# Программирование на языке Java

# Программа курса

ООП в Java:
- Классы и объекты.
- Наследование, полиморфизм, инкапсуляция.
- Абстрактные классы и методы.

Основы Java:
- Синтаксис и основные конструкции языка.
- Типы данных, переменные, операторы.
- Условные операторы, циклы.
- Массивы, строки.

Коллекции:
- Списки, множества, карты.

Исключения:
- Обработка исключений.

Работа с файлами:
- Чтение и запись файлов.

JDBC:
- Подключение к базе данных.

Многопоточность:
- Основы параллельного программирования.

Практика:
- Решение задач и выполнение лабораторных работ.

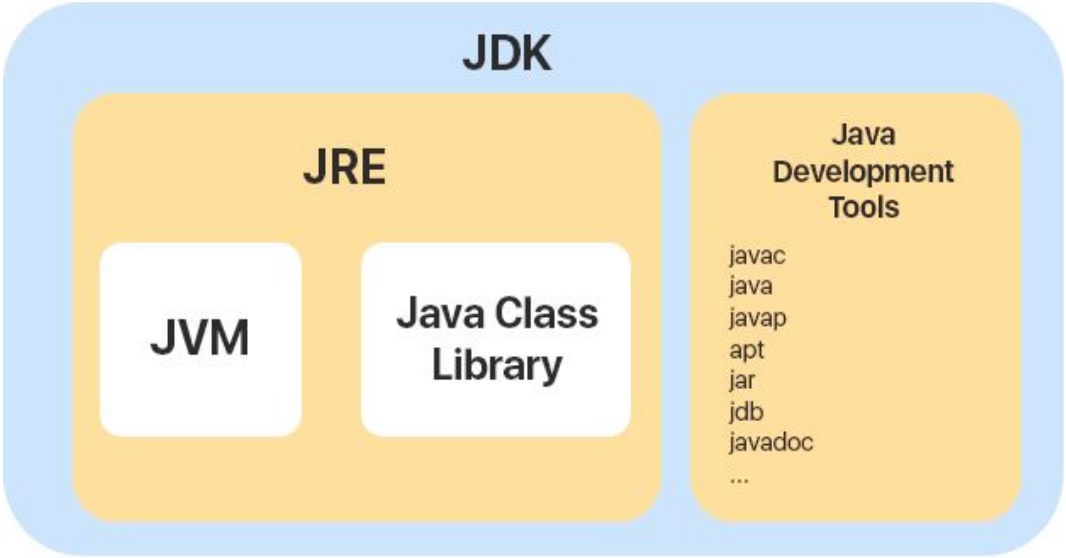
# «Write Once, Run Anywhere»

# История Java

- Java 1.0 появилась в 1995 в Sun Microsystems
- Java 5 (1.5) появилась в 2004 году:
  - Enum, Аннотации, Generics, Autoboxing/Unboxing
- Java 7 (2011 год)
  -
- Java 8 (2014)
  - Лямбда- выражения, DateTime API, Streams
- ....
- Java 24 (03.2025) - актуальная на данный момент
- Java 25 (09.2025) - coming soon

# JRE / JDK / JVM

JDK — комплект для разработки, включающий инструменты для компиляции и тестирования.
JRE — это среда выполнения для запуска Java-приложений.
JVM — виртуальная машина, которая обеспечивает выполнение Java-программ на разных платформах.



# Роль JVM

# Спецификация и реализации JVM

Спецификация* написана для полного документирования архитектуры виртуальной машины Java.

Основные реализации:

- OpenJDK
- Oracle
- Amazon
- GraalVM

*https://docs.oracle.com/javase/specs/jvms/se21/html/

# Состав JVM

- Загрузчик классов
- Области память
  - Heap
  - Stack
  - Metaspace
- Интерпретатор
- JIT-компилятор
- Сборщик мусора

# Java - язык программирования

Объектно-ориентированный ЯП со статической типизацией

# Объектно-ориентированное программирование

методология программирования, основанная на представлении программы в виде совокупности объектов, которые взаимодействуют друг с другом

- Класс
- Интерфейс
- Услуги

# Наследование

Возможность создания новых классов на основе существующих с автоматическим наследованием всех или некоторых методов и свойств базовых классов.

```java
public class Car {
    private String color; // Цвет автомобиля
    private String brand; // Марка автомобиля
    private String model; // Модель автомобиля

    public Car(String color, String brand, String model) {
        this.color = color;
        this.brand = brand;
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public String getBrand() {
        return brand;
    }

    public String getModel() {
        return model;
    }
}
```

```java
public class Sedan extends Car{
    private int doors; // Количество дверей
    private String bodyType; // Тип кузова

    public Sedan(String color, String brand, String model) {
        super(color, brand, model);
    }

    public Sedan(String color, String brand, String model, int doors, String bodyType) {
        super(color, brand, model);
        this.doors = doors;
        this.bodyType = bodyType;
    }

}
```

# Инкапсуляция

Все данные и функции, которые нужны для работы объектов, хранятся внутри объектов и их классов

```java
public class Car {
    private String color; // Цвет автомобиля
    private String brand; // Марка автомобиля
    private String model; // Модель автомобиля

    public Car(String color, String brand, String model) {
        this.color = color;
        this.brand = brand;
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public String getBrand() {
        return brand;
    }

    public String getModel() {
        return model;
    }
}
```

# Полиморфизм

Способность объектов иметь различное поведение в зависимости от их типа.

# Абстракция

Принцип абстракции означает, что при проектировании классов и объектов важно концентрироваться на ключевых методах и атрибутах и отказываться от лишних.

```
public class Car {
    private String color; // Цвет автомобиля
    private String brand; // Марка автомобиля
    private String model; // Модель автомобиля

    public Car(String color, String brand, String model) {
        this.color = color;
        this.brand = brand;
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public String getBrand() {
        return brand;
    }

    public String getModel() {
        return model;
    }
}
```

# Плюсы и минусы ООП

Плюсы:

1. В рамках ООП разработчики проще и быстрее создают код. Достаточно один раз написать класс с нужными атрибутами и методами.
2. Проще проверять чужой код, легче изменять код.
3. Легко масштабировать проект.

Минусы:

1. Сложно разобраться в основах.
2. Не подходит для маленьких программ.
3. Тратится больше ресурсов.

# Типы данных Java

- Объекты
- Примитивы

# Примитивные типы данных в Java

| Тип | Размер | Значение по умолчанию | Диапазон |
|---|---|---|---|
| `boolean` | 1 бит | false | true или false |
| `byte` | 8 бит | 0 | -128 до 127 |
| `short` | 16 бит | 0 | -32,768 до 32,767 |
| `int` | 32 бита | 0 | -2^31 до 2^31-1 |
| `long` | 64 бита | 0L | -2^63 до 2^63-1 |
| `float` | 32 бита | 0.0f | примерно ±3.40282347E+38F |
| `double` | 64 бита | 0.0d | примерно ±1.7976931348623157E+308 |
| `char` | 16 бит | '\u0000' | '\u0000' (0) до '\uffff' (65535) |

# Класс Object

Корневой класс в Java.

Методы:

- `public String toString()`
- `public native int hashCode()`
- `public boolean equals(Object obj)`
- `public final native Class getClass()`
- `public final native void notify()`
- `public final native void notifyAll()`
- `public final native void wait(long timeout)`
- `public final void wait(long timeout, intnanos)`
- `public final void wait()`
- `protected void finalize()`
- `protected native Object clone()`

# toString()

позволяет получить текстовое описание любого объекта

# Контракт equals

- Рефлексивность (Reflexivity)
  - **x.equals(x) == true**;
- Симметричность (Symmetry)
  - Если **x.equals(y) == true**, то **y.equals(x) == true**;
- Транзитивность (Transitivity)
  - Если **x.equals(y) == true && y.equals(z) == true**, то **x.equals(z) == true**
- Непротиворечивость (Consistency)
  - Результат должен быть неизменным, пока объекты не изменяются
- Сравнение с null (Non-nullity)
  - **x.equals(null) == false;**
- Согласованность с hashCode()
  - Если **x.equals(y) == true**, то **x.hashCode() == y.hashCode()**

# hashCode()

Взаимосвязь методов equals() и hashCode()

- Контракт между equals() и hashCode() гласит, что если два объекта равны, то они должны иметь одинаковый хэш-код.
  Обратное неверно!

- Пример нарушения контракта: если вы переопределите equals(), но не переопределите hashCode(), то объекты, которые кажутся равными с точки зрения equals(), могут иметь разные хэш-коды. Это может привести к некорректному поведению при работе с хэш-структурами (например, HashMap или HashSet).

# Управляющие конструкции

- if - else
- while / do-while
- for
- foreach
- return
- break
- continue

# Операторы

- присваивание (a = 2)
- математические (+, -, *, /)
- унарные (x = a * -b;)
- инкремент/декремент (i++)
- сравнение (==, <=, >=, x.equals(y) )
- логические (&, &&, |, ||, !)
- поразрядные (&, |, ^, ~)
- сдвиг (<<, >>)
- тернарный оператор (логическое-условие ? выражение1 : выражение2)
- приведение (short s = (short) 1)

# Таблица истинности

| A | B | A \| B | A & B | A ^ B | !A |
|---|---|---|---|---|---|
| false | false | false | false | false | true |
| true | false | true | false | true | false |
| false | true | true | false | true | true |
| true | true | true | true | false | false |

# Классы обертки и автоупаковка

boolean -> Boolean

byte -> Byte

short -> Short

int -> Integer

long -> Long

float -> Float

double -> Double

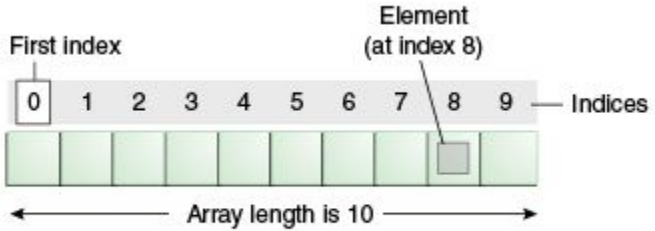
char -> Character

# Массивы

Контейнеры, хранящие фиксированный набор объектов одного типа.

Наиболее эффективный контейнер.

Способен хранить примитивные типы.




An array of 10 elements.

