Programming Fundamentals 2

Pierre Talbot

30 March 2021

University of Luxembourg

Chapter VIII. Error Management

Error management

We are going to study two aspects of error management:

- How to report and handle errors generated by a method.
- How to avoid unwanted errors by testing.

Error handling

Case study: search a Pokemon card

We are considering the laboratory 2 as an example. In this laboratory, we have a deck of cards and we want to search for a card meeting a criterion such as finding a card with a specific name.

An example of signature for this method is:

```
public Card searchByName(String name) { ... }
```

What to do if the card is not in the collection?

We look at various solutions, proposed in your labs, to this problem:

(I) Return a special value (*e.g.*, `null` or `-1`),

(II) Return the string representation of the card directly,

(III) Return an error object of the same type,

(IV) Return a list,

(V) Throw an exception.

Case Study Ia: Return `null`

```
public Card getCardById(String id) {
  for (Card card : deck) {
    if(id.equals(card.getId())) {
      return card;
    }
  }
  return null;
}
```

If the card is not in the deck, `null` is returned.

Case Study Ib: Return -1

```
public int findCardIndexByNumber(int cardNumber)
{
  int index = -1;
  for (int i=0; i < deck.size() ; ++i) {
    if (deck.get(i).getCardNumber() == cardNumber){
      index = i;
      break;
    }
  }
  return index;
}
```

Instead of directly returning a card, we return its index in the deck, and -1 if it is not inside.

Case Study I: Discussion

This kind of error handling is very common in language such as C. One problem with such scheme is on the calling site:

```
public void modifyCard(String id) {
  Card card = getCardById(id);
  if (card != null)
    card.modify();
}
```

- Each time the method is called, we must check for `card != null` or `cardIdx != -1`.
- We can easily forget to check that, and generate a `NullPointerException` or `OutOfBoundException`.
- Normally, **this solution is not the way to go**.
- See also *Code Clean*, Chapter 7, "Don't return `null`".

Case Study I: Discussion

Further, it leads to code harder to read when all the error handling is performed that way:

```
int id = askUserForID();
if(id == -1) {
  wrongUserInput();
}
else {
  Card card = getCardById(id);
  if(card == null) {
    wrongID();
  }
  else {
    // ...
  }
}
```

The logic of the code is lost in error handling, the code is actually quite simple:

```
int id = askUserForID();
Card card = getCardById(id);
// ...
```

Case Study II: Return the String representation

```
public String searchCardByNumber(){
  System.out.println("What is the number you want to search?");
  String searchedNumber= input.getInput();
  for(int i=0;i < cards.size(); i++){
    if(cards.get(i).number.equals(searchedNumber))
      return "Searched card: \n" +cards.get(i).toString();
  }
  return "Card number doesn't exist.";
}
```

Case Study II: Discussion

This solution has massive downsides:

- We cannot reuse `searchCardByNumber` for something else (e.g., modifying the card),
- We cannot use `searchCardByNumber` if we already obtained the ID from another source,
- The user interface is tightly coupled with the business logic: hard to maintain.
- This function has too many responsibilities: (i) ask a number to the user, (ii) search the number, (iii) prepare the resulting output.
- **This is not a good solution neither**.
- Bad variant: `searchCardByNumber` directly prints the message and returns nothing.

Case Study III: Return an error object of the same type

```
public Card searchByName(String name) {
  Card matching = new Card (" Not Found", " ", 0);
  for(int i=0; i < cards.size(); i++)
  {
    if (cards.get(i).getName().equals(name))
    {
      matching = cards.get(i);
      return matching;
    }
  }
  return matching;
}
```

Case Study III: Discussion

This solution is not too bad, but has a strong disadvantage:

This method expects to be called in a specific context.

That is, it expects the card to be printed immediately afterwards.

- What if we use this method in another context, *e.g.*, to find a card to modify?
- Should we let the user modify a card that doesn't exist?
- How to detect the card doesn't exist?

Note that in some other places, an improvement of this solution can be good, c.f., Code Clean, Chapter 7, "Define the Normal Flow".
See also the Special Case Design Pattern.

Case Study IV: Return a list

```
public ArrayList<Card> getCardsByName(String name) {
  ArrayList<Card> searchResults = new ArrayList<Card>();
  for (Card card : deck) {
    if(name.equals(card.getName())) {
      searchResults.add(card);
    }
  }
  return searchResults;
}
```

Case Study IV: Discussion

This solution is a good one:

- It returns an empty list if no card matches the name,
- It is callable in any kind of context,
- It generalizes the previous method to cards with multiple names (why can it happen though?).

