Reflection

Classpath, Casting, Annotations

Yegor Bugayenko

Lecture #7 out of 8
90 minutes

- Type Casting and Subsumption
- Factory Method
- Classpath Scanning
- Annotations
- Discrimination by Type
- Read and Watch

Chapter #1:

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

```
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`, since the signature of `sizeOf()` is the same in both cases.

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 #2:

Factory Method

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 — coupling is loose.

Generating class name from a string

This is wrong:

```
interface Figure
  int surface();

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 surface() {
    if (name.equals("Triangle")) {
      return new Triangle().surface();
    } else if (name.equals("Square")) {
      return new Square().surface();
    } else {
      return new Polygon(?).surface();
    }
  }
```

This is better since the mechanics of class finding is explicit — no surprises expected.

Chapter #3:

Classpath Scanning

Finding Java classes

```
interface Foo {}

class Bar implements Foo {}

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

```
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 #4:

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";
}
```

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?

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 #5:

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 surface(Figure f)
  return f.surface()

// This is type casting:
int surface(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");
  }
```

Chapter #6:

Read and Watch

Dependency Injection Containers are Code Polluters by me (2014)

Class Casting Is a Discriminating Anti-Pattern by me (2015)

Java Annotations Are a Big Mistake by me (2016)

Reflection Means Hidden Coupling by me (2022)

Java Annotations Are a Bad Idea, at JDK.io conference (2017)

Constructors or Static Factory Methods? by me (2017)

Strong Typing without Types by me (2020)