Using Other Defensive Practices

Andrejs Doronins
TEST AUTOMATION ENGINEER

(Better)
Encapsulation

Side effects

Exception handling

Static analysis tools

Next steps

Encapsulation

Encapsulation

Making your fields private and provide public getters and setters for them

```
class Booking {

        private String id;

        private Flight flight;



        public String getId(){ return id; };

        public void setId(String id){ this.id = id; };

        public Flight getFlight(){ return flight;}

        public void setFlight(Flight f){ this.flight = f;}

}
```



```
flight.setId(null);
```

Encapsulation

Information and implementation hiding

Make each member as inaccessible as possible

Don’t provide getters and setters for no reason

```
class Booking {

        private String id;

        private Flight flight;


        private
        public void doSomething(){ /* ... */ }

        public getFlight(){ return flight;

        public setId(String id){ this.id = id; },

}
```

Less exposed (public) code means less hassle

Having unnecessary setters:

- You might forget to add defensive code
- Leads to unnecessary code duplication
- More frequent triggering of guard clauses and causing runtime exceptions

Methods:

- return a value
- modify state (and return void)

```
double result =          multiply(a,b);
```

```
savePerson(john);
```

side effect

Side Effect

Happens if a method modifies some state outside its local scope

```
class SomeClass {

    private int someField;

    scope
    public void doSomething(){

            //..

    }

}
```

Don't make methods return values AND produce side effects

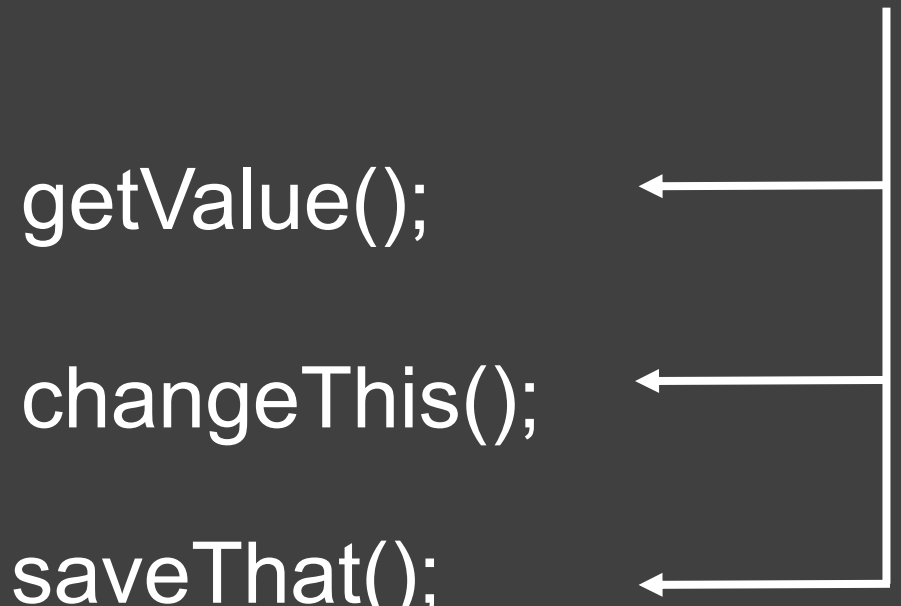
```
var a = obj1.getValue();
```

(quietly change)

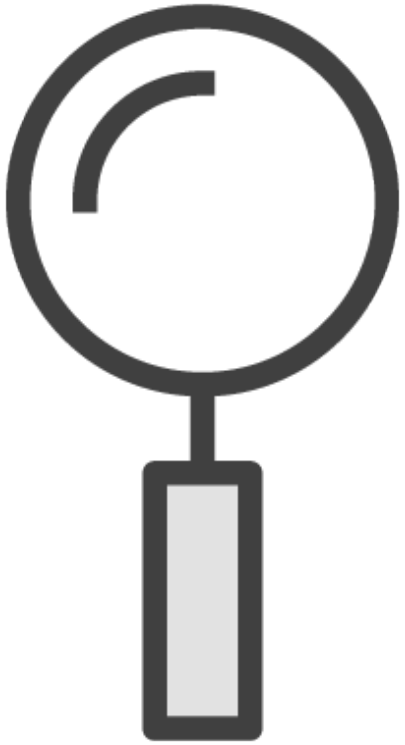
obj2

```
obj2.doThat();
```

getValueAndChangeThisAndSaveToFile();



CQS
Command-Query Separation



Some CQS exceptions:

- Logging
- Intentional design
  - String e = stack.pop();
  - Response r = httpClient.sendPost();

"When used to best advantage, exceptions can improve a program's readability, reliability and maintainability. When used improperly, they can have the opposite effect."

