

Design Patterns

Evgeny Borisov

bsevgeny@gmail.com

Who are you?

Big Data & Java Technical Leader

Mentoring

Consulting

Lecturing

Writing courses

Writing code

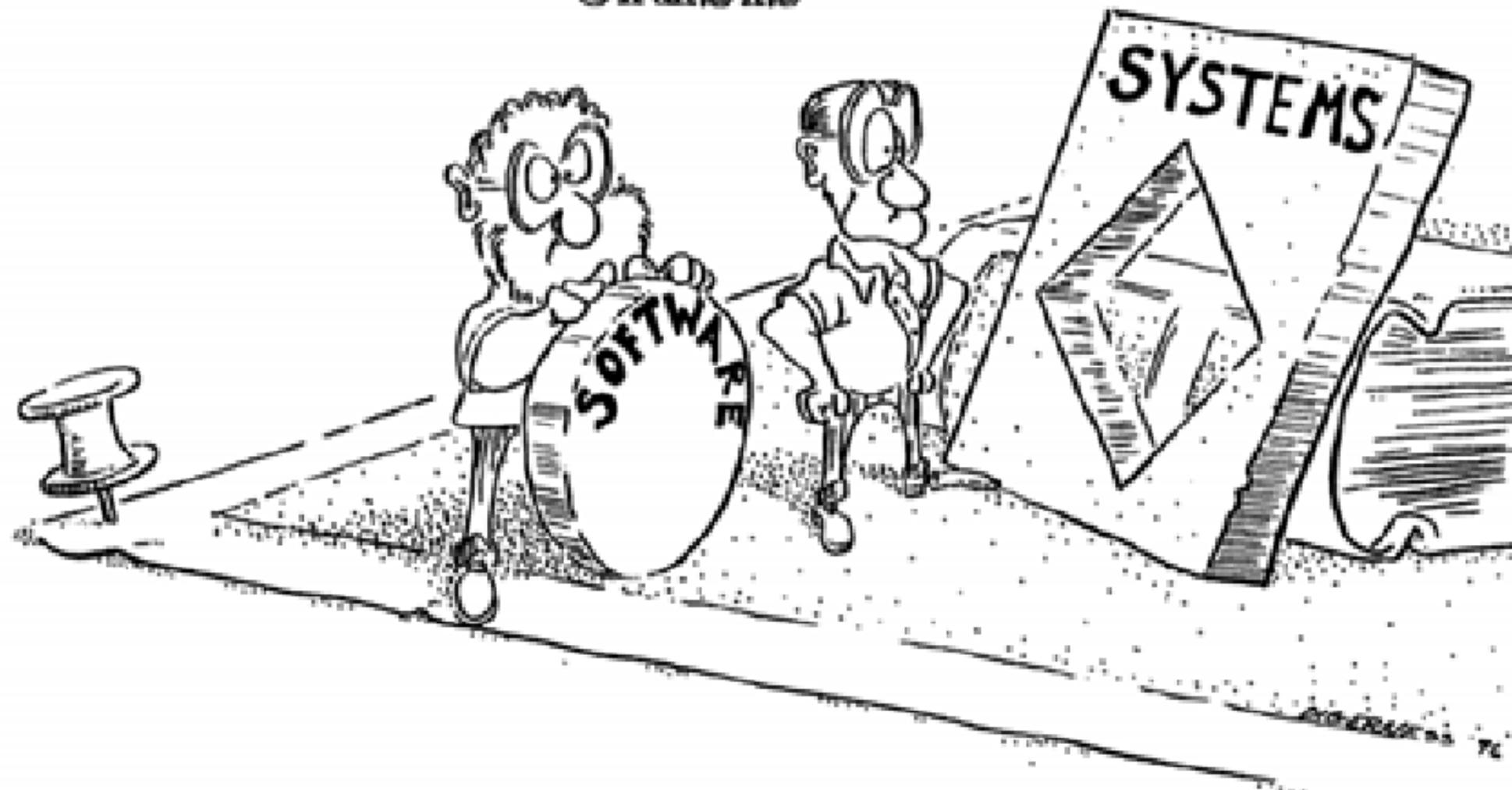
Databases

Big Data

Intelligen



THE SOFTWARE CRISIS





BLACK FRIDAY

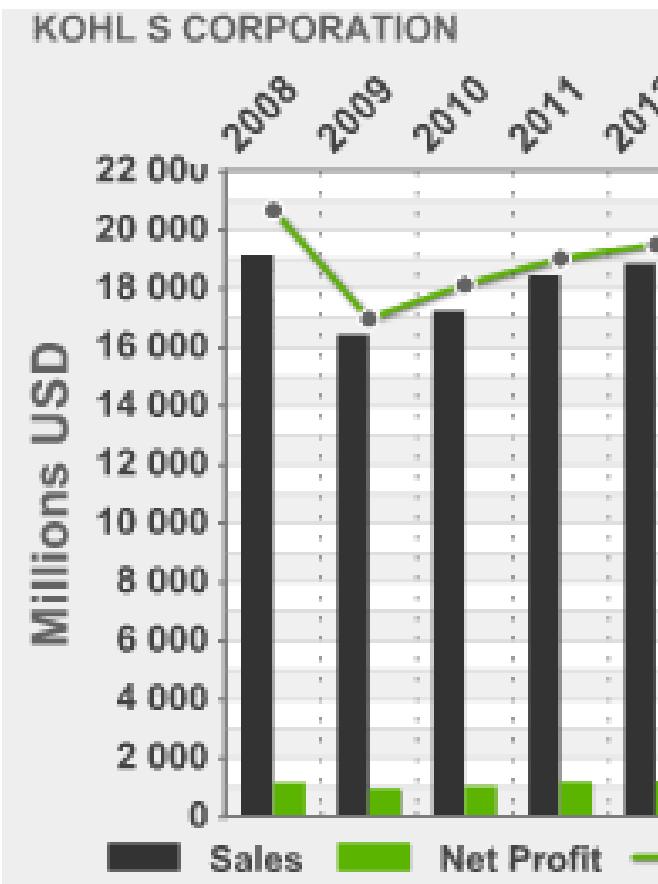
BLACK FRIDAY – forever!!!



Web market



KOHL'S



What happened in 2009?



Real-Life Market Situations



A Service of CNN, Fortune & Money

The Dow's technical glitch, dissected

Publisher expects normal operations on back-up system, but still looking for root cause of Tuesday's glitch.

The publisher of the Dow industrials said that a system problem starting at 1:50 p.m. ET on Tuesday, amid unusually heavy trading volume, caused a 70-minute lag during which the value of the market measure lagged the declines in the underlying stocks.



Search PC World

Sea

Home News Hardware Reviews Software Reviews How-To Videos Downloads



FIND A REVIEW

Select Category

- Audio & Video
- Business Center
- Cameras
- Cell Phones & PDAs
- Communications
- Components & Peripherals

Read More About: iPhone

For all your IT

iPhone Activation Disasters

Three hours after getting my hands on one, I am ready to drop the thing from the 44th floor of the New York Hilton.

Jim Dalrymple, Macworld

Friday, June 29, 2007 8:00 PM PDT

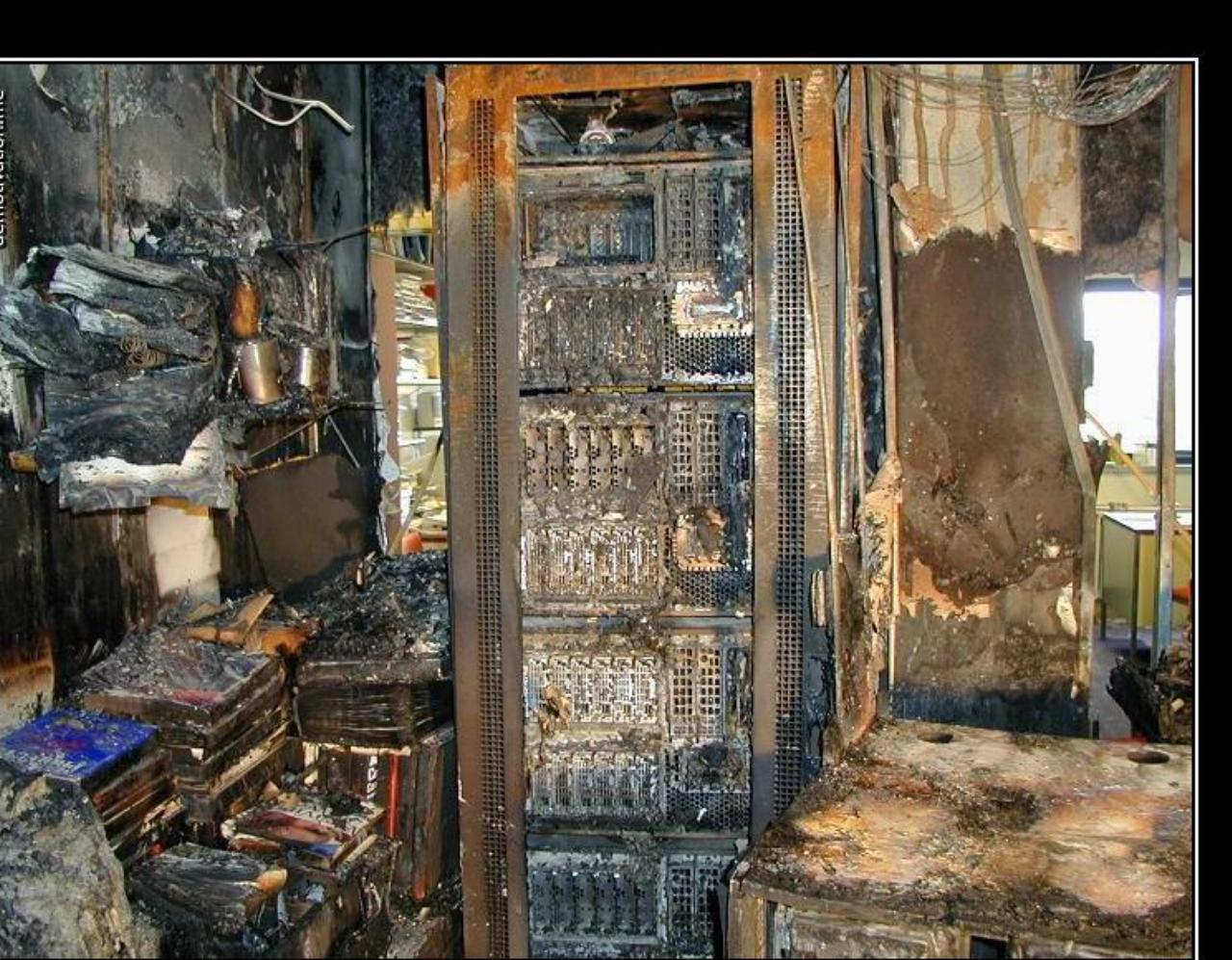
Black Friday Shoppers Crash Kohl's Web Site

By INYOUNG HWANG Bloomberg NEWS

Kohl's Corp.'s Web site crashed because of online traffic on Black Friday, a day when retailers offer bargains and the traditional start of the Christmas shopping season

This article was published November 28, 2009 at 4:37 a.m. Business, p.34

We need more servers!!!



• • •



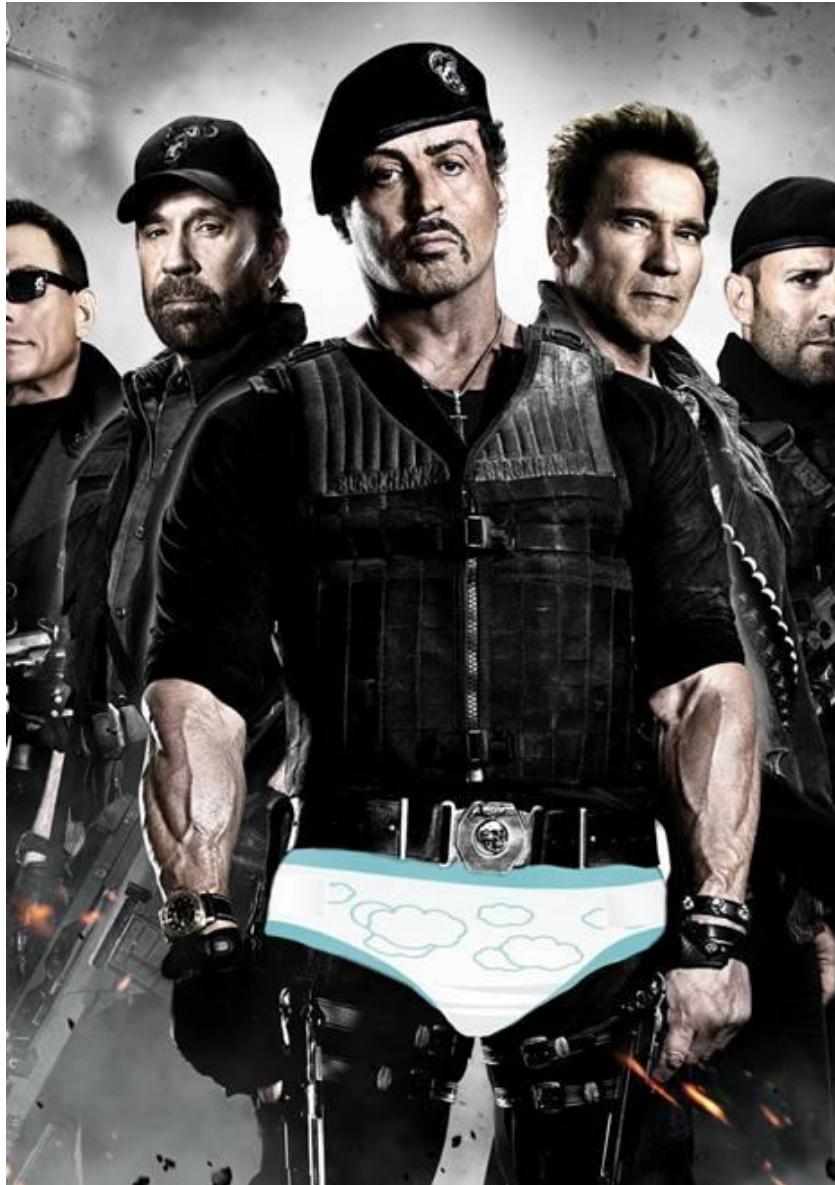
ONE DOES NOT SIMPLY



Because traditional architecture ...



No it is time to change old heroes



Your first task purposes

- Correct using of switch and enum

**WHY JUST NOT TO
USE**

SWITCH???

Remember! People will follow you...



We love you, Switch ...

```

public DistribHandler resolveHandler(Integer.valueOf(documentObject.getDocumentType().getValue())) {
    switch (documentObject.getDocumentType().getValue()) {
        case PDF_STORAGE:
            fileContainer = new PdfRecordFile();
            getPdfFromStorage(fileContainer, documentObject);
            break;
        case PDF_SRC:
            fileContainer = new PdfRecordFile();
            fillObjectsForPdf(fileContainer, documentObject);
            break;
        case PDF_WS:
            fileContainer = new WsPdfRecordFile();
            getPdfFromPdfWs(fileContainer, documentObject);
            break;
        case LIS:
            fileContainer = new PdfRecordFile();
            getLisDocument(j, fileContainer, documentObject);
            break;
        case IMAGE:
            fileContainer = new PdfRecordFile();
            getPdfFromImageDocument(j, fileContainer, documentObject);
            break;
        case FORM:
            fileContainer = new PdfRecordFile();
            getFormDocument(fileContainer, documentObject);
            break;
    }
}

switch (value.getNumericValue()) {
    case 1:
        text = MessageFormat.format("{0} {1}, האעה למייל", emailRequest.getSubject());
        break;
    case 2:
        text = MessageFormat.format("{0} {1}, רביישת במייל");
        break;
    case 3:
        text = MessageFormat.format("{0} {1}, חידוש במייל");
        break;
    case 4:
        text = MessageFormat.format("{0} {1}, שינויים במייל");
        break;
    case 5:
        text = MessageFormat.format("{0} {1}, מכתב חשב ערך");
        break;
    case 8:
        text = MessageFormat.format("{0} {1}, מכתב");
        break;
    case 9:
        text = MessageFormat.format("{0} - {1} {2}, רביישת ביטוח נפשות");
        break;
    case 17:
        text = MessageFormat.format("{0} {1}, רביישת ביטוח נפשות");
        break;
    case 10:
        text = MessageFormat.format("{0} - {1} {2}, מכתב");
        break;
    case 13:
        text = MessageFormat.format("{0} - {1} {2}, מכתב");
        break;
    case 11:
        text = emailRequest.getSubject();
        break;
    case 12:
        if (!resources.getBrandKey().isBituhYashir()) {
            text = format("%{0} %{1} %", brandHebName);
        } else {
            text = format("%{0} %{1} %", brandHebName);
        }
        break;
    case 14:
        text = MessageFormat.format("{0} - {1}, ביטוח תאגונת אושנות - השיקט הנפשו של");
        break;
    case 15:
        text = MessageFormat.format("{0} {1}, בקשה לאירוע קשר לחידוש ביטוח");
        break;
    case 16:
        text = MessageFormat.format("{0} {1}, האעה למייל");
        break;
    default:
        if (item.getAddresseeCode().length() >= 1) {
            String itemCode = item.getAddresseeCode().trim();
            if ("0".equals(itemCode)) {
                String addressee = resources.getBrandKey().getBrandName();
                currentPath.setBasketPath(item.getSeqNr(), addressee);
                icyDatFileConsumer();
            } else {
                String addresseeCode = item.getAddresseeCode();
                currentPath.setBasketPath(item.getSeqNr(), addresseeCode);
                icyDetailsConfConsumer();
            }
        }
}

```

Switch is anti-pattern, don't use switch in Java

- Unreadable code
- The reason of God class born
- Hard to maintain
- RuntimeException like NullPointer
- Someday somebody will forget “break” and than...
- Commit is very difficult!!! Merge per each commit

Do you love to mess with dirty code?



What can replace the switch?

- Map
- Enum
- Annotations

Topic of the game: Enum



10 euro question:

How many constructors can have an enum?

- A – 1
- B – 57
- C – As many as you want
- D - Zero

10 euro question:

How many constructors can have an enum?

- A – 1
- B – 57
- C – As many as you want
- D - Zero

50 euro question:

What is the best way to compare enums?

- A – ==
- B – equals
- C – with reflections
- D – with lot of beer

50 euro question:

What is the best way to compare enums?

- A – ==
- B – equals
- C – with reflections
- D – with lot of beer

100 euro question:

Enum's constructor is:

- A – public
- B – protected
- C – private
- D – there is no correct answer

100 euro question:
Enum's constructor is:

- A – public
- B – protected
- C – private
- D - There is no correct answer

250 euro question:

Choose correct answer:

- A – enum can have only static methods
- B – enum can not have static methods
- C – enum can have both static and non-static methods
- D – What methods?? Enum is just a constant

250 euro question:

Choose correct answer:

- A – enum can have only static methods
- B – enum can not have static methods
- C – enum can have both static and non-static methods
- D – What methods?? Enum is just a constant

500 euro question:

How many methods in enum can be overridden?

- A – 1
- B – 2
- C – 0
- D – 3

500 euro question:

How many methods in enum can be overridden?

- A – 1
- B – 2
- C – 0
- D – 3

1000 euro question:

Who can create an enum object:

- A - Chuck Norris
- B - Classloader
- C - Garbage collector
- D - With reflection

1000 euro question:

Who can create an enum object:

- A - Chuck Norris
- **B - Classloader**
- C - Garbage collector
- D - With reflection

Lets summarize

- Enum can have any number of constructors, but usually it will be one, and only classloader can invoke it.
- Enum “class” define not only the metadata, but also all objects from that type.
- All enums objects are final and static
- But they are like regular objects
 - Methods
 - Properties

What else we know about Enum

- Only `toString()` can be overridden
- So `equals` (which by default implemented as `==`) will never be overridden
- So it is better to compare enums with `==` in order to not to think about NPE

The next question is:

- What should enum return if requested object is not exists?
 1. Null
 2. Empty object
 3. Throw an exception
- Which exception is better?

Checked
against
Round 1



Runtime



Checked Exceptions



Obligates handling. No chance that someone, who use the method will forget to put try and catch or throw the exception further

In Runtime Exception handling can be forgotten. More than that, some who use the method not necessarily knows, that exception can be thrown

Runtime Exceptions

But clean code without
try & catch

What is written in 99%
inside the catch?

Correct: log.error(e)

But it will happen in any case if the
exception isn't caught



Checked
against
Round 2



Runtime



Checked Exceptions



Checked exception handle mechanism, propagate programmers make more correct handle.

Exception log message usually more informative

In Runtime Exception you can see something like: NullpointerException without any additional information

Runtime Exceptions

Not informative logs are exactly of developers who return null instead of throwing an exception with message

And than even if you don't have try & catch you still will see the message in the log



Runtime Exceptions

Tell me honestly,
Do you ever saw empty catches?
This is real problem!!!



Checked



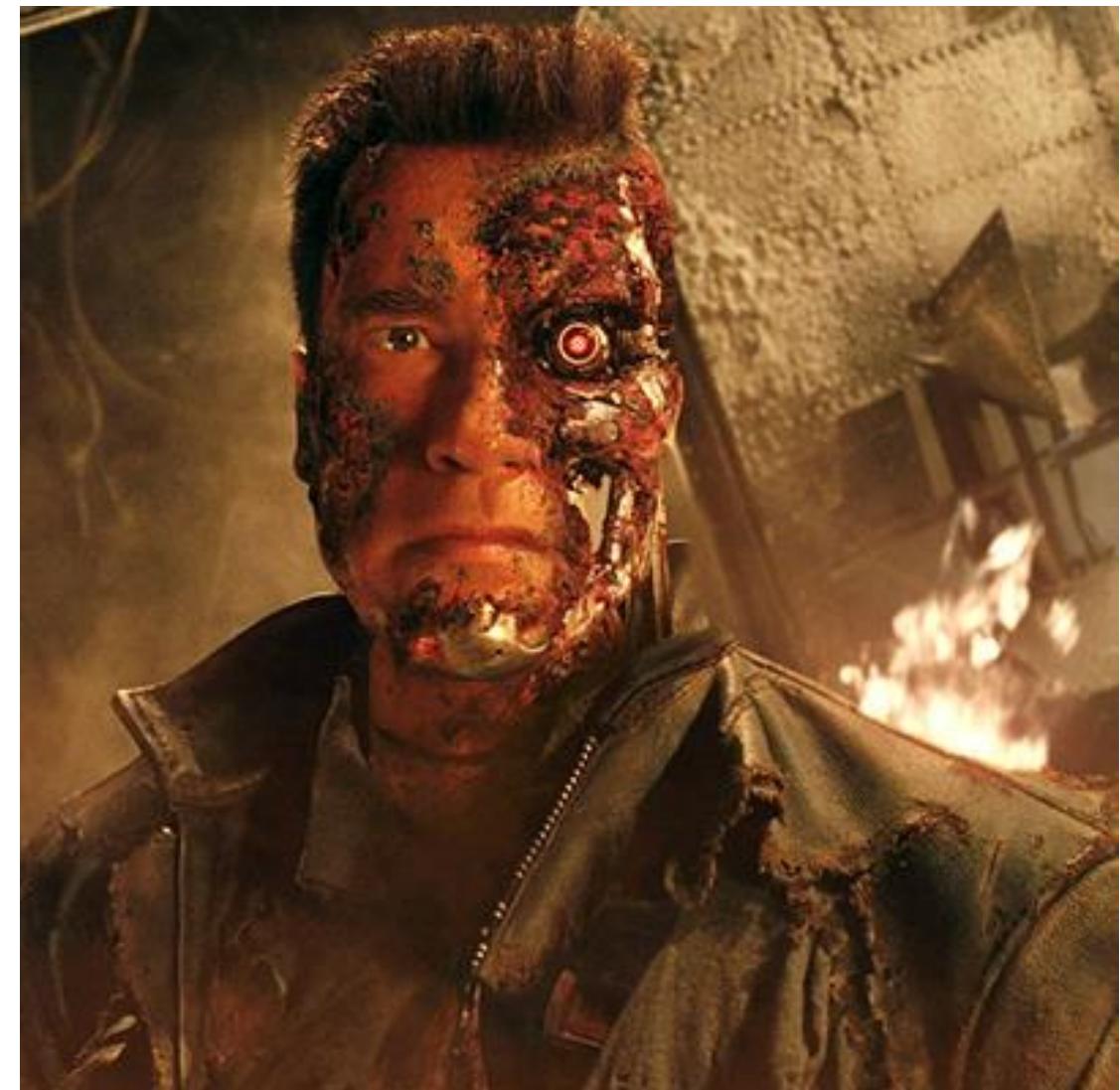
against
Round 3

Runtime



Checked Exceptions

But you always can add throws to method signature and than it will behave exactly like runtime



Runtime Exceptions

Not always! Exception becomes a part of a signature and it is not always that we can add it to signature



After the fight



Who is the best, I can hardly say, but Runtime Exceptions are better 100%



Ok, and what is the
annotation solution?

