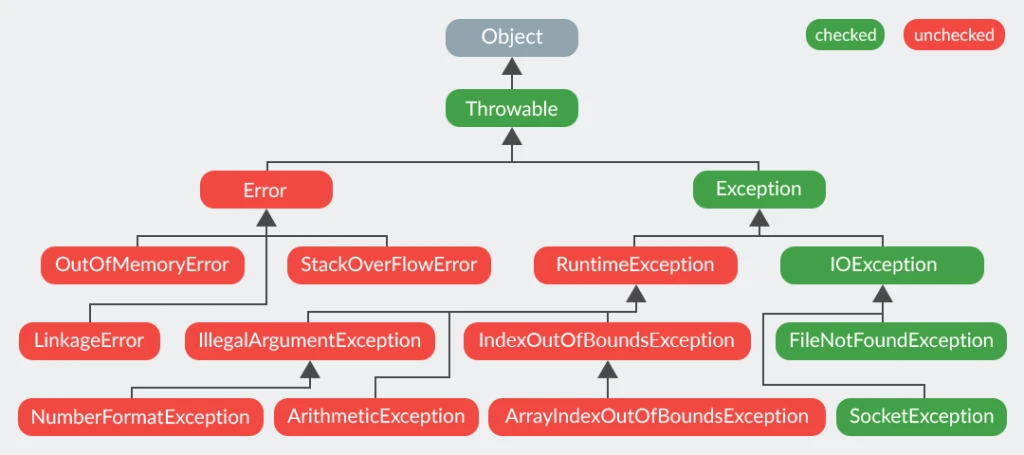
Обработка ошибок в Java



Все исключения в Java делятся на 2 вида — проверяемые (**checked**) и непроверяемые (**unchecked**).

**Проверяемые исключения** — это ошибки, которые Java-компилятор может предсказать во время компиляции кода. Это чаще не ошибки програмной логики, а ошибки ввода-вывода, ошибки SQL и другие. Пример – IOException

Код, который может вызвать проверяемое исключение, должен либо обработать его с помощью блока try-catch, либо передать его вверх по стеку вызовов с помощью ключевого слова throws.

**Непроверяемые исключения** — это ошибки, которые Java-компилятор не может предсказать во время компиляции. Поэтому они обычно связаны с логическими ошибками в коде. Примеры таких исключений — это NullPointerException, ArrayIndexOutOfBoundsException и другие.

Код, который может вызвать непроверяемое исключение, не обязан его обрабатывать или объявлять с помощью throws.

**@ControllerAdvice** - Аннотация, которая позволяет объявить класс в качестве глобального обработчика исключений для всех контроллеров в приложении. Это означает, что методы в этом классе будут вызываться при возникновении исключений в любом из контроллеров. Обычно это используется для централизованной обработки исключений и управления ошибками в приложении.

**@ExceptionHandler** – Аннотация, которая используется внутри классов @ControllerAdvice или контроллеров для определения методов, которые обрабатывают конкретные типы исключений. Когда исключение указанного типа происходит в контроллере, метод, помеченный @ExceptionHandler, будет вызван для его обработки.

**@Valid** - Аннотация используется в Spring Boot для включения автоматической валидации объектов. Когда объект помечен аннотацией @Valid перед его параметром в методе контроллера, Spring будет автоматически проводить валидацию этого объекта перед его использованием в методе.

public class User {  
 @NotBlank(message = "Имя не должно быть пустым")  
 private String name;  
  
 @Email(message = "Некорректный адрес электронной почты")  
 private String email;  
  
 // геттеры и сеттеры  
}

@RestController  
public class UserController {  
  
 @PostMapping("/users")  
 public ResponseEntity<String> createUser(@Valid @RequestBody User user) {  
 // метод будет вызван только если объект User прошел валидацию  
 return ResponseEntity.*ok*("Пользователь успешно создан");  
 }  
}

Если проверка не пройдена, генерируется исключение типа MethodArgumentNotValidException, и возвращает сообщение об ошибках валидации в формате строки.

**Лог (log)** - это хронологическая запись наиболее значимой информации о работе системы. Подобная фиксация событий дает представление и том, что происходило в системе, в какой именно момент, какой пользователь спровоцировал то или иное событие, какие ошибки возникли и т.д.

**Логирование** – процесс формирования логов : )

**Уровни логирования:**

**1. TRACE**: Самый низкий уровень логирования, который предназначен для записи наиболее детальной информации, например, информации о каждом шаге выполнения методов, потоков выполнения и т.д. Обычно используется для отладки и анализа деталей выполнения приложения.

