Обработка ошибок

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2020

Содержание лекции

Обработка ошибок

Процесс обработки ошибки можно представить в виде последовательности, состоящей из трех действий:

1. Выявление ошибочной ситуации. Этап связан с особенностями предметной области и вытекающими из этих особенностей требованиям к данным и организации вычислительного процесса.
2. Генерация информации о возникновении ошибочной ситуации.
3. Обработка ошибки (определение действий в связи с выявленной ситуацией).

Основная сложность **не в том, как обнаружить ошибку**, а в том:

- как сообщить об этой ошибке отдельным модулям проекта;
- как определить набор действий, обрабатывающих ошибочную ситуацию в том случае, когда эта обработка не может быть осуществлена в пределах одной функции и даже в пределах отдельного модуля.

Обработка ошибок

Варианты обработки ошибок:

1. Обработка ошибки на месте.
2. Использование возвращаемых значений.
3. Использование глобальных переменных или переменных-членов класса, исполняющих роль индикаторов состояния приложения или объекта.
4. Использование специально разработанных функций.
5. Использование функций обратного вызова (*callback functions*).
6. Использование генерации и обработки исключений.

Обработка ошибок на месте (Pascal)

```
//Записать элемент
procedure QueueInt.Enqueue(Elem: integer);
begin
  if length = size then begin
            write('Enqueue error. Queue is full.');
    exit;
  end;

  last := (last+1) mod size;
  inc(length);
  body[last] := Elem;
end;
```

Обработка ошибок на месте

Недостатки:

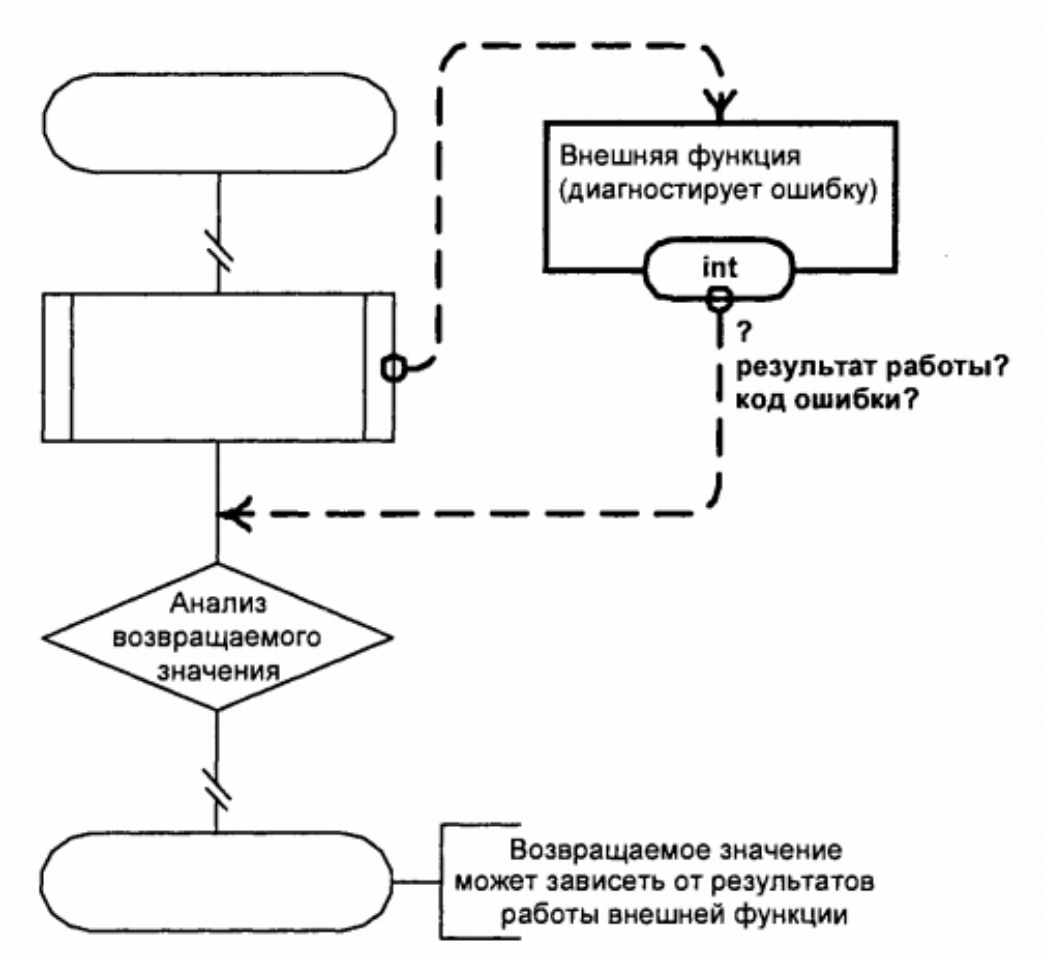
- находясь в пределах области видимости функции, где была диагностирована ошибочная ситуация, мы можем не иметь доступа к данным и функциям, необходимым для корректной и полной обработки возникшей ситуации.
- трудность реализации обработки ошибок разного типа (например, ошибок, требующих или не требующих остановки выполнения программы).

Обработка ошибок на месте затрудняет разработку унифицированных компонентов, которые могли бы использоваться в различных приложениях.

```
//Использование возвращаемого значения
const READ_OK = 0;
      ERROR_UNEXP_EOF = 1;
      ERROR_BAD_FORMAT = 2;
procedure ReadData(var F: Text; var X: integer;
  var ErrorCode: integer);
begin
  if EOF(F) then begin
    ErrorCode := ERROR_UNEXP_EOF;
    exit;
  end;
  {$I-} read(F, X);  {$I+}
  if IOResult <> 0 then begin
    ErrorCode := ERROR_BAD_FORMAT;
    exit;
  end;
  ErrorCode := READ_OK;
end;
```

```
ReadData(InputFile, InputData, ErrorCode);
case ErrorCode of
  ERROR_UNEXP_EOF: //обработка ошибки - EOF
    begin
      writeln('Error - end of file ', FileName);
      halt(2);
    end;
  ERROR_BAD_FORMAT: //обработка ошибки - неверные данные
    begin
      writeln('Error - bad format in ', FileName);
      halt(3);
    end;
end;
writeln('X = ', InputData);
writeln('Terminated succesfully');
```

Возвращаемые значения как инструмент обработки ошибок



Возвращаемые значения как инструмент обработки ошибок

Преимущества:

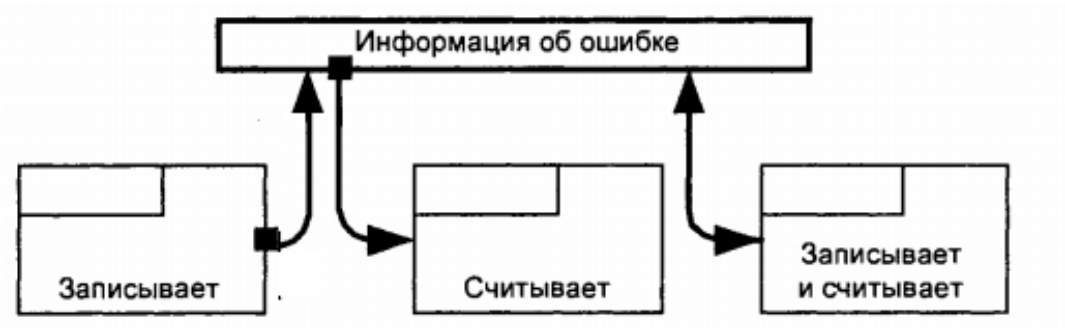
- использование возвращаемого значения позволяет отделить реализацию выявления ошибки от обработчика ошибки;

Недостатки:

- усложняется интерфейс функций, в которых диагностируется ошибка;
- некоторые функции могут использовать возвращаемое значение для реализации элементов решения, не связанных с обработкой ошибок;
- некоторые функции могут заканчиваться с разными кодами ошибок;
- в ОО-программах невозможно сообщить об ошибке в ходе выполнения конструктора или деструктора, которые вообще не имеет возвращаемого значения.

Использование глобальных переменных

- функциональные модули взаимодействуют посредством использования разделяемого ресурса;
- в случае обнаружения ошибочной ситуации данной переменной присваивается определенное кодовое значение, которое может быть проанализировано за пределами функции, диагностировавшей ошибку;
- вместо возвращаемого значения используется обособленный интерфейс для взаимодействия функций в связи с обработкой ошибок.



```
const
  ErrorCode: integer = NONE;  //код ошибки, начальное состояние

//функция чтения из текстового файла
procedure ReadData(var F: Text; var X: integer);
begin
  if EOF(F) then begin
    ErrorCode := ERROR_UNEXP_EOF;
    exit;
  end;
  {$I-} read(F, X); {$I+}
  if IOResult <> 0 then begin
    ErrorCode := ERROR_BAD_FORMAT;
    exit;
  end;
  ErrorCode := READ_OK;
end;
```

```
ReadData(InputFile, InputData);
case ErrorCode of
  ERROR_UNEXP_EOF: //обработка ошибки - EOF
    begin
      writeln('Error  - end of file ', FileName);
      halt(2);
    end;
  ERROR_BAD_FORMAT: //обработка ошибки - неверные данные
    begin
      writeln('Error  - bad format in', FileName);
      halt(3);
    end;
end;
ErrorCode := PASSED;
```

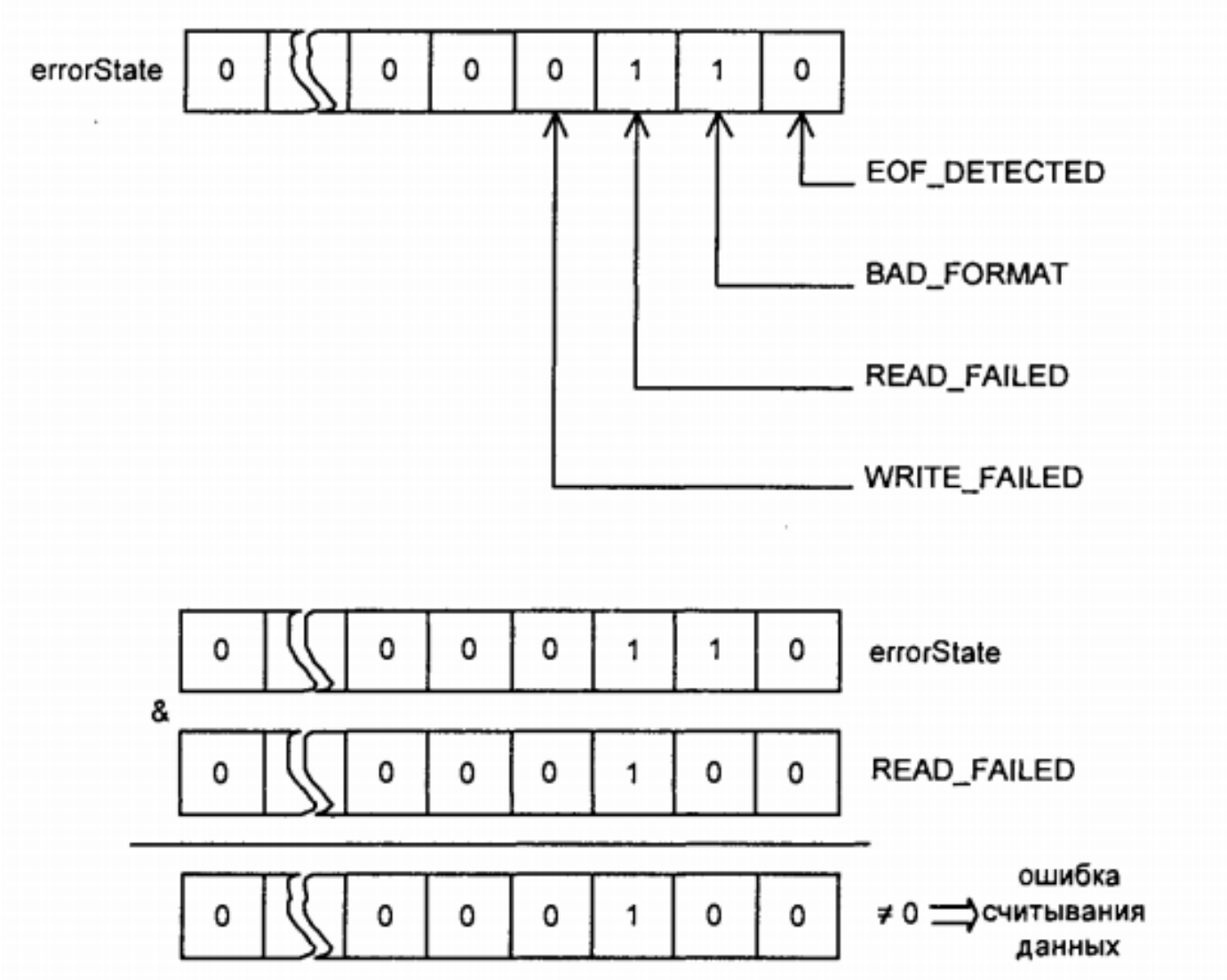
```
const READ_OK = 0;
      ERROR_UNEXP_EOF = 1;
      ERROR_BAD_FORMAT = 2;
      ERROR_READ_FAILED = 4;
      ERROR_WRITE_FAILED = 8;
  ErrorState: integer = 0;
```