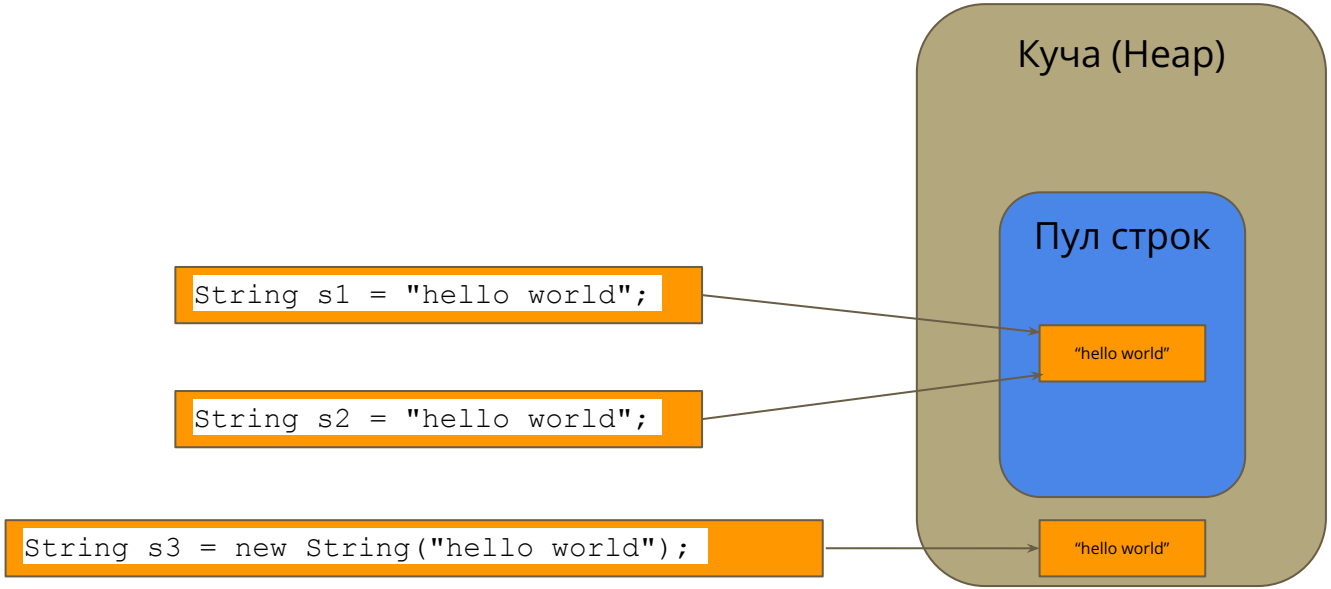
# Строки

Представляет собой последовательность символов (char).

Неизменяемый объект.

Внутри хранится в виде массива байт.

# Пул строк


Куча (Heap)
Пул строк
String s1 = "hello world";
String s2 = "hello world";
String s3 = new String("hello world");
“hello world”
“hello world”

# Основные методы String

- int length()
- boolean isEmpty()
- char charAt(int index)
- byte[] getBytes()
- boolean equals(Object anObject)
- boolean startsWith(String prefix)
- boolean endsWith(String suffix)
- int indexOf(int ch)
- int lastIndexOf(int ch)
- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)
- String[] split(String regex)
- String toLowerCase()
- String toUpperCase()
- String trim()
- static String format(String format, Object... args)

# StringBuilder и StringBuffer

Изменяемый объект строк.

Эффективнее при склейке большого количества строк.

StringBuffer - ориентирован на многопоточную работу.

# Formatter

| Плейсхолдер | Тип данных | Компилируемая строка | Итоговый результат |
|---|---|---|---|
| **%s** | String | String.format("Привет, %s", "мир") | Привет, мир |
| **%d** | int/long | String.format(«Число: %d", 21) | Число: 21 |
| **%f** | float/double | String.format("Сумма: %.2f", 25.349) | Сумма: 25.35 |
| **%b** | boolean | String.format("Флаг: %b", true) | Флаг: true |
| **%c** | char | String.format("Буква: %c", 'J') | Буква: J |
| **%x/%X** | int/long | String.format("HEX: %x", 255) | HEX: ff |
| **%e** | float/double | String.format("Экспоненциально: %e", 12345.678) | Экспоненциально: 1.234568e+04 |
| **%t** | Date/Calendar | String.format("Дата: %tY-%tm-%td", new Date()) | Дата: 2025-09-23 |
| **%%** | - | String.format("Готово: 100%%") | Готово: 100% |

# RegEx

шаблон для поиска строки в тексте

| Метасимвол | Описание | Пример использования |
|---|---|---|
| . | Любой одиночный символ | a.b→ "acb", "arb" |
| ^ | Начало строки | ^abc→ "abc", не "xabc" |
| $ | Конец строки | abc$→ "abc", не "abcd" |
| * | 0 и более повторений | ab*→ "a", "ab", "abb" |
| + | 1 и более повторений | ab+→ "ab", "abb" |
| ? | 0 или 1 повторение | ab?→ "a", "ab" |
| {n} | Ровно n повторений | a{3}→ "aaa" |
| {n,} | n и более повторений | a{2,}→ "aa", "aaa" |
| {n,m} | От n до m повторений | a{1,3}→ "a", "aa", "aaa" |
| [] | Один из символов в скобках | [aeiou]→ "a", "e"... |
| () | Группа | (abc)+→ "abcabc" |
| \\d | Цифра | \\d→ "3" |
| \\D | Нецифра | \\D→ "A", "-" |
| \\w | Буква/цифра/подчеркивание | \\w→ "A", "7", "_" |
| \\W | Не буква/цифра/подчеркивание | \\W→ "!" |
| \\s | Пробел | \\s→ " ", "\t" |
| \\S | Не пробел | \\S→ "a", "1" |

# Класс

Шаблон, позволяющий описать в программе объект, его свойства (атрибуты или поля класса) и поведение (методы класса).

```
class MyClass {}
```

# Поле класса

переменная, описывающая какое-либо из свойств данного класса.

```
class MyClass {
    int a;
}
```

# Конструктор

```
class MyClass {
    int a;
    MyClass(int a) {
        this.a = a;
    }
}
```

# Методы класса

блок кода, описывающий функции, которые способен выполнять экземпляр данного класса.

```
возвращаемыйТип названиеМетода(аргументы) {
    //code
    return значение;
}
```

```
class MyClass {
    void doSmth() {}
}
```

# Абстрактный класс

Класс с одним или несколькими неопределенными методами

# Интерфейс

Используется для описания методов без реализации (в Java 8 добавили возможность создать default метод с реализацией).

Позволяют реализовать множественное наследование.

# Static

Модификатор, применяемый к полю, блоку, методу или внутреннему классу.

Указывает на привязку к классу, а не объекту.

# Final

Модификатор, применяемый к переменной, полю, методу или классу.

Поле / переменная - нельзя присвоить новое значение (так задаются константы);
Метод - нельзя переопределить в наследние (~~@Override~~)
Класс - нельзя унаследовать

# Модификаторы доступа

Помогают реализовать принцип инкапсуляции - сокрытие внутреннего устройства объекта и защита его от неправильного использования.

private позволяет обращаться к элементам класса только внутри класса

default (отсутствует) дает доступ к элементам класса только внутри пакета

protected дает доступ к элементам класса только внутри пакета и наследникам

public делает элементы класса доступными везде

# Внутренний класс (Inner class)

Класс внутри класса
Привязан к экземпляру внешнего класса

```
class OuterClass {
    class InnerClass {
    }
}

OuterClass os = new OuterClass();
OuterClass.InnerClass is = os.new InnerClass();
```

# Вложенный класс (Nested class)

Обычный класс, который объявлен внутри другого класса

Не привязан к внешнему классу, но имеет доступ к его static полям и методам.

```
class OuterClass {
    static class NestedClass {
    }
}
```

```
OuterClass.InnerClass is = new OuterClass.InnerClass();
```

# Локальный класс(Local class)

Ограничен областью видимости метода.

```
class OuterClass {
        void someMethod() {
                class LocalClass {
                }
                LocalClass ls = new LocalClass();
        }
}
```

# Анонимный класс

Объединяет создание переменной с описанием класса.

Подход для создания объектов для одноразового использования.

```
interface Anonymus{
   void doSmth();
}

class OuterClass {
   AnonymusClass getAnObject() {
         return new AnonymusClass() {
               @Override
               public void doSmth() {
                     // some logic here
               }
         };
   }
}
```

# Enum

Тип данных, который представляет собой набор заранее определенных значений.

Использует, когда нужно ограничить значения переменной конкретным набором.

```
enum DaysOfWeek {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
 SUNDAY
}
```

# Функциональный интерфейс

Интерфейс, который содержит только один абстрактный метод.

```
@FunctionalInterface
public interface Runnable {
   void run();
}
```

# Лямбда-выражения

По сути представляют собой анонимный метод.

Выводятся компилятором в один из функциональных интерфейсов.

```
(parameter list) -> lambda body
```

```
() -> System.out.println("hello");
```

# Обобщения (Generics)

Термин обобщения оз начает параметризованные типы.

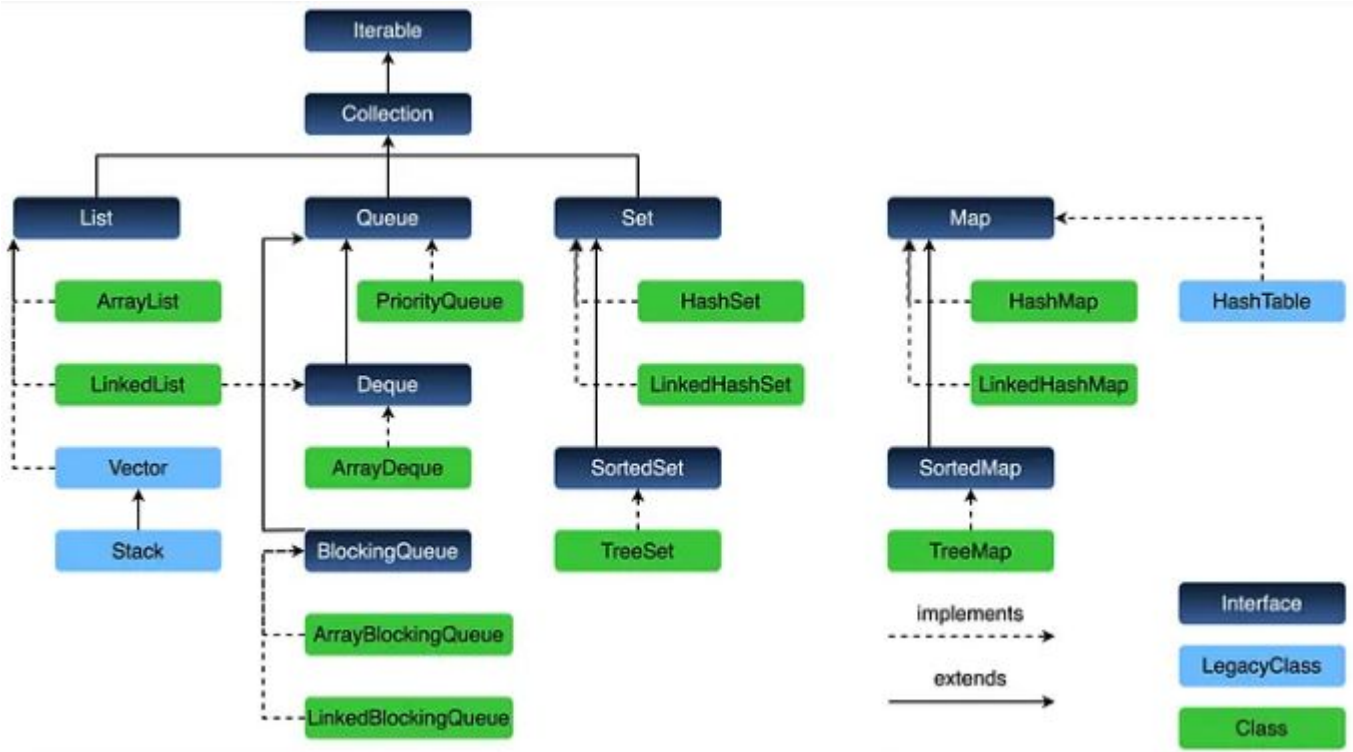
Обеспечивает типобезопасность при создании классов, интерфейсов или методов, работающих с различными типами данных.

```
public <T> T[] toArray(T[] a) {
  // some logic here
  return a;
}
```