**2. DEBUG**: Уровень, который используется для записи отладочной информации. Сообщения уровня DEBUG могут содержать дополнительную информацию о состоянии приложения, значения переменных, потоках выполнения и т.д. Этот уровень логирования полезен при разработке и отладке программы.

**3. INFO**: Уровень, который предназначен для записи информации о нормальном выполнении приложения. Сообщения уровня INFO могут включать в себя информацию о выполнении ключевых операций, статусе приложения, запросах и т.д. Этот уровень логирования используется для сообщений о нормальной работе приложения.

**4. WARN**: Уровень, который используется для записи предупреждений о потенциальных проблемах или нежелательных событиях, которые могут привести к ошибкам в будущем. Сообщения уровня WARN обычно указывают на события, которые требуют внимания и возможно нуждаются в коррекции.

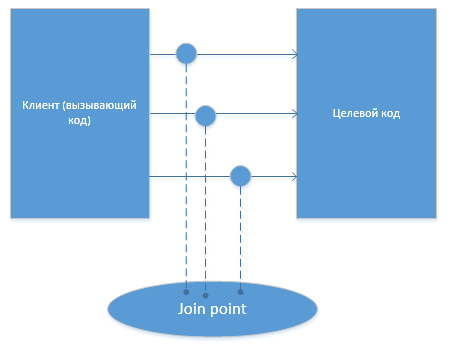
**5. ERROR**: Уровень, который предназначен для записи сообщений об ошибках, которые могут привести к некорректной работе приложения. Сообщения уровня ERROR указывают на реальные ошибки или исключения, возникшие в процессе выполнения программы.

**6. FATAL**: Некоторые системы логирования также поддерживают уровень FATAL, который используется для записи сообщений об критических ошибках, после которых приложение может быть принудительно завершено.

**Spring AOP (Aspect-Oriented Programming)** предоставляет возможность создавать аспекты, которые позволяют модульно разделять пересекающиеся аспекты функциональности в приложении. В Spring Framework есть несколько основных библиотек для работы с AOP:

**AspectJ** - это более мощная библиотека AOP, предоставляющая полный набор возможностей, таких как использование различных типов советов (before, after, around, after-throwing, after-returning), точное сопоставление советов с точками среза (pointcuts) и т.д. AspectJ поддерживает как компиляцию аспектов в байт-код (compile-time weaving), так и их применение во время выполнения (load-time weaving).

**Join point (точка присоединения)** - это конкретная точка выполнения в программе, в которой аспект может быть выполнен. В контексте аспектно-ориентированного программирования (AOP), join point представляет собой определенное место в выполнении программы, такое как вызов метода, завершение метода, обращение к полю объекта и т. д., где аспект может вступить в действие и применить свои советы (advice)



**Паттерн проектирования** в Java (и в общем случае в разработке программного обеспечения) представляет собой шаблон архитектуры программы, который описывает способ организации классов и объектов для решения конкретной проблемы в разработке программного обеспечения.

**Порождающие паттерны**: Эти паттерны связаны с процессом создания объектов и обеспечивают гибкость и повторное использование кода

**Структурные паттерны**: Они определяют способы композиции классов и объектов для создания новых структур

**Поведенческие паттерны**: Эти паттерны определяют взаимодействие между объектами и облегчают обмен информацией между ними

[Типы паттернов в Java](https://refactoring.guru/ru/design-patterns/java)

**Singleton** — это порождающий паттерн, который гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру.

Пример Singleton

public class Singleton {  
 // Приватное статическое поле для хранения единственного экземпляра класса  
 private static Singleton *instance*;  
  
 // Приватный конструктор, чтобы предотвратить создание экземпляров класса извне  
 private Singleton() {  
 }  
  
 // Публичный статический метод для получения экземпляра класса  
 public static Singleton getInstance() {  
 // Проверяем, существует ли уже экземпляр класса  
 if (*instance* == null) {  
 // Если экземпляр еще не создан, создаем новый  
 *instance* = new Singleton();  
 }  
 // Возвращаем единственный экземпляр класса  
 return *instance*;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 // Получаем экземпляр Singleton  
 Singleton singleton1 = Singleton.getInstance();  
 Singleton singleton2 = Singleton.getInstance();  
  
 // Проверяем, что это один и тот же объект  
 System.*out*.println(singleton1 == singleton2); // Выведет: true  
 }  
}

**Рефакторинг**— это переработка исходного кода программы, чтобы он стал более простым и понятным. Рефакторинг не меняет поведение программы, не исправляет ошибки и не добавляет новую функциональность. Он делает код более понятным и удобочитаемым.