

# Dynamic Proxies in Java

**Dr Heinz M. Kabutz**

Last updated 2020-05-26

**© 2020 Heinz Kabutz – All Rights Reserved**



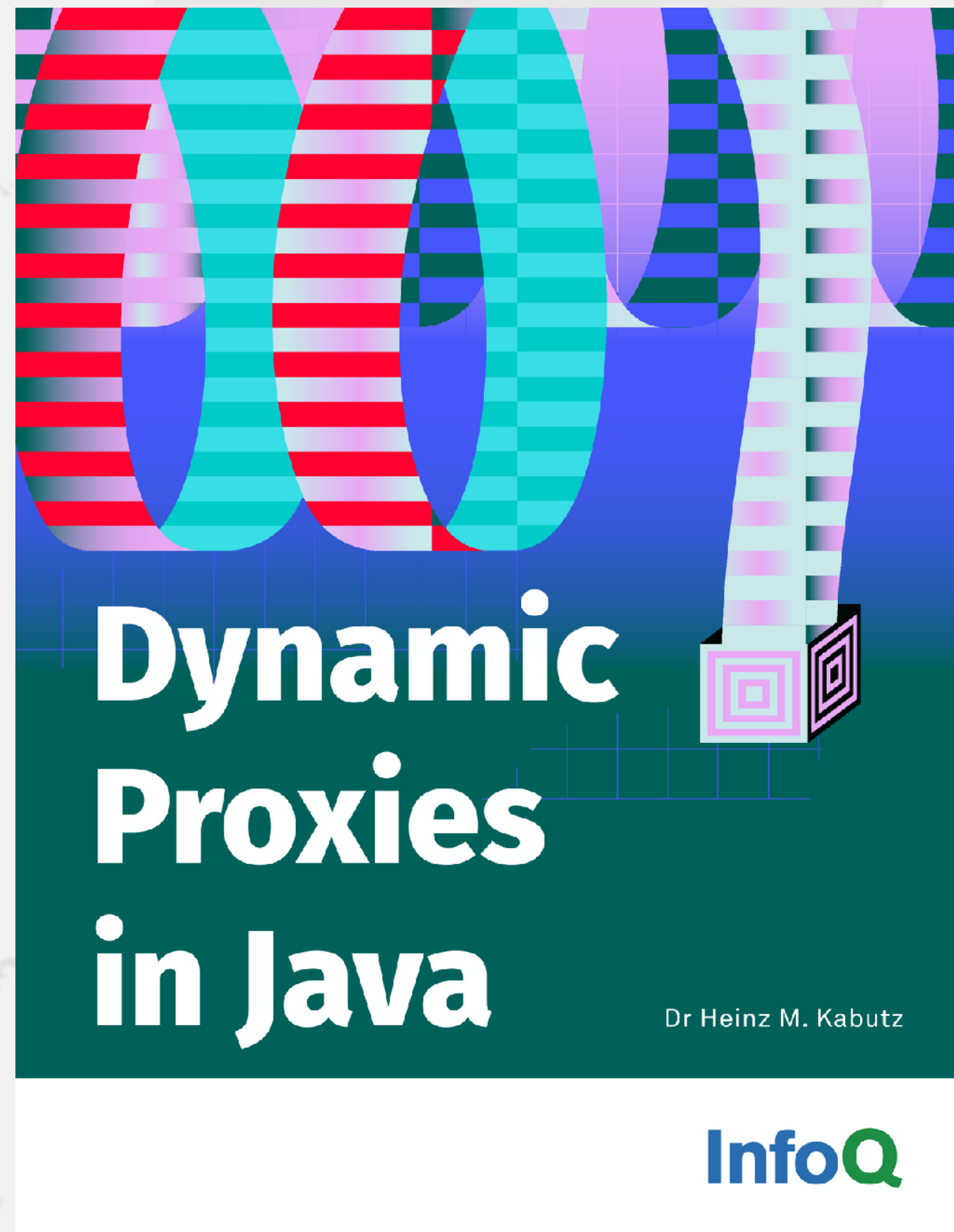


## whois

- **Not ketchup**
- **10+ years teaching remotely from Crete**
  - **learning.javaspecialists.eu**
  - **Threading and concurrency**
  - **Design patterns**
  - **Advanced Java topics**
  - **Refactoring old code to modern Java constructs**
  - **Contact: [heinz@javaspecialists.eu](mailto:heinz@javaspecialists.eu)**
- **Java Champion, Speaker, bla bla**
- **Gift: [tinyurl.com/JAX-Online-2020](https://tinyurl.com/JAX-Online-2020)**



## Dynamic Proxies in Java



● **Free download from**

– <https://www.infoq.com/minibooks/java-dynamic-proxies/>

## Questions

- **Please ask questions in the chat**
  - My assistant John Green will send them to me
- **Questions make a talk far more fun**
  - And we all learn more
- **Note:  $\approx$  15 second lag**



## History of Dynamic Proxies

- **RMI used to need a separate compile step**
  - Tool "rmic" still found in JDK/bin directory
    - Creates *stubs* and *skeletons* to manage remote method invocations
- **Java 1.3 released in May 2000**
  - First version with dynamic proxies
  - Functional interface of InvocationHandler to service *all* methods on proxy
  - Not necessary to use "rmic" or similar tools for deployment
  - Made it possible to build flexible, dynamic systems

## Big Win

- **Don't Repeat Yourself (DRY) at its best**
  - Write a single InvocationHandler implementation
  - Reuse for hundreds of classes
- **We once replaced 600,000 code statements with 1 dynamic proxy**
  - Code had been generated, but was maintained by hand
  - Dynamic proxy easier to maintain
  - Less code

