

java.lang.reflect.*

Dr Heinz M. Kabutz

Last Updated 2022-08-14

Copyright Notice

© © 2022 Heinz Kabutz, All Rights Reserved

- No part of this course material may be reproduced without the express written permission of the author, including but not limited to: blogs, books, courses, public presentations.
- A license is hereby granted to use the ideas and source code in this course material for your personal and professional software development.
- No part of this course material may be used for internal company training

© Please contact heinz@javaspecialists.eu if you are in any way uncertain as to your rights and obligations.

1. Introduction to Reflection



1.1. Welcome

Who Am I?

© Dr Heinz Kabutz

- Born in Cape Town, South Africa, now lives on Crete
 - Founder of JCrete - See <https://www.jcrete.org>
- Created The Java Specialists' Newsletter
 - <https://www.javaspecialists.eu/archive/>
- One of the first Java Champions
 - <http://javachampions.org>



Questions

- ◎ **There are some stupid questions**

- They are the ones we did not ask
- Once asked, they are no longer stupid

- ◎ **The more you ask, the more we all learn**

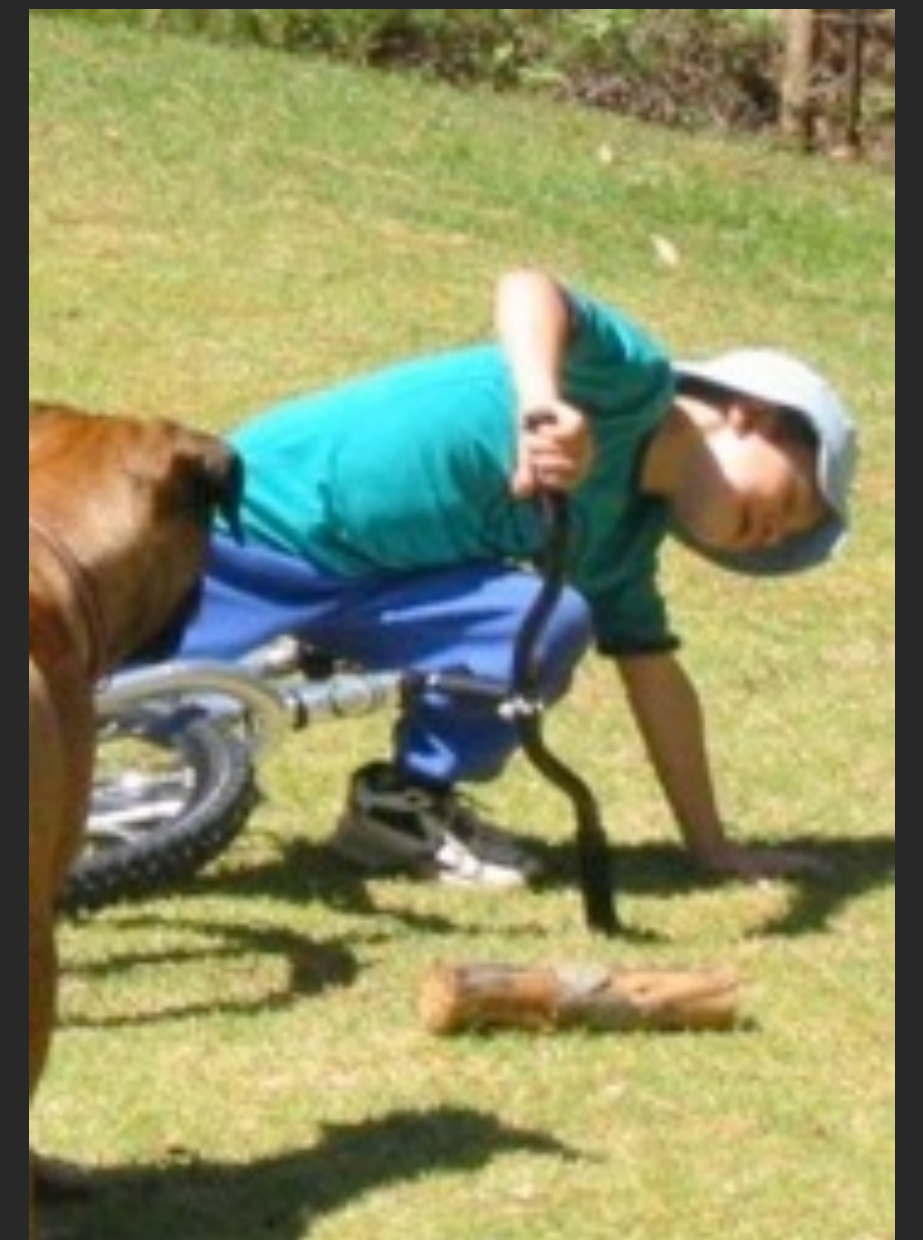
- And the more fun we will have today

- ◎ **We can ask questions in the group chat**

- Let's try it now to make sure that it works for you
 - How warm is it where you are at the moment? In C or F

Exercises

- ◎ **We learn cycling by "falling"**
 - Listening to lectures is not enough
- ◎ **Exercises help us to try out the ideas ourselves**
 - We have a git repository for this course
 - My work will be in the "kabutz" branch
 - Please don't push to master :-)



1.2. Why learn reflection?

1.2. Why learn reflection?

◎ Reflection is like Opium

- A bit too strong for every day use
- But can relieve serious pain
- Please do not become a Reflection Addict!

Reflection Introduction

- ◎ **We can discover what class an object is**
- ◎ **Once we know, we can**
 - call methods
 - read and write fields
 - construct new instances

Usefulness of Reflection

● Frameworks

- A lot of the magic behind the scenes happens with reflection
- Dependency injection, data serialization, dynamic configuration

● Serialization

- `ObjectOutputStream.writeObject()` uses reflection extensively

● A little bit of code can do a lot

Dangers of Reflection

- ◎ **Reflection can make code harder to understand**
 - No longer WYSIWYG (what you see is what you get)
- ◎ **Static code tools do not work anymore**

Popular interview question:
"Do you know reflection?"

*"Yes, I do. You can use it to
modify private final fields and
call methods dynamically."*

*"This interview is over.
Thanks for applying and
good luck for your future."*

1.3. Class Class

1.3. Class Class

◎ Each object in Java knows its class

- We can find out what it is with `Object.getClass()`
 - Note: we call the variable "clazz" because "class" is a keyword
 - Alternative names are `cls` and `klass`
- `getClass()` returns the actual object type, not the declaration

```
Collection<String> list = new ArrayList<>();  
var clazz = list.getClass();  
System.out.println("clazz = " + clazz);
```

```
clazz = class java.util.ArrayList
```


Who has a Class?

◎ Every type in Java has a representative Class

- Even primitive types, interfaces, enums, annotations!
- Instead of getClass(), we can use .class

```
System.out.println(Collection.class);  
System.out.println(String.class);  
System.out.println(int.class);  
System.out.println(Integer.class);  
System.out.println(void.class);  
System.out.println(Thread.State.class);
```

```
interface java.util.Collection  
class java.lang.String  
int  
class java.lang.Integer  
void  
class java.lang.Thread$State
```

Differentiating between class types

◎ Class has lots of methods for figuring out the type

```
isAnnotation()           // @Override
isAnonymousClass()       // (new Object() {}).getClass()
isArray()                 // char[]
isEnum()                  // Thread.State
isHidden()                 // Hidden classes Java 15+
isInterface()             // Collection.class
isLocalClass()            // Class defined inside a method
isMemberClass()           // Nested or inner class
isPrimitive()             // int.class
isRecord()                // record Person(String name) {}
isSealed()                // Executable.class
isSynthetic()             // lambdas and method references
Proxy.isProxyClass(Class<?> cl) // Dynamic Proxies
```


What is the type of Class?

