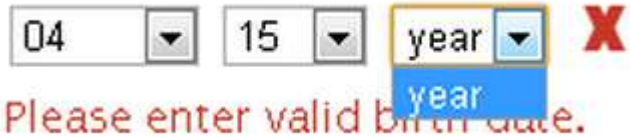
Java Best Practices

@rtoal
2013-06-12

Why?

We professionals need to write code that is correct, reliable, maintainable, efficient, robust, resilient, readable, reusable, scalable, etc.




```
Public Function EmailList() As Boolean
    Try
        Return True
    Catch ex As Exception
        Return False
    End Try
End Function
```

How do we learn best practices?

By *understanding* bad code

Okay, maybe not as bad as CodingHorror, SerialDate, and the thedailywtf (e.g.,this and this and this and this and this and this and this and this) ...

We mean innocent-looking code arising from misconceptions and inexperience

What will we look at?

Immutability
Collections
Guava
Exceptions
Polymorphism
Null
Exceptions
Concurrency

Formatting
Serialization
I/O
Comments
Validation
Logging
Generics
Security

What's wrong here?

```
public final class Task {
    private final String name;
    private final Date start;
    public Task(final String name, final Date start) {
        this.name = name;
        this.start = start;
    }
    public String getName() {return name;}
    public Date getStart() {return start;}
}
```

java.util.
Date is
mutable

Immutability

Immutable objects

- are thread-safe
- can be shared (cached)
- can't be trashed by someone else's code
- make great hashtable keys!
- lead to simpler code (e.g. no need to "undo" anything when backtracking)

BEST PRACTICE

Favor immutable objects, using mutable objects only when absolutely necessary

Defensive Copying

```
public final class Task {
    private final String name;
    private final Date start;
    public Task(final String name, final Date start) {
        this.name = name;
        this.start = new Date(start.getTime());
    }
    public String getName() {return name;}
    public Date getStart() {return new Date(start.getTime());}
}
```

COPY

COPY

BEST PRACTICE

Use defensive copying if your immutable class contains mutable fields

Maybe you can avoid mutable fields

- Date is mutable; **use Joda-Time instead**
- SimpleDateFormat is mutable; **use Joda-Time instead**
- Standard Java collections are mutable (*even the "unmodifiable" ones*); **use the immutable Guava collections instead**

BEST PRACTICE

Use Joda-Time and Guava

Speaking of Guava

Why Guava?

- It's already written (reinventing takes too long and you make mistakes)
- It's extensively tested
- It's optimized
- It's constantly being evolved and improved

BEST PRACTICE

Know and use the libraries — especially Guava

Guava awesomes

```
Map<String, Map<Integer, Budget>> m = Maps.newHashMap();

ImmutableSet<Integer> s = ImmutableSet.of(1, 3, 9, 6);

Collection<?> b = filter(a, notNull());

Multimap<String, Integer> scores = HashMultimap.create();
scores.put("Alice", 75);
scores.put("Alice", 22);
scores.put("Alice", 99);
System.out.println(Collections.max(scores.get("Alice")));

Splitter.on(',').trimResults().omitEmptyStrings().split("63,22,,   9");
```

More Guava wins

```
@Override
public int compareTo(final Dog d) {
        return ComparisonChain.start().compare(
         age, d.age).compare(breed, d.breed).result();
}
```

Precondition

```
checkArgument(count > 0, "must be positive: %s", count);
```

BEST PRACTICE

Use preconditions (to remove if-statements from your code)

Speaking of Joda-Time

Check it out

Know the concepts!

- *Instant* - DateTime, DateMidnight, MutableDateTime
- *Partial* - LocalDateTime, LocalDate, LocalTime
- *Duration* (a number of millis)
- *Interval* (two instants)
- *Period* (in human-understandable terms, e.g, days/weeks)
- *Chronology*

