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All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as web sites.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

Chapter #1:

What Is Reflection?

Simple Example of Reflection in Java

```
void print(Object x) {
  if (x instance of File) {
    System.out.println("Yes!");
  }
}
```

"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

Chapter #2:

Type Casting and Subsumption

Iterable → Collection

Downcasting (wrong!):

```
int sizeOf(Iterable items) {
  int size = 0;
  if (items instanceof Collection) {
    size = ((Collection) items).size();
  } else {
    for (Object item : items) {
      ++size;
    }
  }
  return size;
}
```

Method overloading (right!):

```
int sizeOf(Iterable items) {
  int size = 0;
  for (Object item : items) {
    ++size;
  }
  return size;
}

int sizeOf(Collection items) {
  return items.size();
}
```

Implicit Coupling

```
int sizeOf(Iterable items) {
  int size = 0;
  if (items instanceof Collection) {
    size = ((Collection) items).size();
  } else {
    for (Object item : items) {
      ++size;
    }
  }
  return size;
}
```

```
Iterable<Book> books1 =
  new PgBooks1("localhost:5432");
int s1 = sizeOf(books1);

Collection<Book> books2 =
  new PgBooks2("localhost:5432");
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!):

```
int sizeOf(Iterable items) {
  int size = 0;
  if (items instanceof Collection) {
    size = ((Collection) items).size();
  } else {
    for (Object item : items) {
      ++size;
    }
  }
  return size;
}
```

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

```
int sizeOf(Iterable items) {
  int size = 0;
  if (items instanceof Collection c) {
    size = c.size();
  } else {
    for (Object item : items) {
      ++size;
    }
  }
  return size;
}
```

C#, Rust, and pattern matching

C#:

```
public int sizeOf<T>(IEnumerable<T> items) {
  if (items is IList<T> list) {
    return list.Count;
  } else {
    return // count them one by one
  }
}
```

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

Rust:

```
enum Color {
  RGB(u8, u8, u8),
  Transparent
}
fn paint(c: Color) {
  match c {
    Color::RGB(r, g, b) =>
      println!("#{r}{g}{b}"),
    Color::Transparent =>
      println!("none")
  }
}
fn main() {
  let c = Color::RGB(64, 16, 0);
  paint(c);
}
```

Chapter #3:

Factory Method

Conditional object construction

This is wrong:

```
interface Figure
  int area();
class Square implements Figure
class Triangle implements Figure
class Polygon implements Figure

class FactoryOfFigures
  Figure make(int sides) {
    if (sides == 3) {
      return new Triangle();
    } else if (sides == 4) {
      return new Square();
    } else {
      return new Polygon(sides);
    }
  }
```

This is better:

```
class PolymorphicFigure implements Figure
  PolymorphicFigure(int sides)
  @Override int area() {
    if (sides == 3) {
      return new Triangle().area();
    } else if (sides == 4) {
      return new Square().area();
    } else {
      return new Polygon(sides).area();
    }
  }
```

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

Generating class name from a string

This is wrong:

```
interface Figure
  int area();

class Square implements Figure
class Triangle implements Figure
class Polygon implements Figure

class FactoryOfFigures
  Figure make(String name) throws Exception {
    Class<?> c = Class.forName(name);
    return c.getConstructor().newInstance();
  }
```

This is better:

```
class PolymorphicFigure implements Figure
  PolymorphicFigure(String name)
  @Override int area() {
    if (name.equals("Triangle")) {
      return new Triangle().area();
    } else if (name.equals("Square")) {
      return new Square().area();
    } else {
      return new Polygon(?).area();
    }
  }
```

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

Chapter #4:

Classpath Scanning

Finding Java classes

```java
interface Foo {}

class Bar implements Foo {}

Reflections rts =
  new Reflections("");
Set<Class<?>> types = rts.get(
  SubTypes.of(Foo.class).asClass()
);
```

```java
public @interface Foo {}

@Foo
class Bar {}

Reflections rts =
  new Reflections("");
Set<Class<?>> types = rts.get(
  SubTypes.of(
    TypesAnnotated.with(Foo.class)
  ).asClass()
);
```

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

Chapter #5:

Annotations

I lieu of static methods

```
interface Pub
  String isbn();

class Book implements Pub
  @Override public String isbn()
    /* ... */
  public static String category()
    return "book";

class Journal implements Pub
  @Override public String isbn()
    /* ... */
  public static String category()
    return "journal";
```

```
interface Pub
  String isbn();

@Target(ElementType.CLASS)
@Retention(RetentionPolicy.SOURCE)
public @interface Category
  String value();

@Category("book")
class Book implements Pub
  @Override public String isbn()
    /* ... */

@Category("journal")
class Journal implements Pub
  @Override public String isbn()
    /* ... */
```

Locating methods

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Path
  String url;

class BookController
  @Path("/book-title")
  String title()
    // Build HTML page and return it
```

```
String dispatch(String url) {
  c = new BooksController();
  b = BooksController.class;
  for (Method m : b.getDeclaredMethods()) {
    if (m.isAnnotationPresent(Path.class)) {
      Annotation a = m.getAnnotation(Path.class);
      if (a.url().equals(url)) {
        return m.invoke(c);
      }
    }
  }
  return "404 Page not found";
}
```

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

Dependency Injection Container

```
interface Shipment
  int cost();

class Cart
  @Inject private Shipment shmt;
  private Book book;
  void setBook(Book b)
    this.book = b;
  int cost()
    return this.book.price() + this.shmt.cost();

container = new Container();
c = container.make(Cart.class);
c.setBook(new Book("1984"));
x = c.cost();
```

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

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

Dependency Injection *without* a Container

```
interface Shipment
  int cost();

class Cart
  @Inject private Shipment shmt;
  private Book book;
  void setBook(Book b)
    this.book = b;
  int cost()
    return this.book.price() + this.shmt.cost();

container = new Container();
c = container.make(Cart.class);
c.setBook(new Book("1984"));
x = c.cost();
```

```
interface Shipment
  int cost();

class Cart
  private final Shipment shmt;
  private final Book book;
  Cart(Shipment s, Book b)
    this.shmt = s;
    this.book = b;
  int cost()
    return this.book.price() + this.shmt.cost();

c = new Cart(new MyShipment(), new Book("1984"));
x = c.cost();
```

Chapter #6:

Discrimination by Type

Polymorphism vs. Casting

```
interface Figure
  void rotate(int d);
class Circle implements Figure
  void rotate(int d) //...
  int radius() //...
class Square implements Figure
  void rotate(int d) //...
  int side() //...
```

```
// This is polymorphism:
int area(Figure f)
  return f.area();

// This is type casting:
int area(Figure f)
  if (f instanceof Circle c) {
    return c.radius()
  } else if (f instanceof Square s) {
    return s.side() * s.side();
  } else {
    throw new Exception("oops");
  }
```

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

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.

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