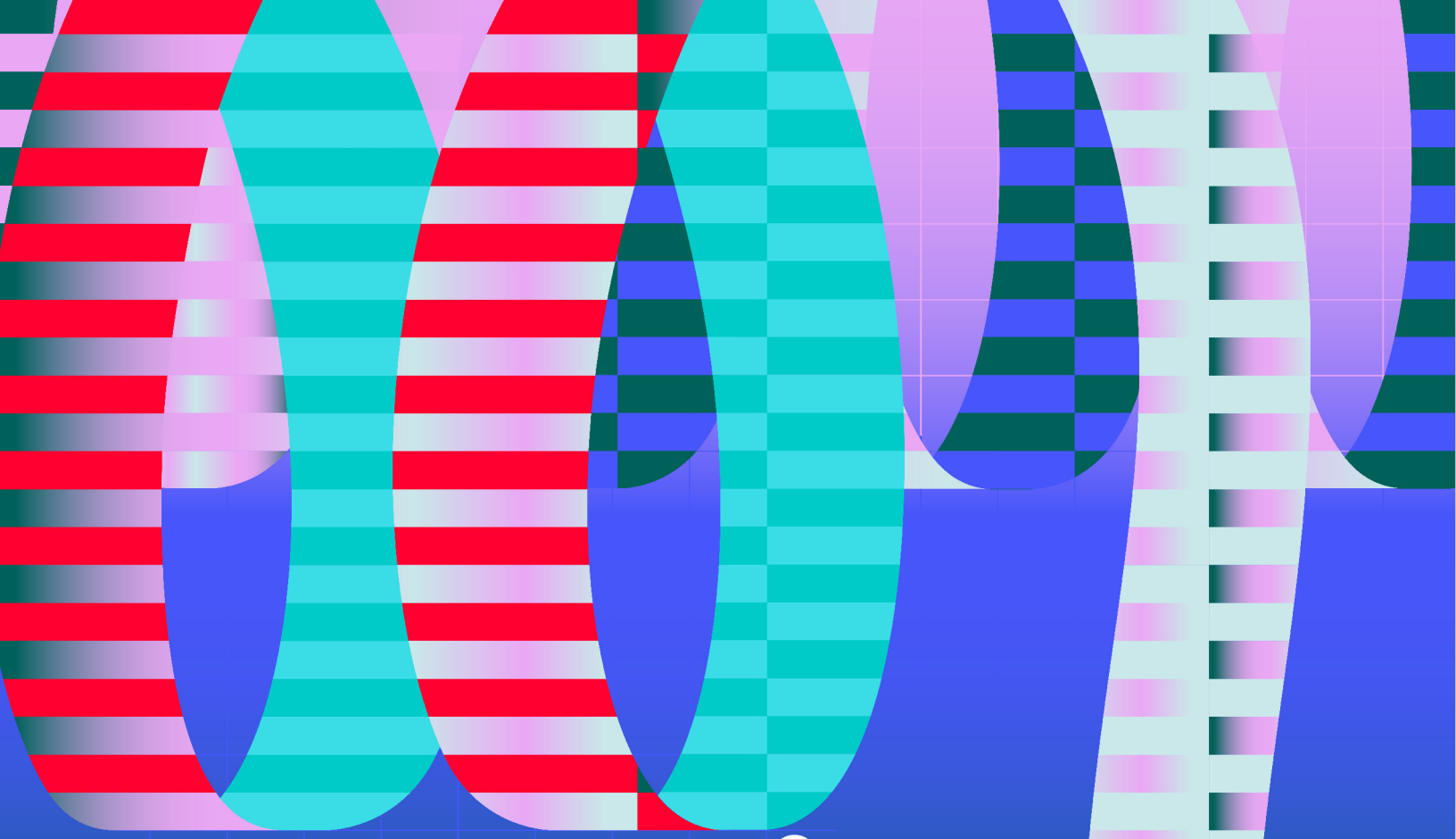


Dynamic Proxies in Java

2nd edition

Dr Heinz M. Kabutz
With foreword by Brian Goetz

InfoQ

Dynamic Proxies in Java: Second Edition

Published by C4Media, publisher of InfoQ.com.

Production Editor: Ana Ciobotaru
Copy Editor: Lawrence Nyveen
Cover and Interior Design: Dragos Balasoiu
Cover Art work: Ana Kova

Library of Congress Cataloguing-in-Publication Data:
ISBN: 978-1-71663-008-8

Table of Contents

Dedication

This book is dedicated to all those poor souls who had to maintain our code. We are *truly* sorry.

Acknowledgements

I thought an early draft of this book was pretty good. Then I opened it up for review. Instead of asking only a handful of technical experts, I invited the readers of my Java Specialists' Newsletter to take a look. Over 500 kind souls volunteered to help. They tore into the book and gave me over 1,000 incredibly valuable comments. I applied almost all of their suggestions and this book is now so much better than it otherwise would have been. This is not **my** book. This is **our** book, a group effort by our amazing and generous Java Specialists' Community.

By far the most valuable comments came from my friends Gail Anderson, John Stephen Green, and Joseph Ottinger. Gail and Joseph jumped into the book with vigor and changed almost all the sentences to make them better. Any bad text that remains was added after they finished their review. John painstakingly ran all the code samples and was one of the only readers to figure out how the `VTable` works.

Our top 10 super reviewers contributed at least 25 comments each:

- Gail Anderson
- Alasdair Collinson
- Joseph Ottinger
- Vlad Lazurenko
- John Stephen Green
- Charles A. Sharp
- Cor Takken
- Chris Menard
- Heinz Huber
- Ronak Patel

Special thanks also to Graeme Rocher, creator of Grails Framework and Micronaut, for reading an early draft of the book and providing valuable feedback.

Another special call out goes to Brian Goetz, long-time friend and best-selling author, for delaying the release of this book with his fantastic suggestions that I simply had to add.

Kirk Pepperdine has been my mentor re Java performance for over a decade and was very helpful in looking over some of the benchmark results with me.

I modularized the samples in the book using the Java Platform Module System (JPMS). Simone Bordet and Robert Scholte kindly fixed the project structure to make it work with Maven.

Furthermore, I would like to thank the following reviewers for their contributions:

Manu Anand, Fabrice Aupert, Tymur Berezhnoi, Cristian Borta, Borisas Bursteinas, Robin Coe, Richard Cumberland, Thomas Darimont, Ian Darwin, Vianney Dervaux, Vlad Dinu, Wirianto Djunaidi, Mark Doppelfeld, Alex Dunwell, Dirk Estievenart, Elmar Fasel, Ondřej Fischer, Remi Forax, Kyc Fumbi, Asad Ganiev, M. Eduardo Giolo, German Gonzalez-Morris, William Goodwin, Mithun Gooty, Emmanuel Gosse, Alex Gout, Ido Green, Emmanuel Hugonnet, Enoobong Ibanga, Hans-Peter Iten, Ron Jacobs, Ulf Jährig, Harinath K., Piotr Kania, Dave Kant, Uladzimir Karabeinik, Frank Karlstrøm, Maksim Kashynski, Paul Keogh, Chandana Kithalagama, Stephen Kitt, Julian Koch, John Kostaras, Gregor Koukkoullis, Eugene Kravchenko, Michael Kuhlmann, Sanjeev Kumar, Abhi Kutty, Tim Lavers, Naga Linga S. Ala, Sina Madani, Florian Maier, Abhishek Manocha, Krzysztof Mazur, John Mermigkis, Daniel Migowski, Marijan Mihaljev, Pim Moerenhout, Milan Mrdjen, Jeff Mutonho, Maurice Naftalin, Bekezela Ncube, Andrzej Paluch, Karan Pant, Markus Peetzen, Federico Peralta Schaffner, Marcos Pinto, Morgan Pittkin, Nick Pratt, Eugene Rabii, A. N. M. Bazlur Rahman, Priya Rammohan, Alexander Rathai, Thomas Révész, Jonathan Rosenne, Leonid Rozenblyum, Krishna S., Igal Sapir, Daniel Schwering, Andrea Selva, Karthick Selvaraj, Adeel Shahzad, Sebastian Sickelmann, Martin Stefanko, Jens-Hagen Syrbe, Hasan Tahan, Simon Tenbeitel, Aleksandr Teterin, Olli Tietäväinen, Luke Torjussen, Henri Tremblay, Jesper Udby, Karel Van Der Walt, Szymon Walkowski, Torsten Werner, Alan Williamson, Kelli Wiseth, and Chris Wraith.

Thank you **so much**. This book is so much better as a result of your kind willingness to help.

Sven Ruppert and I published a book on the same topic in 2015, albeit in German. Thank you Sven for introducing me to the fascinating world of being an author.

A big thank you to Lawrence Nyveen, our editor at InfoQ, for applying a good deal of polish to our creation.

Before we get to the technical bits, a short personal story. Holding their warm cups of tea, Tippy and Mike Scriven approached me after church one day in September '19. Tippy looked like she had something urgent to tell me, but hesitated. Mike jumped in with "God told Tippy that you should write a book."

(This did not come as a complete surprise. My teenage daughter had asked me a few weeks prior if I had any regrets. Regrets? Oh yeah, too many. Not writing my first book at age 30 was at the top of that list.)

Thanks to Tippy's encouragement, I began writing the next day. My plan was to create a gigantic volume of my newsletters. It was a daunting task, and I realized that a better approach was to first try small, hence this mini-book. May this be the first of many. Thank you Tippy and Mike for giving me that push I needed.

Foreword

I was excited when Heinz told me he was working on a mini-book on dynamic proxies in Java. Most Java developers have *heard* of dynamic proxies and have a vague idea of what they are, but haven't used them in anger — and would be surprised to learn all of the ways in which they can be used effectively. Relatively few books even cover the topic, and when they do, it is limited to a chapter on the API and mechanics but fail to really cover their motivation or myriad applications. Which is a perfect scope for a mini-book like this one.

Most people would describe Java as a "static" language (and would surely be surprised to find out that, when Java was new, it was widely described as a dynamic language). This is because many tend to associate the terms *static* and *dynamic* in relation to programming languages only with one aspect — typing. Static vs dynamic describes the *timing* of a multitude of activities — including typing — static typing means "type checking done at compile time", and dynamic typing means "type checking done at run time." And Java does both! Similarly, Java is dynamically (and statically) compiled, dynamically linked, dynamically verified, provides dynamic class loading and introspection (reflection), etc.

Behind the one-bit popular perception as a "static" language, Java is in fact quite dynamic — much of the interesting work happens at run time. And dynamic proxies are an important, though lesser-known, part of that dynamism story. Dynamic proxies power many of the frameworks we use, moving work to run time that might ordinarily be done at coding time by hand, or at build time by code generators — and in the process, usually reducing the amount of code that needs to be maintained.

Dynamic proxies offer us a form of *metaprogramming* in Java: writing code that is about modifying or manipulating the behavior of other code, rather than about a specific business task. And for some tasks, this is exactly the tool we need to replace large swathes of brittle boilerplate adaptation code with a few lines that captures how to convert some existing code into exhibiting the shape we want.

Let Heinz be your guide on how to apply dynamic proxies to a range of problems that we encounter every day. It is full of illustrative examples likely to elicit an "I would have never thought of that" from even the most grizzled Java veterans.

Brian Goetz
Java Language Architect, Oracle

Preface

It is the year 2000. Our industry survived the Y2K bug. A “smartphone” can also tell the time. Java is a language without generics, lambdas, or type inference. Deploying our EAR files takes ages. EAR files? Yes, the Spring Framework is still years away from release. Real programmers use RMI and CORBA.

Tools generate buckets of infrastructure code. For example, Java Remote Method Invocation (RMI) uses the `rmic` tool to generate stub and skeleton classes to support remote proxies. (Remarkably, `rmic` lives on in the JDK/bin directory in modern Java distributions.)

In May 2000, Sun Microsystems ships Java 1.3 — and with it, dynamic proxies. Instead of a separate build step, new classes are created on the fly by specifying which interfaces they should implement. All method calls on these dynamic classes are routed through a single `InvocationHandler`. Infrastructure code uses dynamic proxies to create stub and skeleton classes in memory.

Big win

Dynamic proxies make code easier to maintain. Instead of a large number of handcrafted classes, we write a single dynamic proxy. It is the “don’t repeat yourself” principle on steroids.

For example, one company managed to replace 600,000 generated code statements with a single dynamic proxy. They had written a magic tool to migrate their proprietary language to Java. Their business system had hundreds of entities. Their tool converted each entity to several Java classes. However, after this initial generation, programmers had to maintain the code by hand. Maintenance was laborious and error-prone. A single dynamic proxy replaced all that. Less code means easier maintenance.

Infrastructure code

Most of us have used dynamic proxies, often without realizing it. A lot of frameworks and tools are built on dynamic proxies. Even the humble annotation is implemented with dynamic proxies. These examples are a small sample of where we might find them in use.

Build tools like Gradle use proxies for models, tasks, actions, and listeners. Examples include `BuildOperationListener`, `StandardOutputListener`, and `ProgressListener`. Maven uses them for plugin management through the `MavenPluginManager` and `CircularDependencyProxy`.

Application containers like JBoss and Spring Framework use proxies for dependency injection. Some examples are `@Inject`, `@Qualifier`, `@Autowired`, and `@Bean`. Scoping and lifecycle management also use dynamic proxies.

Persistence management with JPA within these containers uses proxies — for example, `@Entity`, `@Id`, `@GeneratedValue`, `@Table`, and `@JoinColumn`. In Spring, we can derive a custom repository by extending the `Repository` interface. In the background, a multi-interface dynamic decorating proxy takes care of the details.

If we choose Hibernate for ORM, then proxies surround `HibernateEntityManagerFactory`, `HibernateEntityManager`, and `EntityManagerProxy`. Spring's `ConnectionProxy` is a dynamic proxy of `java.sql.Connection`.

Again in Spring, a dynamic proxy for `InitBinder` binds HTTP form fields with a data model. Dynamic proxies are used under the hood by many of the frameworks we use every day: Spring Framework, WildFly, NetBeans Platform, Apache Tomcat, and OpenJDK, among many others.

But wait...

Dynamic proxies are not always the best tool to use. Although highly convenient, method calls on a dynamic proxy suffer a small increase in performance overhead compared to direct method calls. In this book, we will learn how to minimize the performance impact of using dynamic proxies.

What will we learn?

There are occasions when dynamic proxies are the perfect fit for a particular problem. By studying them in depth and seeing them in a variety of scenarios, we will learn to spot places where it makes sense to use them. We will learn to recognize their strengths and weaknesses.

This makes the debugging of systems that use dynamic proxies easier.

TL;DR?

Even a mini-book like this can take days to read and fully understand. To make this material more accessible, I have created a self-study course with detailed explanations of each building block. The course has exercises for each chapter, together with model solutions.

Java version

We compiled the code for this book with Java 11 LTS. Where it makes code clearer, we use the Java 10 `var` keyword for local variables.

While we do use shallow reflection in this book, we try to avoid using deep reflection. Mark Reinhold defined these terms on the OpenJDK JPMS (JSR 376) Expert Group mailing list.

- Shallow reflective access is access at runtime via the Core Reflection API (`java.lang.reflect`) without using `setAccessible`. Only public elements in exported packages can be accessed.
- Deep reflective access is access at run time via the Core Reflection API, using the `setAccessible` method to break into non-public elements. Any element, whether public or not, in any package, whether exported or not, can be accessed. Deep reflective access implies shallow reflective access.

Enjoy!

I hope that you will enjoy this book and that you find it both useful and intellectually stimulating. Perhaps even entertaining at times.

Try out the concepts in the book; best with a cup of hot java at your side. Play around with the concepts in your IDE. All the code samples from the book are available at https://github.com/kabutz/dynamic-proxies-samples.

Dr. Heinz M. Kabutz

CHAPTER ONE

Design-Patterns Cousins

An architect is a person who plans, designs, and reviews the construction of buildings. Their finished product is hard to change as it is, literally, poured in concrete. Software changes all the time, even many years after installation, so it is surprising just how much architect Christopher Alexander's 1977 book *A Pattern Language: Towns, Buildings, Construction* has influenced the software industry. Alexander's book is about the timeless way of building. He distils habitat patterns that have repeatedly occurred throughout history — for example, pattern number 159, called "Light on Two Sides of Every Room". We can easily visualize the pattern because of its descriptive name.

In the 1990s, the software industry took the principles that Alexander proposed and applied them to software design. The concept of object-oriented design patterns was immortalized in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These four authors are commonly referred to as the Gang of Four or simply the GoF.

A design pattern is a general solution for a common problem in computer programming. It is usable across most languages and describes the interconnections between objects in a language-agnostic way.

One of the most important identifiers of a pattern is its name. In *Timeless Way of Building*, Alexander wrote "The search for a name is a fundamental part of the process of inventing or discovering a pattern. So long as a pattern has a weak name, it means that it is not a clear concept, and you cannot tell me to make 'one'."

Design patterns similarly apply names to common coding techniques, making it easier to communicate intentions. Knowing the patterns helps us to understand existing designs more quickly. Unfortunately, the GoF pattern names are not as colorful as Alexander's. A name like "wrapper" could mean anything. We thus also rely on the *intent* to clearly identify each pattern.

In this book, we will focus primarily on four patterns: **proxy**, **adapter**, **decorator**, and **composite**. They are all structural patterns, which means that they don't describe behavior or the creation of objects, but rather how classes and objects interrelate.

Our four patterns have a similar class hierarchy, so with a bit of effort we can use dynamic proxies to implement each of them. They are so similar in appearance that we can call them "design-pattern cousins".

Proxy pattern

Intent: "Provide a surrogate or placeholder for another object to control access to it." (GoF 1994, p. 207)

In the proxy pattern, the client communicates via a substitute (the proxy), which acts a bit like a ventriloquist's doll (such as Conky in *Trailer Park Boys*).

The standard GoF structure is shown below in Figure 1. The `Client` uses the `Subject`, which can either be a `RealSubject` or a `Proxy`. The `Client` does not know which, nor care. In the standard GoF structure, the `Proxy` points to a `RealSubject`. A variation of the proxy structure is to let the `Proxy` point to a `Subject`. This allows us to cascade proxies. We will examine cascading proxies in more detail at the end of the next chapter.

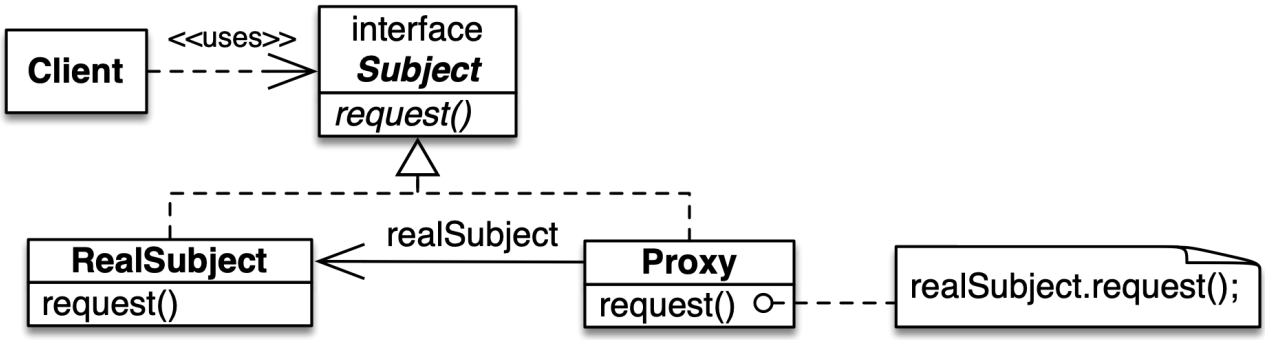


Figure 1. Structure of the proxy design pattern.

Proxy has several use cases:

- **Virtual proxy** creates expensive objects on demand. An example is the `LazyConnectionDataSourceProxy` in the Spring Framework, which lazily creates the real JDBC `DataSource` when the first `Statement` is created.
- **Remote proxy** provides a local representation of an object that sits in a different address space, for example on a remote machine. Note that the remote proxy does not have a real pointer to the remote object, but invokes the methods via RPC, RMI, Thrift, ProtoBuf, gRPC, REST, HTTP, TCP/UDP,

or a different remoting mechanism.

- **Protection proxy** controls access to the original object. This could be for access control or thread safety. An example is `Collections.synchronizedCollection()`, which returns a new `Collection` that is guarded against race conditions.

Adapter pattern

Intent: “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.” (GoF 1994, p. 139)

The adapter pattern’s goal is to make objects work together, even if they have incompatible interfaces. It is the diplomat of patterns. The adapter pattern can be implemented either as an object adapter that uses composition or as a class adapter that uses inheritance.

In Figure 2, the `Adapter` implements the `Target` interface and delegates to the class that we want to adapt, in our case `Adaptee`. Since our adaptee is an object inside our adapter, we call this an “object adapter”. The object adapter uses composition to adapt the `Adaptee`.

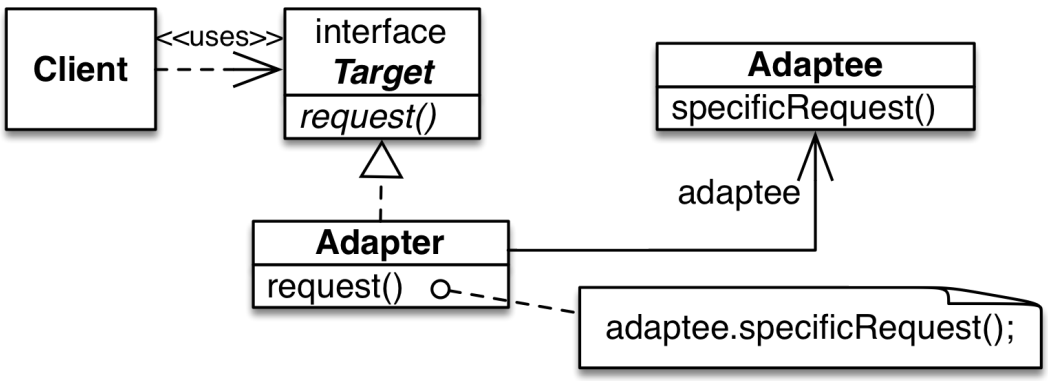


Figure 2. Structure of the object-adapter design pattern.

An example of an object adapter in Java is the `InputStreamReader`, which adapts an `InputStream` into a `Reader`.

A lesser used form of adapter occurs when our `Adapter` extends the `Adaptee` and implements the `Target` interface. Since the adaptee is a superclass of our adapter, we call this a “class adapter”, as shown in Figure 3. The class adapter uses inheritance to adapt the `Adaptee`.

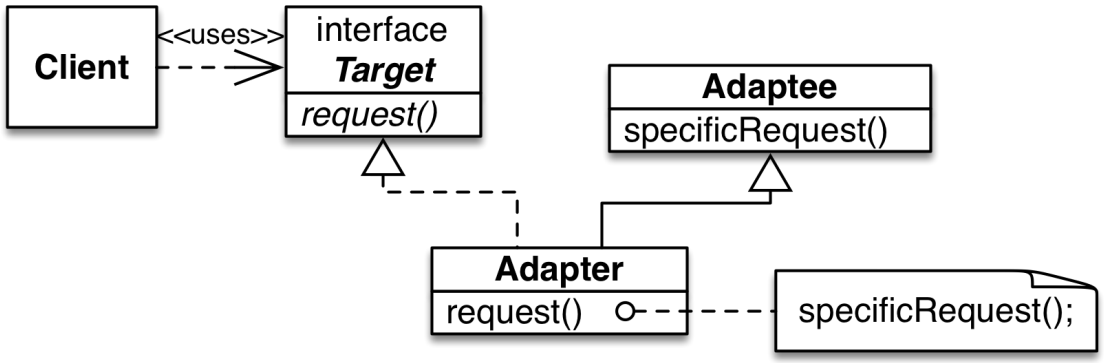


Figure 3. Structure of the class-adapter design pattern.

An advantage of the class adapter is that it makes it easier to modify parts of an adaptee. Unfortunately, the trade-off is that it becomes harder to adapt a hierarchy of objects. In a later chapter, we shall see how to get the extensibility of the class adapter and the flexibility of the object adapter using a dynamic proxy.

How is an object adapter like a proxy?

In the proxy pattern, the `RealSubject` implements the `Subject` interface. If we take away the inheritance relationship, is this still a proxy or does it become an adapter? Similarly, if in an adapter the `Adaptee` happens to implement `Target`, does the pattern become a proxy?

It depends on the intent. For example, with session beans in EJB, the implementation bean does not inherit directly from the remote interface, because the methods throw incompatible exceptions. The remote interface would throw `RemoteException`, whereas the implementation bean would throw business exceptions.

An object adapter has a similar structure to the proxy. In both cases, the client uses the top-level interfaces `Target` or `Subject`. We have implementations of this interface in `Adapter` or `Proxy`.

Decorator pattern

Intent: “Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.” (GoF 1994, p. 175)

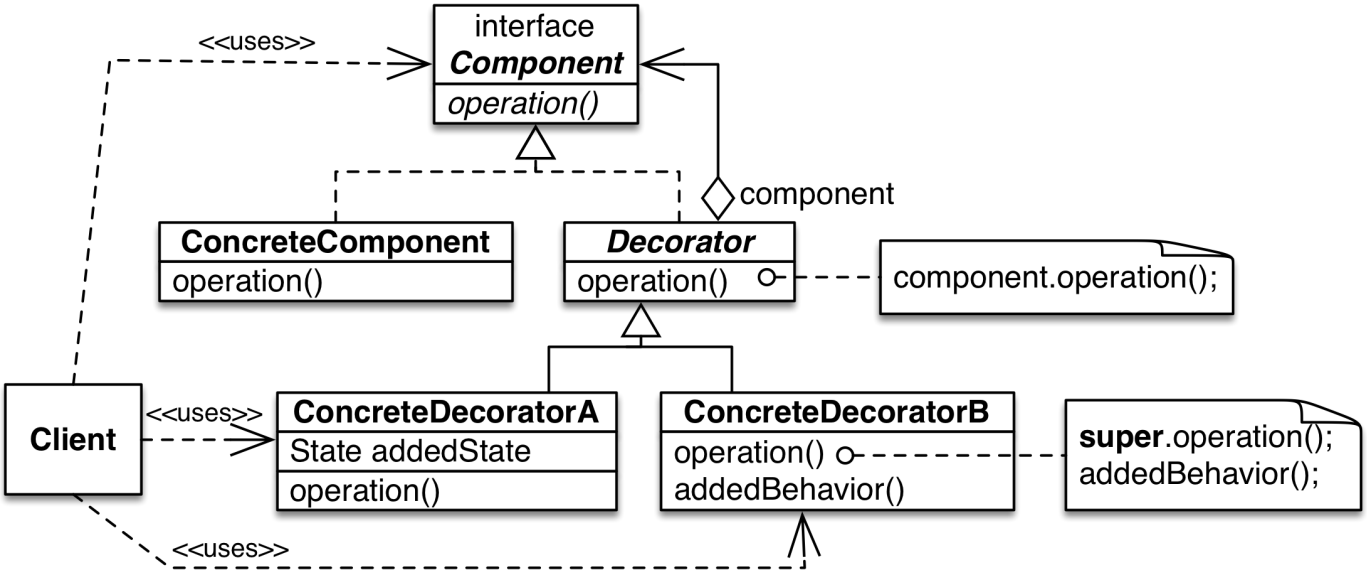


Figure 4. Structure of the decorator design pattern.

When we want to add new methods to a class, the easiest approach is to add them to a subclass. However, this may lead to clumsy class hierarchies, and risks duplication of code due to the inability to extend more than one superclass. A better approach is the decorator pattern, with which we decorate an object with a richer class and thereby "attach additional responsibilities". We can also use this pattern to remove methods that we do not want to support. This is called "filtering".

In the GoF book, both adapter and decorator are called a "wrapper". Be careful with naming. A weak name like "wrapper" is bound to cause confusion.

