

Reflection

YEGOR BUGAYENKO

Lecture #7 out of 8

80 minutes

The slidedeck was presented by the author in this [YouTube Video](#)

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as web sites. Copyright belongs to their respected authors.

What Is Reflection?

Type Casting and Subsumption

Factory Method

Classpath Scanning

Annotations

Discrimination by Type

Chapter #1:

What Is Reflection?

Simple Example of Reflection in Java

```
1 void print(Object x) {  
2     if (x instanceof File) {  
3         System.out.println("Yes!");  
4     }  
5 }
```

“Reflective computational systems allow computations to observe and modify properties of their own behavior. It would be desirable for computations to avail themselves of these reflective capabilities, examining themselves in order to make use of meta-level information in decisions about what to do next.”

Source: Jonathan M. Sobel and Daniel P. Friedman.
An Introduction to Reflection-Oriented
Programming, 1996



YUE LI

“As reflection is increasingly used in Java programs, the cost of imprecise reflection handling has increased dramatically... Almost all the analyses reported are unsound in the presence of reflection since it is either ignored or handled partially.”

— Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-Inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, pages 27–53. Springer, 2014.
[doi:10.1007/978-3-662-44202-9_2](https://doi.org/10.1007/978-3-662-44202-9_2)

Chapter #2:

Type Casting and Subsumption

Iterable → Collection

Downcasting (**wrong!**):

```
1 int sizeOf(Iterable items) {  
2     int size = 0;  
3     if (items instanceof Collection) {  
4         size = ((Collection) items).size();  
5     } else {  
6         for (Object item : items) {  
7             ++size;  
8         }  
9     }  
10    return size;  
11 }
```

Method overloading (**right!**):

```
1 int sizeOf(Iterable items) {  
2     int size = 0;  
3     for (Object item : items) {  
4         ++size;  
5     }  
6     return size;  
7 }  
8  
9 int sizeOf(Collection items) {  
10    return items.size();  
11 }
```

Implicit Coupling

```
1 int sizeOf(Iterable items) {  
2     int size = 0;  
3     if (items instanceof Collection) {  
4         size = ((Collection) items).size();  
5     } else {  
6         for (Object item : items) {  
7             ++size;  
8         }  
9     }  
10    return size;  
11 }
```

```
1 Iterable<Book> books1 =  
2     new PgBooks1("localhost:5432");  
3 int s1 = sizeOf(books1);  
4  
5 Collection<Book> books2 =  
6     new PgBooks2("localhost:5432");  
7 int s2 = sizeOf(books2);
```

It may be hard to understand why `|s2|` is evaluated much faster than `|s1|`, while the signature of `|sizeOf()|` is the same [Bugayenko, 2015].

Pattern Matching in Java 16

Java 11 (**wrong!**):

```
1 int sizeOf(Iterable items) {  
2     int size = 0;  
3     if (items instanceof Collection) {  
4         size = ((Collection) items).size();  
5     } else {  
6         for (Object item : items) {  
7             ++size;  
8         }  
9     }  
10    return size;  
11 }
```

Java 16 (**even worse!**):

```
1 int sizeOf(Iterable items) {  
2     int size = 0;  
3     if (items instanceof Collection c) {  
4         size = c.size();  
5     } else {  
6         for (Object item : items) {  
7             ++size;  
8         }  
9     }  
10    return size;  
11 }
```



C#, Rust, and pattern matching

C#:

```
1 public int sizeof<T>(IEnumerable<T> items) {  
2     if (items is IList<T> list) {  
3         return list.Count;  
4     } else {  
5         return // count them one by one  
6     }  
7 }
```

Some other languages have pattern matching feature, including Kotlin, Scala, Haskell, Elixir, Swift, F#, and Erlang, ... which **contradicts** the principle of encapsulation.

