# REFLECTION API

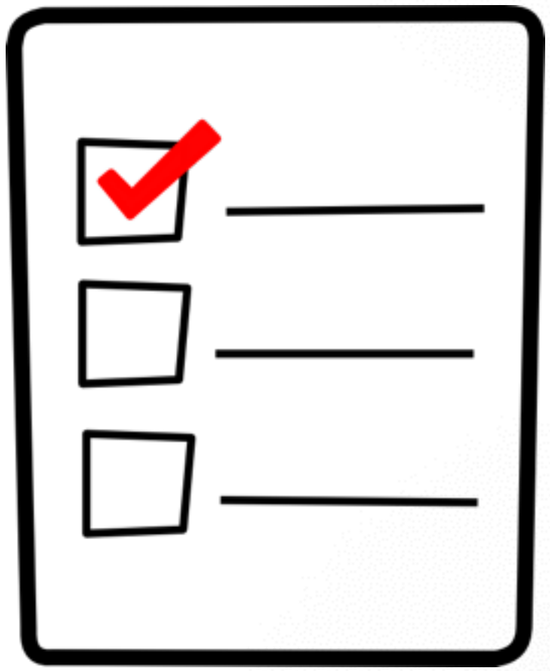
## JAVA FUNDAMENTALS

@ANTONARHIPOV

ZEROTURNAROUND

# AGENDA

- Reflection API
- Dynamic Proxies
- Some dirty stuff...



# METAPROGRAMMING

… is the writing of computer programs that write or manipulate other programs as their data, or that do part of the work at compile time that would otherwise be done at **runtime**