Установка кода ошибки чтения в связи с некорректным форматом данных:

```
  if IOResult <> 0 then begin
    ErrorCode := ERROR_BAD_FORMAT or ERROR_READ_FAILED;
    exit;
  end;
```

Функция проверки ошибки чтения:

```
function IsBadFormat(): boolean;
begin
  Result := ErrorState and ERROR_BAD_FORMAT;
end;
```


errorState
0 0 0 0 1 1 0
EOF_DETECTED
BAD_FORMAT
READ_FAILED
WRITE_FAILED
0 0 0 0 1 1 0
errorState
&
0 0 0 0 1 0 0
READ_FAILED
0 0 0 0 1 0 0
≠ 0 ⇒ ошибка считывания данных

Использование специально разработанных функций

При возникновении ошибочной ситуации функция, диагностировавшая эту ситуацию, просто вызывает подходящую функцию-обработчик, передавая ей код ошибки.

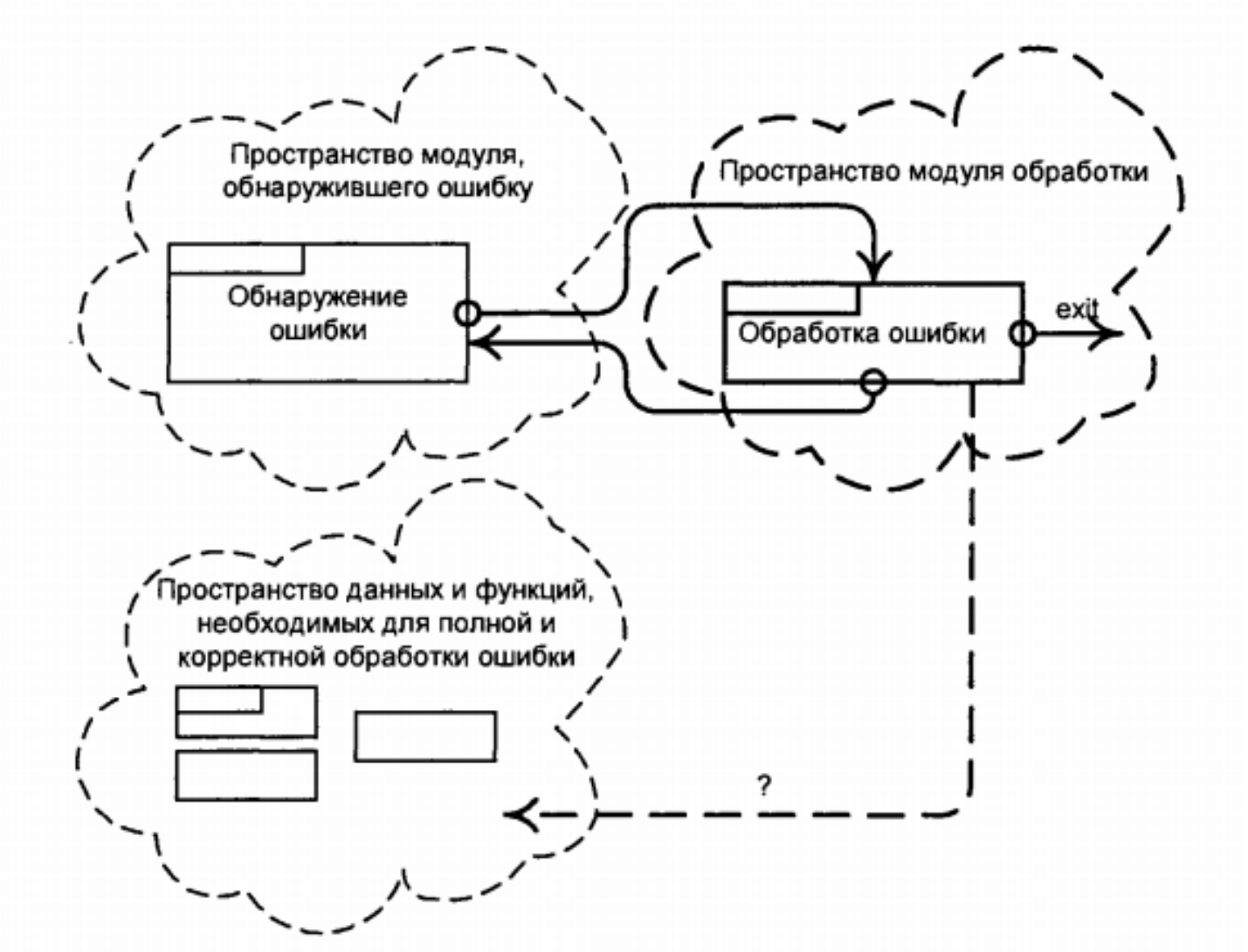
Преимущества:

- коды ошибок обычно глобальны на уровне проекта и должны тщательно документироваться.
- обычно реализуется отдельный модуль обработчика ошибок;

Недостатки:

- не очень подходит для реализации многоуровневых систем обработки ошибок;
- подход не всегда может обеспечить универсальный характер обработки ошибочных ситуаций.