# Предопределенные функциональные интерфейсы

| | |
|---|---|
| UnaryOperator<T> | Применяет унарную операцию к объекту типа T и возвращает результат типа T. |
| BinaryOperator<T> | Применяет операцию к двум объектам типа T и возвращает результат типа T. |
| Consumer<T> | Применяет операцию к объекту типа T. |
| Function<T , R> | Применяет операцию к объекту типа T и возвращает в качестве результата объект типа R. |
| Predicate<T> | Применяет операцию к объекту типа T и возвращает в качестве результата объект типа Boolean |

# Коллекции / Collections Framework



# Iterable

```
Iterator<T> iterator();
default void forEach(Consumer<? super T> action);
default Spliterator<T> spliterator();
```

# Collection

Интерфейс, являющийся основой всего Collections Framework.

Должен быть реализован любым классов, определяющим коллекцию.

```
int size();
boolean isEmpty();
boolean contains(Object o);
boolean containsAll(Collection<?> c);
Iterator<E> iterator();
Object[] toArray();
<T> T[] toArray(T[] a);
boolean add(E e);
boolean addAll(Collection<? extends E> c);
boolean remove(Object o);
boolean removeAll(Collection<?> c);
default boolean removeIf(Predicate<? super E> filter);
boolean retainAll(Collection<?> c);
void clear();
boolean equals(Object o);
int hashCode();
default Spliterator<E> spliterator();
default Stream<E> stream();
Stream<E> parallelStream()
```

# List

Объявляет поведение коллекции, в которой хранится последовательность элементов, т.е они расположены в порядке вставки.

Элементы можно вставлять или получать к ним доступ по позиции в списке, применяя индекс.

Может содержать повторяющиеся элементы.

# Основные методы List

```
void add(int index, E element);
E get(int index);
E set(int index, E element);
E remove(int index);
int indexOf(Object o);
boolean addAll(int index, Collection<? extends E> c);
List<E> subList(int fromIndex, int toIndex);
default void replaceAll(UnaryOperator<E> operator);
default void sort(Comparator<? super E> c);
```

# Set

Представляет собой множество - набор уникальных элементов.

За исключением LinkedHashSet, не сохраняет порядок вставки.

# SortedSet

Расширяет Set и обеспечивает поведение набора, отсортированного в возрастающем порядке.
Принимает на вход Comparator для изменения сортировки.

```
Comparator<? super E> comparator();
SortedSet<E> subSet(E fromElement, E toElement);
SortedSet<E> headSet(E toElement);
SortedSet<E> tailSet(E fromElement);
E first();
E last();
default Spliterator<E> spliterator();
```

# NavigableSet

Расширяет SortedSet и обеспечивает поведение коллекции, которая поддерживает извлечение элементов на основе наиболее точного совпадения с заданным значением или значениями.

```
E lower(E e);
E floor(E e);
E ceiling(E e);
E higher(E e);
E pollFirst();
E pollLast();
NavigableSet<E> descendingSet();
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive);
NavigableSet<E> headSet(E toElement, boolean inclusive);
NavigableSet<E> tailSet(E fromElement, boolean inclusive);
```

# Queue

Расширяет Collection и обеспечивает поведение очереди, которая часто представляет собой список, действующий по принципу "первым пришел - первым обслужен”.

```
boolean add(E e);
boolean offer(E e);
E remove();
E poll();
E element();
E peek();
```

# Deque

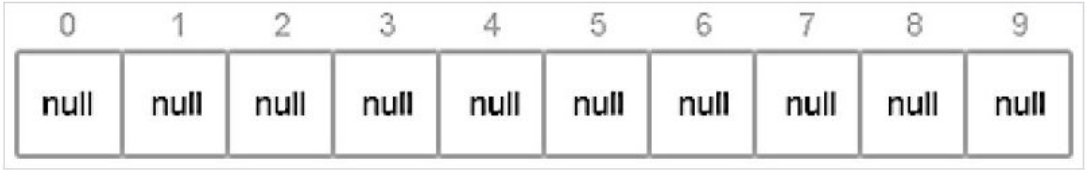
Расширяет Queue и обеспечивает поведение двусторонней очереди.

```
void addFirst(E e);
void addLast(E e);
boolean offerFirst(E e);
boolean offerLast(E e);
E removeFirst();
E removeLast();
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
void push(E e);
E pop();
```

# ArrayList

Динамический массив.

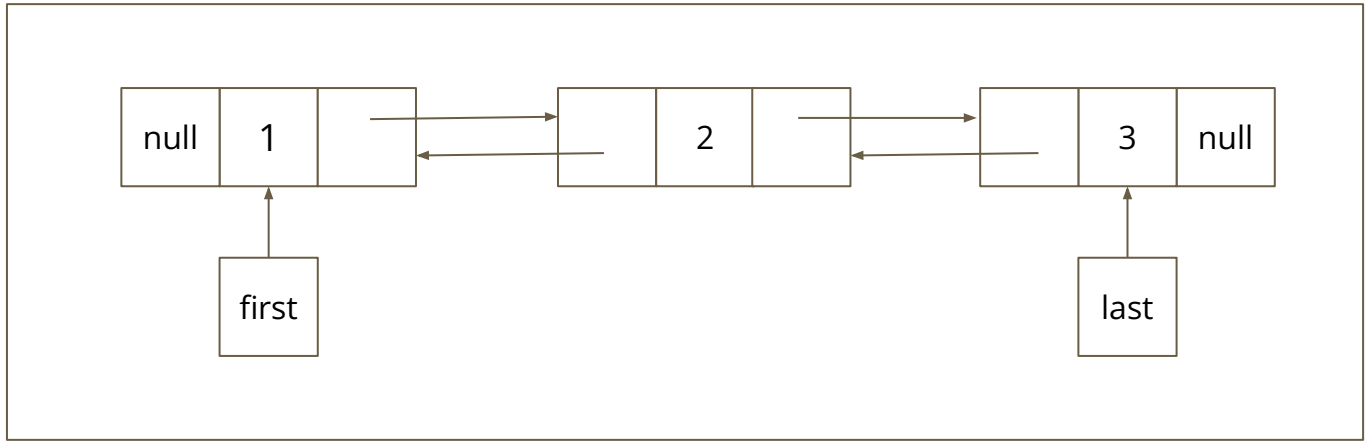
Реализует интерфейс List.



# LinkedList

Представляет собой связный список.

Реализует интерфейсы List, Queue и Deque.



# HashSet

Множество, для хранения данных которого применяется хеш-таблица.

Реализует интерфейс Set.

Не гарантирует порядок элементов.

# LinkedHashSet

Расширяет HashSet.

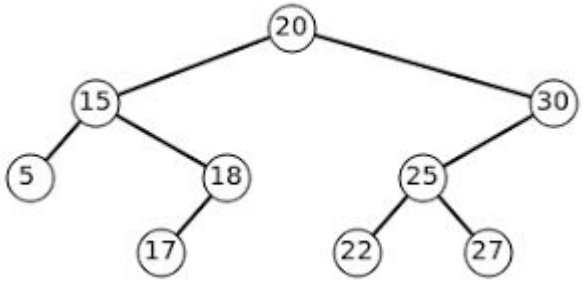
Поддерживает порядок вставки.

# TreeSet

Реализует NavigableSet.

В качестве хранилища использует дерево.

Объекты хранятся в отсортированном порядке по возрастанию.



# PriorityQueue

Реализует Queue.

Хранит данные с приоритетом на основе компаратора.

По умолчанию порядок возрастает.

Нет ограничения по capacity.

# ArrayDeque

Реализует Deque.

Не имеет ограничения по capacity.

# Iterator

Инструмент для итерации по элементам коллекции.

Необходим для доступа через for-each.

```
boolean hasNext();
E next();
default void remove();
default void forEachRemaining(Consumer<? super E> action);
```

# ListIterator

Наследуется от Iterator.

Позволяет сделать двунаправленный обход.

Существует только у коллекций, реализующих List.

# Map

Представляет собой объект, в котором хранятся ассоциации ключ-значение.

Не наследуется от Collection.

# Методы Map

```
int size();
boolean isEmpty();
boolean containsKey(Object key);
boolean containsValue(Object value);
V get(Object key);
V put(K key, V value);
V remove(Object key);
void putAll(Map<? extends K, ? extends V> m);
void clear();
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet();
default V getOrDefault(Object key, V defaultValue);
default void forEach(BiConsumer<? super K, ? super V> action);
default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function);
default V putIfAbsent(K key, V value);
default V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);
default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);
default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction);
```

# SortedMap

Расширяет Map.

Гарантирует, что все элементы хранятся в определенном порядке.

```
SortedMap<K,V> subMap(K fromKey, K toKey);
SortedMap<K,V> headMap(K toKey);
SortedMap<K,V> tailMap(K fromKey);
K firstKey();
K lastKey();
default SortedMap<K, V> reversed();
```

# NavigableMap

Расширяет SortedMap.

Обеспечивает поведение, которое поддерживает извлечение элементов на основе поиска с наиболее точным совпадением с заданным ключом.

