

□ Конспекты курса: “Продвинутая Java Платформа”

Лектор: Александр Маторин

Кафедра БИТ, МФТИ

Ведётся студентом: [Андрей Бодакин](#)

□ [Скачать полный PDF со всеми лекциями](#) — генерируется автоматически через GitHub Actions при каждом обновлении.

□ Описание курса

Курс посвящён глубокому изучению языка **Java**, начиная с основ синтаксиса и заканчивая продвинутыми темами: многопоточность, JVM, коллекции, работа с памятью и многое другое.

Лектор: **Александр Маторин** — практик, эксперт в Java-экосистеме.

Цель репозитория — систематизировать знания, вести конспекты лекций, делиться материалами и примерами кода.

□ Оглавление

□ Лекции

- [Лекция 1](#) — Основы синтаксиса Java + Настройка окружения и работа в IntelliJ IDEA
 - [Лекция 2](#) — Прimitives типы, классы-обёртки, Пакеты Java, Object, equals/hashCode
 - [Лекция 3](#) — Структуры данных в Java
 - [Лекция 4](#) - Generics
 - Следующие лекции будут добавляться по мере прохождения курса...
-

□ Как использовать

- Все конспекты в формате **Markdown** — легко читать на GitHub.
 - Примеры кода — в папке [code/](#) (если есть).
 - Pull Request'ы и Issues приветствуются — если нашли ошибку или хотите дополнить материал.
-

□ Сотрудничество

Если ты тоже учишься на курсе — присылай свои конспекты, дополнения, примеры кода!
Открыт для совместного ведения и улучшения материалов.

□ Лицензия

Этот репозиторий распространяется под лицензией **MIT** — используйте свободно для обучения и распространения знаний.

□ “Знания растут, когда ими делишься.”

□ Лекция 1 — Основы синтаксиса Java + Настройка окружения и работа в IntelliJ IDEA

□ Основные темы

- Установка и настройка JDK
- Компиляция и запуск через `javac` и `java`
- Переменные среды: `PATH`, `CLASSPATH`
- Выбор и настройка IDE (IntelliJ IDEA)
- Горячие клавиши и рефакторинги в IntelliJ IDEA
- Основы синтаксиса: классы, методы, переменные, управляющие конструкции

□ Настройка Java-окружения

Установка JDK

→ Скачать можно с: - [Oracle JDK](#) - [OpenJDK \(Adoptium / Temurin\)](#) ← **рекомендуется**

Проверка установки:

```
java -version
javac -version
```

→ Должны вывести версию Java и компилятора.

Переменная окружения `PATH`

→ `PATH` — список директорий, где система ищет исполняемые файлы.

□ Добавь путь к `bin` JDK в `PATH`:

Linux/macOS (в `~/.bashrc` или `~/.zshrc`):

```
export PATH="/path/to/jdk/bin:$PATH"
```

Windows: - Панель управления → Система → Дополнительные параметры → Переменные среды → `PATH`

→ Добавить путь, например:

`C:\Program Files\Java\jdk-21\bin`

CLASSPATH

- Указывает JVM, где искать .class-файлы и библиотеки.
- Обычно не нужно настраивать вручную при работе с IDE или Maven/Gradle.
- Если компилируешь вручную:

```
javac -cp ".;lib/*" MyClass.java
java -cp ".;lib/*" MyClass
```

- . — текущая директория.
- ; — разделитель в Windows, : — в Linux/macOS.

□ Работа с IntelliJ IDEA

Почему IntelliJ IDEA?

- Самая популярная и мощная IDE для Java.
- Умное автодополнение, рефакторинги, отладка, интеграция с Maven/Gradle, Git.
- Community Edition — бесплатна и достаточно для обучения.

