Reflection

Accessing and dynamically using class, field, method, and constructor information at runtime in Java.

Advanced 50 min

Last updated: November 1, 2026
Read online: https://www.learnforgex.com/en/topics/reflection

---

Reflection

Java **Reflection** is an API that lets a running program inspect its own classes, fields, methods, and constructors at runtime — and even invoke or create them dynamically. It gives the program access, while it is running, to type information that would normally only be known at compile time — the Java answer to the idea that "code can examine itself."

What Is Reflection?

Imagine being able to learn the type of an object you're holding, at runtime, without knowing anything about it in advance (just an `Object` reference):

```
Object obj = "hello";
System.out.println(obj.getClass().getName()); // java.lang.String
```

This is actually the simplest reflection example everyone has used dozens of times — `getClass()`. The Reflection API takes this much further: it can list which fields, methods, and constructors a class has, access even `private` members, and even invoke a constructor whose parameters you didn't know about to create a new object — all of this without ever `import`-ing the class at compile time, just by knowing its name (for example, as a string).

Why Does It Exist?

Java is a statically typed language — the compiler knows the type of every variable at compile time and checks accordingly. This provides a strong safety net, but some real-world problems require exactly the opposite: a framework needs to discover, at runtime, a class it has never seen (a DTO, a controller, an entity written by the user) and populate its fields, invoke its methods, and change behavior based on its annotations. Spring injecting a dependency with `@Autowired`, Jackson converting a JSON into your class, JUnit finding and running methods marked with `@Test` — none of these are possible by that library "knowing"

your classes at compile time. Reflection fills exactly this gap: a library can discover types by name and annotation at runtime and work with them through a generic mechanism. We'll cover each of these frameworks one by one in the upcoming "Real-World Use Cases" section.

History

Reflection was added to Java very early on, in JDK 1.1 (1997), under the `java.lang.reflect` package — one of the language's oldest APIs. While its core infrastructure has stayed largely the same since then, important additions were made around it: Java 5 brought generic type information into reflection; Java 7 introduced the `MethodHandle` mechanism with the `java.lang.invoke` package — a modern alternative that is much more performant than classic reflection (we'll go into detail in "Performance Considerations"); the module system in Java 9 (JPMS) restricted reflection's ability to access *everything* — unless a module explicitly `opens` its internal packages, it became inaccessible from the outside even with `setAccessible(true)` (we'll cover this in "Security Considerations"). Java 21, which we use in this project, holds all of these layers together (classic reflection + `MethodHandle` + module restrictions).

Getting a Class Object

Everything in reflection starts from a `java.lang.Class` object — this object represents the runtime "identity card" of a type: its name, fields, methods, and constructors are all accessed from here. There are three ways to obtain a `Class` object:

```
class Book {
    private String title;
    private String author;
    private int pages;

    Book() {
        this("Untitled", "Unknown", 0);
    }

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    @Override
    public String toString() {
        return "Book{title='" + title + "', author='" + author + "', pages=" + pages + "}";
    }
}

class ThreeWaysToGetClass {
    public static void main(String[] args) throws ClassNotFoundException {
        Book book = new Book("Effective Java", "Joshua Bloch", 412);

        Class<?> fromInstance = book.getClass();
        Class<?> fromLiteral = Book.class;
        Class<?> fromName = Class.forName("Book");

        System.out.println(fromInstance.getName());       // Book
        System.out.println(fromInstance == fromLiteral);  // true
        System.out.println(fromInstance == fromName);     // true
    }
}
```

Expected Output

```
Book
true
true
```

- `obj.getClass()` — the most natural way if you already have an instance.
- `TypeName.class` — a compile-time constant reference that doesn't need an instance; if you already know the type name directly (which you usually do), this is the cleanest way.
- `Class.forName("TypeName")` — for cases where you only know the type name as a **string** (for example, a class name read from a config file); this is exactly the method frameworks use the most.

All three return the same `Class` object — this is because the JVM keeps only **one** `Class` instance per class, per classloader: that's why the comparison `fromInstance == fromLiteral` returns `true`, without even needing `equals()`.

Primitive types also have their own `Class` objects — like `int.class`, `boolean.class`, `void.class`. These are **different** from the `Class` of the corresponding boxed type: `int.class != Integer.class`. Forgetting this distinction while matching parameter types in the Reflection API is a common cause of `NoSuchMethodException`.

`Class.forName(String)` doesn't just "find" the class — by default it also **loads and initializes** it (static blocks and static field initializers run). If you want to inspect a class without initializing it (for example, to avoid the side effects of a static block), use the three-argument overload: `Class.forName(name, false, classLoader)`.

