Программирование
1 семестр

ИТМО

# Исключения

# Ошибка, дефект, сбой

- Человеку свойственно ошибаться
- Ошибка - некорректное действие разработчика ПО
- Дефект - результат ошибки в коде программы
- Сбой - проявление дефекта во время работы ПО
  - не всегда сразу
  - не всегда заметно

# Когда проявляются ошибки?

- На этапе компиляции - проверяет компилятор (IDE)
  - синтаксические ошибки
  - ошибки типов данных
- На этапе выполнения - возникают во время работы
  - отсутствие ресурса
  - ошибки данных
  - логические ошибки

# Реакция на ошибки

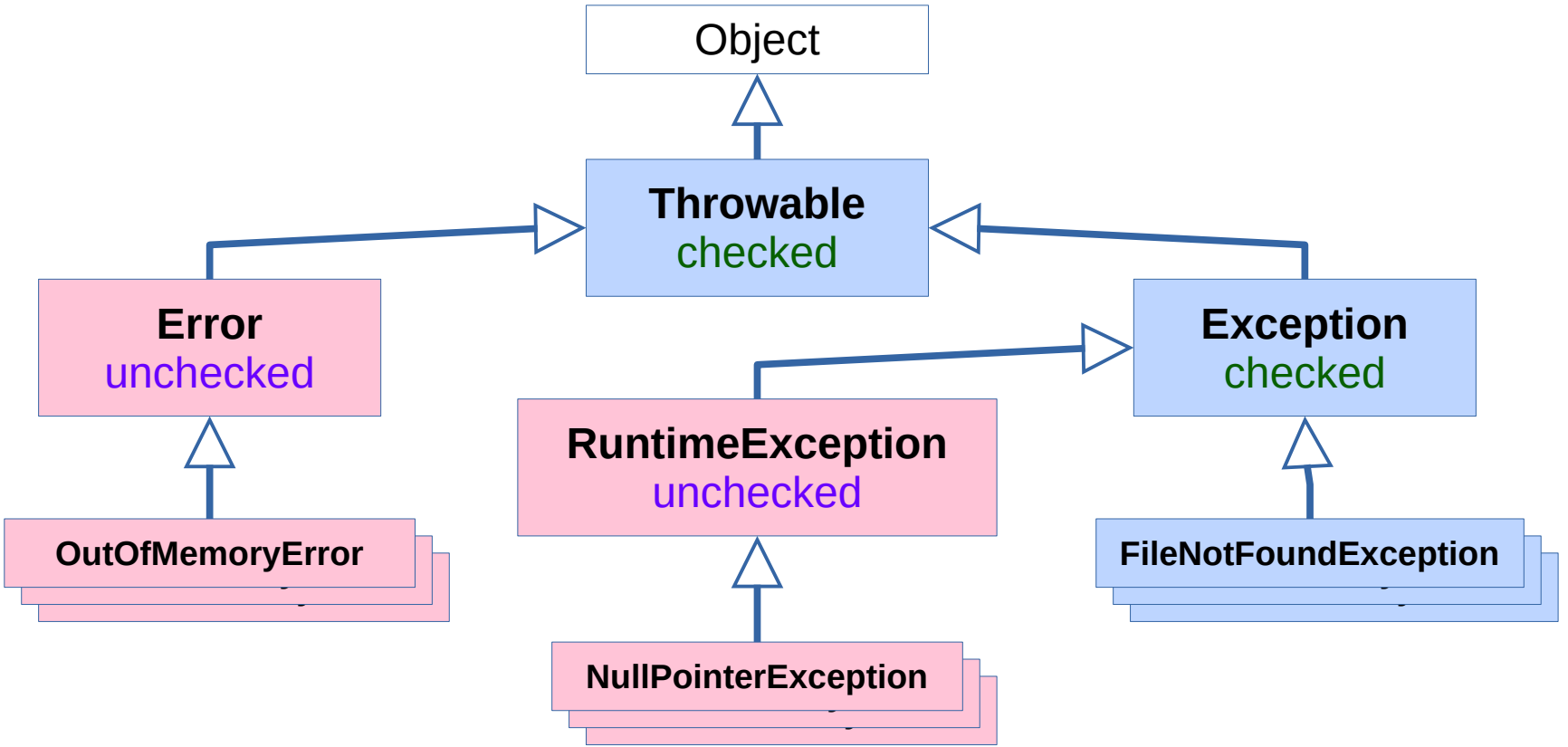
- Способы реакции на сбой
  - код возврата (0 - OK, не 0 - ошибка)
    - неудобно возвращать (нужно вернуть и результат работы)
    - неудобно проверять (много дополнительного кода)
  - флаг ошибки или глобальная переменная
    - можно забыть проверить
  - исключения

# Система исключений

ИТМО

- Специальный объект - исключение (что-то пошло не так)
  - создается системой (JVM) или программой (new)
  - выбрасывается (throw)
  - требует обработки в коде
    - try - код, где может возникнуть исключение
    - catch - код, отлавливающий исключение
    - finally - код, завершающий обработку

# Иерархия классов исключений



# Unchecked

ИТМО

- Непроверяемое или неконтролируемое исключение
  - Компилятору все равно, обработано оно или нет
  - Обычно выбрасывается виртуальной машиной
    - Класс RuntimeException и его потомки
      - Чаще всего проблема в коде (можно было проверить)
      - Можно обработать, но пусть лучше проявится
    - Класс Error и его потомки
      - Критическая ошибка
      - Можно обработать, но это уже бесполезно