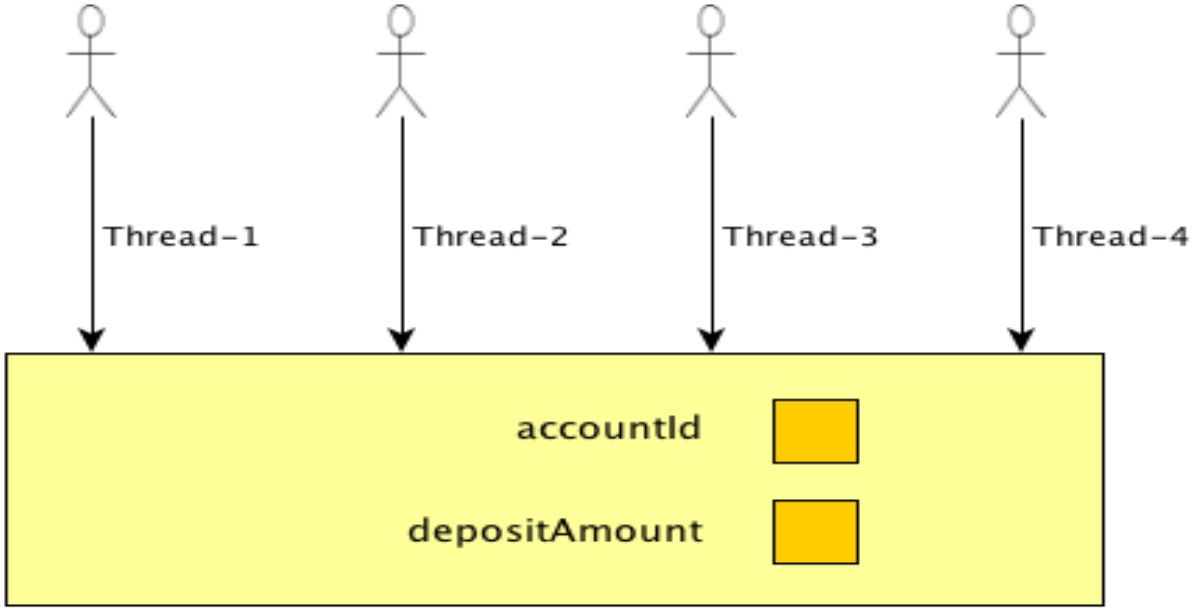
What's wrong here?

```
@Controller
public class DepositController {
    private int accountId;
    private BigDecimal depositAmount;

    @POST @RequestMapping("/deposit/{id}")
    public Response handleDeposit(@Param("id") String id,
            String amount) {
        this.accountId = validateId(id);
        this.depositAmount = validateAmount(amount);
        service.makeDeposit(accountId, depositAmount);
```

Grrrrr — Singletons!

Each request is running on a separate thread most likely



The fix is obvious, isn't it?

```
public class DepositController {
    @POST @RequestMapping("/deposit/{id}")
    public Response handleDeposit(@Param("id") String id,
            String amount) {
        int accountId = validateId(id);
        BigDecimal deposit = validateAmount(amount);
        service.makeDeposit(accountId, deposit);
```

BEST PRACTICE

Don't put state in shared singletons like controllers, services, and daos

What's wrong here?

```
public class FeedConfig {
    public FeedConfig(String feedFileId,
            String feedId, String name, String url,
            String compressionType,
            ConversionType conversionType,
            ProviderType providerType,
            boolean createsNewListing) {
        . . .
    }
```

Too Many Parameters!

- When you *call* a constructor (or any method, for that matter) with a zillion arguments, what do they all mean?
- In dynamic languages, we pass hashes (usually)
- Can we do that in Java?
- What is *wrong* with you?

Fluent Builders

```
config = new FeedFileConfigBuilder()
        .feedFileId("483")
        .feedId("22")
        .name("iusa-CA")
        .url("ftp://example.com/iusa/ca/feed")
        .compressionType("zip")
        .conversionType(Conversion.CUSTOM)
        .createsNewListing(false)
        .build();
```

The builder is mutable but the object that is built is immutable.

BEST PRACTICE

Consider builders for classes with many properties

What's wrong here?

```
start = System.currentTimeMillis();
price = computePrice();
finish = System.currentTimeMillis();
logger.debug("Computed price of $"
    + new DecimalFormat("#0.00").format(price)
    + " in " + (finish - start) + " milliseconds");
```

- Timing clutters the code .... it's an aspect
- Should use a currency formatter (i18n)
- Is that the only formatter we'll need?
- And what if we are not in debug mode?