Inspecting Class Information

Once you have a `Class` object, you can access a rich set of information about that type:

```
import java.lang.reflect.Modifier;

class Book {
    private String title;
    private String author;
    private int pages;

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}

class ClassInfoExample {
    public static void main(String[] args) {
        Class<Book> type = Book.class;

        System.out.println(type.getName());        // Book
        System.out.println(type.getSimpleName());   // Book
        System.out.println(type.getPackageName());  // (empty string — default package)
        System.out.println(type.isInterface());     // false
        System.out.println(type.isRecord());        // false
        System.out.println(Modifier.isPublic(type.getModifiers())); // false — package-
private
    }
}
```

Expected Output

```
Book
Book

false
false
false
```

The difference between `getName()` and `getSimpleName()` becomes apparent with nested classes and arrays — `getName()` always returns the JVM's internal representation (with `$` separators for nested classes, `[` prefixes for arrays), while `getSimpleName()` returns the readable name as written in source code. Note that `getPackageName()` returns an empty string for the default package (as in the examples in this project).

You can also query the class hierarchy and implemented interfaces through `Class` — with the well-known `ArrayList`:

```
Class<?> listType = java.util.ArrayList.class;
System.out.println(listType.getSuperclass());
// class java.util.AbstractList
System.out.println(java.util.Arrays.toString(listType.getInterfaces()));
// [interface java.util.List, interface java.util.RandomAccess, interface java.lang.
Cloneable, interface java.io.Serializable]
```

> Static helper methods like `Modifier.isPublic(type.getModifiers())`, `Modifier.isFinal(...)`, and `Modifier.isAbstract(...)` turn the raw `int` bitmask returned by `getModifiers()` into readable boolean queries — you never need to do bit manipulation by hand.

> Don't confuse `isInstance(Object)` and `isAssignableFrom(Class)` with the `instanceof` operator: `type.isInstance(obj)` means the same thing as `obj instanceof Type` (it asks whether an **object** matches a type); `typeA.isAssignableFrom(typeB)` instead asks whether two **types** are assignable to each other (can an instance of `typeB` be assigned to `typeA`?). Reversing the direction is a very common mistake that silently produces the wrong result.

Reading Fields

Listing a class's fields uses one of two methods, and the difference between them is often confused:

```
import java.lang.reflect.Field;

class Book {
    public static final String CATEGORY = "Literature";
    private String title;
    private String author;
    private int pages;

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}

class FieldsExample {
    public static void main(String[] args) throws Exception {
        Class<Book> type = Book.class;

        System.out.println("getFields():");
        for (Field f : type.getFields()) {
            System.out.println("  " + f.getName());
        }
        // getFields(): CATEGORY

        System.out.println("getDeclaredFields():");
        for (Field f : type.getDeclaredFields()) {
            System.out.println("  " + f.getName() + " : " + f.getType().getSimpleName());
        }
        // getDeclaredFields(): CATEGORY : String, title : String, author : String, pages :
int

        Field categoryField = type.getField("CATEGORY");
        System.out.println(categoryField.get(null)); // Literature — static field, no
instance needed
    }
}
```

`getFields()` returns only **public** fields — but this includes those inherited from superclasses /interfaces. `getDeclaredFields()` is the opposite: it returns only the fields **defined in that class itself** (regardless of access modifier — including `private`), but does **not** include those inherited from a superclass. Remembering that these two methods behave as exact opposites along the "public or not" and "inherited or not" axes is the most practical way to decide which one to pick.

`Field.get(Object)` reads a field's value; if the field is `static`, `null` can be passed as the argument (as we did above for `CATEGORY`) — for a non-`static` field you must supply an actual instance.

> In the example above we only read a **public** field (`CATEGORY`) — if you tried to read a `private` field with `get()` , you would get an `IllegalAccessException` . The fix for this (we'll cover it in "Accessing Private Fields and Methods") is calling `setAccessible(true)` , but we're deferring that for now — it's important to first separate out "just discovering" from actually accessing.