A well known example of a decorator is the `java.io.InputStream` and related classes, with the design copied from C++. Concrete components include `FileInputStream` and `SocketInputStream`. We also have a myriad of decorators, such as `BufferedInputStream` for adding I/O buffering, `DataInputStream` for reading primitive data in little-endian format, and `ObjectInputStream` for reading Java objects using serialization. There is even a `ProgressMonitorInputStream` that opens up a Swing progress dialog if it takes longer than two seconds to read the input.

In Listing 1.1, we create a `FileOutputStream` to the file data.bin.gz. We decorate this with `GZIPOutputStream`, `BufferedOutputStream`, and `DataOutputStream`. Each link in the chain adds functionality. We then write 10 million random integers between zero and 1,000.

Listing 1.1: Writing to a `DataOutputStream`.

```
//:~ samples/src/main/.../ch01/InputStreamDemo.java
try (var out = new DataOutputStream(
    new BufferedOutputStream(
        new GZIPOutputStream(
            new FileOutputStream(
                "data.bin.gz"))))) {
  // write ten million random ints between 0 and 1000 to
  // data.bin.gz, compressing the file with GZIP on the fly
  ThreadLocalRandom.current().ints(10_000_000, 0, 1_000)
      .forEach(i -> {
        try {
          out.writeInt(i);
        } catch (IOException e) {
          throw new UncheckedIOException(e);
        }
      });
  out.writeInt(-1); // our EOF marker
}
```

In Listing 1.2, we read from a `DataInputStream`, which in turn reads from `BufferedInputStream`, `GZIPInputStream`, and `FileInputStream`.

Listing 1.2: Reading from a `DataInputStream`.

```
//:~ samples/src/main/.../ch01/InputStreamDemo.java
try (
    // declared "fis" separately to ensure it is closed, even
    // if the file does not contain a proper GZIP header
    var fis = new FileInputStream("data.bin.gz");
    var in = new DataInputStream(
        new BufferedInputStream(
            new GZIPInputStream(fis)))) {
  long total = 0;
  int value;
  // keep reading until value == -1, our EOF marker
  while ((value = in.readInt()) != -1) {
    total += value;
  }
  // we expect total to be approximately 5 billion
  System.out.println("total = " + total);
}
```

(The reason we declared the `FileInputStream` separately is that the `GZIPInputStream` reads the file header to verify that it is indeed a Gzip file. If the header is incorrect, the constructor to `GZIPInputStream` throws an exception, in which case the `FileInputStream` will not be auto-closed if we do not declare it separately.)

How is a decorator like a proxy?

Decorator is most similar in structure to proxy. `Component` is like `Subject`, `Decorator` is like `Proxy`, and `ConcreteComponent` is like `RealSubject`. One difference is in the classes that the client typically sees. In the proxy, the client uses the interface of the `Subject`, whereas in decorator, the client may use the concrete decorator instances. Another difference is that a decorator normally has a component as constructor parameter.

Composite pattern

Intent: “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” (GoF 1994, p. 163)

Part-Whole Hierarchy?

Non-native English speakers often struggle with the term “part-whole hierarchy”. The Dictionary.com definition of part is “a piece or portion of a whole'" and of whole is "`the whole assemblage of parts or elements belonging to a thing”.

An object is either a part of a whole or it is the whole made up of parts. For example, a file is part of a directory, and a directory contains files and potentially other directories. One of the constraints of part-whole is that the object graph must by its definition be non-circular. This is not a bad constraint. Traversals over circular data structures may cause infinite loops or terminate with a `StackOverflowError`.

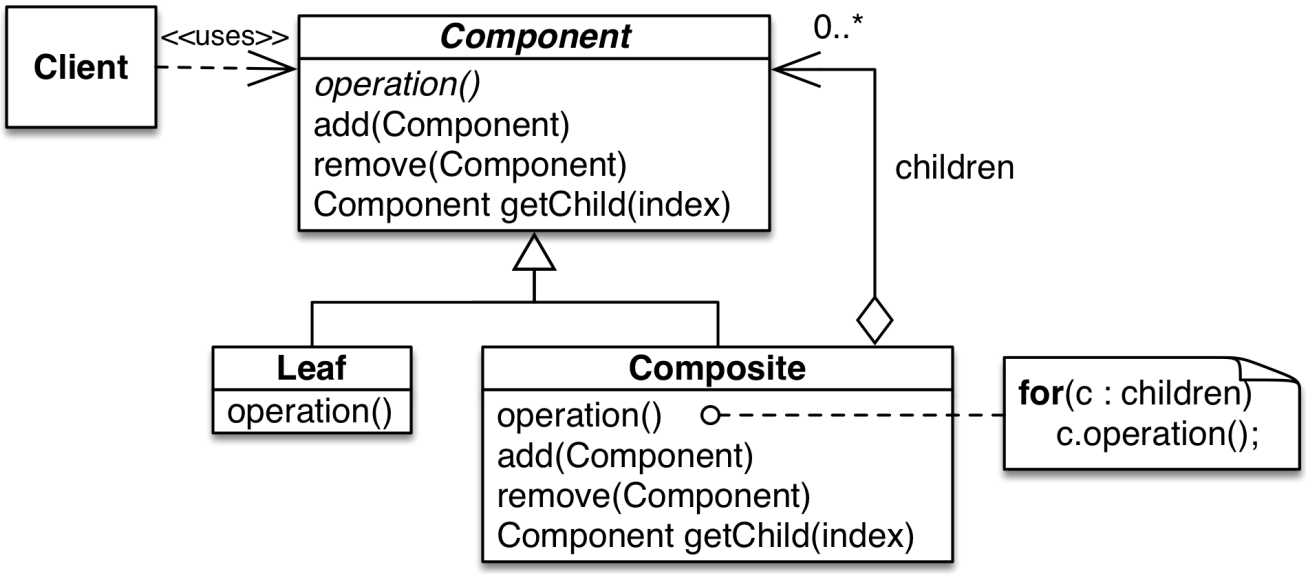


Figure 5. Structure of the composite design pattern.

The structure is again similar to proxy and object adapter. The difference is that there is a one-to-many relationship between the composite and the component. The client speaks to the top-level component class.

How is a composite like a proxy?

In both composite and proxy patterns, the client would use the top-level interfaces of `Subject` or `Component`. However, the composite pattern has a one-to-many relationship between its `Composite` and `Component`, whereas the proxy pattern has a one-to-one association between its `Proxy` and `Subject`.

Chain-of-responsibility pattern

Intent: “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.” (GoF 1994, p. 223)

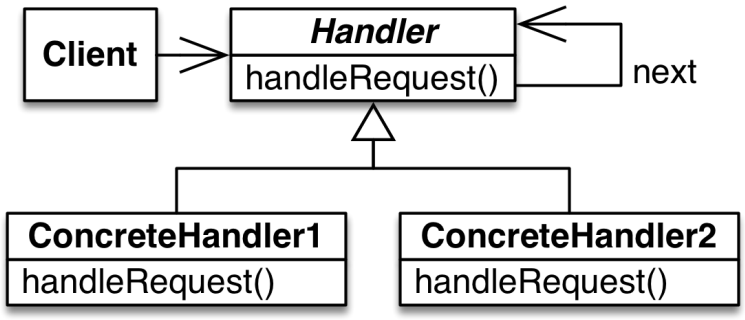


Figure 6. Structure of the chain-of-responsibility design pattern.

We use the chain-of-responsibility pattern to make our method-handling code easier to understand. This pattern is useful when we have code of the form shown in Listing 1.3.

Listing 1.3: Nested if-else-if constructs.

```
//:~ samples/src/main/.../ch01/ChainDemo.java
public void handleWithoutChain(
    String key, Consumer<String> processor) {
  var item = map1.get(key);
  if (item != null) processor.accept(item);
  else {
    item = map2.get(key);
    if (item != null) processor.accept(item);
    else {
      item = map3.get(key); // ad nauseum
    }
  }
}
```

Instead of the multiconditional if-else statement, we create an abstract `Handler` class. The default implementation of `handle()` is to pass the call down the chain to the next handler.

Listing 1.4: Chain-of-responsibility handler.

```
//:~ samples/src/main/.../ch01/Handler.java
public abstract class Handler {
  private final Handler next;
  public Handler(Handler next) {
    this.next = next;
  }
  public void handle(String key, Consumer<String> processor) {
    if (next != null) next.handle(key, processor);
  }
}
```

Our `MapHandler` contains the implementation that calls `process(item)` if the item is found in our map; else it calls `super.handle(key)`.

Listing 1.5: Concrete handler for maps.

```
//:~ samples/src/main/.../ch01/MapHandler.java
public class MapHandler extends Handler {
  private final Map<String, String> map;
  public MapHandler(Map<String, String> map, Handler next) {
    super(next);
    this.map = map;
  }
  @Override
  public void handle(String key, Consumer<String> processor) {
    String item = map.get(key);
    if (item != null) processor.accept(item);
    else super.handle(key, processor);
  }
}
```

Inside our class, we would set up the chain as we see in Listing 1.6.

Listing 1.6: Chain of handlers as field.

```
//:~ samples/src/main/.../ch01/ChainDemo.java
private final Handler chain =
    new MapHandler(map1,
        new MapHandler(map2,
            new MapHandler(map3, null)));
```

Instead of the complicated if-else-if construct from Listing 1.3, we can write:

Listing 1.7: Using the chain-of-responsibility handler.

```
//:~ samples/src/main/.../ch01/ChainDemo.java
private void handleWithChain(
    String key, Consumer<String> processor) {
  chain.handle(key, processor);
}
```

The pattern makes complicated if-else-if methods much easier to understand. One known weakness of this pattern is that a request can drop off the end of the chain if no suitable handler is found. In Chapter 4, we will learn how this pattern helps us to handle method calls. We will also see how we can prevent a request from falling off the end of the chain.

CHAPTER TWO

Handcrafted Proxies

Since our book has “proxies” in its title, we should examine the proxy pattern in a bit more detail. Proxies come in various forms and flavors. For example, we have mentioned the virtual proxy for lazy initialization, the remote proxy for giving access to objects in remote address spaces, and the protection proxy for adding a layer of security.

As a reminder, the proxy design-pattern structure from the GoF book is seen in Figure 7.

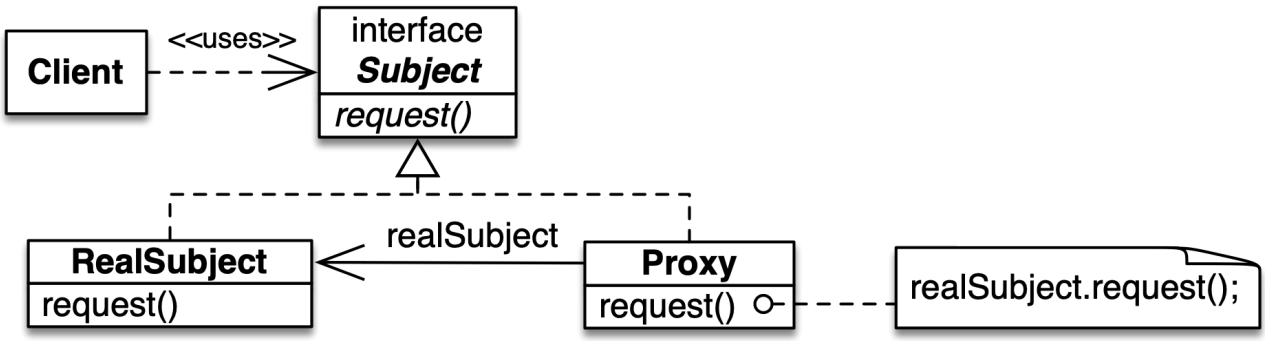


Figure 7. Proxy structure.

Lazy initialization with virtual proxy

In some situations, we may want to delay the creation of objects until they are absolutely needed. Postponing might even save us the cost of object creation altogether. Examples of such objects with a high creation cost include threads, database connections, large arrays, and files.

For example, a new `HashMap` is 48 bytes, excluding its array of entries. This array traditionally starts at length 16 and grows once it has more than 75% occupancy. An array of 16 objects occupies another 80 bytes. This means that the combined cost of a new `HashMap` comes to 128 bytes, before containing a single element.

Sizeof in Java

We can estimate object sizes in Java by adding sizes according to the formula:

- 12 bytes for the object header
- 4 bytes for each pointer
- 8 bytes for each `long` and `double` field
- 4 bytes for each `int` and `float` field
- 2 bytes for each `short` and `char` field
- 1 byte for each `byte` and `boolean` field

Furthermore, objects are allocated in 8-byte increments. See also the Java Object Layout (JOL) tool.

CustomMap

The `CustomMap` interface contains a subset of methods defined in `java.util.Map`. In the proxy design-pattern structure, `CustomMap` is an example of the `Subject` interface.

Listing 2.1: `CustomMap` *interface.*

```
//:~ samples/src/main/.../ch02/virtual/CustomMap.java
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);
  // etc.
}
```

The class structure of our `CustomMap` can be seen in Figure 8.

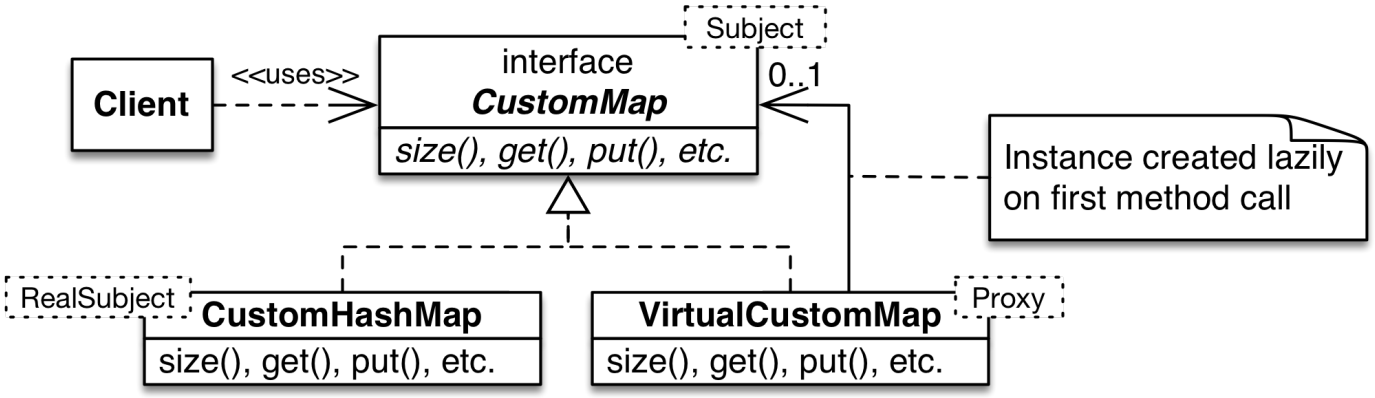


Figure 8. `CustomMap` *structure.*

CustomHashMap

`CustomHashMap` implements `CustomMap`. Because we don't want to reinvent the wheel, we delegate all methods to a `java.util.HashMap`. In the proxy design-patterns structure, `CustomHashMap` is an example of the `RealSubject` class.

Listing 2.2: `CustomHashMap` *class.*

```
//:~ samples/src/main/.../ch02/virtual/CustomHashMap.java
public class CustomHashMap<K, V> implements CustomMap<K, V> {
  private final Map<K, V> map = new HashMap<>();
  {
    System.out.println("CustomHashMap constructed");
  }
  @Override
  public int size() {
    return map.size();
  }
  @Override
  public V get(Object key) {
    return map.get(key);
  }
  @Override
  public V put(K key, V value) {
    return map.put(key, value);
  }
  @Override
  public V remove(Object key) {
    return map.remove(key);
  }
  @Override
  public void clear() {
    map.clear();
  }
  @Override
  public void forEach(BiConsumer<? super K, ? super V> action) {
    map.forEach(action);
  }
  @Override
  public String toString() {
    return map.toString();
  }
}
```

Wow, look at all that repetition. Wouldn't it be nice to delegate all of the methods in `CustomMap` to a `java.util.Map` of our choice? We will learn how to do this in chapter 3.

`VirtualCustomMap`

Lastly, we create our virtual proxy, `VirtualCustomMap`. When we construct the virtual proxy, we specify a `Supplier<CustomMap<K, V>>`. The first time any method is called on `CustomMap`, we can create the real map using the supplier and delegate operations to this real map. `VirtualCustomMap` is an example of the `Proxy` in the proxy structure.

Listing 2.3: `VirtualCustomMap` *proxy class.*

```
//:~ samples/src/main/.../ch02/virtual/VirtualCustomMap.java
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>> mapSupl) {
    this.mapSupplier = mapSupl;
  }
  private CustomMap<K, V> getRealMap() { // not thread-safe
    if (realMap == null) realMap = mapSupplier.get();
    return realMap;
  }
  @Override
  public int size() {
    return getRealMap().size();
  }
  @Override
  public V get(Object key) {
    return getRealMap().get(key);
  }
  @Override
  public V put(K key, V value) {
    return getRealMap().put(key, value);
  }
  @Override
  public V remove(Object key) {
    return getRealMap().remove(key);
  }
  @Override
  public void clear() {
    getRealMap().clear();
  }
  @Override
  public void forEach(BiConsumer<? super K, ? super V> action) {
    getRealMap().forEach(action);
  }
}
```

Our `VirtualCustomMap` is not thread-safe. If two threads call a method at the same time, we might accidentally create two real maps. We will fix this later by combining the virtual with a protection proxy.

Let's have a look at how we can use this. Our local variable `map` points to a `VirtualCustomMap` instance. The actual `HashMap` is not created until we call a method such as `put()` or `get()` for the first time. When we do, the supplier is invoked and the `HashMap` created. After that, all calls are routed to the contained `HashMap`.

Listing 2.4: `VirtualProxyDemo`.

```
//:~ samples/src/main/.../ch02/virtual/VirtualProxyDemo.java
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("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());
```

Listing 2.5 shows the output from `VirtualProxyDemo`.

Listing 2.5: Output From `VirtualProxyDemo`.

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

Since Java 7, both `ArrayList` and `HashMap` have zero-length element arrays by default. `ArrayDeque`, `Vector`, and `Hashtable` still allocate their arrays during construction.

Calling remote methods with remote proxy

Embassies are local representations of remote countries. We visit them when we need to apply for a visa, to renew our passport, or to interact with the remote entity (the country's government) without getting on a plane. Embassies are a bit like remote proxies.

Let's say we wish to apply for a visa to visit Canada.

Listing 2.6: `Canada`.

```
//:~ samples/src/main/.../ch02/remote/Canada.java
public interface Canada {
  boolean canGetVisa(String name,
                     boolean married,
                     boolean rich);
}
```

Our implementation class is defined inside `RealCanada`, which checks the most important attributes. The class name looks like that of a beer: *"Real Canada - goes down so smooth. Ahhh. Quenches the deepest thirst...."*

Listing 2.7: `RealCanada`.

```
//:~ samples/src/main/.../ch02/remote/RealCanada.java
public class RealCanada implements Canada {
  @Override
  public boolean canGetVisa(String name,
                            boolean married,
                            boolean rich) {
    return married && rich || !married && rich || rich;
    // every country loves rich tourists ... :-)
  }
}
```

We will publish this with the Spark Framework.

Listing 2.8: Remote proxy publisher.

```
//:~ samples/src/main/.../ch02/remote/ServicePublisher.java
public class ServicePublisher {
  public static void main(String... args) {
    Canada canada = new RealCanada();
    Spark.port(8080);
    Spark.get("/canGetVisa/:name/:married/:rich",
        (req, res) -> {
          var name = req.params("name");
          var married = "true".equals(req.params("married"));
          var rich = "true".equals(req.params("rich"));
          return canada.canGetVisa(name, married, rich);
        });
  }
}
```

That's it for the server side. When we run the `ServicePublisher`, it creates a REST service that we can connect to.

Next, we create a remote proxy that connects to the REST service and calls the method.

Listing 2.9: Our `CanadianEmbassy`.

```
//:~ samples/src/main/.../ch02/remote/CanadianEmbassy.java
public class CanadianEmbassy implements Canada {
  private final HttpClient httpClient =
      HttpClient.newBuilder().build();
  @Override
  public boolean canGetVisa(String name,
                            boolean married,
                            boolean rich) {
    try {
      var encodedName = URLEncoder.encode(name,
          StandardCharsets.UTF_8);
      var url = "http://0.0.0.0:8080/canGetVisa/" +
                  encodedName + "/" +
                  married + "/" + rich;
      var req = HttpRequest.newBuilder()
                  .uri(URI.create(url))
                  .build();
      var res = httpClient.send(
          req, HttpResponse.BodyHandlers.ofString());
      return res.statusCode() == HttpURLConnection.HTTP_OK &&
                Boolean.parseBoolean(res.body());
    } catch (IOException e) {
      throw new UncheckedIOException(e);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      throw new CancellationException("interrupted");
    }
  }
}
```

The `Client` looks like plain ol' Java, unaware of the remote implementation.

Listing 2.10: Client.

```
//:~ samples/src/main/.../ch02/remote/Client.java
public class Client {
  public static void main(String... args) {
    Canada canada = new CanadianEmbassy();
    boolean visaObtained = canada.canGetVisa(
        "Heinz Kabutz", true, false);
    System.out.println("visaObtained = " + visaObtained);
    System.out.println("Wins lottery ...");
    visaObtained = canada.canGetVisa(
        "Heinz Kabutz", true, true);
    System.out.println("visaObtained = " + visaObtained);
  }
}
```

We run this example by first starting the `ServicePublisher` class and then running our `Client`.

Listing 2.11: Did we get the visa?

```
visaObtained = false
Wins lottery ...
visaObtained = true
```

Security with the protection proxy

Security is either extremely boring, when it works, or super exciting (in a bad way), when it doesn't. Want to make sure you'll get a seat at a conference talk? Look for the one about Java security. It will be you and the speaker.

The protection proxy can guard our object for many reasons:

- We want to protect our object from race conditions.
- We want to protect our state from modification.
- We want to ensure the caller has the correct credentials.

We will look at examples for the first two scenarios. The third is essential for protecting our digital assets, but unless we do it extremely well, some clever hacker will break in and empty our wallets. We will sneakily ignore access control in our examples.

`SynchronizedCustomMap`

Our `CustomHashMap` in Listing 2.2 is not thread-safe. It relies on `java.util.HashMap`, which does not

guard the state against concurrent access. To illustrate, consider Listing 2.12. We try to put 46,000 entries into the map using a parallel stream. We then iterate through the map with `forEach()` and count how many entries we find. Lastly, we print out the value of `size()`. If the map were thread-safe, the values of `size()` and `entries` would both be equal to 46,000.

Listing 2.12: `ConcurrentTest` *modifies map concurrently using parallel streams.*

```
//:~ samples/src/test/.../ch02/protection/ConcurrentTest.java
public class ConcurrentTest {
  public static final int SQUARES = 46_000;
  public void check(CustomMap<Integer, Integer> map) {
    System.out.println(
        "Checking " + map.getClass().getSimpleName());
    try {
      IntStream.range(0, SQUARES)
          .parallel()
          .forEach(i -> map.put(i, i * i));
    } catch (Exception e) {
      System.err.println(e); // carry on with check
    }
    // count actual entries
    var entries = new LongAdder();
    map.forEach((k, v) -> entries.increment());

    System.out.println("entries = " + entries);
    System.out.println("map.size() = " + map.size());

    assertTrue("entries=" + entries + ", " +
                   "map.size=" + map.size(),
        entries.intValue() == map.size());
  }
}
```

Our `CustomHashMapTest` supplies the `CustomHashMap` to ConcurrentTest.

Listing 2.13: `CustomHashMapTest`.

```
//:~ samples/src/test/.../ch02/protection/CustomHashMapTest.java
public class CustomHashMapTest extends ConcurrentTest {
  @Test
  public void testCustomHashMap() {
    try {
      check(new CustomHashMap<>());  // won't work
    } catch (AssertionError error) {
      System.err.println(error);
    }
  }
}
```

Over several runs of this code on a MacBook Pro (2018) i9 with 6 cores, the counted `entries` never equaled `map.size()`. Nor did the count equal 46,000. For example, Listing 2.14 is the output of one run.

Listing 2.14: Output from `CustomHashMapTest`.

```
CustomHashMap constructed
Checking CustomHashMap
entries = 35474
map.size() = 41663
java.lang.AssertionError: entries=35474, map.size=41663
```

The consequences could be worse than merely an incorrect value for `size()` and missing entries. Corruption inside a `HashMap` might cause a node inside the map to point back to itself, in turn causing an infinite loop when calling methods such as `forEach()`. Even `get()`, `put()`, and `remove()` might not return if our key matches the bucket containing the node cycle.

To solve the race condition in `CustomHashMap`, we will proxy it with a `SynchronizedCustomMap`. This marks all methods as `synchronized` before delegating to the `CustomMap` methods.

Listing 2.15: `SynchronizedCustomMap` *uses* `synchronized` *for thread safety.*

```
//:~ samples/src/main/.../ch02/protection/SynchronizedCustomMap.java
public class SynchronizedCustomMap<K, V>
    implements CustomMap<K, V> {
  private final CustomMap<K, V> map;
  public SynchronizedCustomMap(CustomMap<K, V> map) {
    this.map = map;
  }
  @Override
  public synchronized int size() {
    return map.size();
  }
  @Override
  public synchronized V get(Object key) {
    return map.get(key);
  }
  @Override
  public synchronized V put(K key, V value) {
    return map.put(key, value);
  }
  @Override
  public synchronized V remove(Object key) {
    return map.remove(key);
  }
  @Override
  public synchronized void clear() {
    map.clear();
  }
  @Override
  public synchronized void forEach(
      BiConsumer<? super K, ? super V> action) {
    map.forEach(action);
  }
}
```

Our `SynchronizedTest` creates a `SynchronizedCustomMap` that proxies the `CustomHashMap` and passes that to `ConcurrentTest`.

Listing 2.16: SynchronizedTest for testing `SynchronizedCustomMap`.

```
//:~ samples/src/test/.../ch02/protection/SynchronizedTest.java
public class SynchronizedTest extends ConcurrentTest {
  @Test
  public void testSynchronizedCustomMap() {
    check(new SynchronizedCustomMap<>(new CustomHashMap<>()));
  }
}
```

As expected, we are free from race conditions when we protect all access to the `CustomMap` with `synchronized`.

Choose `synchronized` or `ReentrantLock`?

The coding idioms for `synchronized` are simpler than those for `ReentrantLock`. The `synchronized` mechanism also benefits from internal JVM optimizations such as biased locking, lock eliding, and lock coarsening, which do not apply to `ReentrantLock`. Unless we need the extended features of `ReentrantLock`, such as polled or timed locks, the preferred choice is `synchronized`.

`UnmodifiableCustomMap`

Collections returned directly from methods are often a source of bugs. It is not always clear what should happen when these collections are changed by the caller. Should the original collection be modified or should the change have no effect? One of the recommended approaches is to return an unmodifiable collection, that is, one which throws an `UnsupportedOperationException` if a method is called that would modify the contents of the collection. Our `CustomMap` is modified by methods `put()`, `remove()`, and `clear()`.

If we wanted to make sure that whoever got the map would not be able to modify it, we could send them an `UnmodifiableCustomMap`. This throws an `UnsupportedOperationException` whenever we call a modifier method such as `put()`.

Listing 2.17: `UnmodifiableCustomMap` *throws* `UnsupportedOperationException` *for modifier methods.*

```
//:~ samples/src/main/.../ch02/protection/UnmodifiableCustomMap.java
public class UnmodifiableCustomMap<K, V>
    implements CustomMap<K, V> {
  private final CustomMap<K, V> map;
  public UnmodifiableCustomMap(CustomMap<K, V> map) {
    this.map = map;
  }
  @Override
  public int size() {
    return map.size();
  }
  @Override
  public V get(Object key) {
    return map.get(key);
  }
  @Override
  public V put(K key, V value) {
    throw new UnsupportedOperationException("unmodifiable");
  }
  @Override
  public V remove(Object key) {
    throw new UnsupportedOperationException("unmodifiable");
  }
  @Override
  public void clear() {
    throw new UnsupportedOperationException("unmodifiable");
  }
  @Override
  public void forEach(BiConsumer<? super K, ? super V> action) {
    map.forEach(action);
  }
}
```

Our `UnmodifiableTest` checks what happens when we try to modify an `UnmodifiableCustomMap` using our previously shown `ConcurrentTest`.