◎ Class is a generic class, defined as Class<T>

- Objects' generic type parameters are erased at compile time
 - All ArrayList objects share the same class in Java
- int.class and Integer.class are both type Class<Integer>
 - But they are not the same object

```
Class<ArrayList> clazz1 = ArrayList.class; // raw type
Class<String> clazz2 = String.class;
Class<Integer> clazz3 = int.class;
Class<Integer> clazz4 = Integer.class;
Class<Void> clazz5 = void.class;
```

Name of Class

◎ Class Class can return "name" of class

- getName()
 - Oldest method, with strange output for arrays (more later)
- getSimpleName()
 - Java 5: strips away the package name
- getCanonicalName()
 - Java 5: Full class name, better array output, null for anon types
- getTypeName()
 - Java 8: Full class name, better array output, never null
 - This is the one we should probably use
- getPackageName()
 - Java 9: package in which class resides; java.lang for primitives

Superclass and interfaces

◎ Class gives us direct ancestor and interfaces

- Does not return inherited ancestor interfaces

```
interface A {}  
interface B extends A {}  
class C implements B {}  
interface D {}  
class E extends C implements D {}
```

```
System.out.println(  
    Arrays.toString(E.class.getInterfaces())); // [D]  
System.out.println(E.class.getSuperclass()); // C  
System.out.println(C.class.getSuperclass()); // Object  
System.out.println(Object.class.getSuperclass()); // null  
System.out.println(A.class.getSuperclass()); // null  
System.out.println(B.class.getSuperclass()); // null
```

What about `Class.forName()`?

◎ `Class.forName()` has three forms:

- `forName(String className)`
 - Most simple, all that is needed is the full class name
- `forName(String name, boolean initialize, ClassLoader loader)`
 - Most useful - we can avoid initializing until we need it and specify another class loader
- `forName(Module module, String name)`
 - Java 9 support for modules

◎ All `forName()` methods return a `Class<?>`

- Cast to the correct type using `asSubClass(Class)`

DemoClass

```
public class DemoClass implements Runnable {  
    static {  
        System.out.println("Initializing class");  
    }  
  
    public DemoClass() {  
        System.out.println("DemoClass object");  
    }  
  
    @Override  
    public void run() {  
        System.out.println("Running DemoClass");  
    }  
}
```

Using DemoClass with forName()

```
public class ClassForNameBasic {  
    public static void main(String... args)  
        throws ClassNotFoundException,  
        InstantiationException,  
        IllegalAccessException {  
        System.out.println("1");  
        Class<?> clazz = Class.forName(args[0]);  
        System.out.println("2");  
        Runnable demo = (Runnable) clazz.newInstance();  
        System.out.println("3");  
        demo.run();  
    }  
}  
  
// java ClassForNameBasic DemoClass
```

1
Initializing class
2
DemoClass object
3
Running DemoClass

What was wrong with previous code?

◎ There were a few things not that good:

- Too many checked exceptions hurt our eyes
- Compiler warning when we cast to `Class<DemoClass>`
- Class initialized before we check it is the correct type
 - And we are perhaps using the wrong class loader
- `Class.newInstance()` is deprecated

ReflectiveOperationException

◎ **ReflectiveOperationException has 7 subclasses**

- ClassNotFoundException - Class.forName() did not resolve
- IllegalAccessException - trying to access a private member
- InstantiationException - trying to instantiate an abstract type
- NoSuchFieldException - field by that name not found
- NoSuchMethodException - method / constructor not found
- InvocationTargetException - wrapper for another exception

◎ **Better to simply throw ReflectiveOperationException**

Using ReflectiveOperationException

```
public class ClassForNameBetterException {  
    public static void main(String... args)  
        throws ReflectiveOperationException {  
        System.out.println("1");  
        Class<?> clazz = Class.forName(args[0]);  
        System.out.println("2");  
        Runnable demo = (Runnable) clazz.newInstance();  
        System.out.println("3");  
        demo.run();  
    }  
}
```

Getting rid of the unchecked cast

◎ We usually know the super-type of `Class.forName()`

- Instead of casting directly, let's use `asSubclass()`
 - We pass in `Class<U>` and it returns `Class<? extends U>`

```
Class<? extends Runnable> clazz =  
    Class.forName(args[0])  
        .asSubclass(Runnable.class);  
Runnable demo = clazz.newInstance();
```


Class initialized before cast

- ◎ **Class.forName(String) calls all the static initializers**
 - This happens before asSubclass can check the type
 - Better is to load the class uninitialized
 - It will be initialized before we use it for the first time
- ◎ **By default loaded into the current class' class loader**
 - Better to use Thread.currentThread().getContextClassLoader()
 - Allows frameworks to inject different class loaders

Better way of loading unknown class

```
public class ClassForNameBetter {  
    public static void main(String... args)  
        throws ReflectiveOperationException {  
        System.out.println("1");  
        Class<?> clazz = Class.forName(args[0], false,  
            Thread.currentThread().getContextClassLoader());  
        Class<? extends Runnable> runnableClazz =  
            clazz.asSubclass(Runnable.class);  
        System.out.println("2");  
        Runnable demo = runnableClazz.newInstance();  
        System.out.println("3");  
        demo.run();  
    }  
}
```

```
1  
2  
Initializing class  
DemoClass object  
3  
Running DemoClass
```


Deprecation warning on newInstance()

- © We will explain that in a bit

Exercise 1A:

◎ Find all distinct interfaces implemented by an object

```
public class LinkedList<E>  
    extends AbstractSequentialList<E>  
    implements List<E>, Deque<E>, Cloneable, Serializable
```

– But recursively, LinkedList implements

```
Iterable<E>, Collection<E>, List<E>, Queue<E>,  
Deque<E>, Cloneable, Serializable
```

1.3.1. Method

1.3.1. Method

◎ Class returns Method objects

- `getMethod(String name, Class<?>... parameterTypes)`

```
public class MethodFromClass {  
    public static void main(String... args)  
        throws ReflectiveOperationException{  
        Class<String> clazz = String.class;  
        // Might throw NoSuchMethodException  
        Method method = clazz.getMethod("toUpperCase");  
        // Might throw InvocationTargetException or  
        //      IllegalAccessException  
        String result =  
            (String) method.invoke("Hello World!");  
        System.out.println("result = " + result);  
    }  
}
```

result = HELLO WORLD!

Exceptions from `getMethod()` & `invoke()`

◎ We face several `ReflectionOperationExceptions`:

- `NoSuchMethodException`: A method with that name and parameters was not found
- `InvocationTargetException`: An exception (checked or unchecked) occurred when calling method. This wraps it.
- `IllegalAccessException`: The method is not public we can't call it
 - Strange though - `getMethod()` only returns public methods!

NoSuchMethodException

© **ArrayDeque** does not have method **get(int)**

```
var clazz = ArrayDeque.class;
try {
    clazz.getMethod("get", int.class); // from List
} catch (NoSuchMethodException e) {
    System.out.println("Could not find method get(int)");
}
```

Could not find method get(int)

InvocationTargetException

◎ pop() might throw NoSuchElementException

```
var clazz = ArrayDeque.class;
Deque<Integer> deque = new ArrayDeque<>();
try {
    Method pop = clazz.getMethod("pop");
    pop.invoke(deque);
} catch (InvocationTargetException e) {
    System.out.println("pop() caused a " + e.getCause());
}
```

```
pop() caused a java.util.NoSuchElementException
```

IllegalAccessException

© Why would this throw an IllegalAccessException?