Reading Methods

The same paired pattern applies to methods — `getMethods()` / `getDeclaredMethods()` :

```
import java.lang.reflect.Method;

class Book {
    private String title;

    Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    private boolean isLong() {
        return title.length() > 50;
    }
}

class MethodsExample {
    public static void main(String[] args) {
        Class<Book> type = Book.class;

        System.out.println("getMethods() count: " + type.getMethods().length);
        // includes getTitle() plus everything inherited from Object (toString, equals,
hashCode, ...)

        System.out.println("getDeclaredMethods():");
        for (Method m : type.getDeclaredMethods()) {
            System.out.println("  " + m.getName() + " -> " + m.getReturnType().
getSimpleName());
        }
        // getTitle -> String, isLong -> boolean (declared here, private included,
inherited excluded)
    }
}
```

`getMethods()` returns a class's **public** methods — including those you define yourself **and** methods inherited from `Object` such as `toString()` , `equals()` , and `hashCode()` (which is why the list turns out bigger than expected even for a simple class). `getDeclaredMethods()` returns only the methods defined **in the class's own body** — the access modifier doesn't matter (including `private`), but inherited ones are excluded.

Through each `Method` object you can fully discover its signature with methods like `getReturnType()`, `getParameterTypes()`, and `getParameterCount()` — we'll use this information in the next two sections, first to create objects (for constructors), then to actually **invoke** these methods.

> If you want to get an overloaded method by its exact signature rather than by name, use `getMethod(String, Class<?>...)` — for example, `getMethod("process", String.class)` versus `getMethod("process", int.class)` distinguishes between two different methods with the same name. There is no method that searches by name alone without specifying parameter types, because overload resolution in Java is done by type.

Reading Constructors

Constructors are discovered much like methods — `getConstructors()` (public ones) and `getDeclaredConstructors()` (all of them, regardless of access modifier):

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

class Book {
    private String title;
    private String author;
    private int pages;

    Book() {
        this("Untitled", "Unknown", 0);
    }

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}

class ConstructorsExample {
    public static void main(String[] args) {
        Class<Book> type = Book.class;

        for (Constructor<?> c : type.getDeclaredConstructors()) {
            System.out.println(c.getParameterCount() + " parameter(s): "
                    + Arrays.toString(c.getParameterTypes()));
        }
        // 0 parameter(s): []
        // 3 parameter(s): [class java.lang.String, class java.lang.String, int]
    }
}
```

Each `Constructor` object reveals its own signature via `getParameterCount()` and `getParameterTypes()` — if a class has multiple (overloaded) constructors, you can distinguish and select the right one based on parameter types. In the next section we'll do exactly that: use the constructor we selected to produce an actual object.

Creating Objects Dynamically

Once you've found a constructor, you can actually invoke it with `newInstance(Object...)` to produce a new object:

```
import java.lang.reflect.Constructor;

class Book {
    private String title;

    Book(String title) {
        this.title = title;
    }

    @Override
    public String toString() {
        return "Book{title='" + title + "'}";
    }
}

class DynamicObjectCreation {
    public static void main(String[] args) throws Exception {
        Class<Book> type = Book.class;

        Constructor<Book> constructor = type.getDeclaredConstructor(String.class);
        Book book = constructor.newInstance("Domain-Driven Design");

        System.out.println(book); // Book{title='Domain-Driven Design'}
    }
}
```

Expected Output

```
Book{title='Domain-Driven Design'}
```

`getDeclaredConstructor(Class<?>...)` finds the correct constructor based on parameter types; `newInstance(...)` then invokes that constructor with the arguments you provide and returns a new object — just as if you had written `new Book(...)`, but without ever knowing the type name (`Book`) at compile time.

> There's an old API, now **deprecated**, that should not be confused with `Constructor.newInstance(...)`: the parameterless `Class.newInstance()`. It has been deprecated since Java 9, because it has two significant problems: (1) it throws the constructor's **checked** exceptions directly without wrapping them — effectively bypassing the compiler's checked-exception checking; (2) it doesn't enforce access control on `private` / `protected` constructors as consistently as `Constructor.newInstance()` does. Always use `getDeclaredConstructor(...)` + `Constructor.newInstance(...)` in new code.

Invoking Methods Dynamically

`Method.invoke(Object, Object...)` runs a method you've found, on the object you provide, with the arguments you provide:

```
import java.lang.reflect.Method;

class Book {
    private String title;

    Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public static String describe() {
        return "A Book represents a published work.";
    }
}

class DynamicMethodInvocation {
    public static void main(String[] args) throws Exception {
        Book book = new Book("Refactoring");
        Class<Book> type = Book.class;

        Method getTitle = type.getMethod("getTitle");
        System.out.println(getTitle.invoke(book)); // Refactoring

        Method describe = type.getMethod("describe");
        System.out.println(describe.invoke(null)); // static method -> no target instance
needed
    }
}
```

Expected Output