🖱️ Горячие клавиши IntelliJ IDEA (must-have)

| Действие | Windows/Linux | macOS |
|---------------------------------|------------------|------------------|
| Автодополнение | Ctrl + Space | Cmd + Space |
| Быстрое исправление / подсказки | Alt + Enter | Option + Enter |
| Запуск программы | Shift + F10 | Ctrl + R |
| Отладка | Shift + F9 | Ctrl + D |
| Поиск по проекту | Ctrl + Shift + F | Cmd + Shift + F |
| Поиск класса | Ctrl + N | Cmd + O |
| Поиск файла | Ctrl + Shift + N | Cmd + Shift + O |
| Переход к определению | Ctrl + B | Cmd + B |
| Рефакторинг: переименование | Shift + F6 | Shift + F6 |
| Закомментировать строку | Ctrl + / | Cmd + / |
| Форматирование кода | Ctrl + Alt + L | Cmd + Option + L |
| Открыть структуру класса | Ctrl + F12 | Cmd + F12 |

□ Полезные Live Templates (шаблоны кода)

| Шаблон | Результат | Описание |
|--------|----------------------------------|-------------------------|
| sout | System.out.println(); | Быстрый вывод в консоль |
| iter | for (Type item : collection) { } | Цикл for-each |

| Шаблон | Результат | Описание |
|--------|--|---------------------|
| psvm | <code>public static void main(String[] args) { }</code> | Главный метод |
| itar | <code>for (int i = 0; i < arr.length; i++) { }</code> | Цикл по индексу |
| ifn | <code>if (var == null) { }</code> | Проверка на null |
| inn | <code>if (var != null) { }</code> | Проверка на не-null |

→ Просто введи шаблон и нажми Tab.

□ Рефакторинги в IntelliJ IDEA

□ Extract Method (Вынести метод)

Выдели код → Ctrl + Alt + M → дай имя методу → готово!

Было:

```
public void process() {
    int a = 5;
    int b = 10;
    int sum = a + b;
    System.out.println("Sum: " + sum);
}
```

Стало:

```
public void process() {
    printSum(5, 10);
}

private void printSum(int a, int b) {
    int sum = a + b;
    System.out.println("Sum: " + sum);
}
```

→ Улучшает читаемость и переиспользование.

□ Inline Method (Встроить метод)

Если метод слишком простой — можно “встроить” его обратно:

Ctrl + Alt + N

→ Полезно при оптимизации или упрощении.

□ Основы синтаксиса Java

Структура программы

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, Java!");  
    }  
}
```

→ Каждая программа начинается с main.

Переменные и типы

```
int age = 25;  
double price = 19.99;  
boolean isActive = true;  
String name = "Alice";
```

Управляющие конструкции

```
if (age >= 18) {  
    System.out.println("Совершеннолетний");  
} else {  
    System.out.println("Несовершеннолетний");  
}  
  