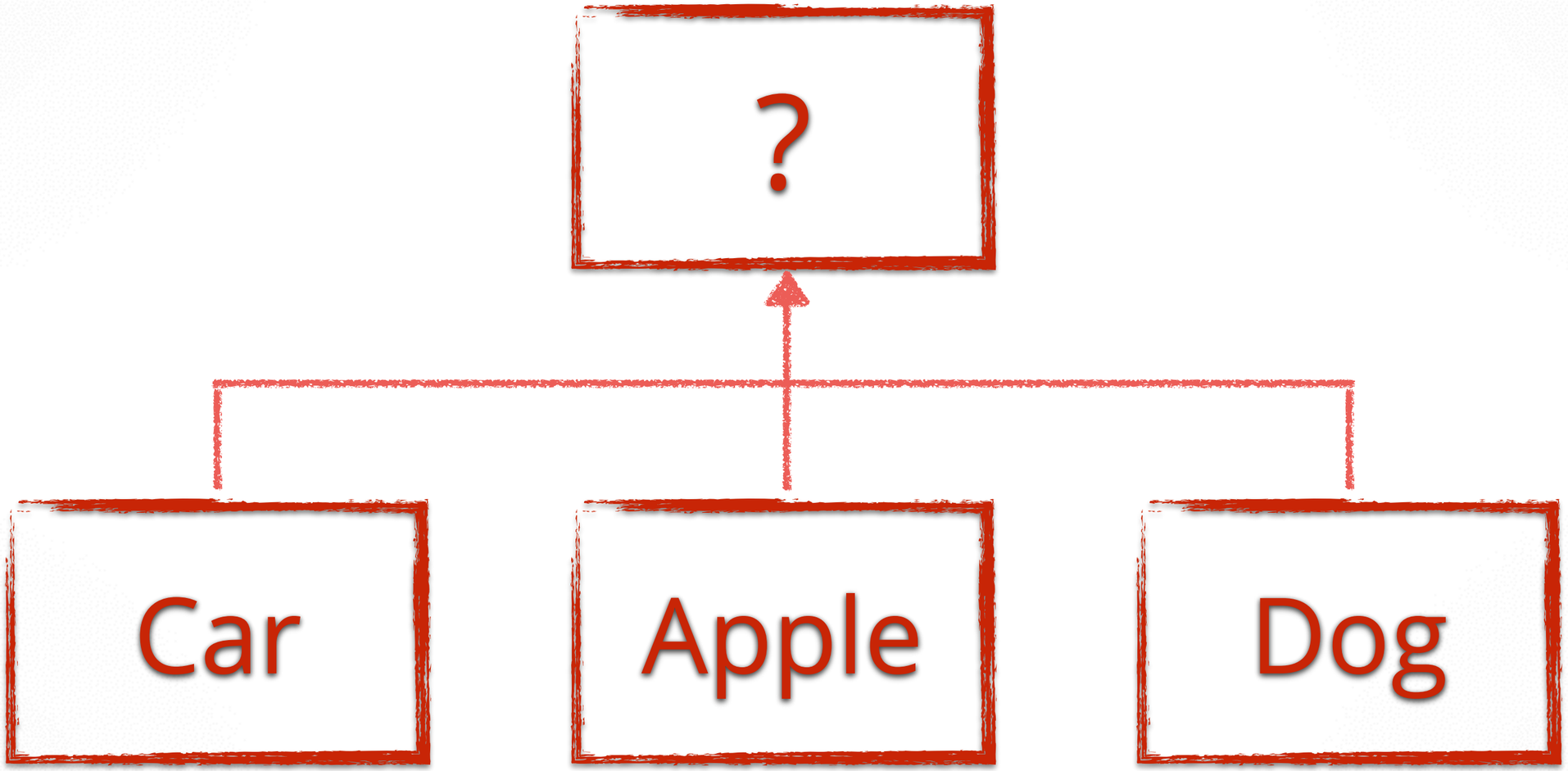
# REFLECTION API

- Find details about the object **at runtime**
- Call methods, manipulate fields, create instances, etc

# BENEFITS

- ***Flexibility*** - dynamically decide upon program execution
- ***Raw power*** - reading private data
- ***Pure magic*** - do things you should not be able to do otherwise

```
car.setColor(RED);
apple.setColor(RED);
dog.setColor(RED);
```


?
Car
Apple
Dog

```
public class ColorSettingUtil {

  public static void setColor(Car car){
    car.setColor(Color.RED);
  }

  public static void setColor(Apple apple){
    apple.setColor(Color.RED);
  }

  public static void setColor(Dog dog){
    dog.setColor(Color.RED);
  }

}
```

```
public class ColorSettingUtil {

   public static void setColor(Object object){
      if(object instanceof Car){
         ((Car) object).setColor(Color.RED);
      } else if (object instanceof Apple){
         ((Apple) object).setColor(Color.RED);
      } else if (object instanceof Dog){
         ((Dog) object).setColor(Color.RED);
      }
   }

}
```

```
public class ColorSettingUtil {

  public static void setColor(Object object, Color color){

    Class cls = object.getClass();

    Method method = cls.getMethod("setColor",
       new Class[] {Color.class});

    method.invoke(object, new Object[] {color});

  }

}
```

```
Class c1 = object.getClass();
Class c2 = SomeClass.class;
```

# ACCESSING METHODS

```
Method method = cls.getMethod("setColor",
        new Class[] {Color.class});

Method[] methods = cls.getMethods();

for(Method m : ms){ … }

Method declaredMethod = cls.getDeclaredMethod(…);

Methods[] declaredMethods = cls.getDeclaredMethods();
```

# REFLECTIVE INVOCATION

```
Object invoke(Object obj, Object… args)
  throws IllegalAccessException,
         IllegalArgumentException,
         InvocationTargetException


// instance method
method.invoke(object, new Object[] { color });
// static method
method.invoke(null);
```

# INTROSPECTING FIELDS

```
Object data = …
Class c = data.getClass();

Field f1 = c.getDeclaredField("id");
Field f2 = c.getField("name");
Field f3 = c.getField("address");

Integer id = f1.getInt(data);
String name = (String) f2.get(data);
Address name = (Address) f3.get(data);
```

# ACCESSING PRIVATE STATE

```
Object data = …
Class c = data.getClass();
Field field = data.getDeclaredField("someField");
field.setAccessible(true);    // allows us to access private members
Object value = field.get(data);
field.set(data, new Object()); // set new value for the private field
```

# `setAccessible(true)`

Disables “private”

Disables “final”

Disables security checks

# MODIFYING PRIVATE STATE

```
public class StringDestroyer {

  public static void main(String[] args) throws Exception {

    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    value.set("hello!", "cheers!".toCharArray());
    System.out.println("hello!"); // prints “cheers!”

  }

}
```

# MODIFYING PRIVATE STATE

- String is a special case!
- Shared object between classes in the same static context:

```
System.out.println("hello!"); // prints “hello!”
StringDestroyer.main(null);
System.out.println("hello!".equals("cheers!")); // true
```