```
Refactoring
A Book represents a published work.
```

In the first call, we ran the instance method `getTitle` on the `book` object — that's the first parameter (the target object of `invoke`). In the second call, since `describe()` is a **static** method, we passed `null` as the target object — invoking static methods doesn't require an instance, just like reading a static field (the same logic as the `CATEGORY` example we saw in "Reading Fields").

> If the method you invoke throws an exception, `invoke()` doesn't throw it directly — it wraps the original exception in an **`InvocationTargetException`** and throws that instead; to see the real error you need to unwrap the wrapped exception with `getCause()`. Forgetting this wrapping and trying to catch the exception type you expected directly is a common reflection mistake:

```
try {
    method.invoke(target, args);
} catch (java.lang.reflect.InvocationTargetException e) {
    Throwable realCause = e.getCause();
    // the real error is here, not in e itself
}
```

Accessing Private Fields and Methods

So far we've only worked with **public** members. But perhaps reflection's most well-known (and most misused) feature is that it can also access `private` fields and methods — via `AccessibleObject.setAccessible(true)`:

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

class Book {
    private String title = "Untitled";

    private boolean isLong() {
        return title.length() > 50;
    }
}

class PrivateAccessExample {
    public static void main(String[] args) throws Exception {
        Book book = new Book();
        Class<Book> type = Book.class;

        Field titleField = type.getDeclaredField("title");
        titleField.setAccessible(true);
        titleField.set(book, "Effective Java");
        System.out.println(titleField.get(book)); // Effective Java

        Method isLong = type.getDeclaredMethod("isLong");
        isLong.setAccessible(true);
        System.out.println(isLong.invoke(book)); // false
    }
}
```

Expected Output

```
Effective Java
false
```

`setAccessible(true)` disables the access control (private/protected/package-private) that Java normally enforces at compile time, for reflection calls — this applies to both `Field.get()` / `Field.set()` and `Method.invoke()`. Without this line, the example above would throw `IllegalAccessException`.

> `setAccessible(true)` no longer **always** works with the module system (JPMS) introduced in Java 9 — as we mentioned in "History," if the target class is inside a named module and that module hasn't explicitly opened the relevant package with `opens`, calling `setAccessible(true)` throws `InaccessibleObjectException`. We'll go deep into this restriction and how to manage it in "Security Considerations" — this project runs without modules (classpath-based), so it works fine here, but it's something you need to watch out for in a real modular application.

> Call `setAccessible(true)` only on the single `Field` / `Method` object you actually need, in as narrow a scope as possible — making all of a class's private members "accessible" in bulk and using them broadly weakens both readability and testability. Accessing private members via reflection is generally a framework/test-infrastructure need, not something you'd choose in business logic.

Working with Annotations

For you to be able to read an annotation via reflection, that annotation must **survive until runtime** — this is determined by `@Retention` in the annotation's definition:

```
@Retention(RetentionPolicy.SOURCE)   // source code only, discarded after compilation — e.
g. @Override
@Retention(RetentionPolicy.CLASS)    // stays in the .class file but isn't accessible while
the JVM runs (default)
@Retention(RetentionPolicy.RUNTIME)  // readable via reflection at runtime — this is what
we need
```

`@Override` is a well-known example: it's marked with `RetentionPolicy.SOURCE`, existing only for the compiler's check of "does this method actually override a superclass method?" — it leaves no trace in the compiled `.class` file, and you can never see it via reflection.

You can read an annotation marked with `RetentionPolicy.RUNTIME` using the same methods on `Class`, `Field`, `Method`, and `Constructor` (all of which implement the `AnnotatedElement` interface):

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Auditable {
    String value() default "";
}

class AccountService {
    @Auditable("balance-check")
    void checkBalance() {
        // ...
    }

    void internalHelper() {
        // not annotated
    }
}

class AnnotationExample {
    public static void main(String[] args) throws Exception {
        Method checkBalance = AccountService.class.getDeclaredMethod("checkBalance");
        Method internalHelper = AccountService.class.getDeclaredMethod("internalHelper");

        System.out.println(checkBalance.isAnnotationPresent(Auditable.class));   // true
        System.out.println(internalHelper.isAnnotationPresent(Auditable.class)); // false

        Auditable auditable = checkBalance.getAnnotation(Auditable.class);
        System.out.println(auditable.value()); // balance-check
    }
}
```

Expected Output