for (int i = 0; i < 5; i++) {  
    System.out.println(i);  
}  
  
while (condition) {  
    // ...  
}
```

□ Советы для новичков

- Всегда проверяй, что `java -version` работает в терминале.
 - Не бойся использовать `Alt + Enter` — IntelliJ IDEA часто знает, как исправить ошибку.
 - Учись пользоваться рефакторингами — они экономят кучу времени.
-

☐ Полезные ссылки

- [Скачать IntelliJ IDEA Community](#)
- [OpenJDK \(Adoptium\)](#)
- [Горячие клавиши IntelliJ IDEA \(официальная шпаргалка\)](#)
- [Теория по Java](#)

☐ Лекция 2 – Примитивные типы, классы-обёртки, Пакеты Java, Object, equals/hashCode

☐ Основные темы

- Примитивные типы данных в Java (включая размер в битах)
- Классы-обёртки (Wrapper Classes)
- Автобоксинг и автораспаковка
- Передача параметров по значению
- Пакеты: ``java.lang``, ``java.util``, ``java.io``
- Класс ``Object`` – корень иерархии
- Контракт ``equals()`` и ``hashCode()``
- Почему важно переопределять их вместе
- Переполнение (overflow) примитивных типов
- ``BigDecimal`` – точные вычисления (для денег)
- ``StringBuilder`` – эффективная работа со строками
- Практические примеры и антипаттерны

☐ Примитивные типы данных

В Java 8 примитивных типов. Размер указан в ****битах и байтах****.

| Тип | Размер (биты) | Размер (байты) | Диапазон значений | Значение по умолчанию |
|------------------------|---------------|----------------|--|-----------------------|
| ----- | ----- | ----- | ----- | ----- |
| <code>`byte`</code> | 8 бит | 1 байт | -128 до 127 | <code>`0`</code> |
| <code>`short`</code> | 16 бит | 2 байта | -32768 до 32767 | <code>`0`</code> |
| <code>`int`</code> | 32 бита | 4 байта | -2 ³¹ до 2 ³¹ -1 | <code>`0`</code> |
| <code>`long`</code> | 64 бита | 8 байт | -2 ⁶³ до 2 ⁶³ -1 | <code>`0L`</code> |
| <code>`float`</code> | 32 бита | 4 байта | ±3.4e38 (7 знаков) | <code>`0.0f`</code> |
| <code>`double`</code> | 64 бита | 8 байт | ±1.7e308 (15 знаков) | <code>`0.0d`</code> |
| <code>`char`</code> | 16 бит | 2 байта | <code>`\u0000`</code> до <code>`\uffff`</code> (Unicode) | <code>`\u0000`</code> |
| <code>`boolean`</code> | не определён* | не определён* | <code>`true`</code> / <code>`false`</code> | <code>`false`</code> |

> ☐ ``*`` – размер ``boolean`` не определён спецификацией JVM – зависит от реализации. Обычно хранится как ``int`` (32 бита)

☐ Классы-обёртки (Wrapper Classes)

Каждый примитив имеет объектный аналог:

| Примитив | Класс-обёртка |
|-----------|---------------|
| ----- | ----- |
| `int` | `Integer` |
| `long` | `Long` |
| `double` | `Double` |
| `boolean` | `Boolean` |
| `char` | `Character` |
| ... | ... |

Зачем нужны?

- Для хранения в коллекциях (`List<Integer>`, а не `List<int>`).
- Для использования `null`.
- Для вызова методов: `Integer.parseInt()`, `Character.isDigit()` и т.д.

☐ Автобоксинг и автораспаковка

→ **Автобоксинг** — автоматическое преобразование примитива в объект.

→ **Автораспаковка** — наоборот.

```
```java
Integer a = 10; // ← автобоксинг: int → Integer
int b = a; // ← автораспаковка: Integer → int
```

```
List<Integer> list = new ArrayList<>();
list.add(5); // ← автобоксинг
int first = list.get(0); // ← автораспаковка
```

### ☐ Осторожно с null!

```
Integer x = null;
int y = x; // ← NullPointerException!
```

→ Всегда проверяй на null перед распаковкой.

---

## ☐ Передача параметров по значению

☐ В Java всё передаётся по значению — даже объекты!

→ При передаче объекта — копируется **ссылка на объект**, а не сам объект.

### Пример:

```
public static void main(String[] args) {
 Person p = new Person("Alice");
 changeName(p);
 System.out.println(p.name); // → "Bob"
}

static void changeName(Person person) {
 person.name = "Bob"; // ← изменяем объект по ссылке
}
```

→ Объект изменился, потому что мы **изменяли данные по скопированной ссылке**.

### Но:

```
static void reassign(Person person) {
 person = new Person("Charlie"); // ← переприсваиваем ссылку — оригинал не меняется!
}
```

→ После вызова `reassign(p)` — `p.name` всё ещё "Alice".

---

## □ Пакеты Java

Пакеты — это пространства имён для классов.

### Основные пакеты:

- `java.lang` — автоматически импортируется (`String`, `Object`, `System`, `Math`).
- `java.util` — коллекции, дата/время, `Scanner`, `Random`.
- `java.io` — ввод/вывод, файлы.
- `java.time` — современные даты (Java 8+).

### Импорт:

```
import java.util.List;
import java.util.ArrayList;
import java.time.LocalDate;

// Или импорт всего пакета:
import java.util.*;
```

□ \* — не нагружает приложение, только упрощает написание кода.

---



## □ Класс Object — корень иерархии

→ Корневой класс всей иерархии в Java.

→ Любой класс — наследник Object (явно или неявно).

