Exceptions

Checked vs Unchecked

Error checking in C

- Use of special return values
- Problems
  - Must remember specific failure values for many functions
  - A different implementation of the same function may choose to use a different failure value
  - If the failing function wishes to communicate useful information about why it had failed, one failure value is insufficient.

Semipredicate problem

- What would be the return value of a division by zero?
- It occurs when a function intended to return a useful value can fail, but the signalling of failure uses an otherwise valid return value.

Solutions

- Multivalued return (Python tuples)
- Global variable for return status (C errno)
- “Out” error arguments (Objective-C NSError)
- Expanding the return type
  - Nullable reference types
  - Implicitly hybrid types (PHP return "false", "none" or "null" when the function call fails)
  - Explicitly hybrid types (Haskell Either)
- Exceptions

What are exceptions

- Exception handling is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions
- An exception breaks the normal flow of execution and executes a pre-registered exception handler
- Breaks normal program flow

History

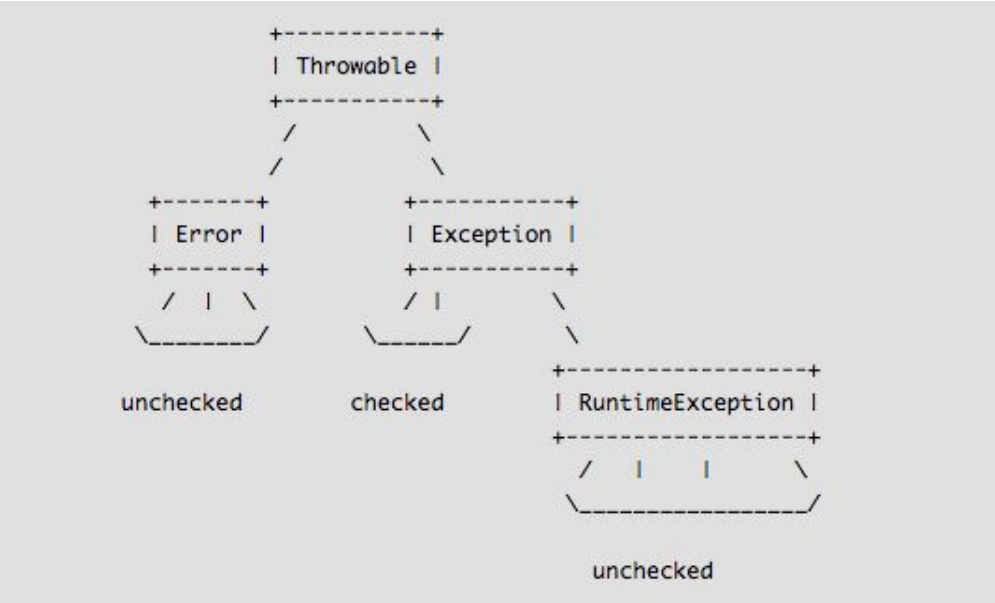
- Software exception handling developed in Lisp in the 1960s and 1970s.
- Contemporary languages can roughly be divided into two groups:
  - Languages where exceptions are designed to be used as flow control structures: Ada, Java, Modula-3, ML, OCaml, Python, and Ruby.
  - Languages where exceptions are only used to handle abnormal, unpredictable, erroneous situations: C++, C#, Common Lisp, Eiffel, and Modula-2.

Checked vs Unchecked exceptions

Checked are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

Exceptions in Java



If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

Why Kotlin chose unchecked exceptions

Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

```
Appendable append(CharSequence csq) throws IOException;
```

```
try {
    log.append(message)
}
catch (IOException e) {
    // Must be safe
}
```

How programmers handle checked exceptions

“In a well-written application there's a ratio of ten to one, in my opinion, of try finally to try catch. Or in C#, using statements, which are like try finally.

In the finally, you protect yourself against the exceptions, but you don't actually handle them. Error handling you put somewhere else.” Anders Hejlsberg

Versionability issue

“Let's say I create a method foo that declares it throws exceptions A, B, and C.

In version two of foo, I want to add a bunch of features, and now foo might throw exception D. It is a breaking change for me to add D to the throws clause of that method, because existing caller of that method will almost certainly not handle that exception.” Anders Hejlsberg