```
true
false
balance-check
```

The difference between `getAnnotations()` and `getDeclaredAnnotations()` isn't like the `getFields()` / `getDeclaredFields()` distinction for fields/methods — what matters here is whether the annotation itself is marked with `@Inherited`. `@Inherited` is meaningful **only for class-level** annotations and is off by default: if an annotation isn't `@Inherited`, a subclass doesn't inherit it from its superclass, and `getAnnotations()` won't show it either.

> Forgetting `@Retention` when defining an annotation (or leaving it at the default `CLASS`) is one of the most common reflection traps — the code compiles, the annotation "looks like it's there" in the source, but `isAnnotationPresent()` always returns `false` and you spend hours debugging "why isn't this showing up?" Always double-check that every annotation you plan to read via reflection has `@Retention(RetentionPolicy.RUNTIME)`.

Real-World Use Cases

Every piece we've seen so far — `Class` discovery, dynamic object creation, dynamic method invocation, annotation reading — forms the core operating mechanism of real frameworks:

- **Spring** scans the classpath and finds classes marked with `@Component` / `@Service` / `@Controller` (annotation reading), inspects each one's constructor to resolve dependencies based on parameter types (`@Autowired` — a scaled-up version of the `Constructor.newInstance()` logic we saw in "Creating Objects Dynamically"), and assigns values to fields with `@Value` / `@ConfigurationProperties` (`Field.set()`).
- **Hibernate/JPA** inspects an entity's fields and maps them to database columns, populating objects from query results by writing directly to `private` fields (usually via a combination of a no-arg constructor + `setAccessible(true)` + `Field.set()`), and can even generate a proxy class at runtime that extends your entity class for lazy loading.
- **Jackson** finds fields/setters (`Field` / `Method`) via reflection when converting a JSON into your class — as we saw in the "Real-World Examples" section (in the Record topic), for a record it directly recognizes and uses the canonical constructor.
- **JUnit** finds methods marked with `@Test` via `getDeclaredMethods()` plus an annotation check, and runs each one individually with `Method.invoke()` . We can write this mechanism ourselves, in a much simplified form:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyTest {
}

class Calculator {
    @MyTest
    void additionWorks() {
        if (2 + 2 != 4) {
            throw new AssertionError("Addition is broken");
        }
        System.out.println("additionWorks: PASSED");
    }

    @MyTest
    void subtractionWorks() {
        if (5 - 3 != 2) {
            throw new AssertionError("Subtraction is broken");
        }
        System.out.println("subtractionWorks: PASSED");
    }

    void notATest() {
        System.out.println("This should never run");
    }
}

// A drastically simplified sketch of what JUnit does internally: scan for
// annotated methods, then invoke each one reflectively.
class MiniTestRunner {
    public static void main(String[] args) throws Exception {
        Calculator target = new Calculator();

        for (Method method : Calculator.class.getDeclaredMethods()) {
            if (method.isAnnotationPresent(MyTest.class)) {
                method.invoke(target);
            }
        }
    }
}
```

> What all four of these frameworks have in common is this: they discover and use **your** classes at runtime, without ever `import`-ing your source code — you've now seen concretely exactly how the problem we defined in "Why Does It Exist?" is solved in production-quality libraries.

Performance Considerations

Reflection calls are always slower than direct calls, for three main reasons: (1) access control is checked on every call (`setAccessible(true)` skips this, but not entirely); (2) primitive parameters have to be boxed into an `Object[]` and unboxed back; (3) the JIT compiler can't optimize a single generic call site like `Method.invoke()` as aggressively as a direct method call.

In practice this translates into two recommendations:

Cache your lookups. Lookup operations like `getMethod()` / `getDeclaredField()` are more expensive than `invoke()` / `get()` themselves — finding a `Method` / `Field` object once and storing it in a variable/`static final` field is much better than looking it up again on every call.

Consider `MethodHandle` for a hot path. The `java.lang.invoke` package introduced in Java 7 offers a more modern alternative to classic reflection — a `MethodHandle` resolved once via `MethodHandles.Lookup` can be optimized far better by the JIT (it can reach performance approaching a direct call on repeated invocations):

```
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.lang.reflect.Method;

class Greeter {
    public String greet(String name) {
        return "Hello, " + name + "!";
    }
}

class MethodHandleExample {
    public static void main(String[] args) throws Throwable {
        Greeter greeter = new Greeter();

        // Classic reflection
        Method method = Greeter.class.getMethod("greet", String.class);
        System.out.println(method.invoke(greeter, "Ada")); // Hello, Ada!

        // MethodHandle — resolved once via a Lookup, then invoked directly.
        // The JIT can optimize a MethodHandle call site much more aggressively
        // than a classic Method.invoke() call.
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle handle = lookup.findVirtual(Greeter.class, "greet",
                MethodType.methodType(String.class, String.class));
        System.out.println((String) handle.invoke(greeter, "Grace")); // Hello, Grace!
    }
}
```

Expected Output

```
Hello, Ada!
Hello, Grace!
```

Java 9 also added `VarHandle` — `MethodHandle`'s sibling focused on field access (get/set, and even atomic compare-and-swap operations); it's the official, supported replacement for the low-level field manipulation that used to be done with `sun.misc.Unsafe`.

