Конспект спринта №3

CodeStyle

1. Длина строки кода

Оптимальная длина строки, которая помещается полностью на экране современных мониторов — не более 120 символов. Так же такая длина удобна при одновременной работе с двумя файлами, расположенными бок о бок на экране широкоформатного монитора.

2. Перенос строки

Если код не влезает в одну строку, его нужно перенести на следующую, это называется перенос строки (англ. *line-wrapping*). Иногда так делают и просто для улучшения визуального восприятия кода.

Нет единственно верного способа перенести строку, который работал бы во всех ситуациях.

При переносе строк придерживайтесь следующих правил:

- Перенос строки должен происходить перед оператором (например, оператором сложения, конкатенации, инкремента, оператором доступа к членам класса (оператор точка) и т.д.). Исключение составляет только оператор присваивания `=`.

```
// Перенос строки перед оператором конкатенации "+".
final String lineWrappingExample = "Пример очень длинной строки которую необходимо разбить на две и осуществить "
        + "перенос второй части. Разбиение производится до оператора конкатенации: +";

// Перенос строки перед оператором точки "."
long longStringsCount = listOfStrings.stream()
        .filter(string -> string.length() > 100)
        .peek(string -> System.out.println("Filtered string: " + string))
        .count();

// Перенос строки перед оператором "логическое и" (&&)
if(firstConditionOccurs
        && (secondConditionOccurs || thirdConditionOccurs)) {
    return true;
}
```

- Перенос строки должен происходить после запятой, разделяющей параметры или аргументы.

```
public static void lineBreak(int firstParameter, int secondParameter,
        int thirdParameter)
```

3. Отступы при переносе строк

При разделении строки кода и переносе её частей на несколько строк следует придерживаться следующих правил отступа:

- При переносе на новую строку нужно отступить на 8 пробелов от начала первой строки.

```
String text = "Например, при разбиении текста на две строки "
        + "вторая часть размещается с отступом в 8 пробелов от начала первой строчки кода.";
```

- Но если элементы, которые переносятся на новую строку, синтаксически одинаковы, отступ можно оставить на том же уровне, что и в первой строке.

```
// Параметры метода перенесены на новые строки. Но secondArgument и
// thirdArgument это тоже параметры, как и firstArgument. Поэтому
// отступы для secondArgument и thirdArgument располагают их на
// том же уровне, что и firstArgument
public static void someNameOfMethod(List<String> firstArgument,
                                    Long secondArgument,
                                    Set<Integer> thirdArgument) {
    // ...
}
```

4. Комментарии

В Java есть два типа комментариев:

- Однострочные — `//`

  Допустимо использовать только для небольших комментариев. От кода отделены одним или более пробелом. После `//` идёт один пробел и далее комментарий с большой буквы.

```
public static void main(String[] args) {
  // Это однострочный комментарий
  int a = 42; // Тоже однострочный комментарий
}
```

- Многострочные — `/* */`

  Для многострочных комментариев:

```
/*
 * Очень важный комментарий,
 * который содержит много строк
 *
 *    one
 *        two
 *            three
 */
```

5. Объявление переменных

Одна строка — одна переменная.

```
int a; // Правильное объявление переменной
int b;

int a, b; // А так делать не нужно
```

6. Объявление методов и классов

При объявлении методов и классов необходимо придерживаться следующего подхода:

- Открывающая фигурная скобка стоит на той же строке, что и название класса/метода, в конце строки, отделена одним пробелом.
- Закрывающая фигурная скобка стоит на отдельной строке. Выравнивание — на том же уровне, что и описание класса/метода.
- Открывающая круглая скобка у метода стоит вплотную к его названию. Аргументы метода при перечислении разделены одним пробелом.

```
public class Cat { // Фигурная скобка отделена одним пробелом

    // Нет пробела между именем метода и круглой скобкой
    public Cat getCat(String name, int age) {
        return new Cat();
    } // Выравнивание закрывающей скобки - по началу метода

} // Выравнивание закрывающей скобки - по началу класса
```

7. Каждое выражение должно быть на отдельной строке

```
// Так - хорошо:
int a = 10;
int b = a + 10;

// Так - плохо:
int a = 10; int b = a + 10;
```

8. Эталонные примеры сложных операторов

- `if-else`

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

- `for`

```
for (initialization; condition; update) {
    statements;
}
```

- `while`

```
while (condition) {
    statements;
}
```

- `try-catch`

```
try {
    statements;
} catch (Exception e) {
     statements;
}
```

9. Пустые строки

Пустые строки часто используют для улучшения читаемости кода. Без них большие фрагменты кода сложно воспринимать.