Listing 2.18: `UnmodifiableTest`.

```
//:~ samples/src/test/.../ch02/protection/UnmodifiableTest.java
public class UnmodifiableTest extends ConcurrentTest {
  @Test
  public void testUnmodifiableCustomMap() {
    check(new UnmodifiableCustomMap<>(new CustomHashMap<>()));
  }
}
```

As expected, calling `put()` causes an `UnsupportedOperationException`.

Listing 2.19: Output from `UnmodifiableTest`.

```
CustomHashMap constructed
Checking UnmodifiableCustomMap
java.lang.UnsupportedOperationException: unmodifiable
entries = 0
map.size() = 0
```

The Trouble with `UnsupportedOperationException`

Throwing `UnsupportedOperationException` is common practice in Java. This exception is used 2,603 times in OpenJDK 14. Here are some other popular exceptions:

- `NullPointerException` is used 4,879 times.
- `IllegalStateException` is used 2,602 times.
- `IllegalArgumentException` is used 10,417 times.
- `ConcurrentModificationException` is used 242 times.
- `IllegalMonitorStateException` is used only 79 times.

As we see, `UnsupportedOperationException` is extensively used in the OpenJDK. When a method declares in its Javadoc documentation that it might throw it, we must expect it to happen. Unfortunately, we cannot predict which method will and will not throw it. Every time we call `add()` on a `Collection`, it potentially could blow up with an `UnsupportedOperationException`. This can make interfaces with methods that throw this exception hard to use. A better approach is to filter out unwanted methods using a decorator. We will demonstrate this in a later chapter.

Cascaded proxies

The original GoF proxy class diagram has an association from `Proxy` to `RealSubject`. A more flexible approach is for `Proxy` to instead point to the `Subject` interface. This is how we've coded every one of our proxy examples so far. By pointing to `Subject`, we can cascade our proxies: a protection proxy might point to a virtual proxy, which in turn might point to a remote proxy, and then to the `RealSubject`.

For example, let's take our previous `SynchronizedCustomMap` and combine that with our `VirtualCustomMap`.

Listing 2.20: Cascaded protection and virtual proxies.

```
//:~ samples/src/test/.../ch02/protection/SynchronizedVirtualTest.java
public class SynchronizedVirtualTest extends ConcurrentTest {
  @Test
  public void testCascadedSynchronizedVirtualCustomMap() {
    var map =
        new SynchronizedCustomMap<Integer, Integer>(
            new VirtualCustomMap<>(CustomHashMap::new));
    check(map);
    System.out.println("map = " + map);
  }
}
```

Let's look at the output in Listing 2.21. This time, the `CustomHashMap` is only constructed once we begin calling methods on the map.

Listing 2.21: Output from cascaded proxies.

```
Checking SynchronizedCustomMap
CustomHashMap constructed
entries = 46000
map.size() = 46000
map = eu.javaspecialists...SynchronizedCustomMap@6f79caec
```

Astute readers might have noticed that the output was not exactly what we expected! We would have liked to see the familiar output from printing a `Map` to `System.out`.

What happened?

In our great haste to handcraft these proxies, we forgot to override the `toString()` method. It thus defaulted to `Object.toString()`, which returns the class name and the default hash code.

We could change all our handcrafted proxies to include the `toString()` method, but let us instead solve this with dynamic proxies — in the next chapter.

Implementing `equals()` in proxies

The proxy object is obviously different than the object it is wrapping. However, the client might not want to see a difference between the `RealSubject` and the `Proxy`. We call this "object identity transparency".

We need an `equals()` method that is reflexive, symmetric, and transitive. Furthermore, we want `proxy.equals(real_subject)` to be `true` for object identity transparency. It follows from our requirement for symmetry that `real_subject.equals(proxy)` must also be `true`.

`Object.equals()` Javadoc Specification

The `equals` method implements an equivalence relation on non-null object references, as follows:

- It is **reflexive**, so that for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is **symmetric**, so that for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is **transitive**, so that for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is **consistent**, so that for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

Let's take another look at our `CustomHashMap`. Besides the `toString()` method, we also forgot to add `equals()` and `hashCode()`. A typical `equals()` auto-generated by an IDE such as IntelliJ or Eclipse would look like Listing 2.22.

Listing 2.22: Auto-generated `CustomHashMap.equals()`.

```
//:~ samples/src/main/.../ch02/equals/BrokenEqualsInCustomHashMap.java
@Override
public boolean equals(Object o) { // auto-generated by IDE
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;

  CustomHashMap<?, ?> that = (CustomHashMap<?, ?>) o;

  if (!map.equals(that.map)) return false;

  return true;
}
@Override
public int hashCode() {
  return map.hashCode();
}
```

An auto-generated `equals()` method for our `UnmodifiableCustomMap` would look similar. See Listing 2.23.

Listing 2.23: Auto-generated simple proxy `equals()`.

```
//:~ samples/src/main/.../ch02/equals/BrokenEqualsInHandCraftedProxy.java
@Override
public boolean equals(Object o) { // auto-generated by IDE
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;

  UnmodifiableCustomMap<?, ?> that =
      (UnmodifiableCustomMap<?, ?>) o;

  if (!map.equals(that.map)) return false;

  return true;
}
@Override
public int hashCode() {
  return map.hashCode();
}
```

While fulfilling the documented requirements, the code above does not achieve the desired object

identity transparency between the proxy and the real subject.

To achieve this transparency, let's begin by refining the proxy's `equals()` method. Note that delegating to `map` is consistent with how we've written proxy methods before.

Listing 2.24: Consistent proxy `equals()`.

```
//:~ samples/src/main/.../ch02/equals/FixedEqualsInHandCraftedProxy.java
@Override
public boolean equals(Object o) {
  return map.equals(o);
}
@Override
public int hashCode() {
  return map.hashCode();
}
```

The `CustomHashMap` requires a bit more work.

1. We ensure that the parameter is an instance of `CustomMap`. This returns `false` when the parameter is either `null` or an object other than a `CustomMap`.
2. We check whether the class of the argument is a `CustomHashMap`. If it is the same class, we downcast and compare the two `map` fields. A best practice is to mark the `equals()` method in `CustomHashMap` as `final` to prevent unexpected overriding, as that could create non-symmetric `equals()` methods. Since `hashCode()` needs to be paired with `equals()`, we also mark that as `final`.
3. Otherwise, we return the result of reversing the equals condition as `o.equals(this)`. Without the reversing, we would end up in an infinite loop. Since `equals()` is required to be symmetric, the result of `o.equals(this)` has to be the same as `this.equals(o)`.

Listing 2.25: Correct `CustomHashMap.equals()`.

```
//:~ samples/src/main/.../ch02/equals/FixedEqualsInCustomHashMap.java
@Override
public final boolean equals(Object o) {
  if (this == o) return true;
  if (!(o instanceof CustomMap)) return false;

  if (o instanceof CustomHashMap) {
    CustomHashMap<?, ?> that = (CustomHashMap<?, ?>) o;
    return this.map.equals(that.map);
  }
  return o.equals(this); // reversing the equals() condition
}
@Override
public final int hashCode() {
  return map.hashCode();
}
```

We see the `equals()` method in action in `CustomMapEqualsTest`. For a more complete set of tests, please visit github.com/kabutz/dynamic-proxies-samples.

Listing 2.26: `equals()` *example.*

```
//:~ samples/src/test/.../ch02/equals/CustomMapEqualsTest.java
public class CustomMapEqualsTest {
  @Test
  public void testEqualsContract() {
    var real = new CustomHashMap<Integer, Integer>();
    for (int i = 0; i < 10; i++) real.put(i, i * i);
    var proxy1 = new UnmodifiableCustomMap<>(real);
    var proxy2 = new UnmodifiableCustomMap<>(real);
    var empty = new CustomHashMap<Integer, Integer>();

    // reflexive
    assertEquals(real, real);
    assertEquals(proxy1, proxy1);
    assertEquals(proxy2, proxy2);

    // symmetric
    assertEquals(real, proxy1);
    assertEquals(proxy1, real);
    assertEquals(proxy1, proxy2);
    assertEquals(proxy2, proxy1);

    // transitive
    if (real.equals(empty) && empty.equals(proxy2))
      assertEquals(real, proxy2);
    if (real.equals(proxy1) && proxy1.equals(proxy2))
      assertEquals(real, proxy2);
  }
}
```

Voila! We have an `equals()` that fulfills all the requirements described above and has allowed our real subject to be “proxiable”. It is reflexive, symmetric, and transitive. Plus, if we compare an object to its proxy, we get the desired result of `true`.

CHAPTER THREE

Dynamic Proxy

Until now, we've coded the proxy classes by hand. It makes sense to handcraft the code when each type of proxy is unique in structure and function. However, the majority of proxy implementations are similar. For example, remote proxies all have to shunt parameters and return values back and forth. Java RMI generates the glue code for remote proxies in the form of stubs and skeletons. IDEs will generate delegation boilerplate for us, but we still have to read all that code to understand what it is doing.

Code generation can be either static or dynamic. Static is less convenient as it requires an extra build step. Dynamic creates a new proxy class at run time. It is super convenient and flexible. In this chapter, we will learn how the dynamic proxy works in Java.

`Proxy.newProxyInstance()`

We construct dynamic proxies with the static factory method `newProxyInstance()` with these three parameters:

- `ClassLoader loader` is where our new proxy class is loaded. The `ClassLoader` needs to be able to see the interfaces that we want to implement. For example, the bootstrap class loader knows about the Java interfaces (`java.lang.Runnable`, `java.util.Collection`, etc.) but not about application interfaces. Our classes are loaded with the system class loader (also called `AppClassLoader`).
- `Class<?>[] interfaces` is an array that contains all the interfaces that our proxy object needs to implement.
- `InvocationHandler` is a functional interface that is invoked whenever any method is called on the proxy object.

The Factory Design Pattern

One of the most quoted patterns from the GoF is the factory. The only problem is that there is no factory pattern in the book.

The GoF describe a "factory method", which is an abstract method for initializing objects, delegating the construction to subclasses. An example is the `iterator()` method, where `ArrayList` creates an `ArrayList$Itr` and `LinkedList` creates a `LinkedList$ListItr`.

The GoF also describe an "abstract factory" for creating families of related products. An example is `java.awt.Toolkit`, which creates GUI components for each operating system.

The pattern that most programmers think of as "factory" is a static method for creating complex objects. We will name this a "static factory method". Examples of this abound in the JDK: `Arrays.asList()`, `Proxy.newProxyInstance()`, `DriverManager.getConnection()`, `ByteBuffer.allocateDirect()`, etc.

`IllegalArgumentException` abounds

`Proxy` is restrictive about the parameters it will accept in the `newProxyInstance()` method. Any incorrect combination is greeted with an `IllegalArgumentException`, where the message gives clues about what caused the error. This is an example of how dynamic proxies are *dynamic* — the static type system cannot capture all the restrictions, so failure to conform results in dynamic errors.

For example, what is wrong with the call to `Proxy.newProxyInstance()` in Listing 3.1?

Listing 3.1: Incorrect `Proxy.newProxyInstance()` call.

```
//:~
samples/src/test/.../ch03/gotchas/ClassLoaderVisibilityForDynamicProxiesTe
st.java
Proxy.newProxyInstance(
    Map.class.getClassLoader(),
    new Class<?>[] {CustomMap.class},
    (proxy, method, args) -> null
);
```

When we run it, we see the output of Listing 3.2.

Listing 3.2: `IllegalArgumentException` from incorrect class loader.

```
java.lang.IllegalArgumentException: eu.javaspecia...CustomMap
    referenced from a method is not visible from class loader
```

`Map.class` is a JDK class, thus loaded by the bootstrap class loader, which does not know about our `CustomMap`.

Instead, we should use the class loader that loaded our `CustomMap`, in this case the system class loader.

Listing 3.3: Correct `Proxy.newProxyInstance()` call.

```
//:~
samples/src/test/.../ch03/gotchas/ClassLoaderVisibilityForDynamicProxiesTe
st.java
Proxy.newProxyInstance(
    CustomMap.class.getClassLoader(),
    new Class<?>[] {CustomMap.class},
    (proxy, method, args) -> null
);
```

The dynamic proxy is magically injected into whichever class loader we specify. We need to make sure it is the correct one, especially in an application server environment.

Further details can be found in the "newProxyInstance" section of the Class Proxy Javadoc documentation.

Java Platform Module System

The Java Platform Module System (JPMS) also applies to dynamic proxies. Proxy classes have the same restrictions as the interfaces that they implement. For example, if one of the interfaces is non-public, then the proxy class is created in the same package as that interface. Similarly, if one of the proxy interfaces is in a package that is non-exported and non-open, then the proxy class would also be non-exported and non-open, possibly constructed in a dynamic module if all the interfaces are public.

We can print debug information about how the JPMS creates the dynamic proxy with the environment variable `-Djdk.proxy.debug=debug`.

Further details and rules of using dynamic proxies and JPMS can be read in the "Package and Module Membership of Proxy Class" section of the Class Proxy Javadoc documentation.

`InvocationHandler`

Any method called on the proxy object is dispatched to the `invoke()` method of the `InvocationHandler`.

Listing 3.4: `InvocationHandler`.

```
//:~ samples/src/main/.../ch03/InvocationHandlerDeclaration.java
public interface InvocationHandler {
  public Object invoke(Object proxy,
                       Method method,
                       Object[] args) throws Throwable;
}
```

The method `invoke()` has three parameters:

- `Object proxy` is the instance of the dynamic proxy class that is calling `invoke()`. We will use this in a later chapter.
- `Method method` is a `java.lang.reflect.Method` object for the method that was called. The method comes from either one of the interfaces used to create the dynamic proxy or one of the three public non-final methods from `Object`, that is `hashCode()`, `equals(Object)`, or `toString()`.
- `Object[] args` is an array of the parameters passed into the method. This will be `null` when the

method does not have any parameters.

The `invoke()` method declares that it throws `Throwable`. However, methods should only throw those checked exceptions that are declared in their signature. This is another example of how dynamic proxies are *dynamic* — the static type system says `Throwable`, but there are dynamic constraints on which exceptions they can throw.

`LoggingInvocationHandler`

Let us have a look at how to create our own `InvocationHandler`. In our `LoggingInvocationHandler`, we log method calls and how long they take. We assume that `Object obj` implements the same interfaces as the dynamic proxy. We thus delegate the method call to `obj` with `return method.invoke(obj, args)`, surrounding this with our logging code.

Most of the code is self-explanatory, except for the `logFine` optimization. `System.nanoTime()` is an expensive system call and we want to avoid calling it except when fine logging is enabled. Since the level may change during our call to `method.invoke()`, we write the original setting into a `final` local variable `logFine`. Once again, `method.invoke()` is an example of how dynamic proxies are *dynamic* — we often delegate to underlying functionality, and we do so via reflection.

Listing 3.5: `LoggingInvocationHandler`.

```
//:~ samples/src/main/.../ch03/logging/LoggingInvocationHandler.java
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;
  }
  @Override
  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");
    }
  }
  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(", ")));
  }
}
```

In our `LoggingProxyDemo` in Listing 3.6, we create a `LoggingInvocationHandler` that logs all method calls to a `ConcurrentHashMap`.

We then create a dynamic proxy with a call to `Proxy.newProxyInstance()`, passing in the class loader of `Map` (thus the bootstrap class loader), an array containing the `Map` interface, and our instance of `LoggingInvocationHandler`.

The result from the `newProxyInstance()` call is of type `Object`, which we need to manually cast to the correct type, in our case `Map<String, Integer>`. This is an example of how dynamic proxies are *dynamic* — the static type system says `Object`, but the dynamic type is narrower.

A dynamic proxy may implement a lot of interfaces, so it would not be clear which one we would want to cast to.

Once we have our dynamic proxy object cast to `Map`, all the methods we call on it are logged, together with their execution time.

Listing 3.6: Logging methods of a map.

```
//:~ samples/src/main/.../ch03/logging/LoggingProxyDemo.java
var handler = new LoggingInvocationHandler(
    Logger.getGlobal(), new ConcurrentHashMap<>());
@SuppressWarnings("unchecked")
var map = (Map<String, Integer>)
              Proxy.newProxyInstance(
                  Map.class.getClassLoader(),
                  new Class<?>[] {Map.class},
                  handler);
map.put("one", 1);
map.put("two", 2);
System.out.println(map);
map.clear();
```

Name of dynamic class

What is the class name of the dynamic proxy? For example, consider the `ISODateParser` interface in Listing 3.7, which parses dates of the format `yyyy-MM-dd`. This equates to the `Subject` interface in Figure 7.

Listing 3.7: `ISODateParser` interface.

```
//:~ samples/src/main/.../ch03/ISODateParser.java
/**
 * Parses String of the format yyyy-MM-dd and returns LocalDate.
 */
public interface ISODateParser {
  LocalDate parse(String date) throws ParseException;
}
```

We create a dynamic proxy and print the class name to System.out.

Listing 3.8: `ProxyName`.

```
//:~ samples/src/main/.../ch03/ProxyName.java
public class ProxyName {
  public static void main(String... args) {
    System.out.println(
        Proxy.newProxyInstance(
            ISODateParser.class.getClassLoader(),
            new Class<?>[] {ISODateParser.class},
            (proxy, method, arguments) -> null
        ).getClass()
    );
  }
}
```

The output shows the class name as `com.sun.proxy.$Proxy0`.

Listing 3.9: `ProxyName` *output.*

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

Dynamic proxies are normally called `$Proxy` followed by a number. When all the interfaces are public, the package will typically be `com.sun.proxy`, but it could be another package on your operating system and JDK combination. The class name and package could change from version to version or even from run to run. The package name is different when we create a dynamic proxy from a non-exported package inside a module. When at least one of the interfaces is non-public, the proxy class is in the same package as that interface. It is not possible to create a dynamic proxy of several non-public interfaces in different packages.

Dissecting `$Proxy0`

We can dump the generated proxy classes to our current working directory with the system property `-Djdk.proxy.ProxyGenerator.saveGeneratedFiles=true` in Java 9+ (or `-Dsun.misc.ProxyGenerator.saveGeneratedFiles=true` in earlier versions of Java).

For example, Listing 3.10 shows the dynamic proxy generated from the `ISODateParser` interface. We can see that `$Proxy0` is derived from `java.lang.reflect.Proxy`, which stores the `InvocationHandler` instance `h`. The public interface methods, as well as `hashCode()`, `equals()`, and `toString()` from `java.lang.Object`, have associated `Method` objects stored in static fields. These fields are initialized in the static initializer block when the class is loaded. In each method implementation, the proxy instance, the `Method` instance, and an array containing the parameters are passed to the `InvocationHandler`. For example, see the implementation of `parse(String)`.

Listing 3.10: What the `$Proxy0` class looks like.

```
package com.sun.proxy;

//:~ samples/src/main/.../ch03/generated/$Proxy0.java
import eu.javaspecialists.books.dynamicproxies.samples.ch03.*;

import java.lang.reflect.*;
import java.text.*;
import java.time.*;

public final class $Proxy0 extends Proxy
    implements ISODateParser {
  private static Method m0;
  private static Method m1;
  private static Method m2;
  private static Method m3;

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

  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);
    }
  }

  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);
    }
  }

  static {
    try {
      m0 = Object.class.getMethod("hashCode");
      m1 = Object.class.getMethod("equals", Object.class);
      m2 = Object.class.getMethod("toString");
      m3 = ISODateParser.class.getMethod("parse", String.class);
    } catch (NoSuchMethodException e) {
      throw new NoSuchMethodError(e.getMessage());
    }
  }
}
```

As we can see, the dynamic proxy is not magic.

Turbo-boosting reflection methods

One of the parameters passed into the `invoke()` method of the `InvocationHandler` is the `Method` that was called. We often need to invoke this `Method` using reflection, and want this to be as fast as possible. Since the proxy implements public methods, we also know that any `Method` field stored inside the dynamic proxy classes is also public. Public methods are accessible, but unfortunately reflection suffers from amnesia. Every time we call `invoke()` on the `Method` object, it checks from the stack who the calling class is and whether it has the correct permissions. We can turn this check off by calling `setAccessible(true)` on each of the `Method` fields. This reduces the overhead of method calls when using reflection.

For example, here is our `MethodTurboBooster`. If, for some reason, we get an exception, we return the proxy object unchanged. Turbo-boosting is on by default, but we can disable it with `-Deu.javaspecialists.books.dynamicproxies.util.MethodTurboBooster.disabled=true`.

Listing 3.11: `MethodTurboBooster`.

```
//:~ core/src/main/.../util/MethodTurboBooster.java
/**
 * Method turbo boosting is enabled by default.  We call
 * setAccessible(true) on Method objects. Exceptions are silently
 * ignored.
 * <p>
 * Method turbo boosting can be disabled with
 * -Deu.javaspecialists.books.dynamicproxies.util.\
 * MethodTurboBooster.disabled=true
 */
public final class MethodTurboBooster {
  private final static Booster BOOSTER =
      Boolean.getBoolean(
          MethodTurboBooster.class.getName() + ".disabled") ?
          new BoosterOff() : new BoosterOn();

  public static <E> E boost(E proxy) {
    return BOOSTER.turboBoost(proxy);
  }

  public static Method boost(Method method) {
    return BOOSTER.turboBoost(method);
  }

  private interface Booster {
    <E> E turboBoost(E proxy);
    Method turboBoost(Method method);
  }

  private static class BoosterOn implements Booster {
    @Override
    public <E> E turboBoost(E proxy) {
      if (!(proxy instanceof Proxy))
        throw new IllegalArgumentException(
            "Can only turboboost instances of Proxy"
        );
      try {
```

```
        for (var field : proxy.getClass().getDeclaredFields()) {
          if (field.getType() == Method.class) {
            field.setAccessible(true);
            turboBoost((Method) field.get(null));
          }
        }
        return proxy;
      } catch (IllegalAccessException | RuntimeException e) {
        // could not turbo-boost - return proxy unchanged;
        return proxy;
      }
    }
    @Override
    public Method turboBoost(Method method) {
      try {
        method.setAccessible(true);
      } catch (RuntimeException e) {
        // could not turbo-boost - return method unchanged;
      }
      return method;
    }
  }

  private static class BoosterOff implements Booster {
    @Override
    public <E> E turboBoost(E proxy) {
      if (!(proxy instanceof Proxy))
        throw new IllegalArgumentException(
            "Can only turboboost instances of Proxy"
        );
      return proxy;
    }
    @Override
    public Method turboBoost(Method method) {
      return method;
    }
  }
}
```

Unwrapping the `InvocationTargetException`

`InvocationHandler.invoke()` declares that it throws `Throwable`, but, as we saw in the `$Proxy0` class, any unexpected exception is wrapped in an `UndeclaredThrowableException`. Consider the code in Listing 3.12. On careful inspection of the code, we see that we are reversing the comparison in our reflective call `method.invoke()`. Thus, we are comparing `String` to our proxy, which should result in a `ClassCastException`.

Listing 3.12: `RecastingExceptionsBroken`.

```
//:~ samples/src/main/.../ch03/gotchas/RecastingExceptionsBroken.java
Comparable<String> comp =
    (Comparable<String>) Proxy.newProxyInstance(
        Comparable.class.getClassLoader(),
        new Class<?>[] {Comparable.class},
        (proxy, method, params) ->
            method.invoke(params[0], proxy));
System.out.println(comp.compareTo("hello"));
```

When we run this code, we instead see three exceptions chained together.

Listing 3.13: Output from `RecastingExceptionsBroken`.

```
Exception in thread "main" java.lang.reflect.UndeclaredThrowableException
    at com.sun.proxy.$Proxy0.compareTo(Unknown Source)
    at ...ch03.gotchas.RecastingExceptionsBroken.main
Caused by: java.lang.reflect.InvocationTargetException
    at ...
    at ...ch03.gotchas.RecastingExceptionsBroken.lambda$main$0
Caused by: java.lang.ClassCastException: class com.sun.proxy.$Proxy0
     cannot be cast to class java.lang.String
```

If we had thrown `ClassCastException` directly, since that is a `RuntimeException`, our proxy code would have passed it on to the caller. Thus, if we use reflection inside our `InvocationHandler.handle()` method, we need to throw the exception contained inside the `InvocationTargetException`. To make this easier, we wrote an `ExceptionUnwrappingInvocationHandler` in Listing 3.14.

Listing 3.14: `ExceptionUnwrappingInvocationHandler`.

```
//:~ core/src/main/.../handlers/ExceptionUnwrappingInvocationHandler.java
public final class ExceptionUnwrappingInvocationHandler
    implements InvocationHandler, Serializable {
  private static final long serialVersionUID = 1L;
  private final InvocationHandler handler;
  public ExceptionUnwrappingInvocationHandler(
      InvocationHandler handler) {
    this.handler = handler;
  }
  @Override
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    try {
      return handler.invoke(proxy, method, args);
    } catch (InvocationTargetException ex) {
      throw ex.getCause();
    }
  }
  public InvocationHandler getNestedInvocationHandler() {
    return handler;
  }
}
```

Applying this to our previous example from Listing 3.12 is a minimal amount of work. We simply wrap our `InvocationHandler`.

Listing 3.15: `RecastingExceptionsFixed`.

```
//:~ samples/src/main/.../ch03/gotchas/RecastingExceptionsFixed.java
Comparable<String> comp =
    (Comparable<String>) Proxy.newProxyInstance(
        Comparable.class.getClassLoader(),
        new Class<?>[] {Comparable.class},
        new ExceptionUnwrappingInvocationHandler(
            (proxy, method, params) ->
                method.invoke(params[0], proxy)));
System.out.println(comp.compareTo("hello"));
```

This time, when we run it, the output is a single `ClassCastException`. Just what the doctor ordered!

Listing 3.16: Output from `RecastingExceptionsFixed`.

```
Exception in thread "main" java.lang.ClassCastException: class
      com.sun.proxy.$Proxy0 cannot be cast to class java.lang.String
    at java.base/java.lang.String.compareTo(String.java:134)
    at ...
```

Casting

This section contains static factory methods that we will use throughout the rest of this book.

In Listing 3.6, we had to manually cast the proxy instance to `Map<String, Integer>`. But casting is so last millennium. Instead, generics can do the cast for us.

Listing 3.17: `Proxies.castProxy()`.

```
//:~ core/src/main/.../Proxies.java
/**
 * @param intf    The interface to implement and cast to
 * @param handler InvocationHandler for all methods
 */
@SuppressWarnings("unchecked")
public static <S> S castProxy(Class<? super S> intf,
                              InvocationHandler handler) {
  Objects.requireNonNull(intf, "intf==null");
  Objects.requireNonNull(handler, "handler==null");
  return MethodTurboBooster.boost(
      (S) Proxy.newProxyInstance(
          intf.getClassLoader(),
          new Class<?>[] {intf},
          new ExceptionUnwrappingInvocationHandler(handler)));
}
```

To demonstrate, we also define `Proxies.simpleProxy()` to create dynamic proxies that delegate all method calls to the wrapped object.

Listing 3.18: `Proxies.simpleProxy()`.

```
//:~ core/src/main/.../Proxies.java
public static <S> S simpleProxy(
    Class<? super S> subjectInterface, S subject) {
  return castProxy(subjectInterface,
      (InvocationHandler & Serializable)
          (proxy, method, args) -> method.invoke(subject, args)
  );
}
```

We'll now use this with our `RealISODateParser` class in Listing 3.19 that implements our `ISODateParser` interface from Listing 3.7.

Listing 3.19: `RealISODateParser` *example.*

```
//:~ samples/src/main/.../ch03/RealISODateParser.java
public class RealISODateParser implements ISODateParser {
  private static final DateTimeFormatter formatter =
      DateTimeFormatter.ISO_LOCAL_DATE;
  @Override
  public LocalDate parse(String date) throws ParseException {
    try {
      return LocalDate.parse(date, formatter);
    } catch (DateTimeParseException e) {
      throw new ParseException(e.toString(), e.getErrorIndex());
    }
  }

  @Override
  public String toString() {
    return getClass().getSimpleName();
  }
}
```

To create our dynamic proxy, we call the `Proxies.simpleProxy()` method and pass in the interface that we are looking to implement, as well as the object that we would like to wrap. The generic specification of `simpleProxy(Class<? super S> subjectInterface, S subject)`, ensures that the instance `subject` matches the interface. We use the `castProxy()` method to internally do the casting for us.