Использование специально разработанных функций



Использование функции обратного вызова

Пользователь функции-обработчика ошибок в этом случае должен сначала зарегистрировать функцию обратного вызова, определенную в том модуле, в котором имеется доступ к данным, необходимым для корректной и полной обработки возникшей ситуации.

```
type
  TCallbackProc = procedure;
  TErrors = array[0..4] of string;
const
  ERROR_MSG: TErrors = (
    '',
    'Error while opening input file',
    'Error while opening output file',
    'Input file contains no data',
    'Wrong input data format');
```

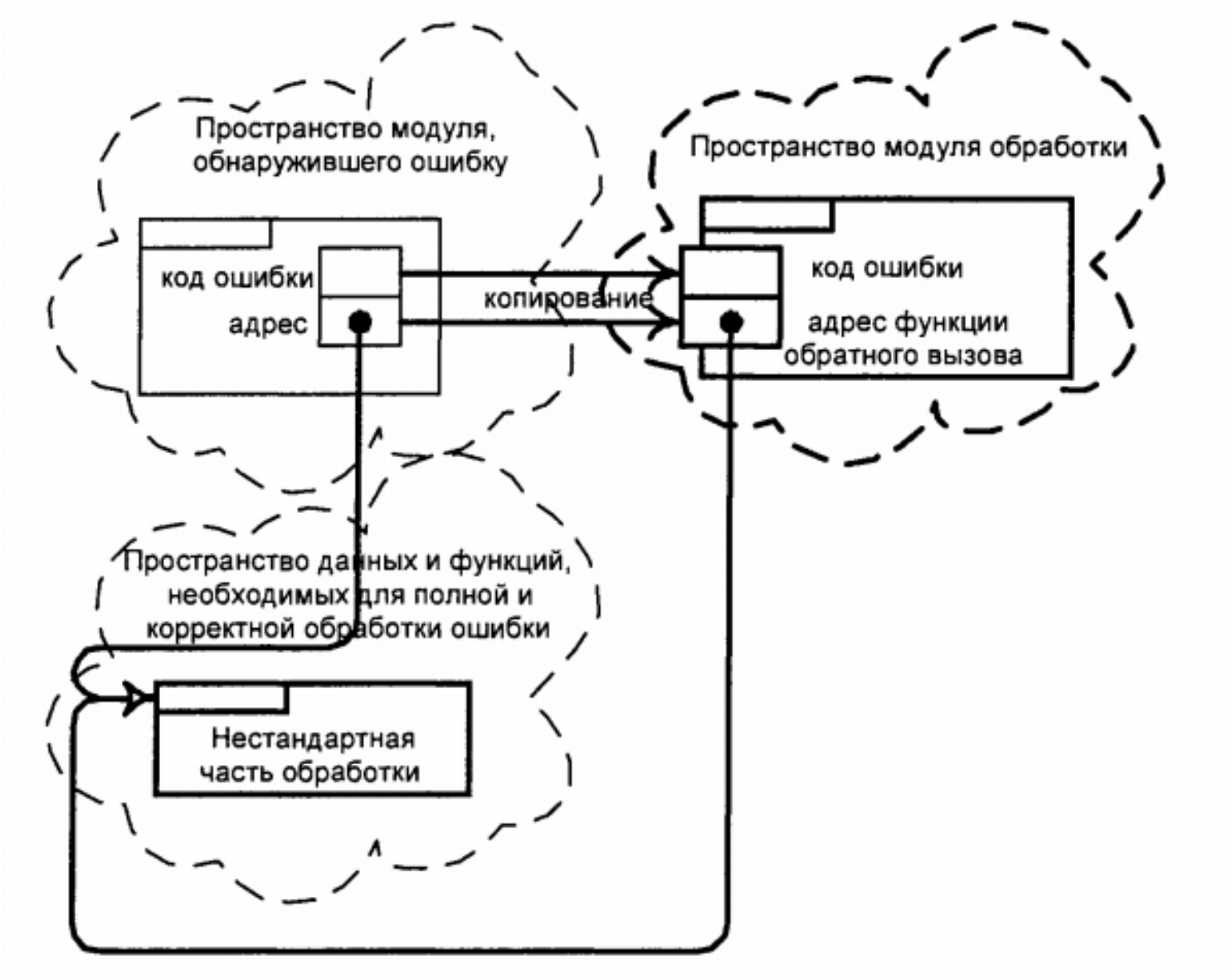
Использование функции обратного вызова

```
procedure Error(ErrorCode: integer;
  CallbackProc: TCallbackProc);
begin
  writeln('Error No', ErrorCode, ' ', ERROR_MSG[ErrorCode]);
  if CallbackProc = nil then
    halt(0)
  else
    CallbackProc();   //Вызов внешнего обработчика
end;
procedure Terminatel(); //Реализация первого варианта завершени
begin
  writeln('Unexpected end of input file');
  writeln('Terminated with error');
  Close(InputFile);
  halt(1);
end;
```

Использование функции обратного вызова

```
if IOResult <> 0 then
  Error(2, @Terminate0);

ReadData(InputFile, InputData);

if IsEOF then
  Error(3, @Terminate1)
else if IsBadFormat() then
  Error(4, @Terminate2);

writeln('X = ', InputData);
writeln('Terminated succesfully');

Close(InputFile);
Error(0, nil);
```

Использование функций обратного вызова



Использование функций обратного вызова

Преимущества:

- механизм с использованием функций обратного вызова частично разрешает проблему «разграничения полномочий» при обработке ошибок;

Недостатки:

- сигнатура функций обратного вызова должна в точности соответствовать сигнатуре, объявленной в модуле обработки ошибок;
- механизм не может считаться полноценной реализацией многоуровневой модели.

Исключения

Исключительная ситуация

возникает при ошибке или прерывании нормального хода выполнения программы каким-либо событием.

- исключение передает контроль выполнения программы обработчику исключительной ситуации, который позволяет отделить нормальную логику работы программы от обработки ошибок.
- поскольку исключения являются объектами, они могут быть сгруппированы в иерархию, использующую наследование, а новые исключения могут объявляться без изменения уже готового кода.
- Исключение может передавать информацию (например, сообщение об ошибке) из точки возникновения исключительной ситуации к месту ее обработки.