- hasNext() is a public interface method

```
Deque<Integer> deque = new ArrayDeque<>();  
Iterator<Integer> iterator = deque.iterator();  
var hasNext = iterator.getClass().getMethod("hasNext");  
try {  
    System.out.println(hasNext.invoke(iterator));  
} catch (IllegalAccessException e) {  
    System.out.println("hasNext() is not accessible!");  
}
```

```
hasNext() is not accessible!
```

Private nested classes

◎ Methods in private nested classes aren't accessible

- `getMethod()` sometimes gives us these inaccessible methods
 - Solution in www.javaspecialists.eu/archive/Issue273.html

◎ Better to call `getMethod()` on Iterator interface:

```
Deque<Integer> deque = new ArrayDeque<>();
Iterator<Integer> iterator = deque.iterator();
var hasNext = Iterator.class.getMethod("hasNext");
try {
    System.out.println(hasNext.invoke(iterator));
} catch (IllegalAccessException e) {
    System.out.println("hasNext() is not public!");
}
```

false

Modifiers

- ◎ **Modifiers such as public / static / final are in a bitset**
 - e.g. public is 0b0001, private is 0b0010, protected 0b0100
- ◎ **Best to use them with java.lang.reflect.Modifier**
 - e.g. Modifier.isPublic(method.getModifiers())

isPublic	isPrivate	isProtected
isAbstract	isFinal	isStatic
isSynchronized	isVolatile	isTransient
isNative	isStrict	isInterface

Mystery of non-accessible hasNext()

◎ Whilst hasNext() is public, the class is not

```
Deque<Integer> deque = new ArrayDeque<>();
Iterator<Integer> iterator = deque.iterator();
var clazz = iterator.getClass();
var hasNext = clazz.getMethod("hasNext");
System.out.println("hasNext() public? " +
    Modifier.isPublic(hasNext.getModifiers())
);
System.out.println("Iterator class public? " +
    Modifier.isPublic(clazz.getModifiers())
);
```

```
hasNext() public? true
Iterator class public? false
```

Finding all methods

- ◎ **We can also get all methods with `getMethods()`**
 - It also shows inherited public methods
 - Methods are not ordered in any particular way

OutputStream.class.getMethods()

```
public void java.io.OutputStream.flush() throws java.io.IOException
public void java.io.OutputStream.write(byte[]) throws java.io.IOException
public void java.io.OutputStream.write(byte[],int,int) throws java.io.IOException
public abstract void java.io.OutputStream.write(int) throws java.io.IOException
public void java.io.OutputStream.close() throws java.io.IOException
public static java.io.OutputStream java.io.OutputStream.nullOutputStream()
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```


Co-variant return types

◎ Methods are distinguished by name and parameters

- A class can have identical methods with different return types
- e.g. `ArrayDeque.clone`:

```
public java.util.ArrayDeque java.util.ArrayDeque.clone()  
public java.lang.Object java.util.ArrayDeque.clone()  
    throws java.lang.CloneNotSupportedException
```

- `ArrayDeque.class.getMethod("clone")` returns correct method:

```
public java.util.ArrayDeque java.util.ArrayDeque.clone()
```

Methods with parameters

◎ Adding items to a collection:

```
List<String> names = new ArrayList<>();  
Method add = List.class.getMethod("add", Object.class);  
add.invoke(names, "John");  
add.invoke(names, "Anton");  
add.invoke(names, "Heinz");  
System.out.println("names = " + names);
```

```
names = [John, Anton, Heinz]
```


Careful with varargs

◎ This throws an IllegalStateException

```
add.invoke(names, null);
```

- Instead, we can cast null to Object, then it's considered as null

```
add.invoke(names, (Object)null);
```

- Or we can add the explicit Object[] creation

```
add.invoke(names, new Object[]{null});
```

Static methods

◎ We pass in "null" instead of the instance

```
List<String> strings = new ArrayList<>();
Collections.addAll(strings, args);
System.out.println(strings);
for (Method m : Collections.class.getMethods()) {
    if (m.getParameterCount() == 1
        && m.getReturnType() == void.class
        && m.getParameterTypes()[0] == List.class
    ) {
        System.out.println(m.getName());
        m.invoke(null, strings);
        System.out.println(strings);
    }
}
```

```
[3, 1, 4, 1, 5, 9]
reverse
[9, 5, 1, 4, 1, 3]
sort
[1, 1, 3, 4, 5, 9]
shuffle
[1, 1, 4, 3, 9, 5]
```

Parameter names

- ◎ **Java does not consider parameter names important**
 - Traditionally they are not stored in the class files
 - Only the parameter type and their order matter
 - This makes it tricky to introduce named and default parameters
- ◎ **Include names by compiling with -parameters flag**

Parameter names

```
public class ParameterNames {  
    public void names(String hello, boolean world) {}  
    public static void main(String... args)  
        throws ReflectiveOperationException {  
        Method names = ParameterNames.class.getMethod(  
            "names", String.class, boolean.class);  
        Arrays.stream(names.getParameters())  
            .map(Parameter::getName)  
            .forEach(System.out::println);  
    }  
}
```

```
// without -parameters javac flag:  
arg0  
arg1  
// with -parameters javac flag:  
hello  
world
```

Exercise 1B:

◎ Write a ListMangler for Collections methods `sort(List)`, `reverse(List)`, `shuffle(List)`

- Should return a new immutable list and not change input list
- Find the method and call it on our copy of the input list
- More instructions in the comments of the class

```
public class ListMangler {  
    public static List<String> reorder(  
        List<String> list, String methodName) {  
        // TODO  
    }  
}
```


1.3.2. Constructor

1.3.2. Constructor

- ◎ **Constructor is a special case of a method**

- We do not specify a name, but do specify the parameter types
- Even the exceptions are the same - NoSuchMethodException

- ◎ **Instead of calling `invoke()`, we call `newInstance()`**

```
private static final Collection<Integer> PI =  
    List.of(3, 1, 4, 1, 5, 9, 265, 358);  
  
public static void createCollection(String name)  
    throws ReflectiveOperationException {  
    Class<? extends Collection> clazz =  
        Class.forName(name)  
            .asSubclass(Collection.class);  
    System.out.println(clazz.getSimpleName());  
  
    Constructor<? extends Collection> noArgs =  
        clazz.getConstructor();  
    System.out.println(noArgs.newInstance());  
  
    Constructor<? extends Collection> copy =  
        clazz.getConstructor(Collection.class);  
    System.out.println(copy.newInstance(PI));  
}
```

ArrayDeque

[]

[3, 1, 4, 1, 5, 9, 265, 358]

What about `getConstructors()`

- ◎ **Similarly to `getMethods()`, but not as useful**
 - Constructors are differentiated only by parameter types
 - If we know the type we are looking for, just use `getConstructor()`
- ◎ **Unlike `getMethods()`, does not include superclasses**
- ◎ **Returns all the public constructors**

Defensive copies in reflection API

- ◎ **Mutable objects in reflection are returned as copies**
 - For example, `Method.getParameterTypes()` is an array
 - We can change these without affecting safety in other code

Class.newInstance()?

◎ Inside the bowels of this method we see