## Infrastructure Code

- **Dynamic proxies in tools and frameworks**
  - Spring
  - Annotations
  - Dependency injection
  - Hibernate
  - Gradle



# 1: Handcrafted Proxies





## Handcrafted Proxies

- **Before learning how to avoid duplicate code, we will copy & paste**
  - And then in next section will use *dynamic proxies* instead

# Virtual Proxy





## Virtual Proxy

- **Delays expensive object creation**
  - placeholder object creates costly object on demand

# CustomMap Interface

- Reduced version of the Map interface

```
public interface CustomMap<K, V> {  
    int size();  
    V get(Object key);  
    V put(K key, V value);  
    V remove(Object key);  
    void clear();  
    void forEach(BiConsumer<? super K, ? super V> action);  
}
```



# CustomHashMap Implementation

- Delegates methods to a `java.util.HashMap`

- Repetitive and error prone

```
public class CustomHashMap<K, V> implements CustomMap<K, V> {  
    private final Map<K, V> map = new HashMap<>();  
    { System.out.println("CustomHashMap constructed"); }  
    public int size() { return map.size(); }  
    public V get(Object key) { return map.get(key); }  
    public V put(K key, V value) { return map.put(key, value); }  
    public V remove(Object key) { return map.remove(key); }  
    public void clear() { map.clear(); }  
    public void forEach(BiConsumer<? super K, ? super V> action) {  
        map.forEach(action);  
    }  
    public String toString() { return map.toString(); }  
}
```

## VirtualCustomMap Virtual Proxy

- **Has a reference to a Supplier for CustomMap**
  - Is created in the `getRealMap()` method

```
public class VirtualCustomMap<K, V> implements CustomMap<K, V> {  
    private final Supplier<CustomMap<K, V>> mapSupplier;  
    private CustomMap<K, V> realMap;  
    public VirtualCustomMap(Supplier<CustomMap<K, V>> mapSupplier) {  
        this.mapSupplier = mapSupplier;  
    }  
    private CustomMap<K, V> getRealMap() { // not thread-safe  
        if (realMap == null) realMap = mapSupplier.get();  
        return realMap;  
    }  
}
```



## VirtualCustomMap Methods

```
public int size() {
    return getRealMap().size();
}
public V get(Object key) {
    return getRealMap().get(key);
}
public V put(K key, V value) {
    return getRealMap().put(key, value);
}
public V remove(Object key) {
    return getRealMap().remove(key);
}
public void clear() { getRealMap().clear(); }
public void forEach(BiConsumer<? super K, ? super V> action) {
    getRealMap().forEach(action);
}
}
```

## Using VirtualCustomMap

- **CustomHashMap made when method called**

- **Does not matter which method we call first**

```
CustomMap<String, Integer> map =  
    new VirtualCustomMap<>(CustomHashMap::new);  
System.out.println("Virtual Map created");  
map.put("one", 1);  
map.put("life", 42);  
System.out.println("get(\"life\") = " + map.get("life"));  
System.out.println("size() = " + map.size());  
System.out.println("clearing map");  
map.clear();  
System.out.println("size() = " + map.size());
```

```
Virtual Map created  
CustomHashMap constructed  
get("life") = 42  
size() = 2  
clearing map  
size() = 0
```



# 2: Dynamic Proxy



## 2: Dynamic Proxy

- **Avoid copy and paste programming**
  - A bug needs to be fixed everywhere
- **Better is static or dynamic code generation**



# Proxy.newProxyInstance()



# Proxy.newInstance()

- **Takes three parameters**

- **ClassLoader** where the new proxy class is loaded
- **Class<?>[]** an array containing all interfaces our proxy object must implement
- **InvocationHandler** a handler that's called when any proxy method is invoked



## InvocationHandler

- **Invoked when *any* method is called on proxy**

```
public interface InvocationHandler {  
    public Object invoke(Object proxy,  
                        Method method,  
                        Object[] args) throws Throwable;  
}
```

- Object proxy is the instance of the dynamic proxy class that is calling invoke()
- Method method is a `java.lang.reflect.Method` for the method that was called
  - Either an interface method or `equals()`, `hashCode()`, or `toString()`
- Object[] args is an array of parameters passed into the method
  - This is null when method has no parameters



# LoggingInvocationHandler





## LoggingInvocationHandler

- **We will log all method calls**
  - Optionally measuring how long they take
- **The constructor parameters are**
  - Logger log a `java.util.Logger` to log to
  - Object obj **the object that we want to delegate the calls to**
    - **Must implement the same interfaces as the proxy**

```
public final class LoggingInvocationHandler
    implements InvocationHandler {
    private final Logger log;
    private final Object obj;
    public LoggingInvocationHandler(Logger log, Object obj) {
        this.log = log;
        this.obj = obj;
    }
}
```

## invoke() Method for Logging

```
public Object invoke(
    Object proxy, Method method, Object[] args)
    throws Throwable {
    log.info(() -> "Entering " + toString(method, args));
    // optimization - nanoTime() is an expensive native call
    final boolean logFine = log.isLoggable(Level.FINE);
    long start = logFine ? System.nanoTime() : 0;
    try {
        return method.invoke(obj, args);
    } finally {
        long nanos = logFine ? System.nanoTime() - start : 0;
        log.info(() -> "Exiting " + toString(method, args));
        if (logFine) log.fine(() -> "Time " + nanos + "ns");
    }
}
```