# SAME FOR INTEGERS...

```
Field value = Integer.class.getDeclaredField("value");
value.setAccessible(true);
value.set(42, 43);
System.out.printf("Six times seven = %d%n", 6 * 7);
```

Yup... Six times Seven = 43

# ANNOTATIONS

- Annotations, a form of **metadata**, provide additional information about a program that is not part of the program itself
- Use cases:

  Information for the compiler (ex.: `@Override`)

  Compile-time and deployment-time processing

  Runtime processing

https://docs.oracle.com/javase/tutorial/java/annotations/

# ANNOTATIONS

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Resource {
  String path() default "/"
}

@Resource("/hello")
public class Hello { }
```

# ANNOTATIONS

```
Class c = …;
c.isAnnotationPresent(Resource.class);
Resource r = c.getAnnotation(Resourse.class);
String = r.path();
```

# DYNAMIC PROXY

***Dynamic proxy class*** implements a list of interfaces specified when the class is created

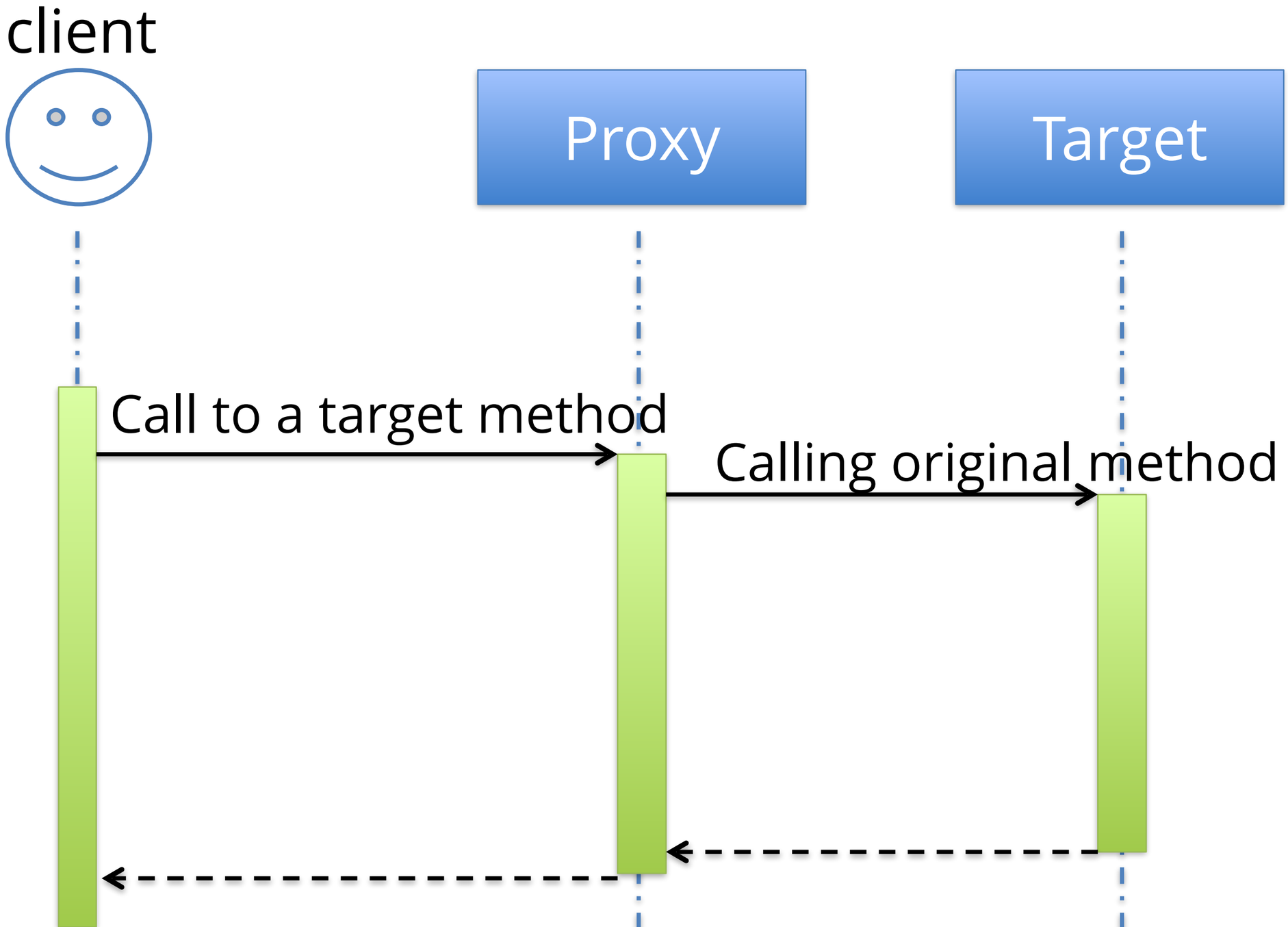
https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

# DYNAMIC PROXY

Use cases:

- Database connections & transaction management
- Mock objects for unit testing
- AOP-like method interception
- Dependency injection

http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html


client
Proxy
Target
Call to a target method
Calling original method

