Error handling

Introduction

What

🎓: ★★☆☆ 🏆 Can explain error handling

> Well-written applications include error-handling code that allows them to recover gracefully from unexpected errors. When an error occurs, the application may need to request user intervention, or it may be able to recover on its own. In extreme cases, the application may log the user off or shut down the system. -- Microsoft

---

Exceptions

What

🎓: ★☆☆☆ 🏆 Can explain exceptions

Exceptions are used to deal with *'unusual' but not entirely unexpected* situations that the program might encounter at runtime.

> **Exception**:
>
> > The term *exception* is shorthand for the phrase "exceptional event." An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. –- Java Tutorial (Oracle Inc.)

📦 Examples:

- A network connection encounters a timeout due to a slow server.
- The code tries to read a file from the hard disk but the file is corrupted and cannot be read.

How

🎓: ★☆☆☆ 🏆 Can explain how exception handling is done typically

Most languages allow code that encountered an "exceptional" situation to encapsulate details of the situation in an *Exception* object and *throw/raise* that object so that another piece of code can *catch* it and deal with it. This is especially useful when the code that encountered the unusual situation does not know how to deal with it.

The extract below from the -- Java Tutorial (with slight adaptations) explains how exceptions are typically handled.

> **When an error occurs at some point in the execution, the code being executed creates an *exception object* and hands it off to the runtime system.** The exception object contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing* an exception.

After a method throws an exception, the runtime system attempts to find something to handle it in the *call stack*. The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch* the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the program terminates.

Advantages of exception handling in this way:

- The ability to propagate error information through the call stack.
- The separation of code that deals with 'unusual' situations from the code that does the 'usual' work.

Exercises

When

: ★★☆☆ Can avoid using exceptions to control normal workflow

In general, use exceptions only for 'unusual' conditions. Use normal `return` statements to pass control to the caller for conditions that are 'normal'.

Assertions

What

: ★★☆☆ Can explain assertions

***Assertions* are used to define assumptions about the program state so that the runtime can verify them.** An assertion failure indicates a possible bug in the code because the code has resulted in a program state that violates an assumption about how the code *should* behave.

An assertion can be used to express something like *when the execution comes to this point, the variable* `v` *cannot be null*.

If the runtime detects an *assertion failure*, it typically takes some drastic action such as terminating the execution with an error message. This is because an assertion failure indicates a possible bug and the sooner the execution stops, the safer it is.

In the Java code below, suppose you set an assertion that `timeout` returned by `Config.getTimeout()` is greater than `0`. Now, if `Config.getTimeout()` returns `-1` in a specific execution of this line, the runtime can detect it as an *assertion failure* -- i.e. an assumption about the expected behavior of the code turned out to be wrong which could potentially be the result of a bug -- and take some drastic action such as terminating the execution.

```
1 int timeout = Config.getTimeout();
```

How

🎓: ★★★☆ 🏆 Can use assertions

Use the `assert` keyword to define assertions.

📦 This assertion will fail with the message `x should be 0` if `x` is not 0 at this point.

```
x = getX();
assert x == 0 : "x should be 0";
...
```

Assertions can be disabled without modifying the code.

📦 `java -enableassertions HelloWorld` (or `java -ea HelloWorld`) will run `HelloWorld` with assertions enabled while `java -disableassertions HelloWorld` will run it without verifying assertions.

🚩 **Java disables assertions by default.** This could create a situation where you think all assertions are being verified as `true` while in fact they are not being verified at all. Therefore, remember to enable assertions when you run the program if you want them to be in effect.

💡 Enable assertions in Intellij (how?) and get an assertion to fail temporarily (e.g. insert an `assert false` into the code temporarily) to confirm assertions are being verified.

ℹ️ **Java `assert` vs JUnit assertions: They are similar in purpose but JUnit assertions are more powerful and customized for testing.** In addition, JUnit assertions are not disabled by default. We recommend you use JUnit assertions in test code and Java `assert` in functional code.