Listing 3.20: Using `Proxies` to create a simple proxy.

```
//:~ samples/src/main/.../ch03/DynamicProxies.java
ISODateParser parser = Proxies.simpleProxy(
    ISODateParser.class, new RealISODateParser());
LocalDate palindrome = parser.parse("2020-02-02");
System.out.println("palindrome = \"" + palindrome + "\"");
System.out.println(parser);
LocalDate funnyDate = parser.parse("2020-04-31");
```

Listing 3.21: Output from the simple proxy example.

```
palindrome = "2020-02-02"
RealISODateParser
Exception in thread "main" java.text.ParseException:
    java.time.format.DateTimeParseException: Text '2020-04-31'
        could not be parsed: Invalid date 'APRIL 31'
```

In our handcrafted proxy in Listing 2.20 (Chapter 2), we had forgotten to proxy the `toString()` method, so the default `Object.toString()` method was called. With dynamic proxies, the `toString()` method is automatically proxied, together with `equals()` and `hashCode()`.

Virtual dynamic proxy

In Listing 2.3, we wrote a virtual proxy for lazily creating our `CustomHashMap` instances. Following the same approach as our handcrafted proxy, we can create a `VirtualProxyHandler` that takes a `Supplier<? extends S>` as a parameter.

Listing 3.22: `VirtualProxyHandler`.

```
//:~ core/src/main/.../handlers/VirtualProxyHandler.java
public final class VirtualProxyHandler<S>
    implements InvocationHandler, Serializable {
  private static final long serialVersionUID = 1L;
  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;
  }
  @Override
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    return method.invoke(getSubject(), args);
  }
}
```

We can add a static factory method to our `Proxies` façade.

Listing 3.23: `Proxies.virtualProxy()`.

```
//:~ core/src/main/.../Proxies.java
public static <S> S virtualProxy(
    Class<? super S> subjectInterface,
    Supplier<? extends S> subjectSupplier) {
  Objects.requireNonNull(subjectSupplier,
      "subjectSupplier==null");
  return castProxy(subjectInterface,
      new VirtualProxyHandler<>(subjectSupplier));
}
```

Our virtual proxy test now creates the `CustomMap` with `Proxies.virtualProxy(CustomMap.class, CustomHashMap::new)`.

Listing 3.24: `VirtualProxyDemo`.

```
//:~ samples/src/main/.../ch03/virtual/VirtualProxyDemo.java
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());
```

The output is exactly the same as with our handcrafted virtual proxy in Listing 2.5.

Synchronized dynamic proxy

Instead of handcrafting a synchronized protection proxy like we did in Listing 2.15, we can write a `SynchronizedHandler`. This synchronizes all calls to methods on our `subject`, using `proxy` as our monitor lock.

Listing 3.25: `SynchronizedHandler`.

```
//:~ core/src/main/.../handlers/SynchronizedHandler.java
public final class SynchronizedHandler<S>
    implements InvocationHandler, Serializable {
  private static final long serialVersionUID = 1L;
  private final S subject;
  public SynchronizedHandler(S subject) {
    this.subject = subject;
  }
  @Override
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    // synchronize on the proxy instance, which is similar to
    // how Vector and Collections.synchronizedList() work
    synchronized (proxy) {
      return method.invoke(subject, args);
    }
  }
}
```

Our `Proxies` façade now gets a `synchronizedProxy()` static factory method.

Listing 3.26: `Proxies.synchronizedProxy()`.

```
//:~ core/src/main/.../Proxies.java
public static <S> S synchronizedProxy(
    Class<? super S> subjectInterface, S subject) {
  Objects.requireNonNull(subject, "subject==null");
  return castProxy(subjectInterface,
      new SynchronizedHandler<>(subject));
}
```

We can verify the synchronization works as expected with the `SynchronizedTest` from Listing 2.16.

Listing 3.27: `SynchronizedTest`.

```
//:~ samples/src/test/.../ch03/protection/SynchronizedTest.java
public class SynchronizedTest extends ConcurrentTest {
  @Test
  public void testDynamicSynchronizedProxy() {
    check(Proxies.synchronizedProxy(
        CustomMap.class, new CustomHashMap<>()
    ));
  }
}
```

The output from our `SynchronizedTest` proves that it probably works correctly. Yes, that's cautious wording, but it is hard to prove the absence of bugs, especially with concurrent code.

Listing 3.28: Output from `SynchronizedTest`.

```
CustomHashMap constructed
Checking $Proxy0
entries = 46000
map.size() = 46000
```

Cascading multiple dynamic proxies

In our previous chapter, we cascaded a synchronized protection proxy and a virtual proxy. We can do the same with our newly minted dynamic proxies. Note that we need to add a type witness to the call to `virtualProxy()` to specify that the generic type `S` is `CustomMap<Integer, Integer>`. Otherwise, the compiler will cast our virtual proxy to `CustomHashMap`, resulting in a `ClassCastException`.

Listing 3.29: `SynchronizedVirtualTest`.

```
//:~ samples/src/test/.../ch03/protection/SynchronizedVirtualTest.java
public class SynchronizedVirtualTest extends ConcurrentTest {
  @Test
  public void testDynamicSynchronizedVirtualProxy() {
    CustomMap<Integer, Integer> map =
        Proxies.synchronizedProxy(CustomMap.class,
            Proxies.<CustomMap<Integer, Integer>>virtualProxy(
                CustomMap.class, CustomHashMap::new));
    check(map);
    System.out.println("map = " + map);
  }
}
```

The output of our `SynchronizedVirtualTest' is shown in Listing 3.30.

Listing 3.30: Output from cascaded dynamic proxies.

```
Checking $Proxy0
CustomHashMap constructed
entries = 46000
map.size() = 46000
map = {0=0, 1=1, 2=4, 3=9, 4=16, 5=25, ..., 45999=2115908001}
```

This time, the `toString()` method is auto-magically routed by our dynamic proxies.

`EnhancedStream` with dynamic proxies

I recently discovered a nice use case for dynamic proxies. Java 8's streams have a rather narrow-minded view of how to implement `distinct()`. If two objects are `equal()`, then they are distinct and typically the first element is kept and subsequent elements discarded.

What if we would like a different equality definition? To paraphrase George Orwell's *Animal Farm*, when two objects are equal, can one be more equal than the other?

For example, `ArrayDeque` has two `clone()` methods, one that returns `Object` and the other `ArrayDeque`. If deciding which one to keep, we should choose the one with the most derived return type: `ArrayDeque clone()`. The ordinary `distinct()` method in `Stream` cannot make that choice.

One of the nice features of `Stream` is its fluent interface. Once we have a `Stream`, a lot of methods return the same `Stream` type, so that we can chain method calls. If we are going to enhance the stream, then these methods should also return our `EnhancedStream`.

Our interface `EnhancedStream` in Listing 3.31 extends `Stream` and adds a new `distinct()` method. It also overrides all the methods that are returning `Stream` to instead return an `EnhancedStream`. Lastly, we have two static factory methods for creating streams.

Listing 3.31: `EnhancedStream<T>`.

```
//:~ samples/src/main/.../ch03/enhancedstream/EnhancedStream.java
/**
 * Described in The Java Specialists Newsletters:
 * https://www.javaspecialists.eu/archive/Issue274.html
 * https://www.javaspecialists.eu/archive/Issue275.html
 */
public interface EnhancedStream<T> extends Stream<T> {
  // new enhanced distinct()
  EnhancedStream<T> distinct(ToIntFunction<T> hashCode,
                             BiPredicate<T, T> equals,
                             BinaryOperator<T> merger);
  EnhancedStream<T> distinct(Function<T, ?> keyGenerator,
                             BinaryOperator<T> merger);

  // inherited methods with enhanced return type
  EnhancedStream<T> filter(Predicate<? super T> predicate);
  <R> EnhancedStream<R> map(
      Function<? super T, ? extends R> mapper);
  <R> EnhancedStream<R> flatMap(
      Function<? super T, ? extends Stream<? extends R>> mapper);
  EnhancedStream<T> distinct();
  EnhancedStream<T> sorted();
  EnhancedStream<T> sorted(Comparator<? super T> comparator);
  EnhancedStream<T> peek(Consumer<? super T> action);
  EnhancedStream<T> limit(long maxSize);
  EnhancedStream<T> skip(long n);
  EnhancedStream<T> takeWhile(Predicate<? super T> predicate);
  EnhancedStream<T> dropWhile(Predicate<? super T> predicate);
  EnhancedStream<T> sequential();
  EnhancedStream<T> parallel();
  EnhancedStream<T> unordered();
  EnhancedStream<T> onClose(Runnable closeHandler);

  // static factory methods
  @SafeVarargs
  @SuppressWarnings("varargs")
  static <E> EnhancedStream<E> of(E... elements) {
```

```
    return from(Stream.of(elements));
  }
  @SuppressWarnings("unchecked")
  static <E> EnhancedStream<E> from(Stream<E> stream) {
    return (EnhancedStream<E>) Proxy.newProxyInstance(
        EnhancedStream.class.getClassLoader(),
        new Class<?>[] {EnhancedStream.class},
        new EnhancedStreamHandler<>(stream)
    );
  }
}
```

Our `EnhancedStreamHandler` contains a static inner class `Key` that is used for determining "uniqueness" among stream elements. All method calls are routed via the `invoke()` method. Inside `invoke()`, we first decide whether the method is our enhanced `distinct()` method. If it is, we call that directly. Otherwise, if the return type is `EnhancedStream`, we find the matching method in our `methodMap` and invoke that on our `delegate`. In this case, we return the `proxy`, which is an instance of type `EnhancedStream`. Alternatively, we return the result of calling the method directly on our `delegate`.

Listing 3.32: `EnhancedStreamHandler<T>`.

```
//:~ samples/src/main/.../ch03/enhancedstream/EnhancedStreamHandler.java
public class EnhancedStreamHandler<T>
    implements InvocationHandler {
  private Stream<T> delegate;

  public EnhancedStreamHandler(Stream<T> delegate) {
    this.delegate = delegate;
  }

  @Override
  @SuppressWarnings("unchecked")
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    if (method.getReturnType() == EnhancedStream.class) {
      if (method.equals(ENHANCED_DISTINCT)) {
        distinct((ToIntFunction<T>) args[0],
            (BiPredicate<T, T>) args[1],
            (BinaryOperator<T>) args[2]);
      } else if (method.equals(ENHANCED_DISTINCT_WITH_KEY)) {
        distinct((Function<T, ?>) args[0],
            (BinaryOperator<T>) args[1]);
```

```
      } else {
        Method match = methodMap.get(method);
        this.delegate = (Stream<T>) match.invoke(delegate, args);
      }
      return proxy;
    } else {
      return method.invoke(this.delegate, args);
    }
  }

  private void distinct(ToIntFunction<T> hashCode,
                        BiPredicate<T, T> equals,
                        BinaryOperator<T> merger) {
    distinct(t -> new Key<>(t, hashCode, equals), merger);
  }
  private void distinct(Function<T, ?> keyGen,
                        BinaryOperator<T> merger) {
    delegate = delegate.collect(Collectors.toMap(keyGen::apply,
        Function.identity(), merger, LinkedHashMap::new))
                   .values()
                   .stream();
  }

  private static final class Key<E> {
    private final E e;
    private final ToIntFunction<E> hashCode;
    private final BiPredicate<E, E> equals;

    public Key(E e, ToIntFunction<E> hashCode,
               BiPredicate<E, E> equals) {
      this.e = e;
      this.hashCode = hashCode;
      this.equals = equals;
    }

    @Override
    public int hashCode() {
      return hashCode.applyAsInt(e);
    }

    @Override
    public boolean equals(Object obj) {
      if (!(obj instanceof Key)) return false;
```

```
      @SuppressWarnings("unchecked")
      Key<E> that = (Key<E>) obj;
      return equals.test(this.e, that.e);
    }
  }

  private static final Method ENHANCED_DISTINCT;
  private static final Method ENHANCED_DISTINCT_WITH_KEY;

  static {
    try {
      ENHANCED_DISTINCT =
          EnhancedStream.class.getMethod(
              "distinct", ToIntFunction.class, BiPredicate.class,
              BinaryOperator.class);
      ENHANCED_DISTINCT_WITH_KEY =
          EnhancedStream.class.getMethod(
              "distinct", Function.class, BinaryOperator.class);
    } catch (NoSuchMethodException e) {
      throw new Error(e);
    }
  }

  // methodMap contains a map from all non-static Stream methods
  // to the matching EnhancedStream methods
  private static final Map<Method, Method> methodMap =
      Stream.of(Stream.class.getMethods())
          .filter(m -> !Modifier.isStatic(m.getModifiers()))
          .collect(Collectors.toUnmodifiableMap(
              EnhancedStreamHandler::getEnhancedStreamMethod,
              Function.identity()));

  // since EnhancedStream is a subclass of Stream, it has to
  // contain all the methods of Stream.  We can safely do the
  // lookup, throwing an Error if we cannot find a method
  private static Method getEnhancedStreamMethod(Method m) {
    try {
      return EnhancedStream.class.getMethod(
          m.getName(), m.getParameterTypes());
    } catch (NoSuchMethodException e) {
      throw new Error(e);
    }
  }
```

```
}
```

For example, `EnhancedStreamDemo` shows what happens when we use our enhanced stream to create distinct methods over `ArrayDeque` and `ConcurrentSkipListSet`.

Listing 3.33: `EnhancedStreamDemo`.

```
//:~ samples/src/main/.../ch03/enhancedstream/EnhancedStreamDemo.java
public class EnhancedStreamDemo {
  public static void main(String... args) {
    System.out.println("Normal ArrayDeque clone() Methods:");
    Stream.of(ArrayDeque.class.getMethods())
        .filter(method -> method.getName().equals("clone"))
        .distinct()
        .forEach(EnhancedStreamDemo::print);

    System.out.println();

    System.out.println("Distinct ArrayDeque:");
    EnhancedStream.of(ArrayDeque.class.getMethods())
        .filter(method -> method.getName().equals("clone"))
        .distinct(HASH_CODE, EQUALS, MERGE)
        .forEach(EnhancedStreamDemo::print);

    System.out.println();

    System.out.println("Normal ConcurrentSkipListSet:");
    Stream.of(ConcurrentSkipListSet.class.getMethods())
        .filter(method -> method.getName().contains("Set"))
        .distinct()
        .sorted(METHOD_COMPARATOR)
        .forEach(EnhancedStreamDemo::print);

    System.out.println();

    System.out.println("Distinct ConcurrentSkipListSet:");
    EnhancedStream.of(ConcurrentSkipListSet.class.getMethods())
        .filter(method -> method.getName().contains("Set"))
        .distinct(HASH_CODE, EQUALS, MERGE)
        .sorted(METHOD_COMPARATOR)
        .forEach(EnhancedStreamDemo::print);
  }
```

```
  private static void print(Method m) {
    System.out.println(
        Stream.of(m.getParameterTypes())
            .map(Class::getSimpleName)
            .collect(Collectors.joining(
                ", ",
                "  " + m.getReturnType().getSimpleName()
                    + " " + m.getName() + "(",
                ")")))
    );
  }

  public static final ToIntFunction<Method> HASH_CODE =
      method -> method.getName().hashCode() +
                    method.getParameterCount();

  public static final BiPredicate<Method, Method> EQUALS =
      (method1, method2) ->
          method1.getName().equals(method2.getName()) &&
              method1.getParameterCount() ==
                  method2.getParameterCount() &&
              Arrays.equals(method1.getParameterTypes(),
                  method2.getParameterTypes());

  public static final BinaryOperator<Method> MERGE =
      (method1, method2) -> {
        if (method1.getReturnType()
                .isAssignableFrom(method2.getReturnType()))
          return method2;
        if (method2.getReturnType()
                .isAssignableFrom(method1.getReturnType()))
          return method1;
        throw new IllegalArgumentException(
            "Conflicting return types " +
                method1.getReturnType().getCanonicalName() +
                " and " +
                method2.getReturnType().getCanonicalName());
      };

  public static final Comparator<Method> METHOD_COMPARATOR =
      Comparator.comparing(Method::getName)
          .thenComparing(method ->
                              Arrays.toString(
```

```
                                    method.getParameterTypes()));
}
```

The output shows that our `EnhancedStream` returns methods unique by name and parameter types. Furthermore, it always keeps the methods with the most derived return type. Thus `ArrayDeque clone()` and `NavigableSet headSet(Object)` are kept, while `Object clone()` and `SortedSet headSet(Object)` are discarded.

Listing 3.34: Output from `EnhancedStreamDemo`.

```
Normal ArrayDeque clone() Methods:
  ArrayDeque clone()
  Object clone()

Distinct ArrayDeque:
  ArrayDeque clone()

Normal ConcurrentSkipListSet:
  NavigableSet descendingSet()
  NavigableSet headSet(Object, boolean)
  SortedSet headSet(Object)
  NavigableSet headSet(Object)
  NavigableSet subSet(Object, boolean, Object, boolean)
  NavigableSet subSet(Object, Object)
  SortedSet subSet(Object, Object)
  NavigableSet tailSet(Object, boolean)
  SortedSet tailSet(Object)
  NavigableSet tailSet(Object)

Distinct ConcurrentSkipListSet:
  NavigableSet descendingSet()
  NavigableSet headSet(Object, boolean)
  NavigableSet headSet(Object)
  NavigableSet subSet(Object, boolean, Object, boolean)
  NavigableSet subSet(Object, Object)
  NavigableSet tailSet(Object, boolean)
  NavigableSet tailSet(Object)
```

In The Java Specialists' Newsletter 274, we read how one can handcraft an `EnhancedStream` decorator. It is mundane and repetitive. We have to implement all the methods and painstakingly delegate them. Our dynamic proxy solution has a single method, `invoke()`, with logic for three different types of method.

There are some disadvantages with our dynamic proxy approach. There might be a slight method-call overhead with some of the proxied methods. For example, each time our return type is `EnhancedStream`, we need to do a map lookup. We will learn in the next chapter how a specialized type of map, which we call a `VTable`, can speed things up. For now, we can bask in the satisfaction that we avoided repeating the same code snippet over and over.

Restrictions with dynamic proxies

Interfaces only, please

One of the biggest restrictions is that we can only proxy interfaces, because they are derived from `java.lang.reflect.Proxy`. Java does not support multiple class inheritance.

When the classes belong to us, we can try to refactor our classes to be interfaces. This will not help us when we are trying to proxy third-party classes.

Tools like CGLib can create dynamic proxies of classes, but are not part of the standard JDK.

`UndeclaredThrowableException`

While the `invoke` method is defined to throw any `Throwable`, we are only allowed to throw checked exceptions that are declared by the method we are proxying.

Listing 3.35: `UndeclaredExceptionThrown`.

```
//:~ samples/src/main/.../ch03/gotchas/UndeclaredExceptionThrown.java
public class UndeclaredExceptionThrown {
  public static void main(String... args) {
    Runnable job = Proxies.castProxy(
        Runnable.class,
        (proxy, method, params) -> {
          // will be wrapped with an UndeclaredThrowableException
          throw new IOException("bad exception");
        });
    job.run();
  }
}
```

When we call `run()` on the `Runnable`, the checked `IOException` is wrapped in an unchecked `UndeclaredThrowableException`. This only applies to checked exceptions that are not part of the method signature of the proxied interface.

Listing 3.36: Bad exception causes the unchecked `UndeclaredThrowableException`.

```
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
```

Mystery `com.sun.proxy.Proxy$0` class

As mentioned, the names of the proxy classes typically follow the naming convention of `com.sun.proxy.Proxy$0` for public interfaces and `packagename.Proxy$0` for package-access interfaces. When an interface is not exported from a module, the proxy is in a non-exported package: for example, `com.sun.proxy.jdk.proxy1`.

Listing 3.37: `ProxyNaming`.

```
//:~ samples/src/main/.../ch03/gotchas/ProxyNaming.java
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(Proxies.simpleProxy(
        intf, null).getClass());
  }
}
```

The output most likely will be what we see in Listing 3.38.

Listing 3.38: Output of `ProxyNaming`.

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

The Java Language Specification does not declare that proxies must follow this naming convention, but this is how OpenJDK currently does it. Note that when we make proxies of public interfaces, the classes are added to the `com.sun.proxy` package. However, when we make proxies of non-public interfaces, the classes have to be in the same package as the interface — in our example, `eu.javaspecialists.books.dynamicproxies.ch03`.

Deeper call stacks

Our call stacks might also contain references to `$Proxy0` and the reflective method invocations. IDEs like IntelliJ IDEA might hide the gory details, as shown in Figure 9.

```
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>
micProxy$Factorial.invoke(RecursiveDynamicProxy.java:49) <5 internal calls>
micProxy$Factorial.invoke(RecursiveDynamicProxy.java:49) <1 internal call>
micProxy.main(RecursiveDynamicProxy.java:33)
```

Figure 9. Stack trace of proxied method call.

Here is an example of a recursive dynamic proxy to calculate factorial. In German, we would call this example *"an den Haaren herbeigezogen"* , meaning that we have pulled it here by its hair. In other words, it is a shockingly impractical way of writing factorial code and is only meant to illustrate how a recursive dynamic proxy call could possibly look. Please don't write code like this!

Listing 3.39: `RecursiveDynamicProxy`.

```
//:~ samples/src/main/.../ch03/gotchas/RecursiveDynamicProxy.java
/**
 * Ridiculous impractical example showing recursive dynamic proxy
 * calls.  Please do not code like this!
 */
public class RecursiveDynamicProxy {
  public static void main(String... args) {
    IntFunction<BigInteger> factorial =
        Proxies.castProxy(IntFunction.class,
            new Factorial());
    System.out.println(factorial.apply(5));
  }

  private static class Factorial implements InvocationHandler {
    private final static MethodKey apply =
        new MethodKey(IntFunction.class, "apply", int.class);
    @Override
    public Object invoke(Object proxy,
                         Method method,
                         Object[] params) throws Throwable {
      if (new MethodKey(method).equals(apply)) {
        int n = (int) params[0];
        if (n == 0) {
          Thread.dumpStack();
          return BigInteger.ONE;
        }
        BigInteger other = (BigInteger) method.invoke(
            proxy, n - 1);
        return BigInteger.valueOf(n).multiply(other);
      } else {
        throw new UnsupportedOperationException(
            "only apply(int) supported");
      }
    }
  }
}
```

The output looks something like Listing 3.40. Some details of the individual stack trace lines were omitted to make the output fit in the book.

Listing 3.40: Deep call stack thanks to dynamic proxies and reflection.

```
java.lang.Exception: Stack trace
    at java.base/java.lang.Thread.dumpStack()
    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$Factorial.invoke()
    at com.sun.proxy.$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$Factorial.invoke()
    at com.sun.proxy.$Proxy0.apply(Unknown Source)
    at java.base/NativeMethodAccessorImpl.invoke0()
    at java.base/NativeMethodAccessorImpl.invoke()
    at java.base/DelegatingMethodAccessorImpl.invoke()
    at java.base/Method.invoke()
etc.
```

The extra lines in our stack traces are most likely only going to be a slight nuisance. If they do cause a `StackOverFlowError`, it would be appropriate to either avoid dynamic proxies in that particular case or to increase the size of the thread stack with the `-Xss` JVM parameter.

Leaky Abstractions

Dynamic proxies are currently treated as ordinary Java classes, albeit dynamically created. This leaks the infrastructure code through to the user. It would be better if it were possible to hide the classes from the user.

At time of writing, JEP 371 is targeted for Java 15 and proposes to add *hidden classes*. These are classes that cannot be used directly by the bytecode of other classes, but rather, would be accessed by framework code via reflection.

Hidden classes have many benefits: Their methods do not appear in stack traces. It is easier to unload them. Access control is more fine grained and strict.

Shared proxy classes

Java tries to minimize the number of dynamic proxy classes it creates. Each time we call

`Proxy.newProxyInstance()`, it checks whether it has previously created a dynamic proxy for the given combination of class loader and target interface. The order in which the interfaces are stored inside the array is significant. If a dynamic proxy class is found, then that is used to create the proxy object. The `InvocationHandler` is always passed to the `java.lang.reflect.Proxy` superclass from which all dynamic proxy classes are derived.

For example, consider `ProxyClassesByInterface` in Listing 3.41. `Collection` extends `Iterable`, so we could consider a proxy consisting of a combination of `Collection` and `Iterable` to be equivalent to a proxy of only `Collection`. However, we see that for every interface combination, unique proxy classes are created.

Listing 3.41: `ProxyClassesByInterface`.

```
//:~ samples/src/main/.../ch03/gotchas/ProxyClassesByInterface.java
public class ProxyClassesByInterface {
  static final InvocationHandler BLANK = (p, m, a) -> null;
  public static void main(String... args) {
    test(Iterable.class, Collection.class); // $Proxy0
    test(Iterable.class, Collection.class); // $Proxy0
    test(Iterable.class); // $Proxy1
    test(Collection.class); // $Proxy2
    test(Collection.class, Iterable.class); // $Proxy3
    test(Iterable.class, Collection.class); // $Proxy0
  }
  private static void test(Class<?>... targetInterfaces) {
    Object proxy = Proxy.newProxyInstance(
        ProxyClassesByInterface.class.getClassLoader(),
        targetInterfaces,
        BLANK
    );
    System.out.println(proxy.getClass());
  }
}
```

Java uses weak references in its cache of dynamic proxy classes to ensure that we do not get class-loader leaks.

Performance of dynamic proxies

Dynamic proxies belong in the realm of infrastructure and framework code. It is highly likely that our dynamic code will be called billions of times. It is thus essential for us to know how to benchmark this type of code and to make sure that it runs quickly enough.

Calling methods on a dynamic proxy may be slower for several reasons:

- Primitive return types and parameters need to be boxed to their wrapper objects. This might add object-creation overhead when primitive values are outside of their cached range.
- Parameters need to be wrapped with an array. This array might not be eliminated with escape analysis if it escapes from the `InvocationHandler.invoke()` method.
- The `Method` class suffers from amnesia — it checks permission on every call. We try to eliminate this cost with the turbo-boost mechanism shown in Listing 3.11.
- All dynamic proxies are derived from a single `java.lang.reflect.Proxy` class, which stores the reference to our `InvocationHandler`. When a field only ever holds one implementation of an interface, the call sites that invoke methods on that field's value are monomorphic, as opposed to polymorphic. The HotSpot compiler has optimizations for monomorphism and bimorphism. For monomorphism, the compiler can inline the method call at run time. However, since a dynamic proxy can hold one of many implementations of `InvocationHandler`, HotSpot might not be able to inline the method call to `invoke()`, thus making it megamorphic. For more information about polymorphism performance, please see "Polymorphism Performance Mysteries" and "Polymorphism Performance Mysteries Explained" in The Java Specialists' Newsletter.

Programmers wishing to write performance-critical code need to know how to benchmark code. The recommended tool for running benchmarks is the Java Microbenchmark Harness (JMH).

For our experiment, we define a `Worker` interface with a two methods: `increment()` and `consumeCPU()`. The `increment()` method returns a `long`, which will be converted to a `Long` object through autoboxing. The `consumeCPU()` method returns `void`, thus no objects will need to be created.

Listing 3.42: `Worker`.

```
//:~ benchmark/src/main/.../ch03/benchmarks/Worker.java
public interface Worker {
  long increment();
  void consumeCPU();
}
```

Our `RealWorker` implements `Worker`. In the `increment()` method, it increments a field of type `long` and returns the new value. In the `consumeCPU()` method, it uses the JMH `Blackhole` class to use up a couple of CPU cycles.

Listing 3.43: `RealWorker`.

```
//:~ benchmark/src/main/.../ch03/benchmarks/RealWorker.java
public class RealWorker implements Worker {
  private long counter = 0;
  @Override
  public long increment() {
    return counter++;
  }
  @Override
  public void consumeCPU() {
    Blackhole.consumeCPU(2);
  }
}
```

Our `ProxyWorker` follows the proxy pattern and points to the `RealWorker`.