Stop! How deep you familiar with Reflection?



What is your level?

1. Reflection? Never heard of it
2. I know what it is
3. I'm using it
4. I wrote my own custom annotations and scanned them at runtime
5. I know the difference between `RetentionPolicy.RUNTIME`,
`RetentionPolicy.SOURCE` и `RetentionPolicy.CLASS`
6. I know what is the default retention policy

Reflections

Evgeny Borisov

Task – Zoo Game

- Write tree of animal classes (Tiger, Dog, Cow...)
- All classes extends from Animal and override method makeSound.
- Write Factory class which will have method createZoo.
- This method will get an integer and will return list of random animals equal to this number
- Test your application.

Animal

```
public abstract void makeSound();
```

Dog

```
@Override  
public void makeSound() {  
    out.println("Muuuuuu");  
}
```

Tiger

```
@Override  
public void makeSound() {  
    out.println("au au au");  
}
```

Cow

```
@Override  
public void makeSound() {  
    out.println("Rrrrrrr");  
}
```

```
public interface AnimalFactory {  
    List<Animal>createZoo(int number);  
}  
  
public class SimpleAnimalFactory implements AnimalFactory {  
    public List<Animal> createZoo(int number) {  
        ArrayList<Animal> animals = new ArrayList<Animal>();  
        for (int i = 0; i < number; i++) {  
            animals.add(createRandomAnimal());  
        }  
        return animals;  
    }  
  
    public Animal createRandomAnimal() { ... }  
}
```



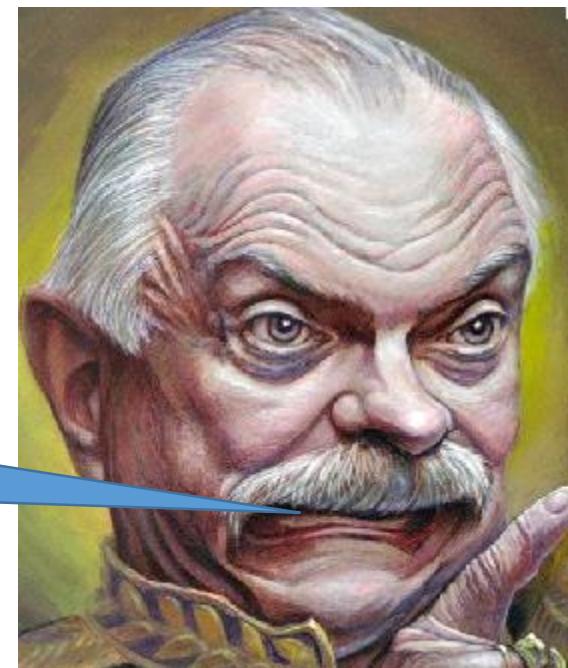
And what code
is there?

CreateRandomAnimal Method

```
public Animal createRandomAnimal() {  
    int randomNumber = (int) (Math.random() * 3);  
    switch (randomNumber) {  
        case 0:  
            return new Cow();  
        case 1:  
            return new Tiger();  
        case 2:  
            return new Dog();  
    }  
    return null;  
}
```

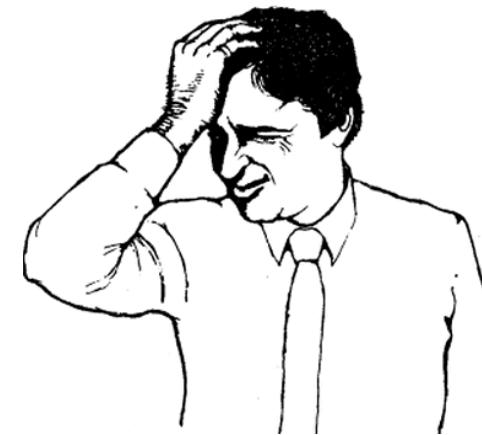
Add a cat please

שינוי איפיון



```
public Animal createRandomAnimal() {  
    int randomNumber = (int) (Math.random() * 3);  
    switch (randomNumber) {  
        case 0:  
            return new Cow();  
        case 1:  
            return new Tiger();  
        case 2:  
            return new Dog();  
        case 3:  
            return new Cat();  
    }  
    return null;  
}
```

Yes... Some day it
will happen



Why this solution is bad?

- Hard to maintain (someday you will forget)
- Don't use switch! Remember?
- New animal type require change in factory class

Good Solution

- Animal factory should read all animal class names in bootstrap and add them to its cache
- createRandomAnimal method should fetch random animal class type from cache and create new instance from it

HOW???





Introduction

- With the reflection API you can examine or manipulate:
 - Classes
 - Get modifiers, fields, methods, constructors, and superclasses.
 - Create an instance of a class whose name is not known until runtime.
 - Interfaces
 - Get constants and method
 - Objects
 - Get class of an object
 - Get and set the value of field, even if the field name is unknown to your program until runtime
 - Invoke a method on an object, even if the method is not known until runtime
 - Arrays
 - Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

Uses

- Software development tools
 - Frameworks, services, factories
 - Debuggers
 - GUI builders
 - implementing drag and drop of components
 - etc.

The Reflection API

- `java.lang` package
 - `Class` – Represents, or reflects, classes and interfaces.
 - `Object` – Provides the `getClass` method.

The Reflection API – `java.lang.reflect`

- Reflection package provides API for:
 - Examining Classes
 - construct new class instances and new arrays
 - information about class's modifiers
 - Manipulating Objects
 - access and modify fields of objects and classes
 - invoke methods on objects and classes
 - Arrays
 - access and modify elements of arrays

class Class

- For each class, the Java Runtime Environment (JRE) maintains an immutable `Class` object that contains information about the class. (metadata)
- A `Class` object represents, or reflects, the class
- `class Class` include methods which return `Constructor`, `Method`, and `Field` objects

```
class Class
```

- **Class** objects also represent *interfaces*.
 - `isInterface` method
 - Determines if the specified `Class` object represents an interface type
 - You invoke `Class` methods to find out about an interface's *modifiers*, *methods*, and public *constants*

Retrieving Class Objects

- Retrieving Class Objects
 - invoke Object.getClass
 - Class c = unknownObject.getClass()
 - getSuperclass method
 - List t = new ArrayList();
Class c = t.getClass();
Class s = c.getSuperclass();

Retrieving Class Objects

- **forName** method - If the class name is unknown at compile time

- ```
String className = "java.io.File";
...
Class c = Class.forName(className);
```

  - The method `forName` creates (instantiates) a `Class` object

# Examining Classes

- **Constructor** – Provides information about, and access to, a constructor for a class.  
Allows you to instantiate a class dynamically.
- **Modifier** – Provides static methods and constants that allow you to get information about the access modifiers of a class and its members.

# Class Constructors

- Class's `getConstructors` method
  - returns an array of `Constructor` objects
  - `Constructor` class includes methods to
    - retrieve constructor's name, set of modifiers, parameter types, and set of throwable exceptions.
    - create a new instance of the `Constructor` object's class with the `Constructor.newInstance` method

# Class Constructors

```
// retrieve constructors' information
static void showConstructors(Object o) {
 Class c = o.getClass();
 Constructor[] theConstructors =
 c.getConstructors();
 for (int i = 0; i < theConstructors.length; i++) {
 System.out.print("(");
 Class[] parameterTypes =
 theConstructors[i].getParameterTypes();
 for (int k=0; k<parameterTypes.length; k++) {
 String parameterString =
 parameterTypes[k].getName();
 System.out.print(parameterString + " ");
 }
 System.out.println(")");
 }
}
```

# Constructors - Creating an Object

- Creating Objects
  - Create an instance (new object) of class which is unknown until runtime
    - Class's `newInstance()` method
      - Creates a new instance of the class represented by this `Class` object
      - used for parameterless constructors
    - Constructor's `newInstance(Object[] args)` method
      - Uses the constructor represented by this `Constructor` object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters.
      - used for constructors with parameters

# Constructors - Creating an Object

```
// Constructor without parameters
static Object createObject(String className) {
 Object object = null;
 try {
 Class classDefinition = Class.forName(className);
 object = classDefinition.newInstance();
 } catch (InstantiationException e) {
 // if this Class represents an abstract class,
 // an interface, an array class, a primitive
 // type, or void; or if the instantiation
 // fails for some other reason.
 System.out.println(e);
 } catch (IllegalAccessException e) {
 //if the class or initializer is not accessible
 System.out.println(e);
 } catch (ClassNotFoundException e) { System.out.println(e);
 }
 return object;
}
```

# Constructors - Creating an Object

```
// Constructor with parameters
public static Object createObject(Constructor constructor,
 Object[] arguments) {

 System.out.println ("Constructor: " + constructor.toString());
 Object object = null;
 try {
 object = constructor.newInstance(arguments);
 System.out.println ("Object: " + object.toString());
 return object;
 } catch (InstantiationException e) {
 System.out.println(e);
 } catch (IllegalAccessException e) {
 System.out.println(e);
 } catch (IllegalArgumentException e) {
 System.out.println(e);
 } catch (InvocationTargetException e) {
 System.out.println(e);
 }
 return object;
}
```

# class Class – Modifiers

- Class modifiers: public, abstract, final, ...
  - Method `getModifiers` - retrieve a set of modifiers.
  - class Modifier includes static methods: `isPublic`, `isAbstract`, `isFinal`, ...

```
public static void printModifiers(Object o) {
 Class c = o.getClass();
 int m = c.getModifiers();
 if (Modifier.isPublic(m))
 System.out.println("public");
 if (Modifier.isAbstract(m))
 System.out.println("abstract");
 if (Modifier.isFinal(m))
 System.out.println("final");
}
```

# class Class – interfaces

- Identifying the Interfaces Implemented by a Class
  - getInterfaces method

```
static void printInterfaceNames(Object o) {
 Class c = o.getClass();
 Class[] theInterfaces = c.getInterfaces();
 for (int i=0; i<theInterfaces.length; i++) {
 String interfaceName =
 theInterfaces[i].getName();
 System.out.println(interfaceName);
 }
}
```

# Manipulating Objects

- **Field** – Provides information about, and dynamic access to, a field of a class or an interface.
- **Method** – Provides information about, and access to, a single method on a class or interface. Allows you to invoke the method dynamically.

# Class Fields

- Class's `getFields` method
  - returns an array of `Field` objects
    - includes the fields of all superclass of the class and all the interfaces it implements.
  - `Field` class includes methods to
    - retrieve the field's name, type, set of modifiers
    - get and set the value of a field

# Class Fields

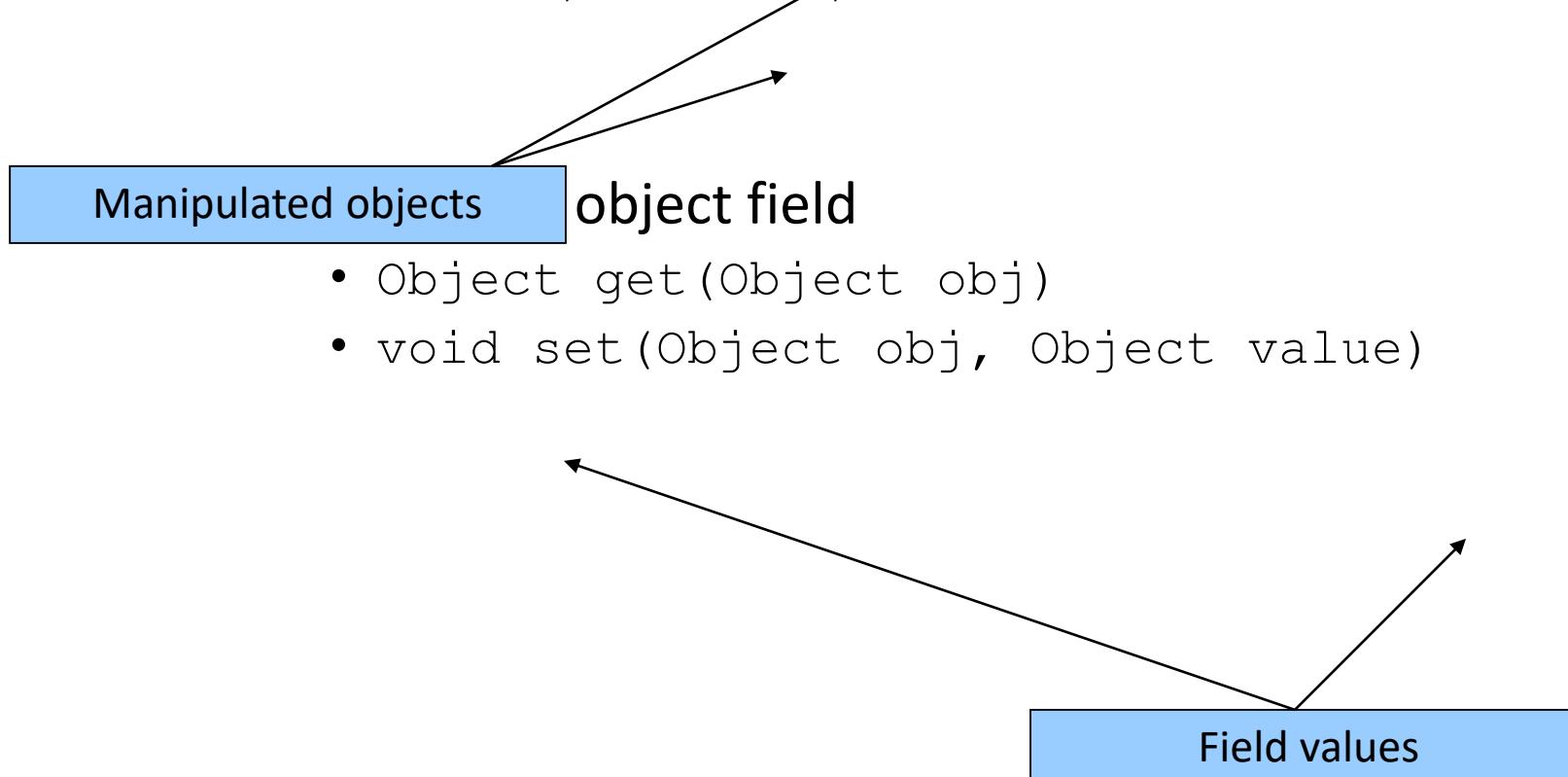
```
// retrieving field information
static void printFieldNames(Object o) {
 Class c = o.getClass();
 Field[] publicFields = c.getFields();
 for (int i = 0; i < publicFields.length; i++) {
 String fieldName = publicFields[i].getName();
 Class typeClass = publicFields[i].getType();
 String fieldType = typeClass.getName();
 System.out.println("Name: " + fieldName + ", Type: "
+ fieldType);
 }
}
```

# Class Fields – Field Manipulation

- get and set the value of a field

- values of primitive types

- `getInt`, `getFloat`, ...
- `setInt`, ~~setFloat~~, ...



# Class Fields – Field Manipulation

```
// getting field value
static void printHeight(Rectangle r) {
 Field heightField;
 Integer heightValue;
 Class c = r.getClass();
 try { heightField = c.getField("height");
 heightValue = (Integer) heightField.get(r);
 System.out.println("Height: " +
 heightValue.toString());
 } catch (NoSuchFieldException e) {
 System.out.println(e);
 } catch (SecurityException e) {
 System.out.println(e);
 } catch (IllegalAccessException e) {
 System.out.println(e);
 }
}
```

# Class Methods

- Class's `getMethods` method
  - returns an array of `Method` objects
  - `Method` class includes methods to
    - retrieve method's name, return type, parameter types, set of modifiers, and set of throwable exceptions
    - `Method.invoke` - call the method itself

# Class Methods

```
// retrieve method's information

static void showMethods(Object o) {
 Class c = o.getClass();
 Method[] theMethods = c.getMethods();
 for (int i = 0; i < theMethods.length; i++) {
 String methodString = theMethods[i].getName();
 System.out.println("Name: " + methodString);
 String returnType =
 theMethods[i].getReturnType().getName();
 System.out.println(" Return Type: " +
 returnType);
 Class[] parameterTypes =
 theMethods[i].getParameterTypes();
 System.out.print(" Parameter Types:");
 for (int k=0; k<parameterTypes.length; k++) {
 String parameterString =
 parameterTypes[k].getName();
 System.out.print(" " + parameterString);
 }
 System.out.println();
 }
}
```

# Class Methods

- **Output:**

Name: equals

Return Type: boolean

Parameter Types: java.lang.Object

Name: getClass

Return Type: java.lang.Class

Parameter Types:

Name: hashCode

Return Type: int

Parameter Types:

... .

# Class Methods – Invoking Methods

```
//invoke a method dynamically
Object invokeMethod(Object o, String methodName,
 Class[] parameterTypes, Object[] arguments){

 Object result;
 Class c = o.getClass();
 try {
 Method method = c.getMethod(methodName,
 parameterTypes);

 Object result = method.invoke(o, arguments);
 } catch (NoSuchMethodException e) {
 System.out.println(e);
 } catch (IllegalAccessException e) {
 System.out.println(e);
 } catch (InvocationTargetException e) {
 System.out.println(e);
 }
 return result;
}
```

# Lets go back to the Task

```
public class ReflectionalAnimalFactory implements AnimalFactory {

 private List<Class<? extends Animal>> animalsMDList = new ArrayList<Class<? extends Animal>>();

 public ReflectionalAnimalFactory() throws ClassNotFoundException {...}

 public List<Animal> createZoo(int number) throws IllegalAccessException, InstantiationException

 private Animal createRandomAnimal() throws IllegalAccessException, InstantiationException {...}
}
```

# AnimalFactory constructor

```
public ReflectionalAnimalFactory() throws ClassNotFoundException {

 File file = new File("../ZooWebApplication/src/main/java/com/idi");
 String[] list = file.list();
 for (String s : list) {
 if (Character.isLowerCase(s.charAt(0))) {
 continue; —→ Not a class (class starts with capital letter)
 }
 File tempFile = new File("../src/zoo/animals/" + s);
 if (tempFile.isDirectory()) {
 continue;
 }
 String className = s.split("\\.")[0];
 Class clazz = Class.forName("com.idi." + className);
 if (com.idi.Animal.class.isAssignableFrom(clazz) &&
 !Modifier.isAbstract(clazz.getModifiers())) {
 animalsMDList.add(clazz);
 }
 }
}
```

# Creating a zoo

```
public List<Animal> createZoo(int number) throws IllegalAccessException
 ArrayList<Animal> animals = new ArrayList<Animal>();
 for (int i = 0; i < number; i++) {
 animals.add(createRandomAnimal());
 }
 return animals;
}
```

```
private Animal createRandomAnimal() throws IllegalAccessException
```



And what code  
is there?

# Creating a random animal

```
private Animal createRandomAnimal() throws IllegalAccessException,
 InstantiationException {

 int randomNumber = (int) (Math.random() * animalsMDList.size());
 Class<? extends Animal> animalMD = animalsMDList.get(randomNumber);
 return animalMD.newInstance();
}
```

# Annotations

# Handling multiple exception types

Annotations are metadata or *data about data*. An annotation indicates that the declared element should be processed in some special way by a compiler, development tool, deployment tool, or during runtime.

Annotations can be applied to several Java Elements like:

- package declarations,
- class,
- constructors,
- methods,
- fields,
- Variables and etc.

# Example to Define & Use an Annotation

```
public @interface MyTransactionAnnotation {
}

@MyTransactionAnnotation
public void persistPerson2DB(Person p){
 //all business logic here
}
```

# Annotation Types

- **Marker**
- **Single-Element**
- **Full-value or multi-value**

# Marker

Marker annotations take no parameters. They are used to mark a Java element to be processed in a particular way.

```
public @interface MyTransactionAnnotation {
}
```

```
@MyTransactionAnnotation
public void persistPerson2DB(Person p){
 //all business logic here
}
```

# Single-Element

Single-element, or single-value type, annotations provide a single piece of data only. This can be represented with a data=value pair or, simply with the value (a shortcut syntax) only, within parenthesis.

```
public @interface SQL {
 String value();
}

@SQL("update PERSONblablabla")
public Person updatePersonsAge(Person p, int age){
}
```

# Full-value or multi-value

Full-value type annotations have multiple data members.

```
public @interface Cached {
 int maxElementsInMemory();
 long secondsToLive();
}
```

```
@Cached(maxElementsInMemory = 30, secondsToLive = 24*60*60)
public Data getSomeData(String whatToFind){
```

# The Target of annotation

- @Target(ElementType.TYPE)—can be applied to any element of a class
- @Target(ElementType.FIELD)—can be applied to a field or property
- @Target(ElementType.METHOD)—can be applied to a method level annotation
- @Target(ElementType.PARAMETER)—can be applied to the parameters of a method
- @Target(ElementType.CONSTRUCTOR)—can be applied to constructors
- @Target(ElementType.LOCAL\_VARIABLE)—can be applied to local variables
- @Target(ElementType.ANNOTATION\_TYPE)—indicates that the declared type itself is an

# Annotation Retention Policy

- Source
  - Annotations are to be discarded by the compiler
- Class
  - Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time
- Runtime
  - Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively

# Annotation Retention Policy

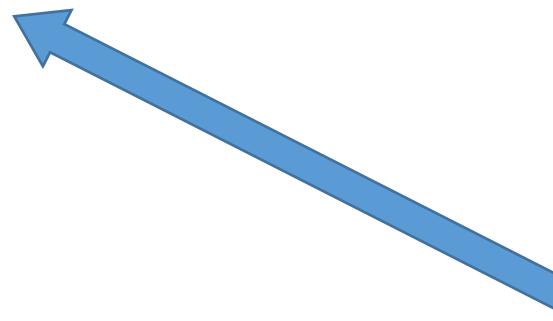
- Runtime

 ***What is the most powerful retention policy?***

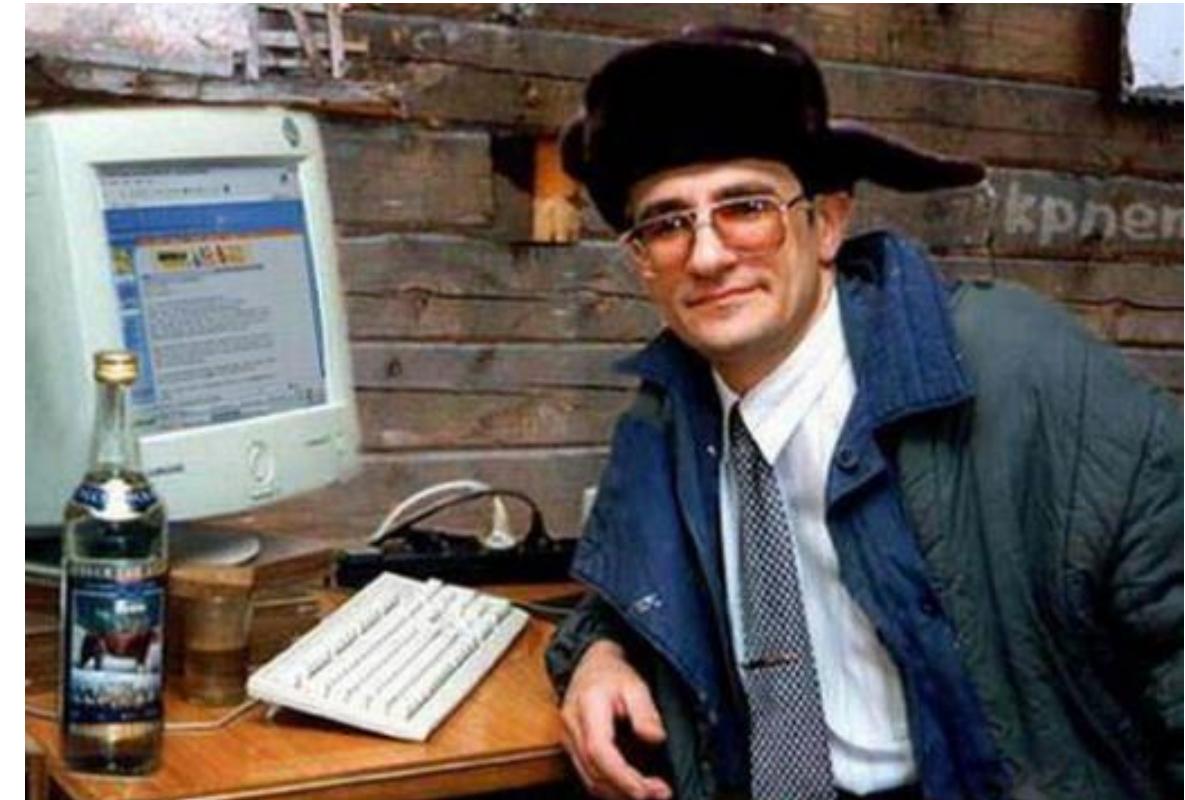
- Class

 ***What is the most useless retention policy?***

- Source

 ***And what is default retention policy?***

Retention: Source... Who needs this type? It is exactly like a comment



# Riddle

```
public class BetterThanBestService extends BestService {

 public void best() {
 System.out.println("Android the best");
 }
}
```

```
public static void main(String[] args) {
 new BetterThanBestService();
}
```



```
public class BestService {
 public BestService(){
 best();
 }

 private void best() {
 out.println("iphone the best");
 }
}
```