# Checked

ИТМО

- Проверяемое или контролируемое исключение
  - Класс Exception и его потомки (кроме RuntimeException)
    - Обычно явно выбрасывается в приложении или библиотеке
    - Обычно вызывается внешними причинами
  - Компилятор проверяет, что про исключение не забыли
    - Нельзя оставлять без внимания (код не компилируется)
    - Правило "catch or throw"

# Checked - 2 варианта действий

ИТМО

- Checked исключение нельзя игнорировать
  - если понятно, что делать - обработать
  - если не хватает информации для решения - передать выше

```
void readConfig() {
  open("file");
}
```

```
void readImportantData() {
  open("file");
}
```

```
void open(fileName)
```

# Обработка исключений - try-catch-finally

ИТМО

```
try {
   // проблемный код
} catch (Exception e) {
   // если исключение
   // e – объект исключения
} finally {
   // выполнится всегда
}
```

```
try {
   File f = open("name");
   int x = f.readInt();
   result = 1 / x;
   calculate(result);
} catch (ArithmeticException e) {
   System.out.println("/ 0 !!");
} catch (IOException e) {
   System.out.println("No file!");
} finally {
   f.close();
}
```

# Проброс исключения - throws

ИТМО

```
void main(String... args) {

  methodWithCheckedException();



}
void methodWithCheckedException() {
  libraryMethod(); // может вызвать исключение
}
```

# Проброс исключения - throws

```
void main(String... args) {

  methodWithCheckedException();




}
void methodWithCheckedException() throws Exception {
   libraryMethod();
}
```

# Проброс исключения - throws

ИТМО

```
void main(String... args) {
  try {
    methodWithCheckedException();
  } catch (Exception e) {
    System.err.println("Exception happened");
  }
}
void methodWithCheckedException() throws Exception {
  libraryMethod();
}
```

# Выбрасывание исключения - throw

ИТМО

- Исключение выбрасывается и
  - Ловится в первом подходящем блоке catch
  - Иначе пробрасывается выше в вызывающий метод
- Процесс повторяется, пока не дойдет до main

# try-with-resources

- Автозакрытие ресурсов
  - интерфейс AutoClosable

```
try (File f = open("name")) {
  int x = f.readInt();
  result = 1 / x;
  calculate(result);
} catch (ArithmeticException e) {
  System.out.println("/ 0 !!");
} catch (IOException e) {
  System.out.println("No file!");
}
```

# Множественный catch

ИТМО

- Стандартный вариант
  - чем дальше - тем более обобщенное исключение
- Если действие такое же
  - можно в один блок

```
try (File f = open("name")) {
  int x = f.readInt();
  result = 1 / x;
  calculate(result);
} catch (ArithmeticException |
         IOException e) {
  System.out.println("Error!");
}
```

# Собственные исключения

ИТМО

- Моделирование ошибок предметной области.
- VeggyBreakException extends Exception (RuntimeException)
- В конструктор можно передать
  - поясняющее сообщение (String)
  - вызвавшее сбой исключение-причину (Exception)
- throw new VeggyBreakException("Репка разломилась!");

# Цепочка исключений (Chained Exceptions)

ИТМО

- Возможность связать исключение с причиной.
- Конструктор Exception(String message, Throwable cause)
- Метод getCause() для получения исходного исключения

# Исключения - как надо

ИТМО

- Проверяемые исключения
  - обработать или пробросить
  - try лучше с ресурсами, чтобы они закрылись сами
  - catch от наиболее специфичного к более обобщенным
- Непроверяемые исключения
  - лучше всего фиксировать в логах

# Исключения - как не надо

ИТМО

- Проверяемые исключения - не надо
  - размещать весь метод в блоке try с пустым catch
  - во всех методах (и в main) указывать throws Exception
  - превращать в непроверяемые
- Непроверяемые исключения - не надо
  - ловить ошибки
  - оставлять printStackTrace

# ОО анализ и проектирование (OOA&D)

ИТМО

- Концепции ООП:
  - абстракция
  - инкапсуляция
  - наследование
  - полиморфизм

- Принципы проектирования
  - DRY
  - KISS
  - YAGNI
  - SOLID
  - GRASP

# Анализ предметной области

ИТМО

- Основные сущности
- Существительные
- Границы ответственности
- Атрибуты
- Характеристики
- Действия
- Глаголы

- Потенциальные классы
  - кто? что?
  - за что отвечает класс?
- Переменные (состояние)
  - чем отличаются объекты?
- Методы (поведение)
  - что могут делать объекты?