```
try {  
    Class<?> caller = Reflection.getCallerClass();  
    return getReflectionFactory().newInstance(  
        tmpConstructor, null, caller);  
} catch (InvocationTargetException e) {  
    Unsafe.getUnsafe().throwException(e.getTargetException());  
    // Not reached  
    return null;  
}
```

Class.newInstance() abuse

```
public class UncheckedThrower {  
    private static final ThreadLocal<Throwable> throwable =  
        new ThreadLocal<>();  
    public UncheckedThrower() throws Throwable {  
        Throwable t = throwable.get();  
        throwable.remove();  
        throw t;  
    }  
    @SuppressWarnings("deprecation")  
    public static void uncheckedThrow(Throwable t) {  
        throwable.set(t);  
        try {  
            UncheckedThrower.class.newInstance();  
        } catch (ReflectiveOperationException ignored) {}  
    }  
}
```


Do not use deprecated newInstance()

© Instead, get the constructor, then call newInstance()

Exercise 1C: MagicClassInstantiator

- ◎ **Taking the following command line parameters:**
 - `ClassName [Parameter1 ... ParameterN] MethodName`
 - `java.lang.String "Hello World" toUpperCase` → `HELLO WORLD`
 - `java.util.ArrayDeque poll` → `null`
- ◎ **Write a MagicClassInstantiator that:**
 - Loads the class specified in `ClassName`
 - Finds the constructor taking correct number of parameters
 - You may assume all parameters are `String.class`
 - Constructs the object
 - Calls the method (with no parameters)
 - If the return type is not void, prints the result to the console

1.3.3. Generics

1.3.3. Generics

1.4. Nested classes

1.4. Nested classes

- ◎ **We can discover nested classes `Class.getClasses()`**
 - But not local or anonymous classes
 - As before, only public classes are returned with `getClass()`


```
public class SuperNestedClassDemo {  
    public static class SuperNestedClass {}  
}  
  
public class NestedClassDemo extends SuperNestedClassDemo {  
    public static class PublicStaticNestedClass {}  
    public class PublicInnerClass {}  
    class PackageAccessInnerClass {}  
    private class PrivateInnerClass {}  
    protected class ProtectedInnerClass {}  
    public final Object anonInner = new Object() {};  
    public static final Object anonNested = new Object() {};  
  
    public static void main(String... args) {  
        Object anonLocal = new Object() {};  
        class Local {}  
        Arrays.stream(NestedClassDemo.class.getClasses())  
            .map(Class::getSimpleName)  
            .forEach(System.out::println);  
    }  
}
```

PublicInnerClass
PublicStaticNestedClass
SuperNestedClass

1.5. Sealed classes

1.5. Sealed classes

© Java 15 introduced sealed classes

- Limits who can subclass
- Also allows us to discover all subclasses of a class via reflection

All permitted subclasses

◎ Returns only the most direct descendants

```
public class SealedClassExplorer {  
    public static void showSubclasses(Class<?> clazz) {  
        System.out.println("Subclasses of " + clazz);  
        Arrays.stream(clazz.getPermittedSubclasses())  
            .map(Class::getName)  
            .forEach(System.out::println);  
    }  
}
```

Subclasses of Executable

◎ Returns only the most direct descendants

```
SealedClassExplorer.showSubclasses(Executable.class);
```

```
Subclasses of class java.lang.reflect.Executable  
java.lang.reflect.Constructor  
java.lang.reflect.Method
```


Subclasses of ConstantDesc

◎ Missing quite a few subclasses this time

```
SealedClassExplorer.showSubclasses(ConstantDesc.class);
```

```
Subclasses of interface java.lang.constant.ConstantDesc  
java.lang.constant.ClassDesc  
java.lang.constant.MethodHandleDesc  
java.lang.constant.MethodTypeDesc  
java.lang.Double  
java.lang.constant.DynamicConstantDesc  
java.lang.Float  
java.lang.Integer  
java.lang.Long  
java.lang.String
```


Exercise 1D: Find *all* permitted classes

◎ Implement the following class

- Needs to find *all* permitted subclasses

```
public class PermittedSubclassExplorer {  
    /**  
     * Return all permitted subclasses underneath a root  
     * class or interface. The root must be a sealed type.  
     *  
     * @param root class to find permitted subtypes under  
     * @return a set of all permitted subclasses  
     * @throws IllegalArgumentException if root isn't sealed  
     */  
    public static Set<Class<?>> find(Class<?> root) {  
        throw new UnsupportedOperationException("TODO");  
    }  
}
```

1.6. Records

1.6. Records

◎ Java 16 records provide simplified data object

◎ e.g. Class PersonBean has fields and a constructor

```
public class PersonBean {  
    private final String firstName;  
    private final String lastName;  
    private final int age;  
  
    public PersonBean(String firstName, String lastName,  
                      int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
}
```


More plumbing

◎ An equals() and hashCode() methods

```
@Override
public boolean equals(Object o) {
    return o instanceof PersonBean p
        && age == p.age
        && Objects.equals(firstName, p.firstName)
        && Objects.equals(lastName, p.lastName);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(firstName, lastName, age);
}
```

Standard methods ...

◎ Don't forget the toString() method

```
@Override
public String toString() {
    return "PersonBean{" +
        "firstName='" + firstName + '\\\'' +
        ", lastName='" + lastName + '\\\'' +
        ", age=" + age +
        '}';
}
```

And more ...

◎ Getters for all the fields

```
public String getFirstName() {  
    return firstName;  
}
```

```
public String getLastName() {  
    return lastName;  
}
```

```
public int getAge() {  
    return age;  
}
```

```
}
```


Plus more ...

- ◎ **And maybe the class also has to be Serializable**
 - With a serialVersionUID

Records simplify this to

```
public record Person(String firstName,  
                    String lastName,  
                    int age) {}
```

RecordComponents

◎ We can also get all the record components

```
public static <T extends Record> void showRecordDetails(
    Class<T> clazz) throws NoSuchMethodException {
    System.out.println("record " + clazz.getSimpleName());
    Arrays.stream(clazz.getRecordComponents())
        .map(Object::toString)
        .map(s -> "\t" + s)
        .forEach(System.out::println);
}
```

```
record Person
    java.lang.String firstName
    java.lang.String lastName
    int age
```


We can find "canonical" constructor

◎ The RecordComponents are in the correct order

```
static <T extends Record>
Constructor<T> getCanonicalConstructor(Class<T> clazz)
    throws NoSuchMethodException {
    Class<?>[] paramTypes =
        Arrays.stream(clazz.getRecordComponents())
            .map(RecordComponent::getType)
            .toArray(Class<?>[]::new);
    return clazz.getDeclaredConstructor(paramTypes);
}
```

JavaBean convention

- ◎ **The JavaBean convention used getters and setters**
 - `getFirstName()`, `getLastName()`, `getAge()`
- ◎ **Booleans are prepended with "is"**
 - `isMarried()`
- ◎ **What if we have a property called "iPhone"?**
 - `getIphone()`
 - But what if the actual property is called IMEI?
 - `getIMEI()`?
- ◎ **Since records are part of the Java Language, they decided that it is best to have property == getter**

JavaBean Introspector

◎ Old legacy approach to reading JavaBean properties

- Converts getXyz() and setXyz() to property xyz
 - Provides getter and setter methods that match property