1. Will not compile.
2. Runtime exception.
3. iPhone the best.
4. Android the best.

# Riddle

```
public class BetterThanBestService extends BestService {
 @Override
 Method does not override method from its superclass
 System.out.println("Android the best");
}
```

```
public static void main(String[] args) {
 new BetterThanBestService();
}
```

```
public class BestService {
 public BestService(){
 best();
 }

 private void best() {
 out.println("iphone the best");
 }
}
```

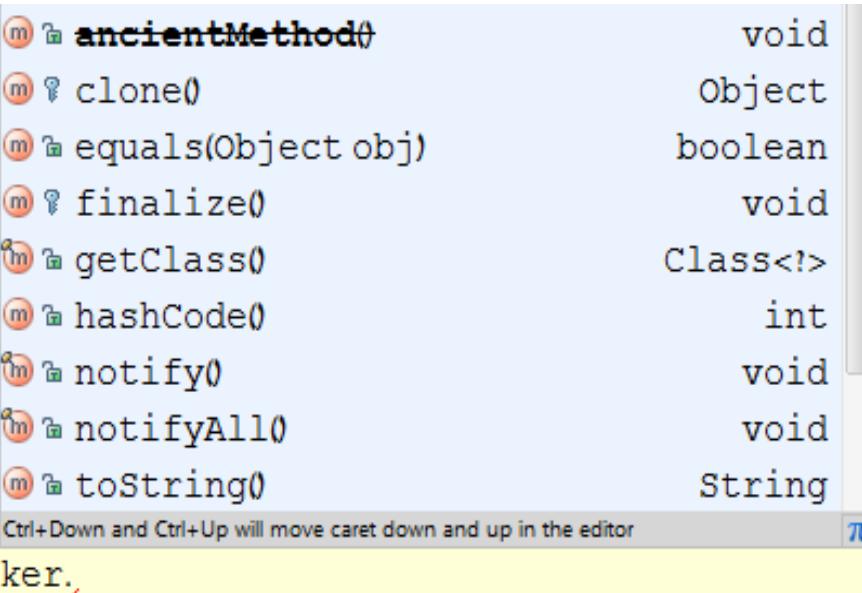
# RetentionPolicy = Source

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {

}

@Deprecated
public void ancientMethod() {
 //any code
}

public static
RussianSpeaker
russianSpeaker.
```



The screenshot shows an IntelliJ IDEA code editor with a completion dropdown open over the line 'russianSpeaker.'. The dropdown lists various methods from the Object class, each with its name, return type, and parameter types. The methods listed are: ancientMethod(), clone(), equals(Object obj), finalize(), getClass(), hashCode(), notify(), notifyAll(), and toString(). The 'ancientMethod()' method is highlighted in the list. A tooltip at the bottom of the dropdown says 'Ctrl+Down and Ctrl+Up will move caret down and up in the editor'.

| Method             | Return Type |
|--------------------|-------------|
| ancientMethod()    | void        |
| clone()            | Object      |
| equals(Object obj) | boolean     |
| finalize()         | void        |
| getClass()         | Class<?>    |
| hashCode()         | int         |
| notify()           | void        |
| notifyAll()        | void        |
| toString()         | String      |

# RetentionPolicy = RUNTIME

- RUNTIME: Annotations are to be recorded in the class file by the compiler and **retained by the VM at run time, so they may be read reflectively.**

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTransactionAnnotation {

}

@MyTransactionAnnotation
public void persistPerson2DB(Person p){
 //all business logic here
}
```

# Reflections.jar



# This is another anti-pattern: God Class



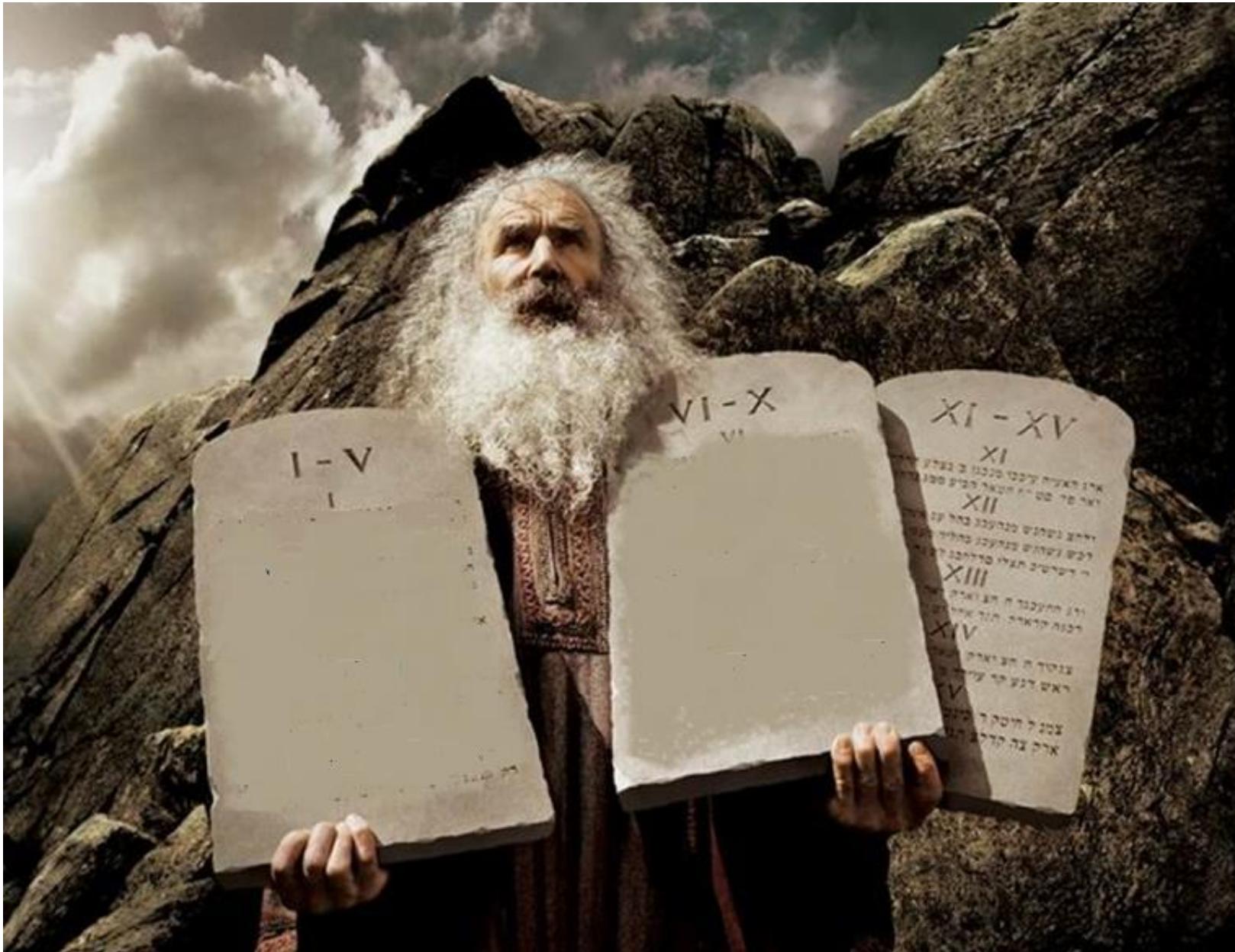
The reasons for it:

1. If / else
2. While
3. Switch / case
4. goto

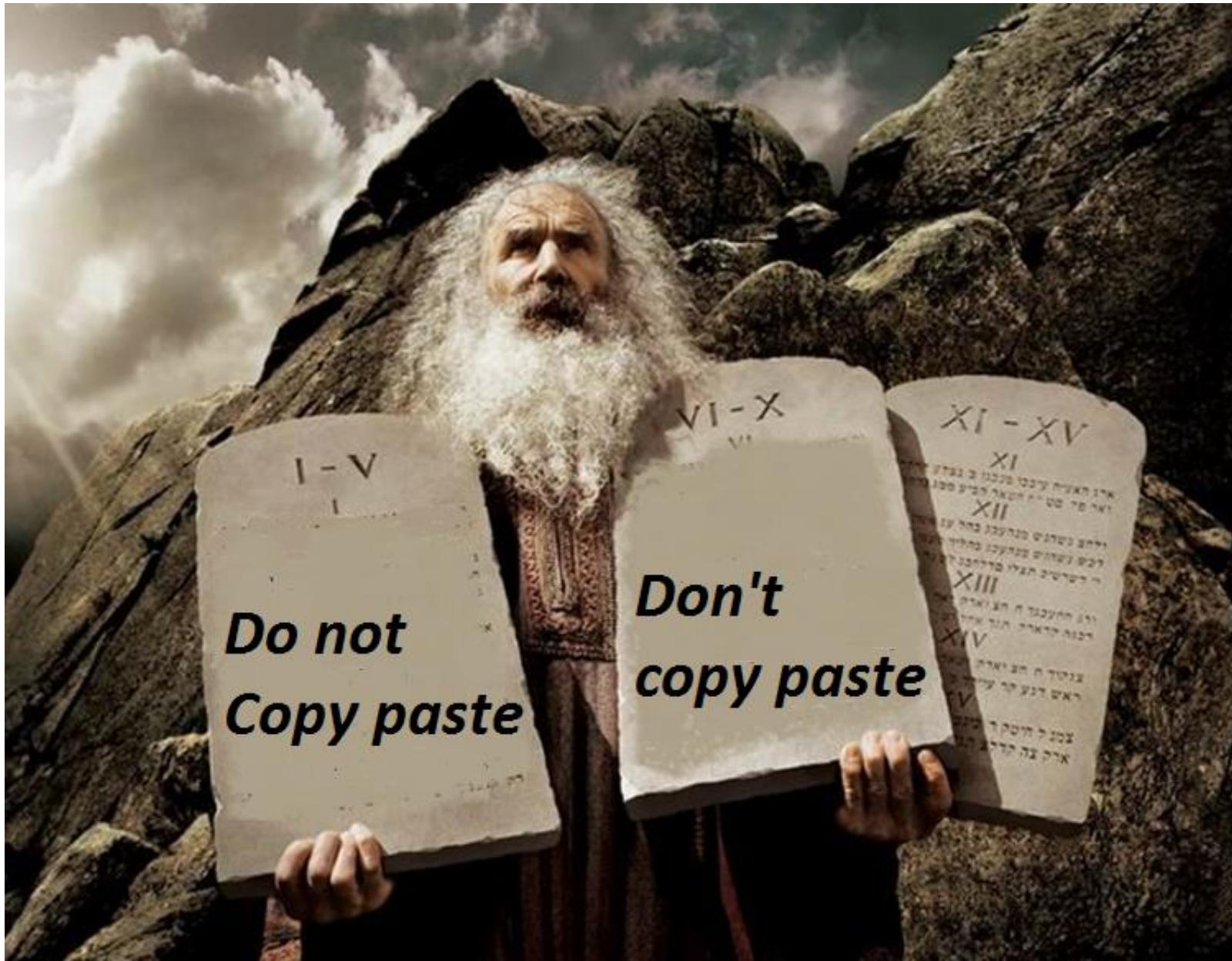
# What is a bad design?

- Rigid – Hard to change a part of the system without affecting too many other parts
- Fragile – When making a change, unexpected parts of the system break
- Immobile – Hard to reuse it in another application because it cannot be disentangled from the current application

# 2 Main Commandments of developer



# 2 Main Commandments of developer



# How can you know that code is ugly



# Comments

**“Comments often are used as a deodorant”**

**Who is Martin Fowler ?**

**Martin Fowler** is an author and international speaker on software development, specializing in object-oriented analysis and design, UML, patterns, and agile software development methodologies, including extreme programming.

Fowler is a member of the *Agile Alliance* and helped create the Manifesto for Agile Software Development in 2001, along with more than 15 co-authors. Martin Fowler was born in Walsall England, and lived in London a decade before moving to United States in 1994.

A portrait photograph of Martin Fowler, a middle-aged man with a beard and mustache, wearing a dark green shirt, smiling at the camera.

# When comments are ok?

- Explain algorithm
- For public API
- Describe variables

# When comments are ok?

- ~~Explain algorithm~~
- For public API
- ~~Describe variables~~

# Removing the comments

- Rename
  - Fields
  - Constants
  - Methods
  - Input parameters
  - Classes
- Extract method

# What's in a name?



"What's in a name?  
That which we  
call a rose  
By any other name  
would smell as sweet."



# Java conventions

- Class starts from Uppercase
- THIS\_IS\_CONSTANT
- Only here you can use \_ long package name(package – lowercased)
- Never use this: \_methodVariable=3
- Only class (type) starts from uppercase.
- Use upperCaseToDevideBewtweenWords

# Java convention examples

- Interface name: PersonService
- Not IPersonService – this is not C#
- Class name: PersonServiceImpl
- Method name: printPersonDetails
- Variable names:  
String name, int age, Person person
- Constant: final int NUMBER\_OF\_LEGS = 2

Why use  
conventions?



# How many mistakes can you find here?





**new balance()**

935 6549  
999 23040

0.16  
0.0744

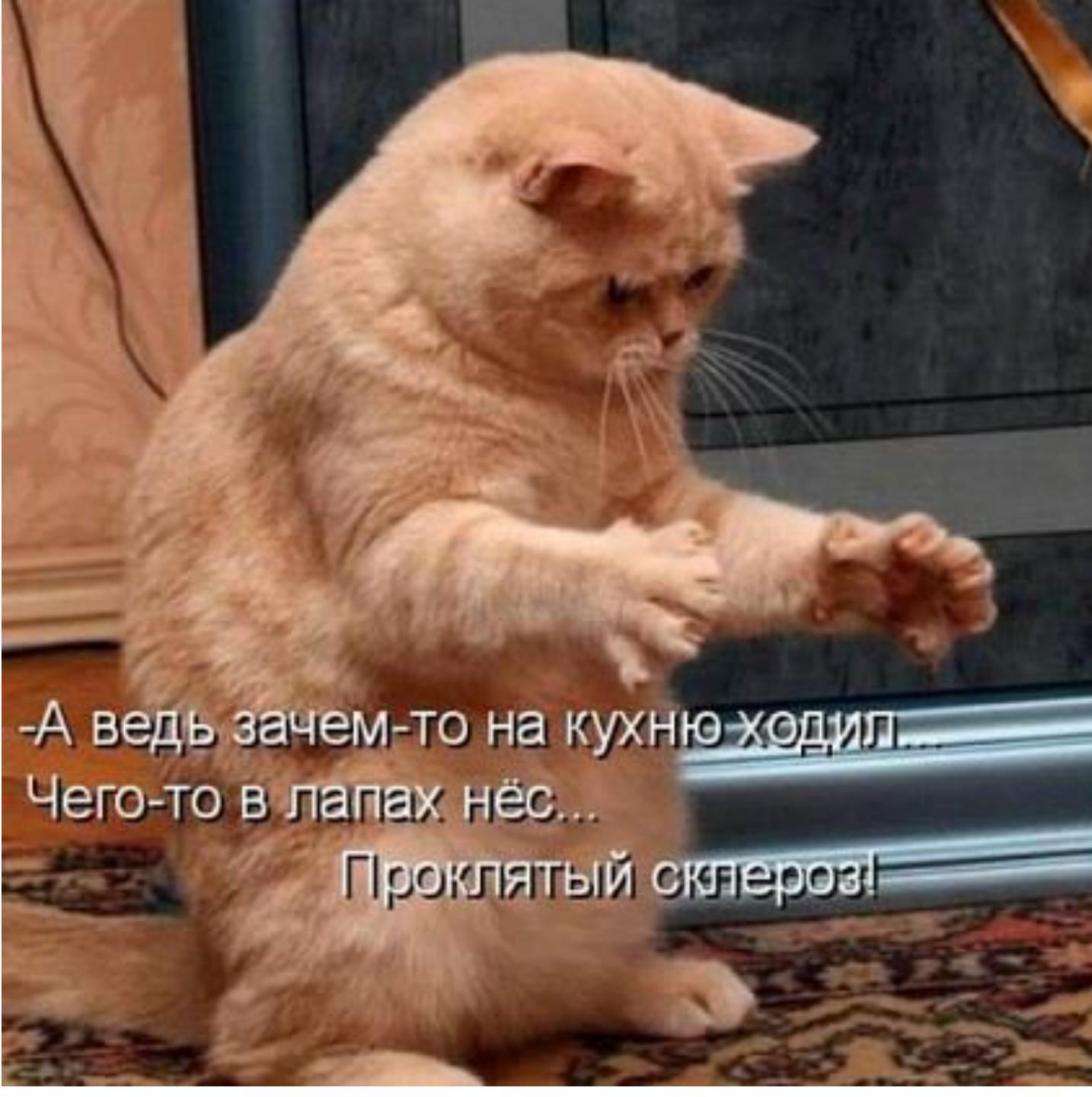
NEW

NEW

NEW

# How to give names

- Make the compiler happy
- Keep java naming convention
- Never use Ifc, use - Impl
- Don't make spell errors
- Make it pronouncable (dlgFnEvalItr)
- Add logical names (Service, Listener, Model...)
- Don't use abuse words (Class, Object...)
- Be consistent (calcPrice, calculateTime, getCalculatedResult)
- Use alt+ctrl+shift+N before creating new method
- Don't economize in method or class name length



-А ведь зачем-то на кухню ходил...  
Чего-то в лапах нёс...  
Проклятый склероз!

# More code smells

- Long methods



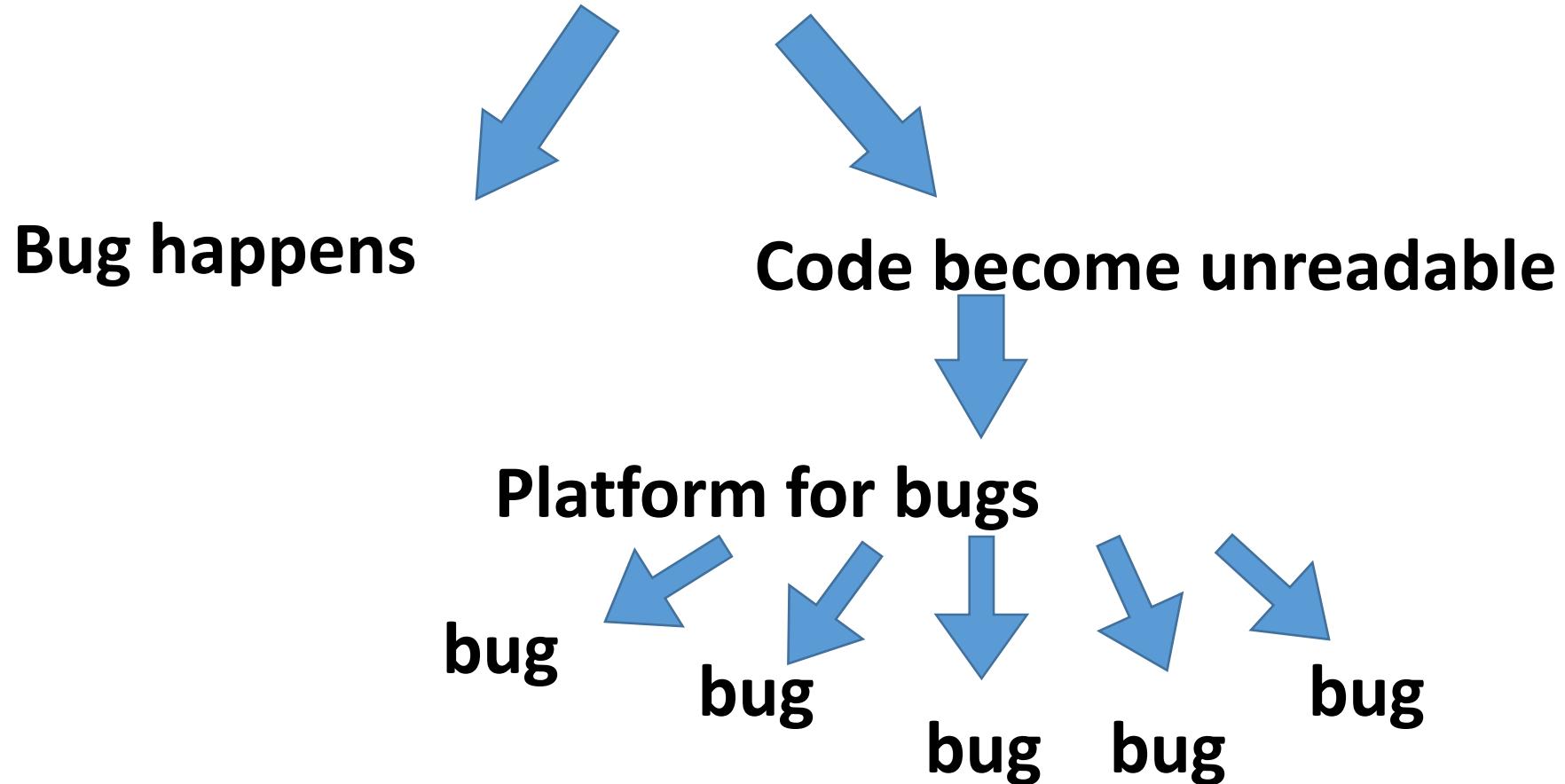
# Why people write long methods

- Unrestrained feature growth
- Blind Conformance
- Ignorance
- Fear of “losing the thread”
- Performance Paranoia

# About performance



**There is a people which think too much about performance...  
They try to write custom optimization for the code**



<http://habrahabr.ru/post/165729/>

14 января в 14:03

## Предельная производительность: C#

из песочницы

 Программирование\*, Параллельное программирование\*, Высокая производительность\*

Я поделюсь 30 практиками для достижения максимальной производительности приложений, которые этого требуют. Затем, я расскажу, как применил их для коммерческого продукта и добился небывалых результатов!

Приложение было написано на C# для платформы Windows, работающее с Microsoft SQL Server. Никаких профайлеров – содержание основывается на понимании работы различных технологий, поэтому многие топики пригодятся для других платформ и языков программирования.



Предисловие

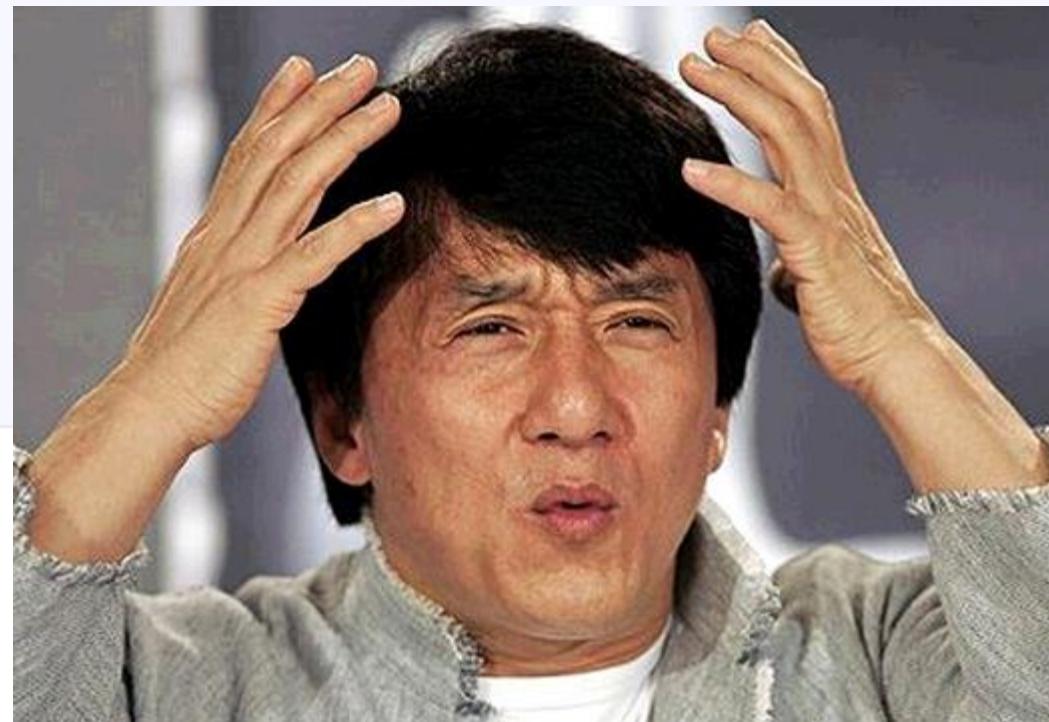
# Don't do that in Java

## 7. Разматывайте циклы.

Любой цикл – это та же конструкция «if». Если количество итераций небольшое, и их количество заранее известно, то иногда цикл лучше заменить на его тело, которое повторяется необходимое кол-во итераций (да, один раз Copy и N раз Paste).