По одной пустой линии ставим:

- Между определениями методов

```
public void method1() {
}

public void method2() {
}

// Методы разделены пустой строкой
```

- Между блоком с переменными и блоком с непосредственно кодом

```
public void method1() {
    int firstOperand = 10;
    int secondOperand = 20;

    int total = firstOperand * secondOperand;
}

// Код разделён пустой строкой на две части:
// блок с объявлением переменных и блок вычислений
```

- Между логическими частями кода внутри метода

10. Пробелы

Пробелы ставим:

- После запятых в перечислениях аргументов методов `int a, int b`
- Между зарезервированным словом/оператором и открывающей скобкой

```
while (true)
if (false)
```

- Пробелами отделяем все бинарные операторы: `+ - / * % =`

```
int a = 10 + 2 * 8 - 4 / 6 % 2;
String hello = "Привет" + a + "И еще привет";
```

Не ставим пробелы перед унарными операторами

```
a++;
b--;
```

11. Именования

Все названия даются на английском языке, без использования транслитерации. Классам, методам и переменным нужно давать точные, «говорящие» имена — чтобы они были понятны не только вам, но любому программисту, который будет читать ваш код.

- Классы

  В названиях классов нужно использовать только имена существительные или словосочетаниями с ними, написанные по-английски, в UpperCamelCase.

  ```
  class Apple
  class FinancialCalculator
  ```

- Методы

  Названия методов должны начинаться с глаголов. Все слова нужно писать по-английски, в lowerCamelCase. Тип метода указывать в названии не нужно.

  ```
  get();
  setNewValue();
  eatApple();
  ```

- Переменные

  Названия переменных пишутся по-английски, в lowerCamelCase. Они должны чётко отражать содержимое этих переменных. При этом тип переменной указывать в названии не нужно.

  Имя, состоящее из одного символа, можно использовать только в общепринятых случаях (например, `i` и `j` для циклов или `e` в блоке обработки исключений).

  ```
  int result;
  String myCity;
  for (int i = 0; i < 10; i++)
  ```

- Константы

  Константы пишутся в верхнем регистре, каждое новое слово отделено `_`.

  ```
  int MY_AWESOME_CONSTANT = 42;
  ```

12. Отступы

В качестве отступа всегда нужно использовать четыре пробела. Не один, не два и не символ табуляции. 1 отступ = 4 пробела.

13. Символ переноса строки

В качестве символа переноса строки используем `\n`, если задача явно не требует другого.

Наследование и инкапсуляция

Java — объектно-ориентированный язык. Это значит, что в основе всех программ лежат объекты и классы. Под классом понимается шаблон или общее описание атрибутов и методов, а объект — это конкретный экземпляр класса. Из набора взаимодействующих между собой объектов состоит любая программа в Java.

В методологии объектно-ориентированного программирования (ООП) есть четыре ключевых принципа:

- инкапсуляция,
- наследование,
- полиморфизм,
- абстракция.

Инкапсуляция

Инкапсуляция важна по следующим причинам:

- Не нужно знать, как устроен внутренний процесс, достаточно внешнего интерфейса.
- Высокая скорость понимания кода.
- Удобство работы с кодом.
- Минимизация ошибок и надёжное хранение данных.

В коде инкапсуляция реализуется за счёт следующих инструментов:

- пакетов,
- модификаторов доступа,
- методов, включая `set-` и `get-` методы.

Пакеты структурируют классы в программе. Сокрытие данных достигается с помощью модификаторов доступа: `private`, `public`, `protected` и модификатора по умолчанию `default` (`package-private`) — каждый из них определяет свой уровень доступности. Методы позволяют реализовать интерфейс. С помощью `set-` и `get-` методов можно настроить доступ к данным.