### Основные методы:

- toString() — строковое представление объекта.
  - equals(Object obj) — сравнение объектов.
  - hashCode() — хеш-код для использования в хеш-таблицах.
  - getClass() — получить класс объекта.
  - clone() — неглубокое клонирование (осторожно!).
  - finalize() — освобождение ресурсов перед удалением, deprecated (Java 9+).
- 

### 1.1.1 equals

```
public boolean equals(@Nullable Object obj)
```

□ **Назначение:** проверяет, равны ли два объекта *логически* (по содержимому), а не физически (по ссылке).

#### □ Поведение по умолчанию

→ В классе Object метод equals() сравнивает **ссылки**:

```
Object a = new Object();
Object b = new Object();
System.out.println(a.equals(b)); // false — разные объекты
```

→ Это эквивалентно a == b.

---

#### □ Переопределение

→ В подклассах equals() **часто переопределяют**, чтобы сравнивать объекты по полям:

```
@Override
public boolean equals(Object o) {
 if (this == o) return true;
 if (o == null || getClass() != o.getClass()) return false;
 Person person = (Person) o;
 return Objects.equals(name, person.name) &&
 Objects.equals(age, person.age);
}
```

---

## □ Контракт equals()

Для **ненулевых объектов** метод equals() должен задавать **отношение эквивалентности**:

1. **Рефлексивность**:  $x.equals(x) \rightarrow true$
2. **Симметричность**: если  $x.equals(y) == true$ , то  $y.equals(x) == true$
3. **Транзитивность**: если  $x.equals(y) == true$  и  $y.equals(z) == true$ , то  $x.equals(z) == true$
4. **Консистентность**: если данные объекта не менялись, то  $x.equals(y)$  должен возвращать одно и то же значение при повторных вызовах.
5. Для **любого ненулевого x**:  $x.equals(null) == false$

□ Нарушение контракта → непредсказуемое поведение в коллекциях (HashSet, HashMap и др.).

---

## 1.1.2 hashCode

```
public native int hashCode();
```

□ **Назначение**: возвращает целочисленный хеш-код объекта. Используется в хеш-структурах: HashMap, HashSet, HashTable и др.

---

## □ Контракт hashCode()

1. **Консистентность**: если данные объекта не менялись, hashCode() должен возвращать **одно и то же значение** при каждом вызове.
  2. **Согласованность с equals()**: если  $x.equals(y) == true$ , то  $x.hashCode() == y.hashCode()$  — **обязательно**.
  3. **Необязательное условие**: если  $x.equals(y) == false$ , то  $x.hashCode()$  и  $y.hashCode()$  **могут совпадать** — это называется **коллизия хешей** (нормально для хеш-таблиц).
- 

## □ Реализация по умолчанию

→ В ранних версиях JVM hashCode() возвращал **адрес объекта в памяти**.

→ **Сейчас** — используется **псевдослучайное число**, которое: - Генерируется при первом вызове hashCode().

- Записывается в **заголовок объекта** (object header). - **Не меняется** в течение жизни объекта, даже если объект перемещается сборщиком мусора.

□ Почему изменили?

— При маленьком heap-е адреса были близки → хеши были не равномерны → **ухудшалась производительность хеш-таблиц**.

— Новая реализация даёт **лучшее распределение хешей** → меньше коллизий → быстрее работа HashMap.

---

## ❑ Почему важно переопределять hashCode() вместе с equals()

→ Представь, что ты положил объект в HashMap, а потом изменил поле, участвующее в equals(), но не обновил hashCode():

```
Person p = new Person("Alice");
map.put(p, "value");

p.setName("Bob"); // ← изменили поле, которое участвует в equals/hashCode

map.get(p); // → null! Потому что hashCode изменился, а HashMap ищет в другой "корзине".
```

→ **Решение:** поля, используемые в equals() и hashCode(), должны быть **неизменяемыми (immutable)**.