Listing 3.44: `ProxyWorker`.

```
//:~ benchmark/src/main/.../ch03/benchmarks/ProxyWorker.java
public class ProxyWorker implements Worker {
  private final RealWorker worker = new RealWorker();
  @Override
  public long increment() {
    return worker.increment();
  }
  @Override
  public void consumeCPU() {
    worker.consumeCPU();
  }
}
```

In our `MethodCallBenchmark`, we have five ways of calling `increment()` and `consumeCPU()`:

- `directCall` calls the methods directly on the `RealWorker` instance.
- `staticProxy` calls the methods on the `ProxyWorker` instance, which then delegates the calls to the `RealWorker`.
- `dynamicProxyThenDirectCall` creates a dynamic proxy of `Worker`, but delegates the calls directly to the `RealWorker` methods.
- `dynamicProxyThenReflectiveCall` (turbo) is our dynamic proxy of `Worker`, which delegates the methods to the `RealWorker` instance using a reflective call with the given `method` parameter. The dynamic proxy instance is turbo-boosted.

- `dynamicProxyThenReflectiveCall` (no turbo) is the same as the previous, except without turbo-boosting.

Listing 3.45: `MethodCallBenchmark`.

```
//:~ benchmark/src/main/.../ch03/benchmarks/MethodCallBenchmark.java
@Fork(value = 3, jvmArgsAppend = "-XX:+UseParallelGC")
@Warmup(iterations = 5, time = 3)
@Measurement(iterations = 10, time = 3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class MethodCallBenchmark {
  // direct call to RealWorker
  private final RealWorker realWorker = new RealWorker();

  // static proxies
  private final Worker staticProxy = new ProxyWorker();

  //   dynamic proxies
  private final Worker dynamicProxyDirectCallIncrement =
      Proxies.castProxy(Worker.class,
          (proxy, method, args) -> realWorker.increment());
  private final Worker dynamicProxyDirectCallConsumeCPU =
      Proxies.castProxy(Worker.class,
          (proxy, method, args) -> {
            realWorker.consumeCPU();
            return null;
          });
  private final Worker dynamicProxyReflectiveCall =
      Proxies.castProxy(Worker.class,
          (proxy, method, args) ->
              method.invoke(realWorker, args));

  @Benchmark
  public long directCallIncrement() {
    return realWorker.increment();
  }
  @Benchmark
  public long staticProxyIncrement() {
    return staticProxy.increment();
  }
  @Benchmark
```

```
  public long dynamicProxyDirectCallIncrement() {
    return dynamicProxyDirectCallIncrement.increment();
  }
  @Benchmark
  public long dynamicProxyReflectiveCallIncrement() {
    return dynamicProxyReflectiveCall.increment();
  }

  @Benchmark
  public void directCallConsumeCPU() {
    realWorker.consumeCPU();
  }
  @Benchmark
  public void staticProxyConsumeCPU() {
    staticProxy.consumeCPU();
  }
  @Benchmark
  public void dynamicProxyDirectCallConsumeCPU() {
    dynamicProxyDirectCallConsumeCPU.consumeCPU();
  }
  @Benchmark
  public void dynamicProxyReflectiveCallConsumeCPU() {
    dynamicProxyReflectiveCall.consumeCPU();
  }
}
```

The results of our `MethodCallBenchmark` come from running it on a MacBook Pro (2018) i9 with six cores on Java 14+36-1461 with the throughput collector. To get more consistent results, we turned off the CPU Turbo using Turbo Boost Switcher Pro and ran all the tests at 2.9GHz. Furthermore, we only show the fastest results, rather than averages. All units are nanoseconds/operation and results are rounded to two significant digits. Detailed results are available on request.

Results for `increment()`

Our results table shows that the `dynamicProxyDirectCall` is approximately 2.1 nanoseconds slower than `staticProxy` in Java 14. In Java 13 the difference was approximately 4.6 nanoseconds, since each primitive `long` returned by `increment()` needed to be boxed to a `Long` object, churning 24 bytes per method call.

Calling the `Method` object via reflection in the `dynamicProxyReflectiveCall` adds another 4.1 nanoseconds and if we turn off the method turbo boost, this increases by a further 2.3 nanoseconds. In all versions of Java that we tried, `dynamicProxyReflectiveCall` allocated 24 bytes for the method call for the returned Integer.

Benchmark `increment()`	best ns/op	B/op EA on / off
directCall	2.9	0 / 0
staticProxy	3.5	0 / 0
dynamicProxyDirectCall	5.6	0 / 24
dynamicProxyReflectiveCall (turbo)	9.7	24 / 24
dynamicProxyReflectiveCall (no turbo)	12	24 / 24

Results for `consumeCPU()`

The results for `consumeCPU()` are much closer, because we do not have the cost of boxing the primitive `long` to the `Long` wrapper object. The `dynamicProxyDirectCall` is approximately 1.1 nanoseconds slower than `staticProxy`.

Calling the `Method` object via reflection in the `dynamicProxyReflectiveCall` adds only one more nanosecond and if we turn off the method turbo boost, this increases by a further 3.4 nanoseconds.

Benchmark `consumeCPU()`	best ns/op
directCall	4.8
staticProxy	5.5
dynamicProxyDirectCall	6.6
dynamicProxyReflectiveCall (turbo)	7.6
dynamicProxyReflectiveCall (no turbo)	11

Summary of benchmark results

A method call overhead of 6.2 nanoseconds for `increment()` and 2.1 nanoseconds for `consumeCPU()` would seldom be noticed with a typical business-logic method.

Still, it is good to be aware that there is a slightly higher method-call cost and perhaps to avoid dynamic proxies for performance-sensitive code, especially if we are calling the proxied methods in a tight loop using reflection.

CHAPTER FOUR

Dynamic Decorator

A decorator attaches additional responsibilities to an object dynamically without directly subclassing it. As we saw in Chapter 1, the decorator is used in the Java I/O design to add Gzip, buffering, etc. functionality to a bare I/O stream. In this chapter, we will see how to create a dynamic decorator.

Decorator design pattern

Up to now, we have used the dynamic proxies of Java to implement the proxy design pattern. We can also use the same mechanism to create decorators or filters.

Figure 10 shows the classical structure of decorator design patterns as described in the Gang-of-Four book.

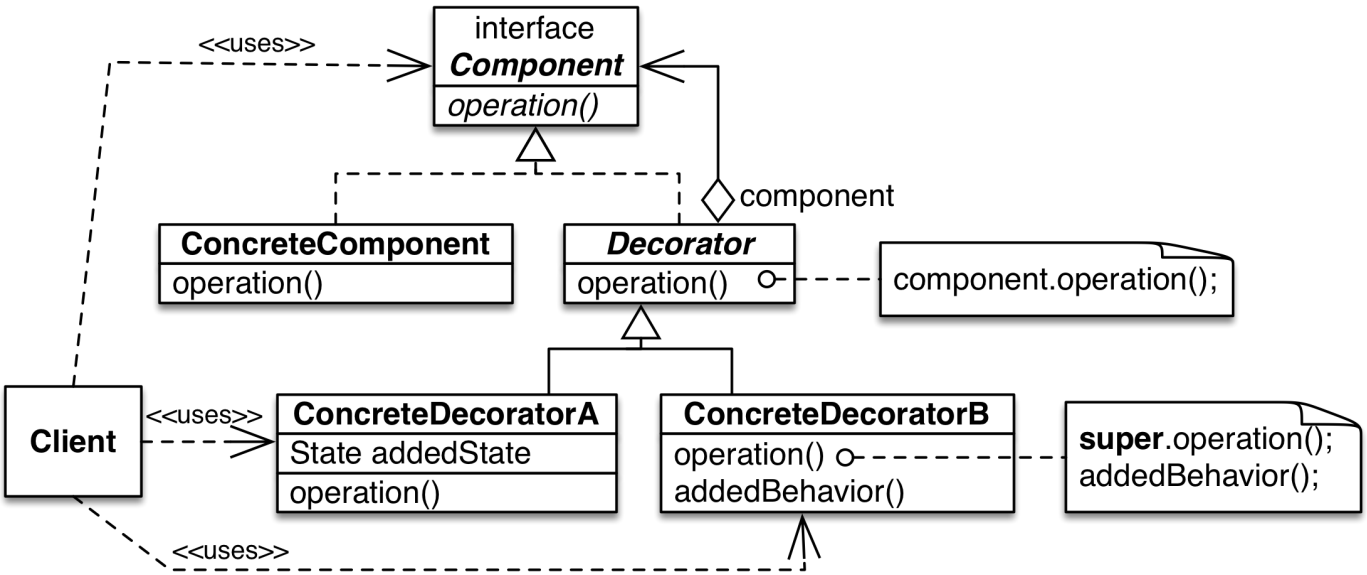


Figure 10. Decorator structure.

Decorator is also known by the name of “filter”. In my mind, a decorator adds, whereas a filter takes away. If we want a truly immutable collection, we could create a filter interface that does not contain any mutator methods. Take for example our `ImmutableIterable` in Listing 4.1. In it, we leave out the `iterator()` method, as the `Iterator` contains the `remove()` method. We could of course also have our `iterator()` method return a filtered `ImmutableIterator`, but I will leave that as an exercise for the reader.

Listing 4.1: `ImmutableIterable`.

```
//:~ samples/src/main/.../ch04/immutablecollection/ImmutableIterable.java
public interface ImmutableIterable<E> {
  void forEach(Consumer<? super E> action);
  Spliterator<E> spliterator();

  // mutator method from Iterable filtered away
  //    Iterator<E> iterator();
}
```

Our `ImmutableCollection` also filters out any method that may change the collection. We added a default interface method `printAll()`. Dynamic proxies implement all public methods of the interface, even default methods. In Java, we can call superclass methods from within a subclass using the `super` keyword, but we cannot do so from the outside. For example, when a class overrides `hashCode()`, there is no way for a user of that class to call the default `Object.hashCode()` method. For this reason, Java provides a `System.identityHashCode()` in case we need to generate the same `hashCode()` as `Object.hashCode()` would produce. This gets us in trouble with default interface methods. If the method had not been overridden inside the generated dynamic proxy, we could call the default interface method and it would route us to the interface's method. Fortunately there is a hack that allows us to call the default interface methods on interfaces, as we will see when we look at the `VTable`.

Listing 4.2: `ImmutableCollection`.

```
//:~
samples/src/main/.../ch04/immutablecollection/ImmutableCollection.java
public interface ImmutableCollection<E>
    extends ImmutableIterable<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);
  }

  // mutator methods from Collection filtered away
  //    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();
}
```

To show how we could use this to filter existing collections, let us write a `HandcodedFilter`. Each of the methods inside `ImmutableCollection` is delegated to the enclosed `Collection`. You will probably agree that writing code like this is rather pedestrian.

Listing 4.3: `HandcodedFilter`.

```
//:~ samples/src/main/.../ch04/immutablecollection/HandcodedFilter.java
public class HandcodedFilter<E>
    implements ImmutableCollection<E> {
  private final Collection<E> c;

  public HandcodedFilter(Collection<E> c) {
```

```
    this.c = c;
  }

  @Override
  public int size() {
    return c.size();
  }
  @Override
  public boolean isEmpty() {
    return c.isEmpty();
  }
  @Override
  public boolean contains(Object o) {
    return c.contains(o);
  }
  @Override
  public Object[] toArray() {
    return c.toArray();
  }
  @Override
  public <T> T[] toArray(T[] a) {
    return c.toArray(a);
  }
  @Override
  public <T> T[] toArray(IntFunction<T[]> generator) {
    return c.toArray(generator);
  }
  @Override
  public boolean containsAll(Collection<?> c) {
    return this.c.containsAll(c);
  }
  @Override
  public Spliterator<E> spliterator() {
    return c.spliterator();
  }
  @Override
  public Stream<E> stream() {
    return c.stream();
  }
  @Override
  public Stream<E> parallelStream() {
    return c.parallelStream();
  }
```

```
  @Override
  public void forEach(Consumer<? super E> action) {
    c.forEach(action);
  }
}
```

Here is how the `HandcodedFilter` could be used to filter out those methods that we do not want to support.

Listing 4.4: `HandcodedFilterDemo`.

```
//:~
samples/src/main/.../ch04/immutablecollection/HandcodedFilterDemo.java
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("Are there names? " + !names.isEmpty());
System.out.println("Printing the names:");
names.forEach(System.out::println);
System.out.println("Class: " +
                       names.getClass().getSimpleName());

names.printAll();
```

When we run the code, we see the output in Listing 4.5.

Listing 4.5: Output from `HandcodedFilterDemo`.

```
eu.javaspecialists....HandcodedFilter@1b4fb997
Is Mary in? true
Are there names? true
Printing the names:
Peter
Paul
Mary
Class: HandcodedFilter
Peter
Paul
Mary
```

We forgot to delegate the `toString()` method, so in the first line of output we see the default `toString()` implementation from `Object`. Forgetting `toString()`, `equals()`, and `hashCode()` is a common mistake when delegating to other objects, such as with the proxy, decorator, adapter, and composite patterns.

Using dynamic proxy to implement filter

Instead of writing the decorator by hand, we can abuse the good will of Java's dynamic proxy and use it to create a dynamic filter/decorator.

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.

Listing 4.6 shows our `FilterHandler`. We have written our own `VTable` to implement a virtual-method table, a concept used in object orientation to route virtual methods to the correct receiver. I will explain the `VTable` and the `VTableHandler` in great detail later. For now, it is enough to know that the `VTable` is used to match methods between unrelated interfaces, including default interface methods. The `VTableHandler` uses the chain-of-responsibility design pattern to link multiple `VTable` lookups. As part of our chain construction, we also verify that all the methods in the filter interface have some handler in the chain.

Listing 4.6: `FilterHandler`.

```
//:~ core/src/main/.../handlers/FilterHandler.java
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);
  }

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

We can now create our dynamic filter with the `filter()` method.

Listing 4.7: `Proxies.filter()` *method.*

```
//:~ core/src/main/.../Proxies.java
public static <F> F filter(
    Class<? super F> filter, Object component) {
  Objects.requireNonNull(component, "component==null");
  return castProxy(filter,
      new FilterHandler(filter, component));
}
```

We can use our dynamically created class to filter out the methods that we do not wish to support.

Listing 4.8: `DynamicFilterDemo`.

```
//:~ samples/src/main/.../ch04/immutablecollection/DynamicFilterDemo.java
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("Are there names? " + !names.isEmpty());
System.out.println("Printing the names:");
names.forEach(System.out::println);
System.out.println("Class: " +
                       names.getClass().getSimpleName());

names.printAll();
```

When we run the code, we see the output in Listing 4.9.

Listing 4.9: Output from `DynamicFilterDemo`.

```
[Peter, Paul, Mary]
Is Mary in? true
Are there names? true
Printing the names:
Peter
Paul
Mary
Class: $Proxy0
Peter
Paul
Mary
```

`InfiniteRandomDouble` decorator

Imagine that we wish to generate an infinite stream of random `Double` instances, so we write our `InfiniteRandomDouble` class in Listing 4.10.

Listing 4.10: `InfiniteRandomDouble`.

```
//:~ samples/src/main/.../ch04/infiniterandom/InfiniteRandomDouble.java
/**
 * No relation to the Iterator interface.
 */
public class InfiniteRandomDouble {
  public boolean hasNext() {
    return true;
  }
  public Double next() {
    return ThreadLocalRandom.current().nextDouble();
  }
}
```

At a later stage, we realize that it would be great if we could decorate this with the `Iterator<Double>`. Instead of changing the `InfiniteRandomDouble` class to implement the interface, we can create an `Iterator` with `Proxies.filter(Iterator.class, new InfiniteRandomDouble())`. If we are running inside a JPMS module, we need to explicitly open up the `java.base` module's `java.util` package. If our module is `eu.javaspecialists.books.dynamicproxies`, then we would need to use the JVM parameters `--add-opens java.base/java.util=eu.javaspecialists.books.dynamicproxies`. Otherwise, the `VTable` will not include the default methods.

Listing 4.11: `DefaultMethodCallDemo`.

```
//:~ samples/src/main/.../ch04/infiniterandom/DefaultMethodCallDemo.java
/**
 * If we are inside a module, we need to explicitly open the
 * java.util package inside the java.base module to our module.
 * We can do this with the JVM parameters --add-opens \
 * java.base/java.util=eu.javaspecialists.books.dynamicproxies
 */
public class DefaultMethodCallDemo {
  public static void main(String... args) {
    Iterator<Double> filter = Proxies.filter(
        Iterator.class, new InfiniteRandomDouble());

    System.out.println(filter.hasNext());
    System.out.println(filter.next());

    try {
      filter.remove();
      throw new AssertionError(
          "Expected an UnsupportedOperationException");
    } catch (UnsupportedOperationException expected) {
      System.err.println(expected);
    }

    System.out.println("Iterating forever ...");

    filter.forEachRemaining(System.out::println);
  }
}
```

Running without the `--add-opens` parameters throws an `IllegalArgumentException`, because the default methods `remove()` and `forEachRemaining()` can only be included when the `.java.util` package is open for deep reflection.

Listing 4.12: Output from `DynamicFilterDemo` *without* `--add-opens`.

```
Exception in thread "main" java.lang.IllegalArgumentException: \
    Unhandled methods: [ \
      public default void java.util.Iterator.remove(), \
      public default void java.util.Iterator.forEachRemaining( \
        java.util.function.Consumer)]
  at eu.javaspecialists.books.dynamicproxies/...
```

However, when we run the `DefaultMethodCallDemo` with `--add-opens java.base/java.util=eu.javaspecialists.books.dynamicproxies`, it works as expected.

Listing 4.13: Output from `DynamicFilterDemo` *with* `--add-opens`.

```
true
0.9833487941165752
java.lang.UnsupportedOperationException: remove
Iterating forever ...
0.07265625125554698
0.49716358229383684
0.44842977216243907
0.44363950828146614
0.5560995018129454
0.9779642433900788
0.7941409326858004
... etc.
```

The rest of this chapter shows the gory details of how the `VTable` works, together with our `ChainedInvocationHandler`. These utilities are used in the next pattern, but feel free to skip the rest of this chapter and perhaps come back to it at a later stage.

`VTable`

There is no inheritance relationship between the `ImmutableCollection` interface and `Collection`. However, we still need to match methods from the interface through to the decorated object. To do this, we have created a `VTable`, which returns the matching method for a given interface method. It is rather long and complex, and we first need to cover a few items:

1. `ParameterTypesFetcher` helps to increase performance of matching methods.
2. `MethodKey` compares methods by name and parameter types.
3. `VTable` matches target and receiver methods, including default interface methods.
4. `VTable.Builder` is an *Effective Java* builder pattern for constructing the `VTable`.

5. `VTables` is a façade for building the most common `VTable` configurations.

`ParameterTypesFetcher`

A significant cost in matching methods is the call to `getParameterTypes()`, which clones the internal `Class<?>[]` every time we call it. The `ParameterTypesFetcher` gives us direct access to the array inside. We can enable this performance optimization with the `-Deu.javaspecialists.books.dynamicproxies.util.ParameterTypesFetcher.enabled=true` JVM parameter.

When we enable fast parameter fetching, calling `getParameterTypes()` on a `Method` object will return the actual `Class<?>[]` stored inside the `Method` object. Treat this with care, and do not change the contents.

We use the strategy design pattern to decide how to return the parameter types. When the `ParameterTypesFetcher` class is loaded, we bind the `PARAMETER_FETCHER` to the correct type of strategy. Since we only ever use one strategy within one JVM, the HotSpot compiler can optimize the methods aggressively, possibly inlining the code and thus eliminating the method-call overhead.

Listing 4.14: `ParameterTypesFetcher`.

```
//:~ core/src/main/.../util/ParameterTypesFetcher.java
/**
 * Fast fetching of parameter types array is disabled by default
 * and can be enabled with
 * -Deu.javaspecialists.books.dynamicproxies.util. \
 * ParameterTypesFetcher.enabled=true
 */
public final class ParameterTypesFetcher {
  private final static ParameterFetcher PARAMETER_FETCHER =
      Boolean.getBoolean(
          ParameterTypesFetcher.class.getName() + ".enabled") ?
          new FastParameterFetcher() :
          new NormalParameterFetcher();

  /**
   * Warning: When "fast parameter fetching" is enabled, the
   * array returned is the actual array stored inside the Method
   * object.  Do not change it!
   */
  public static Class<?>[] get(Method method) {
    return PARAMETER_FETCHER.getParameterTypes(method);
  }
```

```
  @FunctionalInterface
  private interface ParameterFetcher {
    Class<?>[] getParameterTypes(Method method);
  }

  /**
   * Returns the parameter types by calling getParameterTypes()
   * on the method parameter.  This should always work.
   */
  private static class NormalParameterFetcher
      implements ParameterFetcher {
    @Override
    public Class<?>[] getParameterTypes(Method method) {
      return method.getParameterTypes(); // clones the array
    }
  }

  /**
   * Creates a VarHandle pointing directly to the private field
   * parameterTypes stored inside Method. Since the VarHandle is
   * declared as final static, the cost of reading it is the
   * same as reading an ordinary field.
   */
  private static class FastParameterFetcher
      implements ParameterFetcher {
    private final static VarHandle METHOD_PARAMETER_TYPES;

    static {
      try {
        METHOD_PARAMETER_TYPES = MethodHandles.privateLookupIn(
            Method.class, MethodHandles.lookup()
        ).findVarHandle(Method.class, "parameterTypes",
            Class[].class);
      } catch (ReflectiveOperationException e) {
        throw new Error(e);
      }
    }
    @Override
    public Class<?>[] getParameterTypes(Method method) {
      return (Class<?>[]) METHOD_PARAMETER_TYPES.get(method);
    }
  }
}
```

MethodKey

`MethodKey` in Listing 4.15 is used to compare `Method` objects based on their names and parameter types. It does not compare the class each is defined in or its return type. We use `MethodKey` as key for maps when we want to match methods.

Listing 4.15: `MethodKey`.

```
//:~ core/src/main/.../util/MethodKey.java
/**
 * MethodKey is used to compare Methods by name and parameter
 * types. It has equals(), hashCode(), compareTo() and toString()
 * implemented.  We can use it as a key in a map. MethodKey does
 * not know the return type of the method.
 */
public final class MethodKey implements Comparable<MethodKey> {
  private final String name;
  private final Class<?>[] paramTypes;

  public MethodKey(Method method) {
    name = method.getName();
    paramTypes = ParameterTypesFetcher.get(method);
  }

  public MethodKey(Class<?> clazz, String name,
                   Class<?>... paramTypes) {
    try {
      // check that method exists in the given class
      var method = clazz.getMethod(name, paramTypes);
      // method names are all interned in the JVM
      this.name = method.getName();
      this.paramTypes = Objects.requireNonNull(paramTypes);
    } catch (NoSuchMethodException e) {
      throw new IllegalArgumentException(e);
    }
  }

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof MethodKey)) {
      return false;
    }
    // name and paramTypes cannot be null
    var other = (MethodKey) obj;
```

```
    return name == other.name && // method names are interned
               equalParamTypes(paramTypes, other.paramTypes);
  }

  /**
   * We compare classes using == instead of .equals().  We know
   * that the arrays will never be null.  We can thus avoid some
   * of the checks done in Arrays.equals().
   */
  private boolean equalParamTypes(Class<?>[] params1,
                                  Class<?>[] params2) {
    if (params1.length == params2.length) {
      for (int i = 0; i < params1.length; i++) {
        if (params1[i] != params2[i])
          return false;
      }
      return true;
    }
    return false;
  }

  /**
   * Convenience method for quickly matching a Method to our
   * MethodKey.
   */
  public boolean matches(Method method) {
    return name == method.getName() &&
               equalParamTypes(paramTypes,
                   ParameterTypesFetcher.get(method));
  }

  @Override
  public int hashCode() {
    return name.hashCode() + paramTypes.length;
  }

  @Override
  public int compareTo(MethodKey that) {
    int result = this.name.compareTo(that.name);
    if (result != 0) return result;
    return Arrays.compare(this.paramTypes,
        that.paramTypes,
        Comparator.comparing(Class::getName));
```

```
  }

  @Override
  public String toString() {
    return Stream.of(paramTypes)
                 .map(Class::getName)
                 .collect(Collectors.joining(", ",
                     name + "(", ")"));
  }
}
```

`VTable` and `VTable.Builder`

The `VTable` creates a custom hash table for looking up methods based on name and parameter type. Normal methods are discovered using `lookup(Method)`, whereas default interface methods are discovered with `lookupDefaultMethod(Method)`.

We create instances of the `VTable` with the `VTable.Builder`, using the *Effective Java* builder.

Further comments are shown inside the `VTable` source code. I encourage you to spend a bit of time reading through the code, but if it is too difficult, feel free to skip it for now and come back later.

Listing 4.16: `VTable` *and* `VTable.Builder`.