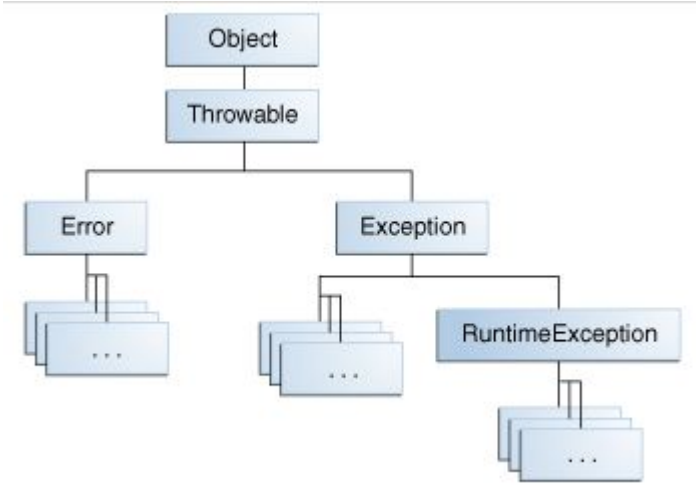
```
Map.Entry<K,V> lowerEntry(K key);
K lowerKey(K key);
Map.Entry<K,V> floorEntry(K key);
K floorKey(K key);
Map.Entry<K,V> ceilingEntry(K key);
K ceilingKey(K key);
Map.Entry<K,V> higherEntry(K key);
K higherKey(K key);
Map.Entry<K,V> firstEntry();
Map.Entry<K,V> lastEntry();
Map.Entry<K,V> pollFirstEntry();
Map.Entry<K,V> pollLastEntry();
NavigableSet<K> navigableKeySet();
```

# Map.Entry

Хранит ключ, значение и вспомогательную информацию.

```
K getKey();
V getValue();
V setValue(V value);
boolean equals(Object o);
int hashCode();
```

# Исключения



# Работа с исключениями

try
catch
finally
throw
throws

# File

Описывает свойства самого файла.

Применяется для получения и управления информацией, связанной с файлом: разрешение, дата, время и путь к каталогу.

```
File(String pathname);
String getName();
String getPath();
String getAbsolutePath();
boolean exists();
boolean createNewFile()
boolean isDirectory();
long lastModified();
boolean delete();
File[] listFiles();
File[] listFiles(FilenameFilter filter);
```

# Сериализация/Десериализация

Сериализация - процесс записи состояния объекта в байтовый или символьный поток.

Десериализация - восстановление объекта из потока.

Применяется при удаленных вызовах и записях в файлы.

# Поток ввода/вывода

Это абстракция, которая либо потребляет, либо производит информацию.

Бывают байтовые и символьные.

Потоки реализованы через иерархию классов в пакете java.io

# InputStream

Абстрактный класс, определяющий модель потокового байтового ввода.

```
int available()
abstract int read();
int read(byte[] b);
int read(byte[] b, int off, int len);
byte[] readAllBytes();
byte[] readNBytes(int len);
int readNBytes(byte[] b, int off, int len);
long skip(long n);
boolean markSupported();
void mark(int readlimit);
void reset();
long transferTo(OutputStream out);
void close();
```

# Наследники InputStream

FileInputStream - чтение из файла;

ByteArrayinputStream - чтение из массива байтов;

BufferedInputStream - буфферизированный поток чтения;

PushbackInputStream - поток, позволяющий вернуть прочитанное назад;

SequenceInputStream - позволяет объединять несколько потоков в один;

DataInputStream - чтение примитивных типов данных;

ObjectInputStream - чтение объектов.

# Flushable

Интерфейс для принудительной записи в поток данных, к которому присоединен объект.
Очистка потока данных обычно приводит к физической записи буферизованного вывода на базовое устройство.

# OutputStream

Абстрактный класс, определяющий модель потокового байтового вывода.

```
void write(int b);
void write(byte[] b);
void write(byte[] b, int off, int len);
void flush();
void close();
```

# Наследники OutputStream

FileOutputStream - запись в файл;

ByteArrayOutputStream - запись в байтовый массив;

BufferedOutputStream - буфферизированный выходной поток;

PrintStream - добавляет функциональность для вывода другому потоку;

DataOutputStream - запись примитивных типов;

ObjectOutputStream - запись объектов.

# Reader

```
boolean ready();
int read();
int read(char[] cbuf);
int read(char[] cbuf, int off, int len);
long skip(long n);
void mark(int readAheadLimit);
void reset();
void close();
long transferTo(Writer out);
```

# Наследники Reader

FileReader - чтение из файла;

CharArrayReader - чтение из символьного массива;

StringReader - чтение из строки;

BufferedReader - буфферизированный поток чтения;

PushbackReader - поток, позволяющий вернуть прочитанное назад;

# Writer

```
write(int c);
void write(char[] cbuf);
void write(String str);
void write(char[] cbuf, int off, int len);
void write(String str, int off, int len);
Writer append(CharSequence csq);
```

# Наследники Writer

FileWriter - запись в файл;

CharArrayWriter - запись в массив символов;

StringWriter - запись в строку;

BufferedWriter - буфферизированный выходной поток;

PrintWriter - добавляет функциональность для вывода другому потоку;

# InputStreamReader/OutputStreamWriter

Классы-мосты между InputStream/Reader и OutputStream/Writer

# NIO (new I/O)

Новая система ввода/вывода, основанная на буферах и каналах.

Буфер - абстракция для хранения данных.

Канал - абстракция для подключения к устройству ввода/вывода.

# Buffer

Пакет java.nio

Представляет общую функциональность:

- Текущая позиция;
- Предел;
- Ёмкость.

# Методы Buffer

```
int capacity();
int position();
Buffer position(int newPosition);
int limit();
Buffer limit(int newLimit);
Buffer mark();
Buffer reset();
Buffer clear();
Buffer flip();
Buffer rewind();
boolean hasRemaining();
int remaining();
boolean isReadOnly();
Buffer slice(int index, int length);
Buffer duplicate();
boolean hasArray();
byte[] array();
```

# Наследники Buffer

ByteBuffer
CharBuffer
DoubleBuffer
FloatBuffer
IntBuffer
LongBuffer
ShortBuffer

# Методы наследников (ByteBuffer)

```
ByteBuffer allocate(int capacity);
ByteBuffer wrap(byte[] array);
ByteBuffer wrap(byte[] array, int offset, int length);
byte get();
byte get(int index);
ByteBuffer put(byte b);
ByteBuffer put(int index, byte b);
ByteBuffer put(ByteBuffer src);
```

# Channel

Пакет java.nio.channels

Представляет собой открытое подключение с источником или адресатом ввода-вывода.

# FileChannel

Канал для чтения, записи и манипуляций с файлом.

```
FileChannel open(Path path, OpenOption... options);
FileChannel open(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs);
int read(ByteBuffer dst);
long read(ByteBuffer[] dsts, int offset, int length);
long read(ByteBuffer[] dsts);
int write(ByteBuffer src);
long write(ByteBuffer[] srcs, int offset, int length);
long write(ByteBuffer[] srcs);
long position();
FileChannel position(long newPosition);
long size();
FileChannel truncate(long size);
long transferTo(long position, long count, WritableByteChannel target);
long transferFrom(ReadableByteChannel src, long position, long count);
```

# Path

Инкапсулирует путь к файлу и представляет расположение файла в структуре каталогов.

```
Path of(String first, String... more);
Path of(URI uri);
Path getRoot();
Path getFileName();
Path getParent();
boolean startsWith(Path other);
boolean startsWith(String other);
boolean endsWith(Path other);
boolean endsWith(String other);
Path toAbsolutePath();
File toFile();
```

# Files

Утилитный класс для операций с файлами и директориями.

```
static InputStream newInputStream(Path path, OpenOption... options);
static OutputStream newOutputStream(Path path, OpenOption... options);
static Path createFile(Path path, FileAttribute<?>... attrs);
static void delete(Path path);
static boolean deleteIfExists(Path path);
static Path copy(Path source, Path target, CopyOption... options);
static Path move(Path source, Path target, CopyOption... options);
static long size(Path path);
static boolean exists(Path path, LinkOption... options);
static byte[] readAllBytes(Path path);
static List<String> readAllLines(Path path);
static Path walkFileTree(Path start, FileVisitor<? super Path> visitor);
```

# Stream API

Поток - канал для данных.

Не хранит данные, только оперирует последовательностью данных из источника.

```
BaseStream<T, S extends BaseStream<T, S>>
Stream<T> extends BaseStream<T, Stream<T>>
IntStream extends BaseStream<Integer, IntStream>
LongStream extends BaseStream<Long, LongStream>
DoubleStream extends BaseStream<Double, DoubleStream>
```

# Порождающие операции

Collection
- `default Stream<E> stream();`
- `Stream<E> parallelStream();`

Arrays
- `static <T> Stream<T> stream(T[] array);`

Stream
- `static <T> Stream<T> of(T t);`
- `static <T> Stream<T> of(T... values);`
- `static <T> Stream<T> iterate(final T seed, final UnaryOperator<T> f);`
- `static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next);`
- `static <T> Stream<T> generate(Supplier<? extends T> s);`
- `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b);`

# Промежуточные операции

```
Stream<T> filter(Predicate<? super T> predicate);
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
IntStream mapToInt(ToIntFunction<? super T> mapper);
LongStream mapToLong(ToLongFunction<? super T> mapper);
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper);
LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper);
DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper);
Stream<T> distinct();
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
Stream<T> peek(Consumer<? super T> action);
Stream<T> limit(long maxSize);
Stream<T> skip(long n);
default Stream<T> takeWhile(Predicate<? super T> predicate);
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

# Терминальные операции

```
void forEach(Consumer<? super T> action);
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> generator);
T reduce(T identity, BinaryOperator<T> accumulator);
Optional<T> reduce(BinaryOperator<T> accumulator);
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R>
combiner);
<R, A> R collect(Collector<? super T, A, R> collector);
default List<T> toList();
Optional<T> min(Comparator<? super T> comparator);
Optional<T> max(Comparator<? super T> comparator);
long count();
boolean anyMatch(Predicate<? super T> predicate);
boolean allMatch(Predicate<? super T> predicate);
boolean noneMatch(Predicate<? super T> predicate);
Optional<T> findFirst();
Optional<T> findAny();
```

# Аннотации

Аннотации позволяют встраивать дополнительную информацию в исходный код.

```
@Override
@FunctionalInterface
```

```
public @interface MyAnno {
    String str ( ) ;
    int val ( ) ;
}
```

# RetentionPolicy / Политики хранения аннотаций

- SOURCE - удерживается (существует) только на этапе исходного кода и отбрасывается на этапе компиляции;
- CLASS - сохраняется в классе на этапе компиляции, но не будет доступна во время работы приложения;
- RUNTIME - аннотация доступна во время выполнения программы.

# ElementType

- *TYPE*
- *FIELD*
- *METHOD*
- *PARAMETER*
- *CONSTRUCTOR*
- *LOCAL_VARIABLE*
- *PACKAGE*

# Маркерные аннотации

Аннотации, не содержащие членов. Их единственная цель пометить элемент своим присутствием.