> Don't try to "benchmark" reflection versus direct calls with a simple loop and `System.currentTimeMillis()` — JIT warmup, dead-code elimination, and similar optimizations easily make a naive few-line measurement meaningless. Use JMH (Java Microbenchmark Harness) for a real performance comparison; for this lesson it's enough to understand **that** the performance difference exists and **why** — never trust a naive loop for exact numbers.

> In practice, the vast majority of application code never has to deal with these optimizations — frameworks like Spring and Hibernate already do this caching and `MethodHandle` usage internally. The situation where you actually need to know this is when you're writing your own reflection-based tool (a mini framework, a test runner, a serializer).

Security Considerations

The `InaccessibleObjectException` we mentioned in "Accessing Private Fields and Methods" is actually a deliberate security/encapsulation decision of the module system (JPMS) in Java 9. If a module wants to open its own package to reflection, it must state this explicitly in `module-info.java`:

```
module com.example.app {
    opens com.example.app.internal to some.reflecting.module;
}
```

In cases where you can't modify the module (for example, if it's a third-party library), you can achieve the same effect with a JVM startup parameter:

```
java --add-opens com.example.app/com.example.app.internal=ALL-UNNAMED -jar app.jar
```

In older Java versions there was another way to restrict reflection access: `SecurityManager` (via `ReflectPermission` checks) — but this mechanism has been deprecated for removal starting with Java 17, and you shouldn't rely on it in new designs; JPMS's `opens` restriction is now the recommended approach.

Reflection itself can also constitute an attack surface: most known security vulnerabilities related to Java deserialization in particular (recall the deserialization mechanism we touched on in "Serialization and Reflection" in the Record topic) rely on an attacker-controlled byte stream chaining arbitrary method calls via reflection (a "gadget chain").

> Avoid passing a **string that comes from the user** directly into `Class.forName(String)` — this could allow an attacker to load any class on the classpath (even triggering static initializers with unwanted side effects, in some cases). If a class name comes from outside, always validate it against a known, permitted list (allow-list).

Best Practices

- Treat reflection as a **last resort** — it's almost never needed in ordinary business logic; its real home is framework/library/test-infrastructure code.
- Cache `Method` / `Field` / `Constructor` lookups (detailed in "Performance Considerations").
- Consider `MethodHandle` instead of classic reflection for a frequently called path.
- Call `setAccessible(true)` only on the single member you actually need, in a narrow scope (mentioned in "Accessing Private Fields and Methods").
- Never feed externally-sourced class/method names into reflection without validation ("Security Considerations" section).
- Make reflection usage **explicit** in the code (comments, naming) — an IDE's "rename" refactor will **never** update a string literal like `getDeclaredField("title")`; when you rename a field, compilation gives no error, but the reflection call silently throws `NoSuchFieldException` at runtime.

> If you find yourself needing to constantly access a class's `private` members via reflection, that's usually a sign of a design smell — that class's public API may not actually be meeting what's needed. Use reflection when you genuinely need a general-purpose mechanism (framework, test, serializer), not as a tool for "working around" a class.

Common Mistakes

1. Forgetting `IllegalAccessException` and trying to access a `private` member directly. Don't forget you need to call `setAccessible(true)` first (see "Accessing Private Fields and Methods").

2. Forgetting `InvocationTargetException` and trying to catch the actual exception thrown by the invoked method. `invoke()` always wraps the real error; you need to unwrap it with `getCause()` (see "Invoking Methods Dynamically").

3. Continuing to use the deprecated `Class.newInstance()`. Use `getDeclaredConstructor(...)` + `Constructor.newInstance(...)` instead (see "Creating Objects Dynamically").

4. Forgetting that generic type information is erased at runtime (type erasure). A `List<String>` only appears as `List` at runtime — reflection can't see the generic parameter type (`String`) in most cases. Methods like `getGenericType()` provide some limited information (for example, via the generic signature in a field's declaration), but there's no guaranteed way to learn the generic type of the actual runtime object.

5. Forgetting that reflection has a fragility that isn't caught at compile time after refactoring. Renaming a field or method with the IDE (rename) does **not** update a string literal like `getDeclaredField("oldName")` — the code compiles fine, but you get a `NoSuchFieldException` / `NoSuchMethodException` at runtime.