Пакеты

Пакет — это папка с файлами классов. Пакеты объединяют классы по смыслу и назначению. С помощью пакетов можно избежать конфликтов при использовании классов с одинаковыми именами. Если класс не относится ни к одному пакету, то будет использован пакет по умолчанию.

Чтобы взаимодействовать в коде с классами из пакетов стандартной библиотеки, их нужно импортировать или, другими словами, подключить. Для этого надо ключевое слово `import`, имя пакета и имя класса. Если не импортировать класс, то при создании объекта произойдёт ошибка — программа не скомпилируется.

При помощи символа звёздочки — `*` можно импортировать пакет целиком.

Чтобы создать новый пакет в среде разработки IDEA, нужно кликнуть правой клавишей на папку `📁src` (или папку внутри) и выбрать *New → Package.* После этого надо ввести название пакета, например — `practicum`. После этого в меню появится папка с таким именем. В эту папку можно перетащить нужные классы. При переносе нажать кнопку *Refactor.* После того как классы перенесены в пакет, в первой строке кода появится ключевое слово `package` и имя пакета.

Как и папки в файловой системе, пакеты могут быть вложенными. Их название записывается через точку. При именовании пакетов принято использовать обратный порядок. Например, для сервиса “Catify” на сайте “mycompany.com” пакет будет называться `com.mycompany.catify`. В пакете с именем домена платформы будет подпакет с названием компании и подпакет с названием сервиса.

Модификаторы доступа

Модификаторы доступа позволяют скрывать или, наоборот, делать общедоступными разные части программы: методы, поля и классы.

Всего в Java четыре модификатора. Для обозначения трёх из них в коде используются ключевые слова `public`, `private`, `protected`. Четвёртый не имеет обозначения — это модификатор по умолчанию `default` или `package-private`.

Если в одном файле хранится несколько классов, то только один из них может иметь модификатор `public`. Имя файла при этом должно совпадать с именем класса с `public.` Другие классы должны иметь более низкий уровень доступа.

Геттеры и сеттеры

Геттеры и сеттеры нужны для работы с полями класса, закрытыми модификатором `private`. Чтобы взаимодействовать с защищённой переменной, но при этом не открывать к ней доступ, и используются `get-` и `set-`методы.

`Get-`методы (от англ. *get* — получать) позволяют получать значения из закрытых переменных, а `set-` методы (от англ *set* — установка) сохранять в такие переменные новые значения. В названии таких

методов принято указывать слова `get` или `set` и имя переменной.

Код внутри `get-` и `set-`методов может быть любым. С его помощью можно реализовать дополнительную логику: обновление других переменных, вывод в консоль информации и многое другое. На внешнем интерфейсе это не отразится. То, насколько сложные операции происходят внутри методов, не должно затрагивать интересы пользователей.

Для получения значений из переменных типа `boolean` вместо `get` используется глагол `is` (для установки значений также используется префикс `set-`).

Геттеры и сеттеры ничем и не отличаются от обычных методов. Можно прописать получение и установку значений закрытых переменных с помощью методов с любыми другими именами и программа скомпилируется. Однако любая нотация кроме `get-` и `set-` считается некорректной, так как не позволяет разработчику быстро понять, с чем он работает, а заставляет дополнительно разбираться в логике работы методов.

Наследование

Благодаря наследованию дочерние классы автоматически приобретают функционал класса-родителя. Не нужно раз за разом прописывать одни и те же поля и методы — их можно передать по наследству.

Чтобы передать все характеристики родительского класса `Animal` другому классу, например, `Canidae`(англ. «псовые»), нужно применить принцип наследования. Для этого требуется ключевое слово `extends`. Класс `Canidae` наследует все поля и методы класса `Animal`, при этом у него появляются свои, которые также можно передать по наследству.

Важно запомнить, что у классов в Java может быть сколько угодно предков, но только один родитель. То есть при помощи ключевого слова `extends` можно наследовать только от одного класса. Написать `class Fox extends Animal, Canidae` не получится.