Making Logging Efficient

```
if (logger.isDebugEnabled()) {
    logger.debug(. . .);
}
if (logger.isInfoEnabled()) {
    logger.info(. . .);
}
```

BEST PRACTICE

Wrap logging calls for complex messages in isXXXEnabled() conditions

Logging Levels

FATAL — app not expected to recover
ERROR — error that app might recover from
WARN — take notice, potentially harmful
INFO — coarse-grained app progress
DEBUG — to help you debug
TRACE — super-fine-grained progress

BEST PRACTICE

Know and use the proper logging levels

What's wrong here?

```
// tagging data case 2: tags field is not null and primaryTagId is not null, but
// primary tag is not included in the tags field, append primaryTagId
tagIds = (StringUtils.isNotBlank(tagIds)&&StringUtils.isNotBlank(primaryTagId)
    ? (tagIds.contains(primaryTagId)
      ? tagIds
       : new StringBuilder(tagIds).append(",").append(primaryTagId).toString())
   : tagIds);
```

What's wrong here?

```
// ************************************* //
// ***** INSTANCE METHODS ***** //
// ************************************* //
/**
 * Returns the count.
 * @return the count
 */
public int getCount(/* no args */) {
    // NOTE: count is a field
    return count; // return the count
} // end of instance method getCount
```

You KNOW how I feel about comments!

What's wrong here?

```
public void registerItem(Item item) {
    if (item != null) {
        Registry registry = store.getRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getId());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

From Robert C Martin's Clean Code book (page 110).

Don't return null

- Actually, there are **too many** null checks, not too few
- Returning null as a normal case forces users to clutter their code

BEST PRACTICE

Don't return null! For collections, return an empty collection. For plain objects, throw an exception or return a special case object.

What's wrong here?

```
public void writeToFile(String filename, List<String> lines) {
    try {
        Writer writer = new PrintWriter(new FileWriter(filename));
        for (String line : lines) {
            writer.append(line);
            writer.append(System.getProperty("line.separator"));
        }
    } catch (IOException e) {
        logger.error("FAILED WRITING TO: " + filename + ", RESUMING");
    }
}
```

OH! - Not closing!! - Won't flush!!!!

Improved, but still wrong-ish

```
public void writeToFile(String filename, List<String> lines) {
    Writer writer = null;
    try {
        writer = new PrintWriter(new FileWriter(filename));
        for (String line : lines) {
            writer.append(line);
            writer.append(System.getProperty("line.separator"));
        }
    } catch (IOException e) {
        logger.error("FAILED WRITING TO: " + filename + ", RESUMING");
    } finally {
        if (writer != null) {
            try {
                writer.close();
            } catch (IOException e) {
                logger.error("FAILEDTO CLOSE: " + filename + ", RESUMING");
            }
        }
    }
}
```

The code duplication is bad, too

You're kidding me? Added 8 lines just to close the file?!?!?

Getting Better

```
public void writeToFile(String filename, List<String> lines) {
    PrintWriter writer = null;
    try {
        writer = new PrintWriter(new FileWriter(filename));
        for (String line : lines) {
            writer.println(line);
        }
    } catch (IOException e) {
        logger.error("FAILED WRITING TO: " + filename + ", RESUMING");
    } finally {
        if (writer != null) {
            writer.close();
        }
    }
}
```

PrintWriter.close() eats the IOException, if any, saving a few lines....

A Little Bit Better