6. Repeating a `Method` / `Field` lookup every time in a hot loop. Instead of calling `getMethod()` again before every call, find it once and cache it (see "Performance Considerations").

Summary, Cheat Sheet, and Glossary

Reflection is the API Java has had since JDK 1.1 that lets a program inspect its own types at runtime and use them dynamically. Key points:

- Everything starts from a `Class` object — obtained via `getClass()`, `.class`, or `Class.forName()`, with exactly one instance guaranteed per JVM
- The `getX()` (public + inherited) / `getDeclaredX()` (only that class's own, regardless of access modifier) pair — the same pattern applies to fields, methods, and constructors
- To create an object, use `Constructor.newInstance(...)` (not the deprecated `Class.newInstance()`); to invoke a method, use `Method.invoke(target, arguments...)`
- `private` members are accessed via `setAccessible(true)` — but this isn't always guaranteed under the Java 9+ module system (`InaccessibleObjectException`)
- `@Retention(RetentionPolicy.RUNTIME)` is required to read an annotation via reflection
- Spring, Hibernate, Jackson, JUnit — all use the same basic mechanism (discover, create, invoke)
- Performance: cache lookups, consider `MethodHandle` for frequently called paths, never trust a naive benchmark
- Security: don't feed unvalidated external class names into `Class.forName()`, respect JPMS's `opens` restriction

Quick reference:

```
// Getting a Class object
Class<?> type = obj.getClass();
Class<?> type2 = MyClass.class;
Class<?> type3 = Class.forName("com.example.MyClass");

// Reading/writing a field
Field field = type.getDeclaredField("fieldName");
field.setAccessible(true);
Object value = field.get(obj);
field.set(obj, newValue);

// Invoking a method
Method method = type.getMethod("methodName", ParamType.class);
Object result = method.invoke(obj, arg);

// Creating an object
Constructor<?> constructor = type.getDeclaredConstructor(ParamType.class);
Object instance = constructor.newInstance(arg);

// Reading an annotation
if (method.isAnnotationPresent(MyAnnotation.class)) {
    MyAnnotation ann = method.getAnnotation(MyAnnotation.class);
}
```

Glossary

Reflection — The API that lets a program inspect its own types at runtime and use them dynamically (the `java.lang.reflect` package).

`Class` **object** — The runtime representation of a type; there is exactly one instance per type per JVM.

`AccessibleObject` — The common superclass of `Field`, `Method`, and `Constructor`; they inherit the `setAccessible(boolean)` method from it.

`setAccessible(true)` — The call that disables compile-time access control (private/protected/package-private) for reflection calls.

`InvocationTargetException` — The wrapper exception that wraps the real exception thrown by the invoked method during `Method.invoke()`; the real error is reached via `getCause()`.

`InaccessibleObjectException` — The exception thrown under the Java 9+ module system when trying to access a package that hasn't been `opens`-ed, via `setAccessible(true)`.

`MethodHandle` — A more performant, JIT-friendly dynamic invocation mechanism introduced in Java 7, compared to classic reflection (the `java.lang.invoke` package).

`VarHandle` — Introduced in Java 9, part of the `MethodHandle` family for field access and atomic operations; replaces old uses of `sun.misc.Unsafe`.

`AnnotatedElement` — The common interface implemented by `Class`, `Field`, `Method`, and `Constructor` that provides annotation-reading methods (`getAnnotation()`, `isAnnotationPresent()`, etc.).

`RetentionPolicy.RUNTIME` — The retention policy required for an annotation to be visible via reflection at runtime.

JPMS (Java Platform Module System) — The module system introduced in Java 9; unless modules explicitly open their packages to reflection with the `opens` directive, they remain inaccessible from the outside even with `setAccessible(true)`.

