Язык программирования D

Презентация 4.

Обработка ошибок и исключения. Встроенные средства отладки (assert). Специфические конструкции языка

Морщинин Н. 5030102/20201

Типы ошибок (наследники Throwable)

- **Исключения (Exceptions)** - обрабатываемые ошибки, которые можно перехватить
- **Ошибки (Errors)** - фатальные сбои, которые нельзя перехватить

Базовый принцип

- Исключения представляются как объекты с информацией об ошибке
- Базовый класс `Exception` является родителем для всех исключений

Архитектура исключений

Специализация

- Каждый модуль/класс реализует собственные исключения
- Примеры:
  - Файловая система → исключения работы с файлами
  - База данных → исключения работы с БД
  - XML-обработчик → исключения формата файлов

```
class FileException : Exception { /* ... */ }
class DatabaseException : Exception { /* ... */ }
class XMLException : Exception { /* ... */ }
```

Пример в коде: обработка исключений

```
import std.stdio;
import std.algorithm; // endsWith позволяет проверить конец строки

void main(){
  string filename = "myimage.png";
  try {
    write("My First Exception: ");
    processXMLFile(filename); // передаем имя файла в функцию, ожидающую xml файл
  }
  catch(Exception e)  {
    writeln(e.msg); // обращаемся к полю, содержащему текст исключения
  }
  finally {
    // не зависимо от того, поймали ли мы исключение или нет, выполняем какую-либо операцию
  }
}

void processXMLFile(string name) {
  if(!name.endsWith(".xml"))
    throw new Exception("Wrong file type");
  else  {
    // Продолжаем нормальный ход выполнения
  }
}
```

Пример в коде: обработка нескольких исключений

```
try {
    // выполняем блок кода, который может выбрасывать исключения

    } catch (first_exception_type) {
        // ловим первый тип исключений
        // делаем какую-то обработку
    }
    catch (second_exception_type) {
        // ловим второй тип исключений, который так же может выбрасываться
        // делаем какую-то обработку
    }
    catch (Exception) {// ловим базовый тип исключений
        // ловим второй тип исключений, который так же может выбрасываться
        // делаем какую-то обработку
    }
    finally {
        // Выполняем действия не зависимо от того, были ли брошены исключения или нет
    }
}
```

Атрибут `nothrow`

Назначение

- Гарантирует, что функция не генерирует исключения
- Используется в высококритичных компонентах, где исключения недопустимы

Области применения

- Системы с повышенными требованиями к надежности
- Компоненты, где повреждение данных недопустимо
- Ситуации, когда продолжение работы при ошибке бессмысленно

`nothrow`: пример в коде

```d
//app.d
import std.stdio;
void main(){
  foo();
}

void foo() nothrow {
  throw new Exception("Exception text");
}
```

Результат:

```
dmd app.d
app.d(10): Error: object.Exception is thrown but not caught
app.d(8): Error: nothrow function 'app.foo' may throw
```

Блок `scope`

- Автоматическое выполнение кода при выходе из блока
- Компактная замена try-catch-finally конструкций

Практическое применение

- Освобождение ресурсов (файлы, сетевые соединения)
- Откат транзакций
- Логирование завершения операций
- Гарантированное выполнение cleanup-кода

`scope`: пример в коде

```
import std.stdio;
void main() {
  writeln("Hello before");
  foo();
  writeln("Hello after");
}


void foo() {
    scope(exit) writeln("1");
    scope(success) writeln("2"); // никогда не будет выведено, т.к. успех не равно ошибке
    scope(exit) writeln("3");
    scope(failure) writeln("4");
      throw new Exception("My Exception"); // кидаем исключение, чтобы сработал блок failure
}
```

`assert` - базовое средство проверки

Assert проверяет условие во время выполнения и прерывает программу при false

```
assert(expression, message);
```

- Выполняется только в debug-версиях
- Отключается в release-версиях флагом -release
- Генерирует AssertError при неудаче

Практическое использование assert

Проверка предусловий:

```
void processData(int[] data) {
    assert(data !is null, "Data cannot be null");
    assert(data.length > 0, "Data cannot be empty");
    // обработка данных
}
```

Проверка инвариантов:

```
class BankAccount {
    private double balance;
    void withdraw(double amount) {
        assert(amount > 0, "Withdrawal amount must be positive");
        balance -= amount;
        assert(balance >= 0, "Balance cannot be negative");
    }
}
```

Assert с пользовательскими сообщениями

Детализированная диагностика:

```
assert(index >= 0 && index < array.length,
        "Index " ~ index.to!string ~ " out of bounds [0, " ~
        array.length.to!string ~ ")");
```

Проверка сложных условий:

```
assert(isValidEmail(email), "Invalid email format: " ~ email);
```

Другие средства отладки

`debug` - условная компиляция

```
debug {
    writeln("Debug: processing item ", item);
    // отладочный код
}

debug(important_checks) {
    validateCriticalData(data);
}
```

Запуск с флагами: -debug или -debug=important_checks

`version` - управление версиями

```
version(Development) {
    enableDetailedLogging();
    runExtraTests();
}

version(Production) {
    disableDebugFeatures();
    enableOptimizations();
}
```

Компиляция с флагами: -version=Development

`static assert` - проверка на этапе компиляции

```
static assert(is(T == int), "T must be int");
static assert(size > 0, "Size must be positive");
static assert(int.sizeof == 4, "Requires 32-bit int");
```

- Выполняется во время компиляции
- Гарантирует соблюдение условий типов и констант

`debug` -спецификаторы функций

```
debug void logDebugInfo(string message) {
    writeln("DEBUG: ", message);
}

unittest {
    debug logDebugInfo("Running unit test");
    // тестовый код
}
```

Функция доступна только в debug-сборках

`unittest` - встроенное модульное тестирование

Блоки unittest для написания тестов, выполняемых с флагом -unittest:

```
int add(int a, int b) {
    return a + b;
}

unittest {
    assert(add(2, 2) == 4);
    assert(add(0, 0) == 0);
    assert(add(-1, 1) == 0);
}
```

- Тесты не компилируются в релизной сборке
- Позволяют проверять корректность кода
- Могут быть расположены в любом месте кода

Сочетание средств отладки

Комплексный подход к отладке:

```
version(UnitTest) {
    debug void validateState() {
        assert(invariantCheck(), "Invariant violated");
    }
}

void criticalOperation() {
    debug validateState();
    // основная логика
    debug validateState();
}
```

Использованные ресурсы

1. https://dlang.ru/book
2. https://habr.com/ru/articles/261043/
3. https://github.com/dlang-community/awesome-d

Спасибо за внимание!