# DRY

- Don't Repeat Yourself - Не повторяйся!
- Исключение дублирования кода
- Все изменения - в одном месте
- Вместо Copy-Paste - отдельный метод

# KISS

- Keep It Simple, Stupid! - Делай проще, дурачок!
- Простое решение обычно лучше, быстрее и надежнее
- Не нужно усложнять без необходимости
- Проще код - меньше ошибок

# YAGNI

ИТМО

- You Aren't Gonna Need It! - Тебе это не нужно!
- В коде должно быть только то, что нужно сейчас
- Не нужно делать "на всякий случай"
- Лишний код нужно поддерживать, документировать

# Принципы SOLID

ИТМО

- Автор - Боб Мартин
- 5 принципов
  - SRP = Single Responsibility Principle
  - OCP = Open-Closed Principle
  - LSP = Liskov Substitution Principle
  - ISP = Interface Segregation Principle
  - DIP = Dependency Inversion Principle

# SRP - Single Responsibility Principle

ИТМО

- Принцип единственной ответственности
- Класс (или модуль) должен иметь одну и только одну причину для изменения
  - Поддерживаемость
  - Тестируемость
  - Гибкость

```
Student
+ getID()
+ getName()
+ passExam()
+ checkCard()
```

# SRP - Single Responsibility Principle

ИТМО

- Принцип единственной ответственности
- Класс (или модуль) должен иметь одну и только одну причину для изменения
  - Поддерживаемость
  - Тестируемость
  - Гибкость

```
AccessControl
+ check(Person)
```

```
AccessToken
+ isValid()
```

```
Student
+ getID()
+ getName()
+ passExam()
+ getToken()
```

# OCP - Open-Closed Principle

ИТМО

- Принцип открытости-закрытости
- Класс должен быть открыт для расширений, но закрыт для модификации
  - Расширяемость
  - Стабильность
  - Гибкость

```
Student
+ getID()
+ getName()
+ passExam()
+ checkCard()
```

```
Security

if (s.checkCard())
```

# OCP - Open-Closed Principle

ИТМО

- Принцип открытости-закрытости
- Класс должен быть открыт для расширений, но закрыт для модификации
  - Расширяемость
  - Стабильность
  - Гибкость

```
Student
+ getID()
+ getName()
+ passExam()
+ getToken()
```

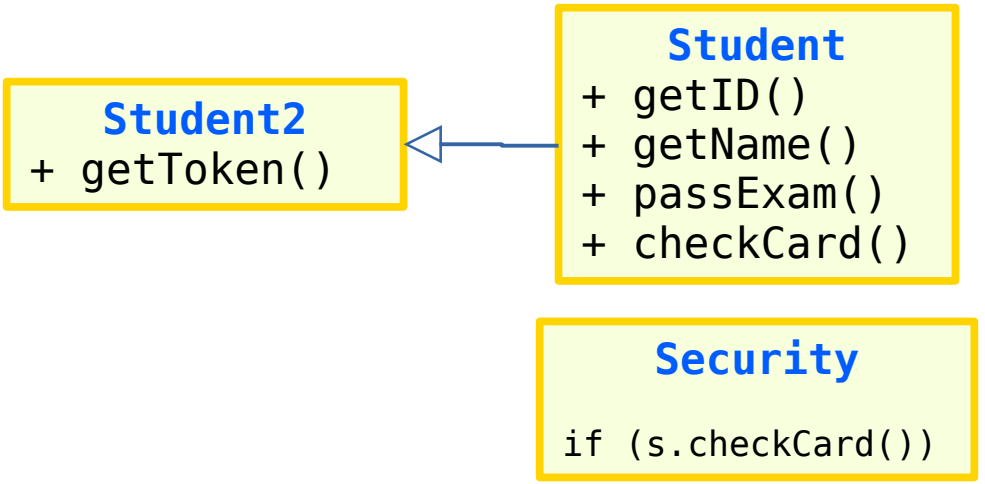
```
Security

if (s.checkCard())
```

# OCP - Open-Closed Principle

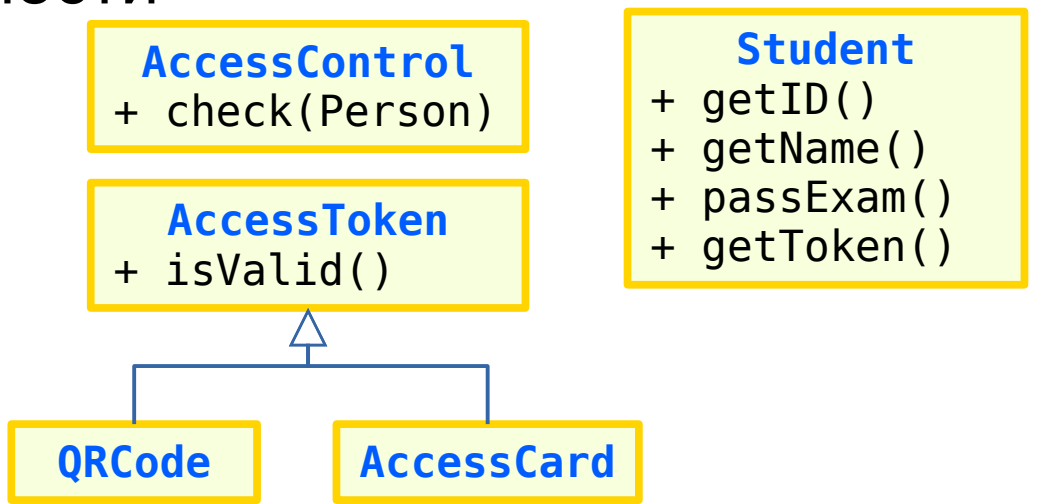
ИТМО

- Принцип открытости-закрытости
- Класс должен быть открыт для расширений, но закрыт для модификации
  - Расширяемость
  - Стабильность
  - Гибкость