See also *"Item 54: Return empty collections or arrays, not nulls"*, *Effective Java.*

Case Study IVb: Return a Optional

```
public Optional<Card> getCardByName(String name) {
  for (Card card : deck) {
    if(name.equals(card.getName())) {
      return Optional.of(card);
    }
  }
  return Optional.empty();
}
```

Case Study IVb: Discussion

On the calling site, we cannot forget to check that the card exists (unlike with `null` and `-1`), because this information is built in the return type.

```
Optional<Card> card_opt = getCardByName(name);
if(card_opt.isPresent()) {
  Card card = card_opt.get();
}
else {
  // ...
}
```

However, the code can become less clear (similarly than with return code). The methods `Optional.ifPresent`, `Optional.map`,... can help for this purpose.

See also *"Item 55: Return optionals judiciously", Effective Java.*

Case Study V: Throw an exception

```
public Card searchCardByName(String name){
  for (Card card : cards) {
    if(card.name().equal(name)) {
      return card;
    }
  }
  throw new RuntimeException("Card not found");
}
```

Case Study V: Discussion

This is the most idiomatic way of reporting an error in Java. Exceptions are, however, not perfect. We give a more in-depth explanation of exceptions in the following slides.

Exception

Syntax of exception

```
ThrowStatement:
  throw Expression ;      throw new CardNotFound(cardName);

TryStatement:
  try Block Catches             try { ... } catch(CardNotFound c) { ... }
  try Block [Catches] Finally   try { ... } catch(CardNotFound c) { ... } finally { ... }
```

Exception by example

```
public class CardNotFoundException extends RuntimeException {
  private String name;
  public CardNotFound(String name) { this.name = name; }
  public String toString() {
    return "The card " + name + " could not be found in the deck.";
  }
}

public Card searchCardByName(String name) {
  for (Card card : cards) { ... }
  throw new CardNotFoundException(name);
}

public void printCard(String name) {
  try {
    Card card = searchCardByName(name);
    System.out.println(card);
  }
  catch(CardNotFoundException e) {
    System.out.println(e);
  }
}
```

Advantages and disadvantages of exceptions

Advantages

- Exceptions provide a **clean way** to handle errors separately from the normal flow of the code.
- They are **non-intrusive**, meaning that the signature of the method does not need to be modified.
- Exceptions can be **arbitrarily rich** in information.

Disadvantages

- It is sometimes hard to figure out the exceptions a method can throw, documentation is therefore important for this purpose.

  See "Item 74: Document all exceptions thrown by each method", Effective Java
- Can be easily ignored, and ends up in the `main` function, which then exits and prints the exception calling stack.

Additional feature I: Checked exception

Place the exceptions a method can throw in the signature of its method:

```
public Card searchCardByName(String name) throws CardNotFoundException {
  for (Card card : cards) { ... }
  throw new CardNotFoundException(name);
}
```

This forces the calling method to treat the exception, however:

- If the exception is treated higher in the calling stack, it forces all the intermediate calling methods to add this exception to their signatures.
- It is tedious to use in practice, and not too useful.
- As suggested by *Code Clean* (Chapter 7), we will avoid using checked exceptions.
- However, it is not a universal point of view, see *"Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors", Effective Java.*

Additional feature II: try-with-resources

When acquiring a resource, such as `Scanner`, a file or a network socket, we must close it after using it:

```
public Optional<String> readFirstLineOf(String path) {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return Optional.of(br.readLine());
  } catch(IOException) {
    return Optional.empty();
  } finally {
    br.close();
  }
}
```

Additional feature II: try-with-resources

The *try-with-resources* statement is a convenient syntactic sugar to automatically closing a resource:

```
public Optional<String> readFirstLineOf(String path) {
  try (BufferedReader br =
         new BufferedReader(new FileReader(path))) {
    return Optional.of(br.readLine());
  } catch(IOException) {
    return Optional.empty();
  }
}
```

No need for the finally block, `br` is closed automatically.

Additional feature II: try-with-resources

The try-with-resources statement works with any class implementing the interface `Closeable`. For instance with `Scanner`:

```
public Optional<Integer> readInteger() {
  try (Scanner scanner = new Scanner(System.in)) {
    if(scanner.hasNextInt()) {
      return Optional.of(scanner.nextInt());
    }
  }
  return Optional.empty();
}
```

Testing

Black-box vs white-box testing

Two main categories of testing:

- **Black-box testing**: we test the functionalities of a system based on its input-output. This is how we tested *Connect Four*.
- **White-box testing**: The internal methods of the system are tested. This is (almost) how we tested *DynamicArray*.

