Java Core: Подробный обзор

1. Основы Java

Переменные

Переменные в Java - это именованные ячейки памяти для хранения данных. Каждая переменная имеет:

- **Тип данных** (примитивный или ссылочный)
- **Имя** (идентификатор)
- **Значение**

```
int age = 25; // примитивный тип
String name = "John"; // ссылочный тип
```

Типы переменных:

1. **Локальные** - объявлены в методах, конструкторах или блоках
2. **Экземпляра (поля класса)** - объявлены в классе, но вне методов
3. **Статические** - объявлены с модификатором static

Операторы

- **Арифметические**: `+`, `-`, `*`, `/`, `%`, `++`, `--`
- **Операторы сравнения**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Логические**: `&&`, `||`, `!`
- **Присваивания**: `=`, `+=`, `-=`, `*=`, `/=`
- **Тернарный**: `условие ? выражение1 : выражение2`
- **Оператор instanceof**: проверяет тип объекта

Циклы

1. **for**:

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

2. **while**:

```
while (condition) {
    // код
}
```

3. **do-while**:

```
do {
    // код
} while (condition);
```

4. **Улучшенный for (for-each)**:

```
for (String item : collection) {
    System.out.println(item);
}
```

Условия

1. **if-else**:

```
if (condition) {
    // код
} else if (anotherCondition) {
    // код
} else {
    // код
}
```

2. **switch-case** (начиная с Java 14 поддерживает выражения):

```
switch (variable) {
    case 1 -> System.out.println("One");
    case 2 -> System.out.println("Two");
    default -> System.out.println("Other");
}
```

2. ООП в Java

Основные принципы:

1. **Классы и объекты**:
    - Класс - шаблон для создания объектов
    - Объект - экземпляр класса

```
class Person {
    String name;
    int age;

    void speak() {
        System.out.println("My name is " + name);
    }
}

Person person = new Person();
person.name = "John";
```

2. **Наследование**:
    - Класс-потомок наследует поля и методы родителя
    - Ключевое слово `extends`

```
class Employee extends Person {
    String position;
}
```

3. **Полиморфизм**:
    - Возможность объектов вести себя по-разному в зависимости от контекста
    - Перегрузка методов (overloading) - методы с одним именем, но разными параметрами
    - Переопределение методов (overriding) - замена реализации метода родителя

```
class Animal {
    void makeSound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Bark");
    }

    // Перегрузка
    void makeSound(int times) {
        for (int i = 0; i < times; i++) {
            System.out.println("Bark");
        }
    }
}
```

4. **Инкапсуляция**:
    - Сокрытие внутренней реализации
    - Использование модификаторов доступа (private, protected, public)
    - Геттеры и сеттеры

```
class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
}
```

5. **Абстракция**:

- Выделение существенных характеристик объекта
- Абстрактные классы и интерфейсы

```
abstract class Shape {
    abstract double calculateArea();
}

class Circle extends Shape {
    double radius;

    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

3. Исключения

Иерархия исключений:

- **Throwable**
  - **Error** (непроверяемые, например, OutOfMemoryError)
  - **Exception**
    - **RuntimeException** (непроверяемые, например, NullPointerException)
    - Другие (проверяемые, например, IOException)

Обработка исключений:

1. **try-catch-finally**:

```
try {
    // код, который может вызвать исключение
} catch (IOException e) {
    // обработка исключения
    System.err.println("Error: " + e.getMessage());
} finally {
    // код, который выполнится в любом случае
    System.out.println("Cleanup");
}
```

2. **throws**:

```
public void readFile() throws IOException {
    // код, который может выбросить IOException
}
```

3. **Кастомные исключения**:

```
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

```
// Использование
void withdraw(double amount) throws InsufficientFundsException {
    if (amount > balance) {
        throw new InsufficientFundsException("Not enough money");
    }
    balance -= amount;
}
```

4. Интерфейсы и абстрактные классы

Интерфейсы:

- Контракт, который класс должен реализовать
- До Java 8 содержали только абстрактные методы
- Начиная с Java 8 могут содержать:
    - default-методы (с реализацией)
    - static-методы (с реализацией)
- Начиная с Java 9 могут содержать private-методы

```
interface Drawable {
    void draw(); // абстрактный метод

    default void print() { // default-метод
        System.out.println("Printing...");
    }

    static void show() { // static-метод
        System.out.println("Showing...");
    }
}
```

Абстрактные классы:

- Не могут быть инстанциированы
- Могут содержать абстрактные и обычные методы
- Могут содержать поля с любым модификатором доступа

```
abstract class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    abstract void makeSound();

    public void sleep() {
        System.out.println(name + " is sleeping");
```

```
    }
}
```

Отличия:

Характеристика	Интерфейс	Абстрактный класс
Реализация методов	Только default и static	Любые методы
Поля	Только public static final	Любые поля
Наследование	Множественное	Одиночное
Конструкторы	Не может иметь	Может иметь
Модификаторы доступа	Все методы public по умолчанию	Любые модификаторы

5. Generics (Обобщения)

Позволяют создавать классы, интерфейсы и методы с параметрами типов.