```
@Override
@FunctionalInterface
```

# Одноэлементные аннотации

Содержат только один член.

# Ограничения методов аннотаций

- Аннотации не могут наследоваться друг от друга. Вместо этого, аннотация помечается другой аннотацией.
- Методы аннотаций не должны принимать параметры.
- Методы должны возвращать примитивы, String или Class, enum или другой тип аннотации.
- Аннотации не могут быть обобщенными и в них нельзя указвать throws.

# Рефлексия

С помощью рефлексии можно анализировать и динамически описывать компонент во время выполнения.

Инструменты лежат в пакете java.lang.reflect.

# Классы в пакете java.lang.reflect

| Класс | Описание |
|---|---|
| Array | Позволяет динамически создавать массивы и манипулировать ими. |
| Class | Предоставляет информацию о классе |
| Constructor | Предоставляет информацию о конструкторе |
| Field | Предоставляет информацию о поле |
| Method | Предоставляет информацию о методе |
| Modifier | Предоставляет информацию о модификаторах доступа к классу и его членам |
| Parameter | Предоставляет информацию о параметрах метода |

# Пример

```
void method() {

    // do smth

    Thread t = new Thread(() -> {
        // чтение пользовательского ввода
    });
    t.start();
    t.join();

    // do smth
}
```

# Многозадачность

- На основе процессов
- На основе потоков

# Создание потока

1. Наследование от Thread
2. Имплементация Runnable и передача в Thread

# Состояние потока

NEW - поток только что создан, еще не запущен. Запуск происходит вызовом метода start() у объекта Thread.
RUNNABLE - поток готов к выполнению, но еще не выполняется, либо выполняется.
BLOCKED - поток переходит в это состояние, когда он пытается зайти в ограниченную секцию кода, куда может зайти только один поток, и эта секция уже выполняется другим потоком.
WAITING - поток находится в состоянии ожидания без указания времени, пока другой поток не разбудит его.
TIMED_WAITING - поток находится в ожидании, но с указанием времени. После истечения этого времени поток переходит в RUNNABLE и готов снова быть запущенным.
TERMINATED - переходит в состояние, когда метод run() полностью выполнился, либо упал с ошибкой без перехвата исключения. В этом состоянии поток больше не может быть перезапущен.

# Приоритет потока

Число от 1 до 10, по умолчанию 5.

# Потоки-демоны

Предназначены для фоновых низкоприоритетных задач.

JVM не ждет их завершения, чтобы завершить программу.

# Методы класса Thread

```
void start();
void run();
static void yield();
static void sleep(long millis);
static void sleep(long millis, int nanos);
static void sleep(Duration duration);
void setPriority(int newPriority);
int getPriority();
void setName(String name);
String getName();
void join();
void setDaemon(boolean on);
boolean isDaemon();
State getState();
static native boolean holdsLock(Object obj);
```

# Гонки, объект-монитор и synchronized

Гонки (race condition) - ситуация, когда два или более потоков одновременно обращаются к общему ресурсу.

Монитор - объект позволяющий выполнять код только одному потоку.

Модификатор и блок synchronized.

# Блокировки

Представлена классом ReentrantLock

```
void lock();
boolean tryLock();
boolean tryLock(long time, TimeUnit unit);
void unlock();
Condition newCondition();
```

## Condition

```
void await();
long awaitNanos(long nanosTimeout);
boolean await(long time, TimeUnit unit);
void signal();
void signalAll();
```

# Атомарные операции

Предназначены для хранения и атомарного обновления значений.

Хранятся в пакете java.util.concurrent.atomic

# AtomicInteger

Представляет целое значение int, над которым можно выполнять атомарные операции без synchronized.

Представляет собой обертку над int

```
int get();
void set(int newValue);
int getAndSet(int newValue);
boolean compareAndSet(int expectedValue, int newValue);
int getAndIncrement();
int getAndDecrement();
int getAndAdd(int delta);
int incrementAndGet();
int getAndUpdate(IntUnaryOperator updateFunction);
int updateAndGet(IntUnaryOperator updateFunction);
int intValue();
long longValue();
```

# LongAdder

Оптимизирован для очень частых операций увеличения/уменьшения long в условиях высокой конкуренции потоков.

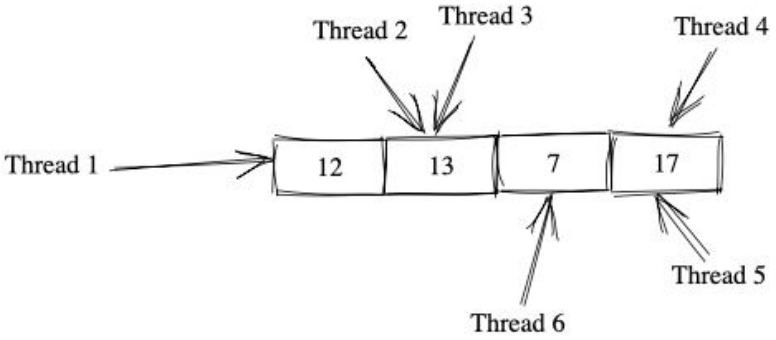
В отличие от AtomicLong (аналог AtomicInteger для long), не держит одно значение, а хранит набор ячеек (cells); разные потоки обновляют разные ячейки, уменьшая конфликты CAS и увеличивая пропускную способность.

Подходит для метрик, счётчиков запросов, статистики, когда много записей и относительно мало чтений.

# LongAccumulator

Более общий вариант LongAdder: позволяет не только суммировать, но и использовать произвольную ассоциативную функцию над long, например max, min, произвольное накопление.

В конструктор передаётся бинарная операция (LongBinaryOperator) и элемент (identity) для этой операции.



# Пулы потоков

FixedThreadPool - пул потоков фиксированного размера.
SingleThreadPool - пул из одного одного потока.
CachedThreadPool - потоки создаются в пуле по мере их необходимости. После завершения одной задачи их можно использовать повторно, но, если поток не использовался в течении 60 секунд, он удаляется.
ScheduledThreadPool - пул потоков с задачами, которые запустятся через некоторое время в будущем.

# ExecutorService

Класс для управления пулами потоков.

```
<T> Future<T> submit(Callable<T> task);
<T> Future<T> submit(Runnable task, T result);
Future<?> submit(Runnable task);
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit
unit);
<T> T invokeAny(Collection<? extends Callable<T>> tasks);
<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit);
void shutdown();
boolean awaitTermination(long timeout, TimeUnit unit);
List<Runnable> shutdownNow();
```

# Future

Объект представляющий собой результат асинхронного выполнения.

Позволяет проверить, выполнено ли задание, дождаться выполнения и получить результат.

Состояния Future:
- RUNNING - задача выполняется
- SUCCESS - задача завершилась успешно с результатом
- FAILED - задача завершилась с исключением
- CANCELLED - задача была отменена

# Методы Future

```
boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
boolean isDone();
V get() throws InterruptedException, ExecutionException;
V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,
TimeoutException;
default V resultNow();
```

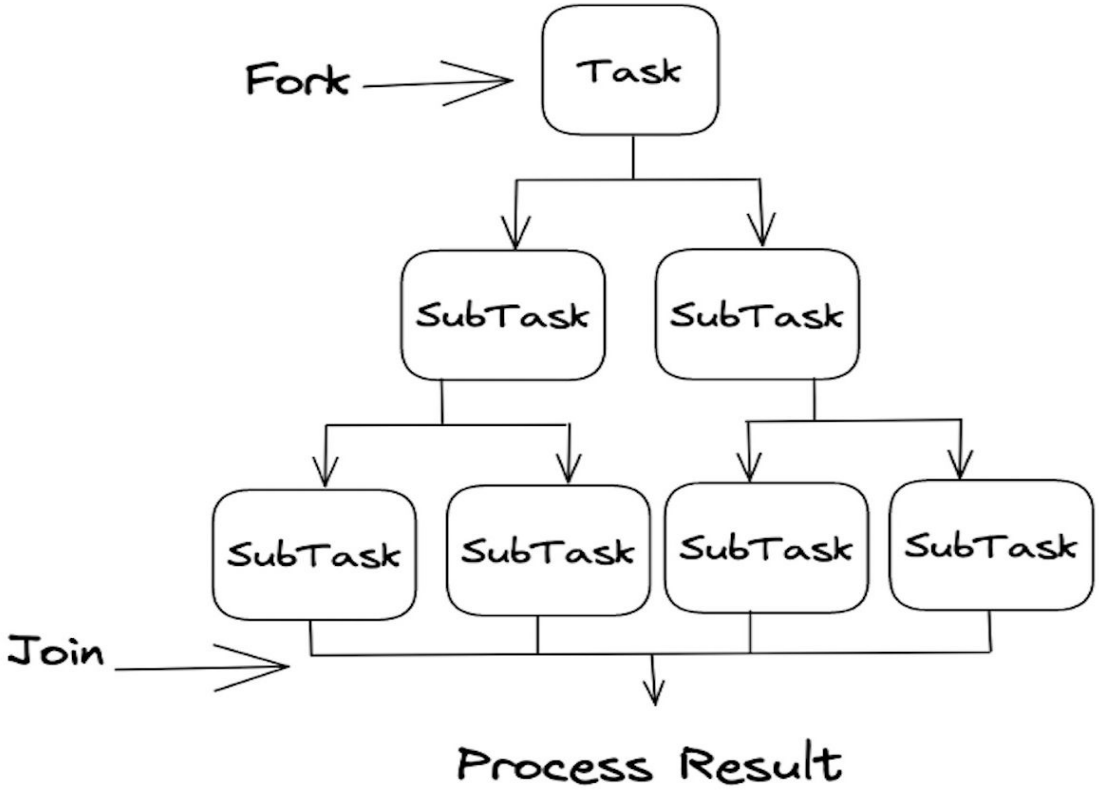
# ForkJoinPool

Он основан на алгоритме “Разделяй и властвуй”.

ForkJoinPool применяется, когда нужно одну большую задачу разбить на несколько маленьких, а потом собрать все в единый результат.

Такими задачами, как правило, являются сортировка, преобразования массива или поиск элемента в массиве.

# Принцип работы



# ForkJoinTask

Абстрактный класс, представляющий собой задачу для ForkJoinPool.

Наследник Future.

Методы:

```
final ForkJoinTask<V> fork();
final V join();
```

Наследники:

RecursiveAction

RecursiveTask<T>

# Semaphore

Класс из пакета java.util.concurrent

Ограничивает доступ к общему ресурсу на основе счётчика разрешений

Где применяется:
- Ограничение одновременного доступа к ресурсам (пул соединений, ограничение количества рабочих потоков).
- Регулирование пропускной способности - ограничение частоты обработки задач.
- Producer-Consumer паттерн - координация между производителем и потребителем через разные семафоры (через единичный семафор).

# CountDownLatch

Дает потокам возможность ожидать выполнение пока значение счетчика не достигнет нуля.

Срабатывает только один раз - после достижения нуля, его нельзя увеличить.

Где применяется:
- Ожидание завершения работы нескольких потоков: главный поток ждёт, пока несколько рабочих потоков закончат задачи.
- Запуск нескольких потоков одновременно: все потоки ждут на CountDownLatch(1), а затем один вызов countDown() запускает их все сразу для максимального параллелизма.

# CyclicBarrier

Синхронизирующее средство из пакета java.util.concurrent

Позволяет фиксированному числу потоков ждать друг друга в определённой точке (барьер), а затем синхронно продолжить выполнение.

Где применяется:
- Синхронизация стартов нескольких потоков: все потоки готовятся, затем одновременно стартуют для параллельного тестирования или загрузочных испытаний.
- Итерационные вычисления: потоки обрабатывают данные в каждой итерации, ждут друг друга на барьере, затем переходят к следующей итерации.
- Фазовое выполнение алгоритмов: разные фазы работы требуют синхронизации всех участников перед переходом к следующей фазе.
- Опциональное действие между фазами: barrier action выполняется перед пробуждением всех потоков, что удобно для агрегации результатов.

# Exchanger

Синхронизирующее средство из пакета java.util.concurrent, которое предназначено для обмена объектами между двумя потоками.

Когда первый поток вызывает метод exchange(), он ждёт, пока второй поток также вызовет exchange(), после чего они обмениваются данными и оба продолжают выполнение.

# ConcurrentHashMap

Потокобезопасная реализация интерфейса Map.

Использует сегментированное блокирование для обеспечения высокой производительности при одновременном доступе.

# CopyOnWriteArrayList

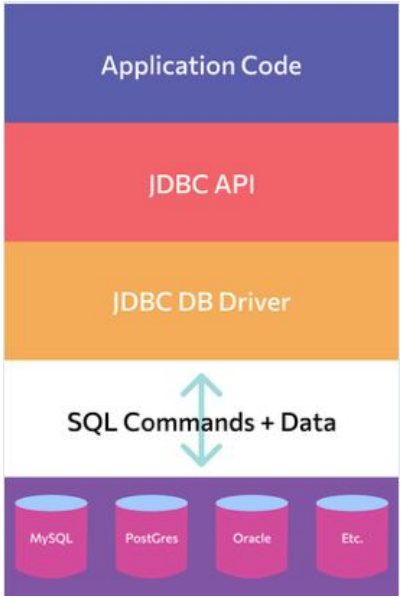
Потокобезопасная реализация List.

Любая операция изменения (add, set, remove и т.п.) делает новую копию внутреннего массива, а чтения работают с неизменяемым снимком данных.

# JDBC

Программный интерфейс для взаимодействия с базами данных.

Позволяет устанавливать соединения с БД, запрашивать и обновлять данные с помощью языка SQL.



# Подключение к БД

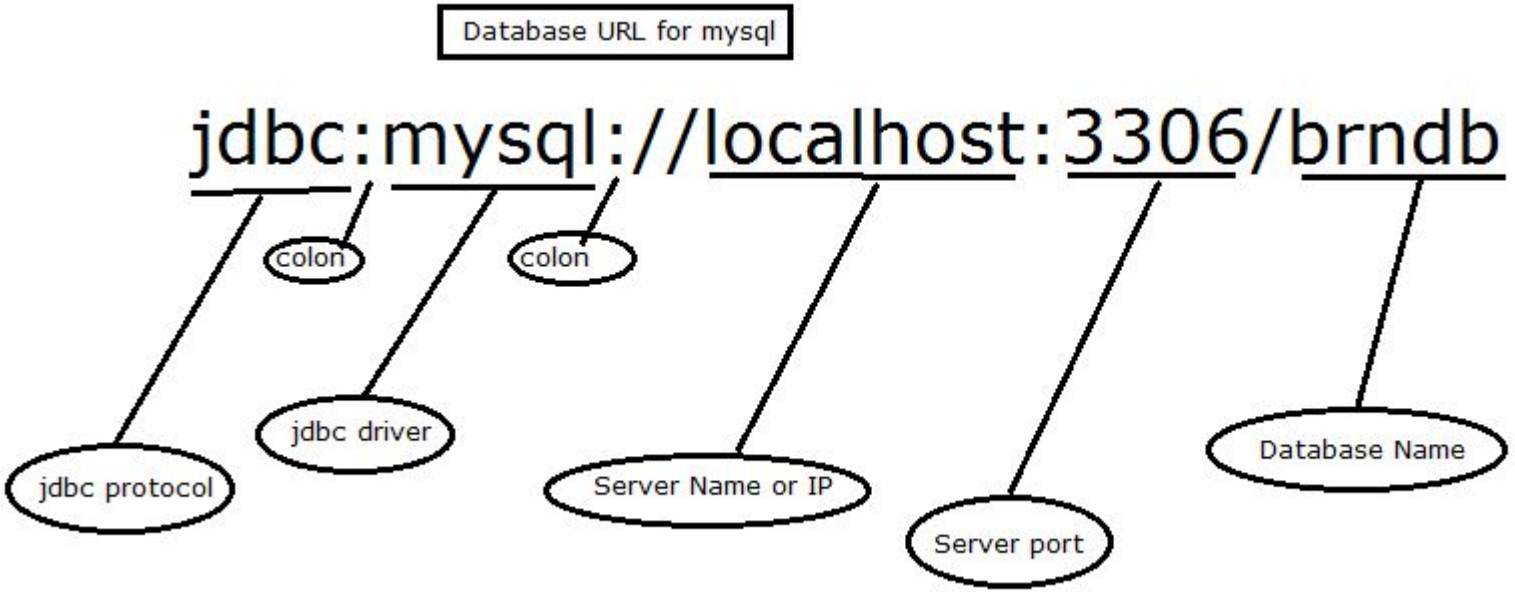
Установка соединения с базой и получение соединения (Connection) с помощью DriverManager.

Создание Statement для выполнения запросов.

Вызвать:

- executeUpdate - для обновления данных (Insert, Update, Delete).
- executeQuery - для получения данных.

# Строка подключения к БД



# Select

SELECT id, name

FROM table

WHERE id = 1 and name = ‘A’;

# Insert

INSERT INTO table (id, name)

VALUES (2, ‘B’);

# Update

UPDATE table SET name = ‘C’

WHERE id = 2;

# Delete