Joshua Bloch

Exception Handling

DOs

Use Java 7 try-with-resources

Pass useful and pertinent information to your exceptions

DONTs

Catch top-level Throwable or Exception

Catch NullpointerException

Catch and swallow (do nothing in catch blocks)

java.lang.IllegalArgumentException: Parsing failed

java.lang.IllegalArgumentException: Could not parse input date

Insufficient Exception Information

java.lang.IllegalArgumentException: Could not parse input date 15/10-2019, the expected format is dd-MM-yyyy

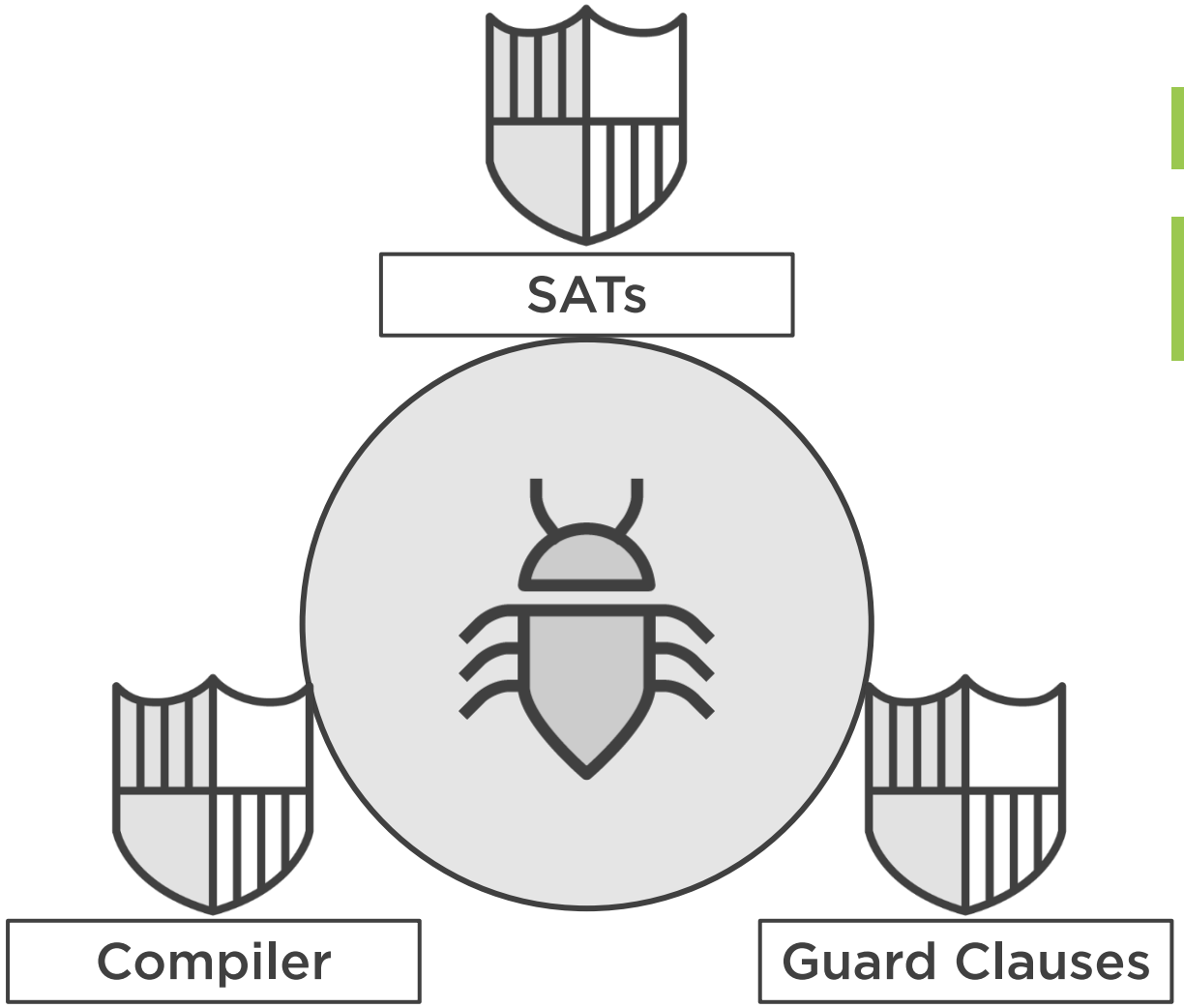
[... stack trace ...]