📎 Resources

Tutorials:

- **Java Assertions** -- a simple tutorial from *javatpoint.com*
- **Programming with Assertions** (first half) -- a more detailed tutorial from *Oracle*

Best practices:

- **Programming with Assertions** (second half) -- *from Oracle* (also listed above as a tutorial) contains some best practices towards the end of the article.

When

🎓: ★★★☆ 🏆 Can use assertions optimally

It is recommended that assertions be used liberally in the code. Their impact on performance is considered low and worth the additional safety they provide.

Do not use assertions to do *work* because assertions can be disabled. If not, your program will stop working when assertions are not enabled.

The code below will not invoke the `writeFile()` method when assertions are disabled. If that method is performing some work that is necessary for your program, your program will not work correctly when assertions are disabled.

```
...
assert writeFile() : "File writing is supposed to return true";
```

Assertions are suitable for verifying assumptions about *Internal Invariants*, *Control-Flow Invariants*, *Preconditions*, *Postconditions*, and *Class Invariants*. Refer to [Programming with Assertions (second half)] to learn more.

Exceptions and assertions are two complementary ways of handling errors in software but they serve different purposes. Therefore, both assertions and exceptions should be used in code.

- The raising of an exception indicates an unusual condition created by the user (e.g. user inputs an unacceptable input) or the environment (e.g., a file needed for the program is missing).
- An assertion failure indicates the programmer made a mistake in the code (e.g., a null value is returned from a method that is not supposed to return null under any circumstances).

Exercises

Logging

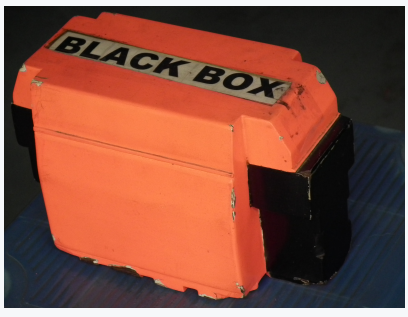
What

★★☆☆ Can explain logging

***Logging* is the deliberate recording of certain information during a program execution for future reference.** Logs are typically written to a log file but it is also possible to log information in other ways e.g. into a database or a remote server.

***Logging* can be useful for troubleshooting problems**. A good logging system records some system information regularly. When bad things happen to a system e.g. an unanticipated failure, their associated log files may provide indications of what went wrong and actions can then be taken to prevent it from happening again.

A log file is like the black box of an airplane; they don't prevent problems but they can be helpful in understanding what went wrong after the fact.



source: https://commons.wikimedia.org

Exercises

How

★★★☆ Can use logging

Most programming environments come with logging systems that allow sophisticated forms of logging. They have features such as the ability to enable and disable logging easily or to change the logging intensity.

This sample Java code uses Java's default logging mechanism.

First, import the relevant Java package:

```
1 import java.util.logging.*;
```

Next, create a `Logger`:

```
1 private static Logger logger = Logger.getLogger("Foo");
```

Now, you can use the `Logger` object to log information. Note the use of a logging level for each message. When running the code, the logging level can be set to `WARNING` so that log messages specified as having `INFO` level (which is a lower level than `WARNING`) will not be written to the log file at all.

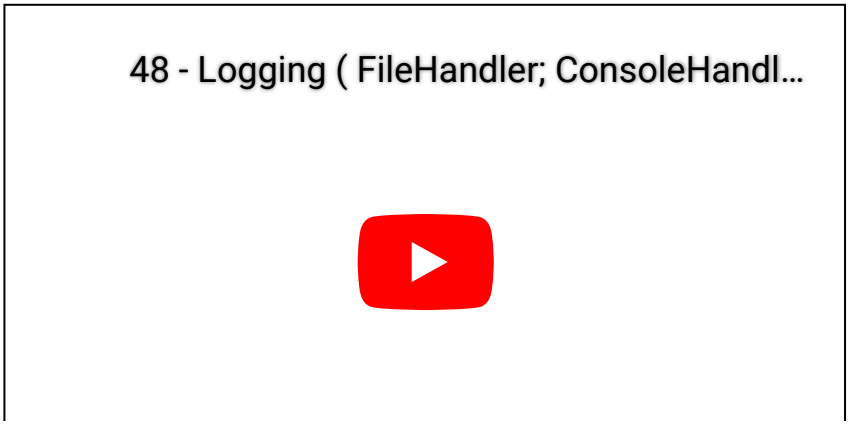
```
1  // log a message at INFO level
2  logger.log(Level.INFO, "going to start processing");
3  // ...
4  processInput();
5  if (error) {
6      // log a message at WARNING level
7      logger.log(Level.WARNING, "processing error", ex);
8  }
9  // ...
10 logger.log(Level.INFO, "end of processing");
```