```
DELETE FROM table

WHERE id = 1;
```

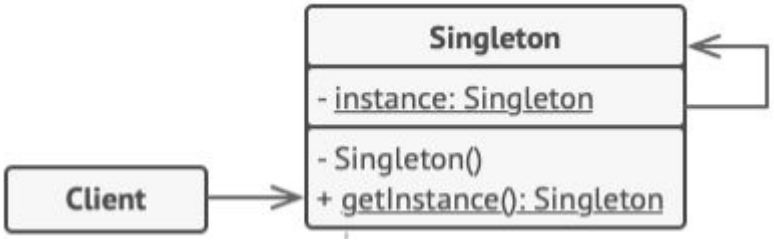
# Паттерны проектирования

Паттерны - часто встречающиеся решения определенных проблем при проектировании архитектуры программ.
Паттерн представляет собой не код, а общую концепцию решения какой-либо проблемы, которую нужно адаптировать под ваши нужды.

Категории паттернов:
- порождающие
- структурные
- поведенческие

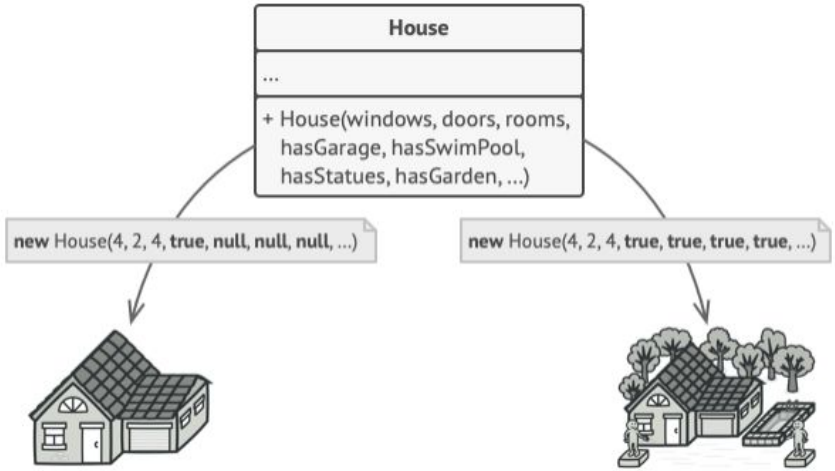
# Singleton / Одиночка

Гарантирует, что у класса есть только один экземпляр.



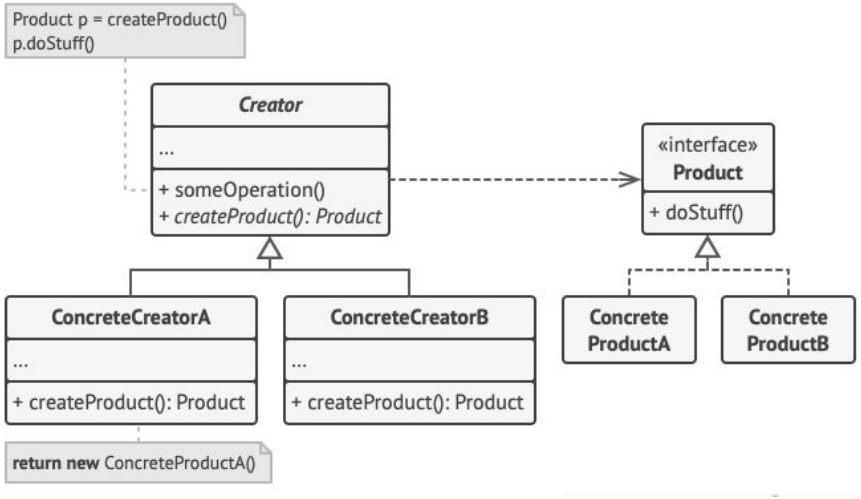
# Builder / Строитель

Позволяет создавать сложные объекты пошагово и даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



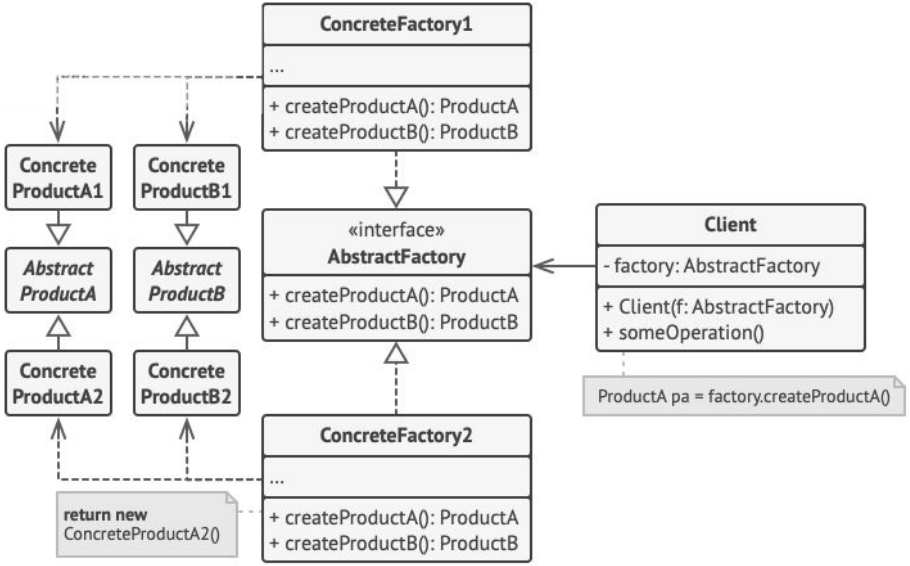
# Factory Method / Фабричный метод

Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



# Abstract Factory / Абстрактная фабрика

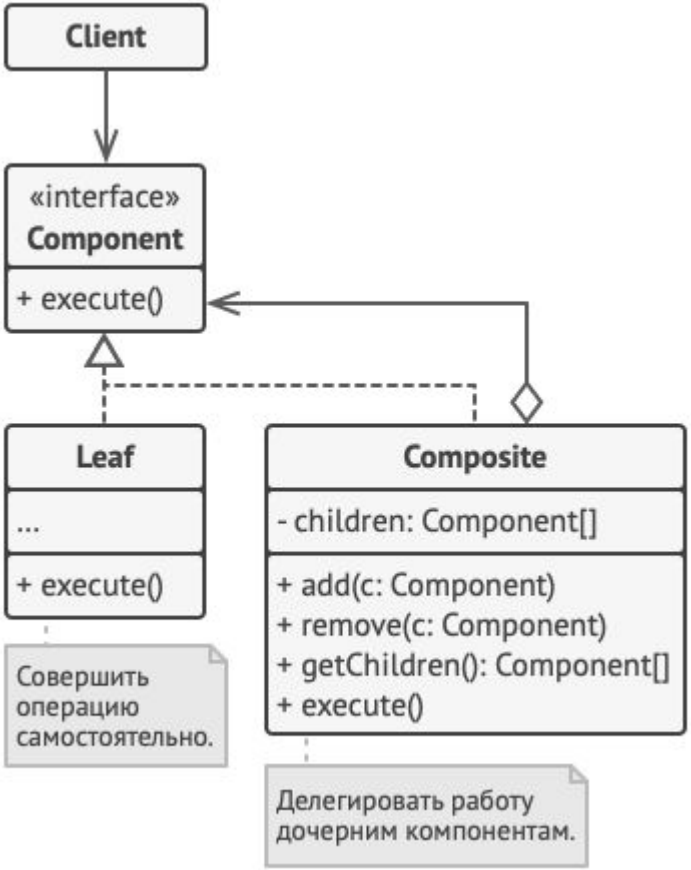
Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



# Composite / Компоновщик

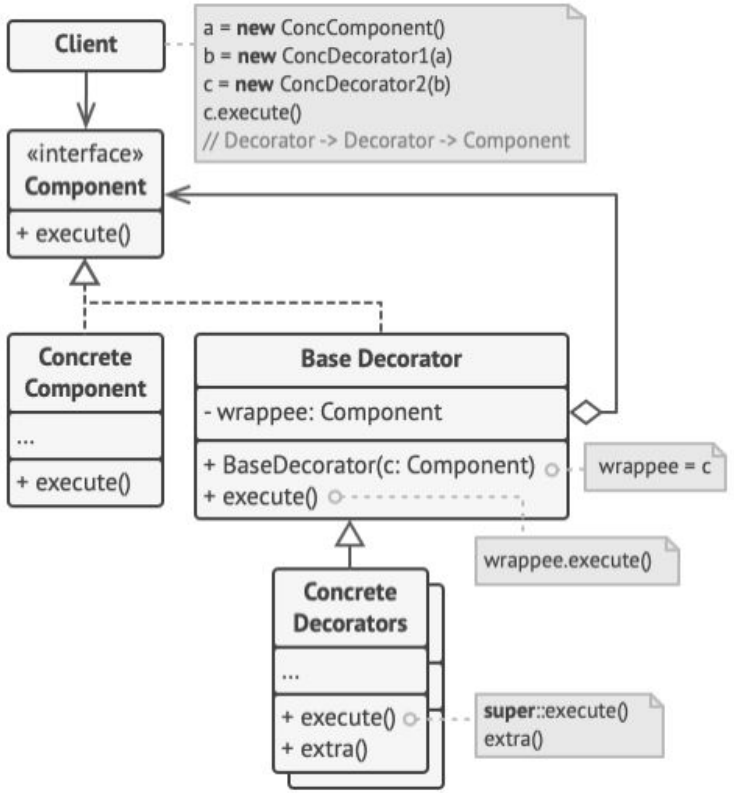
Позволяет сгруппировать множество объектов

в древовидную структуру, а затем работать с ней так, как будто это единичный объект.



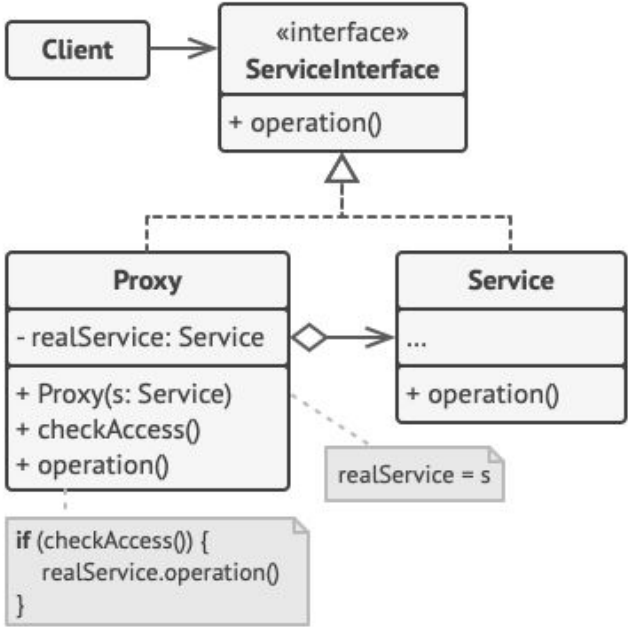
