Exceptions

Table of contents

Outline 1
Exceptions 1
Specification 2
Key takeaways 5

See csse2002[1] for course logistics. A supplementary lecture on java-version-control[2] also ran this week.

Code samples are in the course CodeCamp playlist and Git repo.

Outline

1. **Exceptions**: what an exception is, the hierarchy, checked vs unchecked, `try/catch`, `throw`, `throws`, `finally`, custom exceptions.
2. **Specification**: Javadoc → Restrictiveness → Generality → Formality → Contracts → Defensive Programming.

Exceptions

See java-exceptions[3] — the exception hierarchy, checked vs unchecked exceptions, `try/catch/throw/throws/finally`, and custom exceptions. The `finally` exercise (`Good Morning` / `hello` / `world`) is worked there; week5-tutorial-exceptions[4] then pushes the same tracing into nested blocks.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-version-control.pdf
[3] courses/csse2002/java-exceptions.pdf
[4] courses/csse2002/week5-tutorial-exceptions.pdf

Specification

This repeats Week 3's material: Javadoc, restrictiveness, generality, clarity, formality, contracts, defensive programming. See java-specification[5], which also has an early-return variant of the `search` generality example and an Effective Java citation for defensive programming/doc comments.

The rest of this section covers the examples that are not in that note.

The `search` array — why "an index" isn't a specification

Restrictiveness on `search`, in three steps. The first spec:

```
/**
 * Returns an index (i) of ar such that ar[i] == x, if any.
 */
public int search(int[] ar, int x) { … }
```

This is a bad specification. It is silent about `x` not being in `ar`, and by its silence it allows any value to be returned. A return of 1 or 0 could equally mean "it's at index 1/0" or "I didn't find it".

Adding `else, return -1` is better but still bad. This array is why:

Nothing in that spec requires that you get the lowest index, or that you get the same answer running multiple times. Only "return the **smallest** index (i) of ar such that ar[i] == x, if any, else return -1" fixes both. Two different implementations can both satisfy that wording, which is the generality example.

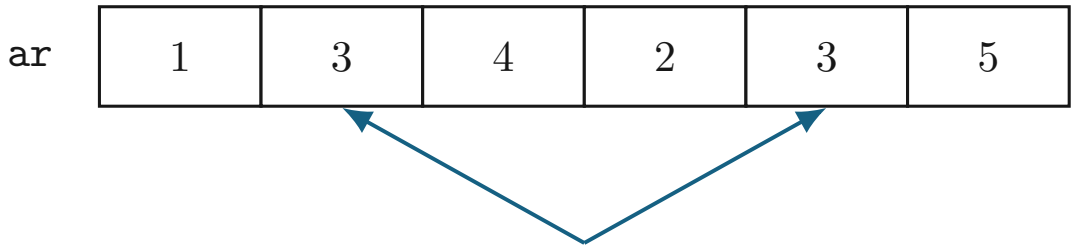
Worked: is this a good specification?

Four specs, each judged against the three criteria: restrictive, general, clear.

```
/**
 * Sorts the given array.
 */
public void sort(int[] ar) { … }
```

Not restrictive enough: ascending or descending is never stated, so both satisfy it.

[5] courses/csse2002/java-specification.pdf



`search(ar, 3)` satisfies "an index `i` with `ar[i] == x`" by returning *either* 1 *or* 4.

The specification is non-deterministic.

Figure 1: The array `ar` is the counterexample to the improved spec: `3` sits at both index 1 and index 4, so a specification asking only for *an* index `i` with `ar[i] == x` lets an implementation return either, and lets it return a different one on each call.

```
/**
 * Calculate x to the power of two with Math.pow.
 */
public int square(int x) { … }
```

Not general enough — the classic generality failure. It names an implementation (`Math.pow`), so it describes *how* rather than *what*, and an `x * x` implementation that computes exactly the same answer technically violates it.

```
/**
 * Returns the list of ResourceContainers stored by this instance.
 */
public List<ResourceContainer> getResources() { … }
```

Not restrictive enough about aliasing: it doesn't say whether the caller gets the internal list or a copy, so it doesn't say whether adding to the returned list changes the object — see java-mutability[6].

[6] courses/csse2002/java-mutability.pdf

```
/**
 * Calculate the nth value of the fibonacci sequence.
 */
public long fib(int n) { … }
```

Not clear enough: it doesn't fix the indexing (is `fib(0)` 0 or 1?), and says nothing about negative `n` or about `n` large enough to overflow `long`.

Contracts — the caller's obligation, and `\result`

The contract form of `withdraw` carries a postcondition on the return value as well as on the state. The call site follows it:

```
/**
 * Withdraws an amount from this account.
 *
 * @require amount >= 0
 * @require amount <= getBalance()
 * @ensure getBalance == \old(getBalance()) - amount
 * @ensure \result == getBalance()
 */
public int withdraw(int amount) {
    // no need to check the conditions
}
```

```
int balance = myAccount.withdraw(amt);
```

The method body needs no checks at all, because it is up to the calling code to make sure it fulfils its side of the contract, i.e. `0 <= amt <= myAccount.getBalance()`. Written **without** a contract, the same method checks both conditions itself and returns the sentinel `-1` on failure, which pushes an `if (balance == -1) { … }` into every call site.

The tags are spelt `@require`/`@ensure` here and `@requires`/`@ensures` elsewhere. Both spellings are used across the course. In the first `@ensure` line the left-hand `getBalance` is the call `getBalance()`.

`findMax` — the undefended version, then the defended one

Defensive programming, before and after. The undefended version is code only, with no Javadoc:

```java
public int findMax(int[] numbers) {
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < numbers.length-1; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}
```

The defended version adds the full specification (`@param`, `@return`, `@throws IllegalArgumentException if the array is null or empty`, `@requires numbers != null && numbers.length > 0`, `@ensures` the maximum is returned) and the guard clause that throws it. That version is in java-specification[7].

The loop bound also changes between the two versions, from `i < numbers.length-1` to `i < numbers.length`. The undefended version never examines the last element.

Key takeaways

- A specification describes the **contract between a method and its client**.
- It explains **what the method does, not how it is implemented**.
- A good specification clearly states its **preconditions** (true before the call), **postconditions** (true after it finishes successfully), and **side effects** (any other changes it makes to the system).
- Clear specifications make code easier to understand, use, and maintain.

Further reading: Oracle's *Preconditions, Postconditions, and Class Invariants* and *Assertions* guides, plus *Effective Java* (3rd Edition) Item 56, "Write doc comments for all exposed API elements".

[7] courses/csse2002/java-specification.pdf