```java
public interface Engine {
  void start();
  void stop();
  void wroom();
}
```

```java
public class V8 implements Engine {

  public void start() {
    System.out.println("V8.start");
  }

  @Override
  public void stop() {
    System.out.println("V8.stop");
  }

  public void wroom() {
    System.out.println("V8.wroom");
  }
}
```

# DYNAMIC PROXY

```
Object target = new V8();

Engine engine = (Engine) Proxy.newProxyInstance(
  target.getClass().getClassLoader(),
  new Class[]{ Engine.class },
  new InvocationHandler() {
  @Override
    public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable {
       // transparent proxy, only delegates the call to target object
      return method.invoke(target, args);
    }
  }
);
```

# DYNAMIC PROXY

```
Object target = new V8();

Engine engine = (Engine) Proxy.newProxyInstance(
  target.getClass().getClassLoader(),
  new Class[]{ Engine.class },
   // transparent proxy, only delegates the call to target object
  (proxy, method, args) -> method.invoke(target, args));
);
```

Since InvocationHandler only has a single method,
we can use Java 8 lambda to implement it

# LET'S COUNT THE TARGET INVOCATIONS!

# A COUNTING HANDLER

```
public class CountingHandler implements InvocationHandler {

  int counter;
  Object target;

  public CountingHandler(Object target) { this.target = target; }

  public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    counter++;
    return method.invoke(target, args);
  }

  public int getCounter() { return counter; }
}
```

```
V8 v8 = new V8();
// create a proxy for V8 instance that conforms to Engine interface
Engine engine = (Engine) Proxy.newProxyInstance(
                    v8.getClass().getClassLoader(),
                    new Class[]{Engine.class},
                    new CountingHandler(v8));

// call some methods:
engine.start(); // V8.start
engine.wroom(); // V8.wroom
engine.stop();  // V8.stop

// we can ask for a handler instance from the particular proxy:
CountingHandler handler = (CountingHandler) Proxy.getInvocationHandler(engine);

System.out.println("nr. of invocations: " + handler.getCounter()); // 3
```

# DELEGATION VIA INVOCATIONHANDLER

*It is possible to trick the target system with a proxy to provide an implementation which doesn't implement the desired interfaces*

```
public interface Engine {
   void start();
   void stop();
   void wroom();
}

public class V12 {   // does not implement Engine interface
   public void wroom(){
      System.out.println("V12.wroom");
   }
}
```

// note that V12 doesn’t implement the Engine interface!

```java
public class EngineHandler implements InvocationHandler{

  Object target;

  public EngineHandler(Object target) {
    this.target = target;
  }

  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {

    System.out.println("Calling " + method.getName());
    Method realMethod = target.getClass().getDeclaredMethod(method.getName());
    return realMethod.invoke(target, args);
  }
}
```

```
V12 v12 = new V12();

Engine engine = (Engine)
Proxy.newProxyInstance(V12.class.getClassLoader(),
        new Class[]{Engine.class},
        new EngineHandler(v12));
```

// engine is now a proxy that mimics
// Engine interface, but doesn't
// actually implement it!

```
System.out.println("engine class = " + engine.getClass());

System.out.println("wroom");
engine.wroom();     // works, since V12 implements wroom() method
engine.stop();      // fails, no such method in V12
```

# REFLECTION API

- Reflection API Tutorial

  http://docs.oracle.com/javase/tutorial/reflect/

- Jakov Jenkov's Reflection API overview

  http://tutorials.jenkov.com/java-reflection/index.html

- Google :)

  Java Reflection

  Java Dynamic Proxy

# HOMEWORK

https://github.com/JavaFundamentalsZT/jf-hw-reflection