# Decorator / Декоратор

Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».



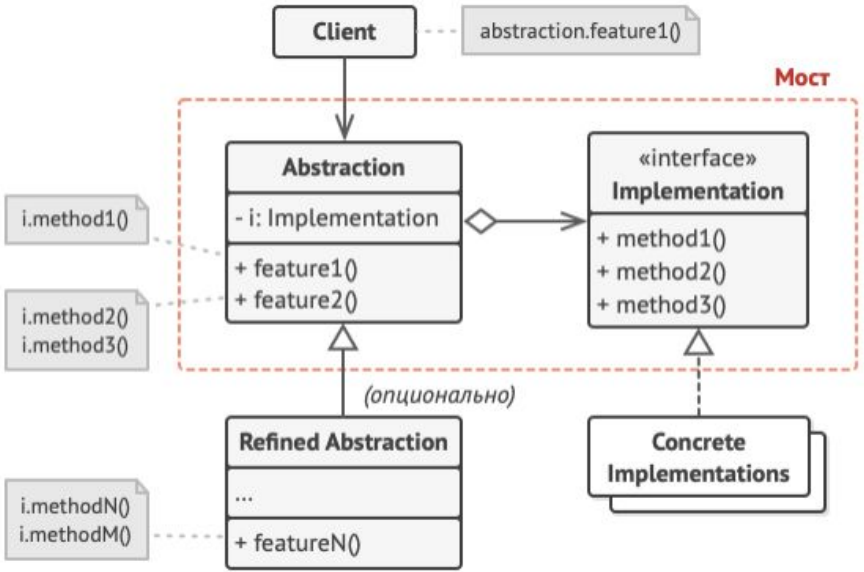
# Proxy / Заместитель

подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.



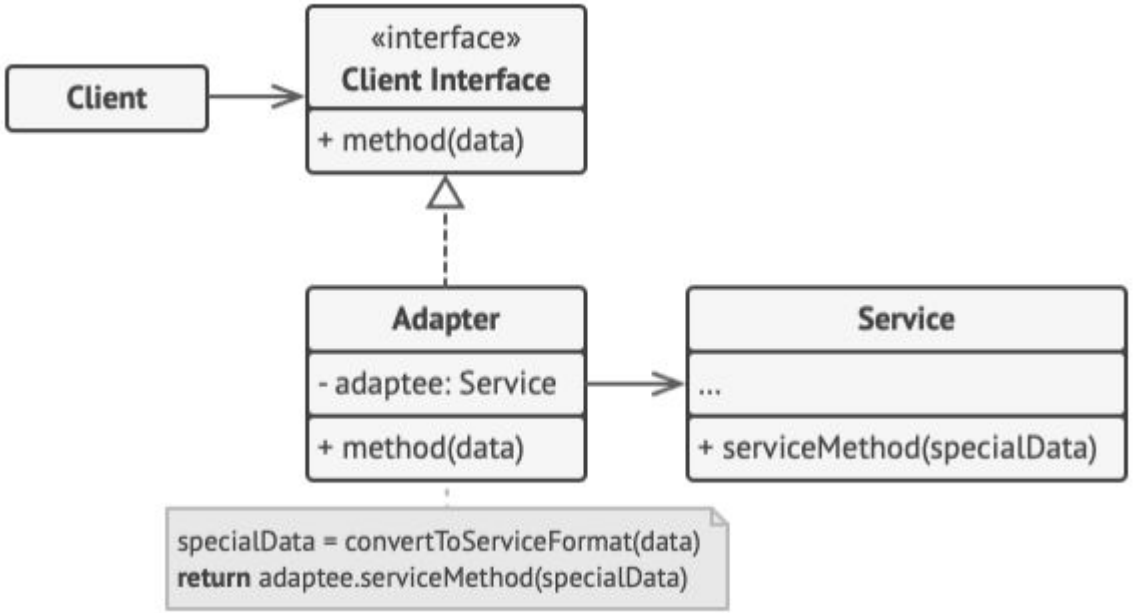
# Bridge / Мост

Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.



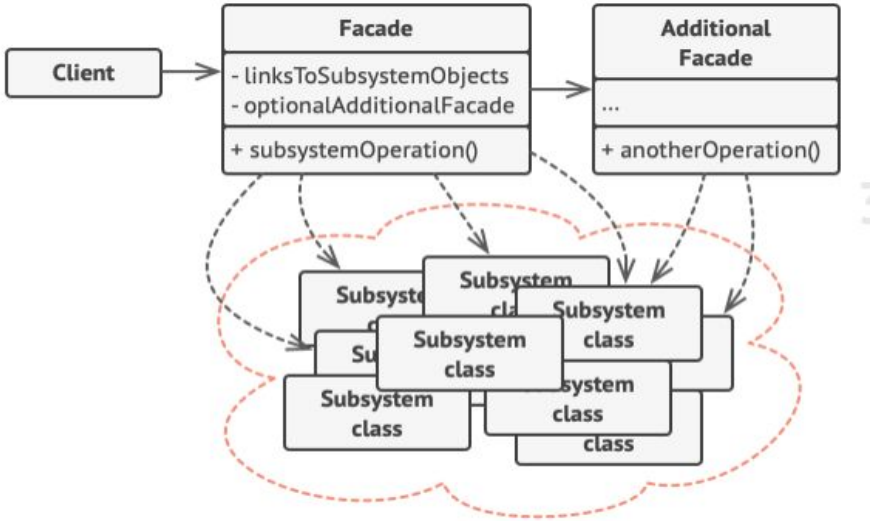
# Adapter / Адаптер

Позволяет объектам с несовместимыми интерфейсами работать вместе.



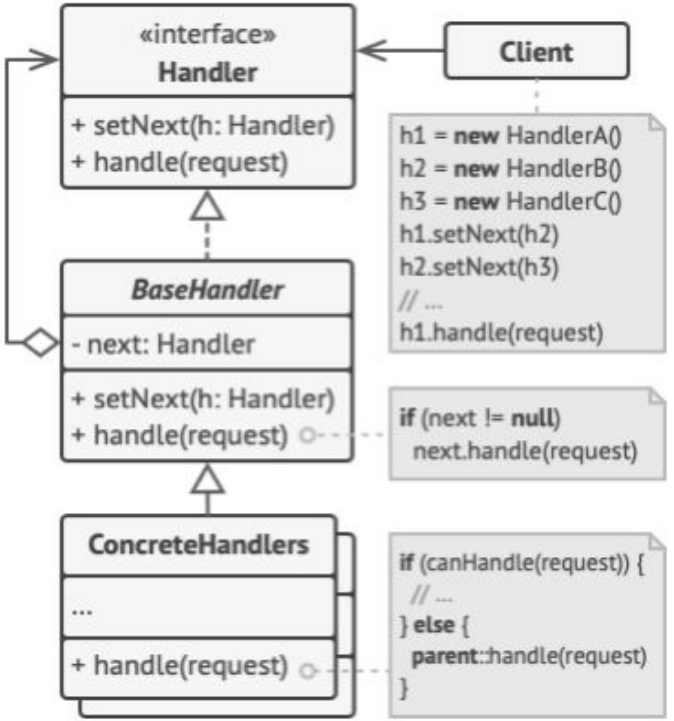
# Facade / Фасад

Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



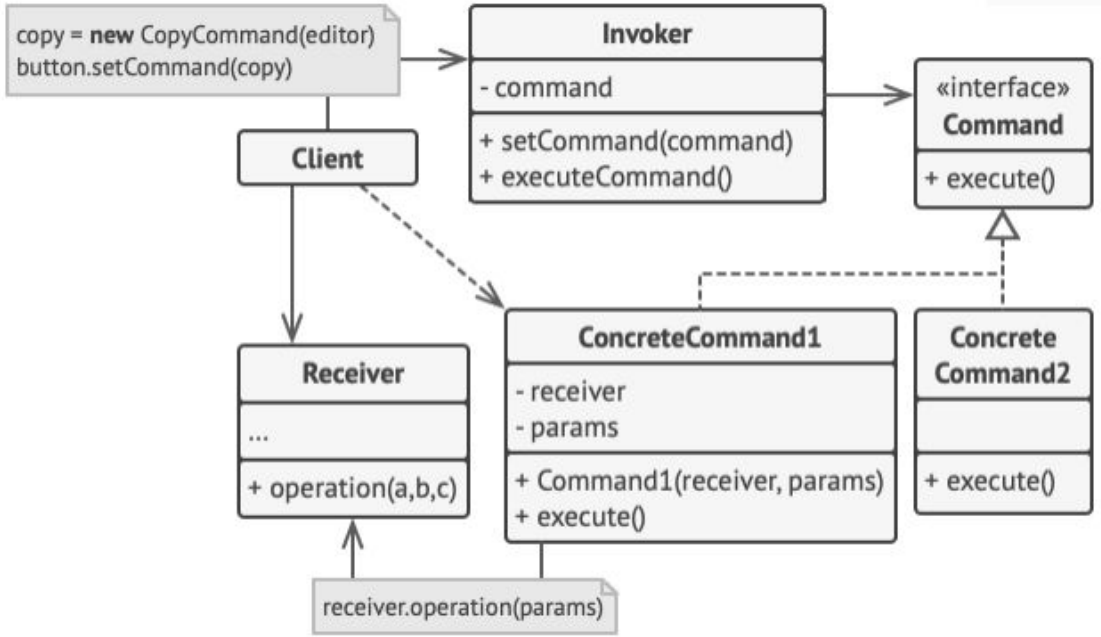
# Chain of Responsibility / Цепочка обязанностей

Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.



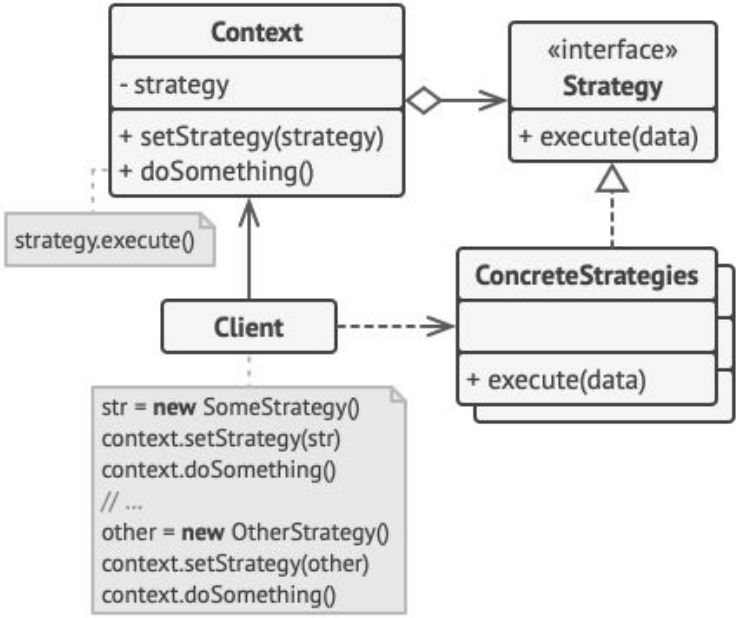
# Command / Команда

Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов.



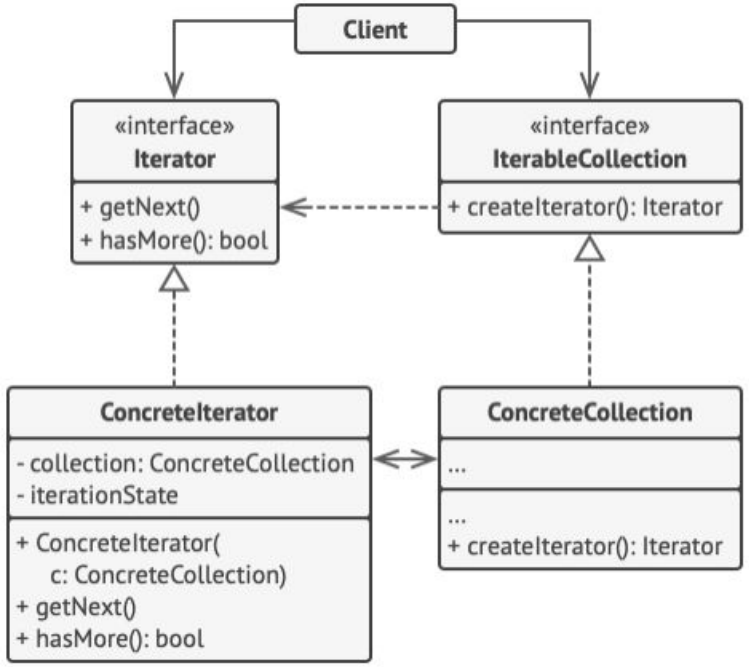
# Strategy / Стратегия

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.



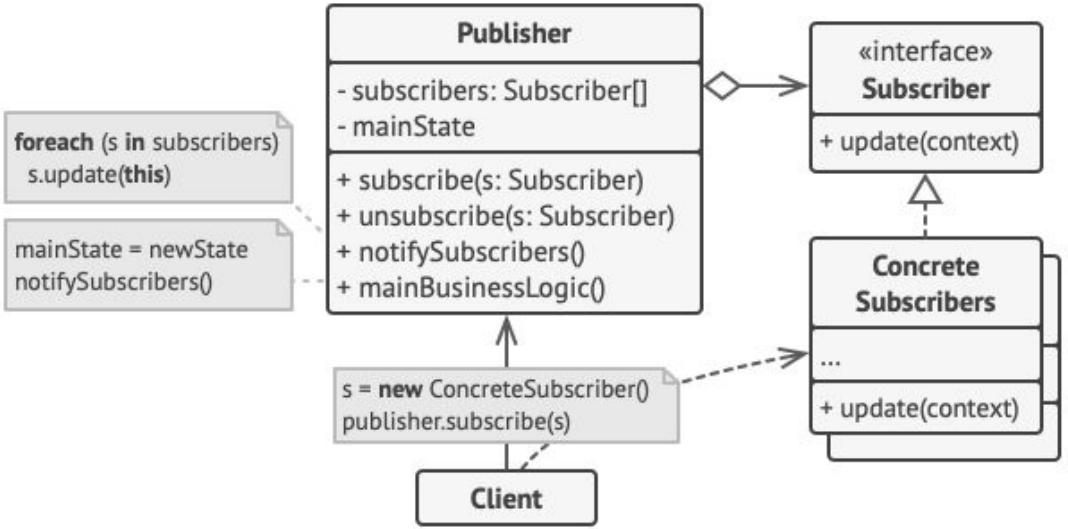
# Iterator / Итератор

Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



# Observer / Наблюдатель

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



# State / Состояние

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.