```
//:~ core/src/main/.../util/VTable.java
/**
 * The purpose of the VTable is as a fast mapper between
 * interface methods and receiver class methods. To minimize
 * object creation, we avoid a normal HashMap, choosing rather
 * to store all the elements in equally sized arrays. We also
 * try to minimize clashes in the table by hashing on the method
 * name + number of parameters. Furthermore, we remember whether
 * a name was distinct in the table. If it was, we do not check
 * that the parameters match.
 * <p>
 * The most important methods in terms of performance are
 * lookup() and lookupDefaultMethod().
 */
public final class VTable {
  private final Method[] entries;
  private final Class<?>[][] paramTypes;
  private final boolean[] distinctName;
  private final MethodHandle[] defaultMethods;
  private final int size;
```

```
  private final int mask;

  /**
   * Builds the VTable according to the collection of methods.
   * The input for this constructor is created in the Builder
   * class.
   *
   * @param methods               all the methods that need to
   *                              be included in this VTable
   * @param distinctMethodNames   names of methods that have
   *                              not been overloaded
   * @param includeDefaultMethods whether or not we want to
   *                              include default interface
   *                              methods
   */
  private VTable(Collection<Method> methods,
                 Set<String> distinctMethodNames,
                 boolean includeDefaultMethods) {
    this.size = methods.size();
    mask = Math.max(
        (-1 >>> Integer.numberOfLeadingZeros(size * 4 - 1)),
        127);
    entries = new Method[mask + 1];
    paramTypes = new Class<?>[entries.length][];
    distinctName = new boolean[entries.length];
    defaultMethods = new MethodHandle[entries.length];
    for (var method : methods) {
      put(method,
          distinctMethodNames.contains(method.getName()),
          includeDefaultMethods);
    }
    methods.forEach(MethodTurboBooster::boost);
  }

  /**
   * Looks up the method in the VTable.  Returns null if it is
   * not found.
   */
  public Method lookup(Method method) {
    int index = findIndex(method);
    return index < 0 ? null : entries[index];
  }
```

```
  /**
   * Looks up the default interface method in the VTable.
   * Returns null if it is not found.
   */
  public MethodHandle lookupDefaultMethod(Method method) {
    int index = findIndex(method);
    return index < 0 ? null : defaultMethods[index];
  }

  /**
   * Returns the number of entries in the VTable.  The size does
   * not change and is fixed at construction.
   */
  public int size() {
    return size;
  }

  /**
   * Returns a stream of Method objects from this VTable.  Used
   * by the ChainedInvocationHandler to verify that all methods
   * in the target have been covered by the various VTables.
   */
  public Stream<Method> stream() {
    return Stream.of(entries).filter(Objects::nonNull);
  }

  /**
   * Returns a stream of Method objects for which we have
   * default interface methods in this VTable.  Used by the
   * ChainedInvocationHandler to verify that all methods in the
   * target have been covered by the various VTables.
   */
  public Stream<Method> streamDefaultMethods() {
    // Heinz: First time I've found a use case for iterate()
    return IntStream.iterate(0, i -> i < entries.length,
        i -> i + 1)
               .filter(i -> defaultMethods[i] != null)
               .mapToObj(i -> entries[i]);
  }

  /**
   * Finds a free position for the entry and then inserts the
   * values into the arrays entries, paramTypes, distinctName,
```

```
     * and optionally, defaultMethods.  Duplicate methods are not
     * allowed and will throw an IllegalArgumentException.
     */
    private void put(Method method, boolean distinct,
                     boolean includeDefaultMethods) {
      int index = findIndex(method);
      if (index >= 0)
        throw new IllegalArgumentException(
            "Duplicate method found: " + new MethodKey(method));
      index = ~index; // flip the bits again to find empty space
      entries[index] = method;
      paramTypes[index] = ParameterTypesFetcher.get(method);
      distinctName[index] = distinct;
      if (includeDefaultMethods && method.isDefault()) {
        defaultMethods[index] = getDefaultMethodHandle(method);
      }
    }

    /**
     * Returns a MethodHandle for the default interface method
     * if it is declared and the module is open to our module;
     * null otherwise.
     */
    private MethodHandle getDefaultMethodHandle(Method method) {
      try {
        Class<?> target = method.getDeclaringClass();
        if (isTargetClassInOpenModule(target)) {
          // Thanks Thomas Darimont for this idea
          MethodHandles.Lookup lookup = MethodHandles.lookup();
          return MethodHandles.privateLookupIn(target, lookup)
                      .unreflectSpecial(method, target)
                      // asSpreader() avoids having to call
                      // invokeWithArguments() and is about 10x
                      // faster
                      .asSpreader(Object[].class,
                          method.getParameterCount());
        }
        return null;
      } catch (IllegalAccessException e) {
        throw new IllegalArgumentException(e);
      }
    }
```

```
  /**
   * Returns true if the method is in an open module; false
   * otherwise.  For example, if our VTable is inside module
   * "eu.javaspecialists.books.dynamicproxies" and we want to
   * use default methods of interfaces in java.util, we need
   * to explicitly open that module with --add-opens \
   * java.base/java.util=eu.javaspecialists.books.dynamicproxies
   */
  private boolean isTargetClassInOpenModule(Class<?> target) {
    Module targetModule = target.getModule();
    String targetPackage = target.getPackageName();
    Module ourModule = VTable.class.getModule();
    return targetModule.isOpen(targetPackage, ourModule);
  }

  /**
   * Returns the index of the method; negative value if it was
   * not found.  Once a method with this name is found, we check
   * if the method was overloaded.  If it was not, then we
   * return the offset immediately.
   */
  private int findIndex(Method method) {
    int offset = offset(method);
    Class<?>[] methodParamTypes = null;
    Method match;
    while ((match = entries[offset]) != null) {
      if (match.getName() == method.getName()) {
        if (distinctName[offset]) return offset;
        if (methodParamTypes == null)
          methodParamTypes = ParameterTypesFetcher.get(method);
        if (matches(paramTypes[offset], methodParamTypes))
          return offset;
      }
      offset = (offset + 1) & mask;
    }
    // Could not find the method, returning a negative value
    return ~offset;
    // (By flipping the bits again, we know what the first
    // available index in our elements array is)
  }

  /**
   * Returns the initial offset for the method, based on method
```

```
     * name and number of parameters.
     */
    private int offset(Method method) {
      return (method.getName().hashCode() +
                  method.getParameterCount()) & mask;
    }

    /**
     * Fast array comparison for parameter types.  Classes can be
     * compared with == instead of equals().
     */
    private boolean matches(Class<?>[] types1, Class<?>[] types2) {
      if (types1.length != types2.length) return false;
      for (int i = 0; i < types1.length; i++) {
        if (types1[i] != types2[i]) return false;
      }
      return true;
    }

    public static class Builder {
      private static final BinaryOperator<Method> MOST_SPECIFIC =
          (method1, method2) -> {
            var r1 = method1.getReturnType();
            var r2 = method2.getReturnType();
            if (r2.isAssignableFrom(r1)) {
              return method1;
            }
            if (r1.isAssignableFrom(r2)) {
              return method2;
            } else {
              throw new IllegalStateException(
                  method1 + " and " + method2 +
                      " have incompatible return types");
            }
          };
      private static final Method[] OBJECT_METHODS;
      static {
        try {
          OBJECT_METHODS = new Method[] {
              Object.class.getMethod("toString"),
              Object.class.getMethod("hashCode"),
              Object.class.getMethod("equals", Object.class),
          };
```

```
      } catch (NoSuchMethodException e) {
        throw new Error(e);
      }
    }
    private final Map<MethodKey, Method> receiverClassMap;
    private List<Class<?>> targetInterfaces = new ArrayList<>();
    private boolean includeObjectMethods = true;
    private boolean includeDefaultMethods = false;
    private boolean ignoreReturnTypes = false;

    /**
     * @param receiver The class that receives the actual
     *                 method calls.  Does not have to be
     *                 related to the dynamic proxy interfaces.
     */
    public Builder(Class<?> receiver) {
      receiverClassMap = createPublicMethodMap(receiver);
    }

    /**
     * Methods equals(Object), hashCode() and toString() are not
     * added to the VTable.
     */
    public Builder excludeObjectMethods() {
      this.includeObjectMethods = false;
      return this;
    }

    public Builder includeDefaultMethods() {
      this.includeDefaultMethods = true;
      return this;
    }

    public Builder ignoreReturnTypes() {
      this.ignoreReturnTypes = true;
      return this;
    }

    /**
     * One of the target interfaces that we wish to map methods
     * to.
     */
    public Builder addTargetInterface(Class<?> targetIntf) {
```

```
      if (!targetIntf.isInterface())
        throw new IllegalArgumentException(
            targetIntf.getCanonicalName() +
                " is not an interface");
      this.targetInterfaces.add(targetIntf);
      return this;
    }

    public VTable build() {
      // Build collection of all methods from the target
      // interfaces, as well as the three Object methods if
      // included: toString(), equals(Object), hashCode()
      Collection<Method> allMethods =
          targetInterfaces.stream()
              .flatMap(clazz -> Stream.of(clazz.getMethods()))
              .collect(Collectors.toList());
      if (includeObjectMethods) {
        for (Method method : OBJECT_METHODS) {
          allMethods.add(method);
        }
      }

      // Reduce the methods to the most derived return type when
      // two have the same name and parameter types.  If we find
      // two methods with incompatible return types, e.g. Integer
      // and String, then throw an exception.
      Map<MethodKey, Method> targetMethods =
          allMethods.stream()
              .collect(Collectors.toUnmodifiableMap(
                  MethodKey::new,
                  Function.identity(),
                  MOST_SPECIFIC));

      // Find all those methods that have a unique name, thus no
      // overloading.  This will speed up matching.
      Set<String> distinctMethodNames =
          targetMethods.values().stream()
              .collect(Collectors.groupingBy(Method::getName,
                  Collectors.counting()))
              .entrySet()
              .stream()
              .filter(entry -> entry.getValue() == 1L)
              .map(Map.Entry::getKey)
```

```
            .collect(Collectors.toSet());

    // Lastly we only include those methods that are also in
    // the receiverClassMap and where the return type is
    // assignment compatible with the target method.
    Collection<Method> matchedMethods =
        targetMethods.entrySet().stream()
            .map(this::filterOnReturnType)
            .filter(Objects::nonNull)
            .collect(Collectors.toList());

    return new VTable(matchedMethods, distinctMethodNames,
        includeDefaultMethods);
  }

  /**
   * Ensure that receiverClassMap method return type of method
   * can be cast to the target method return type.
   */
  private Method filterOnReturnType(
      Map.Entry<MethodKey, Method> entry) {
    var receiverMethod = receiverClassMap.get(entry.getKey());
    if (receiverMethod != null) {
      if (ignoreReturnTypes) return receiverMethod;
      var targetReturn = entry.getValue().getReturnType();
      var receiverReturn = receiverMethod.getReturnType();
      if (targetReturn.isAssignableFrom(receiverReturn))
        return receiverMethod;
    }
    return null;
  }

  /**
   * Our reverse map allows us to find the methods in the
   * component that we are decorating.
   */
  private Map<MethodKey, Method> createPublicMethodMap(
      Class<?> clazz) {
    Map<MethodKey, Method> map = new HashMap<>();
    addTrulyPublicMethods(clazz, map);
    return map;
  }
```

```
    /**
     * Recursively add all "truly" public methods from this
     * class,
     * superclasses and interfaces.  By "truly" we mean methods
     * that are public and which are defined inside public
     * classes.
     */
    private void addTrulyPublicMethods(
        Class<?> clazz, Map<MethodKey, Method> map) {
      if (clazz == null) return;
      for (var method : clazz.getMethods()) {
        if (isTrulyPublic(method)) {
          MethodKey key = new MethodKey(method);
          map.merge(key, method, MOST_SPECIFIC);
        }
      }
      for (var anInterface : clazz.getInterfaces()) {
        addTrulyPublicMethods(anInterface, map);
      }
      addTrulyPublicMethods(clazz.getSuperclass(), map);
    }

    /**
     * Truly public are those methods where the declaring class
     * is also public, hence the bitwise AND.
     */
    private boolean isTrulyPublic(Method method) {
      return Modifier.isPublic(
          method.getModifiers()
              & method.getDeclaringClass().getModifiers());
    }
  }
}
```

`VTables` façade

The most common use cases of `VTable` is to map methods between unrelated interfaces and to map default interface methods. Our `VTables` façade contains static factory methods to construct such `VTable` instances.

Listing 4.17: `VTables` *façade.*

```
//:~ core/src/main/.../util/VTables.java
public final class VTables {
  private VTables() {}

  public static VTable newDefaultMethodVTable(Class<?> clazz) {
    return newVTableBuilder(clazz, clazz)
               .excludeObjectMethods()
               .includeDefaultMethods()
               .build();
  }

  public static VTable newVTable(Class<?> receiver,
                                 Class<?>... targets) {
    return newVTableBuilder(receiver, targets).build();
  }

  public static VTable newVTableExcludingObjectMethods(
      Class<?> receiver, Class<?>... targets) {
    return newVTableBuilder(receiver, targets)
               .excludeObjectMethods()
               .build();
  }

  private static VTable.Builder newVTableBuilder(
      Class<?> receiver, Class<?>... targets) {
    VTable.Builder builder = new VTable.Builder(receiver);
    for (Class<?> target : targets) {
      builder.addTargetInterface(target);
    }
    return builder;
  }
}
```

For example, to make a `VTable` between `List` and `ArrayDeque`, we could simply write what is in Listing 4.18.

Listing 4.18: `VTablesDemo`.

```
//:~ core/src/test/.../util/VTablesDemo.java
public class VTablesDemo {
  public static void main(String... args) {
    VTable vt = VTables.newVTable(ArrayDeque.class, List.class);
    vt.stream().forEach(System.out::println);
  }
}
```

The output shows all the methods in `ArrayDeque` that also appear in `List`.

Listing 4.19: Output from `VTablesDemo`.

```
public boolean java.util.ArrayDeque.add(java.lang.Object)
public boolean java.util.ArrayDeque.isEmpty()
public int java.util.ArrayDeque.size()
public boolean java.util.ArrayDeque.contains(java.lang.Object)
public boolean
java.util.AbstractCollection.containsAll(java.util.Collection)
public boolean java.util.ArrayDeque.remove(java.lang.Object)
public java.lang.Object[] java.util.ArrayDeque.toArray()
public void java.util.ArrayDeque.clear()
etc.
```

`ChainedInvocationHandler`

We briefly introduced the chain-of-responsibility pattern in Chapter 1. We also mentioned that it has a known weakness: a request might drop off the end of the chain when no suitable handler is found.

Our `ChainedInvocationHandler` is the root handler in the pattern. Besides passing requests to the next handler in the chain, it also throws an `AssertionError` if no suitable method is discovered. Users of this class are encouraged to call `checkAllMethodsAreHandled(Class<?>… targets)` once the chain is set up, in order to verify that the chain can handle all the methods in the target interface.

Listing 4.20: `ChainedInvocationHandler`.

```
//:~ core/src/main/.../util/chain/ChainedInvocationHandler.java
/**
 * Chain of responsibility design pattern for invocation
 * handlers.   The invoke method by default passes the call to
 * the next element in the chain.  If no one can handle the
 * call, then that is an AssertionError.  Once the full chain
```

```
 * is constructed, we should check that all methods of the
 * target interface can be handled by our chain by calling
 * #checkAllMethodsAreHandled(targets).
 */
public abstract class ChainedInvocationHandler
    implements InvocationHandler {
  private final ChainedInvocationHandler next;

  public ChainedInvocationHandler(
      ChainedInvocationHandler next) {
    this.next = next;
  }

  @Override
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    if (next != null) return next.invoke(proxy, method, args);
    // we cannot allow a method to not be handled
    throw new AssertionError(
        "No InvocationHandler for " + method);
  }

  /**
   * Should be called on the first link of the chain once
   * everything has been set up. Throws IllegalArgumentException
   * if any methods are not handled by our chain.
   *
   * @throws UnhandledMethodException if any method in the
   *                                  targets is not handled
   *                                  by the chain
   * @throws IllegalArgumentException if one of the targets is
   *                                  not an interface
   */
  public void checkAllMethodsAreHandled(Class<?>... targets) {
        if (Stream.of(targets)
            .anyMatch(Predicate.not(Class::isInterface)))
          throw new IllegalArgumentException(
              "target classes must be interfaces");
    Collection<Method> unhandled =
        findUnhandledMethods(targets)
            .collect(Collectors.toList());
    if (!unhandled.isEmpty())
      throw new UnhandledMethodException(unhandled);
```

```
  }

  /**
   * Last handler in the chain returns a Stream containing all
   * the methods in the given target interfaces.  Subclasses
   * should call super.findUnhandledMethods(targets) and then
   * add filters to remove methods that are handled by their
   * handlers.
   */
  protected Stream<Method> findUnhandledMethods(
      Class<?>... targets) {
    if (next != null) return next.findUnhandledMethods(targets);
    return Stream.of(targets)
                 .map(Class::getMethods)
                 .flatMap(Stream::of)
                 .filter(
                     m -> !Modifier.isStatic(m.getModifiers()));
  }
}
```

We also define a new exception called `UnhandledMethodException`, shown in Listing 4.21. It is a special type of `IllegalArgumentException` that can only be instantiated within the `chain` package as the constructor is package private. You might be tempted to think that I wanted to stop you from instantiating `UnhandledMethodException` from outside the package. No, the real reason is that I was trying to fit the constructor declaration on a single line. Authors do that a lot.

Listing 4.21: `UnhandledMethodException`.

```
//:~ core/src/main/.../util/chain/UnhandledMethodException.java
public final class UnhandledMethodException
    extends IllegalArgumentException {
  private final Collection<Method> unhandled;
  UnhandledMethodException(Collection<Method> unhandled) {
    super("Unhandled methods: " + unhandled);
    this.unhandled = List.copyOf(unhandled);
  }
  public Collection<Method> getUnhandled() {
    return unhandled;
  }
}
```

Our `VTableHandler` takes as parameter the `receiver` object on which to invoke the methods, the `vtable` used to match methods from our target interface to our receiver class, and the `next` link in the

chain.

Inside the `invoke()` method, we look up the method. If we find a match, we invoke this on our `receiver`, else we pass it down the chain by calling `super.invoke()`.

Our `findUnhandledMethods()` removes all methods from the stream that find a match in our `vtable`.

Listing 4.22: `VTableHandler`.

```
//:~ core/src/main/.../util/chain/VTableHandler.java
public final class VTableHandler
    extends ChainedInvocationHandler {
  private final Object receiver;
  private final VTable vtable;
  public VTableHandler(Object receiver, VTable vtable,
                       ChainedInvocationHandler next) {
    super(next);
    this.receiver = Objects.requireNonNull(receiver);
    this.vtable = Objects.requireNonNull(vtable);
  }

  @Override
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    Method match = vtable.lookup(method);
    if (match != null) return match.invoke(receiver, args);
    return super.invoke(proxy, method, args);
  }

  @Override
  protected Stream<Method> findUnhandledMethods(
      Class<?>... targets) {
    return super.findUnhandledMethods(targets)
                .filter(method -> vtable.lookup(method) == null);
  }
}
```

Default interface methods have to be invoked slightly differently. Instead of a receiver object, they are called on the `proxy` parameter that is passed into the `invoke()` method. Our `VTableDefaultMethodsHandler` is thus constructed with just the `vtable` for finding default interface methods and the `next` handler in the chain. In all likelihood, this handler will be the last in the chain.

If we find a match to the default interface method in our `vtable`, we call the `MethodHandle` with our

`proxy` parameter and the proffered `args` array.

Similar to what we did with the `VTableHandler`, we also need to filter out of the stream any methods that this handler matches.

Listing 4.23: `VTableDefaultMethodsHandler`.

```
//:~ core/src/main/.../util/chain/VTableDefaultMethodsHandler.java
public final class VTableDefaultMethodsHandler
    extends ChainedInvocationHandler {
  private final VTable vtable;
  public VTableDefaultMethodsHandler(
      VTable vtable, ChainedInvocationHandler next) {
    super(next);
    this.vtable = Objects.requireNonNull(vtable);
  }

  @Override
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    MethodHandle match = vtable.lookupDefaultMethod(method);
    if (match != null)
      return match.invoke(proxy, args);
    return super.invoke(proxy, method, args);
  }

  @Override
  protected Stream<Method> findUnhandledMethods(
      Class<?>... targets) {
    return super.findUnhandledMethods(targets)
                .filter(
                    method -> vtable.lookupDefaultMethod(method)
                                 == null);
  }
}
```

The observant reader will have noticed that the two previous classes look eerily similar. For fun, we unified the two lookup approaches into one, using generics and lambdas. It works, but it would have been more challenging to explain in the book. The code is in our samples/src/test/.../util/chain package in our dynamic-proxies-samples at GitHub.

CHAPTER FIVE

Dynamic Object Adapter

Rapper wrappers

Imagine a class hierarchy of singers: bass, tenor, and soprano. They all belong to a choir.

Cacophony is Greek for "bad sound".

Now imagine that a rapper is standing next to the choir, all forlorn. The rapper would love to join the choir, but cannot. Only singers allowed, sorry.

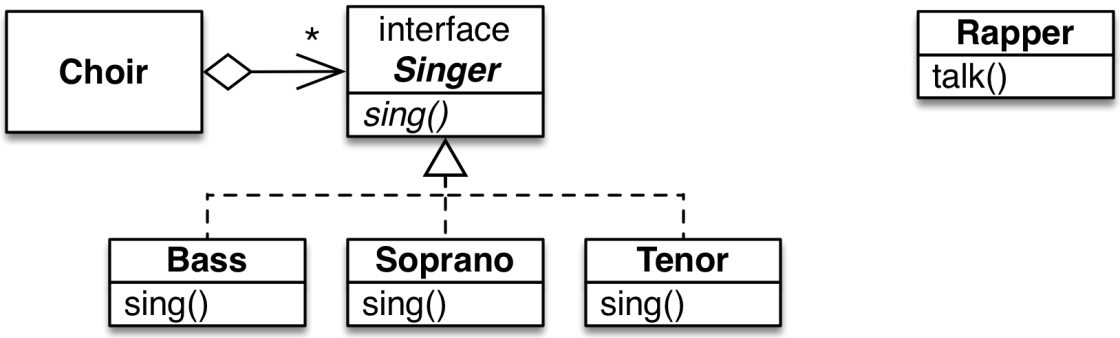


Figure 11. A choir structure with singers. The forlorn rapper looks on.

How can we make the rapper into a singer? Some would say this is an impossibility. Rappers talk, singers sing. End of story.

But let's put aside our feelings about the musical quality of rap for a moment. Let's also assume that we cannot change the `Rapper` class.

Class adapter pattern

The easiest way to create a rapper wrapper is with the class adapter pattern. We create a `SingingRapper` that implements `Singer` and extends `Rapper`. In the `sing()` method, we call `talk()`.

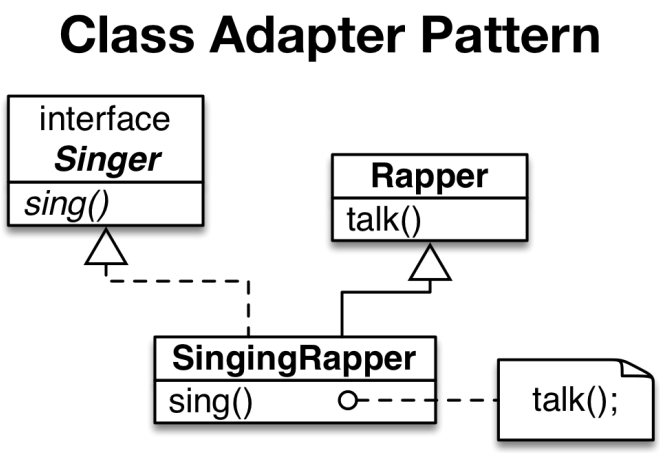


Figure 12. A singing, talking rapper.

Our `SingingRapper` extends `Rapper`, making it both a rapper and a singer. This works fine if there is only one type of `Rapper`, but the list of rapper types seems to be endless: East Coast, West Coast, boom bap, drill, emo rap, mumble rap, gangsta, trap… the list goes on and on. What seems like a great low-code solution turns out to need dozens of similar-looking classes, one for each unique rap style.

Object adapter pattern

The object adapter pattern takes a rapper as a constructor parameter and delegates the `sing()` method to the `talk()` method on its rapper. We can accept any subclass of `Rapper` as an argument, even a `MumbleRapper`. Our rapper wrapper now implements `Singer`, but does not extend `Rapper`, so it is not a rapper. We can call `sing()`, but `talk()` is no longer visible.

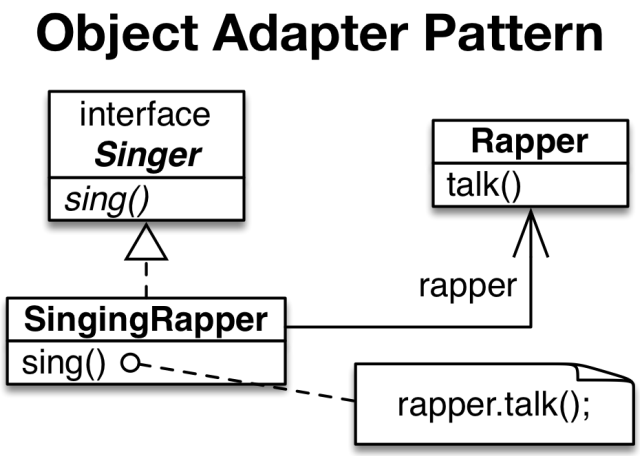


Figure 13. Converting a rapper into a singer.

The object adapter pattern seems the better choice of the two. But what if the `Rapper` and the `Singer` had quite a few methods in common, such as `clap()`, `bow()`, and `dance()`? With the class adapter pattern, we only need to implement the `sing()` method. The object adapter pattern, on the other hand, requires us to delegate all of the methods. This can become onerous with rich target interfaces.

A better `ArrayList`

When we create an `ArrayList<String>`, we go to the trouble of specifying that it contains `String`. However, this information is erased from the class to preserve backwards compatibility of the class file format. The `ArrayList` class has no knowledge of what the generic type parameter is. If we call `toArray()`, what is returned is an `Object[]`, even though we declared our collection to contain `String`.

Java's religion is backwards compatibility. We can use class files from Java 1.0 without recompiling the source code. This makes Java super attractive to banks and other large organizations. When generics were added, they erased the generic types when compiling the class. There is thus no difference between `ArrayList<String>` and `ArrayList<Integer>`. Internally, it is simply an `ArrayList` containing an `Object[]`.

Java 5 introduced covariant return types. An overridden method can return a subclass of the return type of its parent method. Thus, a subclass of `ArrayList` could return a subclass of `Object[]` from its `toArray()` method — for example `String[]`, `Integer[]`, or whatever generic type it was defined with.

Our `BetterArrayList` has an empty `seedArray` of the generic type parameter, which the `toArray()` method uses to create the correct type of array.

Listing 5.1: `BetterArrayList`.

```
//:~ samples/src/main/.../ch05/bettercollection/BetterArrayList.java
public class BetterArrayList<E> extends ArrayList<E> {
  private final E[] seedArray;

  public BetterArrayList(E[] seedArray) {
    if (seedArray.length != 0)
      throw new IllegalArgumentException(
          "seedArray must be empty");
    this.seedArray = seedArray;
  }

  @Override
  public E[] toArray() {
    // NOTE: Shipilev showed that this is the fastest way to
    // create a typed array from a collection:
    // https://shipilev.net/blog/2016/arrays-wisdom-ancients/
    return toArray(seedArray);
  }

  @Override
  public String toString() {
    return "--" + super.toString() + "--";
  }
}
```

Listing 5.2 shows one way we could use this new class. We pass in a `new String[0]`, which is then used to create the correct type of array.

Listing 5.2: `BetterArrayListDemo`.

```
//:~ samples/src/main/.../ch05/bettercollection/BetterArrayListDemo.java
BetterArrayList<String> names =
    new BetterArrayList<>(new String[0]);
names.add("Wolfgang");
names.add("Leander");
names.add("Bobby Tables");
names.add("Klaus");
names.add("Menongahela");
String[] nameArray = names.toArray();
for (String name : nameArray) {
  System.out.println(name.toUpperCase());
}
System.out.println(names);
```

The output is shown in Listing 5.3.

Listing 5.3: Output from `BetterArrayListDemo`.

```
WOLFGANG
LEANDER
BOBBY TABLES
KLAUS
MENONGAHELA
--[Wolfgang, Bobby Tables, Leander, Klaus, Menongahela]--
```

One danger with returning a subclass of `Object[]` is that we might accidentally get an `ArrayStoreException` when third-party code expects the superclass `Object[]`. Since `String[]` is a subclass of `Object[]`, if someone upcasts our array to `Object[]`, they may try to add an `Integer`. This would cause an `ArrayStoreException`.

`BetterCollection` interface

Another approach is to extend the `Collection` interface and override the `toArray()` method to return the correct type. Listing 5.4 shows how this interface looks:

Listing 5.4: `BetterCollection`.

```
//:~ samples/src/main/.../ch05/bettercollection/BetterCollection.java
public interface BetterCollection<E> extends Collection<E> {
  @Override
  E[] toArray();
  default void forEachFiltered(Predicate<? super E> predicate,
                               Consumer<? super E> action) {
    Objects.requireNonNull(predicate, "predicate==null");
    Objects.requireNonNull(action, "action==null");
    for (E e : this) {
      if (predicate.test(e)) action.accept(e);
    }
  }
}
```

Life is good when we use a class adapter. To make a better `ConcurrentSkipListSet`, we can extend this class and implement `BetterCollection`. The code looks similar to `BetterArrayList`.

Listing 5.5: `BetterConcurrentSkipListSet`.

```
//:~
samples/src/main/.../ch05/bettercollection/BetterConcurrentSkipListSet.jav
a
public class BetterConcurrentSkipListSet<E>
    extends ConcurrentSkipListSet<E>
    implements BetterCollection<E> {
  private final E[] seedArray;
  public BetterConcurrentSkipListSet(E[] seedArray) {
    if (seedArray.length != 0)
      throw new IllegalArgumentException(
          "seedArray must be empty");
    this.seedArray = seedArray;
  }
  @Override
  public E[] toArray() {
    return toArray(seedArray);
  }
  @Override
  public String toString() {
    return "--" + super.toString() + "--";
  }
}
```

We can see that the class adapter, while needing only a few lines of code, does have some disadvantages. Firstly, we must subclass each concrete collection individually. This might take a while. In OpenJDK 14, `Collection` is overridden 195 times!

Secondly, we can only extend classes where we control the construction. For example, to construct a concurrent `HashSet` in Java, we use the `ConcurrentHashMap.newKeySet()` static factory method. This returns `KeySetView`, a public static inner class of `ConcurrentHashMap` with a package-access constructor. Since the constructor is not accessible, we cannot extend `KeySetView`, ergo we cannot use the class adapter pattern to make it into a `BetterCollection`.

Instead, we can write an object adapter that implements `BetterCollection` and wraps a `Collection` adaptee. Note that we then have to implement all the methods from `Collection`. Rather tedious, no?

Listing 5.6: `BetterCollectionObjectAdapter`.