Testing an overall behavior is generally done by black-box testing. This is also much easier to test GUI that way.

Here, we will focus on *unit testing* which is a form of white-box testing.

Why testing our project?

- **To find some bugs** before they appear in production.
- **To be the first user of our method**: a method hard to test will be hard to use.
- **To trust our code**: when we modify a part of our code, we can run the tests to verify nothing is broken.
- **To gain time**: debugging is very long and painful.

How to unit test?

As we have shown for `DynamicArray`, we do not need anything special to start unit testing. However, it might be convenient to use a dedicated testing framework, here we will use JUnit 5, aka. JUnit Jupiter. (`https://junit.org/junit5/docs/current/user-guide/`)

Example

You can retrieve a sample test project testing the class `DynamicArray` by doing:

```
git clone https://github.com/ptal/lab2-pokedeck/
cd lab2-pokedeck
git checkout testing
mvn test
```

Add to pom.xml

```
...
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
    </plugins>
</build>
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.7.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.7.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

(See https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven)

Create a JUnit test

- Create the folder `src/test/java/`.
- Inside, it can follow the same package hierarchy than in `src/main/java`, *e.g.*, `src/test/java/lab2/pcg/DeckTest.java`.
- You create one test class per class you want to test, *e.g.*, to test `DynamicArray.java`, you create the class `DynamicArrayTest.java`.

Testing a method

```
class DynamicArrayTest {
  @Test
  @DisplayName("Add elements in DynamicArray")
  void testAdd() {
    DynamicArray array = new DynamicArray();
    assertEquals(array.size(), 0);
    assertTrue(array.isEmpty());
    array.clear();
    assertEquals(array.size(), 0);
    assertTrue(array.isEmpty());
    array.add(4);
    array.add(5);
    array.add(6);
    assertEquals(array.toString(),"[4, 5, 6]");
  }
```

- `@Test`: only the methods with this annotation are called for testing.
- `assertEquals(expr1, expr2)` checks that both expressions are equal.
- `assertTrue(expr)` checks the expression is true.

BeforeEach annotation

`DynamicArrayTest` is a normal class, so we can declare attributes:

```
class DynamicArrayTests {
  private DynamicArray array;

  @BeforeEach
  void init() {
    array = new DynamicArray();
  }

  @Test
  @DisplayName("Add elements in DynamicArray")
  void testAdd() {
    assertEquals(array.size(), 0);
    ...
  }
```

Instead of declaring and initializing `array` in all testing methods, we use `@BeforeEach` to call `init()` before each method.

Testing for exceptions

```
@Test
@DisplayName("Add and remove elements in DynamicArray")
void testRemove() {
  populate(); // add the elements 4, 5, 6 in the array.
  array.remove(1);
  testArrayContent(2, "4, 6");
  array.remove(1);
  testArrayContent(1, "4");
  assertThrows(ArrayIndexOutOfBoundsException.class, () -> array.remove(1));
  array.remove(0);
  testArrayContent(0, "");
}
```

`assertThrows(E.class, () -> x.f())` tests that the method `x.f()` is throwing an exception of type `E`.

What is a good test? FIRST!

- *[F]ast*: Tests must be very fast so we run them frequently.
- *[I]solated*: Tests must not connect to a database, the network, ...
- *[R]epeatable*: Running the same test 10 times must give the same result. Randomness is proscribed.
- *[S]elf-validating*: The process of verifying if a test succeeds must be automatic, *e.g.*, we shall not need to read the output of a test.
- *[T]imely*: Don't write Java code without test, 1 method = 1 test.

Source: Pragmatic unit testing in Java 8 with JUnit

Write a good test: Right-BICEP

- *Right* Are the results right?
- *B* Are all the boundary conditions correct?
- *I* Can you check inverse relationships?
- *C* Can you cross-check results using other means?
- *E* Can you force error conditions to happen?
- *P* Are performance characteristics within bounds?

Source: Pragmatic unit testing in Java 8 with JUnit

Test Driven Development (TDD)

Methodology where the tests are central to the project:

- Instead of writing the code, then the tests, you do the opposite!
- Because we write the tests first, it forces us to think about the usability of our methods.

More about testing in Software Engineering 1 and Software Engineering 2.