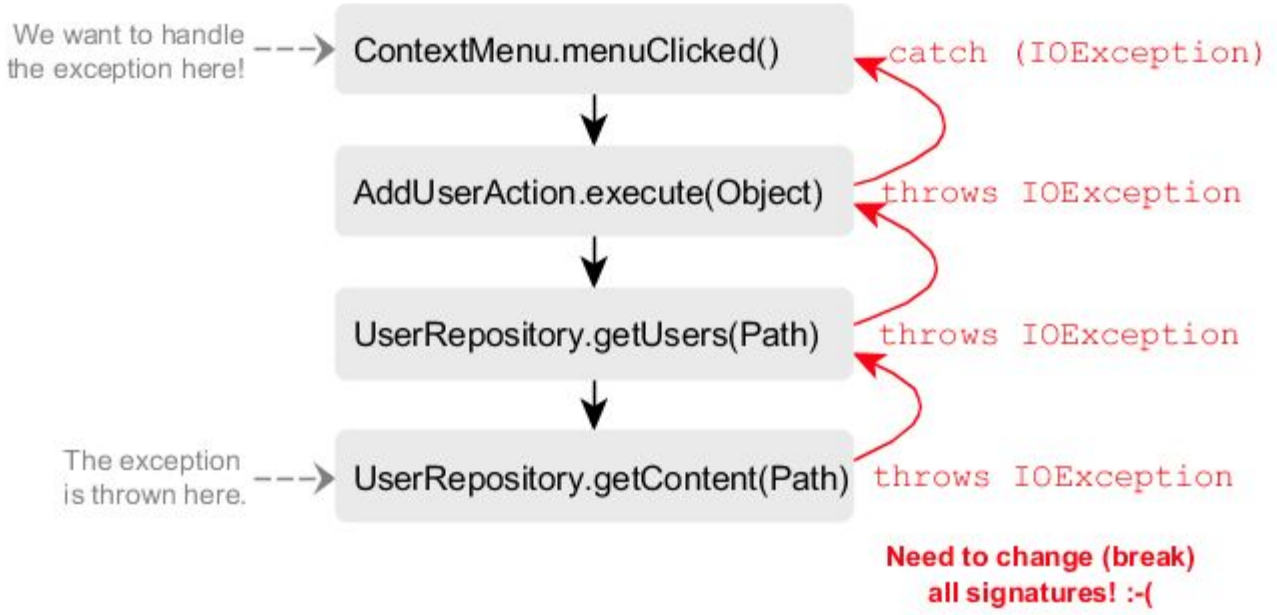
Scalability issue

“In the small, checked exceptions are very enticing. With a little example, you can show that you've actually checked that you caught the FileNotFoundException, and isn't that great? Well, that's fine when you're just calling one API.

The trouble begins when you start building big systems where you're talking to four or five different subsystems. Each subsystem throws four to ten exceptions. Now, each time you walk up the ladder of aggregation, you have this exponential hierarchy below you of exceptions you have to deal with. You end up having to declare 40 exceptions that you might throw. And once you aggregate that with another subsystem you've got 80 exceptions in your throws clause. It just balloons out of control.” Anders Hejlsberg

Checked exceptions are part of the signature

```
private String getContent(Path targetFile) throws IOException {
    byte[] content = Files.readAllBytes(targetFile);
    return new String(content);
}
```



Checked exceptions are part of the signature

If we throw an IOException in UserRepository.getContent(Path) and want to handle them in ContextMenu.menuClicked(), we have to change all method signatures up to this point.

What happens, if we later want to add a new exception, change the exception or remove them completely? Yes, we have to change all signatures. Hence, all clients using our methods will break.

Moreover, if you use an interface of a library (e.g. Action.execute(), Java 8 Stream API) you are not able to change the signature at all.

Exposing Implementation Details in the Signature

```
UserRepository.getUsers(Path) throws IOException
```

Assume that UserRepository is an interface. Will the various implementations of UserRepository throw an IOException like our file system based implementation? Not at all. A database implementation might throw a SqlException or a WebService might throw a TimeoutException. The point is that the type of exceptions (and whether they are thrown at all) is specific to the implementation.

Moreover, the client doesn’t care if an IOException or a SqlException is thrown. He just wants to know, if something went wrong. Consequently the thrown exception should be implementation agnostic.

No appropriate handling possible

```
private String getContent(Path targetFile) {
  try {
    byte[] content = Files.readAllBytes(targetFile);
    return new String(content);
  } catch (IOException e) {
    //What to do?
  }
}
```

What to do with the IOException in this low-level method?

We can’t appropriately handle them here. Maybe it could be handled at a higher level by giving the user a feedback in the UI, but that’s nothing we should do within this method. Nevertheless, the compiler enforces us to handle it here.

What you lose with unchecked exceptions

Cannot communicate to the API what exceptions to expect

How to overcome this?

In Kotlin, you could using Throws annotation

```
@Throws(IOException::class)
fun readFile(name: String): String {...}
```

In OCaml, exception checkers exist. The external tool for OCaml is both invisible (i.e. it does not require any syntactic annotations) and optional (i.e. it is possible to compile and run a program without having checked the exceptions, although this is not recommended for production code).

Kotlin exceptions

```
throw Exception("Hi There!")
```

```
try {
    // some code
}
catch (e: SomeException) {
    // handler
}
finally {
    // optional finally block
}
```

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

The Nothing type

```
val s = person.name ?: throw IllegalArgumentException("Name required")


fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}


val s = person.name ?: fail("Name required")
println(s)     // 's' is known to be initialized at this point
```

References

- https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
- https://blog.philipphauer.de/checked-exceptions-are-evil/
- https://kotlinlang.org/docs/reference/exceptions.html