```
//:~
samples/src/main/.../ch05/bettercollection/BetterCollectionObjectAdapter.j
ava
public class BetterCollectionObjectAdapter<E>
```

```
  implements BetterCollection<E> {
private final Collection<E> adaptee;
private final E[] seedArray;

public BetterCollectionObjectAdapter(Collection<E> adaptee,
                                     E[] seedArray) {
  if (seedArray.length != 0)
    throw new IllegalArgumentException(
        "seedArray must be empty");

  this.adaptee = adaptee;
  this.seedArray = seedArray;
}

@Override
public E[] toArray() {
  return adaptee.toArray(seedArray);
}

@Override
public String toString() {
  return "--" + adaptee.toString() + "--";
}

// this is a typical problem with the object adapter design
// pattern - we have to implement all the methods :-(
@Override
public int size() {
  return adaptee.size();
}

@Override
public boolean isEmpty() {
  return adaptee.isEmpty();
}

@Override
public boolean contains(Object o) {
  return adaptee.contains(o);
}

@Override
public Iterator<E> iterator() {
```

```
    return adaptee.iterator();
  }

  @Override
  public <T> T[] toArray(T[] ts) {
    return adaptee.toArray(ts);
  }

  @Override
  public boolean add(E e) {
    return adaptee.add(e);
  }

  @Override
  public boolean remove(Object o) {
    return adaptee.remove(o);
  }

  @Override
  public boolean containsAll(Collection<?> c) {
    return adaptee.containsAll(c);
  }

  @Override
  public boolean addAll(Collection<? extends E> es) {
    return adaptee.addAll(es);
  }

  @Override
  public boolean removeAll(Collection<?> c) {
    return adaptee.removeAll(c);
  }

  @Override
  public boolean retainAll(Collection<?> c) {
    return adaptee.retainAll(c);
  }

  @Override
  public void clear() {
    adaptee.clear();
  }
}
```

Our object adapter works and we can use it to adapt any collection. Here, we are wrapping the `ConcurrentHashMap.newKeySet()`.

Listing 5.7: `BetterCollectionObjectAdapterDemo`.

```
//:~
samples/src/main/.../ch05/bettercollection/BetterCollectionObjectAdapterDe
mo.java
BetterCollection<String> names =
    new BetterCollectionObjectAdapter<>(
        ConcurrentHashMap.newKeySet(),
        new String[0]
    );
names.add("Wolfgang");
names.add("Leander");
names.add("Bobby Tables");
names.add("Klaus");
names.add("Menongahela");
String[] nameArray = names.toArray();
for (String name : nameArray) {
  System.out.println(name.toUpperCase());
}
System.out.println(names);
```

Listing 5.8: Output from `BetterCollectionObjectAdapterDemo`.

```
MENONGAHELA
LEANDER
WOLFGANG
KLAUS
BOBBY TABLES
--[Menongahela, Leander, Wolfgang, Klaus, Bobby Tables]--
```

It works, but… well… — if we want to now have a better `Set`, `List`, `SortedSet`, `ConcurrentMap`, `Map`, or even `java.sql.Connection`, we will play that rodeo over and over, writing lots of delegating methods that do little besides passing along the method call.

Dynamic object adapter using dynamic proxies

As stated before, the class structures of proxy and object adapter are similar. Can we create a dynamic object adapter?

The static factory method for our dynamic object adapter takes the following parameters:

- `Class<E> target` is the interface class that we want to adapt to. Our new object will implement this — for example, `BetterCollection`.
- `Object adaptee` is the object that we want our new interface to wrap, for example `ArrayList<String>`.
- `Object adapter` is an object that contains the methods that should be used to implement the target interface.

Our `ObjectAdapterHandler` creates `VTables` for the adapter, adaptee, and default methods and arranges them in a chain of responsibility. All methods in the `target` interface need to be implemented in `adaptee`, `adapter`, or as default methods in `target`.

Listing 5.9: `ObjectAdapterHandler`.

```
//:~ core/src/main/.../handlers/ObjectAdapterHandler.java
public final class ObjectAdapterHandler
    implements InvocationHandler {
  private final ChainedInvocationHandler chain;

  public ObjectAdapterHandler(Class<?> target,
                              Object adaptee,
                              Object adapter) {
    VTable adapterVT =
        VTables.newVTable(adapter.getClass(), target);
    VTable adapteeVT =
        VTables.newVTable(adaptee.getClass(), target);
    VTable defaultVT = VTables.newDefaultMethodVTable(target);

    chain = new VTableHandler(adapter, adapterVT,
        new VTableHandler(adaptee, adapteeVT,
            new VTableDefaultMethodsHandler(defaultVT, null)));

    chain.checkAllMethodsAreHandled(target);
  }

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

We add another static factory method to our ubiquitous `Proxies` class, which will create an object adapter based on the above `InvocationHandler`.

Listing 5.10: `Proxies.adapt()`.

```
//:~ core/src/main/.../Proxies.java
public static <T> T adapt(Class<? super T> target,
                          Object adaptee,
                          Object adapter) {
  Objects.requireNonNull(adaptee, "adaptee==null");
  Objects.requireNonNull(adapter, "adapter==null");
  return castProxy(target,
      new ObjectAdapterHandler(target, adaptee, adapter));
}
```

For example, to create a `BetterCollectionFactory`, we can make an adapter that first creates the correct type of array in the `toArray()` method. It then changes the `add()` function to verify that the type of object is correct, similar to how `Collections.checkedCollection()` works.

Listing 5.11: `BetterCollectionFactory`.

```
//:~
samples/src/main/.../ch05/bettercollection/BetterCollectionFactory.java
public class BetterCollectionFactory {
  public static <E> BetterCollection<E> asBetterCollection(
      Collection<E> adaptee, E[] seedArray) {
    return Proxies.adapt(BetterCollection.class, adaptee,
        new AdaptationObject<>(adaptee, seedArray));
  }

  public static <E> BetterSortedSet<E> asBetterSet(
      SortedSet<E> adaptee, E[] seedArray) {
    return Proxies.adapt(BetterSortedSet.class, adaptee,
        new AdaptationObject<>(adaptee, seedArray));
  }

  // this public inner class contains the methods that
  // we want to adapt
  public static class AdaptationObject<E> {
    private final Collection<E> adaptee;
    private final E[] seedArray;
    public AdaptationObject(Collection<E> adaptee,
                            E[] seedArray) {
      this.adaptee = adaptee;
      this.seedArray = seedArray;
    }
```

```
    public E[] toArray() {
      return adaptee.toArray(seedArray);
    }
    @Override
    public String toString() {
      return "--" + adaptee + "--";
    }
    // Whilst we are at it, we could also make it into
    // a checked collection, see java.util.Collections
    // for an example.
    public boolean add(E e) {
      var type = seedArray.getClass().getComponentType();
      if (!type.isInstance(e))
        throw new ClassCastException(
            "Attempt to insert " + e.getClass() +
                " value into collection " +
                "with type " + type);
      return adaptee.add(e);
    }
    // addAll() left as an exercise for the reader :-)
  }
}
```

Let's try it out with a `HashSet`, wrapped with our `BetterCollectionObjectAdapter`.

Listing 5.12: `BetterCollectionDynamicObjectAdapterDemo`.

```
//:~
samples/src/main/.../ch05/bettercollection/BetterCollectionDynamicObjectAd
apterDemo.java
BetterCollection<String> names =
    BetterCollectionFactory.asBetterCollection(
        new HashSet<>(), new String[0]);
names.add("Wolfgang");
names.add("Bobby Tables");
names.add("Leander");
names.add("Klaus");
names.add("Menongahela");
String[] nameArray = names.toArray();
for (String name : nameArray) {
  System.out.println(name.toUpperCase());
}
System.out.println(names);

((Collection) names).add(42); // this will fail
```

Listing 5.13: Output from `BetterCollectionDynamicObjectAdapterDemo`.

```
MENONGAHELA
LEANDER
WOLFGANG
KLAUS
BOBBY TABLES
--[Menongahela, Leander, Wolfgang, Klaus, Bobby Tables]--
Exception in thread "main" java.lang.ClassCastException: Attempt
to insert class java.lang.Integer value into collection with
value type class java.lang.String
  at BetterCollectionFactory$AdaptationObject.add
      (BetterCollectionFactory.java)
    ... etc.
```

Thanks to reflection, we are able to avoid tightly coupling the adapter and adaptee classes. We also only had to implement the part of the adapter that was changed.

The cost of method calls has increased slightly, since each call is at least one `VTable` lookup. As we will see later, this cost goes up a bit when we need to call default interface methods.

Restrictions of adapter

In our approach to the dynamic object adapter, neither the `adapter` nor the `adaptee` would typically implement the `target` interface. We need to ensure that at least one public class in the hierarchy has a public declaration of each method in the `target`. As convenient as they would be, anonymous inner classes are package-protected and thus unsuitable to use as adapters.

Consider the class `AdapterPuzzle1`.

Listing 5.14: `AdapterPuzzle1`.

```
//:~ samples/src/main/.../ch05/AdapterPuzzle1.java
public class AdapterPuzzle1 {
  public static void main(String... args) {
    CharSequence seq =
        Proxies.adapt(CharSequence.class,
            "Good morning Vietnam",
            new Object() {
              @Override
              public String toString() {
                return "Kilroy";
              }
              public int length() {
                return 42;
              }
            });
    System.out.println(seq + " " + seq.length());
  }
}
```

Which of the following is the output?:

- Good morning Vietnam 20
- Good morning Vietnam 42
- Good morning Vietnam 6
- Kilroy 20
- Kilroy 42
- Kilroy 6

The correct answer is “Kilroy 20”.

Let us try to figure out why.

The target interface is a `CharSequence`, the `adaptee` is a `String`, and the `adapter` is an anonymous inner class that is a subclass of `Object`.

The anonymous inner class is package-private, so the method `length()` is not visible outside of the package. When we call `length()`, we call the `String` method and get the value "20".

The method `toString()` is visible since it is inherited from the public class `Object`. When we call `toString()`, the inner class's `toString()` method is called, returning the value "Kilroy".

Now consider the class `AdapterPuzzle2`.

Listing 5.15: `AdapterPuzzle2`.

```
//:~ samples/src/main/.../ch05/AdapterPuzzle2.java
public class AdapterPuzzle2 {
  public interface LengthProvider {
    int length();
  }
  public static void main(String... args) {
    CharSequence seq =
        Proxies.adapt(CharSequence.class,
            "Good morning Vietnam",
            new LengthProvider() {
              @Override
              public String toString() {
                return "Kilroy";
              }
              @Override
              public int length() {
                return 42;
              }
            });
    System.out.println(seq + " " + seq.length());
  }
}
```

Since we are implementing a public interface, the `length()` method is visible to the adapter. The answer this time will be a more consistent "Kilroy 42".

Performance of dynamic-object-adapter method calls

Performing several hash map lookups every time we call a method on our dynamic object adapter is slow. How slow though?

In our `AdapterBenchmark`, we create four collections:

- `plain` is a no-frills `ConcurrentSkipListSet`.
- `classAdapter` is the `BetterConcurrentSkipListSet` that we saw earlier in this chapter.
- `objectAdapter` is the `BetterCollectionObjectAdapter` handcrafted object adapter, wrapping a `ConcurrentSkipListSet`.
- `dynamicObjectAdapter` is an instance of our dynamic object adapter created with our `BetterCollectionFactory` and wrapping a `ConcurrentSkipListSet`.

We call methods `size()`, `toArray()` and `forEachFiltered()` with each of these collections.

Listing 5.16: `AdapterBenchmark`.

```
//:~ benchmark/src/main/.../ch05/benchmarks/AdapterBenchmark.java
@Fork(value = 3, jvmArgsAppend = {"-XX:+UseParallelGC",
    "-Deu.javaspecialists.books.dynamicproxies.util" +
        ".ParameterTypesFetcher.enabled=true"})
@Warmup(iterations = 5, time = 5)
@Measurement(iterations = 10, time = 5)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class AdapterBenchmark {
  private final Collection<String> plain =
      new ConcurrentSkipListSet<>();

  private final BetterCollection<String> classAdapter =
      new BetterConcurrentSkipListSet<>(new String[0]);

  private final BetterCollection<String> objectAdapter =
      new BetterCollectionObjectAdapter<>(
          new ConcurrentSkipListSet<>(), new String[0]);

  private final BetterCollection<String> dynamicObjectAdapter =
      BetterCollectionFactory.asBetterCollection(
          new ConcurrentSkipListSet<>(), new String[0]
      );

  private final Collection<Collection<String>> all =
      List.of(plain, classAdapter, objectAdapter,
          dynamicObjectAdapter);
```

```
  @Setup
  public void init() {
    all.forEach(Collection::clear);
    all.forEach(c -> c.add("John"));
    all.forEach(c -> c.add("Mary"));
    all.forEach(c -> c.add("Menongahela"));
    all.forEach(c -> c.add("Bobby Tables"));
  }

  @Benchmark
  public int plainSize() {
    return plain.size();
  }
  @Benchmark
  public int classAdapterSize() {
    return classAdapter.size();
  }
  @Benchmark
  public int objectAdapterSize() {
    return objectAdapter.size();
  }
  @Benchmark
  public int dynamicObjectAdapterSize() {
    return dynamicObjectAdapter.size();
  }

  @Benchmark
  public Object[] plainToArray() {
    return plain.toArray();
  }
  @Benchmark
  public String[] classAdapterToArray() {
    return classAdapter.toArray();
  }
  @Benchmark
  public String[] objectAdapterToArray() {
    return objectAdapter.toArray();
  }
  @Benchmark
  public String[] dynamicObjectAdapterToArray() {
    return dynamicObjectAdapter.toArray();
  }
```

```
  private static final Predicate<String> predicate =
      s -> s.length() < 10;

  private static class Counter implements Consumer<Object> {
    private int count;
    @Override
    public void accept(Object o) { count++; }
    public void reset() { count = 0; }
    public int get() { return count; }
  }

  private static final Counter COUNTER = new Counter();

  @Benchmark
  public int plainForEach() {
    COUNTER.reset();
    Objects.requireNonNull(predicate, "predicate==null");
    Objects.requireNonNull(COUNTER, "COUNTER==null");
    for (String e : plain) {
      if (predicate.test(e)) COUNTER.accept(e);
    }
    return COUNTER.get();
  }
  @Benchmark
  public int classAdapterForEach() {
    COUNTER.reset();
    classAdapter.forEachFiltered(predicate, COUNTER);
    return COUNTER.get();
  }
  @Benchmark
  public int objectAdapterForEach() {
    COUNTER.reset();
    objectAdapter.forEachFiltered(predicate, COUNTER);
    return COUNTER.get();
  }
  @Benchmark
  public int dynamicObjectAdapterForEach() {
    COUNTER.reset();
    dynamicObjectAdapter.forEachFiltered(predicate, COUNTER);
    return COUNTER.get();
  }

  @Benchmark
```

```
  public Iterator<String> iterator() {
    return plain.iterator();
  }

  @Benchmark
  public Object[] parameterArray() {
    return new Object[] {predicate, COUNTER};
  }
}
```

These results come from running our `AdapterBenchmark` on a MacBook Pro (2018) i9 with six cores with Java 14+36-1461. Times are in nanoseconds per operation (ns/op) and memory allocated is bytes per operation (B/op), with escape analysis on and off, using the `-XX:+/-DoEscapeAnalysis` JVM parameter. The `FastParameterTypesFetcher` and the `MethodTurboBooster` are both enabled.

Performance of size()

The `size()` method is defined inside the adaptee, thus we need to make two `VTable` lookups, first on the adapter and then on the adaptee. The overhead is approximately 17 nanoseconds, thus 8.5 nanoseconds per `VTable` lookup. Since the `size()` method is not overloaded in the `BetterCollection` interface, we do not need to match the parameter types.

None of the tests allocate memory with the current data set. However, if the set increased in size beyond 127, then the dynamic object adapter would need to allocate an `Integer` object for the return value.

Benchmark `size()`	**best ns/op**	**B/op EA on / off**
plainSize()	5.2	0 / 0
classAdapterSize()	5.2	0 / 0
objectAdapterSize()	5.3	0 / 0
dynamicObjectAdapterSize()	22	0 / 0

Performance of toArray()

We see in the table below that there is only an 11 nanosecond difference between the static and dynamic object adapters, which is the cost of one `VTable` lookup. The reason why we only need a single lookup is that the `toArray()` method is defined inside the adapter class, which happens to be the first handler in the chain. However, the `toArray()` method is overloaded, which means that the name is not distinct and we therefore need to compare the parameter types. This explains why the overhead per `VTable` lookup is slightly larger than with `size()`.

When the `FastParameterTypesFetcher` is turned on, all the tests allocate the same amount of

memory, consisting of an iterator (32 bytes) and an array for the result (also 32 bytes). The `toArray()` method calls `iterator()` on the `ConcurrentSkipListSet`. Unfortunately, escape analysis in this case cannot determine that the iterator object cannot escape, thus it has to be created on the Java heap. For more information about escape analysis, see Scalar Replacement by Shipilev.

When we turn the `FastParameterTypesFetcher` off, the dynamic object adapter test needs to allocate an empty class array for matching the correct `toArray()` method. This takes an additional 7 nanoseconds per method call.

Benchmark `toArray()`	**best ns/op**	**B/op EA on / off**
plainToArray()	28	64 / 64
classAdapterToArray()	33	64 / 64
objectAdapterToArray()	32	64 / 64
dynamicObjectAdapterToArray() (FastParameterTypesFetcher=enabled)	43	64 / 64
dynamicObjectAdapterToArray() (FastParameterTypesFetcher=disabled)	50	80 / 80

Performance of forEachFiltered()

The default interface method `forEachFiltered()` defined inside `BetterCollection` has the highest overhead, coming to about 36 nanoseconds. This requires 5 `VTable` lookups. We first lookup in the adapter and the adaptee and when we do not find the method there, we find it as a default interface method defined on our target interface. However, internally, the `forEachFiltered()` method calls `iterator()` during the for-in loop, which requires another two lookups on adapter and adaptee.

All of the tests create an iterator (32 bytes). However, since the iterator object is contained within the method, the heap allocation can usually be optimized away, except with the dynamic object adapter. Our dynamic proxy invocations have deeper call stacks, as we discussed in chapter 3. The computational time complexity of determining whether an object cannot escape is quite high, and thus escape analysis limits how deeply it analyzes the stack. In our case, escape analysis gives up and thus the iterator object always has to be allocated on the heap, with the associated garbage collection cost.

A further 24 bytes are wasted due to the `Object[]` for the parameters of the method.

The allocation rate for the `dynamicObjectAdapterForEach()` benchmark was approximately 500 MB/s. This is a bit high, but nowhere close to the maximum of the machine I ran the tests on. For comparison, on my machine I can allocate 2 GB/s of iterators and 3 GB/s of object arrays. The bottleneck is in the `VTable` lookups and not in the object allocation.

Benchmark `forEachFiltered()`	**best ns/op**	**B/op EA on / off**
plainForEach()	24	0 / 32

Benchmark `forEachFiltered()`	best ns/op	B/op EA on / off
classAdapterForEach()	24	0 / 32
objectAdapterForEach()	26	0 / 32
dynamicObjectAdapterForEach()	62	56 / 56

Conclusion of performance analysis

The overhead of calling methods on the dynamic object adapter is dependent on how many `VTable` lookups we have to make.

When we use dynamic proxies, we also might have a higher object allocation rate, due to parameter arrays, boxed return values, and escape analysis not being as effective as with direct calls.

CHAPTER SIX

Dynamic Composite

We use the composite design pattern to create assemblies of objects. An example is a directory structure with files, symbolic links, and other directories. A ZIP file is also considered a composite, containing directories, files, and even other ZIP files. We also see this pattern in use in GUI frameworks, where a panel can contain buttons, labels, text fields, or other panels.

In this chapter, we examine the composite design pattern and see how we can use dynamic proxies to help implement composite object relationships. First, we see an example with email contacts and distribution lists. Then we demonstrate how to use a dynamic-proxy-powered composite instead of our handcrafted distribution list. Lastly, we see another example for teeing output.

The GoF structure for the composite pattern is shown in Figure 14.

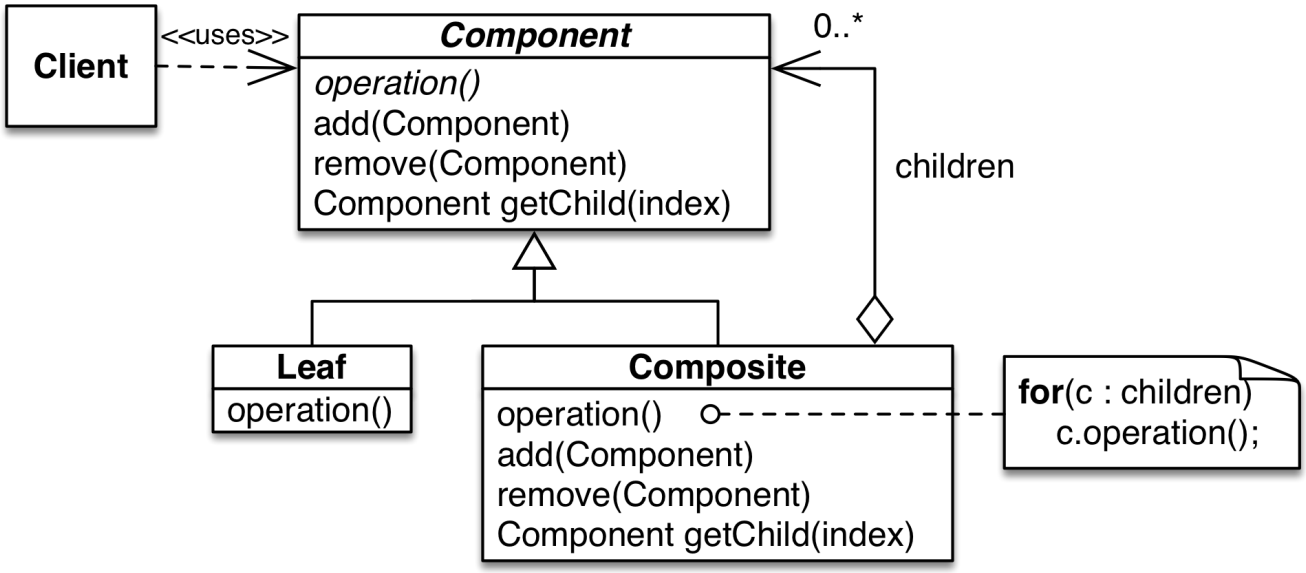


Figure 14. Structure of the composite design pattern.

Astute readers might notice that methods `add()`, `remove()`, and `getChild()` are defined in `Component`. This overly general interface is the biggest criticism for this pattern. John Vlissides wrote an excellent two-part article on the composite pattern, “Composite à la Java Part I” and “Part II”, in which he explained alternatives.

Placing a Check Box on a Button

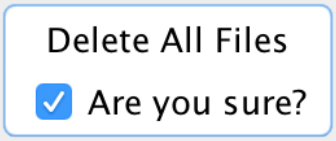
What happens if we add a `JCheckBox` on a `JButton`? Does the universe end?

In the inheritance hierarchy, `JButton` extends `JComponent` extends `Container` extends `Component`. `Component` does not have an `add()` method, but `Container` does. All we need to do is set a `LayoutManager` for the button, and we can then add whatever component we fancy.

Few programmers have tried to create compound buttons that include a check box.

Listing 6.1: Check box on a button.

```
//:~ samples/src/main/.../ch06/gui/CheckBoxButton.java
var button = new JButton();
button.setLayout(new GridLayout(0, 1));
button.add(new JLabel("Delete All Files", CENTER));
button.add(new JCheckBox("Are you sure?"));
```



`Contact`, `Person`, and `DistributionList`

One of Java's first dynamic-proxy tutorials was published in The Java Specialists' Newsletter, distributed via email. The newsletter started with 80 contacts, for which I created a distribution list in Microsoft Outlook. To reduce spam, Microsoft Outlook restricted the number of contacts in a single distribution list. Once the number of subscribers reached the limit, I made a composite of distribution lists.

The structure of our new `Contact` example is shown in Figure 15.

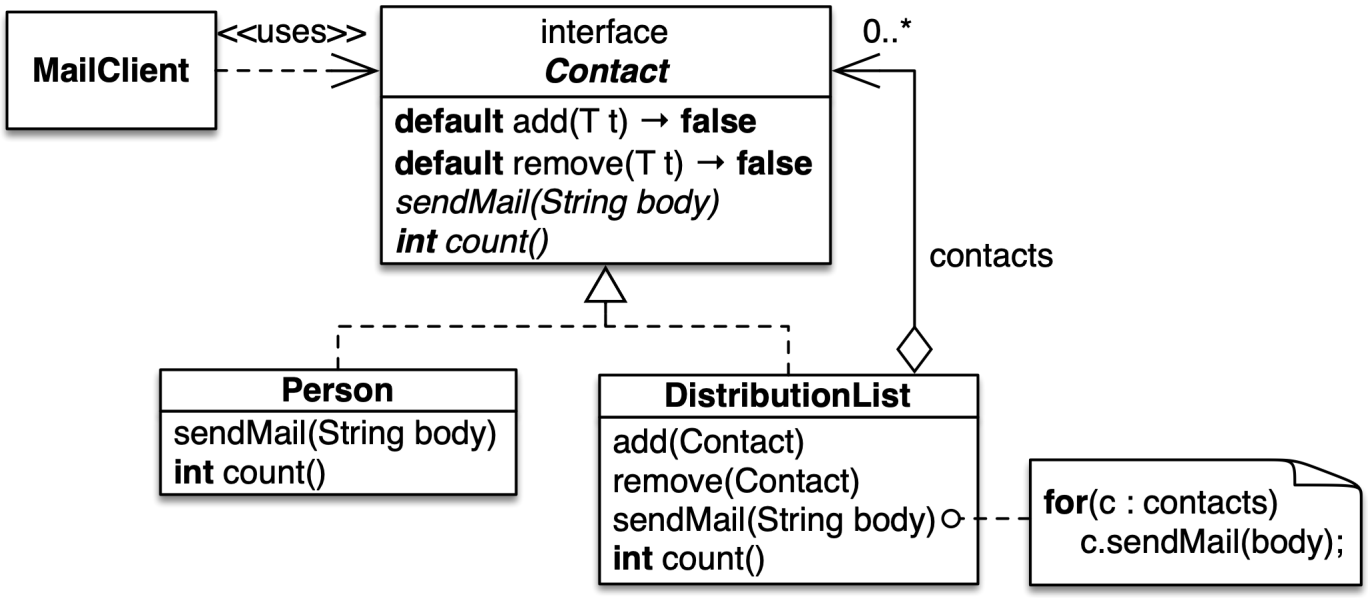


Figure 15. `Contact` example.

`Contact` is an interface representing a contact, which can either be a single contact or a group of contacts. `Contact` has two abstract methods: `sendMail()` and `count()`. It also has two default interface methods that both return `false`: `add()` and `remove()`.

Listing 6.2: `Contact` interface.

```
//:~ samples/src/main/.../ch06/contact/Contact.java
public interface Contact {
  default boolean add(Contact c) { return false; }
  default boolean remove(Contact c) { return false; }
  void sendMail(String body);
  int count();
}
```

`Person` is our spam victim.

In `sendMail()`, we connect to JavaMail and send the actual message. (We have left out the JavaMail implementation details for brevity.)

Listing 6.3: `Person` *class.*

```
//:~ samples/src/main/.../ch06/contact/Person.java
public class Person implements Contact {
  private final String email;
  public Person(String email) {this.email = email;}
  @Override
  public void sendMail(String body) {
    // connecting to JavaMail and off it goes ...
    System.out.println("Sending " + body + " to " + email);
  }
  @Override
  public int count() {
    return 1;
  }
}
```

Our `DistributionList` is our composite class, made up of `Contact` instances. Since `DistributionList` is itself a `Contact`, it can contain not only `Person` instances, but also other `DistributionList` objects. `sendMail()` recursively spams all the contacts in our list. `count()` is the sum of all our contained leaves.