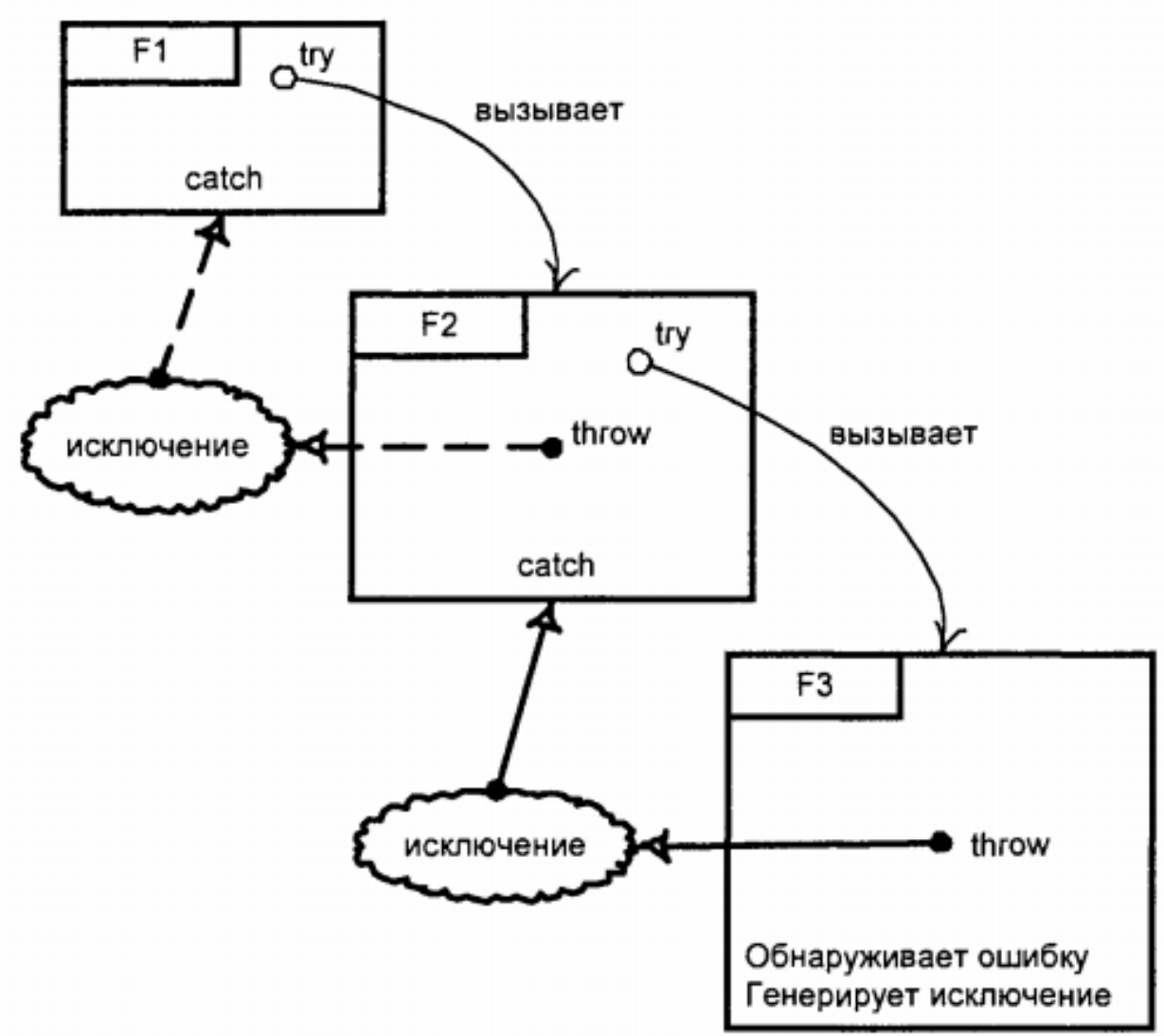
Исключения

Механизм обработки ошибок - механизм, встроенный в язык программирования. Программисту предоставляются:

- встроенные типы исключений
  - исключения, вызываемые переполнением при выполнении арифметических операций,
  - исключения, вызываемые выходом индекса массива за пределы корректного диапазона) и т.д.
- возможности для определения собственных исключительных ситуаций.

Впервые поддержка обработки исключений средствами языка была реализована в языке PL/1.

Основная идея подхода



Основная идея подхода

- Пусть функция *F2*() в иерархии вызовов передает управление другой функции *F3*().
- В ходе работы функции *F3*() выявляется некоторая ошибочная ситуация (т. е. функция *F3*() обнаруживает ошибку).
- Функция, обнаружившая ошибку, генерирует (*throws*, *raise*) специальный объект, описывающий возникшую исключительную ситуацию (*F3* создает объект-исключение).
- Выполнение функции, обнаружившей ошибку, прекращается, и управление передается в вызвавшую ее функцию (*F2*) для реализации действий по преодолению возникшей проблемы (*F2* перехватывает (*catches*) исключение.
- Обработкой исключения может заниматься как функция, непосредственно вызвавшая функцию-генератор исключения, так и предшествующие в иерархии вызовов функции (например, *F1*()).

Исключение

Генерация исключения представляет собой создание объекта-исключения. Объект характеризуется типом, например:

```
{$mode objfpc} //разрешаем использовать объекты Free Pascal
uses
  SysUtils;                    //подключаем модуль, где содержат
                  // описания исключений
type
  //описываем свой класс для обработки исключений
  EMyException = class(Exception)
    public
      constructor Create();
  end;
```

Исключение

Для инициирования исключения вам необходимо использовать экземпляр класса исключения с инструкцией *raise*:

```
raise ExceptionClass.Create;
```

Синтаксическая форма инструкция инициирования исключения в общем виде имеет следующий вид:

```
raise [ExceptionObject] [at Address]
```

- ExceptionObject - экземпляр класса исключения;
- Address - выражение, которое можно обработать как указатель.

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

Блок, обнаруживающий ошибку, инициирует процесс создания объекта исключения:

```
procedure F3();
begin
  if {* ошибка... *} then
    raise EMyException.Create(); // Генерирует исключение
  // Действия при отсутствии ошибок
end
```

Вызывающая подпрограмма может перехватить исключение и выполнить обработку ошибки:

```
procedure F2();
begin
  try
    F3(); // Может генерировать исключение
  except
    on EMyException do ...    // Обработка ошибки
  end;
end;
```

Синтаксис инструкции try...except:

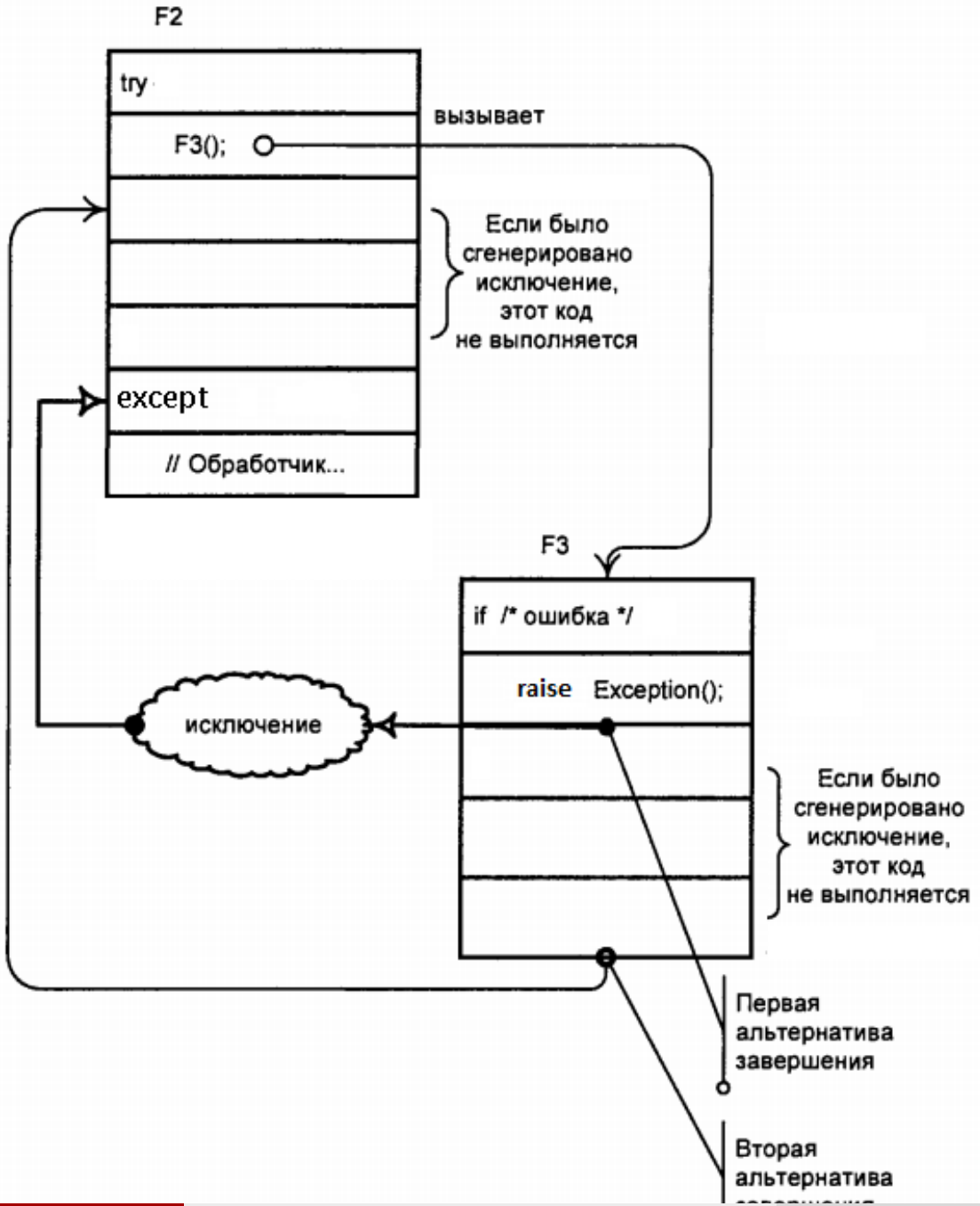
```
try
  Statements
except
  ExceptionBlock
end
```

- Statements - группа операторов;
- ExceptionBlock - блок обработки исключения
  - может быть еще одной последовательностью инструкций;
  - может быть последовательностью обработчиков исключений, разделяемых инструкциями else.

Обработчик исключения имеет вид:

```
on Identifier: IdentifierType do Statement
```

- identifier: необязателен (если присутствует – может быть любым допустимым идентификатором),
- IdentifierType – тип для представления исключений,
- statement – оператор языка.


F2
try
F3();
вызывает
Если было сгенерировано исключение, этот код не выполняется
except
// Обработчик...
F3
if /* ошибка */
raise Exception();
исключение
Если было сгенерировано исключение, этот код не выполняется
Первая альтернатива завершения
Вторая альтернатива

1. Инструкция try...except выполняет команды из изначального списка. Если исключительных ситуаций не возникло, блок исключений игнорируется и управление передается следующей части программы.
2. Если в процессе выполнения списка команд возникла исключительная ситуация (она может быть инициирована инструкцией raise в списке команд или внутри процедуры или функции, вызываемой из этого списка) предпринимается попытка обработки исключительной ситуации:
3. Если какой-либо из обработчиков в *exception block* подходит для обработки инициированного исключения, управление передается первому такому обработчику. Обработчик подходит для исключения только в том случае, когда тип, указанный в нем, является типом исключения или предком его класса.
4. Если обработчика не найдено, управление передается инструкции в секции *else* (если таковая присутствует).

5. Если блок обработчиков – это просто последовательность инструкций без каких либо обработчиков – управление передается первой инструкции в этой последовательности.
6. Если ни одно из указанных выше условий не выполнено, поиск продолжается в блоке обработчиков внешней инструкции *try...except*, выход из которой еще не выполнен. Если и там не находится соответствующего обработчика, секции *else* или последовательности инструкций, поиск продолжается в следующей внешней инструкции и так далее.
7. Если в самой внешней инструкции исключительная ситуация не будет обработана – приложение завершается.

В следующем примере первый обработчик обрабатывает исключение деления на ноль, второй – ошибки переполнения, а последний – остальные математические исключения. SysUtils.EMathError указан последним в блоке обработчиков, так как он является предком обоих этих классов. Если бы он указан первым, последующие обработчики никогда не были бы вызваны:

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

В обработчике исключения перед именем класса исключения можно указать идентификатор. Таким образом объявляется идентификатор для представления объекта исключения в процессе выполнения инструкции, следующей за on...do. Видимость идентификатора ограничивается этой инструкцией. Например:

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

Если блок исключения имеет секцию else, в этой секции происходит обработка всех исключений, которые не были обработаны в блоке обработчика исключения. То есть:

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

В этом примере секция else обрабатывает все исключения, не являющимися SysUtils.EMathError.

Блок исключения, не содержащий обработчиков, но содержащий список инструкций, обрабатывает любые исключения. Например:

```
try
    ...
except
    HandleException;
end;
```

Здесь подпрограмма HandleException обрабатывает все исключенительные ситуации, которые возникают при выполнении инструкций между try и except.