---

## ❑ Правильное переопределение

### Способ 1: Вручную

```
@Override
public boolean equals(Object o) {
 if (this == o) return true;
 if (o == null || getClass() != o.getClass()) return false;
 Person person = (Person) o;
 return Objects.equals(name, person.name) &&
 Objects.equals(age, person.age);
}

@Override
public int hashCode() {
 return Objects.hash(name, age);
}
```

### Способ 2: Через Lombok (если используется)

```
@EqualsAndHashCode
public class Person {
 private String name;
 private Integer age;
}
```

### Способ 3: В IntelliJ IDEA → Alt + Insert → equals() and hashCode()

→ IDE сгенерирует корректный код.

```
@Override
public boolean equals(Object o) {
 if (this == o) return true;
```

```

 if (o == null || getClass() != o.getClass()) return false;
 Person person = (Person) o;
 return Objects.equals(name, person.name) &&
 Objects.equals(age, person.age);
}

@Override
public int hashCode() {
 return Objects.hash(name, age);
}

```

□ `Objects.hash(...)` — безопасно обрабатывает `null` и генерирует хороший хеш на основе переданных полей.

---

## □ Частые ошибки

- Сравнение через `==` для объектов → сравниваются ссылки, а не содержимое.
  - Забыли `@Override` → можно случайно создать перегрузку, а не переопределение.
  - Не переопределили `hashCode()` → проблемы с `HashMap`.
  - Использовали изменяемые поля в `hashCode()` → объект “сломается” в `HashSet`, если поле изменится.
  - Использовали `float/double` в `hashCode()` без округления → нестабильность из-за точности.
- 

## □ Советы

- Всегда переопределяй `equals()` и `hashCode()` **вместе**.
  - Используй `java.util.Objects` — безопасно и читаемо.
  - В IntelliJ IDEA: `Alt + Insert` → генерация методов — экономит время.
  - Тестируй поведение в `HashMap` — это частый вопрос на собеседованиях.
  - Поля в `hashCode()` и `equals()` — лучше делать `final`.
- 

## □ Контракт `equals()` и `hashCode()`

Если два объекта равны по `equals()` — их `hashCode()` **должен быть одинаковым**.

### Почему это важно?

→ `HashMap`, `HashSet`, `HashTable` используют `hashCode()` для определения “корзины”, а `equals()` — для точного сравнения.

### Антипаттерн:

```
@Override
public boolean equals(Object o) {
 // ... логика сравнения
}

// ❌ Забыли переопределить hashCode()
```

→ Объекты могут “потеряться” в HashMap.

## ❑ Частые ошибки

- Сравнение через == для объектов → сравниваются ссылки, а не содержимое.
- Забыли @Override → можно случайно создать перегрузку, а не переопределение.
- Не переопределили hashCode() → проблемы с HashMap.
- Использовали изменяемые поля в hashCode() → объект “сломается” в HashSet, если поле изменится.

## ❑ Переполнение (Overflow)

→ Происходит, когда результат операции **выходит за пределы диапазона типа**.

**Пример с int:**

```
int max = Integer.MAX_VALUE; // 2147483647
int overflow = max + 1; // → -2147483648 (переполнение!)
System.out.println(overflow);
```

→ Никакого исключения — просто “заворачивается” (как одометр в машине).

**Как избежать?**

- Используй long для больших чисел.
- Используй Math.addExact(), Math.multiplyExact() — бросают ArithmeticException при переполнении.

```
try {
 int result = Math.addExact(Integer.MAX_VALUE, 1);
} catch (ArithmeticException e) {
 System.out.println("Переполнение!");
}
```

- Для критических вычислений — используй BigInteger.

## ❑ BigDecimal — для точных вычислений (деньги!)

→ float и double **не подходят** для финансовых расчётов — из-за ошибок округления.

### Пример проблемы:

```
double a = 0.1;
double b = 0.2;
System.out.println(a + b); // → 0.30000000000000004
```

### Решение — BigDecimal:

```
BigDecimal a = new BigDecimal("0.1");
BigDecimal b = new BigDecimal("0.2");
BigDecimal sum = a.add(b);
System.out.println(sum); // → 0.3
```

☐ Всегда создавай `BigDecimal` из `String`, а не из `double`!

```
new BigDecimal(0.1) // ← НЕПРАВИЛЬНО — сохраняет неточность double
new BigDecimal("0.1") // ← ПРАВИЛЬНО
```

### Операции:

```
a.add(b)
a.subtract(b)
a.multiply(b)
a.divide(b, 2, RoundingMode.HALF_UP) // ← обязательно указывать округление!
```

→ Используй `BigDecimal` для: - Денег - Процентов - Точных научных расчётов

### ☐ `StringBuilder` — эффективная работа со строками

→ `String` в Java **неизменяем (immutable)** → каждая операция `"a" + "b"` создаёт новый объект.