## toString() Prints Methods with Args

```
private String toString(Method method,
                        Object[] args) {
    return String.format("%s.%s(%s)",
        method.getDeclaringClass().getCanonicalName(),
        method.getName(),
        args == null ? "" :
            Stream.of(args).map(String::valueOf)
                .collect(Collectors.joining(", ")));
}
```

# Demo of LoggingInvocationHandler

```
@SuppressWarnings("unchecked")
var map = (Map<String, Integer>)
    Proxy.newProxyInstance(
        Map.class.getClassLoader(), new Class<?>[] {Map.class},
        new LoggingInvocationHandler(
            Logger.getGlobal(), new ConcurrentHashMap<>()));
map.put("one", 1);
map.put("two", 2);
System.out.println(map);
map.clear();
```

```
Jan 24, 2020 7:32:20 AM
eu.javaspecialists.books.dynamicproxies.ch03.logging.LoggingInvocationHandler invoke
INFO: Entering java.util.Map.put(one, 1)
Jan 24, 2020 7:32:21 AM
eu.javaspecialists.books.dynamicproxies.ch03.logging.LoggingInvocationHandler invoke
INFO: Exiting java.util.Map.put(one, 1)
Jan 24, 2020 7:32:21 AM
eu.javaspecialists.books.dynamicproxies.ch03.logging.LoggingInvocationHandler invoke
FINE: Time 61622ns
```



# Dissecting a Dynamic Proxy



## Dissecting a Dynamic Proxy

- We will start with a simple interface

```
public interface ISODateParser {  
    LocalDate parse(String date) throws ParseException;  
}
```



## Dynamic Proxy Class Name

- **Dynamic proxy with empty InvocationHandler**

```
System.out.println(
    Proxy.newProxyInstance(
        ISODateParser.class.getClassLoader(),
        new Class<?>[] {ISODateParser.class},
        (proxy, method, arguments) -> null
    ).getClass()
);
```

```
class com.sun.proxy.$Proxy0
```

## Decompiling \$Proxy0

- **We can dump generated proxy classes**
  - And then decompile with a tool like CFR
    - <https://www.benf.org/other/cfr/>
  - **Java 9+: -Djdk.proxy.ProxyGenerator.saveGeneratedFiles=true**
  - **Earlier versions: -Dsun.misc.ProxyGenerator.saveGeneratedFiles=true**



```
public final class $Proxy0 extends Proxy
    implements ISODateParser {
    private static Method m0;
    private static Method m1;
    private static Method m2;
    private static Method m3;
    static {
        try {
            m0 = Class.forName("java.lang.Object").getMethod("hashCode");
            m1 = Class.forName("java.lang.Object").getMethod("equals",
                Class.forName("java.lang.Object"));
            m2 = Class.forName("java.lang.Object").getMethod("toString");
            m3 = Class.forName("eu.javaspecialists.books.dynamicproxies." +
                "ch03.ISODateParser").getMethod("parse",
                Class.forName("java.lang.String"));
        } catch (NoSuchMethodException e) {
            throw new NoSuchMethodError(e.getMessage());
        } catch (ClassNotFoundException e) {
            throw new NoClassDefFoundError(e.getMessage());
        }
    }
}
```

```
public $Proxy0(InvocationHandler h) {super(h);}

public final int hashCode() {
    try {
        return (Integer) h.invoke(this, m0,
            (Object[]) null);
    } catch (RuntimeException | Error e) {
        throw e;
    } catch (Throwable e) {
        throw new UndeclaredThrowableException(e);
    }
}

public final boolean equals(Object o) {
    try {
        return (Boolean) h.invoke(this, m1, new Object[] {o});
    } catch (RuntimeException | Error e) {
        throw e;
    } catch (Throwable e) {
        throw new UndeclaredThrowableException(e);
    }
}
```



```
public final String toString() {
    try {
        return (String) h.invoke(this, m2,
            (Object[]) null);
    } catch (RuntimeException | Error e) {
        throw e;
    } catch (Throwable e) {
        throw new UndeclaredThrowableException(e);
    }
}

public final LocalDate parse(String s) throws ParseException {
    try {
        return (LocalDate) h.invoke(this, m3, new Object[] {s});
    } catch (RuntimeException | ParseException | Error e) {
        throw e;
    } catch (Throwable e) {
        throw new UndeclaredThrowableException(e);
    }
}
}
```

# Virtual Dynamic Proxy





## Virtual Dynamic Proxy

- **InvocationHandler for virtual proxies**

```
public final class VirtualProxyHandler<S>
    implements InvocationHandler, Serializable {
    private final Supplier<? extends S> subjectSupplier;
    private S subject;
    public VirtualProxyHandler(Supplier<? extends S> subjectSupplier) {
        this.subjectSupplier = subjectSupplier;
    }
    private S getSubject() {
        if (subject == null) subject = subjectSupplier.get();
        return subject;
    }
    public Object invoke(Object proxy, Method method,
                        Object[] args) throws Throwable {
        return method.invoke(getSubject(), args);
    }
}
```