- Если в контексте работы функции F2() исключение не может быть обработано полностью, эта функция может инициировать повторное исключение с тем, чтобы оно было обработано на следующих уровнях иерархии вызовов:

```
procedure F2()
begin
  try
    F3(); // Может генерировать исключение
  except
    on EMyException do
      begin
        // Частичная обработка ошибки
        // ...
        raise;
      end;
  end;
end;
```

Код, выполняемый в обработчике исключения может сам инициировать и обрабатывать исключительные ситуации. Поскольку исключения обрабатываются внутри обработчика, они не оказывают влияния на первичное исключение. Тем не менее, если исключение инициированное внутри обработчика, не обрабатывается в нем, первичное исключение теряется.

```
type
   ETrigError = class(EMathError);
   function Tan(X: Extended): Extended;
   begin
      try
        Result := Sin(X) / Cos(X);
      except
        on EMathError do
        raise ETrigError.Create('Invalid argument to Tan');
      end;
   end;
```

Иногда вам необходимо быть уверенными, что некоторые части операции выполнены, вне зависимости от того, прерывается ли выполнение операции исключением или нет. Например, если подпрограмма получает управление ресурсом, важно чтобы память, которую он занимает, была освобождена несмотря на то, что подпрограмма завершается ошибкой. В таких ситуациях вы можете использовать инструкцию try...finally.

```
Reset(F);
try
    ... // process file F
finally
    CloseFile(F);
end;
```

Синтаксис инструкции try...finally выглядит следующим образом:

```
try statementList1 finally statementList2 end
```

1. Инструкция try...finally выполняет инструкции в statementList1 (секция try).
2.
   - Если выполнение statementList1 завершается без ошибок, выполняется statementList2 (секция finally).
   - Если в процессе выполнения statementList1 возникает исключительная ситуация, управление передается в statementList2, когда statementList2 завершает свою работу исключение инициируется повторно.
   - Если в процессе выполнения statementList1 происходит вызов процедур Exit, Break или Continue – управление выходит из statementList1 , statementList2 выполняется автоматически.
   - Секция finally выполняется всегда, вне зависимости от того, как завершается работа секции try.
   - Если в секции finally возникает исключительная ситуация, которая не обрабатывается, исключение выводится из из инструкции try...finally, а исключение, инициированное в секции try, разрушается.

Преимущества встраивания в язык обработки исключительных ситуаций

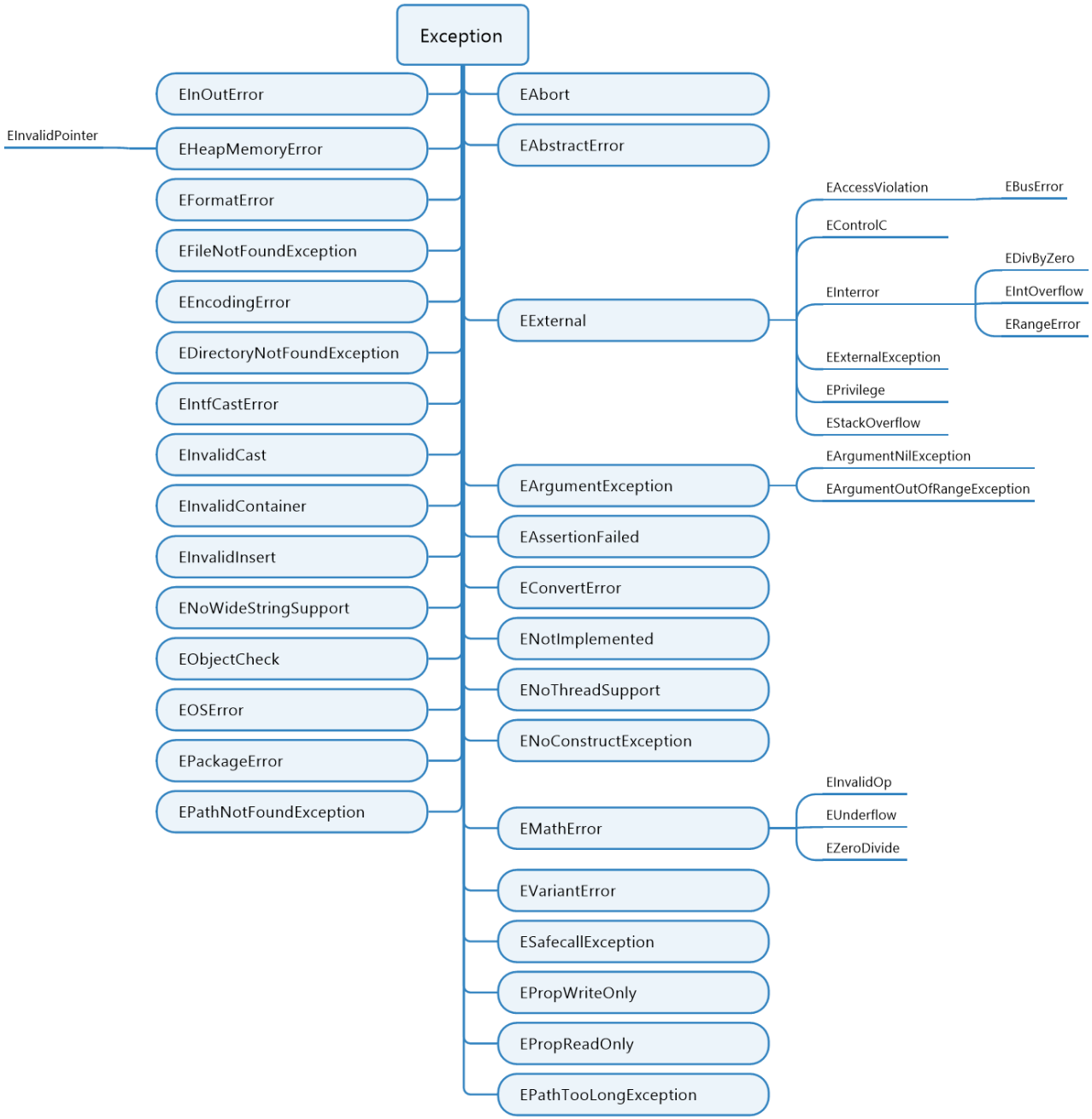
- Без обработки исключений код, необходимый для обнаружения ошибок, может сильно запутать программу.
- Возможность передавать исключительные ситуации для обработки в другие модули.
- Язык поощряет программиста рассматривать все события, которые могут возникнуть в процессе выполнения программы, а не игнорировать их в надежде на то, что "ничего плохого не случится".