Resources

Tutorials:

- **Java Logging API - Tutorial** -- A tutorial by *Lars Vogella*
- **Java Logging Tutorial** -- An alternative tutorial by *Jakob Jenkov*

- A video tutorial by *SimplyCoded*:



Best Practices:

- **10 Tips for Proper Application Logging** -- by *Tomasz Nurkiewicz*
- **What each logging level means** -- conventions recommended by *Apache Project*

Defensive programming

What

★★☆☆ Can explain defensive programming

A defensive programmer codes under the assumption "if you leave room for things to go wrong, they *will* go wrong". Therefore, a defensive programmer proactively tries to eliminate any room for things to go wrong.

Consider a method `MainApp#getConfig()` that returns a `Config` object containing configuration data. A typical implementation is given below:

```
class MainApp {
    Config config;

    /** Returns the config object */
    Config getConfig() {
        return config;
    }
}
```

If the returned `Config` object is not meant to be modified, a defensive programmer might use a more *defensive* implementation given below. This is more defensive because even if the returned `Config` object is modified (although it is not meant to be), it will not affect the `config` object inside the `MainApp` object.

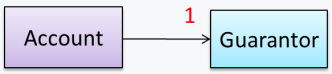
```
    /** Returns a copy of the config object */
    Config getConfig() {
        return config.copy(); // return a defensive copy
    }
```

Enforcing Compulsory Associations

★★★☆ Can use defensive coding to enforce compulsory associations

Consider two classes, `Account` and `Guarantor`, with an association as shown in the following diagram:

Example:



Here, the association is compulsory i.e. an `Account` object should always be linked to a `Guarantor`. One way to implement this is to simply use a reference variable, like this:

```
class Account {
    Guarantor guarantor;

    void setGuarantor(Guarantor g) {
        guarantor = g;
    }
}
```

However, what if someone else used the `Account` class like this?

```
Account a = new Account();
a.setGuarantor(null);
```

This results in an `Account` without a `Guarantor`! In a real banking system, this could have serious consequences! The code here did not try to prevent such a thing from happening. You can make the code more defensive by proactively enforcing the multiplicity constraint, like this:

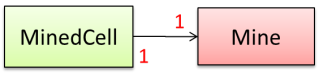
```
class Account {
    private Guarantor guarantor;

    public Account(Guarantor g) {
        if (g == null) {
            stopSystemWithMessage(
                    "multiplicity violated. Null Guarantor");
        }
        guarantor = g;
    }
    public void setGuarantor(Guarantor g) {
        if (g == null) {
            stopSystemWithMessage(
                    "multiplicity violated. Null Guarantor");
        }
        guarantor = g;
    }
    // ...
}
```

Exercises

Enforcing 1-to-1 Associations

★★★★ Can use defensive coding to enforce 1-to-1 associations

Consider the association given below. A defensive implementation requires us to ensure that a `MinedCell` cannot exist without a `Mine` and vice versa which requires simultaneous object creation. However, Java can only create one object at a time. Given below are two alternative implementations, both of which violate the multiplicity for a short period of time.



Option 1:

```
class MinedCell {
    private Mine mine;

    public MinedCell(Mine m) {
        if (m == null) {
            showError();
        }
        mine = m;
    }
    // ...
}
```

Option 1 forces us to keep a `Mine` without a `MinedCell` (until the `MinedCell` is created).