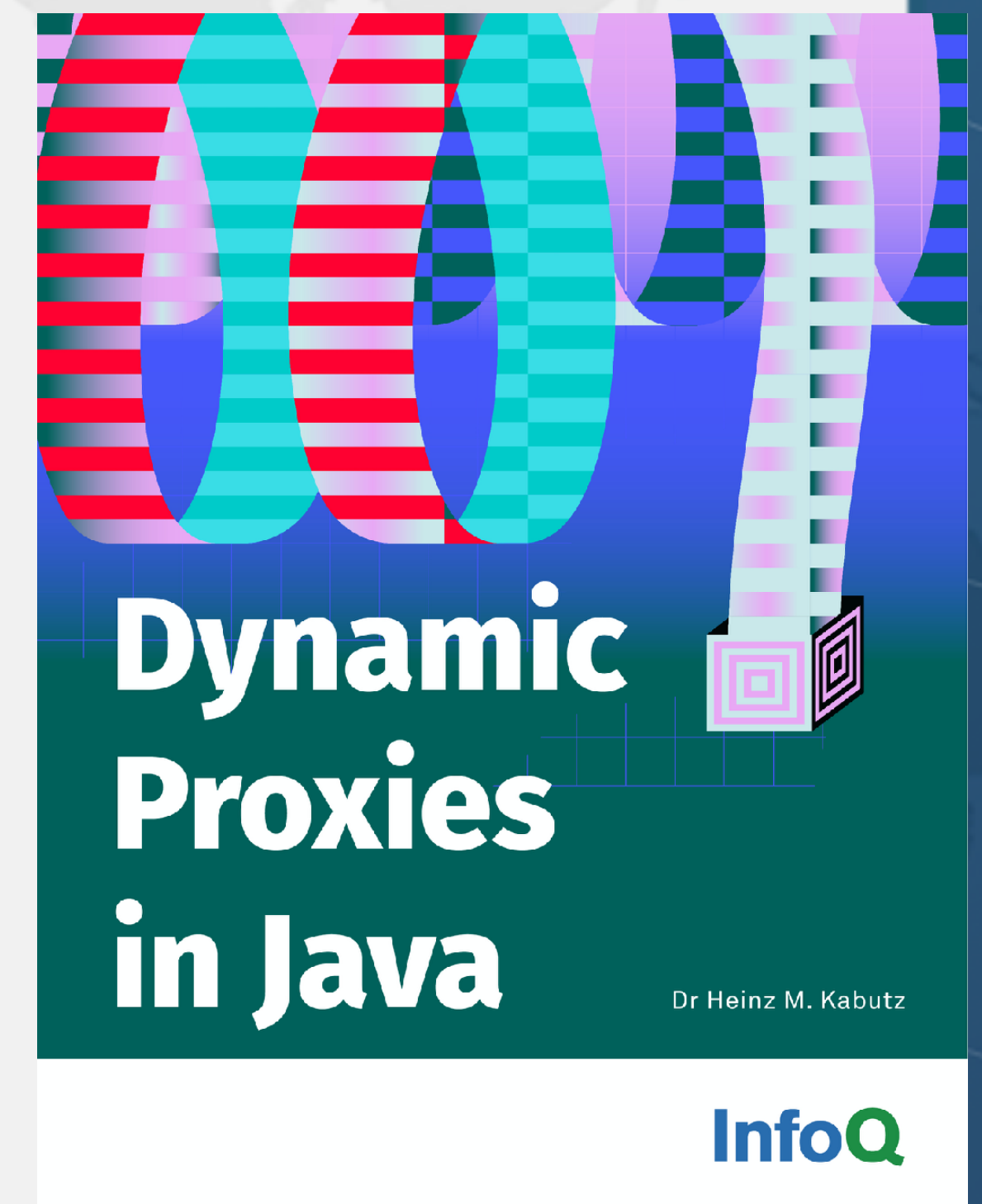
## Proxies Facade virtualProxy()

- Facade has a virtualProxy() method

```
public static <S> S virtualProxy(  
    Class<? super S> subjectInterface,  
    Supplier<? extends S> subjectSupplier) {  
    Objects.requireNonNull(subjectSupplier, "subjectSupplier==null");  
    return castProxy(subjectInterface,  
        new VirtualProxyHandler<>(subjectSupplier));  
}
```

- More details in book

– <https://www.infoq.com/minibooks/java-dynamic-proxies/>





## Creating Virtual Proxy

- We can create virtual proxies of anything
  - Here we replace the handcrafted proxy with dynamic
    - Less code, less chance of bugs

```
CustomMap<String, Integer> map =  
    Proxies.virtualProxy(CustomMap.class, CustomHashMap::new);  
System.out.println("Virtual Map created");  
map.put("one", 1); // creating CustomHashMap as side effect  
map.put("life", 42);  
System.out.println("map.get(\"life\") = " + map.get("life"));  
System.out.println("map.size() = " + map.size());  
System.out.println("clearing map");  
map.clear();  
System.out.println("map.size() = " + map.size());
```

```
Virtual Map created  
CustomHashMap constructed  
map.get("life") = 42  
map.size() = 2  
clearing map  
map.size() = 0
```



# Dynamic Proxy Restrictions





## Interfaces Only

- **Dynamic proxies cannot extend classes**
  - All proxies are subclasses of `java.lang.reflect.Proxy`
    - No multiple inheritance in Java
  - Might need to use tools like **CGLib** or **ByteBuddy**

# UndeclaredThrowableException

- **InvocationHandler.invoke() throws Throwable**
  - However, we should only throw declared exceptions
    - **Error and RuntimeException always allowed**

```
Runnable job = Proxies.castProxy(
    Runnable.class,
    (proxy, method, params) -> {
        // will be wrapped with an UndeclaredThrowableException
        throw new IOException("bad exception");
    });
job.run();
```

```
Exception in "main" java.lang.reflect.UndeclaredThrowableException at
com.sun.proxy.$Proxy0.run(Unknown Source)
at UndeclaredExceptionThrown.main()
Caused by: java.io.IOException: bad exception
at UndeclaredExceptionThrown.lambda$main$0() ... 2 more
```