```
BeanInfo bi = Introspector.getBeanInfo(PersonBean.class);
var beanClass = bi.getBeanDescriptor().getBeanClass();
System.out.println(beanClass.getSimpleName());
for (var propertyDesc : bi.getPropertyDescriptors()) {
    System.out.printf("%s -> %s()%n",
        propertyDesc.getName(),
        propertyDesc.getReadMethod().getName());
}
```

```
PersonBean
age -> getAge()
class -> getClass()
firstName -> getFirstName()
lastName -> getLastName()
```


Exercise 1E: Convert beans to records

◎ We need to convert between JavaBean and records

```
/**
 * Given the input bean, create a record instance of type
 * recordType with all properties set in the constructor.
 */
public static <T, R extends Record>
R convertBeanToRecord(T bean, Class<R> recordType) {}

/**
 * Given the input record, create a bean instance of type
 * beanType with all properties set using setters.
 */
public static <T extends Record, R>
R convertRecordToBean(T record, Class<R> beanType) {}
```

2. Deep Reflection

2. Deep Reflection

- ◎ **Up to now, we have only used "shallow reflection"**
 - 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.
 - [Mark Reinhold on JSR 376 Expert Group mailing list]

In layman's terms

◎ Shallow reflective access

- Only use public members, never violating encapsulation

◎ Deep reflective access

- Use `setAccessible(true)` to also access private members

Use cases for "deep reflection"

- ◎ **Broken encapsulation mainly for accessing fields**
 - Fields are almost always private
- ◎ **Get hold of `sun.misc.Unsafe`**
- ◎ **Serialization and similar mechanisms**

2.1. Making private members accessible

2.1. Making private members accessible

© Let's try create an object without calling constructor

```
public class Chatty {  
    public Chatty() {  
        System.out.println("Creating chatty object");  
        System.out.println("Hi there!");  
        System.out.println("You look great!");  
        System.out.println("Have you been working out?");  
    }  
  
    public void run() {  
        System.out.println("Chatty running");  
    }  
}
```

Chatty is annoying

◎ Creepy friendly

```
public class ChattyObjectDemo1 {  
    public static void main(String... args) {  
        Chatty chatty = new Chatty();  
        chatty.run();  
    }  
}
```

```
Creating chatty object  
Hi there!  
You look great!  
Have you been working out?  
Chatty running
```

Reflection does not help

◎ We still hear the overly friendly dialogue

```
public class ChattyObjectDemo2 {  
    public static void main(String... args)  
        throws ReflectiveOperationException {  
        var constructor = Chatty.class.getConstructor();  
        Chatty chatty = constructor.newInstance();  
        chatty.run();  
    }  
}
```

```
Creating chatty object  
Hi there!  
You look great!  
Have you been working out?  
Chatty running
```


Easiest to create object with Unsafe

◎ Let's try with `Unsafe.allocateInstance(Class<?>)`

```
import sun.misc.Unsafe;
```

```
public class ChattyObjectDemo3 {  
    public static void main(String... args)  
        throws InstantiationException {  
        Chatty chatty = (Chatty) Unsafe.getUnsafe()  
            .allocateInstance(Chatty.class);  
        chatty.run();  
    }  
}
```

```
Exception in thread "main" java.lang.SecurityException: Unsafe  
    at jdk.unsupported/sun.misc.Unsafe.getUnsafe(Unsafe.java:99)  
    at reflection.ch2.ChattyObjectDemo3.main(ChattyObjectDemo3.java:8)
```

sun.misc.Unsafe only for trusted use

```
private static final Unsafe theUnsafe = new Unsafe();

/**
 * Provides the caller with the capability of performing
 * unsafe operations.
 *
 * @throws SecurityException if the class loader of the
 *         caller class is not in the system domain in
 *         which all permissions are granted.
 */
@CallerSensitive
public static Unsafe getUnsafe() {
    Class<?> caller = Reflection.getCallerClass();
    if (!VM.isSystemDomainLoader(caller.getClassLoader()))
        throw new SecurityException("Unsafe");
    return theUnsafe;
}
```

How can we get theUnsafe instance?

- ◎ Let's first find the field with getField(name)

```
Field field = Unsafe.class.getDeclaredField("theUnsafe");
```

- ◎ Next we make it accessible

```
field.setAccessible(true);
```

- ◎ Then we read the value

```
ourUnsafe = (Unsafe) field.get(null);
```


Let's create our own UnsafeProvider

```
public class UnsafeProvider {  
    private static final Unsafe ourUnsafe;  
    static {  
        try {  
            Field field = Unsafe.class.  
                getDeclaredField("theUnsafe");  
            field.setAccessible(true);  
            ourUnsafe = (Unsafe) field.get(null);  
        } catch (ReflectiveOperationException e) {  
            throw new ExceptionInInitializerError(e);  
        }  
    }  
    public static Unsafe getUnsafe() {  
        return ourUnsafe;  
    }  
}
```

Finally, blissful silence!

◎ Can allocate instances without calling constructor

- No fields would be initialized either

```
public class ChattyObjectDemo4 {  
    private static final Unsafe ourUnsafe =  
        UnsafeProvider.getUnsafe();  
  
    public static void main(String... args)  
        throws InstantiationException {  
        Chatty chatty = (Chatty) ourUnsafe  
            .allocateInstance(Chatty.class);  
        chatty.run();  
    }  
}
```

Chatty running

Fields, fields, fields, fields, fields, ...

◎ Most common use case for deep reflection

- Even for our UnsafeProvider, we read the private theUnsafe field

◎ Fields are usually private, and preferably final

- The purpose of private fields is for encapsulation
 - Makes it easier to reason about *who* is reading / writing them
- The purpose of final fields is safety
 - Cannot be changed accidentally
 - State is exactly what we wrote when we constructed the object
 - Concurrency is easier

Danger of reading / writing fields

◎ With encapsulated fields we can protect invariants

```
public class Person {  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        if (age < 0) throw new IllegalArgumentException();  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return name + " is " + age + " years old";  
    }  
}
```

Constructor protects field

© Cannot create a Person with negative age

```
Person wally = new Person("Wally", -42);  
System.out.println("wally = " + wally);
```

```
java.lang.IllegalArgumentException  
    at reflection.ch2.Person.<init>(Person.java:11)  
    at reflection.ch2.PersonDemo.main(PersonDemo.java:9)
```

But we can with reflection

- ◎ We change age, even though it is private and final

```
Person heinz = new Person("Heinz", 50);  
System.out.println("heinz = " + heinz);  
Field ageField = Person.class.getDeclaredField("age");  
ageField.setAccessible(true);  
ageField.set(heinz, -57);  
System.out.println("heinz = " + heinz);
```

```
heinz = Heinz is 50 years old  
heinz = Heinz is -57 years old
```


But isn't that dangerous?

◎ Yes, it is very dangerous

- Final fields are assumed to be final, and can be inlined
 - Either at compile time or during execution
- We should never change a final field after it is published
- With `setAccessible(true)`, we get rid of private **and** final

◎ We cannot change static final fields

- A static field is assumed published when the class is loaded
 - It would thus always be unsafe to allow this to be changed
- We *can* change a static final field with `sun.misc.Unsafe`
 - But it is highly recommended that we do not do that

Use case of writing private final fields

◎ **Serialization writes objects to a stream**

- It reads the private fields, bypassing the security manager

◎ **When it reads them back from a stream**

- It first creates the object, without calling the constructor
 - But calling its superclass constructor
 - `sun.reflect.ReflectionFactory.newConstructorForSerialization()`
- It then writes the fields, whether they are private/final or not

◎ **The object is not published until all fields are written**

Imagine a badly written Animal class

◎ "legs" should definitely have been configurable

```
public class Animal {  
    private final int legs = 4;  
    private final String genus, species;  
  
    public Animal(String genus, String species) {  
        this.genus = genus;  
        this.species = species;  
    }  
    @Override  
    public String toString() {  
        return "Animal %s %s with %d legs".formatted(  
            genus, species, legs);  
    }  
}
```


Trying to change legs with reflection

© No exception, but it also seems to have no effect!