### ▼ Пример – возведение числа в 4-ю степень

```
int power4(int v)
{
 int r = 1;
 r *= v;
 r *= v;
 r *= v;
 r *= v;
 return r;
}
```



# About HotSpot

What does it mean..... HotSpot



# Just-In-Time Compilation

# Just-In-Time Compilation

- Everyone knows about JIT!
- Hot code is compiled to native
- What is “hot”?
  - Server VM – 10000 invocations
  - Client VM – 1500 invocations
  - Use `-XX:CompileThreshold=#` to change
    - More invocations – better optimizations
    - Less invocations – shorter warmup time

# HotSpot optimizations

- JIT Compilation
  - Compiler Optimizations
  - Generates more performant code than you could write in native
- Adaptive Optimization
- Split Time Verification

# Adaptive Optimization

- Allows HotSpot to uncompile previously compiled code
- Much more aggressive, even speculative optimizations may be performed
- And rolled back if something goes wrong or new data gathered
  - E.g. classloading might invalidate inlining

# Two Types of Optimizations

- Java has two compilers:
  - javac bytecode compiler
  - HotSpot VM JIT compiler
- Both implement similar optimizations
- Bytecode compiler is limited
  - Dynamic linking
  - Can apply only static optimizations

# Example – Bounds Check Elimination

```
1 public class GameTest {
2 @Test
3 public void testScore() {
4 Game game = new Game();
5 GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
6 int length = gameMoves.length;
7 for (int i = 0; i < length; i++) {
8 if (i < 0 || length <= i) throw new ArrayIndexOutOfBoundsException();
9 game.score(gameMoves[i].getPlayer(), gameMoves[i].getPoints());
10 }
11 }
12 }
```

# Loop unrolling

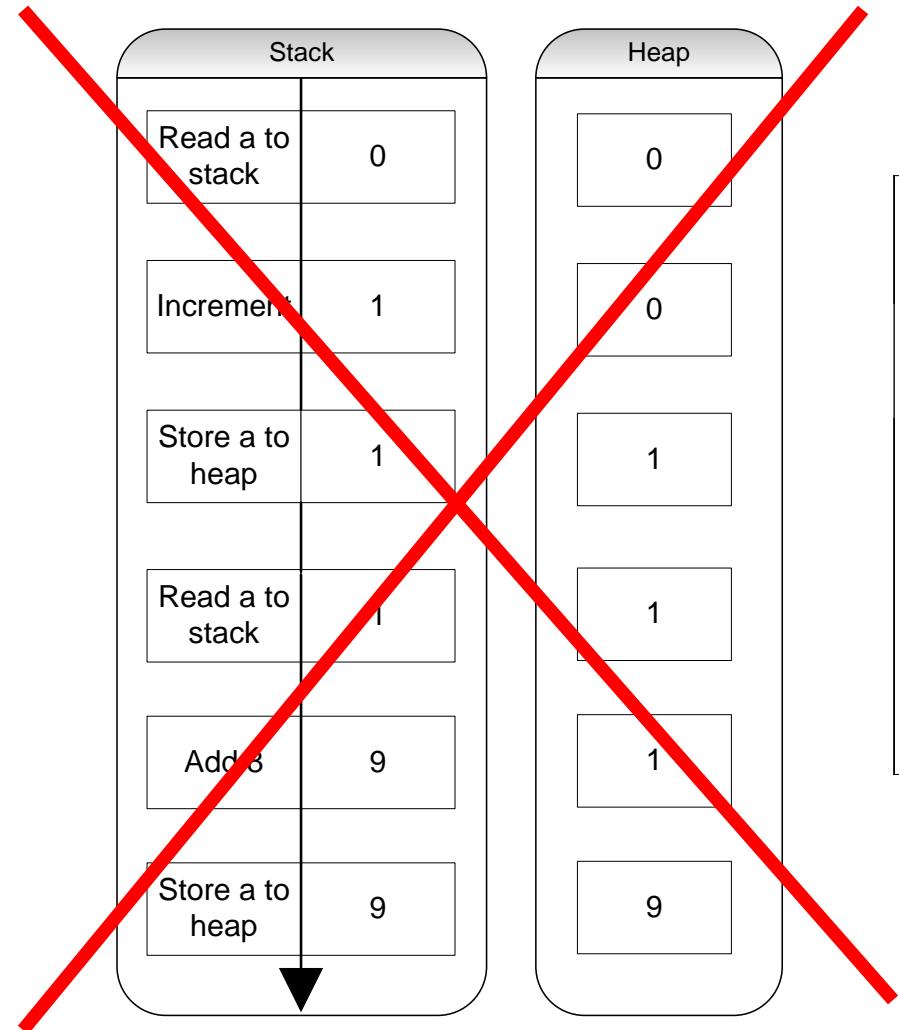
```
1 public class GameTest {
2 @Test
3 public void testScore() {
4 Game game = new Game();
5 GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
6 for (int i = 0; i < gameMoves.length; i++) {
7 GameMove move = gameMoves[i];
8 game.score(move.getPlayer(), move.getPoints());
9 }
10 }
11 }
```

```
1 public class GameTest {
2 @Test
3 public void testScore() {
4 Game game = new Game();
5 GameMove[] gameMoves = {new GameMove("bob", 2), new GameMove("robert", 3)};
6 GameMove move = gameMoves[0];
7 game.score(move.getPlayer(), move.getPoints());
8 move = gameMoves[1];
9 game.score(move.getPlayer(), move.getPoints());
10 }
11 }
```

# OSR - On Stack Replacement

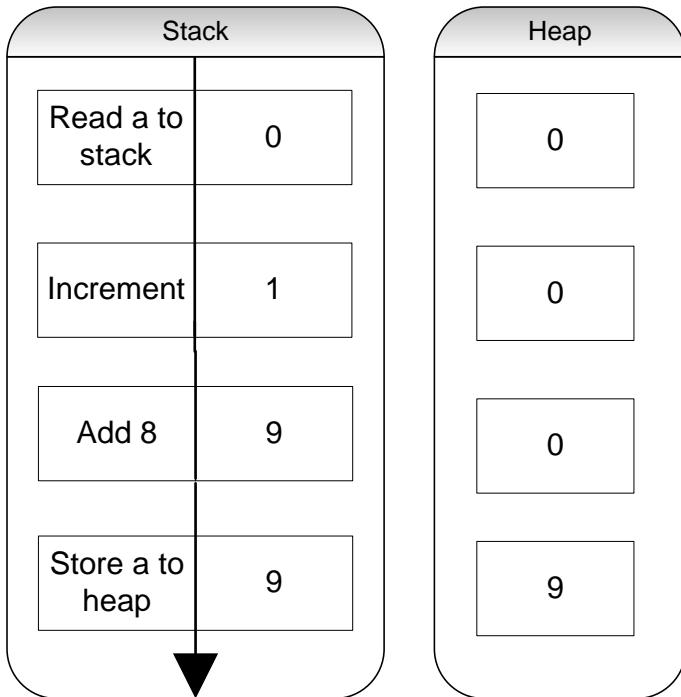
- Normally code is switched from interpretation to native in heap context
  - Before entering method
- OSR - switch from interpretation to compiled code in local context
  - In the middle of a method call
  - JVM tracks code block execution count
- Less optimizations
  - May prevent bound check elimination and loop unrolling

# Out-Of-Order Execution



```
1 public class OutOfOrder {
2
3 private int a;
4
5 public void foo() {
6 a++;
7 a+=8;
8 }
9
10 }
```

# Out-Of-Order Execution



```
1 public class OutOfOrder {
2
3 private int a;
4
5 public void foo() {
6 a++;
7 a+=8;
8 }
9
10 }
```

# Inlining

- Love Encapsulation?
  - Getters and setters
- Love clean and simple code?
  - Small methods
- Use static code analysis?
  - Small methods
- No penalty for using those!
- JIT brings the implementation of these methods into a containing method
  - This optimization known as “Inlining”

# Inlining

- Before inline optimization

```
Point point = new Point(3, 4);
PointService pointService = new PointService();
pointService.printPoint(point);
```

- Two additional object, will be created in the heap

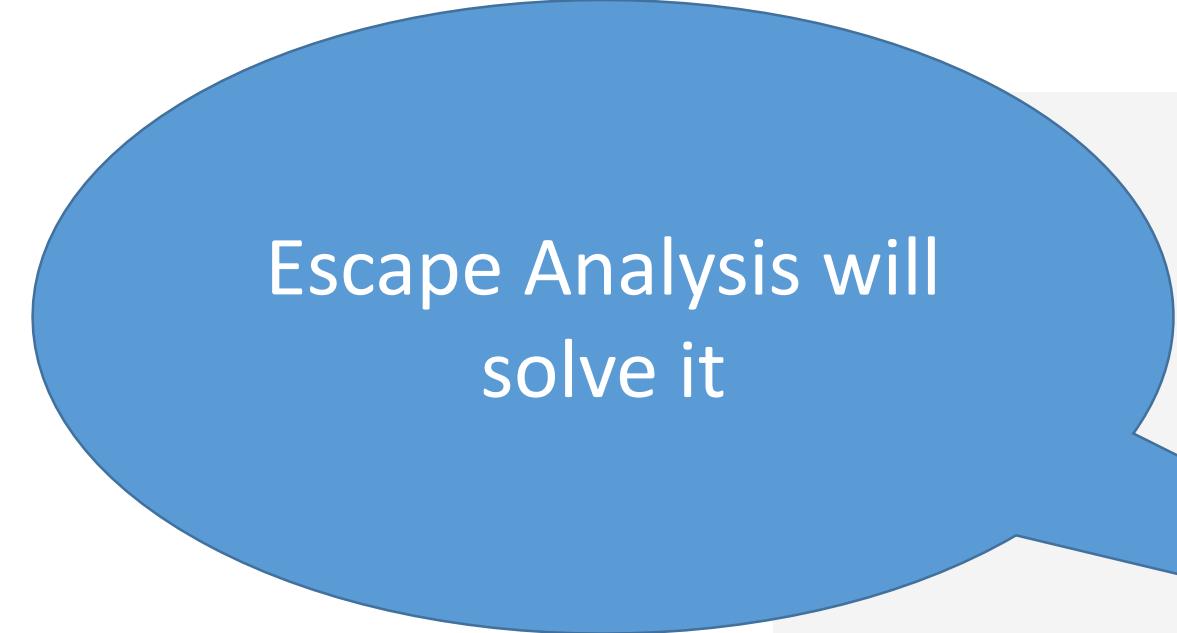
# Inlining

- After optimization

```
Point point = new Point(3, 4);
System.out.println(point.getX() + " " + point.getY());
```

A photograph of a man with short brown hair, wearing a dark suit jacket, a light-colored shirt, and a patterned tie. He is resting his chin on his right hand, looking directly at the camera with a thoughtful expression. A large blue speech bubble originates from his chin and contains the text.

But the object which  
was inlined can be  
used later...



Escape Analysis will  
solve it



# Escape Analysis

- Escape analysis is not optimization
- It is check for object not escaping local scope
  - E.g. created in private method, assigned to local variable and not returned
- Escape analysis opens up possibilities for lots of optimizations

# Example - Lock Elision

```
1 public class GameTest {
2 @Test
3 public void testScore() {
4 Game game = new Game();
5 lock(game);
6 game.logger.info("Bob" + " scores " + 5);
7 game.totalScore += 5;
8 game.logger.info("Jane" + " scores " + 7);
9 game.totalScore += 7;
10 game.logger.info("Dwane" + " scores " + 1);
11 game.totalScore += 1;
12 unlock(game);
13 }
14 }
```

# Scalar Replacement

```
Point point = new Point(3, 4);
System.out.println(point.getX() + " " + point.getY());

int x = 3;
int y = 4;
System.out.println(x + " " + y);
```

# Constants Folding

- Literals folding
  - Before: `int foo = 9*10;`
  - After: `int foo = 90;`
- String folding or StringBuilder-ing
  - Before: `String foo = "hi Joe " + (9*10);`
  - After: `String foo = new StringBuilder().append("hi Joe ").append(9 * 10).toString();`
  - After: `String foo = "hi Joe 90";`

# Constants Folding

```
int x = 3;
int y = 4;
System.out.println(x+" "+y);
```



```
System.out.println("3 4");
```

# Finally

```
public class PointService {
 public void printPoint(Point p){
 System.out.println(p.getX()+" "+p.getY());
 }
}

Point point = new Point(3, 4);
PointService pointService = new PointService();
pointService.printPoint(point);
```



```
System.out.println("3 4");
```

# Optimizations is cool! But what can I do?

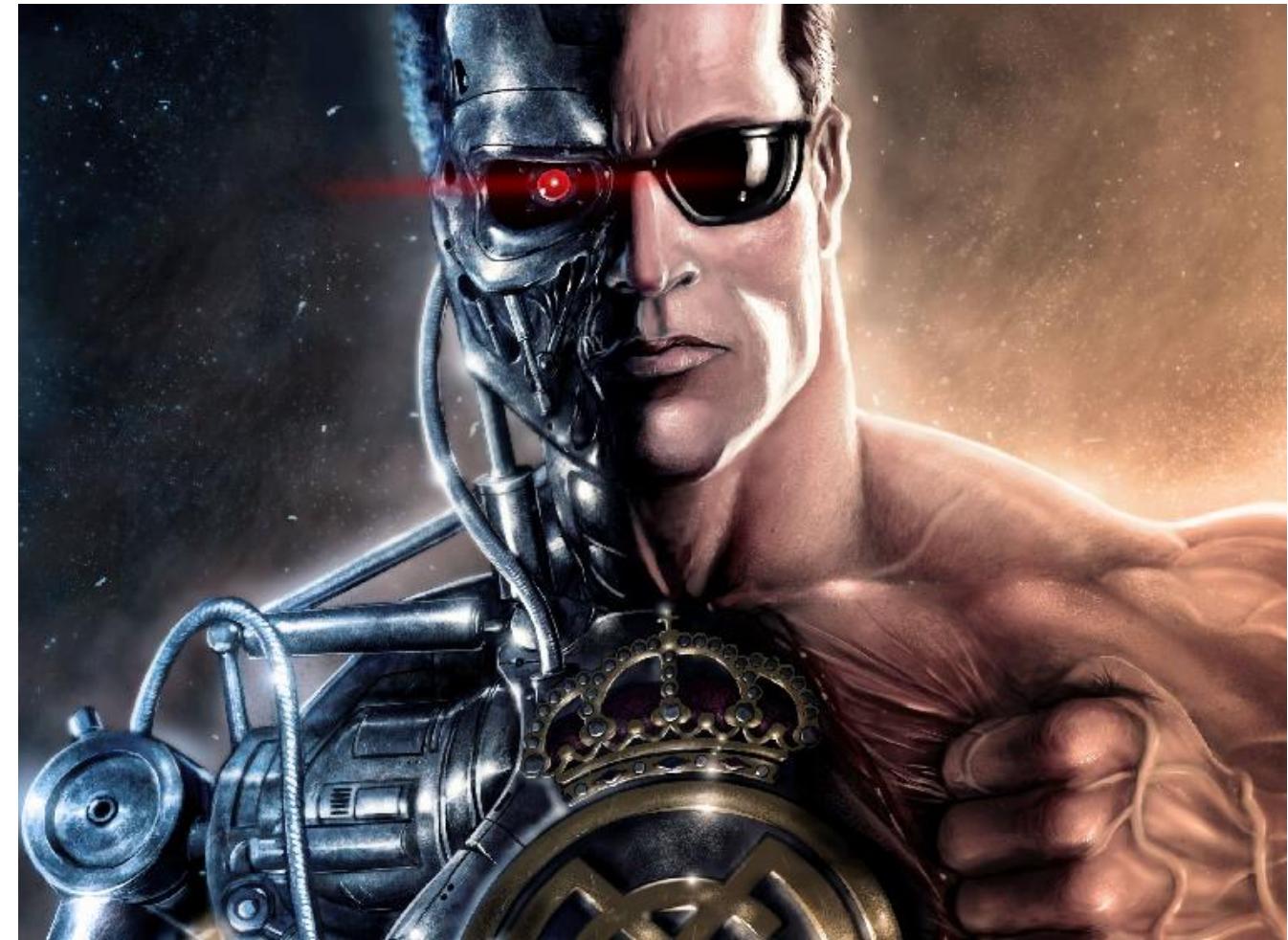


# We need

- Just write good quality Java code
  - Object Orientation
  - Polymorphism
  - Abstraction
  - Encapsulation
  - DRY
  - KISS
  - **Short methods**
- **Let the HotSpot optimize**

# So if long methods are bad, what can be done

- Refactor
- But don't mix abstractions
- Let's see the example



# Method: womanPreparingToGoOut

- Take of the closes
- Enter the bath
- Clean yourself
- Get out from the bath
- Dress yourself
- Open lipstick
- Put lipstick 10 times
- Close lipstick
- Open mascara
- Make up left eye
- Make up right eye
- Look at the mirrow
- Clean up the mascara
- Change mascara color
- ...

# Method: womanPreparingToGoOut

- Take a shower
- Dress yourself
- Make-up
- ...

# SOLID

- Single Responsibility Principle
- Open/Close Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

# Single Responsibility

- A class should have only a single responsibility
- There should never be more than one reason for a class to change

# Not good class

```
Class Person{
```

```
 + getName / setName
```

```
 + getAge / setAge
```

```
 + getEmail
```

```
 + saveToMongo
```

```
 + saveToFile
```

```
}
```

Person Data

Functional methods

# Single Responsibility

Divide  
&  
Conquer

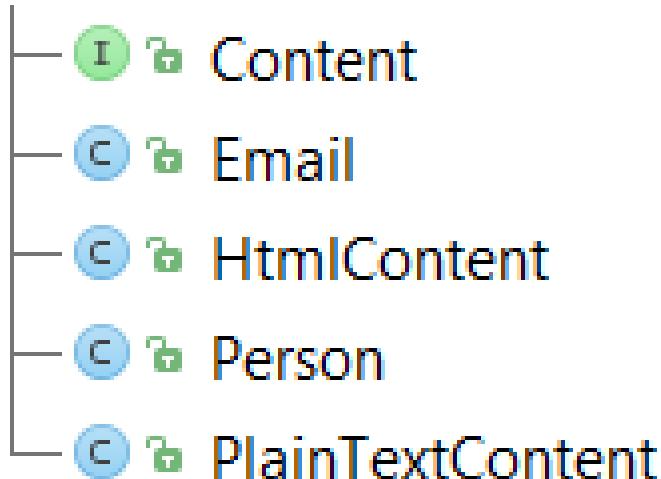
# Task – improve it

```
* / public class Email {
 private String sender;
 private String receiver;
 private String content;

 public String getSender() { return sender; }
 public void setSender(String sender) { this.sender = sender; }
 public String getReceiver() { return receiver; }
 public void setReceiver(String receiver) { this.receiver = receiver; }
 public String getContent() { return content; }
 public void setContent(String content) { this.content = content; }
}
```

# Something like that

```
public class Email {
 private Person sender;
 private Person receiver;
 private Content content;
 public Person getSender() { return sender; }
 public void setSender(Person sender) { this.sender = sender; }
 public Person getReceiver() { return receiver; }
 public void setReceiver(Person receiver) { this.receiver = receiver; }
 public Content getContent() { return content; }
 public void setContent(Content content) { this.content = content; }
}
```



# Open Closed Principle

- New features shouldn't change existing code

# Try to improve it

```
public class Panel{

 private Graphics g;

 public Panel(Graphics g) {
 this.g = g;
 }
 public void drawPoint(Point p){
 // draw point using by using g
 }
 public void drawCircle(Circle circle){
 // draw circle using by using g
 }
}
```

# Something like that!

```
public class Panel{

 private Graphics g;

 public Panel(Graphics g) {
 this.g = g;
 }
 public void drawShape(Shape shape) {
 shape.draw(g);
 }
}
```

# Barbara Liskov

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- Corollary:
- Subtypes need to be **behaviorally** identical from the callers point of view



```
public interface PersistedResource {
 void load();
 void persist();
}

public class ApplicationSettings implements PersistedResource {
 @Override
 public void load() {
 //some logic here
 }

 @Override
 public void persist() {
 //some logic here
 }
}

public class UserSettings implements PersistedResource {
 @Override
 public void load() {
 //some logic here
 }

 @Override
 public void persist() {
 //some logic here
 }
}
```

```
public class SettingsServiceImpl implements SettingsService {
 @Override
 public List<PersistedResource> loadAll() {
 ArrayList<PersistedResource> resources = new ArrayList<PersistedResource>();
 //some logic here
 return resources;
 }

 @Override
 public void saveAll(List<PersistedResource> resources) {
 for (PersistedResource resource : resources) {
 resource.persist();
 }
 }
}

public class Main {
 public static void main(String[] args) {
 SettingsServiceImpl service = new SettingsServiceImpl();
 List<PersistedResource> resources = service.loadAll();
 //some operations with resources
 service.saveAll(resources);
 }
}
```

In the beginning it will be fine



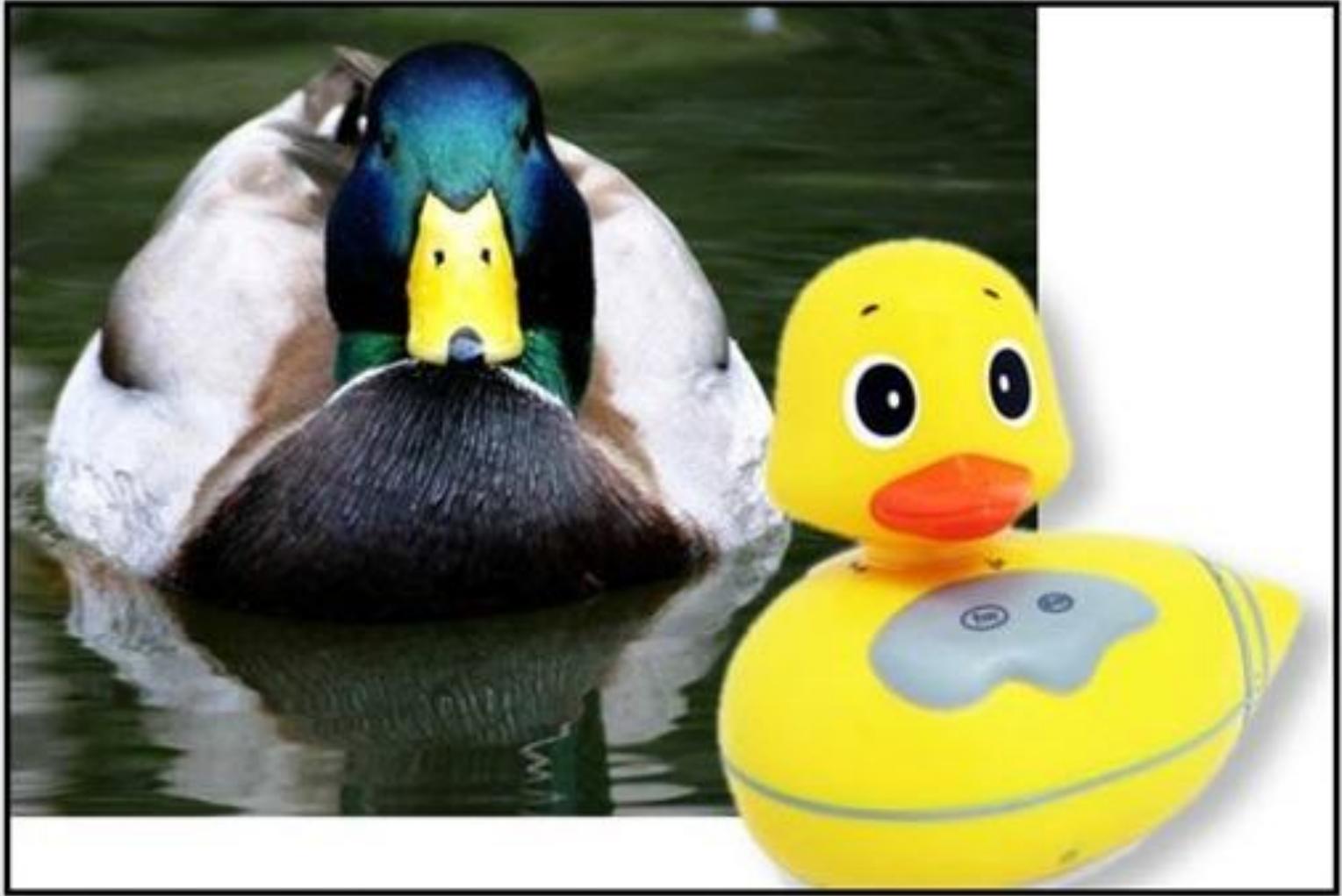
```
public class SpecialResources implements SettingsService {
 @Override
 public List<PersistedResource> loadAll() {
 ArrayList<PersistedResource> resources = new ArrayList<PersistedResource>();
 //some logic here
 return resources;
 }

 @Override
 public void saveAll(List<PersistedResource> resources) {
 throw new UnsupportedOperationException();
 }
}
```

```
public class Main {
 public static void main(String[] args) {
 SettingsServiceImpl service = new SettingsServiceImpl();
 List<PersistedResource> resources = service.loadAll();
 //some operations with resources
 service.saveAll(resources);
 }
}
```



Runtime Exception! Shit!



## LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You  
Probably Have The Wrong Abstraction

# Interface segregation

- Clients should not be forced to depend upon interfaces that they do not use.
- Many client-specific interfaces are better than one general-purpose interface

# Example of breaking this principle

```
public class MyMouseListener implements MouseListener {
 public void mouseClicked(MouseEvent e) {
 //To change body of implemented methods use File
 }

 public void mousePressed(MouseEvent e) {
 //To change body of implemented methods use File
 }

 public void mouseReleased(MouseEvent e) {
 //To change body of implemented methods use File
 }

 public void mouseEntered(MouseEvent e) {
 //To change body of implemented methods use File
 }

 public void mouseExited(MouseEvent e) {
 //To change body of implemented methods use File
 }
}
```

```
public class MyMouseListener implements MouseListener {
 public void mouseClicked(MouseEvent e) {
 //some logic
 }

 public void mousePressed(MouseEvent e) {
 }

 public void mouseReleased(MouseEvent e) {
 }

 public void mouseEntered(MouseEvent e) {
 }

 public void mouseExited(MouseEvent e) {
 }
}
```

```
public interface Resource {
 void Load();
 void Persist();
}
```

```
public interface
LoadableResource {
 void load();
}
```

```
public interface
PersistableResource {
 void persist();
}
```

```
public class ApplicationSettings implements PersistableResource , LoadableResource {
 @Override
 public void load() {
 //some logic here
 }
 @Override
 public void persist() {
 //some logic here
 }
}
```

```
public class SpecialSettings implements LoadableResource {
 @Override
 public void load() {
 //some logic here
 }
}
```

```
public class SettingsServiceImpl implements SettingsService {
 @Override
 public List<LoadableResource> loadAll() {
 ArrayList<LoadableResource> resources = new ArrayList<LoadableResource>();
 //some logic here
 return resources;
 }
 @Override
 public void saveAll(List<PersistableResource> resources) {
 for (PersistableResource resource : resources) {
 resource.persist();
 }
 }
}
```

```
public class Main {
 public static void main(String[] args) {
 SettingsServiceImpl service = new SettingsServiceImpl();
 List<LoadableResource> resources = service.loadAll();
 //some operations with resources
 service.saveAll(resources);
 }
}
```

# Dependency inversion

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Corollary
- Modules should interact through abstraction, and have no knowledge of concrete implementation and life cycle



# And what if db will be changed to MongoDB?

```
public class OracleDao {
 public void savePerson(Person person) {
 //some logic here
 }
}

public class DBService {
 private OracleDao dao;

 public DBService(OracleDao dao) {
 this.dao = dao;
 }

 public void doWork() {
 Person person = readPerson();
 dao.savePerson(person);
 }

 private Person readPerson() {...}
}
```

```
public interface Dao {
 void savePerson(Person person);
}
```



```
public class OracleDao implements Dao {
 @Override
 public void savePerson(Person person) {
 //some logic here
 }
}
```

```
public class DBService {
 private Dao dao;

 public DBService(Dao dao) {
 this.dao = dao;
 }

 public void doWork() {
 Person person = readPerson();
 dao.savePerson(person);
 }

 private Person readPerson() {...}
}
```

# Why do we need Design Patterns?

- Don't reinvent the wheel
- But don't try to use them without a reason

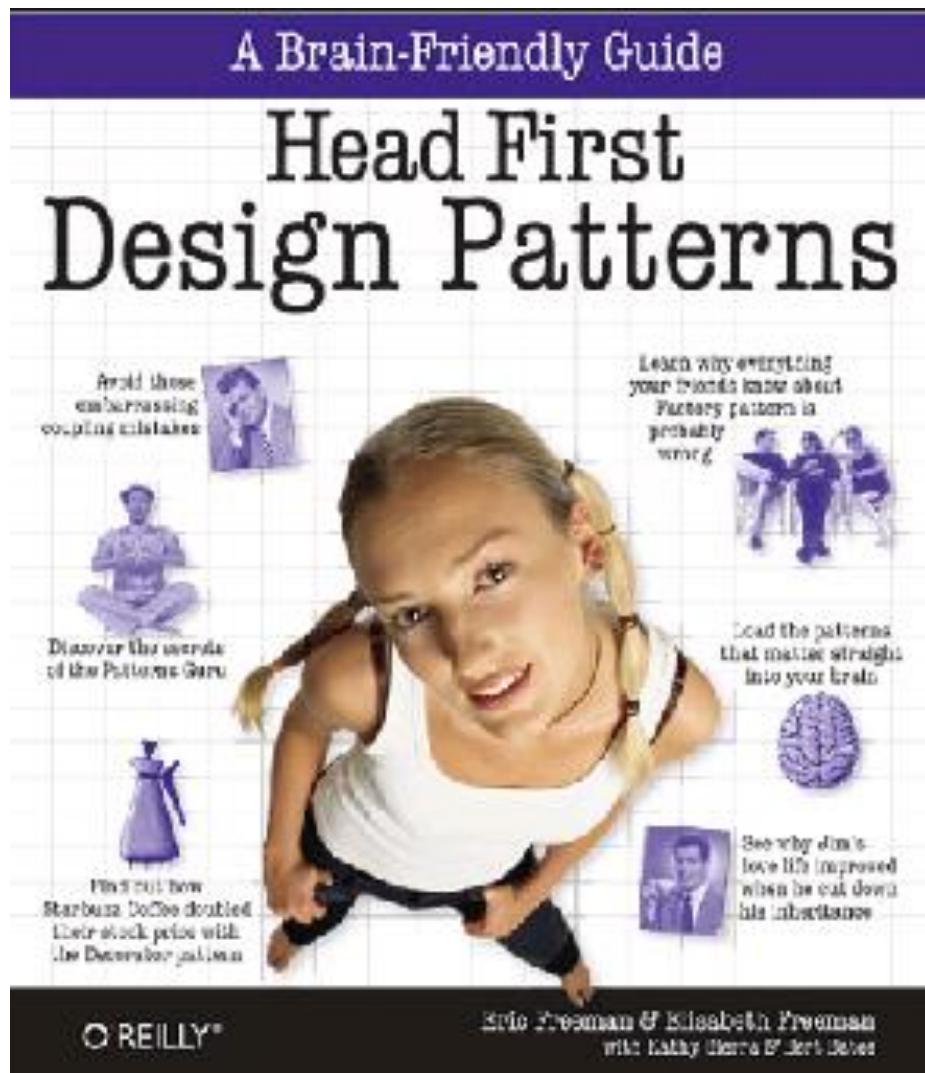


# Design patterns

- 1970 - Christopher Alexander



# Most recommended book

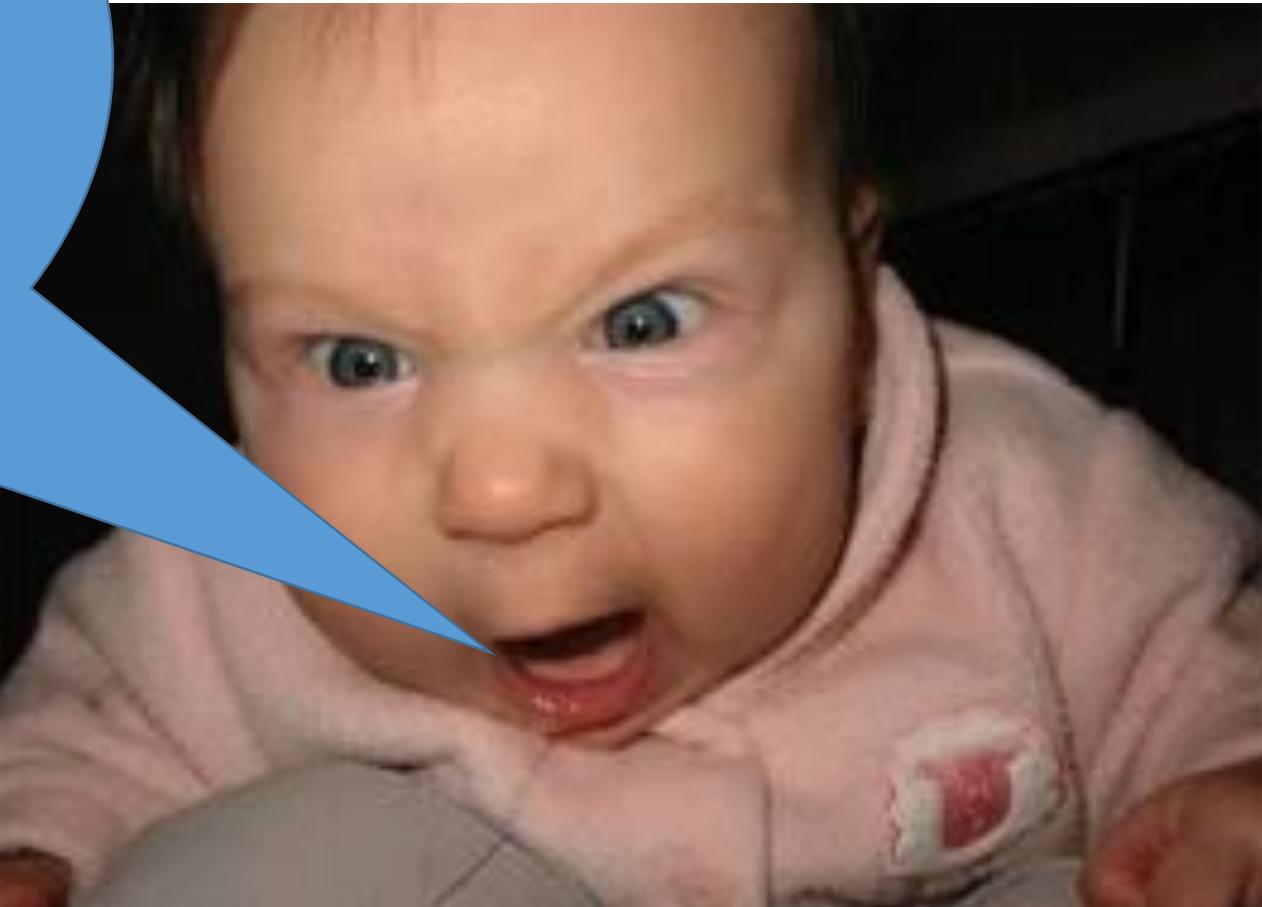


Lets go...

# Task

- Write radio-alarm.
- It has the radio functionality (setChannel, setVolume...)
- And alarm functionality (setAlarmTime, stopAlarm...)

But there is no  
multiple  
inheritance in  
Java!!!



A photograph of a woman with blonde hair, wearing a green tank top, holding a crying baby. The baby has light-colored hair and is wearing a white t-shirt. They are outdoors, with a blurred green background.

I always told that  
inheritance is  
bad...

Use composition  
son...



# Strategy



# Strategy – The Problem

- Sometimes parts of entities change frequently
- In those cases inheritance is limiting by being not dynamic enough
- More so, the changes can even break good inheritance principals
- How can we alter the changing parts of the interface rapidly without touching all the rest?

# Strategy – Problem example

- Action adventure game development in progress:
  - Character interface has `fight()` method
    - Good for Knight, Wizard and Troll implementations
  - King impl is added
    - He fights like the Knight, but can't be its subclass
    - What do you do?
  - Princess impl added
    - She doesn't fight at all
    - What do you do?

# Strategy – The Solution

- Encapsulate what's vary
- Define interface for the changing behavior
- Implement it in different ways
  - Inc. no-op implementation, if needed
- Use the encapsulated algorithms interchangeably

# Strategy – Solution Example

```
1 public interface Character {
2
3 void fight();
4 }
```

```
1 public interface FightBehavior {
2
3 void fight();
4 }
```

```
1 public class Knight implements Character {
2 private FightBehavior swordFightBehavior;
3
4 public Knight() {
5 swordFightBehavior = new SwordFightBehavior();
6 }
7
8 public void fight() {
9 swordFightBehavior.fight();
10 }
11 }
12 }
```

```
1 public class SwordFightBehavior implements FightBehavior {
2
3 public void fight() {
4 ...
5 }
6 }
```

```
1 public class NoFightBehavior implements FightBehavior {
2
3 public void fight() {
4 }
5 }
6 }
```

```
1 public class Princess implements Character {
2
3 FightBehavior noFightBehavior;
4
5 public Princess() {
6 noFightBehavior = new NoFightBehavior();
7 }
8
9 public void fight() {
10 noFightBehavior.fight();
11 }
12 }
13 }
```

# Another way

- Do not declare fight method in Character class
- All fighting characters will implement FightBehavior interface

# No you will write the code...

- Don't forget about SOLID, strategy and composition
- Our customer (Marius Stasus) needs an application for generating mathematic exercises for grade school
- The exercise is summary of two positive random numbers under 100
- Lets add the minus and multiply also
- Another client (Natalia Vasilevna) wants the same application but without negative numbers in minus exercises and result of multiply exercises must be limited till 100

# Creational patterns





THERE CAN BE ONLY ONE

**THERE CAN BE**



**ONLY ONE!**

# There can be only one

- There are many objects we only need one of
  - Thread pools
  - Cache
  - Log objects
  - Objects that gap the software to the hardware
    - Printers
    - Graphic cards

Can you write a singleton?

6 phases of being  
familiar with  
singleton:



# Phase 1: «a Student»



# Phase 2: «Junior software Engineer»

**What about  
multithreading?**

```
public class Singleton {
 private static Singleton singleton;

 private Singleton() {
 }

 public static Singleton getInstance() {
 if (singleton==null) {
 singleton = new Singleton();
 }
 return singleton;
 }

 // all business methods
}
```

# Phase 3: «Middle Software Engineer»

What about  
performance?

```
public class Singleton {
 private static Singleton singleton;

 private Singleton() {}

 public synchronized static Singleton getInstance() {
 if (singleton==null) {
 singleton = new Singleton();
 }
 return singleton;
 }

 // all business methods
}
```

# Phase 4: «Senior Software Engineer»

What about java  
optimization?

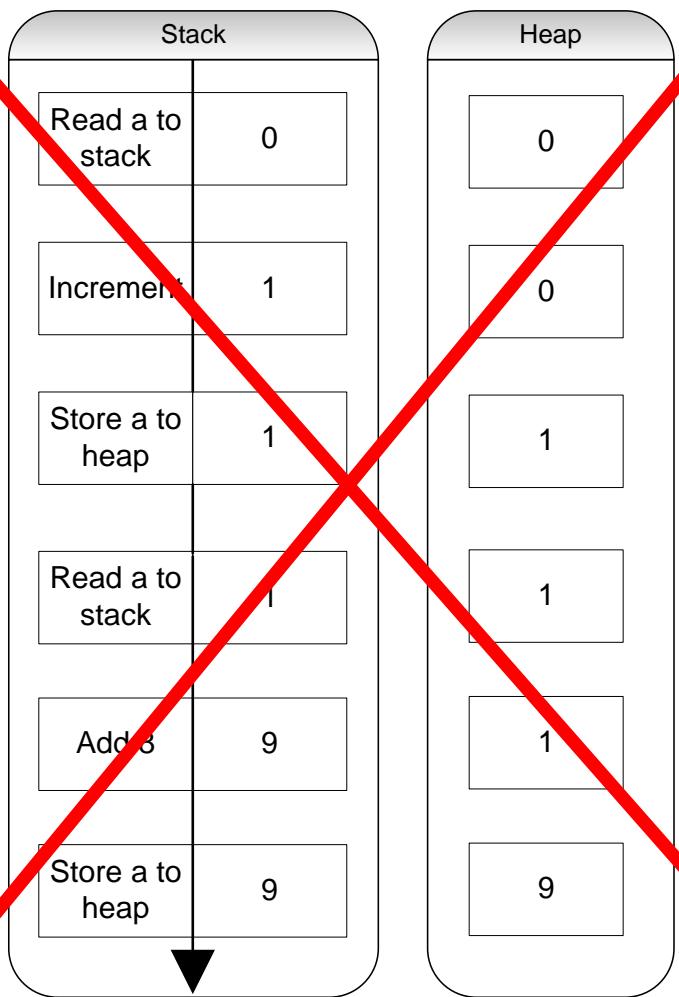
```
public class Singleton {
 private static Singleton singleton;

 private Singleton() {
 }

 public static Singleton getInstance() {
 if (singleton == null) {
 synchronized (Singleton.class) {
 if (singleton == null) {
 singleton = new Singleton();
 }
 }
 }
 return singleton;
 }

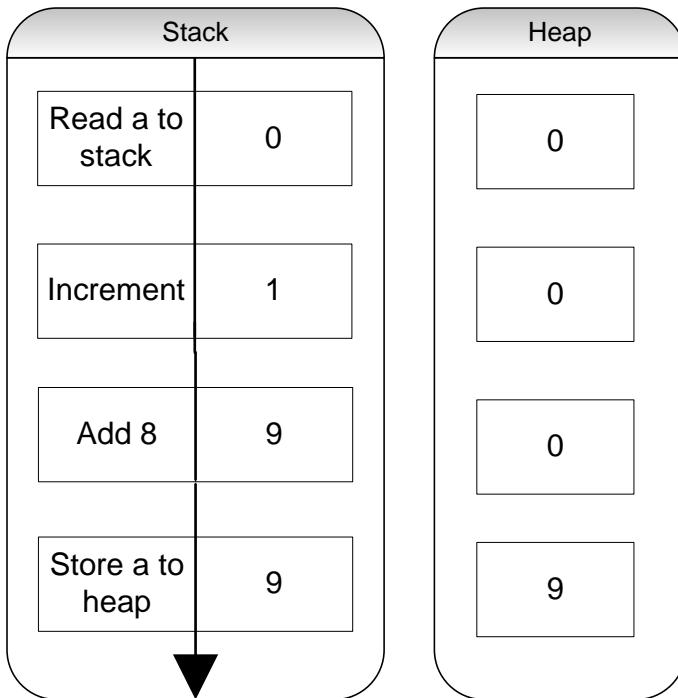
 // all business methods
}
```

# Out-Of-Order Execution



```
1 public class OutOfOrder {
2
3 private int a;
4
5 public void foo() {
6 a++;
7 a+=8;
8 }
9
10 }
```

# Out-Of-Order Execution



```
1 public class OutOfOrder {
2
3 private int a;
4
5 public void foo() {
6 a++;
7 a+=8;
8 }
9
10 }
```

# Phase 4: «Senior Software Engineer»

```
public class Singleton {
 private static Singleton singleton;

 private Singleton() {
 }

 public static Singleton getInstance() {
 if (singleton == null) {
 synchronized (Singleton.class) {
 if (singleton == null) {
 singleton = new Singleton();
 }
 }
 }
 return singleton;
 }

 // all business methods
}
```

# Phase 5: «Lead Software Engineer»

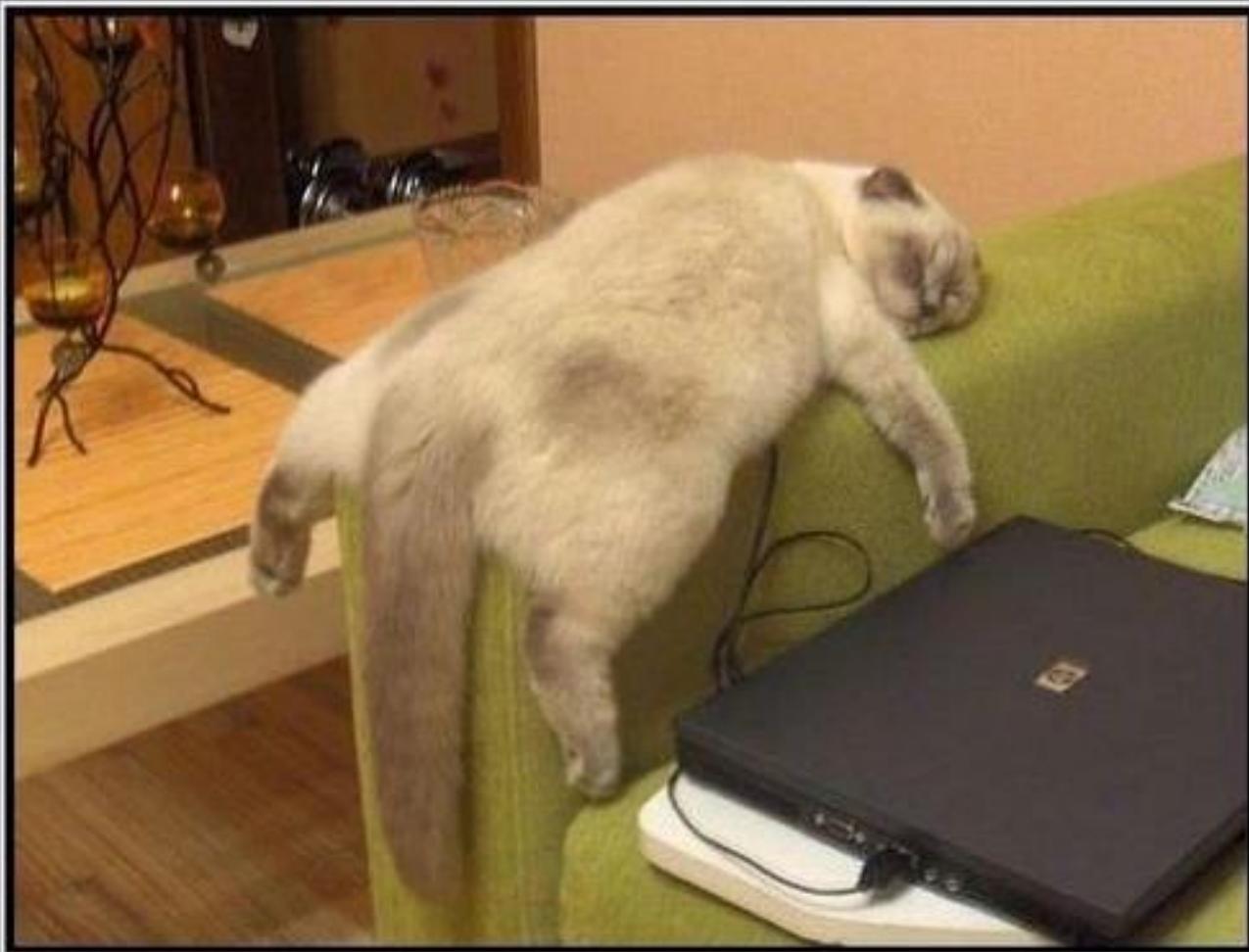
```
public class Singleton {
 private static volatile Singleton singleton;

 private Singleton() {
 }

 public static Singleton getInstance() {
 if (singleton == null) {
 synchronized (Singleton.class) {
 if (singleton == null) {
 singleton = new Singleton();
 }
 }
 }
 return singleton;
 }

 // all business methods
}
```

# Eager Singleton



я не ленивый,  
я энергосберегающий

# Eager Singleton

```
public class Singleton {
 private static Singleton ourInstance = new Singleton();

 public static Singleton getInstance() {
 return ourInstance;
 }

 private Singleton() {
 }
}
```

# Singleton with an enum

```
public enum Singleton {
 INSTANCE;

 public void doWork() {
 System.out.println("singleton is working...");
 }
}

public static void main(String[] args) {
 Singleton.INSTANCE.doWork();
}
```

# Why you shouldn't write a singleton

- You can not trust the future
- Concentrate on your business logic instead of reinventing the wheel
- How will you test?

# Phase 6

## «Architect»

- Don't write  
Singletons.  
You have spring!



# Task

- Create IRobot which will have the only method: cleanRoom
- Robot should say that he started cleaning, run some cleaning logic and then to say that jobs done.