# Return Types Have to be Correct

```
public interface FooBar {  
    void foo();  
    boolean bar();  
    int baz();  
}
```

```
public class FooBarInvocationHandler implements InvocationHandler {  
    public Object invoke(Object proxy, Method method,  
                        Object[] args) throws Throwable {  
        switch (method.getName()) {  
            case "foo": return true; // ignored  
            case "bar": return 42; // ClassCastException  
            case "baz": return null; // NullPointerException  
            default: throw new UnsupportedOperationException();  
        }  
    }  
}
```

# Naming Mysteries

```
public class ProxyNaming {
    public interface PublicNotExported {
        void open();
    }
    interface Hidden {
        void mystery();
    }
    public static void main(String... args) {
        show(BaseComponent.class); // exported from module
        show(PublicNotExported.class);
        show(Hidden.class);
    }
    private static void show(Class<?>... intf) {
        System.out.println(Proxy.newProxyInstance(
            intf[0].getClassLoader(), intf,
            (p, m, a) -> null).getClass());
    }
}
```

```
class com.sun.proxy.$Proxy0
class com.sun.proxy.jdk.proxy1.$Proxy1
class eu.javaspecialists.books.dynamicproxies.ch03.gotchass.$Proxy2
```



## Deeper Call Stacks

- **Call stacks have references to \$Proxy0**
  - Can increase each call by an additional 5 layers
  - IntelliJ IDEA folds this information away

```
micProxy$Factorial.invoke(RecursiveDynamicProxy.java:46) <5 internal calls>  
micProxy$Factorial.invoke(RecursiveDynamicProxy.java:49) <5 internal calls>  
micProxy$Factorial.invoke(RecursiveDynamicProxy.java:49) <5 internal calls>  
micProxy$Factorial.invoke(RecursiveDynamicProxy.java:49) <5 internal calls>
```

- **Actual stack trace contains all the gory details**

```
at RecursiveDynamicProxy$Factorial.invoke()  
at com.sun.proxy.$Proxy0.apply(Unknown Source)  
at java.base/NativeMethodAccessorImpl.invoke0()  
at java.base/NativeMethodAccessorImpl.invoke()  
at java.base/DelegatingMethodAccessorImpl.invoke()  
at java.base/Method.invoke()  
at RecursiveDynamicProxy$Factorial.invoke()
```

# Shared Proxy Classes

- **Java tries to minimize dynamic proxy classes**
  - **When we call `Proxy.newProxyInstance()` it checks**
    - **Have we had the same interfaces (in that order)?**
    - **And for the same class loader?**
  - **If it has seen it before, it returns a cached class**
    - **This is stored inside weak references to prevent memory leaks**



# Performance



## Performance

- **Dynamic proxies used in infrastructure code**
  - Some methods called billions of times
- **Calling methods on dynamic proxies may be slower**
  - Primitive return types and parameters might be boxed
  - Parameters need to be wrapped with `Object[]`
    - Escape analysis can eliminate the array if it does not escape from `invoke()`
  - Method suffers from amnesia and checks our permission on every call



## Model for Benchmark using JMH

```
public interface Worker {
    long increment();
    void consumeCPU();
}

public class RealWorker implements Worker {
    private long counter = 0;

    public long increment() { return counter++; }
    public void consumeCPU() { Blackhole.consumeCPU(2); }
}

public class ProxyWorker implements Worker {
    private final RealWorker worker = new RealWorker();

    public long increment() { return worker.increment(); }
    public void consumeCPU() { worker.consumeCPU(); }
}
```

# increment() and consumeCPU()

- **We use five ways of calling these methods**
  - **directCall**
  - **staticProxy**
  - **dynamicProxyThenDirectCall**
    - **Avoids cost of reflective method calls, but might need to box return value**
  - **dynamicProxyThenReflectiveCall**
    - **Delegates all calls to the RealWorker using reflection**
    - **Method call might be turbo-boosted by turning off accessibility checks**



# Benchmark increment() Results

## ● Analysis of results

- `dynamicProxyDirectCall` 1.4 ns slower in Java 14
- `dynamicProxyReflectiveCall` is another 3.1 ns slower and allocates 24 bytes
- Without our turbo boost, it is another 1.4 ns slower

Benchmark	ns/op ± Error
<code>directCall</code>	1.957 ± 0.004
<code>staticProxy</code>	2.369 ± 0.018
<code>dynamicProxyDirectCall</code>	3.726 ± 0.016
<code>dynamicProxyReflectiveCall (turbo)</code>	6.853 ± 0.201
<code>dynamicProxyReflectiveCall (no turbo)</code>	8.238 ± 0.282

# Benchmark consumeCPU() Results

- **Analysis of results**

- **dynamicProxyDirectCall 0.7 ns slower in Java 14**
- **dynamicProxyReflectiveCall is another 0.8 ns slower**
- **Without our turbo boost, it is another 1.7 ns slower**

Benchmark	ns/op ± Error
<b>directCall</b>	<b>3.270 ± 0.028</b>
<b>staticProxy</b>	<b>3.788 ± 0.018</b>
<b>dynamicProxyDirectCall</b>	<b>4.461 ± 0.020</b>
<b>dynamicProxyReflectiveCall (turbo)</b>	<b>5.231 ± 0.053</b>
<b>dynamicProxyReflectiveCall (no turbo)</b>	<b>6.960 ± 0.409</b>



# Summary of Benchmark Results

- **Method call overhead for our experiments**
  - 4.5 nanoseconds for increment()
  - 1.4 nanoseconds for consumeCPU()
- **In a typical business application, such overheads are negligible**
  - Unless called in performance sensitive code

# 3: Related Patterns





## 3: Related Patterns

- **Proxy has a similar structure to**
  - **Decorator / Filter**
  - **Composite**