- Предпочтительнее использовать в качестве типов исключений типы, определяемые пользователем.
- В классе исключения можно реализовать часть функциональности в связи с обработкой данного типа исключения, например, получение описания исключительной ситуации.

```
type //классы исключений
  EmptyFileException = class(Exception) end;//пустой файл
  BadFormatException = class(Exception) end;//ошибка формата
procedure ReadData(var F: Text; var X: integer);
begin
  if EOF(F) then
    raise EmptyFileException.Create('EmptyFileException');
  try
    read(F, X);
  except
    raise EmptyFileException.Create('EmptyFileException');
  end;
end;
```

```
try
  assign(InputFile, FileName);
  reset(InputFile);
except
  //обработка ошибки открытия файла
  on EInOutError do
    begin
      writeln('Cannot open file');
      readln;
      halt(1);
    end;
end;
```

```
try
  ReadData(InputFile, InputData);
except
  on EmptyFileException do
    begin
      // Обработка ошибки пустого файла
    end;
  on BadFormatException do
    begin
      // Обработка ошибки формата данных файла
    end;
end;
```


Exception
EInOutError
EInvalidPointer
EHeapMemoryError
EFormatError
EFileNotFoundException
EEncodingError
EDirectoryNotFoundException
EIntfCastError
EInvalidCast
EInvalidContainer
EInvalidInsert
ENoWideStringSupport
EObjectCheck
EOSError
EPackageError
EPathNotFoundException
EAbort
EAbstractError
EExternal
EAccessViolation
EBusError
EControlC
EInterror
EDivByZero
EIntOverflow
ERangeError
EExternalException
EPrivilege
EStackOverflow
EArgumentException
EArgumentNilException
EArgumentOutOfRangeException
EAssertionFailed
EConvertError
ENotImplemented
ENoThreadSupport
ENoConstructException
EMathError
EInvalidOp
EUnderflow
EZeroDivide
EVariantError
ESafecallException
EPropWriteOnly
EPropReadOnly
EPathTooLongException

Обработка ошибок в Python

```
>>> a = 10
>>> b = 0
>>> c = a / b
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

Иерархия исключений в Python

```
BaseException
+- SystemExit
+- KeyboardInterrupt
+- GeneratorExit
+- Exception
     +- StopIteration
     +- StopAsyncIteration
     +- ArithmeticError
     |    +- FloatingPointError
     |    +- OverflowError
     |    +- ZeroDivisionError
     +- AssertionError
     +- AttributeError
     +- BufferError
     +- EOFError
     +- ImportError
          +- ModuleNotFoundError
     +- LookupError
     |    +- IndexError
     |    +- KeyError
     +- MemoryError
     +- NameError
     |    +- UnboundLocalError
     +- OSError
     |    +- BlockingIOError
     |    +- ChildProcessError
     |    +- FileExistsError
     |    +- FileNotFoundError
     |    +- InterruptedError
     |    +- IsADirectoryError
     |    +- NotADirectoryError
     |    +- PermissionError
     |    +- ProcessLookupError
     |    +- TimeoutError
     +- ReferenceError
     +- RuntimeError
     |    +- NotImplementedError
     |    +- RecursionError
     +- SyntaxError
     |    +- IndentationError
     |         +- TabError
     +- SystemError
     +- TypeError
     +- ValueError
     |    +- UnicodeError
     |         +- UnicodeDecodeError
     |         +- UnicodeEncodeError
     |         +- UnicodeTranslateError
     +- Warning
          +- DeprecationWarning
          +- PendingDeprecationWarning
          +- RuntimeWarning
          +- SyntaxWarning
```

try...except

В программе возможных два вида исключений – ValueError, возникающее в случае, если на запрос программы «введите число», вы введете строку, и ZeroDivisionError - если вы введете в качестве числа 0.

```
print("start")
val = int(input("input number: "))
tmp = 10 / val
print(tmp)
print("stop")
```

Если ввести 0 на запрос приведенной выше программы, произойдет ее остановка с распечаткой сообщения об исключении:

```
start
 input number: 0
```

try...except

```
print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except Exception as e:
    print("Error! " + str(e))
print("stop")
```

try...except

Согласно документу по языку Python, описывающему ошибки и исключения, оператор try работает следующим образом:

- Вначале выполняется код, находящийся между операторами try и except.
- Если в ходе его выполнения исключения не произошло, то код в блоке except пропускается, а код в блоке try выполняется весь до конца.
- Если исключение происходит, то выполнение в рамках блока try прерывается и выполняется код в блоке except. При этом для оператора except можно указать, какие исключения можно обрабатывать в нем. При возникновении исключения, ищется именно тот блок except, который может обработать данное исключение.
- Если среди except блоков нет подходящего для обработки исключения, то оно передается наружу из блока try. В случае, если обработчик исключения так и не будет найден, то

try...except

Для указания набора исключений, который должен обрабатывать данный блок except их необходимо перечислить в скобках (круглых) через запятую после оператора except.

```
print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except(ValueError, ZeroDivisionError):
    print("Error!")
print("stop")
```

try...except

Если хотим обрабатывать ValueError, ZeroDivisionError по отдельности, при этом, сохранить работоспособность при возникновении исключений отличных от вышеперечисленных:

```
print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except ValueError:
    print("ValueError!")
except ZeroDivisionError:
    print("ZeroDivisionError!")
except:
    print("Error!")
```

try...except

Существует возможность передать подробную информацию о произошедшем исключении в код внутри блока except:

```
print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except ValueError as ve:
    print("ValueError! {0}".format(ve))
except ZeroDivisionError as zde:
    print("ZeroDivisionError! {0}".format(zde))
except Exception as ex:
    print("Error! {0}".format(ex))
print("stop")
```

Использование finally в обработке исключений

Для выполнения определенного программного кода при выходе из блока try/except, используйте оператор finally:

```
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except:
    print("Exception")
finally:
  print("Finally code")
```

Не зависимо от того, возникнет или нет во время выполнения кода в блоке try исключение, код в блоке finally все равно будет выполнен.

Использование finally в обработке исключений

Если необходимо выполнить какой-то программный код, в случае если в процессе выполнения блока try не возникло исключений, то можно использовать оператор else:

```
try:
    f = open("tmp.txt", "r")
    for line in f:
        print(line)
    f.close()
except Exception as e:
    print(e)
else:
    print("File was readed")
```

Генерация исключений в Python

Для принудительной генерации исключения используется инструкция raise. Самый простой пример работы с raise может выглядеть так:

```
try:
    raise Exception("Some exception")
except Exception as e:
    print("Exception exception " + str(e))
```

Пользовательские исключения в Python

В Python можно создавать собственные исключения. Такая практика позволяет увеличить гибкость процесса обработки ошибок в рамках той предметной области, для которой написана ваша программа.

Для реализации собственного типа исключения необходимо создать класс, являющийся наследником от одного из классов исключений:

```
class NegValException(Exception):
    pass
try:
    val = int(input("input positive number: "))
    if val < 0:
        raise NegValException("Neg val: " + str(val))
    print(val + 10)
except NegValException as e:
    print(e)
```

```
f = open('1.txt')

ints = []

try:
        for line in f:
                ints.append(int(line))
except ValueError:
        print('Это не число. Выходим.')
except Exception:
        print('Это что ещё такое?')
else:
        print('Всё хорошо.')
finally:
        f.close()
        print('Я закрыл файл.')
```