Обобщённые классы:

```
class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}

// Использование
Box<String> stringBox = new Box<>();
stringBox.setContent("Hello");
```

Обобщённые методы:

```
public <T> void printArray(T[] array) {
    for (T item : array) {
        System.out.println(item);
    }
}

// Использование
Integer[] intArray = {1, 2, 3};
printArray(intArray);
```

Ограничения (bounded type parameters):

```
class NumberBox<T extends Number> {
    private T number;

    public double getSquare() {
```

```
        return number.doubleValue() * number.doubleValue();
    }
}
```

Wildcards:

- `<?>` - любой тип
- `<? extends T>` - T или его подтип
- `<? super T>` - T или его супертип

```
public void process(List<? extends Number> list) {
    // принимает List<Number>, List<Integer>, List<Double> и т.д.
}
```

6. Stream API

Stream API предоставляет функциональный подход к обработке коллекций.

Основные операции:

1. **Промежуточные (intermediate)**:
    - `filter(Predicate<T>)` - фильтрация элементов
    - `map(Function<T, R>)` - преобразование элементов
    - `sorted()` - сортировка
    - `distinct()` - удаление дубликатов
    - `limit(long)` - ограничение количества элементов
    - `skip(long)` - пропуск первых элементов
2. **Терминальные (terminal)**:
    - `forEach(Consumer<T>)` - выполнение действия для каждого элемента
    - `collect(Collector)` - сбор элементов в коллекцию
    - `reduce(BinaryOperator<T>)` - свертка элементов
    - `count()` - подсчет элементов
    - `anyMatch(Predicate<T>)` / `allMatch()` / `noneMatch()` - проверка условий

Примеры:

```
List<String> names = Arrays.asList("John", "Alice", "Bob", "Anna");

// Фильтрация и преобразование
List<String> result = names.stream()
    .filter(name -> name.startsWith("A"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());

// Сумма длин всех имен
```

```
int totalLength = names.stream()
    .mapToInt(String::length)
    .sum();

// Группировка
Map<Integer, List<String>> groupedByNameLength = names.stream()
    .collect(Collectors.groupingBy(String::length));
```

Параллельные стримы:

```
List<String> result = names.parallelStream()
    .filter(name -> name.length() > 3)
    .collect(Collectors.toList());
```

7. Функциональные интерфейсы

Функциональный интерфейс - интерфейс с одним абстрактным методом. Аннотация `@FunctionalInterface`.

Основные функциональные интерфейсы:

3. **Predicate<T>**:
    - Проверяет условие
    - Метод: `boolean test(T t)`

```
Predicate<String> isLong = s -> s.length() > 10;
boolean result = isLong.test("Hello world"); // true
```

4. **Function<T, R>**:
    - Преобразует T в R
    - Метод: `R apply(T t)`

```
Function<String, Integer> lengthMapper = String::length;
int len = lengthMapper.apply("Java"); // 4
```

5. **Consumer<T>**:
    - Выполняет действие с T
    - Метод: `void accept(T t)`

```
Consumer<String> printer = System.out::println;
printer.accept("Hello"); // выводит "Hello"
```

6. **Supplier<T>**:
    - Поставляет значение типа T
    - Метод: `T get()`

```
Supplier<Double> randomSupplier = Math::random;
double value = randomSupplier.get(); // случайное число
```

7. **Другие**:

- `UnaryOperator<T>` (Function<T, T>)
- `BinaryOperator<T>` (Function<T, T, T>)
- `BiFunction<T, U, R>`
- `BiPredicate<T, U>`
- `BiConsumer<T, U>`

8. Модульность (Java 9+)

Модульность (Project Jigsaw) - система модулей для Java, представленная в Java 9.

Основные понятия:

- **Модуль** - набор пакетов с описанием зависимостей
- **module-info.java** - дескриптор модуля

Структура module-info.java:

```
module com.example.myapp {
    requires java.base; // неявно добавляется
    requires java.sql;
    requires transitive com.example.utils; // транзитивная зависимость

    exports com.example.myapp.api;
    exports com.example.myapp.model to com.example.framework;

    opens com.example.myapp.internal to com.example.test;

    uses com.example.spi.ServiceProvider;
    provides com.example.spi.ServiceProvider
        with com.example.myapp.MyServiceProvider;
}
```

Ключевые директивы:

1. **requires** - объявляет зависимость от другого модуля
2. **requires transitive** - делает зависимость транзитивной
3. **exports** - делает пакет доступным для других модулей
4. **exports ... to** - ограниченный экспорт
5. **opens** - открывает пакет для рефлексии (обычно для фреймворков)
6. **opens ... to** - ограниченное открытие
7. **uses** - объявляет использование сервиса (SPI)
8. **provides ... with** - объявляет реализацию сервиса

Преимущества модульности:

- Улучшенная инкапсуляция (strong encapsulation)

- Явное объявление зависимостей
- Уменьшение размера приложений (jlink)
- Улучшенная производительность и безопасность
- Более четкая структура больших приложений

Пример модульного приложения:

```
myapp/
├── src/
│   ├── com.example.myapp/
│   │   ├── com/example/myapp/
│   │   │   ├── Main.java
│   │   │   └── ...
│   │   └── module-info.java
├── lib/
│   └── ...
└── build/
```

```
// module-info.java
module com.example.myapp {
    requires java.logging;
    requires com.example.utils;
    exports com.example.myapp.api;
}
```

```
// Main.java
package com.example.myapp;

import java.util.logging.Logger;
import com.example.utils.StringUtils;

public class Main {
    private static final Logger LOG = Logger.getLogger(Main.class.getName());

    public static void main(String[] args) {
        LOG.info("Application started");
        String result = StringUtils.reverse("Hello");
        System.out.println(result);
    }
}
```