```
public void writeToFile(String filename, List<String> lines) {
    PrintWriter writer = null;
    try {
        writer = new PrintWriter(new FileWriter(filename));
        for (String line : lines) {
            writer.println(line);
        }
    } catch (IOException e) {
        logger.error("FAILED WRITING TO: " + filename + ", RESUMING");
    } finally {
        IOUtils.closeQuietly(writer);
    }
}
```

IOUtils.closeQuitely from Apache Commons is null-safe, saving a couple more lines....

Solutions

Guava has the **Files** class with utility methods that guarantee the file will be closed no matter what. Check it out for homework....

Or just use Java 7

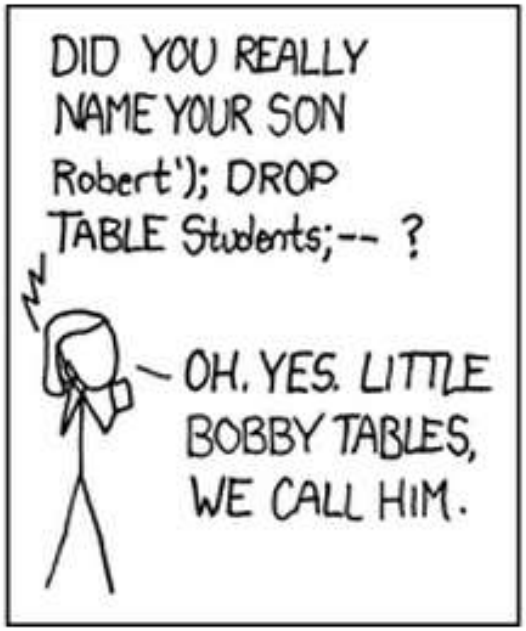
```
try (PrintWriter output = new PrintWriter(new FileWriter(filename)))  {
    for (String line: lines) {
        output.println(line);
    }
} catch (IOException e) {
    logger.error("FAILED WRITING TO: " + filename + ", RESUMING");
}
```

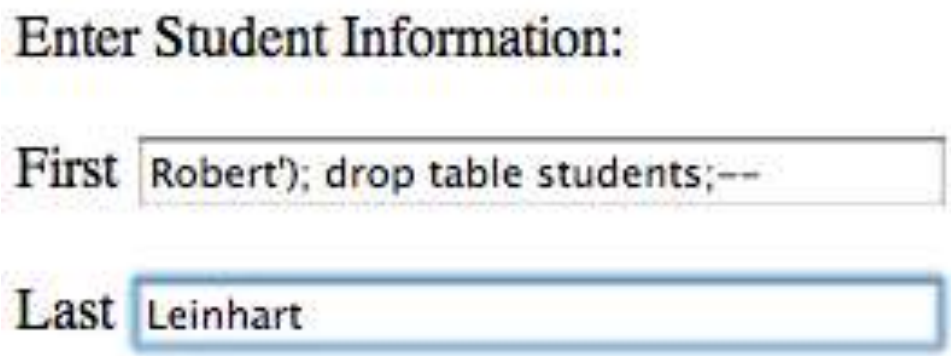
What's wrong here?

```
String first = s                                  n("first");
...
template.upda                                     s ("
    + " ss.nextv
```

1. Parameter p mmed together in the same file ural sense.
2. Things look l ne query call.
3. Is that it? Wa nething seems wrong here..

SQL Injection





insert into students values (ss.nextval, 'Leinhart', 'Robert'); drop table students;--')

JDBC Parameters

Parameters prevent injection attacks and help with the efficiency gain in prepared statements

```
template.update("insert into students values ("
    + " ss.nextval, ?, ?)", last, first);

template.update("insert into students values ("
    + " ss.nextval, :last, :first)", map);
```

BEST PRACTICE

Always use SQL parameters

What's wrong here?