# Filtering for Immutability





## Filtering for Immutability

- **Java collections are sometimes unmodifiable**
  - Might throw `UnsupportedOperationException`
- **Alternative is to split the interfaces into immutable and mutable**

```
public interface ImmutableIterable<E> {  
    void forEach(Consumer<? super E> action);  
    Spliterator<E> spliterator();  
}
```

```
public interface MutableIterable<E> extends ImmutableIterable<E> {  
    Iterator<E> iterator();  
}
```

# ImmutableCollection

- **Does not contain any mutating methods**

```
public interface ImmutableCollection<E>
    extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Object[] toArray();
    <T> T[] toArray(T[] a);
    <T> T[] toArray(IntFunction<T[]> generator);
    boolean containsAll(Collection<?> c);
    Stream<E> stream();
    Stream<E> parallelStream();

    // to try out default methods
    default void printAll() {
        forEach(System.out::println);
    }
}
```



## HandcodedFilter

- We only offer methods that do not mutate

```
public class HandcodedFilter<E>
    implements ImmutableCollection<E> {
    private final Collection<E> c;

    public HandcodedFilter(Collection<E> c) { this.c = c; }

    public int size() { return c.size(); }
    public boolean isEmpty() { return c.isEmpty(); }
    public boolean contains(Object o) { return c.contains(o); }
    public Object[] toArray() { return c.toArray(); }
    // etc.
    // No mutable methods are offered to client
}
```

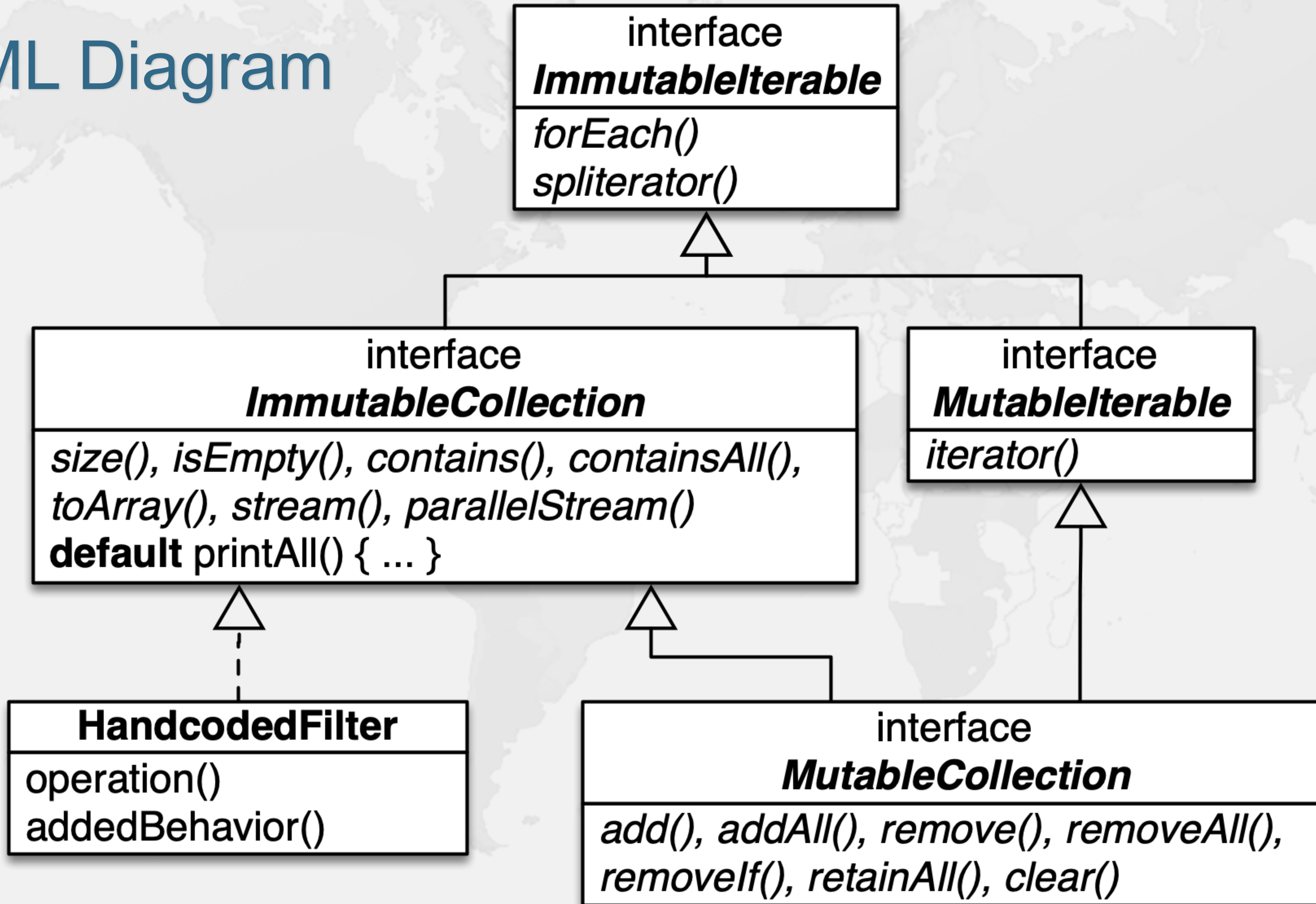
# MutableCollection

- **Allows us to modify the collection**

```
public interface MutableCollection<E>
    extends ImmutableCollection<E>, MutableIterable<E> {
    boolean add(E e);
    boolean remove(Object o);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean removeIf(Predicate<? super E> filter);
    boolean retainAll(Collection<?> c);
    void clear();
}
```