### Проблема:

```
String result = "";
for (int i = 0; i < 1000; i++) {
 result += "a"; // ← создаёт 1000 промежуточных объектов!
}
```

→ Медленно и расходует память.

### Решение — `StringBuilder`:

```

StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
 sb.append("a"); // ← изменяет один объект
}
String result = sb.toString();

```

→ В **100+ раз быстрее** для больших объёмов.

#### Основные методы:

```

sb.append("text")
sb.insert(0, "prefix")
sb.delete(0, 5)
sb.reverse()
sb.toString()

```

□ Если нужна потокобезопасность — используй `StringBuffer` (но он медленнее из-за синхронизации).

---

#### □ Советы

- Всегда переопределяй `equals()` и `hashCode()` **вместе**.
  - Используй `java.util.Objects` — безопасно и читаемо.
  - В IntelliJ IDEA: `Alt + Insert` → генерация методов — экономит время.
  - Для денег — только `BigDecimal`.
  - Для конкатенации строк в цикле — только `StringBuilder`.
  - Проверь переполнение в критических местах — используй `Math.*Exact()`.
- 

#### □ Полезные ссылки

- [Oracle: Primitive Data Types](#)
- [Oracle: BigDecimal](#)
- [Oracle: StringBuilder](#)
- [Хабр: Контракт equals/hashCode](#)
- [Baeldung: Guide to hashCode\(\)](#)
- [Baeldung: BigDecimal](#)

# □ Лекция 3 — Структуры данных в Java

---

## □ Основные темы

- Полная структура коллекций Java: иерархия интерфейсов, основные реализации, сложность операций, когда что использовать

## □ Иерархия интерфейсов и основные реализации

```
package java.util;
```

```
Collection.java
```

```
public interface Collection<E> extends Iterable<E>
```

![Схема интерфейсов коллекций](images/collection-interfaces.png)

### `Iterable` → `Collection<E>`

- `List<E>`
  - `ArrayList`
  - `LinkedList`
- `Set<E>`
  - `HashSet`
  - `LinkedHashSet`
  - `SortedSet<E>`
    - `TreeSet`
- `Queue<E>`
  - `PriorityQueue`
  - `Deque<E>`
    - `ArrayDeque`

> □ `Map<K,V>` **\*\*не наследуется\*\*** от `Collection`, но входит в Collections Framework.

### `Map<K,V>`

- `SortedMap<K,V>`
  - `TreeMap`
- `HashMap`
- `ConcurrentMap<K,V>`
  - `ConcurrentHashMap`

## □ ArrayList – массив с динамическим ростом

```
class ArrayList<E> {
 transient Object[] elementData; // массив элементов
 private int size; // текущее количество элементов
}
```

При создании `List<String> list = new ArrayList<>();` – создается массив длины 0 (lazy init).

При первом добавлении – выделяется массив размером 10.

При переполнении – новый размер:



`newCapacity = oldCapacity + (oldCapacity >> 1) → +50% (не 2x!)`

□ Почему 1.5x, а не 2x? – Экономия памяти + баланс между частотой копирования и фрагментацией.

Capacity не уменьшается автоматически → используй `trimToSize()` для экономии.

## □ Алгоритмическая сложность ArrayList

```
list.get(i); // O(1) Прямой доступ по индексу
list.add(value); // O(1)* - амортизированная, O(n) - худшая при ресайзе
list.add(i, value); // O(n) Сдвиг всех элементов справа
list.remove(i); // O(n) Сдвиг всех элементов слева
list.contains(v); // O(n) Линейный поиск
iterator().next(); // O(1) Быстрый обход
```

## □ LinkedList – двусвязный список

Двусвязный список. Прыгает по памяти, поэтому работает медленно.