```
public class AnimalDemo {  
    public static void main(String... args)  
        throws ReflectiveOperationException {  
        Animal human = new Animal("homo", "sapiens");  
        Field legs =  
            Animal.class.getDeclaredField("legs");  
        legs.setAccessible(true);  
        legs.set(human, 2);  
        System.out.println(human);  
        System.out.println("legs = " + legs.get(human));  
    }  
}
```


```
Animal homo sapiens with 4 legs  
legs = 2
```

Did the field value change or not?

◎ Yes it did!

- However, javac inlined the value of "legs" to where it was used
- Thus we do not see the change, except via reflection

```
public String toString() {  
    return "Animal %s %s with %d legs".formatted(  
        this.genus, this.species, 4);  
}
```



Reading and writing record properties

◎ We do not need deep reflection for reading records

```
PersonRecord heinz = new PersonRecord("Heinz", 50);
System.out.println("From accessors:");
for (RecordComponent component :
     PersonRecord.class.getRecordComponents()) {
    System.out.println(component.getName() + ": " +
                       component.getAccessor().invoke(heinz));
}
System.out.println("From fields:");
for (Field field :
     PersonRecord.class.getDeclaredFields()) {
    field.setAccessible(true);
    System.out.println(field.getName() + ": " +
                       field.get(heinz));
}
```

```
From accessors:
name: Heinz
age: 50
From fields:
name: Heinz
age: 50
```


"Final" is enforced for record

- ◎ We cannot write to fields, even with deep reflection

```
try {  
    PersonRecord heinz = new PersonRecord("Heinz", 50);  
    System.out.println("heinz = " + heinz);  
    var age = PersonRecord.class.getDeclaredField("age");  
    age.setAccessible(true);  
    age.set(heinz, -57); // does not work for records  
    System.out.println("heinz = " + heinz);  
} catch (ReflectiveOperationException e) {  
    System.out.println(e);  
}
```

heinz = Heinz is 50 years old

java.lang.IllegalAccessException: Can not set final
int field PersonRecord.age to java.lang.Integer

Private members in other modules

© Let's change the Integer cache

```
public class IntegerHack {  
    public static void main(String... args)  
        throws ReflectiveOperationException {  
        Field value = Integer.class  
            .getDeclaredField("value");  
        value.setAccessible(true);  
        value.set(42, 43);  
        for (int i = 1; i <= 10; i++) {  
            System.out.printf("%d x 7 = %d%n",  
                i, i * 7);  
        }  
    }  
}
```


Java 8 result

◎ Works exactly as expected

Java 8:

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 43
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```


Java 11 result

◎ Warns us that we should fix the code

Java 11:

WARNING: An illegal reflective access operation has occurred

WARNING: Illegal reflective access by IntegerHack

WARNING: Please consider reporting this to the maintainers of
IntegerHack

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 43
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```


Java 17 result

◎ Crashes with an InaccessibleObjectException

Java 17:

```
Exception in thread "main" InaccessibleObjectException: Unable
to make field private final int java.lang.Integer.value
accessible: module java.base does not "opens java.lang"
to unnamed module @4dca0ecd
```

◎ We can run the code with the JVM parameter

`--add-opens java.base/java.lang=ALL-UNNAMED`

– (Instead of ALL-UNNAMED, use module name if using JPMS)

getDeclaredXYZ() vs getXYZ()

◎ Classes can have several members

- Annotations, nested classes, constructors, fields and methods
- The get() methods would only return public members
 - Includes all public members of similar type from superclasses
 - Note that "public" does not mean that method is accessible
 - We saw before that a class itself can be inaccessible
- The getDeclared() methods return all members of that class only
 - Includes private, protected, package access and public members
 - But only from the current class, not from superclasses

Exercise 2A:

- ◎ **Make `System.out.println("Hello World")` print out "Goodbye, cruel world" by using deep reflection on `java.lang.String`**
 - Run with JVM option
 - `--add-opens java.base/java.lang=ALL-UNNAMED`

Exercise 2B:

- ◎ Return all the methods of a class, including private, protected, package private methods:

```
/**
 * Returns a set of all the methods defined on clazz and
 * its superclasses, including private, protected,
 * public, and package private methods.
 *
 * @param includeObjectMethods true if the Set should
 *                               include Object's methods
 */
public static Set<Method> findAllMethods(
    Class<?> clazz, boolean includeObjectMethods) { }
```

2.2. Dealing with exceptions

2.2. Dealing with exceptions

2.3. Performance considerations

2.3. Performance considerations

◎ Reflection used to be quite slow

- But nowadays, with escape analysis and inlining, it is not too bad
- There might be a small method call overhead from allocation
 - Boxing of parameters
 - Vararg array and boxed return values with dynamic proxies
 - See our Dynamic Proxies in Java Course
 - Extra wrapping of exceptions
- Additional costs from security checks
 - May happen every time we call the method
 - Can turn off by calling `setAccessible(true)`, even on public methods
- Actual method invocation speed is the same as calling it directly

Counter class

```
public class Counter {  
    private long total;  
  
    public void reset() {  
        total = 0;  
    }  
    public void add(int value) {  
        total += value;  
    }  
    public long get() {  
        return total;  
    }  
    public void thrower() {  
        throw new IllegalStateException();  
    }  
}
```

Boxing of parameters

- ◎ **Primitive wrappers for long/int/short/byte cached**
 - At least values -128 to 127 have to be cached
 - Outside of that range, wrapper objects might be created
- ◎ **Return types cast to primitives do not allocate bytes**

Throwing exceptions costs twice

- ◎ **Every exception has a stack trace**
 - And then is wrapped again in an `InvocationTargetException`
 - Which also has a stack trace

Experiment

◎ Calling add(int), get(), throwing() for 10 seconds

– Times in nanoseconds

	Direct	Method	Method Fast	MethodHandle
add(0..100)	5.5	8.4 16 bytes	7.9 16 bytes	5.6
add(200..1000)	5.7	8.1 16 bytes	7.6 16 bytes	5.7
get():42	3.7	4.4	3.7	3.7
get():142	3.7	5.7	3.6	3.6
thrower()	730 728 bytes	1500 1456 bytes	1500 1456 bytes	760 728 bytes

Methods as fields

◎ Reflection optimized if Methods are static final fields

```
private static final Method ADD, GET, THROWER;  
  
static {  
    try {  
        ADD = Counter.class.getMethod("add", int.class);  
        ADD.setAccessible(true);  
        GET = Counter.class.getMethod("get");  
        GET.setAccessible(true);  
        THROWER = Counter.class.getMethod("thrower");  
        THROWER.setAccessible(true);  
    } catch (Throwable e) {  
        throw new ExceptionInInitializerError(e);  
    }  
}
```

3. Arrays

3. Arrays

- ◎ **Arrays are special types of classes**

- Dynamic class – created at runtime
- Can contain primitives and classes

- ◎ **Only common superclass of *all* arrays is Object**

- No other relation between `int[]` and `long[]` and `short[]`
 - Primitive arrays are not subclasses of `Object[]`
- However, multi-dimensional primitive arrays are `Object[]`
 - Thus `new int[0][] instanceof Object[] == true`

Showing component types

◎ Arrays contain either primitives or objects

- Find the type with `getComponentType()`