## UML Diagram



## HandcodedFilter

- We only offer methods that do not mutate

```
ImmutableCollection<String> names =  
    new HandcodedFilter<>(Arrays.asList("Peter", "Paul", "Mary"));  
// names.remove("Peter"); // does not compile  
System.out.println(names);  
System.out.println("Is Mary in? " + names.contains("Mary"));  
System.out.println("Class: " + names.getClass().getSimpleName());  
names.printAll();
```

**Oopsie!**



```
eu.javaspecialists.books.dynamicproxies.ch04.immu  
tablecollection.HandcodedFilter@2d38eb89  
Is Mary in? true  
Class: HandcodedFilter  
Peter  
Paul  
Mary
```



## FilterHandler

- **Easier than writing decorators by hand**
- **Our rules will be**
  - **We will create our filter from an interface and an object**
  - **We will map each public method from the filter to the object**
    - **To keep it simple, we will only consider the name of the method and parameter types. The return type must match, or we will reject that method**
  - **We need to include support for default interface methods**
  - **If the object does not have all the methods in the filter, constructing the dynamic filter should fail**
- **Method matching should be as fast as possible**

## FilterHandler

```
public final class FilterHandler
    implements InvocationHandler {
    private final ChainedInvocationHandler chain;

    public FilterHandler(Class<?> filter, Object component) {
        VTable vt = VTables.newVTable(component.getClass(), filter);
        VTable defaultVT = VTables.newDefaultMethodVTable(filter);

        chain = new VTableHandler(component, vt,
            new VTableDefaultMethodsHandler(defaultVT, null));
        chain.checkAllMethodsAreHandled(filter);
    }

    public Object invoke(
        Object proxy, Method method, Object[] args) throws Throwable {
        return chain.invoke(proxy, method, args);
    }
}
```



## Proxies Facade Gets filter() Method

- **We can filter components with an interface**
  - The component must have matching methods
    - Unless they are default methods in the interface
  - The component does not need to implement the filter interface

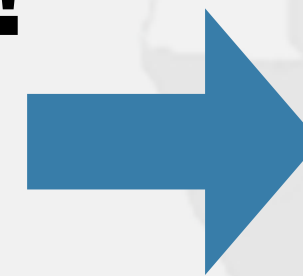
```
public static <F> F filter(Class<? super F> filter,
                          Object component) {
    Objects.requireNonNull(component, "component==null");
    return castProxy(filter, new FilterHandler(filter, component));
}
```

## Dynamic Filter

- We can filter away the mutating methods

```
ImmutableCollection<String> names =  
    Proxies.filter(ImmutableCollection.class,  
        Arrays.asList("Peter", "Paul", "Mary"));  
// names.remove("Peter"); // does not compile  
System.out.println(names);  
System.out.println("Is Mary in? " + names.contains("Mary"));  
System.out.println("Class: " + names.getClass().getSimpleName());  
names.printAll();
```

Yeah!



```
[Peter, Paul, Mary]  
Is Mary in? true  
Class: $Proxy0  
Peter  
Paul  
Mary
```



## What Happened?

- **VTable matches methods**
  - Details left as homework
- **ChainedInvocationHandler uses chain of responsibility pattern**
- **We need *deep reflection* to call default interface methods**
  - Dynamic proxies automatically implement them
    - No easy way to call the super-interface's default interface methods
  - We need to *open* whichever JPMS module we want to call default methods on
    - e.g. `--add-opens java.base/java.util=eu.javaspecialists.books.dynamicproxies` if the interface is in the `java.util` package



# Dynamic Composite





## Dynamic Composite

- **The Java Specialists' Newsletter**
  - Started as 80 contacts in a Microsoft Outlook list
  - Once it grew to maximum size, we created lists of lists
- **Model**
  - A contact is either an individual subscriber or a distribution list
    - Can be nested

## Contact and Person

```
public interface Contact {
    default boolean add(Contact c) {return false;}
    default boolean remove(Contact c) {return false;}
    void sendMail(String body);
    int count();
}

public class Person implements Contact {
    private final String email;
    public Person(String email) { this.email = email; }
    public void sendMail(String body) {
        // connecting to JavaMail and off it goes ...
        System.out.println("Sending " + body + " to " + email);
    }
    public int count() {
        return 1;
    }
}
```



## DistributionList

- Other composites would look similar

```
import java.util.*;

public class DistributionList implements Contact {
    private final Collection<Contact> contacts = new ArrayList<>();

    public boolean add(Contact c) { return contacts.add(c); }
    public boolean remove(Contact c) { return contacts.remove(c); }

    public void sendMail(String body) {
        contacts.forEach(contact -> contact.sendMail(body));
    }
    public int count() {
        return contacts.stream().mapToInt(Contact::count).sum();
    }
}
```

## Let's Build a Mailing List

```
Contact tjsn = new DistributionList();
System.out.println(tjsn.count());
tjsn.add(new Person("john@aol.com"));
tjsn.sendMail("Issue 1");
System.out.println(tjsn.count());
```

```
Contact students = new DistributionList();
students.add(new Person("peter@absa.co.za"));
students.add(new Person("mzani@absa.co.za"));
tjsn.add(students);
```

```
tjsn.sendMail("Issue 2");
System.out.println(tjsn.count());
```

```
0
Sending Issue 1 to john@aol.com
1
Sending Issue 2 to john@aol.com
Sending Issue 2 to peter@absa.co.za
Sending Issue 2 to mzani@absa.co.za
3
```