Option 2:

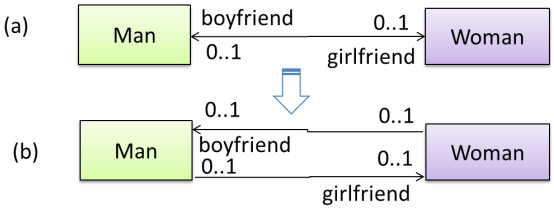
```
class MinedCell {
    private Mine mine;

    public MinedCell() {
        mine = new Mine();
    }
    // ...
}
```

Option 2 is more defensive because the `Mine` is immediately linked to a `MinedCell`.

Enforcing Referential Integrity

★★★★ Can use defensive coding to enforce referential integrity of bidirectional associations

A bidirectional association in the design (shown in (a)) is usually emulated at code level using two variables (as shown in (b)).




```
class Man {
    Woman girlfriend;

    void setGirlfriend(Woman w) {
        girlfriend = w;
    }
    // ...
}
```

```
class Woman {
    Man boyfriend;

    void setBoyfriend(Man m) {
        boyfriend = m;
    }
}
```

The two classes are meant to be used as follows:

```
Woman jean;
Man james;
// ...
james.setGirlfriend(jean);
jean.setBoyfriend(james);
```

Suppose the two classes were used like this instead:

```
Woman jean;
Man james, yong;
// ...
james.setGirlfriend(jean);
jean.setBoyfriend(yong);
```

Now James' girlfriend is Jean, while Jean's boyfriend is not James. This situation is a result of the code not being defensive enough to stop this "love triangle". In such a situation, you could say that *the referential integrity has been violated*. This means that *there is an inconsistency in object references*.

One way to prevent this situation is to implement the two classes as shown below. Note how the referential integrity is maintained.

```
public class Woman {
    private Man boyfriend;

    public void setBoyfriend(Man m) {
        if (boyfriend == m) {
            return;
        }
        if (boyfriend != null) {
            boyfriend.breakUp();
        }
        boyfriend = m;
        m.setGirlfriend(this);
    }

    public void breakUp() {
        boyfriend = null;
    }
    // ...
}
```

```
public class Man {
    private Woman girlfriend;

    public void setGirlfriend(Woman w) {
        if (girlfriend == w) {
            return;
        }
        if (girlfriend != null) {
            girlfriend.breakUp();
        }
        girlfriend = w;
        w.setBoyfriend(this);
    }
    public void breakUp() {
        girlfriend = null;
    }
    // ...
}
```

When `james.setGirlfriend(jean)` is executed, the code ensures that `james` breaks up with any current girlfriend before he accepts `jean` as his girlfriend. Furthermore, the code ensures that `jean` breaks up with any existing boyfriends before accepting `james` as her boyfriend.

Exercises

When

★★★☆ Can explain when to use defensive programming

It is not necessary to be 100% defensive all the time. While defensive code may be less prone to be misused or abused, such code can also be more complicated and slower to run.

The suitable degree of defensiveness depends on many factors such as:

- How critical is the system?
- Will the code be used by programmers other than the author?
- The level of programming language support for defensive programming
- The overhead of being defensive

Exercises

Design-by-contract approach

Design by Contract

★★★★ Can explain the Design-by-Contract approach

Design by contract (DbC) is an approach for designing software that requires defining formal, precise and verifiable interface specifications for software components.

Suppose an operation is implemented with the behavior specified precisely in the API (preconditions, post conditions, exceptions etc.). When following the defensive approach, the code should first check if the preconditions have been met. Typically, exceptions are thrown if preconditions are violated. In contrast, the *Design-by-Contract (DbC)* approach to coding assumes that it is the responsibility of the caller to ensure all preconditions are met. The operation will honor the contract only if the preconditions have been met. If any of them have not been met, the behavior of the operation is "unspecified".

Languages such as Eiffel have native support for DbC. For example, preconditions of an operation can be specified in Eiffel and the language runtime will check precondition violations without the need to do it explicitly in the code. To follow the DbC approach in languages such as Java and C++ where there is no built-in DbC support, assertions can be used to confirm pre-conditions.

Exercises