```
public class ArrayElements {  
    public static void main(String... args) {  
        showComponent(char[].class);  
        showComponent(Integer[][][].class);  
        showComponent(int[][][].class);  
    }  
}
```

```
char  
char  
class [[Ljava.lang.Integer;  
java.lang.Integer[][]  
class [[I  
int[][]
```

```
private static void showComponent(Class<?> c) {  
    Class<?> comp = c.getComponentType();  
    System.out.println(comp);  
    System.out.println(comp.getTypeName());  
}
```

```
}
```

Element Types

◎ These are the standard encodings with `getName()`

- e.g. `char[].class.getName()` = `[C`
- e.g. `int[][][].class.getName()` = `[[[I`

Element Type	Encoding
boolean	Z
byte	B
char	C
class or interface with binary name N	LN;
double	D
float	F
int	I
long	L
short	S

Finding array component type

◎ Class has several methods for arrays:

- `Class.isArray()`
 - Returns true if the class is for a primitive or object array
- `Class.getComponentType()`
 - Returns the component types of the array
 - e.g. `int[].class.getComponentType()` -> `int.class`
- `Class.arrayType`
 - The opposite of `getComponentType()`
 - e.g. `int.class.arrayType()` -> `int[].class`

3.1. Accessing elements

3.1. Accessing elements

◎ We can get() and set() array elements with

- java.lang.reflect.Array - for working with arrays reflectively
- java.util.Arrays - facade with convenience methods for arrays

```
import java.lang.reflect.Array;  
import java.util.Arrays;
```

```
public class ModifyArrayContentsTest {  
    public static void main(String... args) {  
        String[] values = {"a", "b", "c"};  
        Array.set(values, 2, "hello");  
        System.out.println(Arrays.toString(values));  
        System.out.println(Array.get(values, 1));  
    }  
}
```


Exercise 3A

◎ Convert an array to a stream

```
public class CollectionConverter {  
    /**  
     * Converts an array to a stream, either with the  
     * elements being the wrapper class of the primitive  
     * array elements, or if it is an Object[] then the  
     * containing objects.  
     *  
     * @throws IllegalArgumentException if parameter is  
     *                                     not an array  
     */  
    public static Stream<?> asStream(Object array) { }  
}
```

3.2. Creating new

3.2. Creating new

◎ We can create a new Array from any class type

```
import java.lang.reflect.Array;

public class ArrayCreateTest {
    public static void main(String... args) {
        int[] m = (int[]) Array.newInstance(int.class, 5);
        m[0] = 35;
        m[1] = 36;
        m[2] = 9;
        m[3] = 6;
        m[4] = 1;
        for (int i : m) {
            System.out.println(i);
        }
    }
}
```


Exercise 3B

◎ Returns a deep clone of the array source

- If source is an array, clone it; otherwise return it
- For each element, call deepClone() recursively

```
public class ArraysWithReflectionMagic {  
    /**  
     * Returns a deep clone of the array source, but  
     * does not clone the actual elements of the array  
     * (unless they are also arrays). If the source  
     * is not an array, simply return it.  
     */  
    public static <A> A deepClone(A source) {  
        return source;  
    }  
}
```

4. java.lang.invoke

4. java.lang.invoke

- ◎ **Reflection suffers from amnesia**

- Checks permission every time you invoke method

- ◎ **Reflection wraps all exceptions**

- Needs to generate an additional stack trace every time

MethodHandles and VarHandles

- ◎ **MethodHandles** are used to call
 - Methods and constructors
- ◎ **VarHandles** are used to read and modify fields
 - Can do so in a thread-safe way with compare-and-swap

4.1. MethodHandles .Lookup

4.1. MethodHandles.Lookup

◎ We say up front who we are

- This means we can have finer-grained access control

```
public class LookupDemo {  
    public static void main(String... args) {  
        MethodHandles.Lookup lookup =  
            MethodHandles.lookup();  
        // Usually we cannot know our class in a static  
        // context, since getClass() is not available  
        Class<?> whoami = lookup.lookupClass();  
        System.out.println("whoami = " + whoami);  
    }  
}
```

```
whoami = class reflection.ch4.LookupDemo
```


MethodHandles.Lookup

◎ Lookup is used to find any class member

- findClass() - nested classes
- findConstructor() - constructors
- find[Static]Getter() - reading fields
- find[Static]Setter() - writing fields, not for final fields
- find[Static]VarHandle() - VarHandle for accessing fields
- findSpecial() - call to super()
- findStatic() - static method call
- findVirtual() - non-static method call

◎ Can also define hidden classes

4.1.1. privateLookupIn()

4.1.1. privateLookupIn()

© We can also find private members in other classes

```
// --add-opens java.base/java.lang=ALL-UNNAMED
```

```
public class Spyglass {  
    public static void main(String... args)  
        throws IllegalAccessException {  
        // normal lookup  
        System.out.println(MethodHandles.lookup());  
        // private lookup inside java.lang.String  
        MethodHandles.Lookup privateLookup =  
            MethodHandles.privateLookupIn(  
                String.class,  
                MethodHandles.lookup());  
        System.out.println(privateLookup);  
    }  
}
```

reflection.ch4.Spyglass
java.lang.String/reflection.ch4.Spyglass

4.2. MethodHandle

4.2. java.lang.invoke.MethodHandle

- ◎ **Similar to java.lang.reflect.Method**
 - Avoids boxing parameters with @PolymorphicSignature
- ◎ **Throws method exceptions directly, not wrapping**
 - All invoke() methods throw Throwable
- ◎ **Does not check security access**
 - That has already been done when we created the Lookup
- ◎ **Matches return type**
 - Additional safety measure in case we call the wrong method

MethodType

© To find a MethodHandle, we need a MethodType

- Contains the return type, followed by the parameter types
 - Constructor return type is void.class

```
public class FooBar {  
    // MethodType.methodType(void.class)  
    public FooBar() {}  
    // MethodType.methodType(void.class, String.class)  
    public FooBar(String id) {}  
    // MethodType.methodType(void.class)  
    public void foo() {}  
    // MethodType.methodType(void.class, String.class,  
    //      int.class)  
    public void bar(String name, int age) {}  
}
```


Finding Constructors

◎ Constructors have only a `MethodType`, no name

```
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodHandle constructor1 = lookup.findConstructor(
    FooBar.class,
    MethodType.methodType(void.class));
MethodHandle constructor2 = lookup.findConstructor(
    FooBar.class,
    MethodType.methodType(void.class,
        String.class));
```

Finding Methods

◎ Methods need a name and a MethodType

```
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodHandle foo = lookup.findVirtual(
    FooBar.class,
    "foo",
    MethodType.methodType(void.class));
MethodHandle bar = lookup.findVirtual(
    FooBar.class,
    "foo",
    MethodType.methodType(void.class,
        String.class, int.class));
```

Invoking MethodHandle

◎ We can call methods with

- `invoke(Object...args)`
 - Return type does not have to be exactly correct
- `invokeExact(Object...args)`
 - Return type has to be exactly the same, not upcast to `Object`
- `invokeWithArguments(Object...args)`
- `invokeWithArguments(List<?> args)`

ConstructDemo

© Constructing an Integer(String) with invoke()