Rust:

```
1 enum Color {  
2     RGB(u8, u8, u8),  
3     Transparent  
4 }  
5 fn paint(c: Color) {  
6     match c {  
7         Color::RGB(r, g, b) =>  
8             println!("#{r}{g}{b}"),  
9         Color::Transparent =>  
10            println!("none")  
11     }  
12 }  
13 fn main() {  
14     let c = Color::RGB(64, 16, 0);  
15     paint(c);  
16 }
```

Chapter #3:

Factory Method

[[IF](#) forName]

Conditional object construction

This is **wrong**:

```
1 interface Figure
2   int surface();
3 class Square implements Figure
4 class Triangle implements Figure
5 class Polygon implements Figure
6
7 class FactoryOfFigures
8   Figure make(int sides) {
9     if (sides == 3) {
10      return new Triangle();
11    } else if (sides == 4) {
12      return new Square();
13    } else {
14      return new Polygon(sides);
15    }
16  }
```

This is **better**:

```
1 class PolymorphicFigure implements Figure
2   PolymorphicFigure(int sides)
3   @Override int surface() {
4     if (sides == 3) {
5       return new Triangle().surface();
6     } else if (sides == 4) {
7       return new Square().surface();
8     } else {
9       return new Polygon(sides).surface();
10    }
11  }
```

Here, the semantic of object construction is not visible to the client—the coupling is “loose” [Bugayenko, 2022].

[IF [forName](#)]

Generating class name from a string

This is **wrong**:

```
1 interface Figure
2     int surface();
3
4 class Square implements Figure
5 class Triangle implements Figure
6 class Polygon implements Figure
7
8 class FactoryOfFigures
9     Figure make(String name) throws Exception {
10         Class<?> c = Class.forName(name);
11         return c.getConstructor().newInstance();
12     }
```

This is **better**:

```
1 class PolymorphicFigure implements Figure
2     PolymorphicFigure(String name)
3     @Override int surface() {
4         if (name.equals("Triangle")) {
5             return new Triangle().surface();
6         } else if (name.equals("Square")) {
7             return new Square().surface();
8         } else {
9             return new Polygon().surface();
10        }
11    }
```

This is better since the mechanics of class finding is explicit—no surprises expected [Bugayenko, 2017].

Chapter #4:

Classpath Scanning

Finding Java classes

```
1 interface Foo {}
2
3 class Bar implements Foo {}
4
5 Reflections rts =
6     new Reflections("");
7 Set<Class<?>> types = rts.get(
8     SubTypes.of(Foo.class).asClass()
9 );
```

```
1 public @interface Foo {}
2
3 @Foo
4 class Bar {}
5
6 Reflections rts =
7     new Reflections("");
8 Set<Class<?>> types = rts.get(
9     SubTypes.of(
10         TypesAnnotated.with(Foo.class)
11     ).asClass()
12 );
```

The library is called Reflections. Instead, use explicit object instantiation.

Chapter #5:

Annotations

I lieu of static methods

```
1 interface Pub
2     String isbn();
3
4 class Book implements Pub
5     @Override public String isbn()
6         /* ... */
7     public static String category()
8         return "book";
9
10 class Journal implements Pub
11     @Override public String isbn()
12         /* ... */
13     public static String category()
14         return "journal";
```

```
1 interface Pub
2     String isbn();
3
4 @Target(ElementType.CLASS)
5 @Retention(RetentionPolicy.SOURCE)
6 public @interface Category
7     String value();
8
9 @Category("book")
10 class Book implements Pub
11     @Override public String isbn()
12         /* ... */
13
14 @Category("journal")
15 class Journal implements Pub
16     @Override public String isbn()
17         /* ... */
```

Locating methods

```
1 @Target(ElementType.METHOD)
2 @Retention(RetentionPolicy.SOURCE)
3 public @interface Path
4     String url;
5
6 class BookController
7     @Path("/book-title")
8     String title()
9     // Build HTML page and return it
```

```
1 String dispatch(String url) {
2     c = new BooksController();
3     b = BooksController.class;
4     for (Method m : b.getDeclaredMethods()) {
5         if (m.isAnnotationPresent(Path.class)) {
6             Annotation a = m.getAnnotation(Path.class);
7             if (a.url().equals(url)) {
8                 return m.invoke(c);
9             }
10        }
11    }
12    return "404 Page not found";
13 }
```

The `|BookContoller|` doesn't know how exactly the data in its annotation is being used and by whom [Bugayenko, 2016].