# LSP - LIskov Substitution Principle

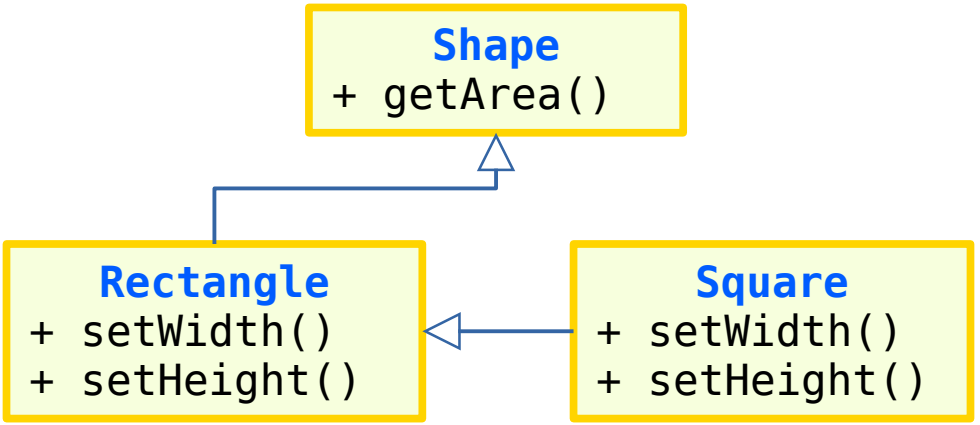
- Принцип подстановки Барбары Лисков
- Подклассы должны уметь заменять базовые классы без нарушения функциональности
  - Полиморфизм
  - Надежность
  - Предсказуемость



# LSP - LIskov Substitution Principle

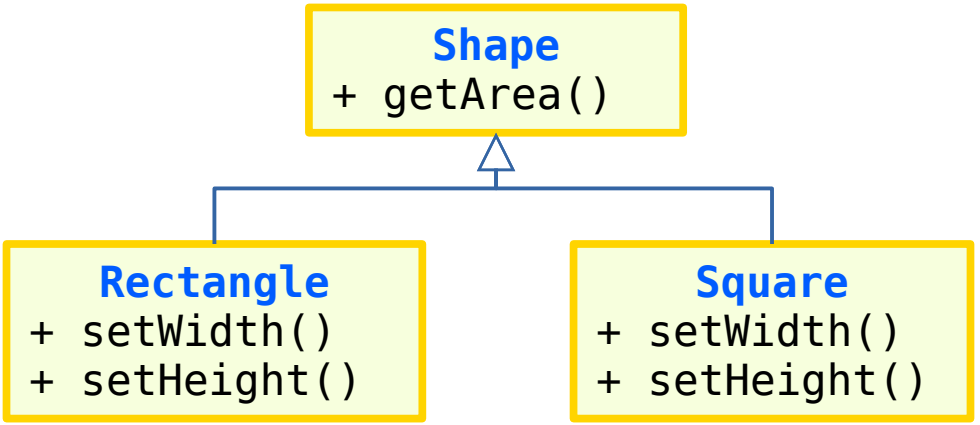
ИТМО

- Принцип подстановки Барбары Лисков
- Подклассы должны уметь заменять базовые классы без нарушения функциональности
  - Полиморфизм
  - Надежность
  - Предсказуемость




# LSP - LIskov Substitution Principle

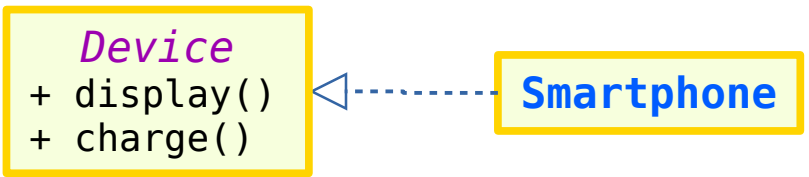
- Принцип подстановки Барбары Лисков
- Подклассы должны уметь заменять базовые классы без нарушения функциональности
  - Полиморфизм
  - Надежность
  - Предсказуемость




# ISP - Interface Segregation Principle

ИТМО

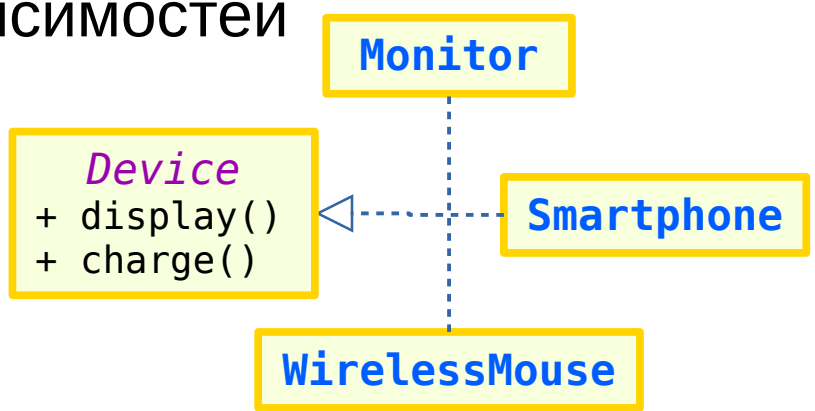
- Принцип разделения интерфейсов
- Клиент не должны зависеть от методов интерфейсов, которые им не нужны
  - Исключение лишних зависимостей
  - Разделение поведения
  - Гибкость




