Hmm; I would probably say something like:
Exceptions in F# provide a mechanism for managing exceptional control flow and is often used for signaling runtime errors, such as division by zero, and for handling such errors, when they occur in a larger application.

18 Handling Errors and Exceptions

18.1 Exceptions

~~Exceptions are runtime errors, such as division by zero.~~ E.g., attempting integer division by zero halts execution and a long ~~a~~ somewhat cryptic error message is written to the screen as illustrated in Listing 18.1.

Listing 18.1: Division by zero halts execution with an error message.

```
> 3 / 0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0002>.$FSI_0002.main@ () [0x00001] in
   <0e5b9fd12a6649c598d7fa8c09a58dd3>:0
  at (wrapper managed-to-native)
   System.Reflection.MonoMethod:InternalInvoke
   (System.Reflection.MonoMethod,object,object[],System.Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj,
   System.Reflection.BindingFlags invokeAttr, System.Reflection.Binder
   binder, System.Object[] parameters, System.Globalization.CultureInfo
   culture) [0x00032] in <c9f8153c41de4f8cbafd0e32f9bf6b28>:0
Stopped due to error
```

The error message starts by `System.DivideByZeroException: Attempted to divide by zero`, followed by a description of which libraries were involved when the error occurred, and finally fsharp informs us that it `Stopped due to error`. The type `System.DivideByZeroException` is a built-in exception type, and the built-in integer division operator ~~chooses~~ is defined to raise the exception, when the undefined division by zero is attempted. Many times such errors can be avoided by clever program design. However, this is not always possible or desirable, which is why F# implements exception handling for graceful control.

Exceptions ~~are a~~ are special values of the basic-type ~~called~~ *exn*, and F# has a number of built-in, a few of which are listed in Table 18.1.

- exn

An exception value can be raised, *

Exceptions are handled by the `try`-keyword expressions. ~~We say that an expression may *raise or cast* an exception, the try-expression may catch and *handle* the exception by another expression.~~

- raising exception
- casting exceptions
- catching exception
- handling exception

Exceptions like in Listing 18.1 may be handled by `try-with` expressions as demonstrated in Listing 18.2.

even hard to distinguish between exceptions and an exception value being raised!

*1 If no try-expression surrounds the expression raising the exception, the top-level handler will print an appropriate error message and abort the program execution.

* which has the effect that the expression being evaluated is aborted and the call-stack cut to the level of a potential try-expression that can be used to ~~intercept~~ handle the ~~ex~~ raised exception. *1

Listing 18.8 exceptionDivByZeroFinally.fsx:
The branch is executed regardless of an exception.

```
let div enum denom =
  printf "Doing division:"
  try
    printf " %d %d." enum denom
    enum / denom
  finally
    printfn " Division finished."

printfn "3 / 1 = %d" (try div 3 1 with ex -> 0)
printfn "3 / 0 = %d" (try div 3 0 with ex -> 0)
```

```
$ fsharpc --nologo exceptionDivByZeroFinally.fsx
$ mono exceptionDivByZeroFinally.exe
Doing division: 3 1. Division finished.
3 / 1 = 3
Doing division: 3 0. Division finished.
3 / 0 = 0
```

Here, the `finally` branch is evaluated following the evaluation of the test expression regardless of whether the test expression raises an exception or not. However, if an exception is raised in a `try-finally` expression and there is no outer `try-with` expression, then execution stops without having evaluated the `finally` branch.

Exceptions can be raised using the *raise*-function

Listing 18.9 Syntax for the raise function that raises exceptions.

```
raise (<expr>)
```

An example of raising the `System.ArgumentException` is shown in Listing 18.10

Listing 18.10 raiseArgumentException.fsx:
Raising the division by zero with customized message.

```
let div enum denom =
  if denom = 0 then
    raise (System.ArgumentException "Error: \"division by 0\"")
  else
    enum / denom

printfn "3 / 0 = %s" (try (div 3 0 |> string) with ex -> ex.Message)
```

```
$ fsharpc --nologo raiseArgumentException.fsx
$ mono raiseArgumentException.exe
3 / 0 = Error: "division by 0"
```

In this example, division by zero is never attempted. Instead an exception is raised, which must be handled by the caller. Note that the type of `div` is `int -> int -> int` because `denom` is compared with

Not really: raise has type
$\forall \alpha.\ exn \to \alpha$

an integer in the conditional statement. This contradicts the typical requirements for `if` statements, where every branch has to return the same type. However, any code that explicitly raises exceptions are ignored, and the type is inferred by the remaining branches.

Programs may define new exceptions using the syntax,

Listing 18.11 Syntax for defining new exceptions.

```
exception <ident> of <typeId> {* <typeId>}
```

An example of defining a new exception and raising it is given in Listing 18.12.

Listing 18.12 exceptionDefinition.fsx:
A user-defined exception is raised but not caught by outer construct.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

printfn "picky %A = %A" 3 (try picky 3 |> string with ex -> ex.Message)
printfn "picky %A = %A" 5 (try picky 5 |> string with ex -> ex.Message)
```

```
$ fsharpc --nologo exceptionDefinition.fsx
$ mono exceptionDefinition.exe
picky 3 = "3"
picky 5 = "Exception of type 'ExceptionDefinition+DontLikeFive' was
   thrown."
```

Here an exception called `DontLikeFive` is defined, and it is raised in the function `picky`. The example demonstrates that catching the exception as a `System.Exception` as in Listing 18.5 the `Message` property includes information about the exception name but not its argument. To retrieve the argument `"5 sucks"`, we must match the exception with correct exception name as demonstrated in Listing 18.13.

Listing 18.20 exceptionReraise.fsx: Reraising an exception.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
      reraise()
```

```
$ fsharpc --nologo exceptionReraise.fsx && mono exceptionReraise.exe
The castle of "Arrrrg"

Unhandled Exception:
System.Exception: Arrrrg
  at <StartupCode$exceptionReraise>.$ExceptionReraise$fsx.main@ ()
   [0x00041] in <599574d491e0c9eea7450383d4749559>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.Exception: Arrrrg
  at <StartupCode$exceptionReraise>.$ExceptionReraise$fsx.main@ ()
   [0x00041] in <599574d491e0c9eea7450383d4749559>:0
```

The `reraise` function is only allowed to be the final call in the expression of a `with` rule.

18.2 Option types

would be the morally

?

At exceptions, it is not always obvious what should be returned. E.g., in the Listing 18.2, the exception is handled gracefully, but the return value is somewhat arbitrarily chosen to be the largest possible integer, which is still far from infinity, which ~~is the~~ correct result. Instead we could use the *option type*. The option type is a wrapper, that can be put around any type, and which extends the type with the special value ***None***. All other values are preceded by the ***Some*** identifier. An example of rewriting Listing 18.2 to correctly represent the non-computable value is shown in Listing 18.21.

- option type
- `None`
- `Some`

Listing 18.21: Option types can be used, when the value in case of exceptions is unclear.

```
> let div enum denom =
-   try
-     Some (enum / denom)
-   with
-     | :? System.DivideByZeroException -> None;;
val div : enum:int -> denom:int -> int option

>
- let a = div 3 1;;
val a : int option = Some 3

> let b = div 3 0;;
val b : int option = None
```

The value of an option type can be extracted by and tested for by its member function, ***IsNone***, ***IsSome***, and ***Value*** as illustrated in Listing 18.22.

- `IsNone`
- `IsSome`
- `Value`