Dependency Injection Container

```
1 interface Shipment
2     int cost();
3
4 class Cart
5     @Inject private Shipment shmt;
6     private Book book;
7     void setBook(Book b)
8         this.book = b;
9     int cost()
10         return this.book.price() + this.shmt.cost();
11
12 container = new Container();
13 c = container.make(Cart.class);
14 c.setBook(new Book("1984"));
15 x = c.cost();
```

```
1 class Container {
2     private HashMap<Class, Object> cache =
3         new ConcurrentHashMap<>();
4     T make(Class<T> type) {
5         // 1. Find @Inject-annotated "shmt" field;
6         // 2. Make an instance of "Shipment";
7         // 3. Store it in the "cache";
8         // 4. Make an instance of "Cart";
9         // 5. Store "cart" in the "cache";
10        // 6. Assign "shipment" to "cart.shmt";
11        // 7. Return "cart".
12    }
13 }
```

How do you think, at the step no.2, what class will be instantiated? [Bugayenko, 2014]

Dependency Injection *without* a Container

```
1 interface Shipment
2     int cost();
3
4 class Cart
5     @Inject private Shipment shmt;
6     private Book book;
7     void setBook(Book b)
8         this.book = b;
9     int cost()
10         return this.book.price() + this.shmt.cost();
11
12 container = new Container();
13 c = container.make(Cart.class);
14 c.setBook(new Book("1984"));
15 x = c.cost();
```

```
1 interface Shipment
2     int cost();
3
4 class Cart
5     private final Shipment shmt;
6     private final Book book;
7     Cart(Shipment s, Book b)
8         this.shmt = s;
9         this.book = b;
10    int cost()
11        return this.book.price() + this.shmt.cost();
12
13 c = new Cart(new MyShipment(), new Book("1984"));
14 x = c.cost();
```

Chapter #6:

Discrimination by Type

Polymorphism vs. Casting

```
1 interface Figure
2     void rotate(int d);
3 class Circle implements Figure
4     void rotate(int d) //...
5     int radius() //...
6 class Square implements Figure
7     void rotate(int d) //...
8     int side() //...
```

Philosophically speaking, type casting is discrimination [Bugayenko, 2020].

```
1 // This is polymorphism:
2 int surface(Figure f)
3     return f.surface();
4
5 // This is type casting:
6 int surface(Figure f)
7     if (f instanceof Circle c) {
8         return c.radius()
9     } else if (f instanceof Square s) {
10        return s.side() * s.side();
11    } else {
12        throw new Exception("oops");
13    }
```

References

Yegor Bugayenko. Dependency Injection Containers Are Code Polluters.

<https://www.yegor256.com/141003.html>, oct 2014. [Online; accessed 22-09-2024].

Yegor Bugayenko. Class Casting Is a Discriminating Anti-Pattern.

<https://www.yegor256.com/150402.html>, apr 2015. [Online; accessed 22-09-2024].

Yegor Bugayenko. Java Annotations Are a Big Mistake.

<https://www.yegor256.com/160412.html>, apr 2016. [Online; accessed 22-09-2024].

Yegor Bugayenko. Constructors or Static Factory Methods?

<https://www.yegor256.com/171114.html>, nov 2017. [Online; accessed 22-09-2024].

Yegor Bugayenko. Strong Typing Without Types.

<https://www.yegor256.com/201110.html>, nov 2020. [Online; accessed 22-09-2024].

Yegor Bugayenko. Reflection Means Hidden Coupling.

<https://www.yegor256.com/220605.html>, jun 2022. [Online; accessed 22-09-2024].

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue.

Self-Inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, pages 27–53.

Springer, 2014. doi:[10.1007/978-3-662-44202-9_2](https://doi.org/10.1007/978-3-662-44202-9_2).

Jonathan M. Sobel and Daniel P. Friedman. An Introduction to Reflection-Oriented Programming, 1996.