```
public class SuperImportantJob {
    public static void main(String[] args) {
        try {
            doSomething();
            doTheNextThing();
            doTheLastThing();
        } catch (Exception e) {
            logger.fatal("Job Failed", e);
        }
    }
}
```

(Job = standalone application)

HINT: THE CONFIGURATION MANAGEMENT TEAM IS NOT HAPPY WITH YOU TODAY

You are not the center of the universe

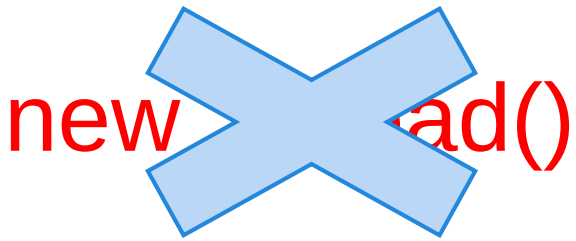
```
public static void main(String[] args) {
    try {
        ....
    } catch
        logg
        Syst
    }
}
```

Your app got called by a bash script or was launched by a Jenkins job. And someone else's job is gonna follow yours.

BEST PRACTICE

Return non-zero status codes from failing jobs (via System.exit or exception)

And what is wrong with this?



BEST PRACTICE

Don't create your own threads; use an executor service as it will do most of the thread pool management for you

Serialization

Some serialization questions for homework...

- What is the serialVersionUID?
- What happens if you don't explicitly specify a value for this field?
- Why is it a best practice to always specify a value for this field?
- What happens if you do specify a value, then change your class, but do not change it?
- What does Josh Bloch say about all this?

A few more practices

- Write DRY code and DAMP tests
- Put calls in the proper place, e.g., don't formulate response JSON or HTML in a dao
- Avoid magic numbers (except maybe 0, 1); use private static final (constants)
- Superclasses should not know about their subclasses
- Consider domain-specific exceptions over built-in general purpose exceptions
- Avoid double negatives, e.g., if (!notFound())
- Use BigDecimal, not double, for money

But wait, there are more!

- Comment only when you must
- Get rid of obsolete, redundant, inappropriate, rambling, crappily written comments
- DELETE COMMENTED OUT CODE
- Follow Uncle Bob's naming guidelines
- No Hungarian Notation, please
- Avoid bad names: tmp, dummy, flag
- Don't write functions that expect booleans or nulls or things to switch on
- Avoid "out parameters", return things instead
- Prefer the single-return style

Ooooh! Yet more Java advice

- Don't make something static when there is an obvious object it can operate on
- Override hashCode if you override equals
- Don't make a new java.util.Random every time
- Put configurable data in their own classes or resources
- Don't put constants in interfaces just so you can implement the interface to avoid qualification; use import static instead
- Make constructors private when you should

Aaah the best practice aliens have control of my brain

- Use enums, not lame int constants
- Inner classes for observers or row mappers often look nicer as nested static classes (and are ever so slightly more efficient)
- Don't do string concatenation in a loop
- Use the AtomicXXX classes
- Make sure .equals() checks for null
- Never call .equals(null)

Clean your code with Java 7

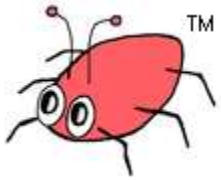
- Strings in switch statement
- Binary integral literals
- Underscores in numeric literals
- Multi-catch and more precise rethrow
- Generic instance creation type inference
- Try-with-resources statement
- Simplified varargs method invocation

http://www.javacodegeeks.com/2011/11/java-7-feature-overview.html

More Java 7 Goodness

ThreadLocalRandom
ForkJoinPool and ForkJoinTask
Phaser
NIO 2.0
Zip File System Provider
Elliptic Curve Cryptography
Disabling of weak cryptographic algorithms
Sockets Direct Protocol

Where can you find more info?



http://findbugs.sourceforge.net/bugDescriptions.html

http://checkstyle.sourceforge.net/availablechecks.html

http://pmd.sourceforge.net/pmd-5.0.4/rules/index.html

Sonar

Sonar can take output from PMD, Checkstyle, etc. and present it to you in a useful way.

Most violated rules Any severity More

Modifier Order	52
Redundant Modifier	27
Unused Modifier	27
Member name	15
Final Field Could Be Static	14

Most violated resources

- CheckoutController
- ProcessService
- MatchWorkspaceDao
- DuplicateListingsWorkspaceDao
- InvalidController

Longest unit tests More

BlacklistDaoTest	10.9 sec
DuplicateListingsCheckoutServiceTest	311 ms
CheckoutControllerTest	76 ms
TagDaoTest	53 ms
MatchRawDataDaoTest	52 ms

Highest untested lines

- BlacklistValidator
- BlacklistController
- SpringConfig
- ProxyService
- ProfileProxyController

Highest complexity More

CheckoutController	39
CheckoutValidator	35
MatchingListingProcessService	30
ProcessController	30

Highest average method complexity

- ProcessUtil
- CheckoutServiceUtil
- CheckoutValidator
- BlacklistValidator

Homework (Hey why not?)

1. Skim the list of FindBugs Bug Descriptions
2. Find an item on JavaPractices.com that you disagree with
3. Read an article on serialization
4. Run FindBugs, using the highest possible analysis settings, on a Java Project that you worked on
5. Refactor some existing code using Guava and Java 7

That's it

Questions or comments?