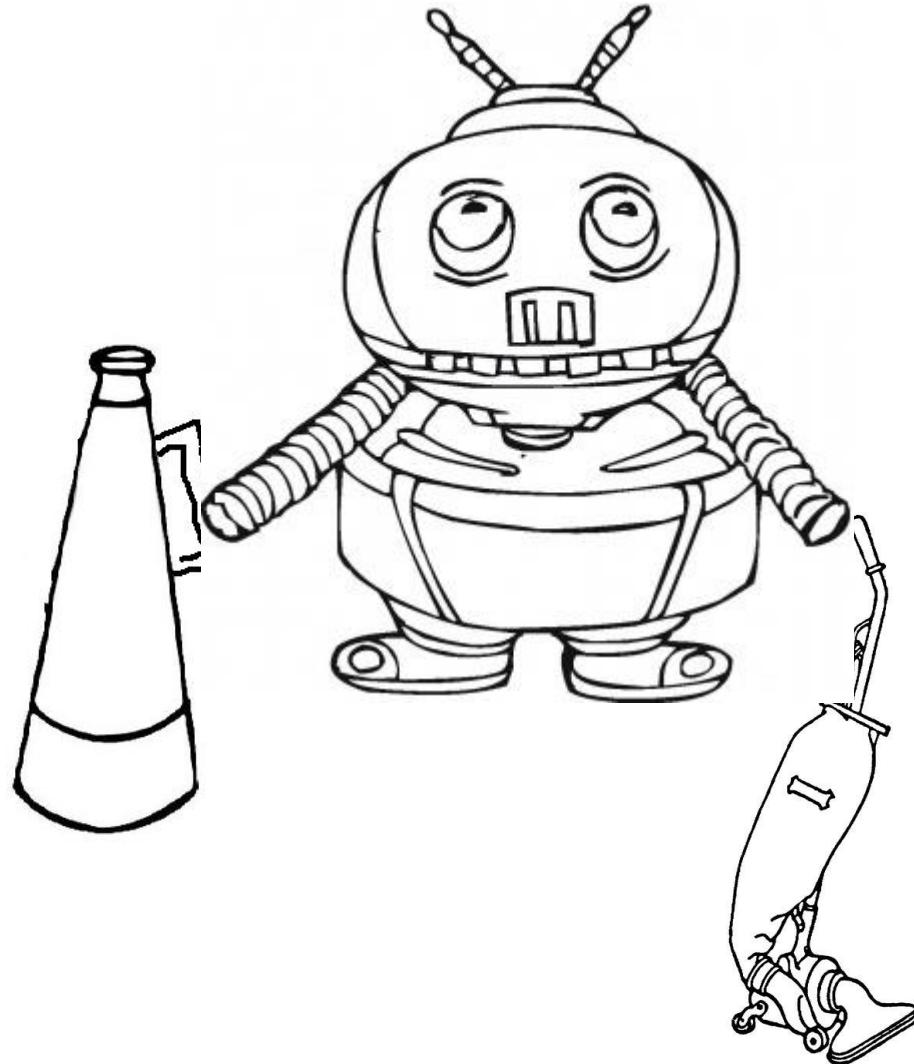


# How can you create an object?



1. Use new keyword
2. I'm a superman, I use only reflection
3. Use a service which creates object
4. Why objects? Static methods are much better!

# What wrong with new?



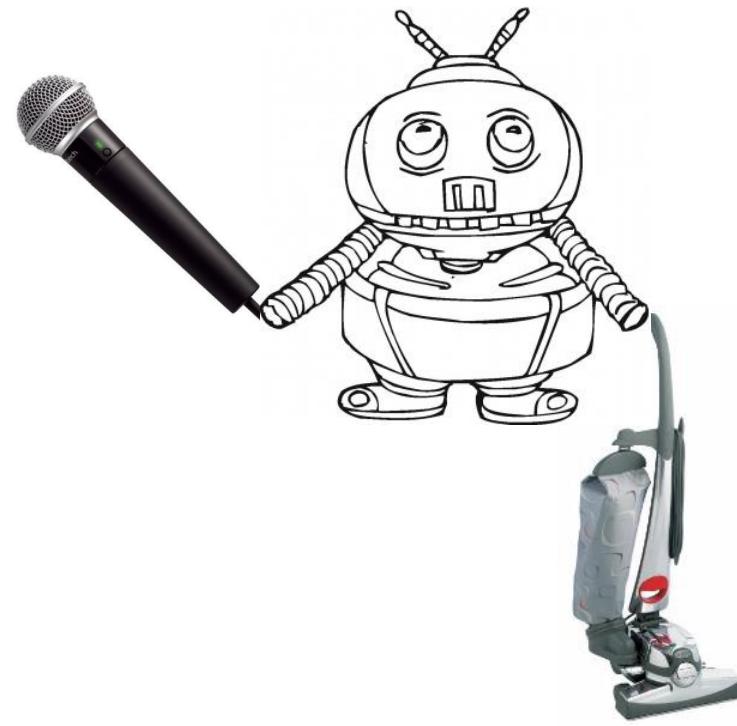
```
public class IRobot {
 private Cleaner cleaner = new Cleaner();
 private Speaker speaker = new Speaker();

 public void cleanRoom() {
 speaker.say("Я начал работать");
 cleaner.clean();
 speaker.say("Я закончил работать");
 }
}
```

# Any problems?

```
private Cleaner cleaner = new Cleaner();
private Speaker speaker = new Speaker();
```

- And if we need to change implementation, we should open the code??



# Interface is much better

```
public interface Speaker {
 void sayJobStarted();

 void sayJobFinished();
}
```

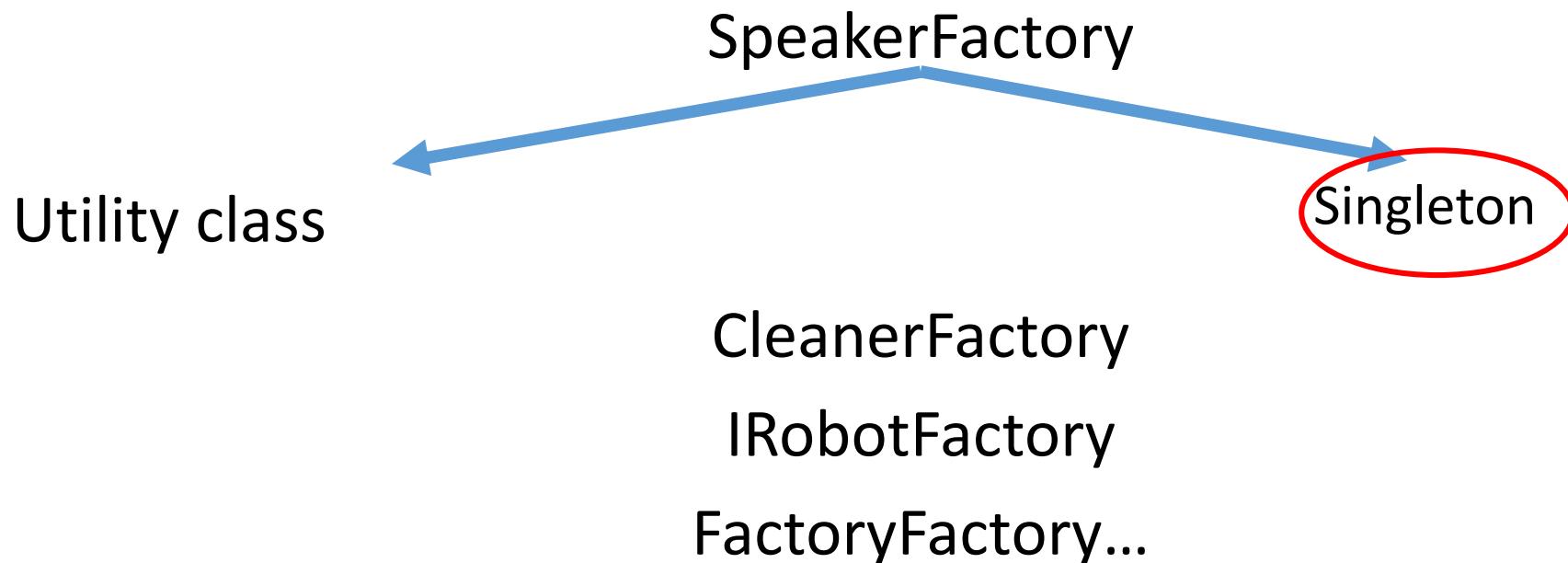
```
public class SimpleSpeaker implements Speaker {
 public void sayJobStarted() {
 System.out.println("Job started");
 }

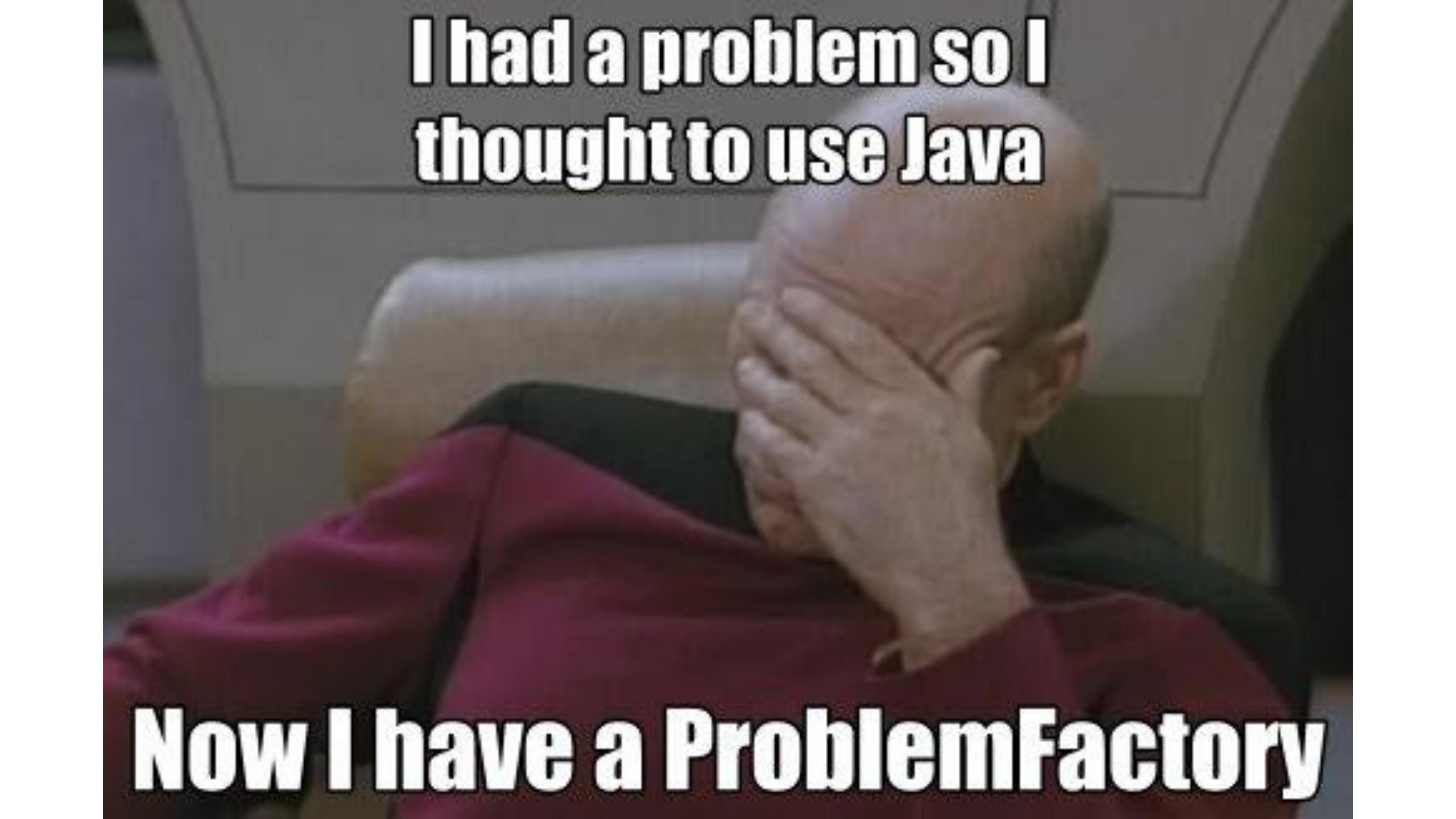
 public void sayJobFinished() {
 System.out.println("Job finished");
 }
}
```

```
public class AdvancedSpeaker implements Speaker {
 public void sayJobStarted() {
 JOptionPane.showMessageDialog(null, "Job started");
 }

 public void sayJobFinished() {
 JOptionPane.showMessageDialog(null, "Job finished");
 }
}
```

And who will decide which impl must be used?



A photograph of a man with light brown hair, wearing a maroon hoodie over a green t-shirt. He is sitting at a light-colored wooden desk, looking down with his hands clasped near his face in a gesture of distress or despair. The background is a plain, light-colored wall.

I had a problem so I  
thought to use Java

Now I have a ProblemFactory

# Why not to do even better? ObjectFactory

- `ObjectFactory.createObject(Speaker.class)`
- Speaker can be both class and interface. If it is an interface factory will choose an implementation according to our configuration
- `createObject` will be used instead “new”
- Why to do that?
  - One central place for each object creation, and if we need to change impl, it will be done only in one place
  - You can make some magic with your objects in the creation phase
  - Like scan annotations and do something with fields or methods according the contract

# Lets improve our ObjectFactory

- Write `@InjectRandomInt` annotation
- With min and max integer parameters
- This annotation can be used upon fields, and in case this field is integer the random number (between num and max) should be injected to that field

# Another annotation

- `@Inject` – will be used upon the fields (the type of field must be some interface)
- `ObjectFactory` will find appropriate implementation and inject it to the field
- Use `@Inject` above field speaker of your robot and check that it is injected
- Try to use injected state in constructor (for example in constructor of your robot delegate to speaker)

# Chain of Responsibility attributes

- In [object-oriented design](#), the **chain-of-responsibility pattern** is a [design pattern](#) consisting of a source of [command objects](#) and a series of **processing objects**.
- Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

# WTF – constructor is not working???

- The state was not configured by our factory in constructor phase
- So we will not use constructors in our new world
- We will use @Postconstruct



# I have a riddle

```
public class Parent {
 public Parent() {
 printPi();
 }
}
```

---

```
public void printPi(){
 System.out.println("Pi");
}
}
```

```
public static void main(String[] args) {
 Son son = new Son();
}
```



```
public class Son extends Parent {
 private double pi = Math.PI;
```

---

```
public Son() {
 printPi();
}
```

```
@Override
public void printPi() {
 System.out.println(pi);
}
```

**Answer:**

0.0  
3.141592653589793

# Can you order this cycles

- @PostConstruct
- @Inject
- Son Initializer
- Parent Initializer
- Son inline
- Parent Inline
- Son Constructor
- Parent Constructor

# The correct order

- Parent Inline
- Parent Initializer
- Parent constructor
- Son Inline
- Son Initializer
- Son constructor
- @Inject
- @PostConstruct

# Replacing the constructor

- Write `@PostConstruct` annotation
- Each method where this annotation present will be invoked by `ObjectFactory` after this object will be configured
- Now all init logic moves from constructor to such methods

Are you tired from reflections?



I'LL BE BACK

# The Problem

- Sometimes parts of the applications need to know when something happened to other parts of it
- How can we notify them without constantly polling for state?
- In other words – how can we fire and receive events in the application?

# Problem Example

- Νίκος Αναστασιάδης publishes time to time an entry in his blog about crisis in Cyprus
- How can I be sure I won't miss his next blog entry and I'll read it as soon as it gets published?
- In other words – how can I **subscribe** to his blog?



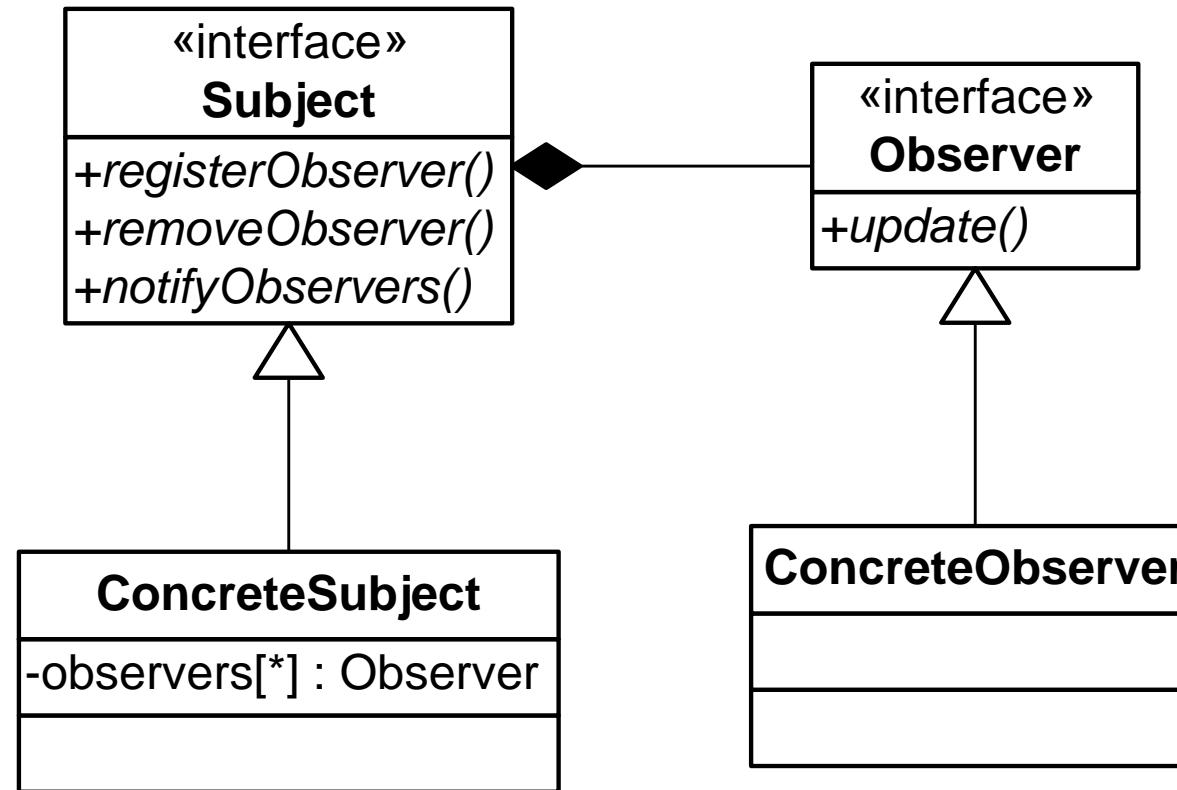
# Task

- Write BlogEngine class, which will have publishPost method.
- Every subscribers should know about every new post

# Observer



# Observer – The Solution



# Solution example

```
public class BlogEngine {
 private Set<Subscriber> subscribers = new HashSet<Subscriber>();
 @Inject
 private BroadCaster broadCaster;

 public void addSubscriber(Subscriber subscriber) {
 subscribers.add(subscriber);
 }

 public void removeSubscriber(Subscriber subscriber) {
 subscribers.remove(subscriber);
 System.out.println("you were succesfully removed");
 }

 public void savePost(BlogPost blogPost) {
 broadCaster.broadcastPost(blogPost);
 for (Subscriber subscriber : subscribers) {
 subscriber.notify(new BlogPostEvent(blogPost));
 }
 }
}
```

# Do you know what is Immutable object?

- You can not change the state of Immutable object after it was created
- And mutators will return a new instance.
- This has a lot of advantages
- Good for optimizations.
  - for example hashCode collection can now that hashCode will never change
- And most important it is good for multithreading



Immutable make a lot of garbage and spend many resources in order to create many new objects!!!

I love Objects, which die young (minor GC better than major GC)

The impact of object creation is overestimated



# Lets talk about different garbage collectors



# Garbage Collector

Who is immortal?

```
public class Immortal {
 //I don't care about encapsulation, I'm immortal!!
 public Immortal immortal;

 public Immortal() {
 }
}

public class ImmortalTest {
 public static void main(String[] args) {
 Immortal immortal = new Immortal();
 immortal.immortal = immortal;
 }
}
```



# Garbage Collector

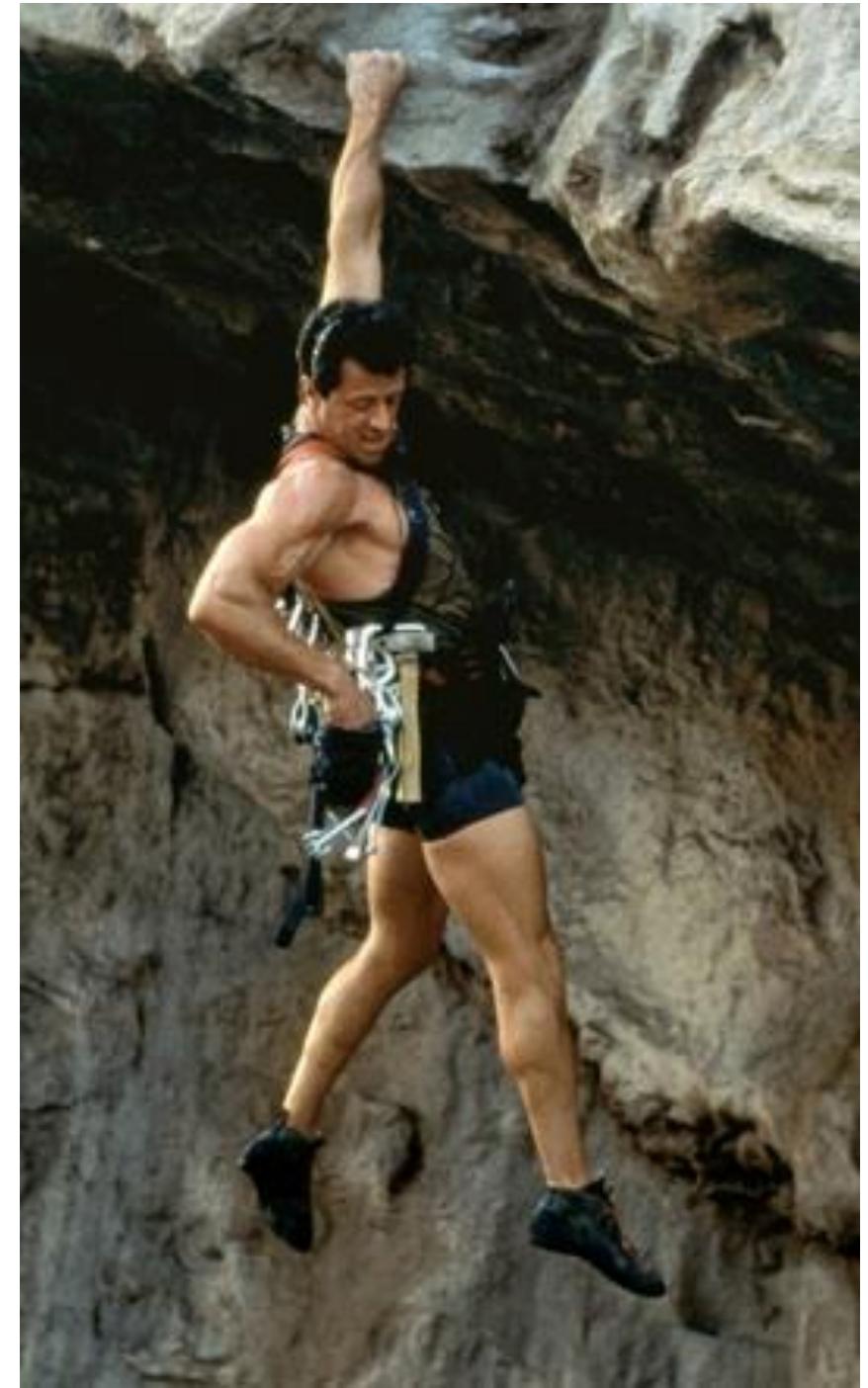
We are immortal till  
we need each other

```
Immortal a = new Immortal();
Immortal b = new Immortal();
a.immortal = b;
b.immortal = a;
```

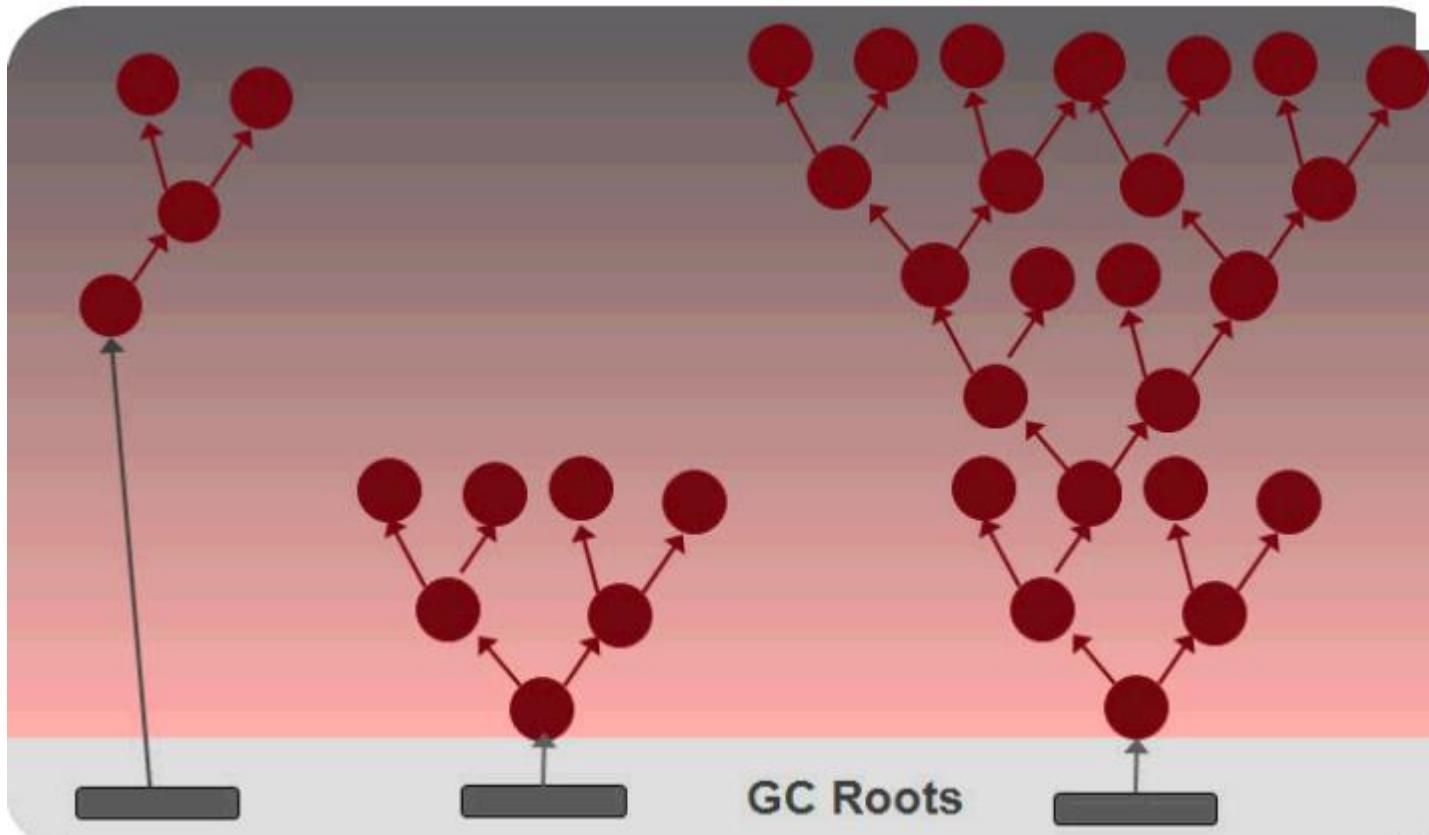


# Strong Reference

- The **regular** java reference everybody use.  
The object on which strong reference points never can be taken by GC in case that there is a way to the root.