## BaseComponent<T>

- Represents any composite pattern hierarchy
  - Generic type parameter T is the extension interface
  - The two default methods both return false
    - Typical behaviour for a leaf component

```
public interface BaseComponent<T> {  
    default boolean add(T t) { return false; }  
    default boolean remove(T t) { return false; }  
}
```

## Contact extends BaseComponent

- Our Contact inherits add() and remove()

```
public interface Contact extends BaseComponent<Contact> {  
    void sendMail(String body);  
    int count();  
}
```

- Person remains exactly the same



## Dynamic Composite Contact

```
var reducers = Map.of(  
    new MethodKey(Contact.class, "count"),  
    new Reducer(0, (r1, r2) -> (int)r1 + (int)r2)  
);
```

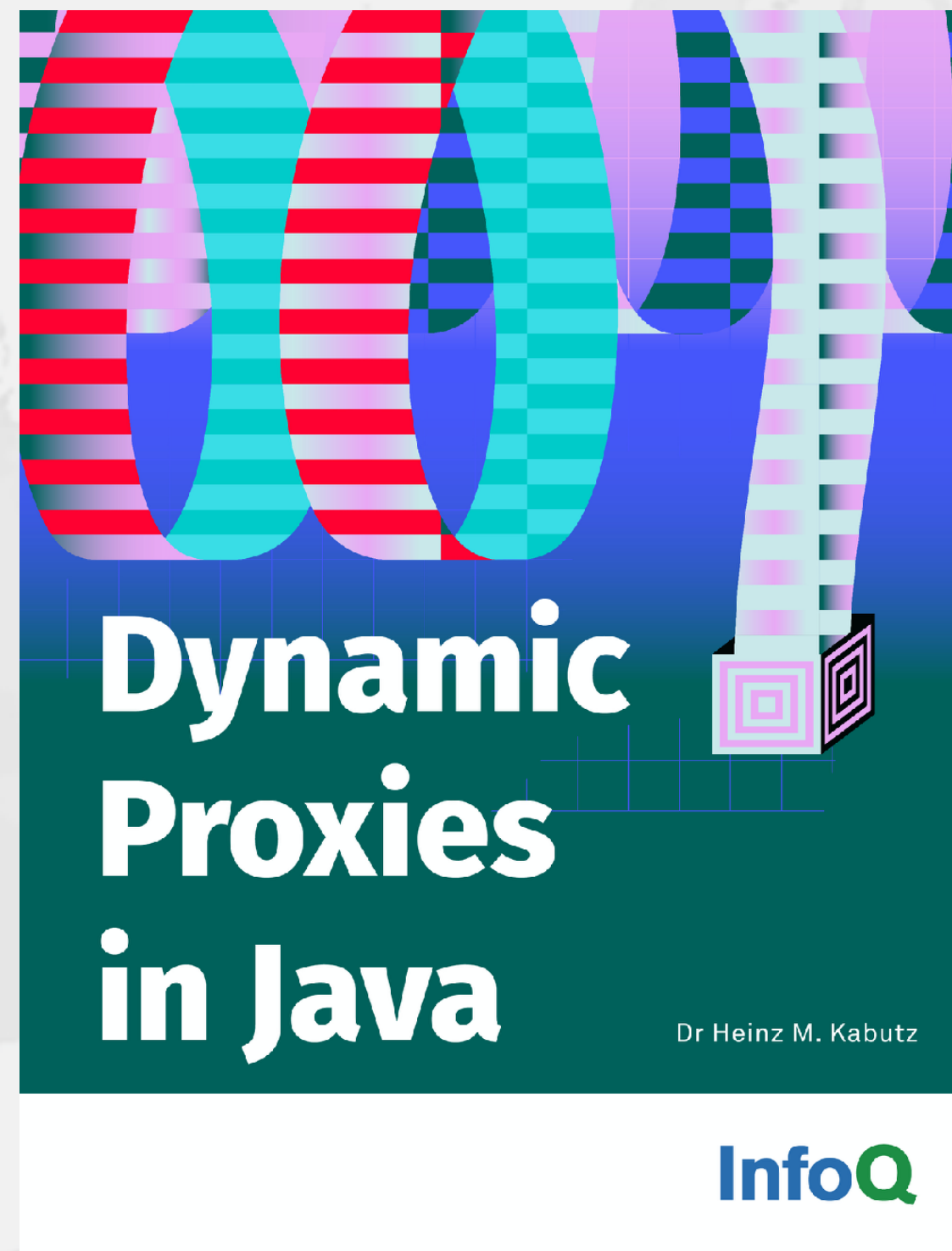
```
Contact tjsn = Proxies.compose(Contact.class, reducers);  
System.out.println(tjsn.count());  
tjsn.add(new Person("john@aol.com"));  
tjsn.sendMail("Issue 1");  
System.out.println(tjsn.count());
```

```
Contact students = Proxies.compose(Contact.class, reducers);  
students.add(new Person("peter@absa.co.za"));  
students.add(new Person("mzani@absa.co.za"));  
tjsn.add(students);  
tjsn.sendMail("Issue 2");  
System.out.println(tjsn.count());
```

```
0  
Sending Issue 1 to john@aol.com  
1  
Sending Issue 2 to john@aol.com  
Sending Issue 2 to peter@absa.co.za  
Sending Issue 2 to mzani@absa.co.za  
3
```

## Questions?

- Don't forget gift: [tinyurl.com/JAX-Online-2020](https://tinyurl.com/JAX-Online-2020)
- Please say "hello" : [heinz@javaspecialists.eu](mailto:heinz@javaspecialists.eu)



- Free download from

– <https://www.infoq.com/minibooks/java-dynamic-proxies/>