Внутреннее устройство:

```
class Node<E> {
 E item;
 Node<E> next;
 Node<E> prev
}
```

Каждый элемент – отдельный объект в куче → прыжки по памяти → плохая локальность → медленнее, чем ArrayList в большинстве случаев.

Добавление/удаление в начале/конце – O(1)

Доступ по индексу – O(n)

□ Цитата от Joshua Bloch (создатель LinkedList):

“Does anyone actually use LinkedList? I wrote it, and I never use it.”

Когда использовать?

□ Только если ты очень часто вставляешь/удаляешь в начале или середине списка, и не нужен доступ по индексу.

□ В 95% случаев – ArrayList будет быстрее и эффективнее.

## □ Queue / Deque – интерфейсы для очередей

ArrayDeque – циклический буфер (ring buffer).

- Все операции в начале/конце – O(1)
- Реализует Deque → можно использовать как стек или очередь.
- Внутри – массив, растёт по формуле: `newSize = 2 * oldSize + 2`
- Не потокобезопасен

□ Лучший выбор для стека/очереди в однопоточном коде → быстрее LinkedList.

## □ HashMap – хеш-таблица

Внутреннее устройство:

```
class HashMap<K,V> {
 Node<K,V>[] table; // массив "корзин"
 int size; // количество элементов
}

static class Node<K,V> implements Map.Entry<K,V> {
 final int hash;
 final K key;
 V value;
 Node<K,V> next; // следующий узел в цепочке
}
```

- Используется цепочечная адресация.
  - Изначально table = null → создаётся при первом put() → размер 16.
  - Load factor = 0.75 → при достижении 12 элементов → ресайз до 32.
  - Ресайз → в 2 раза.
  - Коллизии → цепочки (linked list) → с Java 8, если >8 элементов → красно-чёрное дерево.
- hashCode() → определяет корзину
- equals() → проверяет, тот ли ключ

## □ TreeMap – отсортированная мапа

- Основана на красно-чёрном дереве → сбалансированное BST.
  - Все операции –  $O(\log n)$
  - Ключи должны реализовывать Comparable или передаваться Comparator
  - Итерация – в отсортированном порядке
- □ Используй, если нужна сортировка по ключам или range-запросы (subMap, headMap, tailMap)

## □ HashSet, TreeSet, LinkedHashMap – обёртки

### HashSet

- Внутри использует HashMap<E, Object> → значения – заглушки (PRESENT).
- Сложность операций – как у HashMap:  $O(1)$

### TreeSet

- Внутри использует TreeMap<E, Object>
- Сложность –  $O(\log n)$

### LinkedHashMap

- Наследуется от HashMap

- Добавляет двойной связанный список для сохранения порядка вставки (или доступа)
- Итерация – в порядке вставки (или LRU, если `accessOrder=true`)
- Небольшой overhead на поддержание списка при добавлении → остальные операции – как у `HashMap`
- ☐ Отлично подходит для LRU-кэшей → переопредели `removeEldestEntry()`

## ☐ Практические советы на собеседовании

Почему `ArrayList` чаще `LinkedList`?

→ Локальность данных, меньше аллокаций, быстрее в реальных сценариях.

Что будет, если не переопределить `hashCode()` и `equals()` для ключа в `HashMap`?

→ Объекты не будут находиться → нарушение контракта.

Как работает `ConcurrentHashMap`?

→ Разделён на сегменты (до Java 8) → с Java 8 – использует CAS + `synchronized` на уровне корзины → высокая конкурентность

Fail-Fast vs Fail-Safe?

→ `ArrayList.iterator()` – fail-fast → `ConcurrentModificationException`

→ `ConcurrentHashMap.keySet().iterator()` – fail-safe → работает с копией

Как уменьшить `capacity` `ArrayList`?