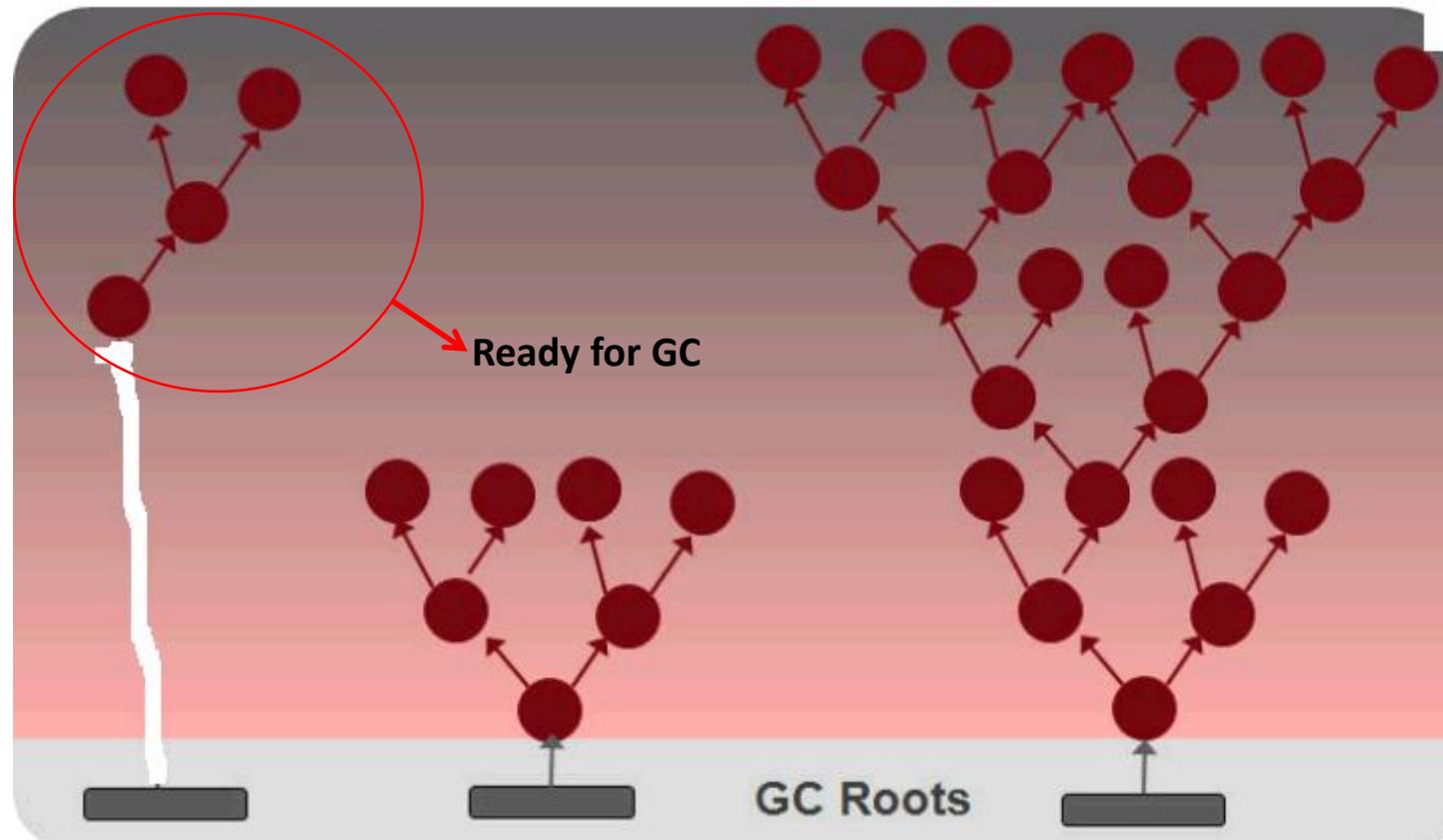


# The heap and GC



The garbage collector starts from “root” references, and walks through the object graph marking all objects that it reaches.

# The heap and GC



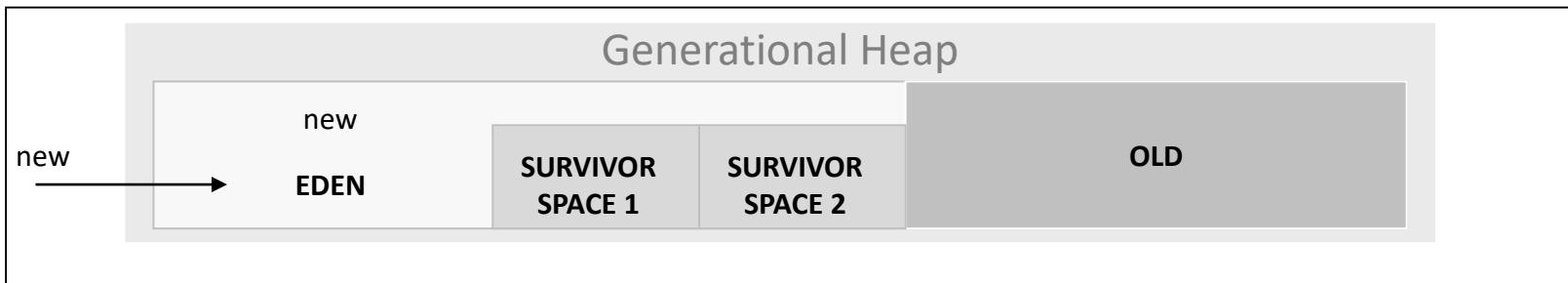
# Hotspot

- Heap:

- Monolithic heap
  - » single heap
  - » easy to tune
  - » not much to manage & therefore consume less memory

- Generational heap

- » separates the heap area into New objects region & Old object region
- » survivor spaces are sub-regions of New
- » each region is tuned and managed separately



# Generational GC

- EDEN & 2 survivor spaces
  - intensive allocations
  - intensive GC activity
  - Inefficient memory usage
- When EDEN is about to overflow – copying collection takes place:
  - Most of its objects are unreferenced
  - Those who are referenced are copied to the survivor space 1 inactive region
  - Next time – objects are copied from both EDEN & SP1 to the SP2
  - SP1 is now clean again – so the phases can repeat until –
  - Long lived object eventually moved to OLD region

# Important question

# Does anybody have allergy on penguins?



# Immutable task

- Write class IDIClient

- State:

name : String

bonusPoints : int

debt : int



# Riddle

- Is it possible do declare instance variable as final without initializing it inline?

```
public final class IDIClient {
 private final String name;
 private final int bonusPoints;
 private final int debt;
```

- Yes, only if you have matching constructor

```
public IDIClient(String name, int debt, int bonusPoints) {
 this.name = name;
 this.debt = debt;
 this.bonusPoints = bonusPoints;
}
```

# Immutable task

```
public class IDIClient {
 private final String name;
 private final int debt;
 private final int bonusPoints;

 public IDIClient(String name, int debt, int bonusPoints) {
 this.name = name;
 this.debt = debt;
 this.bonusPoints = bonusPoints;
 }
 public String getName() {
 return name;
 }
 public int getDebt() {
 return debt;
 }
 public int getBonusPoints() {
 return bonusPoints;
 }
}
```

Something like that?

```
IDIClient orit = new IDIClient(100, 50, "Orit");
```

Any problems?

Three setters can be replaced by the constructor. And what about 5?6?10? How can I remember that second int, which constructor receive is debt?

# Builder Pattern

```
public class IDIClient {
 private int bonusPoints;
 private int debt;
 private String name;

 private IDIClient() {}

 public static class Builder {...}
 public int getBonusPoints() {
 return bonusPoints;
 }

 public int getDebt() {
 return debt;
 }

 public String getName() {
 return name;
 }
 }

 public static class Builder {
 private IDIClient client;

 public Builder() {
 client = new IDIClient();
 }

 public void setBonusPoints(int bonusPoints) {
 client.bonusPoints = bonusPoints;
 }

 public void setDebt(int debt) {
 client.debt = debt;
 }

 public void setName(String name) {
 client.name = name;
 }

 public IDIClient newInstance() {
 IDIClient immutableClient = client;
 client = new IDIClient();
 return immutableClient;
 }
 }
```

# Builder Pattern

```
public static void main(String[] args) {
 IDIClient.Builder builder = new IDIClient.Builder();
 builder.setBonusPoints(100);
 builder.setDebt(50);
 builder.setName("Orit");
 IDIClient idiClient = builder.newInstance();
 System.out.println(idiClient);
}
```

```
IDIClient orit = new IDIClient.Builder().bonusPoints(100).debt(50).name("Orit").newInstance();
```

```
public static class Builder {
 private IDIClient client;

 public Builder() {
 client = new IDIClient();
 }

 public Builder bonusPoints(int bonusPoints) {
 client.bonusPoints = bonusPoints;
 return this;
 }

 public Builder debt(int debt) {
 client.debt = debt;
 return this;
 }

 public Builder name(String name) {
 client.name = name;
 return this;
 }

 public IDIClient newInstance() {
 IDIClient immutableClient = client;
 client = new IDIClient();
 return immutableClient;
 }
}
```

# Builder Pattern



But I can't use  
final variables,  
because builder  
has to set them  
after object had  
been created

```
public class IDIClient {
 private int bonusPoints;
 private int debt;
 private String name;

 private IDIClient() {}

 public static class Builder {...}
```



Look at your  
“immutable  
class”. All your  
state is not **final**  
now.

# Builder Pattern

- Another problem of current solution that you can create object only partly filled with state
- What about mandatory state?

Think about  
another solution



# Builder Pattern

```
public class IDIClient {
 private final String name;
 private final int debt;
 private final int bonusPoints;

 private IDIClient(String name, int debt, int bonusPoints) {
 this.name = name;
 this.debt = debt;
 this.bonusPoints = bonusPoints;
 }
 public static class Builder {
 public String getName() {
 return name;
 }
 public int getDebt() {
 return debt;
 }
 public int getBonusPoints() {
 return bonusPoints;
 }
 }
}
```

Next slide

# Builder Pattern

```
public static class Builder {
 private String name;
 private int debt;
 private int bonusPoints;
 public Builder() {}
 public Builder name(String name) {
 this.name = name;
 return this;
 }
 private void cleanBuilderFields() {...}
 private boolean clientIsValid() {...}

 public Builder debt(int debt) {
 this.debt = debt;
 return this;
 }
 public Builder bonusPoints(int bonusPoints) {
 this.bonusPoints = bonusPoints;
 return this;
 }
 public IDIClient createIDIClient() {
 if (clientIsValid()) {
 IDIClient client = new IDIClient(name, debt, bonusPoints);
 cleanBuilderFields();
 return client;
 }
 throw new IDIClientCanBeBuildException("name is null");
 }
}

private void cleanBuilderFields() {
 name = null;
 debt = 0;
 bonusPoints = 0;
}

private boolean clientIsValid() {
 if (name == null) return false;
 return true;
}
```

# Builder Pattern



And what  
about  
mutators?  
We want  
mutators!

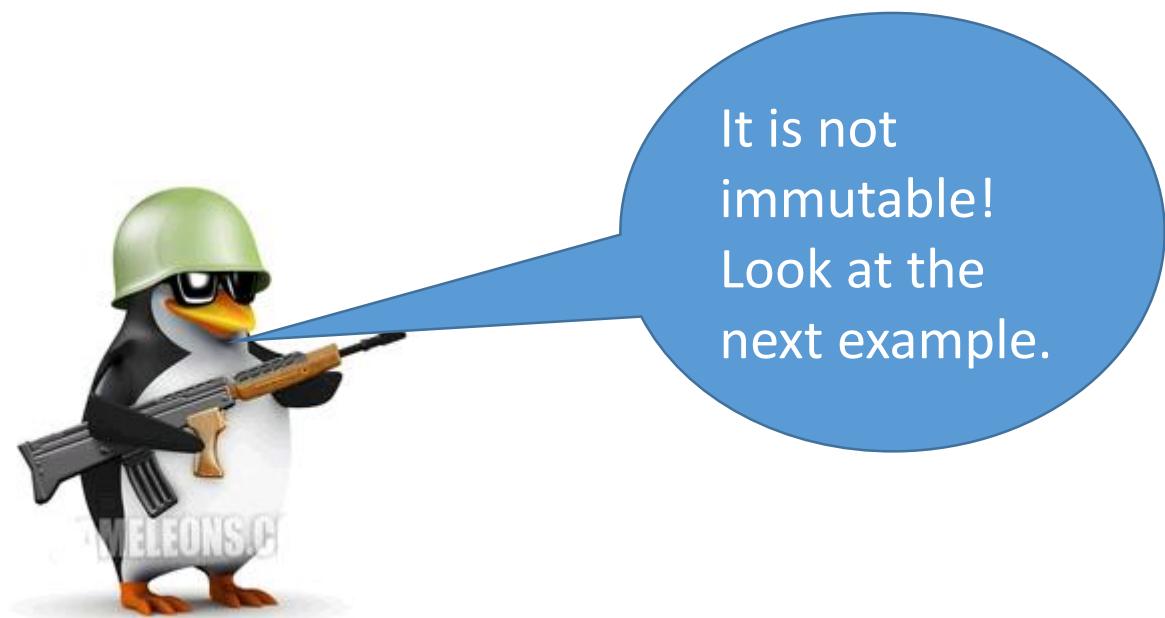
# Builder Pattern

```
public class IDIClient {
 private final String name;
 private final int debt;
 private final int bonusPoints;
 private IDIClient(String name, int debt, int bonusPoints) {
 this.name = name;
 this.debt = debt;
 this.bonusPoints = bonusPoints;
 }
 public static class Builder {...}

 public IDIClient setName(String name) {
 return new IDIClient(name, bonusPoints, debt);
 }
 public IDIClient setBonusPoints(int bonusPoints) {
 return new IDIClient(name, bonusPoints, debt);
 }
 public IDIClient setDebt(int debt) {
 return new IDIClient(name, bonusPoints, debt);
 }
 public String getName() {...}
 public int getDebt() {...}
 public int getBonusPoints() {...}
}
```

# Builder Pattern

- What about the immutable class keep reference to non immutable object?
- It will be possible to get it with getter and change it afterwards.



# Builder Pattern

```
public class IDIClient {
 private final String name;
 private final int debt;
 private final int bonusPoints;
 private final Date date;
 private IDIClient(String name, int debt, int bonusPoints, Date date) {
 this.name = name;
 this.debt = debt;
 this.bonusPoints = bonusPoints;
 this.date = date;
 }
 public Date getDate() {
 return date;
 }
 public Builder date(Date date) {
 this.date = date;
 return this;
 }
}
IDIClient client = new IDIClient.Builder().name("Vadim").date(date).createIDIClient();
client.getDate().setTime(1000)
```



# שוב פינגוין דפק אותו



# Defensive copy

- Defensive copy solution is: **Never** return original reference to inner state of immutable object. Copy inner object. Return the copy.

- It's simple:

```
public Date getDate() {
 return defensiveDateCopy(date);
}
```

- Very simple:
- Don't use mutable objects in immutable ones. In case of Date use JodaTime

# Defensive copy



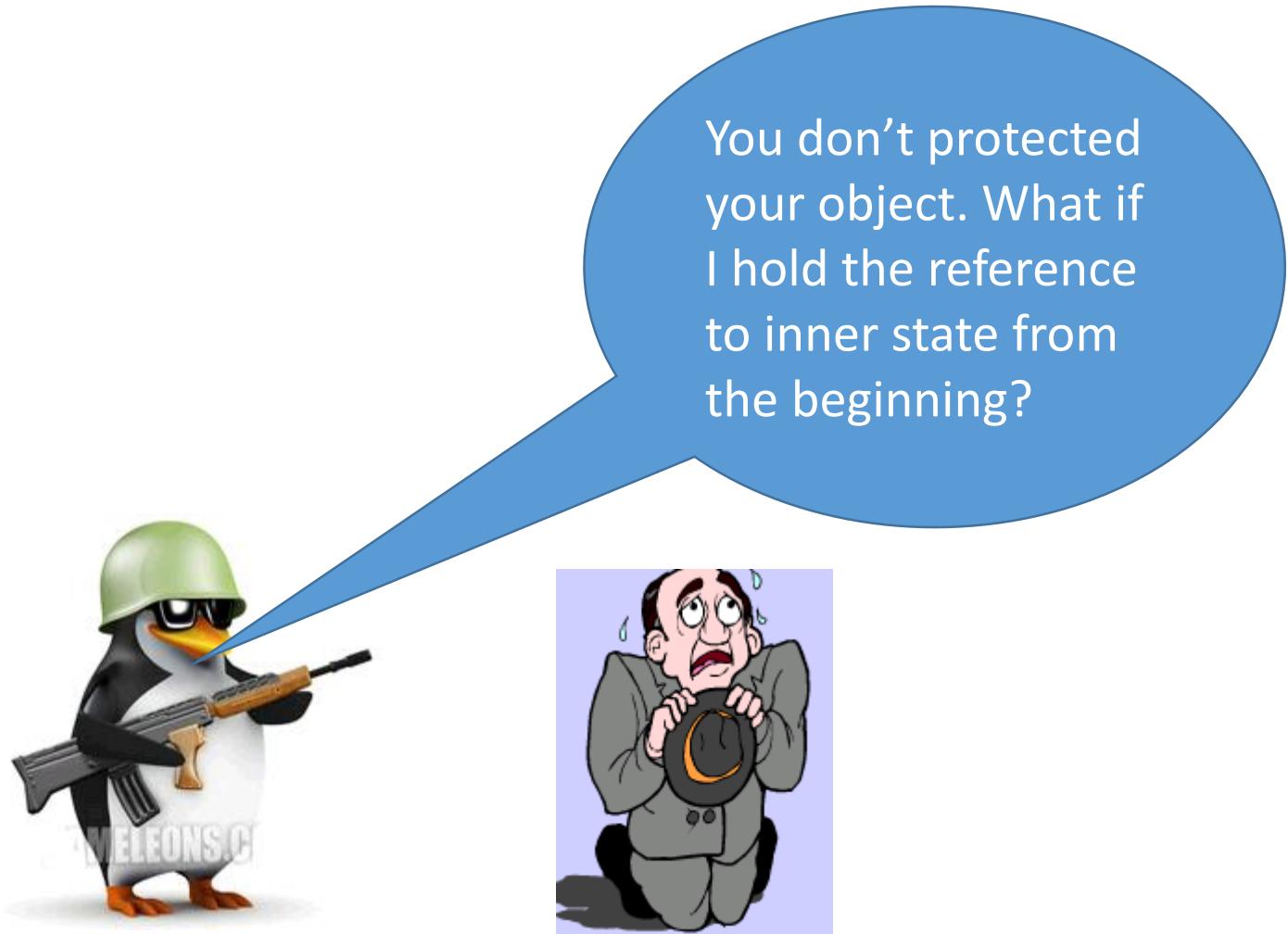
You can't  
see any  
problem?

```
private Date defensiveDateCopy(Date date) {
 return date == null ? null : new Date(date.getTime());
}
}
} else {
 return new Date(date.getTime());
}
```

# Builder Pattern



# Builder Pattern



You don't protected  
your object. What if  
I hold the reference  
to inner state from  
the beginning?

# Builder Pattern

```
Date date = new Date();
IDIClient masha = new IDIClient.Builder().name("Masha").date(date).createIDIClient();

date.setTime(1000000);
```



# Use defensive copy in constructor

```
public Builder date(Date date) {
 this.date = new Date(date.getTime());
 return this;
}
```

# Now you have really immutable object



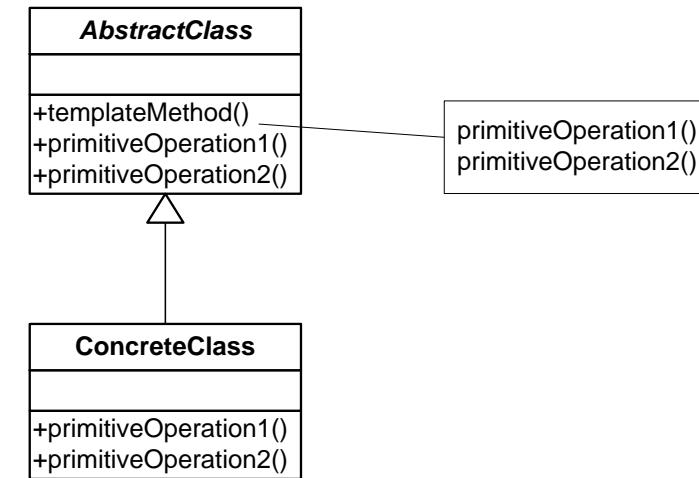
Task – write a platform for board games developers

# Solution – Template method

- Define a method, which will serve as a template for algorithms, handled by the subclasses
- Template method defines the steps of an algorithm and allows subclasses to provide the implementation for some steps, without changing the structure

# Template Method – The Solution

- Optionally, superclass may provide default implementation, then it's the superclass, which decides which to use, based on some conditions



# Another Task

- Suppose you have a computer game like Heroes.
- You need to write utility class, which will be able to calculate players wealth.
- Every player's possession can be transformed to money by it's unique formulae.
- For example each character has it's cost, each castle's cost depends on how much building it has, forest costs depends on amount of trees e.t.c.
- Service will get a list of all these objects and return the total cost.

