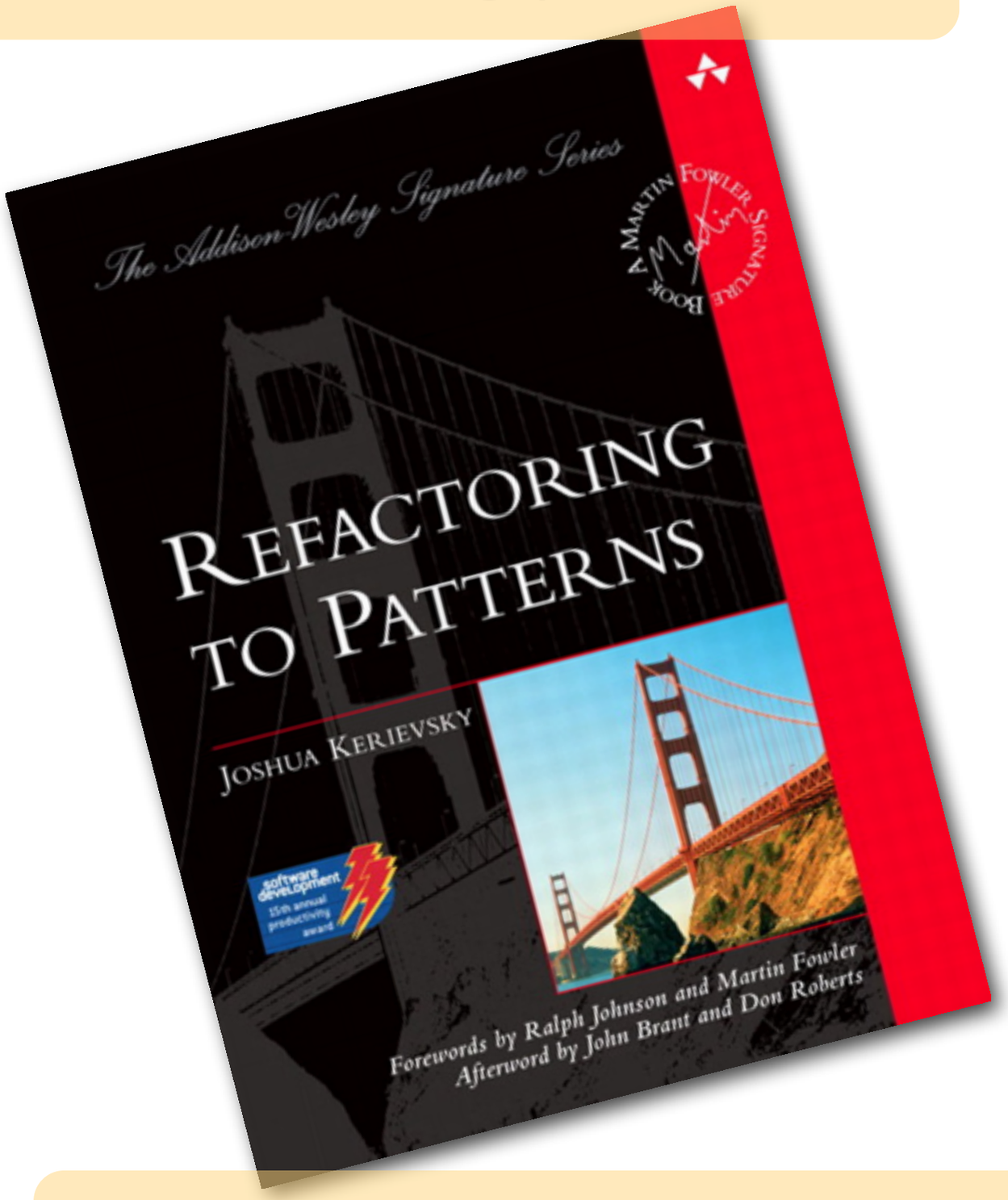


Encapsulate Classes with a Factory



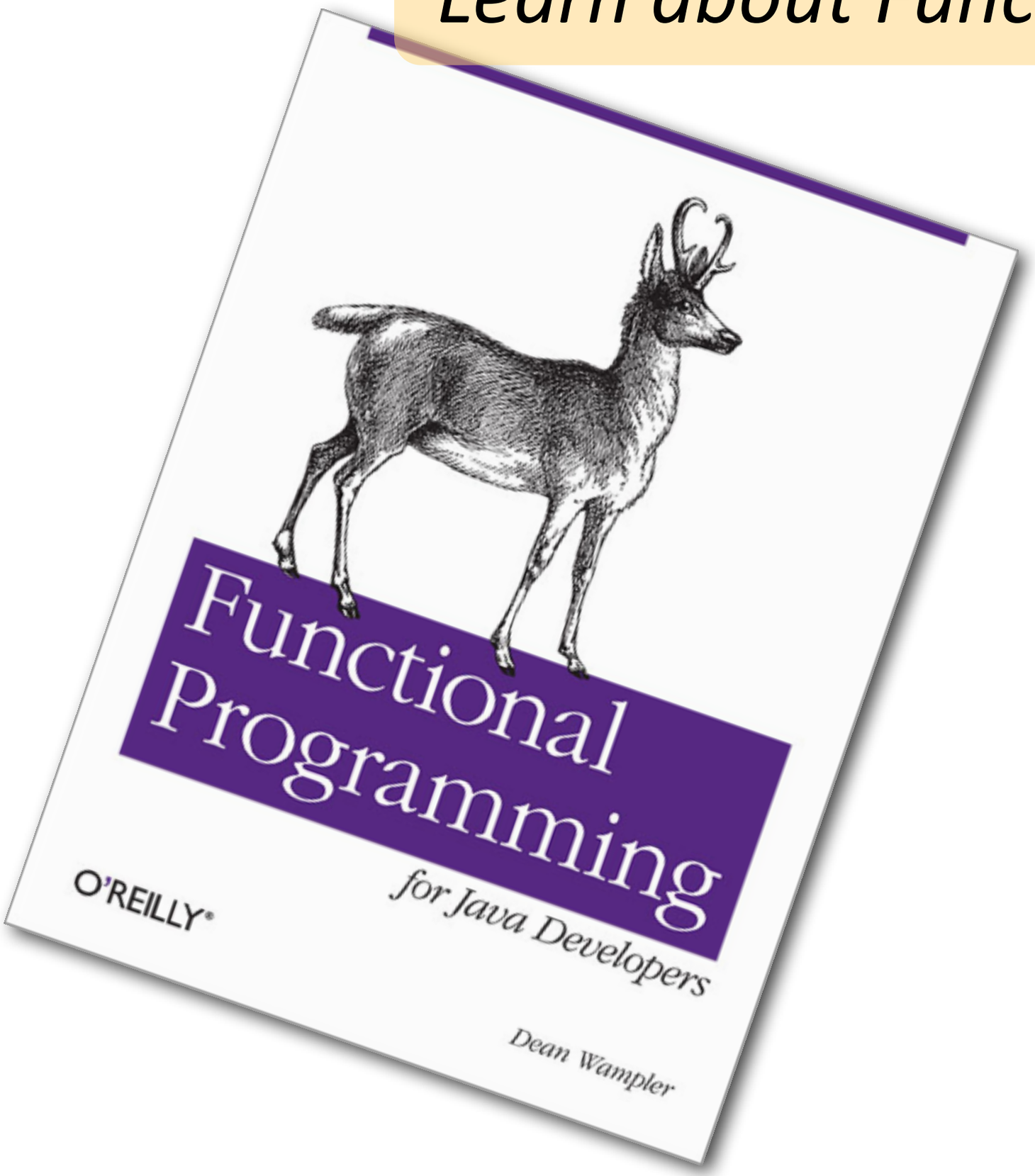
- Easier to create the right instances
- Hides classes that don't need to be exposed
- Encourages "programming to an interface"
- Need a new method when new classes are added
- Need access to factory class to customize/extend

Plenty of other refactoring patterns...