Appendix: Mini Project — A Simple Dependency Injection Container

Let's combine what we've learned so far ("Reading Constructors," "Creating Objects Dynamically") to write, in a few lines, the essence of what Spring does with `@Autowired` constructor injection. The idea is simple: when you want to "resolve" a type, the container first looks at what parameters its constructor needs, resolves each parameter **recursively** on its own, then passes them all to the constructor to create the object:

```
import java.lang.reflect.Constructor;
import java.util.HashMap;
import java.util.Map;

class Engine {
    String start() {
        return "Engine started";
    }
}

class Car {
    private final Engine engine;

    Car(Engine engine) {
        this.engine = engine;
    }

    String drive() {
        return engine.start() + " -> Car is driving";
    }
}

// A minimal constructor-injection container: given a type, it recursively
// resolves each constructor parameter, instantiates it, and caches the
// result as a singleton — the same core idea Spring's ApplicationContext
// uses, stripped down to its essence.
class SimpleContainer {
    private final Map<Class<?>, Object> singletons = new HashMap<>();

    @SuppressWarnings("unchecked")
    <T> T resolve(Class<T> type) throws ReflectiveOperationException {
        if (singletons.containsKey(type)) {
            return (T) singletons.get(type);
        }

        Constructor<?> constructor = type.getDeclaredConstructors()[0];
        Class<?>[] paramTypes = constructor.getParameterTypes();
        Object[] args = new Object[paramTypes.length];

        for (int i = 0; i < paramTypes.length; i++) {
            args[i] = resolve(paramTypes[i]);
        }

        Object instance = constructor.newInstance(args);
        singletons.put(type, instance);
        return (T) instance;
    }
}
```

```
class SimpleContainerDemo {
    public static void main(String[] args) throws ReflectiveOperationException {
        SimpleContainer container = new SimpleContainer();

        Car car = container.resolve(Car.class);
        System.out.println(car.drive()); // Engine started -> Car is driving
    }
}
```

When we want to resolve `Car.class`, the container first sees that `Car`'s single constructor requires an `Engine` parameter, creates the `Engine` (directly, since it has no dependencies), then passes it to `Car`'s constructor to build `Car` — without you ever writing `new Engine()`. A real DI container (including Spring) builds on top of this; it adds layers like scope management (singleton/prototype), circular-dependency detection, and interface-to-implementation mapping — but the core mechanism is exactly the recursive constructor resolution you see here.

> Notice that the `resolve()` method declares `throws ReflectiveOperationException` — this is the **common superclass** (since Java 7) of reflection-specific checked exceptions like `InstantiationException`, `IllegalAccessException`, `InvocationTargetException`, and `NoSuchMethodException`. Managing them through a single common type instead of catching each one separately significantly simplifies this kind of "there could be multiple reflection exceptions" code.

> This simple container doesn't detect circular dependencies (`A` needs `B`, and `B` needs `A`) — in such a case, `resolve()` enters an infinite loop and crashes with a `StackOverflowError`. Real DI containers track the set of "types currently being built" during resolution and throw a meaningful error if the same type appears a second time.

Appendix: Mini Project — Object Inspector

Our final mini project is a general-purpose tool that takes **any** object and dumps all of its fields (name, type, value) and all of its methods (signature) — combining almost everything we've learned in this lesson (Class discovery, field/method listing, `setAccessible`, `Modifier`) in one place:

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

// A generic "dump everything you can see" utility: given any object, prints
// all of its declared fields (with their current values) and all of its
// declared methods (with their signatures) — regardless of access modifier.
class ObjectInspector {
    static void inspect(Object obj) throws IllegalAccessException {
        Class<?> type = obj.getClass();
        System.out.println("Class: " + type.getName());

        System.out.println("Fields:");
        for (Field field : type.getDeclaredFields()) {
            field.setAccessible(true);
            System.out.println("  " + Modifier.toString(field.getModifiers())
                    + " " + field.getType().getSimpleName()
                    + " " + field.getName()
                    + " = " + field.get(obj));
        }

        System.out.println("Methods:");
        for (Method method : type.getDeclaredMethods()) {
            System.out.println("  " + Modifier.toString(method.getModifiers())
                    + " " + method.getReturnType().getSimpleName()
                    + " " + method.getName() + "(...)");
        }
    }
}
```

Let's inspect the `Book` class we've been using since the first section of this lesson once more, but this time entirely from the outside (with a tool that knows nothing about `Book`'s internal structure):

```
class Book {
    private String title;
    private String author;
    private int pages;

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    String getTitle() {
        return title;
    }
}

class ObjectInspectorDemo {
    public static void main(String[] args) throws IllegalAccessException {
        Book book = new Book("Clean Architecture", "Robert C. Martin", 432);
        ObjectInspector.inspect(book);
    }
}
```

The `inspect()` method can list `Book`'s `private` fields (`title`, `author`, `pages`) and its `getTitle()` method, and read their values, without a single `import` from `Book`'s source code — a working, hands-on example of the "code can examine itself" idea we've been discussing since the start of this lesson.

> `ObjectInspector` is a heavily simplified version of what an IDE's "Evaluate Expression" / variable watch panels and debuggers essentially do — a debugger uses exactly this mechanism (reflection + `setAccessible`) to show you the fields of every object it pauses on.