# Iterator

```
public class WealthCalculator {
 public double calculateWealth(List<Costable> costables) {
 double sum=0;
 for (Costable costable : costables) {
 sum += costable.calculateCost();
 }
 return sum;
 }

 public interface Costable {
 double calculateCost();
 }
}
```

# Another task

- We need classes: Shelf, Cupboard, Room and book
- Shelf, cupboard and room a containers and book is item
- Container can include other containers and items
- Items can be included inside containers, but can't include something
- New containers or items can be added later
- Both items and containers have method calcPrice.
- Price of container will be calculated as sum of his own price and all prices items it includes

# Composite

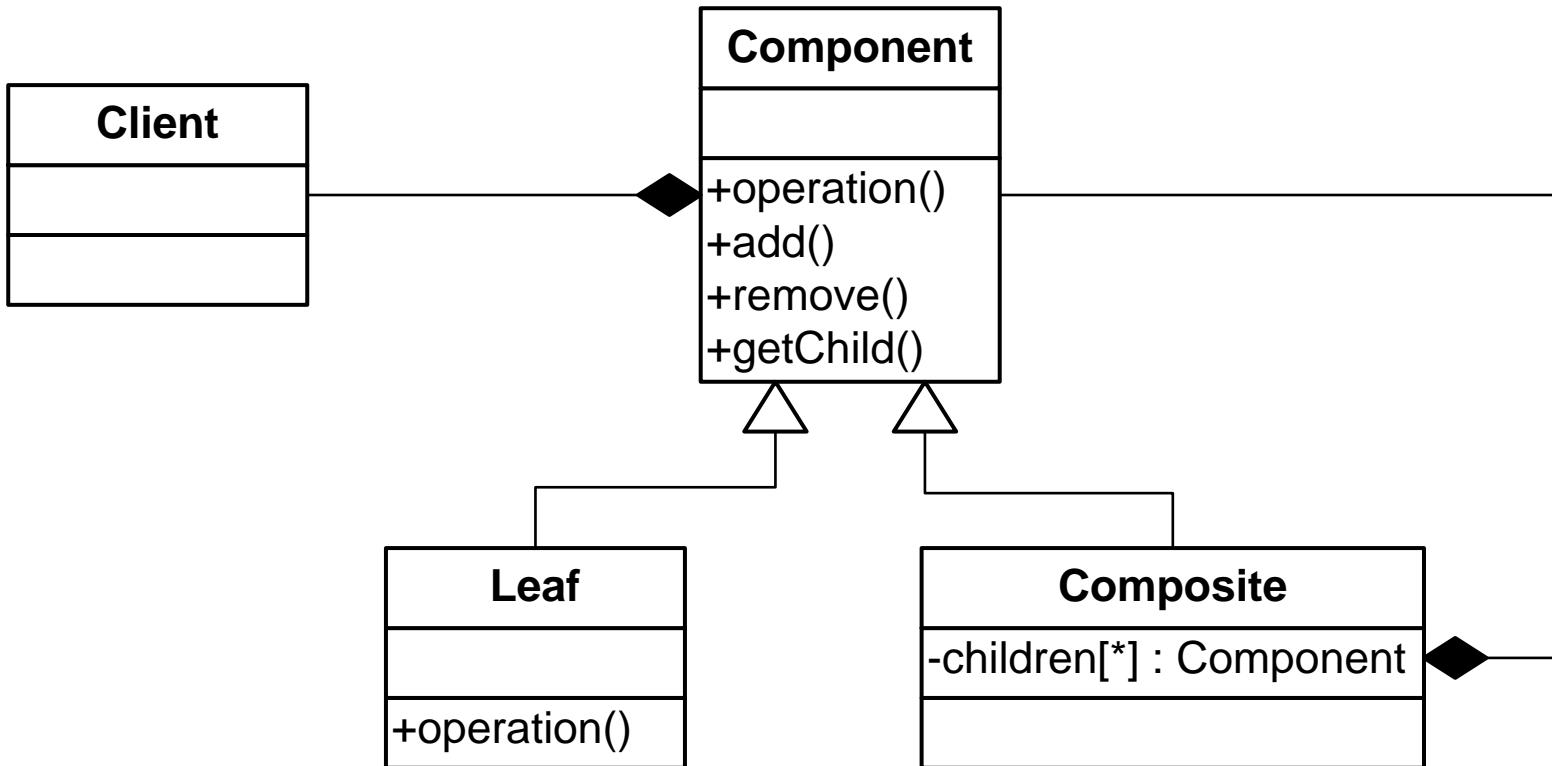
# Composite – The Problem

- Sometimes we want to iterate tree structures and not list structures
- Iterating trees complicates the code because distinguishing between composites and leafs should be made
- Usually, the iterating part doesn't need to know of such complication

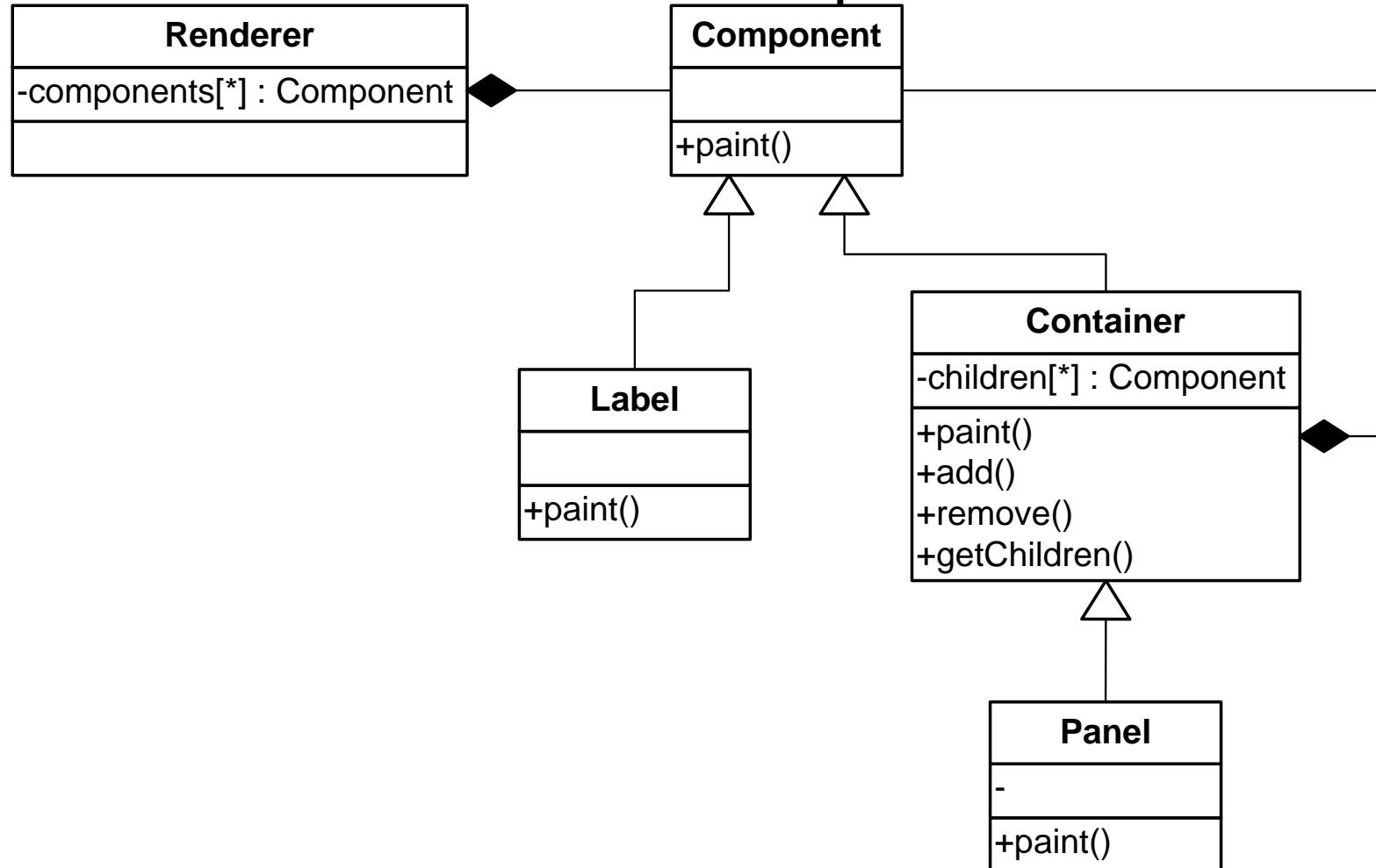
# Composite – Problem Example

- UI framework renders some page
- It commands to the components on page to render themselves
- Part of those components are in fact containers, which contain other components
- UI framework should be agnostic to that fact

# Composite – The Solution



# Composite – Solution Example



# Composite – Solution Example

```
1 public interface Component {
2
3 void paint();
4 }
```

```
1 public abstract class Container implements Component {
2
3 public List<Component> children;
4
5 public boolean add(Component component) {
6 return children.add(component);
7 }
8
9 public Component remove(int index) {
10 return children.remove(index);
11 }
12
13 public void paint() {
14 for (Component component : children) {
15 component.paint();
16 }
17 }
18}
19 }
```

```
1 public class Label implements Component {
2 public void paint() {
3 //paint label
4 }
5 }
6 }
```

```
1 public class Panel extends Container {
2
3 @Override
4 public void paint() {
5 //paint itself - borders etc.
6 super.paint();
7 }
8 }
```

---

```
1 public class UiRenderer {
2
3 List<Component> components;
4
5 public void paintUi() {
6 for (Component component : components) {
7 component.paint();
8 }
9 }
10 }
```

# More complicated task

- Write Utility class which will get a list of objects and specific object
- Method will calculate how much objects in the list are equals to the specific object

Just implement equal method



# Not so simple...

- Suppose that not all objects are implemented by you.
- You have infrastructure objects
- You have proxy objects
- And most important, you want that it will be possible to decide equality per specific method invocation

# Closures

- What can you pass to the method, as its args?
  - primitive, references to objects, collections
- Can you pass some algorithm or another function to method?
- Only in JAVA 8
- What can we do now?
- Closure

# Closures or Calback method

```
public class DuplicateCounterUtil {
 public static <T> int calcDuplicates(List<T> objects, T obj, MyEqualator<T> equalator) {
 int counter = 0;
 for (T object : objects) {
 if (equalator.isEqual(object, obj)) {
 counter++;
 }
 }
 return counter;
 }
}
```

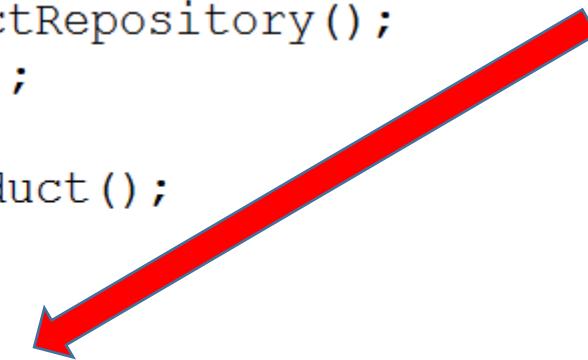
# Closures

- Another example is method sort of java Collections

# New task - migration

```
ProductRepository productRepository = new ProductRepository();
ArrayList<Costable> products = new ArrayList<>();
for (int i = 0; i < 5; i++) {
 Costable product = productRepository.getProduct();
 products.add(product);
}
int total = Calculator.calculateTotal(products);
System.out.println("total = " + total);
```

```
public class Calculator {
 public static int calculateTotal(List<Priceble> items) {
 int total = 0;
 for (Priceble item : items) {
 total += item.getPrice();
 }
 return total;
 }
}
```



# Adapter

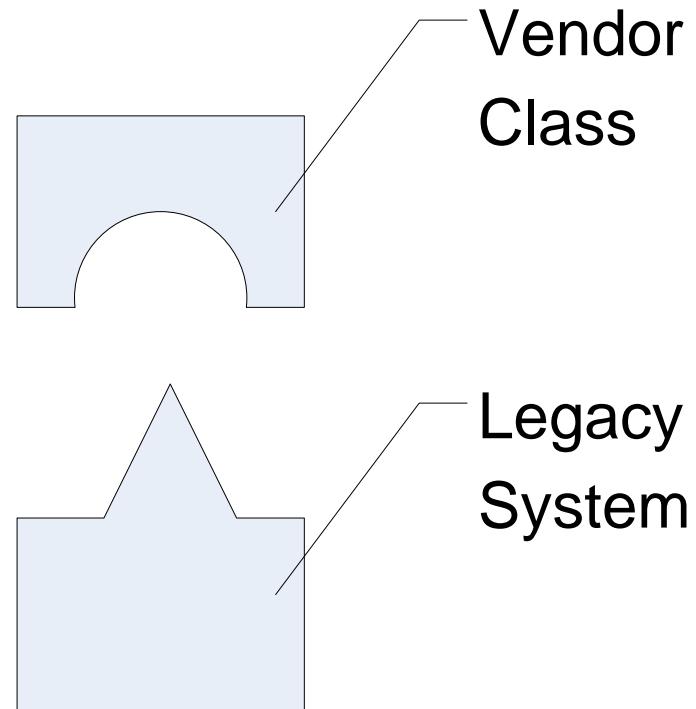
# Adapters all around



- Simple
  - Only change interface
- More complex
  - Have some processing inside

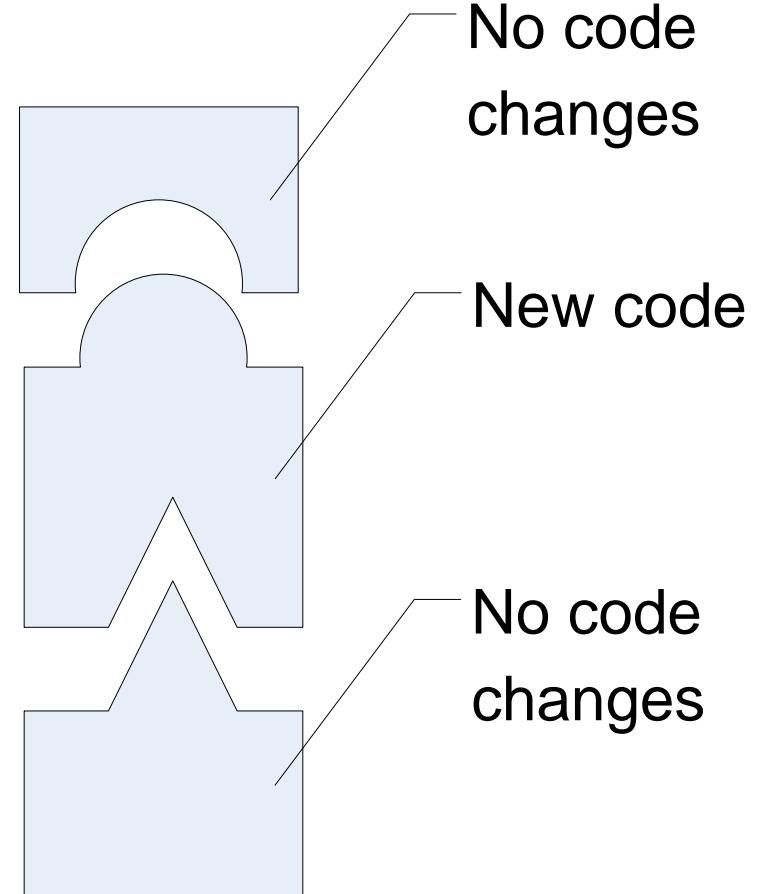
# Adapter - The problem

- Sometimes there is a client with incompatible interface
- How can we use the interface without changing existent code?

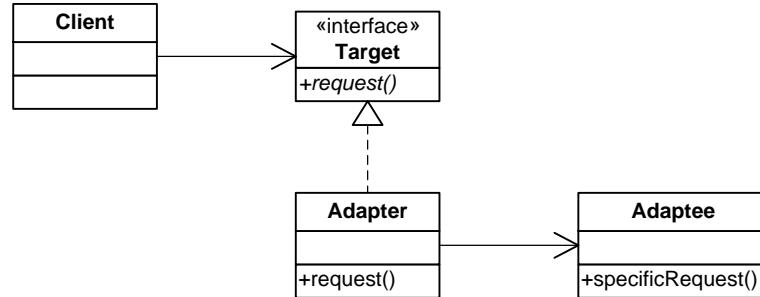


# Adapter – The Solution

- As usual, another layer of indirection solves the problem
- The adapter implements interface of the legacy system
- And talks to the vendor interface to service the requests
- Players:
  - Adapter implements Target interface, which is known to the Client
  - Adapter passes requests to the Adaptee



# Adapter – The Solution



- Adapter uses composition to pass the requests to from the Target to Adaptee
  - Another type of Adapter - Class Adapter subclasses both Target and Adaptee and overrides the requests
    - Multiple inheritance, not possible in Java

# Do you know what is Benchmark?

- 4 levels of benchmark
  - 1. Student
  - 2. Junior Software Engineer
  - 3. Senior Software Engineer
  - 4. Architect

# Level 1 - student



# Level 2- Junior Software Engineer

```
public static void main(String[] args) {
 Random random = new Random();
 long before = System.nanoTime();
 for (int i = 0; i < 1000000; i++) {
 random.nextInt(100);
 }
 long after = System.nanoTime();
 System.out.println((after-before)/1000000);
}
```

# Level 3 - Senior Software Engineer

```
public static void main(String[] args) {
 Random random = new Random();
 int unhappyNumber= 0;
 long before = System.nanoTime();
 for (int i = 0; i < 1000000; i++) {
 unhappyNumber = random.nextInt(100);
 }
 long after = System.nanoTime();
 System.out.println((after-before)/1000000);
 System.out.println(unhappyNumber);
}
```

# Level 4 - architext



# Task

- You have a MainService class, which has 2 methods (setDao and doWork)
- In the doWork method service delegating to savePerson method (the method of Dao interface)
- You have DaoImpl class
- But you want to benchmark the savePerson method everytime when Service delegating to it in doWork.
- You can't change any of existing classes

# Proxy

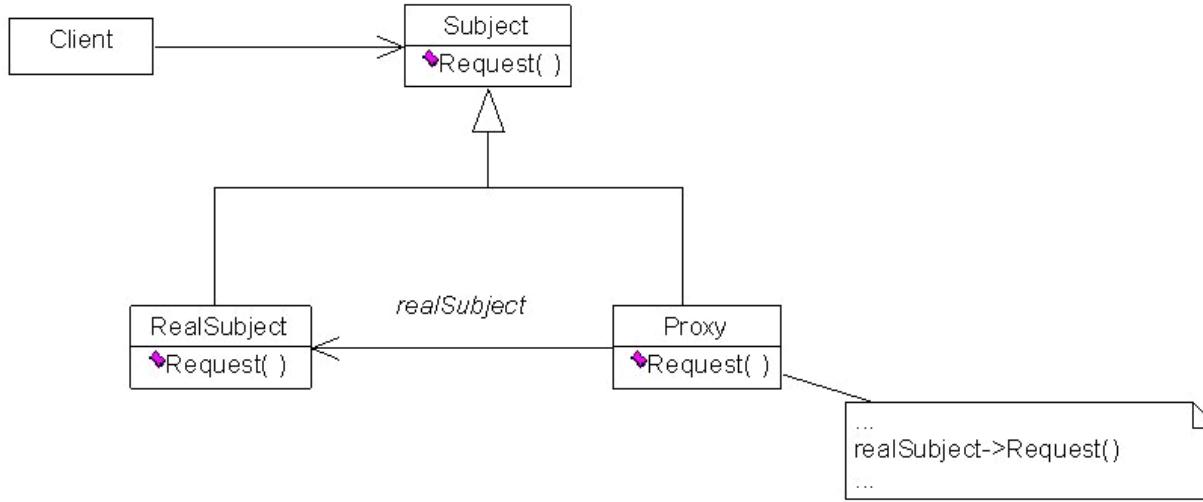
# Proxy - The problem

- Control access to an object
  - Access to remote objects
  - Manage permissions
  - Defer the cost of creation
  - Provide a smart reference

# Proxy – The Solution

- Provide a surrogate/placeholder object
- Proxy pattern composes an object and provides an identical interface to clients
- Proxy pattern describe how to provide a level of indirection to an object, and the implementations of the proxy object keep a reference to another object to which they forwards requests

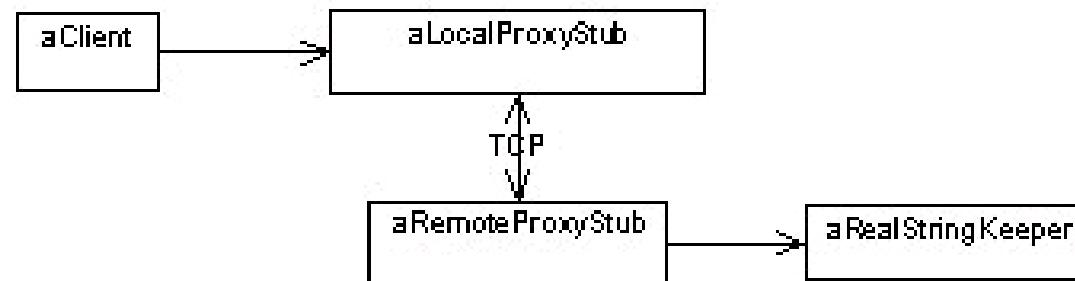
# Proxy – The Solution



- **Proxy**
  - maintains a reference that lets the proxy access the real subject.
  - provides an interface identical to Subject's so that a proxy can be substituted for real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
- **Subject**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**
  - defines the real object that the proxy represents.

# Proxy – Solution Example

- Remote Proxy
  - Provides a local representative for an object in a different JVM.
  - Remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject.



# Another proxy task

- You have the service class which holds PersonDaoImpl class which implements PersonDao interface.
- You need to cache all Persons which method findPersonBild returns.
- Use proxy design pattern

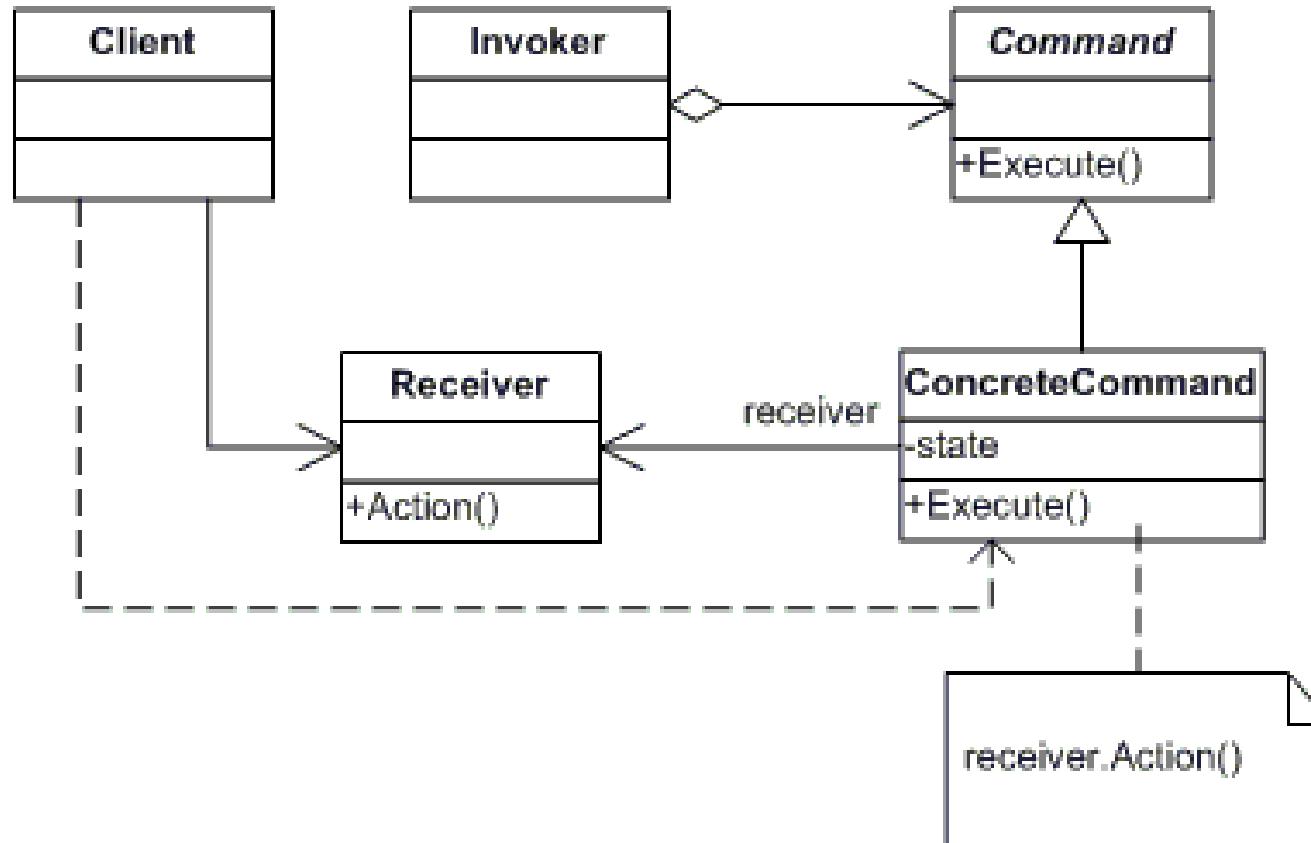
# Dynamic Proxy

- Write Factory class, which is responsible of all object creations.
- All classes which annotated with `@Benchmark` annotation / attribute should be wrapped with proxy which will print to log all benchmark about each method invocation

# Task

- Write BankAccount class with id, balance and increase/decreaseBalance methods
- Write AccountManager with methods:
  - transferMoney(BankAccount a1, BankAccount a2, int amount)
  - depositeMoney(BankAccount account, int amount)
  - withDrawMoney(BankAccount account, int amount)
  - undo() – this method abort last opration. If this method will be executed sever times it will abort several last operations

# Command



# Write Gradle (build tool)

- We need a platform for writing tasks
- Each task can depends on other tasks
- Each task has an action
- The client of our infrastructure must be able to create the lifecycle for his own build and later invoke action of each task in the chain
- Lets see example

# Deploy task

- Deploy depends on build
- Build depends on assemble and check
- Check depends on pack
- Pack depends on runTests
- RunTests depends on compileTests
- CompileTests dependsOn compileSources

If later the action of task deploy will be invoked the output will be:

- Compiling sources
- compiling tests...
- tests are running
- packing classes...
- Assembling...
- checking...
- Build finished!!!
- deploying...