→ `trimToSize()`

## ☐ Рекомендуем к прочтению

☐ `Effective Java` – Joshua Bloch (главы 3, 6 – коллекции и `equals/hashCode`)

☐ `Java Concurrency in Practice` – Brian Goetz (глава 5 – коллекции в многопоточке)

☐ [Oracle Java Collections Tutorial](https://docs.oracle.com/javase/tutorial/collections/?spm=a2ty\_o01.29997173.0.0)

# ☐ Лекция 4 - Дженерики в Java

### 1. \*\*Синтаксис использования\*\*

```
```java
```

```
List<String> list = new ArrayList<>(); // Diamond operator (Java 7+)
```

→ Компилятор гарантирует, что в список можно добавлять **только String**, а при извлечении — **не нужен каст**.

2. Проблема без дженериков: `Object[]` — небезопасен

```
Object[] objects = new Integer[10]; // Компилируется!
objects[0] = "543";                 // Runtime: ArrayStoreException!
```

→ **Ошибка только в рантайме** → нарушение type safety.

□ **Идея дженериков:** если код компилируется — **никаких** `ClassCastException` или `ArrayStoreException` в рантайме быть не должно.

3. Type Erasure (стирание типов)

- Дженерики существуют **только на этапе компиляции**.
- В байт-коде: `List<String>` → `List` (raw type), все операции — с `Object`.
- **Исключение:** информация о дженериках сохраняется в **сигнатурах методов и классов** (через `Signature` атрибут) → доступна через `Reflection`.

□ Поэтому **нельзя**: - Создать массив дженериков: `new T[]` - Проверить тип в рантайме: `if (list instanceof List<String>)` → ошибка компиляции

4. Инвариантность дженериков

```
List<Object> list = new ArrayList<Integer>(); // □ Ошибка компиляции!
```

→ Даже если `Integer` — подтип `Object`, `List<Integer>` **не является подтипом** `List<Object>`.

Это называется **инвариантность**: `List<A>` и `List` не связаны, даже если `A extends B`.

5. Wildcards: ? extends и ? super

```
List<? extends Number> list = new ArrayList<Integer>();
Number n = list.get(0); // □ OK
list.add(42);           // □ Запрещено! (может быть List<Double>)
list.add(null);         // □ Единственное разрешённое значение
```

□ ? extends T — **Producer (только чтение)**

```
List<? super Integer> list = new ArrayList<Number>();
list.add(42);             // □ OK
Number n = list.get(0);   // □ Тип Object — не знаем точный тип
```

□ ? super T — **Consumer (только запись)**

□ PECS (Producer Extends, Consumer Super)

- Если объект **возвращает** значения (producer) → ? extends T
 - Если объект **принимает** значения (consumer) → ? super T
 - Если и то, и другое → **без wildcard**, просто T
-

6. Объявление дженериков

```
public interface List<E> { ... }  
public class Box<T> { ... }
```

На уровне класса:

```
public static <E> E max(List<E> list, Comparator<E> comparator) { ... }
```

На уровне метода:

Ограничения (bounds):

- E extends Comparable<E> — верхняя граница (только подтипы Comparable)
- E super SomeClass — **нельзя** на уровне объявления типа!
→ super работает **только в wildcards**: List<? super Number>

□ Правильно:

```
public static <E extends Comparable<? super E>> E max(List<E> list)
```

→ Это позволяет сравнивать E, даже если compareTo реализован в родителе (например, LocalDateTime наследует Comparable<ChronoLocalDateTime>).

7. Соглашения по именам типов

- E — Element (в коллекциях)
 - K — Key
 - V — Value
 - T — Type
 - R — Return type
 - S, U, W — дополнительные типы
-

8. Примеры сравнения

| Выражение | Можно присвоить | Можно добавить | Можно прочитать как |
|------------------------------|--------------------------------|------------------|---------------------|
| List<Number> | только List<Number> | Number и подтипы | Number |
| List<? extends Number> | List<Integer>, List<Double> | только null | Number |
| List<? super Number> | List<Number>, List<Object> | Number и подтипы | Object |
| List<?> | любой List | только null | Object |

9. Важно помнить

- **Raw types (List) — избегать!** Они отключают проверку типов.
- **Дженерики не работают с примитивами** → используй List<Integer>, а не List<int>.
- **Массивы и дженерики несовместимы** → предпочитай коллекции.