

## Exception specifications and class derivation

Virtual overrides' exception specifications must be at least as restrictive as their bases'.

```
class Base {
public:
    virtual void f(); // can throw anything
    virtual void g() throw( X, Y );
    virtual void h() throw( X );
};

class Derived : public Base {
    void f() throw( X ); // OK
    void g() throw( X ); // OK
    void h() throw( X, Y ); // error
};
```

Otherwise, a caller of `Base::h()` would be unprepared to handle a `Y` thrown by `Derived::h()`.

For the same reason,

```
void f() throw( X );
void (*pf1)() throw( X, Y ) = &f; // OK
void (*pf2)() throw( ) = &f; // error
```

## Error Handling in Java

Garbage collector  $\Rightarrow$  no destructors

Memory resources are released automatically; other resources can be released in a `finally` clause.

If C++ had `finally` blocks:

```
void useFile( const char* fn ) {
    FILE* f = fopen( fn, "w" );
    try {
        // use f
    }
    finally {
        fclose( f );
    }
}
```

A `finally` block is *always* executed, regardless of whether an exception is thrown.

It is executed *after* any of its `try` block's exception handlers:

```
class Switch {
    boolean state = false;
    boolean read() { return state; }
    void on() { state = true; }
    void off() { state = false; }
}

// ...

static Switch sw = new Switch();
```

```
public static void main( ... ) {
    try {
        sw.on();
        // Code that can throw exceptions...
    }
    catch( NullPointerException e ) {
        // ...handle it ...
    }
    catch( IllegalArgumentException e ) {
        // ...handle it ...
    }
    finally {
        sw.off();
    }
}
```

In Java, all classes are derived from `Object`; all exception classes are derived from `Throwable`.

The Java counterpart of C++'s

```
catch ( ... ) { /* ... */ }
is
catch ( Throwable ) { /* ... */ }
```

with the advantage that `Throwable` has a few methods

```
catch ( Throwable e ) {
    ... e.getMessage() ... e.toString() ...
}
```

Classes derived from `Throwable` include

- `Exception`
- `RuntimeException`
- `Error`

In Java, functions' exception specifications

```
void f() throws tooBig, tooSmall { // ...
```

are checked by the compiler (in C++ they are checked only at run time).

If `f`—or any function called by `f`—throws any exception (other than `tooBig` and `tooSmall`) whose type is `Exception` or any of its subtypes, this causes a compilation error.

`Exception` and its subtypes—which include the vast majority of exception types—are known as *checked* exception types.

Hence a function that lacks an exception specification is guaranteed to throw *no checked exceptions*:

```
void g() { // ...
    indicates that g
    • throws no checked exceptions itself
    • handles every checked exception thrown by the
      functions it calls
```

The unchecked exception types—`RuntimeException`, `Error`, and their derivatives—are not checked by the compiler:

- `RuntimeException`—includes `NullPointerException`, `ArrayIndexOutOfBoundsException`, and others resulting from programming errors
- `Error`—indicates a system error

## Error Handling in Haskell

In programming languages that lack the notion of “flow of control”, the try-throw-catch method of exception handling is not a good fit.

In Haskell, errors occur when a *partial function* —i.e., a function which is undefined for certain arguments—causes a computation to halt with an error.

Example: Applying the function

```
> tail :: [a] -> [a]
> tail (_:xs) = xs
```

to an empty list [] halts the entire program:

```
Main> tail (tail [2])
Program error: {tail []}
```

What’s needed is a functional counterpart of exception handling— i.e., a way to enable computations to handle errors and continue.

This means transforming partial functions like tail into *total* functions.

One technique: Replace each partial function with one that returns a list.

```
> tail' :: [a] -> [a]
> tail' (_:xs) = xs
> tail' [] = []
```

Problem: The “error” case is indistinguishable from certain non-error cases:

```
> tail' [3] ~> []
> tail' [] ~> []
```

Another problem: For functions that don’t return lists, this technique doesn’t work at all:

```
> head' :: [a] -> a
> head' (x:_) = x
> head' [] = ???
```

We could change the functions’ types:

```
> tail'' :: [a] -> [[a]]
> tail'' (_:xs) = [xs]
> tail'' [] = []

> head'' :: [a] -> [a]
> head'' (x:_) = [x]
> head'' [] = []
```

This works, but —unlike exceptions in C++/Java— it doesn’t clearly distinguish the normal-case code from the error-handling code.

Error “results” can be distinguished from valid results by defining a polymorphic algebraic type:

```
> data Maybe a = Nothing | Just a
> deriving (Eq, Ord, Read, Show)
```

Maybe a is like a type of “boxes” for values of type a, with a label on the outside of each box indicating whether it contains a good value.

Functions that are not *total* —i.e., functions which, applied to certain arguments, halt the computation— can be redefined to return boxed values.

Instead of a *partial* function whose type is

```
a -> b
```

we define a *total* function of type

```
a -> Maybe b
```

Following this approach with head and tail, we write

```
> mTail :: [a] -> Maybe [a]
> mTail (x:xs) = Just xs
> mTail [] = Nothing -- exception value

> mHead :: [a] -> Maybe a
> mHead (x:_) = Just x
> mHead [] = Nothing
```

This technique works for division by 0:

Instead of

```
> div :: Int -> Int -> Int
```

which does not survive 0 divisors:

```
Prelude> 3 `div` 0
Program error: {primQmInteger 3 0}
```

or

```
> div' :: Int -> Int -> Int
> div' a b
> | b /= 0 = a `div` b
> | otherwise = 0
```

which conceals 0 divisors, we write

```
> mDiv :: Int -> Int -> Maybe Int
> mDiv a b
> | b /= 0 = Just (a `div` b)
> | otherwise = Nothing
```

These functions —mTail, mHead, and mDiv— are said to *raise* exceptions when they can’t produce a normal result.

## Using Maybe

A function that raises exceptions:

try p s applies parser p to string s.

If p succeeds, delivering an item r and consuming all of s, then try p s returns Just r; otherwise, try p s returns Nothing.

```
> try :: Parser a -> String -> Maybe a
> try p s
> = case [ r | (r, "") <- papply p s ] of
>   (r:_) -> Just r
>   [] -> Nothing
```

A function that tests for exceptions:

get prompt p outputs prompt, then inputs a line and applies parser p to it.

If the parser recognizes the line, get returns the recognized item; otherwise, it repeats.

```
> get :: String -> Parser a -> IO a
> get prompt p
> = do putStr prompt
>     line <- getLine
>     case try p line of
>       Just r -> return r
>       Nothing -> do putStrLn " ... syntax error"
>                    get prompt p
```