```
public class ConstructDemo {  
    public static void main(String... args)  
        throws Throwable {  
        var lookup = MethodHandles.lookup();  
        MethodHandle constr = lookup.findConstructor(  
            Integer.class,  
            MethodType.methodType(void.class,  
                String.class));  
        Integer value = (Integer) constr.invoke("42");  
        System.out.println("value = " + value);  
    }  
}
```

value = 42

InvokeDemo

◎ Calling toUpperCase() using a MethodHandle

```
public class InvokeDemo {  
    public static void main(String... args)  
        throws Throwable {  
        var lookup = MethodHandles.lookup();  
        MethodHandle toUpper = lookup.findVirtual(  
            String.class, "toUpperCase",  
            MethodType.methodType(String.class));  
        System.out.println(toUpper.invoke("Java123"));  
    }  
}
```

JAVA123

Lookup figures out security context

```
public class PrivateMethodCalls {  
    private void foo() {  
        System.out.println("PrivateMethodCalls.foo()");  
    }  
    public static void main(String... args)  
        throws Throwable {  
        var obj = new PrivateMethodCalls();  
        MethodHandles.lookup().findVirtual(  
            PrivateMethodCalls.class, "foo",  
            MethodType.methodType(void.class))  
            .invoke(obj); // OK  
        PrivateMethodCalls.class.getMethod("foo")  
            .invoke(obj); // NoSuchMethodException  
    }  
}
```

PrivateMethodCalls.foo()
NoSuchMethodException: reflection.ch4.
PrivateMethodCalls.foo()

Passing in an argument list

© java ConstructWithArgumentListDemo "1023"

```
public class ConstructWithArgumentListDemo {  
    public static void main(String... args)  
        throws Throwable {  
        var lookup = MethodHandles.lookup();  
        MethodHandle constr = lookup.findConstructor(  
            Integer.class,  
            MethodType.methodType(void.class,  
                String.class));  
        Integer value = (Integer) constr  
            .invokeWithArguments(args);  
        System.out.println("value = " + value);  
    }  
}
```

value = 1023

Creating a Spreader for argument lists

© java ConstructWithSpreaderDemo "21342"

```
public class ConstructWithSpreaderDemo {  
    public static void main(String... args)  
        throws Throwable {  
        var lookup = MethodHandles.lookup();  
        MethodHandle constr = lookup.findConstructor(  
            Integer.class,  
            MethodType.methodType(void.class,  
                String.class))  
            .asSpreader(String[].class, 1);  
        Integer value = (Integer) constr  
            .invoke(args);  
        System.out.println("value = " + value);  
    }  
}
```

value = 21342

Experiment

© Calling new Integer(String) with parameter list

	new Integer(args[0])	invoke(args[0])	invokeWith Arguments(args)	invoke(args) Spreader
Nanos	120	140	1300	150
Bytes	16	16	224	16

invoke() vs invokeExact()

- ◎ **We can check return type with invokeExact()**
 - More strict
 - Will discover more mistakes in reflection code

InvokeExactDemo

```
public class InvokeExactDemo {  
    public static void main(String... args)  
        throws Throwable {  
        var lookup = MethodHandles.lookup();  
        MethodHandle mh = lookup.findVirtual(  
            String.class, "toUpperCase",  
            MethodType.methodType(String.class));  
        try {  
            System.out.println(mh.invokeExact("Java"));  
        } catch (WrongMethodTypeException e) {  
            System.out.println(e.getMessage());  
        }  
        System.out.println((String)mh.invokeExact("Java"));  
    }  
}  
expected (String)String but found (String)Object  
JAVA
```


4.3. VarHandle

4.3. VarHandle

◎ Same speed as accessing directly

```
public class ConcurrentLinkedQueue<E> {
    static final class Node<E> {
        volatile E item;
        boolean casItem(E cmp, E val) {
            return ITEM.compareAndSet(this, cmp, val);
        }
    }
    static final VarHandle ITEM;
    static {
        try {
            MethodHandles.Lookup l = MethodHandles.lookup();
            ITEM = l.findVarHandle(Node.class, "item", Object.class);
        } catch (ReflectiveOperationException e) {
            throw new Error(e);
        }
    }
}
```

4.4. Converting from Method to MethodHandle

4.4. Converting from Method to MethodHandle

- ◎ **MethodHandles cannot be used to reflect classes**

- We have to know exactly what we are looking for
 - Including return types for methods and types for fields

- ◎ **Use reflection to discover the available methods**

Unreflecting elements

◎ **We can "unreflect" the following elements:**

- `unreflect(Method)`
- `unreflectConstructor(Constructor)`
- `unreflectGetter(Field)` and `unreflectSetter(Field)`
- `unreflectSpecial(Method, Class)`
- `unreflectVarHandle()`

UnreflectingDemo

◎ Here we are "unreflecting" a method from String

```
public class UnreflectingDemo {  
    public static void main(String... args)  
        throws Throwable {  
        var lookup = MethodHandles.lookup();  
        Method method = String.class  
            .getMethod("toUpperCase");  
        MethodHandle mh = lookup.unreflect(method);  
        System.out.println(mh.invoke("unreflective"));  
    }  
}
```

Unreflecting a setter for a private field

```
// --add-opens java.base/java.lang=ALL-UNNAMED
public class IntegerHackUnreflected {
    public static void main(String... args)
        throws Throwable {
        Field value = Integer.class
            .getDeclaredField("value");
        value.setAccessible(true);
        MethodHandle VALUE_SETTER = MethodHandles.lookup()
            .unreflectSetter(value);
        VALUE_SETTER.invoke(42, 43);
        for (int i = 1; i <= 10; i++) {
            System.out.printf("%d x 7 = %d%n",
                i, i * 7);
        }
    }
}
```

```
...
5 x 7 = 35
6 x 7 = 43
7 x 7 = 49
...
```


Writing final fields with MethodHandle

◎ Remember MethodHandles.privateLookupIn(...)

- Added in Java 9 to give access to private fields with VarHandles
- Will only find getters and not setters for final fields
- Similarly, VarHandles cannot write, only read
 - Even if we unreflect
- Unreflected setters can write non-static final fields
 - Backwards compatibility

IMPL_LOOKUP

- ◎ **Superuser power on all elements**

- Can write to final fields with VarHandles, call private methods

- ◎ **Hidden away inside MethodHandles.Lookup class**

4.6. Exercise

4.6. Exercise 4A

- ◎ **Redo the MagicClassInstantiator from exercise 1C**
 - This time using MethodHandles for the constructor and method
 - Use `privateLookupIn()` in case the constructor is private
 - Either use `invokeWithArguments()` or make a Spreader
 - Unreflect the method into a MethodHandle

5. Conclusion

Conclusion

- ◎ **The fastest way to learn a language is to speak it**
 - Learning a programming language is like a natural language
 - Easiest is to immerse yourself in the language
 - Make mistakes, laugh, try again
 - "Και ένα κιλό λαχανικά σας παρακαλώ."

The Java Specialists' Newsletter

- ◎ **Make sure to subscribe**

- www.javaspecialists.eu/archive/subscribe/

- ◎ **Readers in 150+ countries**

- ◎ **Over 21 years of newsletters on advanced Java**

- All previous newsletters available on www.javaspecialists.eu
 - Courses, additional training, etc.