Listing 6.4: `DistributionList` *class.*

```
//:~ samples/src/main/.../ch06/contact/DistributionList.java
import java.util.*;

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

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

Now that we have seen how `Contact` relates to `Person` and `DistributionList`, it's time to look at an example.

Listing 6.5: `ContactDemo` *creating a composite structure.*

```
//:~ samples/src/main/.../ch06/contact/ContactDemo.java
Contact javaSpecialistsNewsletter = new DistributionList();
System.out.println(javaSpecialistsNewsletter.count());
javaSpecialistsNewsletter.add(new Person("john@aol.com"));
javaSpecialistsNewsletter.sendMail("Hello there 1");
System.out.println(javaSpecialistsNewsletter.count());

Contact allStudents = new DistributionList();
allStudents.add(new Person("peter@absa.co.za"));
allStudents.add(new Person("mzani@absa.co.za"));
javaSpecialistsNewsletter.add(allStudents);

javaSpecialistsNewsletter.sendMail("Hello there 2");
System.out.println(javaSpecialistsNewsletter.count());

Contact extremeJava = new DistributionList();
extremeJava.add(new Person("John@fnb.co.za"));
extremeJava.add(new Person("Hlope@fnb.co.za"));
allStudents.add(extremeJava);

javaSpecialistsNewsletter.sendMail("Hello there 3");
System.out.println(javaSpecialistsNewsletter.count());
```

We can see how the contact objects are connected in Figure 16.

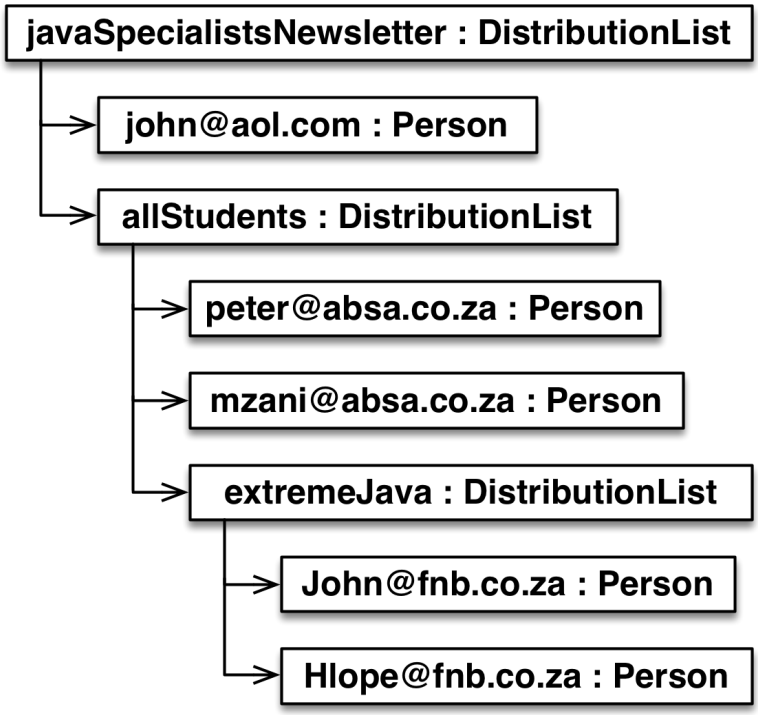


Figure 16. `ContactDemo` object diagram.

When we run this, we see the output shown.

Listing 6.6: `ContactDemo` output.

```
0
Sending Hello there 1 to john@aol.com
1
Sending Hello there 2 to john@aol.com
Sending Hello there 2 to peter@absa.co.za
Sending Hello there 2 to mzani@absa.co.za
3
Sending Hello there 3 to john@aol.com
Sending Hello there 3 to peter@absa.co.za
Sending Hello there 3 to mzani@absa.co.za
Sending Hello there 3 to John@standardbank.co.za
Sending Hello there 3 to Hlope@standardbank.co.za
5
```

Hacking dynamic proxy for composites

Can we use a dynamic proxy instead of handcrafting `DistributionList`?

The biggest difference between the composite and proxy patterns is the multiplicity of the associations.

A proxy pattern has a one-to-one relationship with the subject whereas a composite pattern has a one-

to-many relationship with the component. When we call a method on the proxy, we can return the result to the client. However, when we call a method on the composite, we might be faced with many results. These have to be merged or reduced before returning a single result to the client.

`BaseComponent<T>` represents any hierarchy that implements the composite design pattern. The generic type parameter `T` in `BaseComponent` represents the extension interface — for example, `Contact`. It also has two default methods, `add(T)` and `remove(T)`, which by default do nothing.

Listing 6.7: `BaseComponent` *interface.*

```
//:~ core/src/main/.../handlers/BaseComponent.java
public interface BaseComponent<T> {
  default boolean add(T t) { return false; }
  default boolean remove(T t) { return false; }
}
```

We further change `Contact` to extend `BaseComponent<Contact>`, inheriting the `add()` and `remove()` methods.

Listing 6.8: `Contact` *interface.*

```
//:~ samples/src/main/.../ch06/contactdynamic/Contact.java
public interface Contact extends BaseComponent<Contact> {
  void sendMail(String body);
  int count();
}
```

Our `Contact` extends `BaseComponent<Contact>`, shown in Figure 17.

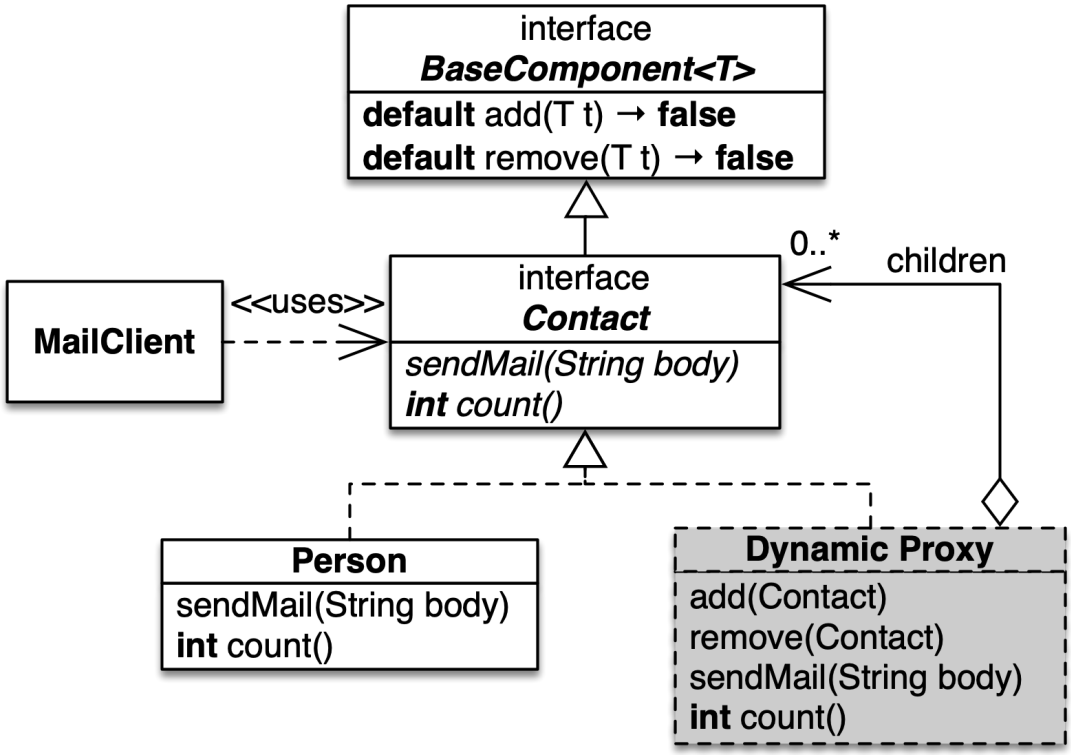


Figure 17. `Contact` *extends* `BaseComponent`.

Our `CompositeHandler` constructor takes as parameters the component type, which must be derived from `BaseComponent`. The `reducers` parameter contains the logic for how to merge the results from the method call. In streams, we can reduce all elements into another object. This is similar. Our `Reducer` contains a `BinaryOperator<T>`, which is derived from `BiFunction<T,T,T>`. We will see more detail a bit later. The last parameter is `typeChecks`. These are interfaces or classes that parameters of the `add()` method must extend. We will see examples later.

The `CompositeHandler` has a `VTable` of default interface methods, called `defaultVT`. If a default method exists, we first call that before calling the children methods, merging all the results together with the relevant reducer.

The `invoke()` method checks whether the method name is `"add"` or `"remove"`, with `Object.class` as the single parameter type, in which case it calls the relevant `add()` or `remove()` method on its list of children. The method `add()` needs a bit of explanation. We first check that the parameter is of the correct type, meaning that it is an instance of all the types in our `typeChecks` array. While these would typically be interfaces, Java does not have multiple inheritance of classes so one of them could also be a class. For each type of child, we also need to find matching methods that we can call, based on the methods in our `component` interface. This is necessary to support a heterogenous hierarchy of components.

All methods besides `"add"` and `"remove"` are managed via a map and reduce on the stream of children. `map()` invokes the desired method on each child and `reduce()` merges the results.

Unfortunately, lambdas do not play nice with checked exceptions, so we define an `UncheckedException` class inside the method. Yes, that is possible and has the advantage of being

invisible outside of that one method. We later unwrap the exception again and throw it from our `invoke()` method. We fail on the first exception that we encounter. Since the `UncheckedException` is method-private, we erase the stack trace by returning `null` from the `fillInStackTrace()` method.

We will look at `Reducer` in a bit. For now, it is enough to know that there are two special types of `Reducer`: the `NULL_REDUCER` and the `PROXY_INSTANCE_REDUCER`. The `NULL_REDUCER` is suitable for methods that return `void`. The `PROXY_INSTANCE_REDUCER` is a special marker for methods that return `this` — in our case, `proxy`.

Listing 6.9: `CompositeHandler`.

```
//:~ core/src/main/.../handlers/CompositeHandler.java
public final class CompositeHandler
    implements InvocationHandler {
  private final Class<?> component;
  private final Map<MethodKey, Reducer> reducers;
  private final Class<?>[] typeChecks;
  private final VTable defaultVT;
  private final Collection<Object> children = new ArrayList<>();
  private final Map<Class<?>, VTable> childMethodMap =
      new ConcurrentHashMap<>();

  @SuppressWarnings({"unchecked", "rawtypes"})
  public <E extends BaseComponent<? super E>> CompositeHandler(
      Class<? super E> component,
      Map<MethodKey, Reducer> reducers,
      Class<?>[] typeChecks) {
    if (!BaseComponent.class.isAssignableFrom(component))
      throw new IllegalArgumentException(
          "component is not derived from BaseComponent");
    this.component = component;
    this.reducers = Objects.requireNonNull(reducers);
    this.typeChecks = Objects.requireNonNull(typeChecks);
    this.defaultVT = VTables.newDefaultMethodVTable(component);
  }

  @Override
  public Object invoke(Object proxy,
                       Method method, Object[] args)
      throws Throwable {
    // Look for "add(Object)" and "remove(Object)" methods
    // from BaseComponent
    if (matches(method, "add")) {
      requiresAllInterfaces(args[0]);
```

```
      addChildMethods(args[0].getClass());
      return children.add(args[0]);
    } else if (matches(method, "remove")) {
      return children.remove(args[0]);
    }

    /**
     * This special class defined inside the method is only
     * visible inside the method. It is used to wrap and later
     * unwrap checked exceptions. We override fillInStackTrace()
     * to return null, so that we do not incur the cost of an
     * additional stack trace.
     */
    class UncheckedException extends RuntimeException {
      public UncheckedException(Throwable cause) {
        super(cause);
      }
      @Override
      public Throwable fillInStackTrace() { return null; }
    }

    // The purpose of the mapFunction is to convert checked
    // exceptions from our call to method.invoke() into
    // an UncheckedException, which we will unwrap later.
    // Unlike the reducers, we need to create a new lambda
    // each time we call the invoke() method, as we need to
    // capture the method and args parameters.
    Function<Object, Object> mapFunction = child -> {
      try {
        VTable vt = childMethodMap.get(child.getClass());
        assert vt != null : "vt==null";
        Method childMethod = vt.lookup(method);
        assert childMethod != null : "childMethod==null";
        return childMethod.invoke(child, args);
      } catch (ReflectiveOperationException e) {
        throw new UncheckedException(e);
      }
    };

    // The reducer is used to "reduce" results from method calls
    // to a single result.  By default we will use the
    // NULL_REDUCER, which always returns null.  This is
    // suitable for methods that return void.
```

```
    var reducer = reducers.getOrDefault(
        new MethodKey(method), Reducer.NULL_REDUCER);

    // Next try call the default interface method, if any.
    // This helps support the visitor pattern in the composite.
    MethodHandle match = defaultVT.lookupDefaultMethod(method);
    Object defaultMethodResult;
    if (match == null) {
      defaultMethodResult = reducer.getIdentity();
    } else {
      // invoke default interface method on component interface
      defaultMethodResult = match.invoke(proxy, args);
    }

    try {
      // We now need to call the method on all our children and
      // do a "reduce" on the results to return a single result.
      var merger = reducer.getMerger();
      var result = children.stream()
                       .map(mapFunction)
                       .reduce(reducer.getIdentity(), merger);
      // A special case of reducer is PROXY_INSTANCE_REDUCER.
      // When that is specified, we return the proxy instance
      // instead.  This is useful to support fluent interfaces
      // that return "this".
      if (reducer == Reducer.PROXY_INSTANCE_REDUCER)
        return proxy;
      else
        return merger.apply(result, defaultMethodResult);
    } catch (UncheckedException ex) {
      // Lastly we unwrap the UncheckedException and throw the
      // cause.
      throw ex.getCause();
    }
  }

  /**
   * We need the childMethodMap to support the visitor pattern
   * inside our composite structures
   */
  private void addChildMethods(Class<?> childClass) {
    childMethodMap.computeIfAbsent(childClass,
        clazz -> {
```

```
          Class<?> receiver;
          if (clazz.getModule().isExported(
              clazz.getPackageName(), component.getModule())) {
            // only map child class methods if its module is
            // and package are exported to the target module
            receiver = clazz;
          } else if (Proxy.class.isAssignableFrom(clazz)) {
            // childClass is a Proxy, use the first interface
            receiver = clazz.getInterfaces()[0];
          } else {
            receiver = component;
          }
          return VTables.newVTableExcludingObjectMethods(
              receiver, component);
        });
  }

  /**
   * Specific match for add(Object) and remove(Object) methods.
   */
  private boolean matches(Method method, String name) {
    return name == method.getName()
               && method.getParameterCount() == 1
               && ParameterTypesFetcher.get(method)[0]
                      == Object.class;
  }

  /**
   * Checks that object implements all required interfaces.
   */
  private void requiresAllInterfaces(Object arg) {
    for (var check : typeChecks) {
      if (!check.isInstance(arg))
        throw new ClassCastException(
            arg.getClass() + " cannot be cast to " + check);
    }
  }
}
```

The `Reducer` provides the two parameters for the `reduce()` method on `Stream` that we would like to use: an `identity` and a `BinaryOperator`.

Listing 6.10: `Reducer`.

```
//:~ core/src/main/.../handlers/Reducer.java
import java.util.function.*;

public final class Reducer {
  private final Object identity;
  private final BinaryOperator<Object> merger;
  public Reducer(Object identity,
                 BinaryOperator<Object> merger) {
    this.merger = merger;
    this.identity = identity;
  }
  public Object getIdentity() {
    return identity;
  }
  public BinaryOperator<Object> getMerger() {
    return merger;
  }
  public static final Reducer NULL_REDUCER =
      new Reducer(null, (o1, o2) -> null);
  /**
   * Result will be substituted with the proxy instance.
   */
  public static final Reducer PROXY_INSTANCE_REDUCER =
      new Reducer(null, (o1, o2) -> null);
}
```

We need to add four methods to our `Proxies` class for creating dynamic composites.

Listing 6.11: `Proxies.compose()`.

```
//:~ core/src/main/.../Proxies.java
public static <T extends BaseComponent<? super T>> T compose(
    Class<T> component) {
  return compose(component, component);
}
public static <T extends BaseComponent<? super T>> T compose(
    Class<T> component, Map<MethodKey, Reducer> reducers) {
  return compose(component, reducers, component);
}
public static <T extends BaseComponent<? super T>> T compose(
    Class<T> component, Class<?>... typeChecks) {
  return compose(component, Map.of(), typeChecks);
}
/**
 * @param component  interface to proxy. Must extend
 *                   BaseComponent
 * @param reducers   map from MethodKey to Reducer, default of
 *                   empty map with Map.of().
 * @param typeChecks object parameter passed to add() must
 *                   extend all these types
 */
public static <T extends BaseComponent<? super T>> T compose(
    Class<T> component, Map<MethodKey, Reducer> reducers,
    Class<?>... typeChecks) {
  return castProxy(component,
      new CompositeHandler(component, reducers, typeChecks));
}
```

To create a composite of `Contact`, we first define a `Map<MethodKey, Reducer>` for the `count()` method. We do not need a mapping for the `sendMail()` method, since its return type is `void`. We then create our new dynamic distribution list with `Proxies.compose(Contact.class, reducers)`.

Listing 6.12 is an example.

Listing 6.12: `ContactDynamicDemo`.

```
//:~ samples/src/main/.../ch06/contactdynamic/ContactDynamicDemo.java
var reducers = Map.of(
    new MethodKey(Contact.class, "count"),
    new Reducer(0, (r1, r2) -> (int) r1 + (int) r2)
);

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

Contact allStudents =
    Proxies.compose(Contact.class, reducers);
allStudents.add(new Person("peter@absa.co.za"));
allStudents.add(new Person("mzani@absa.co.za"));
javaSpecialistsNewsletter.add(allStudents);

javaSpecialistsNewsletter.sendMail("Hello there 2");
System.out.println(javaSpecialistsNewsletter.count());

Contact extremeJava =
    Proxies.compose(Contact.class, reducers);
extremeJava.add(new Person("John@fnb.co.za"));
extremeJava.add(new Person("Hlope@fnb.co.za"));
allStudents.add(extremeJava);

javaSpecialistsNewsletter.sendMail("Hello there 3");
System.out.println(javaSpecialistsNewsletter.count());
```

The output looks the same as for our handwritten `DistributionList` example.

Listing 6.13: `ContactDynamicDemo Output`

```
0
Sending Hello there 1 to john@aol.com
1
Sending Hello there 2 to john@aol.com
Sending Hello there 2 to peter@absa.co.za
Sending Hello there 2 to mzani@absa.co.za
3
Sending Hello there 3 to john@aol.com
Sending Hello there 3 to peter@absa.co.za
Sending Hello there 3 to mzani@absa.co.za
Sending Hello there 3 to John@standardbank.co.za
Sending Hello there 3 to Hlope@standardbank.co.za
5
```

AppendableFlushableCloseable

An interesting use case for composites is to write to several destinations at once, similar to the Unix `tee` utility. Here, we create class `TeeOutputStream`, as follows.

Listing 6.14: `TeeOutputStream`.

```
//:~ samples/src/main/.../ch06/appendableclass/TeeOutputStream.java
public class TeeOutputStream extends OutputStream {
  private final OutputStream out1, out2;
  public TeeOutputStream(OutputStream out1, OutputStream out2) {
    this.out1 = out1;
    this.out2 = out2;
  }
  @Override
  public void write(int b) throws IOException {
    out1.write(b);
    out2.write(b);
  }
  @Override
  public void flush() throws IOException {
    out1.flush();
    out2.flush();
  }
  @Override
  public void close() throws IOException {
    out1.close();
    out2.close();
  }
}
```

We can create a new `TeeOutputStream` that writes both to `System.out` and to a file. It is even possible to set `System.out` to write to our `TeeOutputStream`, as follows.

Listing 6.15: `TeeOutputStreamDemo`.

```
//:~ samples/src/main/.../ch06/appendableclass/TeeOutputStreamDemo.java
System.setOut(
    new PrintStream(
        new TeeOutputStream(
            System.out,
            new FileOutputStream("output.txt"))));

System.out.println("Hello World");
System.out.println("This will show on console and in file");
```

It would be convenient if we could create the `TeeOutputStream` as a dynamic composite. Unfortunately, `java.io.OutputStream` is not a Java interface, a hard requirement for creating

dynamic proxies. Similarly, `java.io.Writer` is a class and not an interface. However, `Writer` implements the `Appendable`, `Flushable`, and `Closeable` interfaces.

Handcrafted `TeeAppendable` composite

The type parameter `E` of `TeeAppendable` ensures that all children implement the three interfaces `Appendable`, `Flushable`, and `Closeable`. The `append()` methods return `this`, whereas `close()` and `flush()` return `void`. Additionally, our `TeeAppendable` gets `add()` and `remove()` methods that by default return `false`.

Listing 6.16: `TeeAppendable`.

```
//:~ samples/src/main/.../ch06/appendables/TeeAppendable.java
public class TeeAppendable<E extends
                                Appendable
                                & Flushable
                                & Closeable>
    implements Appendable, Flushable, Closeable {
  private final Collection<E> children = new ArrayList<>();

  public boolean add(E child) {
    return children.add(child);
  }

  public boolean remove(E child) {
    return children.remove(child);
  }

  @Override
  public Appendable append(CharSequence cs) throws IOException {
    for (var child : children) child.append(cs);
    return this;
  }
  @Override
  public Appendable append(CharSequence cs, int start, int end)
      throws IOException {
    for (var child : children) child.append(cs, start, end);
    return this;
  }
  @Override
  public Appendable append(char c) throws IOException {
    for (var child : children) child.append(c);
    return this;
  }
  @Override
  public void flush() throws IOException {
    for (var child : children) child.flush();
  }
  @Override
  public void close() throws IOException {
    for (var child : children) child.close();
  }
}
```

We can manually create an adapter that converts any class that implements all three of these interfaces into a `Writer`. Enter the `WriterAdapter`. Its adaptee is of type `E` where `E extends Appendable & Flushable & Closeable`. In plain English, this means that any class that implements all three of the `Appendable`, `Flushable`, and `Closeable` interfaces can be converted to a `Writer`. Once we do that, we can decorate it with a `PrintWriter` for all the fancy `println()` and `printf()` calls.

Listing 6.17: `WriterAdapter`.

```
//:~ samples/src/main/.../ch06/appendables/WriterAdapter.java
public class WriterAdapter<E extends Appendable
                                  & Flushable
                                  & Closeable>
    extends Writer {
  private final E adaptee;

  public WriterAdapter(E adaptee) {
    this.adaptee = adaptee;
  }
  @Override
  public void write(char[] cbuf, int off, int len)
      throws IOException {
    for (int i = off; i < off + len; i++) {
      adaptee.append(cbuf[i]);
    }
  }
  @Override
  public void flush() throws IOException {
    adaptee.flush();
  }
  @Override
  public void close() throws IOException {
    adaptee.close();
  }
}
```

`AppendableDemo` shows how to use the `TeeAppendable` in combination with the `WriterAdapter`. Of particular interest is the `var` keyword, introduced in Java version 10. Java is a strongly typed language. When we declare a local variable, we need to be precise about its type. In our case, we want to declare that the local variable must be all three interfaces of `Appendable`, `Flushable`, and `Closeable`. There was no way to do this before `var`.

Listing 6.18: `AppendableDemo`.

```
//:~ samples/src/main/.../ch06/appendables/AppendableDemo.java
// "var" is a lifesaver here.  The "correct" definition would
// be TeeAppendable<Appendable & Flushable & Closeable>, but
// that is not possible in Java.  "var" takes care of it.
var tee = new TeeAppendable<>();
var sw = new StringWriter();
tee.add(new OutputStreamWriter(System.out));
tee.add(new FileWriter("output.txt"));
tee.add(sw);

var out = new PrintWriter(new WriterAdapter<>(tee));
out.println("Hello World");
out.flush();

tee.append("TestingAppender")
    .append('\n')
    .append("Does this work?")
    .append('\n');
tee.flush();

System.out.println("sw = " + sw);
```

When we run this example, it writes the text to `System.out` and to a `StringWriter` called `sw` and a file called `output.txt`.

Listing 6.19: `AppendableDemo` *output.*

```
Hello World
TestingAppender
Does this work?
sw = Hello World
TestingAppender
Does this work?
```

Dynamic-proxy-powered `AppendableFlushableCloseable`

Now let's see how to build the same use case with dynamic proxies. In this case, we create an `AppendableFlushableCloseable` interface that extends the three `Appendable`, `Flushable`, `Closeable` interfaces, as well as `BaseComponent<E>`. We also provide a `getReducers()` method to make it easier to construct composites with our `Proxies` façade. The `append()` methods are supposed to return a pointer to the current object — in other words, `this`. Inside an `InvocationHandler`, the

pointer to the current proxy object is the `proxy` parameter. By specifying the `PROXY_INSTANCE_REDUCER` as a `Reducer`, we tell the `CompositeHandler` to return the `proxy` parameter once all the methods have been called on the children.

Listing 6.20: `AppendableFlushableCloseable`.

```
//:~
samples/src/main/.../ch06/appendables/AppendableFlushableCloseable.java
public interface AppendableFlushableCloseable<E extends
                                                Appendable
                                                & Flushable
                                                & Closeable>
    extends Appendable, Flushable, Closeable, BaseComponent<E> {
  static Map<MethodKey, Reducer> getReducers() {
    return Map.of(
        new MethodKey(Appendable.class, "append",
            CharSequence.class),
        Reducer.PROXY_INSTANCE_REDUCER,
        new MethodKey(Appendable.class, "append",
            CharSequence.class, int.class, int.class),
        Reducer.PROXY_INSTANCE_REDUCER,
        new MethodKey(Appendable.class, "append", char.class),
        Reducer.PROXY_INSTANCE_REDUCER);
  }
}
```

We can use the `AppendableFlushableCloseable` as an interface to create dynamic tee composites. Our sample code looks almost identical to that in Listing 6.18, except that we do not need to manually construct a `TeeAppendable`.

Listing 6.21: `AppendableDynamicDemo`.

```
//:~ samples/src/main/.../ch06/appendables/AppendableDynamicDemo.java
var tee = Proxies.compose(AppendableFlushableCloseable.class,
    AppendableFlushableCloseable.getReducers(),
    Appendable.class, Flushable.class, Closeable.class
);
var sw = new StringWriter();
tee.add(new OutputStreamWriter(System.out));
tee.add(new FileWriter("output.txt"));
tee.add(sw);

var out = new PrintWriter(new WriterAdapter<>(tee));
out.println("Hello World Dynamic");
out.flush();

tee.append("TestingAppenderDynamic")
    .append('\n')
    .append("Does this work?")
    .append('\n');
tee.flush();

System.out.println("sw = " + sw);
```

The output of the `AppendableDynamicDemo` is in Listing 6.22.

Listing 6.22: `AppendableDynamicDemo` *output.*

```
Hello World Dynamic
TestingAppenderDynamic
Does this work?
sw = Hello World Dynamic
TestingAppenderDynamic
Does this work?
```

CHAPTER SEVEN

Conclusion

We've seen dynamic proxy implementations of four similar patterns from the GoF: proxy, decorator, adapter, and composite.

We have also seen how we can achieve performance that's similar to that of handcrafted code.

There might be an additional cognitive load in understanding the way that dynamic proxies work and how we can apply them to our individual projects. But once we have invested in this learning, we are able to create a lot of functionality in relatively few lines of code. We have mastered a new Java superpower, one that has been used by countless frameworks over the past two decades.

A good way to learn is to step through the code, all of which is available in my public GitHub respository. I did not deliberately leave bugs in my code, although it is tempting to claim that I did, and I hope that you will join my 500+ reviewers in looking for bugs and reporting them.

Lastly, JavaSpecialists has created a self-study course in dynamic proxies in which I talk you through each of the sections. Each chapter has exercises to solve, together with model solutions. The course is also a great place to ask questions and meet other students.

Thank you for reading.

— Dr. Heinz M. Kabutz

About the Author

Dr. Heinz M. Kabutz (@heinzkabutz) is the author of the mildly entertaining and somewhat useful The Java Specialists' Newsletter. He is always happy to receive email on heinz@javaspecialists.eu.