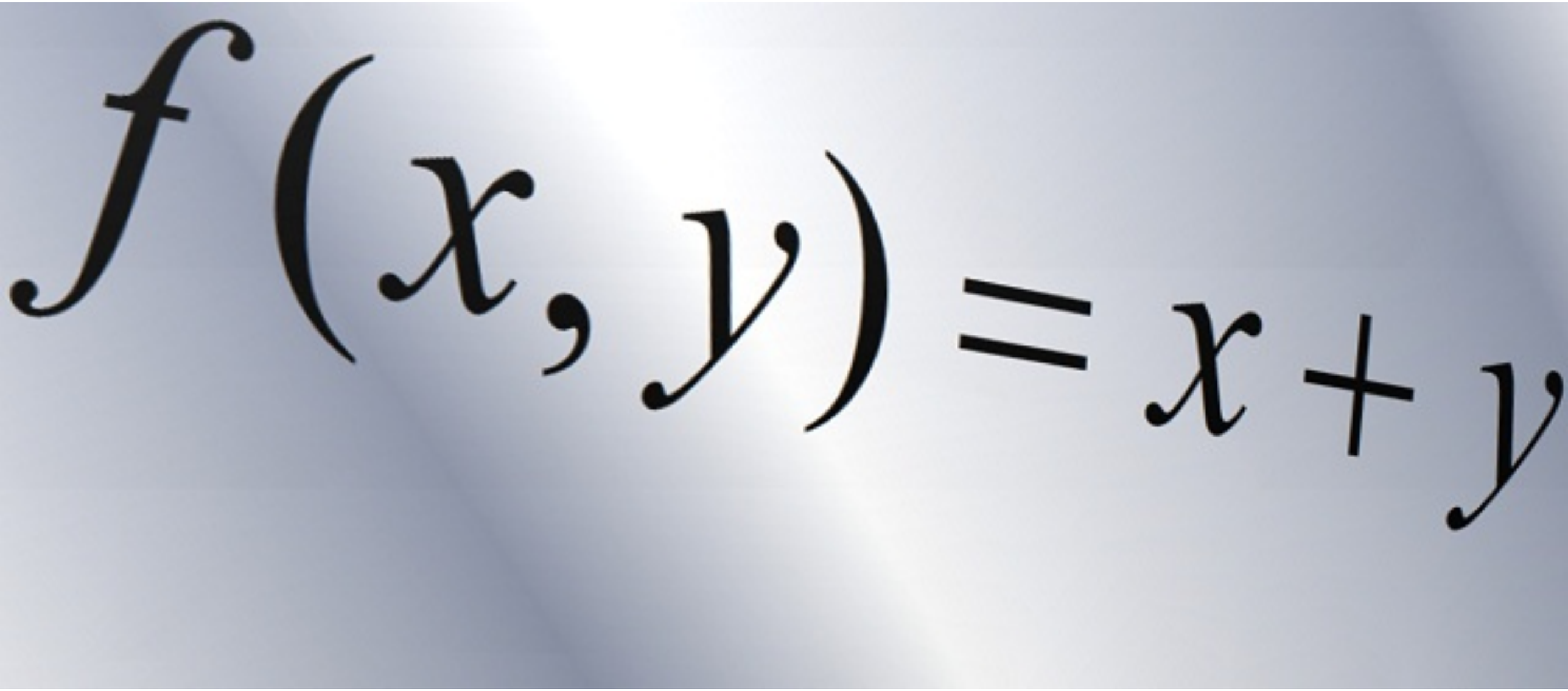


But know *why* you are applying them

Learn about Functional Programming!



Functions as a 1st class concept



Immutable value objects

No side effects

Benefit from concurrency

No loops

Scala

```scala
(1 to 100).sum
```

5050

```scala
val s = List( "a", "b", "f", "d", "c")
s.sorted.map {_.toUpperCase }
```

A, B, C, D, F

```scala
val result = dataList.par.map(line => processItem(line))
```

Parallel processing

```scala
def divisors(n: Int): List[Int] =
    for (i <- List.range(1, n+1) if n % i == 0) yield i

def isPrime(n: Int) = divisors(n).length == 2

isPrime(7)
```

TRUE

Functional programming in Java?

Surely this is madness!

lambdaj

– DSL for manipulating collections in a functional style

– Replace loops with more concise and readable code

Functional constructs in Java

LambdaJ support many high level collection-related functions

filter

aggregate

sort

convert

extract

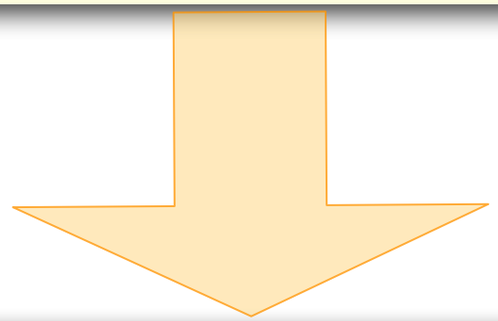
index

group

Find all adult ages

filter

```
List<Integer> adultAges = new ArrayList<Integer>();
for(int age : ages) {
    if (age >= 18) {
        adults.add(age);
    }
}
```