# ISP - Interface Segregation Principle

ИТМО

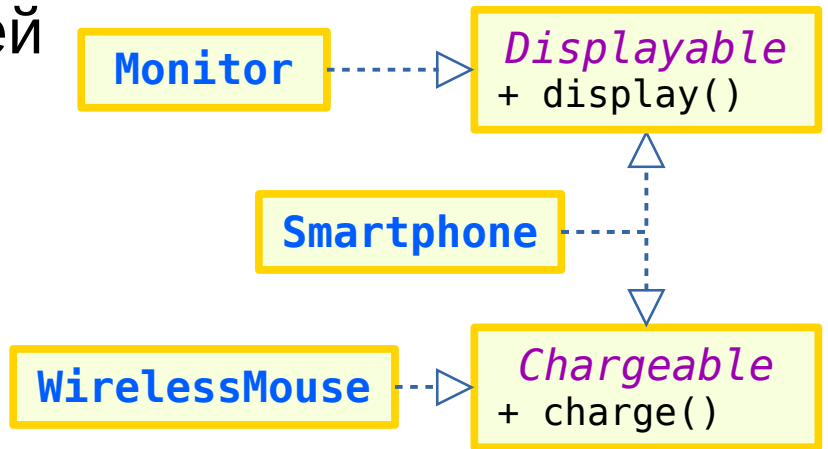
- Принцип разделения интерфейсов
- Клиент не должны зависеть от методов интерфейсов, которые им не нужны
  - Исключение лишних зависимостей
  - Разделение поведения
  - Гибкость




# ISP - Interface Segregation Principle

ИТМО

- Принцип разделения интерфейсов
- Клиент не должны зависеть от методов интерфейсов, которые им не нужны
  - Исключение лишних зависимостей
  - Разделение поведения
  - Гибкость




# DIP - Dependency Inversion Principle

ИТМО

- Принцип инверсии зависимостей
- Зависимость от абстракции, а не от реализации
  - Слабая взаимосвязанность
  - Поддерживаемость
  - Гибкость

```
class BachelorStudent {
  void doExam(Programming prog) {
    prog.getExam().getTasks();
  }
}
class Programming {
  Exam getExam() {}
}
class Exam {
  Task[] getTasks() {}
}
```

# DIP - Dependency Inversion Principle

ИТМО

- Принцип инверсии зависимостей
- Зависимость от абстракции, а не от реализации
  - Слабая взаимосвязанность
  - Поддерживаемость
  - Гибкость

```
class Bachelor extends Student { }
class Programming extends Subject { }
class Exam extends Control { }
```

```
class Student {
  void doExam(Subject subj) {
    subj.getControl().getTasks();
  }
}
class Subject {
  Control getExam() {}
}
class Control {
  Task[] getTasks() {}
}
```

# GRASP

ИТМО

- General Responsibility Assignment Software Patterns
- Общие паттерны назначения ответственности ПО

- Information Expert
- Creator
- Controller

- Low Coupling
- Hith Cohesion
- Pure Fabrication

- Indirection
- Polymorphism
- Protected Variations

# Information Expert

- Информационный эксперт
- Какой класс должен выполнять обязанность?
  - Тот, у которого максимум информации для ее выполнения

# Creator

ИТМО

- Создатель
- Какой класс отвечает за создание объектов класса X?
  - В котором они находятся
  - Который их сохраняет
  - Который активно их использует
  - У которого есть данные для их инициализации

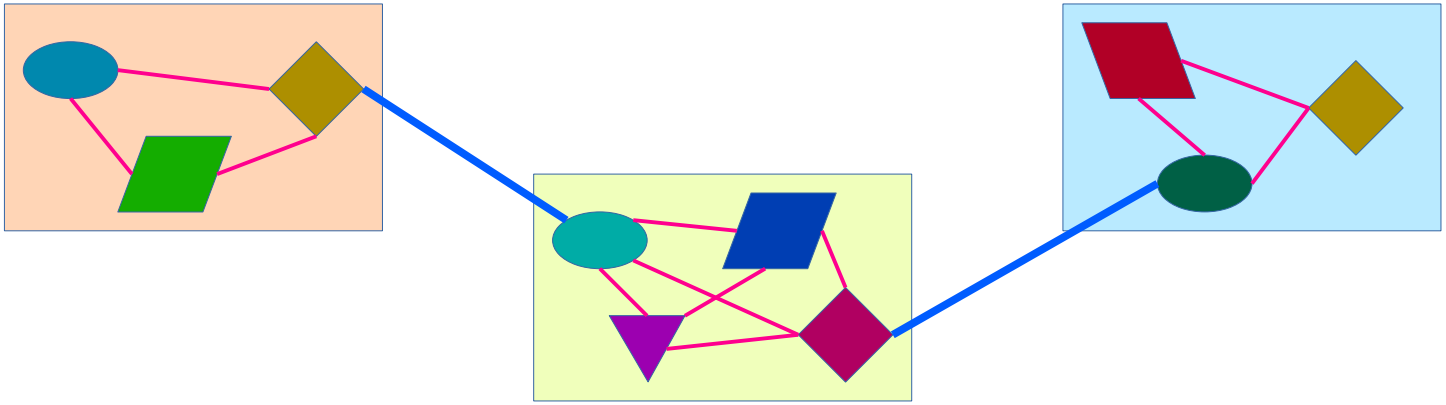
# Controller

- Контроллер
- Какой класс отвечает за обработку запросов пользователя?
  - Представляющий систему или подсистему в целом
  - Отвечающий за сценарий использования
- Контроллер делегирует запросы исполнителям