Caused by: java.time.format.DateTimeParseException: Text '15/10-2019' could not be parsed at index 2

Sufficient Exception Information

Preventing > Reacting

Static Analysis Tools (SATs)



Code Reviews

CI pipelines and Integration Tests

Static Analysis Tools

IDE

SonarLint

React

Fail early with a guard clause in methods

Catch the right exceptions and if you rethrow - provide all pertinent information

Prevent

Fail early with a guard clauses in constructors

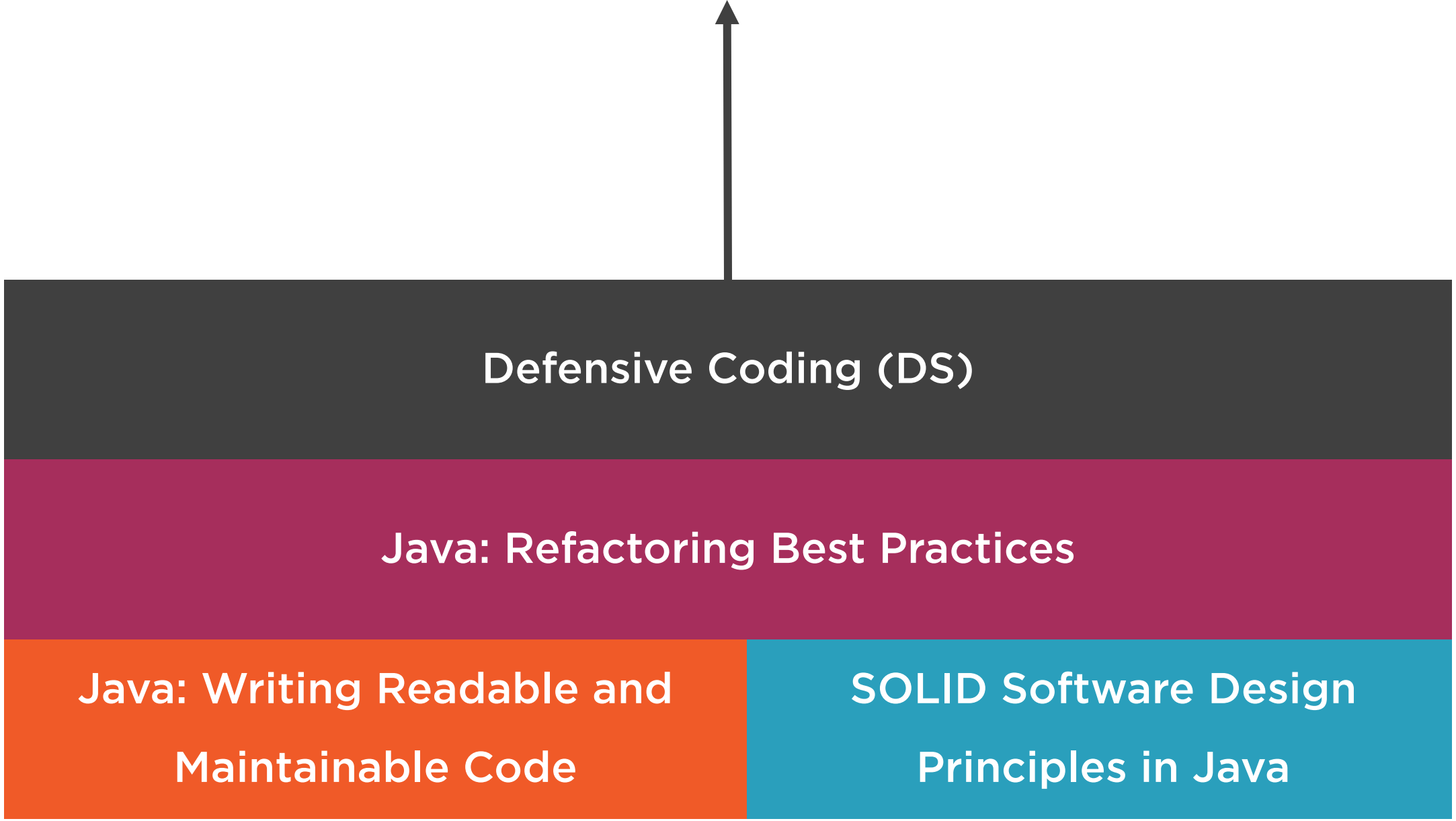
Return only expected values

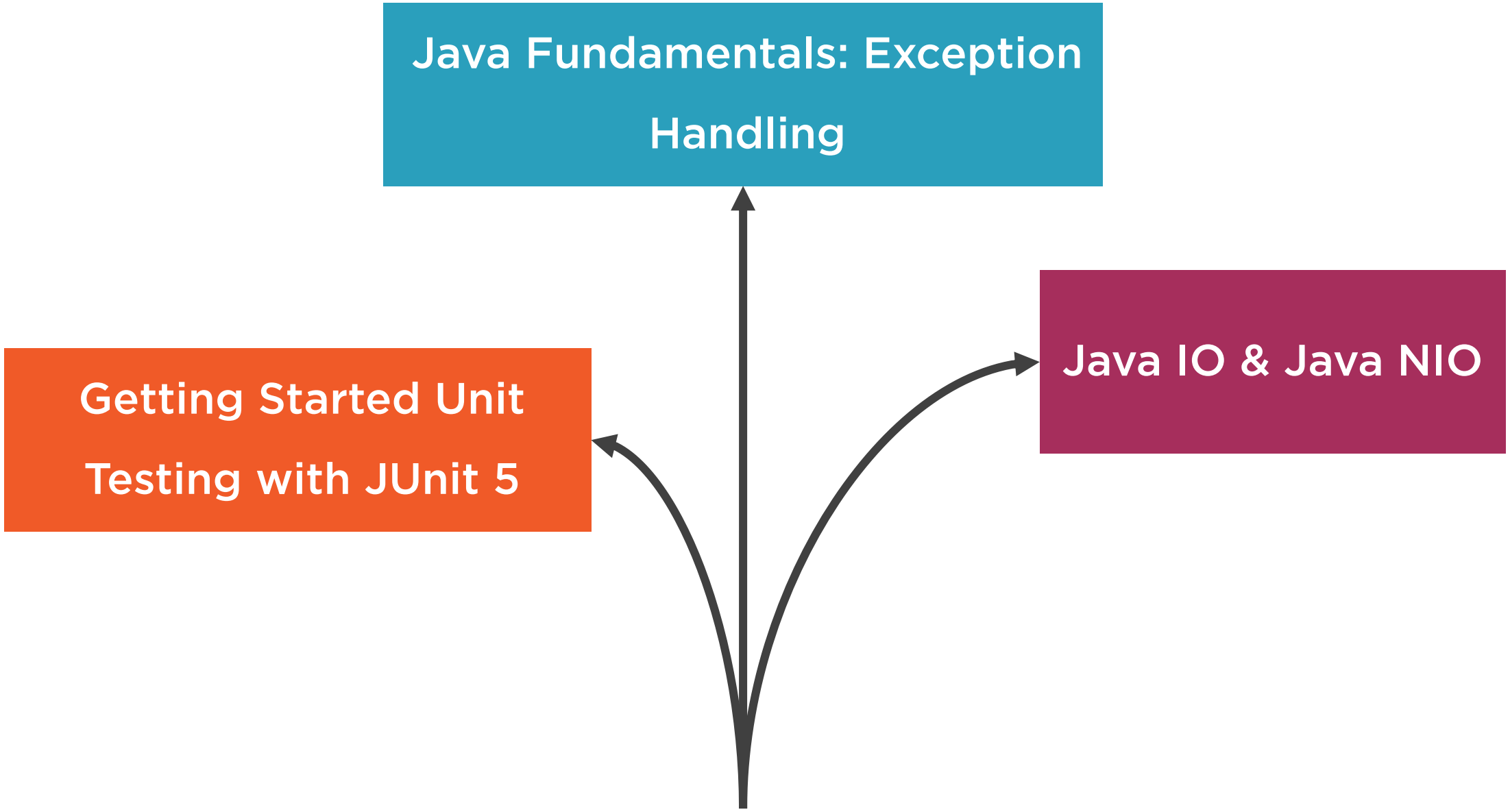
Don’t return nulls

Implement better encapsulation

Follow the CQS principle

Use static analysis tools


Defensive Coding (DS)
Java: Refactoring Best Practices
Java: Writing Readable and Maintainable Code
SOLID Software Design Principles in Java


Java Fundamentals: Exception Handling
Getting Started Unit Testing with JUnit 5
Java IO & Java NIO

Java IO

```
boolean is_accessible =
                    isRegularFile(path) && isReadable(path) &&
                          isExecutable(path) &&  isWritable(path);


if (is_accessible) { /* ... */ }
```

Summary

- Strive for tight encapsulation
- Produce side effects with care
- Exception handling != Exception hiding
- Use static analysis tools

Good luck!