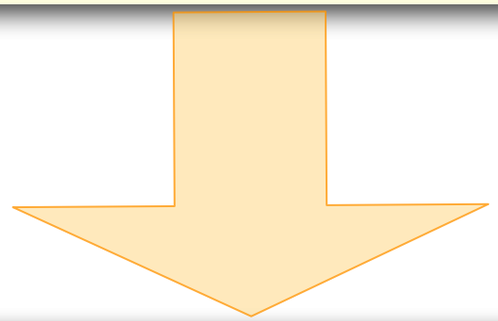


```
List<Integer> adultAges = filter(greaterThanOrEqualTo(18), ages);
```

Find all adults

filter

```
List<Person> adults = new ArrayList<Person>();
for(int person : people) {
    if (person.getAge() >= 18) {
        adults.add(person);
    }
}
```



```
List<Person> adults = filter(having(on(Person.class).getAge(), greaterThanOrEqualTo(18)),
                             people);
```

Find all the sales of Ferraris

filter

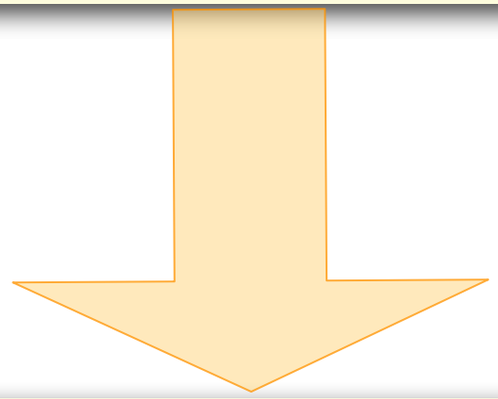
```
List<Sale> salesOfAFerrari = new ArrayList<Sale>();
for (Sale sale : sales) {
    if (sale.getCar().getBrand().equals("Ferrari"))
        salesOfAFerrari.add(sale);
}
```

```
List<Sale> salesOfAFerrari = select(sales,
    having(on(Sale.class).getCar().getBrand(),equalTo("Ferrari")));
```

Sort sales by cost

sort

```
List<Sale> sortedSales = new ArrayList<Sale>(sales);
Collections.sort(sortedSales, new Comparator<Sale>() {
    public int compare(Sale s1, Sale s2) {
        return Double.valueOf(s1.getCost()).compareTo(s2.getCost());
    }
});
```



```
List<Sale> sortedSales = sort(sales, on(Sale.class).getCost());
```

Index cars by brand

index

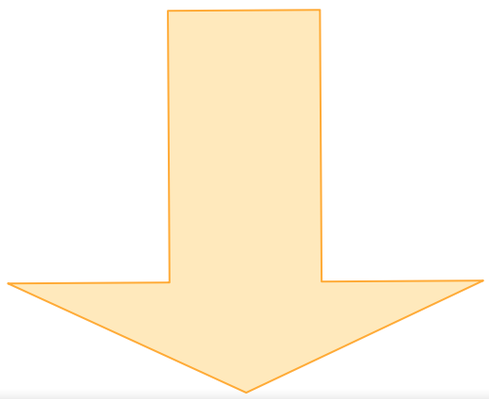
```
Map<String, Car> carsByBrand = new HashMap<String, Car>();
for (Car car : db.getCars()) {
    carsByBrand.put(car.getBrand(), car);
}
```

```
Map<String, Car> carsByBrand = index(cars, on(Car.class).getBrand());
```

Find the total sales

aggregate

```
double totalSales = 0.0;
for (Sale sale : sales) {
    totalSales = totalSales + sale.getCost();
}
```



```
double totalSales = sumFrom(sales).getCost();
```

Find most costly sale

aggregate

```
double maxCost = 0.0;
for (Sale sale : sales) {
    double cost = sale.getCost();
    if (cost > maxCost) maxCost = cost;
}
```

```
double maxCost = maxFrom(sales).getCost();
```

Extract the costs of the cars as a list

extract

```
List<Double> costs = new ArrayList<Double>();
for (Car car : cars) costs.add(car.getCost());
```

```
List<Double> costs = extract(cars, on(Car.class).getCost());
```

guava immutable collections

guava immutable collections

- Defensive
- Thread-safe
- Efficient

Creating an immutable list

```
List<String> colors = ImmutableList.of("red", "green", "blue");
```

```
ImmutableSet<Field> fields =
                    ImmutableSet.copyOf(stepLibraryClass.getDeclaredFields());
```

Creating an immutable copy

The Boy Scout Rule

"Leave the camp ground cleaner than you found it"

Thank You

John Ferguson Smart
john.smart@wakaleo.com
http://www.wakaleo.com
Twitter: wakaleo