# Low Coupling / High Cohesion

ИТМО

- Низкая взаимосвязность / Высокое сцепление
- Coupling - между классами или модулями
- Cohesion - внутри классов или модулей



# Pure Fabrication

- Чистая выдумка
- Искусственно придуманный класс для реализации
  - высокого сцепления
  - низкой взаимосвязанности
  - повторного использования

# Polymorphism

ИТМО

- Полиморфизм
- Реализует
  - альтернативные варианты поведения на базе класса
  - подключаемые компоненты с разным поведением

# Indirection

- Перенаправление
- Применение посредника для снижения взаимосвязанности
  - Клиент обращается к промежуточному сервису
  - Сервис взаимодействует с исполнителями

# Protected Variations

ИТМО

- Защищенные изменения
- Возможные изменения выделяются в отдельную сущность
  - Исключается влияние изменений на другие компоненты
  - Инкапсуляция, полиморфизм
  - Внешние данные
  - Встроенный язык
  - Поиск сервиса

# Наследование vs Композиция

ИТМО

- Наследование (is-a):
  - жесткая связь
  - белый ящик
  - полиморфизм
- Композиция (has-a):
  - гибкая связь
  - черный ящик
  - делегирование.

- Правило: "Предпочитайте композицию наследованию"
  - Favor composition over inheritance

# Закон Деметры

ИТМО

- "Не разговаривай с незнакомцами"
  - Объект не должен ничего знать о структуре других объектов
  - В методе M объекта O можно вызывать только методы:
    - самого объекта O и его полей
    - параметров метода M и объектов, созданных внутри M
  - Не стоит использовать больше одной точки
    - ~~o.m().n()~~

# Что такое чистый код?

ИТМО

- Код, который легко
  - читать
  - понимать
  - изменять
  - поддерживать
- Основная цель — снижение стоимости поддержки.
- Код не для машины, а для человека (и часто не для себя)
- Роберт Мартин. "Чистый код"

# Зачем?

- Меньше затраты времени на поддержку
- Меньше потенциальных ошибок
- Командная работа
- Профессионализм

# Общие принципы

- Самодокументируемость
- Информативность
- Принцип наименьшего удивления
- Единый стиль, особенно в команде
- Имена, комментарии, форматирование

# Именование

- Принципы именования:
  - осмысленные и понятные имена
  - содержательные имена
  - имена, которые проще искать
- Соглашения по именованию
  - модули/пакеты - обратное имя домена строчными буквами
  - классы/интерфейсы - существительные (CamelCase)
  - методы - глаголы (lowerCamelCase, как и переменные)
  - константы - SNAKE_CASE (заглавными буквами)

# Именование классов и объектов

ИТМО

- Имена классов — существительные или словосочетания (Customer, AccountParser)
- Избегать общих слов (Data, Info, Manager)
- Имя должно быть ясным и конкретным

# Именование методов

ИТМО

- Имена методов — глаголы или словосочетания (makePayment, deletePage)
- Методы доступа: get, is, set
- Методы-преобразователи: toString, asList
- Имя должно отражать действие, а не реализацию

# Именование переменных и констант

ИТМО

- Использовать константы вместо "магических чисел".
- i, j, k только для счетчиков в коротких циклах.
- Имя должно указывать на назначение переменной

# Проблемы с именами. Антипаттерны

- Слишком короткие имена (a1, a2).
- Слишком длинные и громоздкие имена.
- Шуточные или непрофессиональные имена.
- Использование разных слов для одного понятия.

# Функции и методы - размер

- Должны быть короткими ("20 строк — это уже много")
  - проще понять, протестировать и переиспользовать.
- Должны делать что-то одно
- Если функция большая - поделить на несколько

# Функции и методы - аргументы

- Идеальное количество аргументов - 0
- 1 или 2 - нормально
- 3 - требует обоснования
- Больше 3-х - потенциально плохой вариант
- Параметры только входные

# Комментарии

ИТМО

- Лучшая документация — сам код

- Хорошие:
  - юридические
  - предупреждения
  - пояснения
  - TODO
  - JavaDoc

- Плохие:
  - закомментированный код
  - избыточные
  - "Капитан Очевидность"
  - неясные

# Форматирование кода - вертикальное

ИТМО

- Пустые строки для разделения логических блоков.
- Связанные понятия — рядом
- Зависимые функции — рядом

# Форматирование кода - горизонтальное

ИТМО

- Длина строки обычно 80-120 символов
- Автоматическое форматирование
- Пробелы вокруг операторов и ключевых слов
- Структурные отступы

# Обработка ошибок

ИТМО

- Ошибки надо обрабатывать, а не игнорировать
- Использовать исключения
- Информативные сообщения и логирование
- Не возвращать null
- Не передавать null в методы

# Тестирование

ИТМО

- Код должен работать
- Тесты должны выполняться
  - TDD - test-driven development

# Рефакторинг — процесс улучшения кода

- Код требует периодических дополнений и улучшений
- Увидел проблему - исправил и сделал лучше
- Использование автоматизированных средств в IDE
- Обязательное условие - не испортить то, что работало

# ИИ в разработке: новая реальность

- ИИ-инструменты меняют процесс разработки ПО
- ИИ ускоряет и упрощает кодирование
- Как применять ИИ эффективно и безопасно?
- Как осознанно использовать ИИ на этапе обучения?

# Плюсы ИИ для разработки

- Генерация и автодополнение кода экономит время
- Быстрое создание шаблонного кода
- Меньше времени на поиск ошибок и статический анализ
- Генерация документации и комментариев

# Плюсы ИИ для обучения

- Развитие навыка точного описания проблемы
  - стали появляться курсы по Prompt Engineering
- Быстрые ответы на вопросы (на родном языке)
- Понятное объяснение концепций
- Генерация примеров кода и практических задач

# Минусы ИИ для обучения

- Сдача работ без получения знаний
- Код есть - понимания нет
- Отстутствие опыта решения проблем
- Плагиат и академическая этика
- Зависимость от ИИ

# Корректное использования ИИ

- ИИ - инструмент и помощник
  - может объяснить концепцию
  - может проверить написанный код и дать советы
  - может сгенерировать код по точному и полному промпту
  - может показать примеры кода

# ИИ надо проверять

- В основе - LLM (большие языковые модели)
- ИИ не пишет код, а предсказывает следующий символ
- ИИ не обязательно выдает хороший и правильный код
- Галлюцинации ИИ
  - генерация убедительных неверных ответов
  - сложно отличить правильный ответ от ошибочного

# Что не стоит доверять ИИ?

- Генерация критического и безопасного кода
- Конфиденциальная информация
- Архитектурные решения
- Проверять факты и код

# Инструменты ИИ

- GitHub Copilot
- Intellij AI Assistant
- Универсальные LLM
  - ChatGPT, Gemini
  - Deepseek, Qwen
  - Gigachat, Алиса
- Специальные LLM (Codex, AlphaCode)

# Практическое задание в лабе 3-4

- Напишите метод самостоятельно
- Попросите ИИ-ассистентов написать тот же метод
- Сравните решения ИИ и свое, проанализируйте результат и сделайте выводы
- Промпты, выводы и код - в отчет

# Не все в Java очевидно

ИТМО

- Много примеров "странно" работающего кода
  - так и было задумано
  - так часто пишут
  - полезно знать, чтобы так не делать

# Четное или нечетное?

ИТМО

```
public isOdd(int x) {
  return x % 2 == 1;
}
```

# Четное или нечетное?

ИТМО

```
public isOdd(int x) {
  return x % 2 == 1;
}
```

```
public isOdd(int x) {
  return x % 2 != 0;
}
```

# Равно или не равно?

ИТМО

```
System.out.println((2.00 - 1.10) == 0.90);
```

# Равно или не равно?

ИТМО

```
System.out.println((2.00 - 1.10) == 0.90);
```

```
System.out.println((200 - 110) == 90);

var b1 = new BigDecimal("2.00");
var b2 = new BigDecimal("1.10");
System.out.println(b1.subtract(b2));
```

# Как правильно делить?

ИТМО

```
long milliseconds = 24*60*60*1000;
long microseconds = 24*60*60*1000*1000;

System.out.println(milliseconds / microseconds);
```

# Как правильно делить?

ИТМО

```
long milliseconds = 24*60*60*1000;
long microseconds = 24*60*60*1000*1000;

System.out.println(milliseconds / microseconds);
```

```
long milliseconds = 24L*60*60*1000;
long microseconds = 24L*60*60*1000*1000;
```

# 66666

ИТМО

```
System.out.println(12345 + 54321);
```

# 66666

```
System.out.println(12345 + 5432l);
```

```
System.out.println(12345 + 5432L);
```

# 66666

ИТМО

```
System.out.println(12345 + 54321);
```

# 66666 / 55555

ИТМО

```
System.out.println(12345 + 54321);
```

```
System.out.println(01234 + 54321);
```

# 55555

ИТМО

```
var sum = BigInteger.ZERO;
for (var five = "5"; !five.equals("500000"); five += "0") {
  sum.add(new BigInteger(five));
}
System.out.println(sum);
```

# 55555

ИТМО

```
var sum = BigInteger.ZERO;
for (var five = "5"; !five.equals("500000"); five += "0") {
  sum.add(new BigInteger(five));
}
System.out.println(sum);
```

```
var sum = BigInteger.ZERO;
for (var five = "5"; !five.equals("500000"); five += "0") {
  sum = sum.add(new BigInteger(five));
}
System.out.println(sum);
```

# i == 0 ?

ИТМО

```
var x = 'X';

System.out.println(true ? x : 0);
```

# i == 0 ?

ИТМО

```
var x = 'X';

System.out.println(true ? x : 0);
```

```
var x = 'X';
var i = 0;
System.out.println(true ? x : i);
```

# HaHa

ИТМО

```
System.out.println('H' + 'a' + "H" + "a");
```

# * 1.5

ИТМО

```
var x = 1000;
x *= 3 / 2;
System.out.println(x);
```

# * 1.5

ИТМО

```
var x = 1000;
x *= 3 / 2;
System.out.println(x);
```

```
var x = 1000;
x = x * 3 / 2;
System.out.println(x);
```

```
public static void main(String[] args) {

  for(var a : args) {
    System.out.println(a);
  }
}
```

```
public static void main(String[] args) {
  OK, Google!
  for(var a : args) {
    System.out.println(a);
  }
}
```

ИТМО

```
public static void main(String[] args) {
  OK, Google!
  for(var a : args) {
    System.out.println(a);
  }
}
```

```
public static void main(String[] args) {
  http://www.java.net
  for(var a : args) {
    System.out.println(a);
  }
}
```

# Что это ?

```
\u0070\u0075\u0062\u006c\u0069\u0063\u0020
\u0063\u006c\u0061\u0073\u0073\u0020\u0048
\u007b\u0070\u0075\u0062\u006c\u0069\u0063
\u0020\u0073\u0074\u0061\u0074\u0069\u0063
\u0020\u0076\u006f\u0069\u0064\u0020\u0020
\u006d\u0061\u0069\u006e\u0020\u0028\u0020
\u0020\u0053\u0074\u0072\u0069\u006e\u0067
\u002e\u002e\u002e\u0020\u0061\u0029\u007b
\u0053\u0079\u0073\u0074\u0065\u006d\u002e
\u0020\u0020\u0020\u006f\u0075\u0074\u002e
\u0070\u0072\u0069\u006e\u0074\u0028\u0020
\u0022\u0048\u0069\u0021\u005c\u006e\u0022
\u0029\u003b\u0020\u007d\u0020\u007d\u0020
```

# null

ИТМО

```
public class Null {
   public static void hello() {
      System.out.println("Hello world!");
   }
}
```

# null

ИТМО

```
public class Null {
  public static void hello() {
    System.out.println("Hello world!");
  }
}
```

```
Null x = null;

x.hello();
```

# null-2

ИТМО

```
String str = null;
String result = "s = " + str != null ? str : "0";
System.out.println(result);
```

# null-2

ИТМО

```
String str = null;
String result = "s = " + str != null ? str : "0";
System.out.println(result);
```

```
String str = null;
String result = "s = " + (str != null ? str : "0");
System.out.println(result);
```

# 0x0101

ИТМО

```
int lower = 0x01;
int higher = 0x01;
System.out.println(higher << 8 + lower);
```

# 0x0101

ИТМО

```
int lower = 0x01;
int higher = 0x01;
System.out.println(higher << 8 + lower);
```

```
int lower = 0xff;
int higher = 0x55;
System.out.println((higher << 8) + lower);
System.out.println(higher << 8 | lower);
```

# varargs

ИТМО

```
void printArray(Object... array) {
    for (var x : array) {
        System.out.println(x);
    }
}
```

```
Object[] a = {"Hello", "World"};
printArray(a);

printArray("Hello", "World");
```

# varargs

ИТМО

```
void printArray(Object... array) {
    for (var x : array) {
        System.out.println(x);
    }
}
```

```
int[] i = {1,2,3};
printArray(i);

printArray(1,2,3);
```

# varargs

ИТМО

```
void printArray(Object... array) {
    for (var x : array) {
        System.out.println(x);
    }
}
```

```
printArray(null, null, null);

printArray(null);
```

# Exception

ИТМО

```
int check(int positive) {
  if (positive < 0) {
    new IllegalArgumentException("Negative");
  }
  return positive;
}
```

```
System.out.println(check(1));

System.out.println(check(-1));
```

# Exception

ИТМО

```
int check(int positive) {
  if (positive < 0) {
    throw new IllegalArgumentException("Negative");
  }
  return positive;
}
```

```
System.out.println(check(1));

System.out.println(check(-1));
```

# Math.abs

ИТМО

```
void check(int positive) {
  positive = Math.abs(positive);
  if (positive < 0) {
    throw new IllegalArgumentException("Negative");
  }
}
```

```
Random rnd = new Random(1010);
while (true) {
  check(rnd.nextInt());
}
```

# Math.abs

ИТМО

```
void check(int positive) {
  positive = Math.abs(positive);
  if (positive < 0) {
    throw new IllegalArgumentException("Negative");
  }
}
```

```
System.out.println(Integer.MIN_VALUE);

System.out.println(- Integer.MIN_VALUE);
System.out.println(Math.abs(Integer.MIN_VALUE));
```

# Nested loop copy-paste

ИТМО

```
int sum = 0;
for (var i = 0; i < 100; i++) {

}
```

# Nested loop copy-paste

```
int sum = 0;
for (var i = 0; i < 100; i++) {

}
```

Copy

# Nested loop copy-paste

```
int sum = 0;
for (var i = 0; i < 100; i++) {
for (var i = 0; i < 100; i++) {

}
}
```

Paste

# Nested loop copy-paste

```
int sum = 0;
for (var i = 0; i < 100; i++) {
for (var j = 0; i < 100; i++) {

}
}
```

j

# Nested loop copy-paste

ИТМО

```
int sum = 0;
for (var i = 0; i < 100; i++) {
for (var j = 0; i < 100; i++) {
  sum += i*j;
}
}
```

# Nested loop copy-paste

```
int sum = 0;
for (var i = 0; i < 100; i++) {
  for (var j = 0; i < 100; i++) {
    sum += i*j;
  }
}
```

Tab

# Nested loop copy-paste

ИТМО

```
int sum = 0;
for (var i = 0; i < 100; i++) {
  for (var j = 0; i < 100; i++) {
    sum += i*j;
  }
}
```

# Nested loop copy-paste

ИТМО

```
int sum = 0;
for (var i = 0; i < 100; i++) {
  for (var j = 0; j < 100; j++) {
    sum += i*j;
  }
}
```

# Вопросы?