При внесении изменений в родительский класс они автоматически перейдут по наследству. К примеру, если мы поменяем значение поля возраста в классе `Animal` с 0 на 10, то объекту класса-наследника сразу же исполнится 10 лет.

Если обобщить, то вот что можно делать в классе-наследнике:

- Унаследованные поля можно использовать напрямую, как и любые другие поля.
- Можно объявить поле в подклассе с тем же именем, что и поле в суперклассе, таким образом скрывая его (но делать это не рекомендуется, это дублирование кода).
- Можно объявить в подклассе новые поля, которых нет в суперклассе.
- Унаследованные методы можно использовать напрямую в классе-наследнике.
- Можно написать новый метод в подклассе, который будет иметь ту же сигнатуру, что и в суперклассе, таким образом переопределив его.
- Можно объявить в подклассе новые методы, которых нет в суперклассе.
- Можно написать конструктор подкласса, который будет вызывать конструктор суперкласса.

Переопределение метода

Чтобы изменить поведение метода суперкласса, его можно **переопределить** внутри подклассов. Механизм переопределения предполагает, что сигнатура остаётся прежней, при этом в тело метода вносятся изменения, а доступ к нему может быть расширен.

Переопределение метода помечается в коде с помощью аннотации `@Override` (от англ. *override* — «переопределение, ручная коррекция»). При вызове переопределённого метода будет задействована реализация из класса-наследника.

> 💡 Аннотации в Java — это специальная форма метаданных в коде. Они начинаются с символа `@` и сообщают компилятору дополнительную информацию. Например, если `@Override` помечает метод как переопределённый, то `@Deprecated` как устаревший. Аннотациями можно помечать методы, классы и переменные.

Отсутствие аннотации `@Override` при переопределении метода — не ошибка. Однако её принято использовать, так как у неё есть два полезных свойства:

- Явно обозначены переопределённые методы — их легко отличить от остальных методов класса.
- Если в переопределённом методе, помеченном `@Override`, поменяется сигнатура (неважно где: в классе-родителе или классе-наследнике), то при компиляции появится сообщение об этом. `Method does not override from its superclass` — означает, что метод больше не переопределяется из своего суперкласса.

Ключевое слово super

Ключевое слово `super` позволяет вызвать метод или конструктор суперкласса, а также обратиться к его полям. Для обращения к методам класса-родителя через `super` нужно применить точечную нотацию — `super.someMethod()`. Это может быть полезно, когда нужно использовать функционал родительского метода и дополнить его новыми действиями.

С помощью ключевого слова `super` и точечной нотации можно обратиться также к скрытым полям родительского класса — `super.someField`. Необходимость обращаться через `super` к скрытым полям суперкласса — достаточно редкое явление. Во-первых, потому что увлекаться сокрытием полей не принято — так возрастает вероятность ошибки. Во-вторых, поля классов всё-таки лучше инициализировать не при объявлении, а в конструкторе.

Через `super` можно также вызвать конструктор класса-родителя. С помощью `super()` — без параметров, а с помощью `super(parameter list)` — с параметрами.

Если в классе-родителе есть конструктор без параметров, то в конструкторе подкласса компилятор вызовет его автоматически, неявно. Явно вызывать конструктор суперкласса без параметров — необходимости. Единственное исключение: для навигации в коде — зажав `ctrl` и кликнув на `super()`, можно быстро перейти в конструктор родителя.

Совсем другая история — наличие в классе-родителе конструктора с параметрами. В этом случае обойтись без вызова суперконструктора в классе-наследнике не получится — произойдёт ошибка компиляции. Чтобы решить эту проблему, нужно либо создать в суперклассе конструктор без параметров, либо вызвать родительский конструктор.

Чтобы не дублировать код конструктора родителя в классе-наследнике, можно вызвать суперконструктор в конструкторе подкласса. Тогда общие поля будут проинициализированы в классе-родителе, а индивидуальные — в наследнике.

> 💡 Важное правило! Вызов конструктора класса-родителя через `super` должен быть первой строкой в конструкторе класса-наследника. Иначе произойдёт ошибка `java: call to super must be first statement in constructor` — англ. «вызов `super` должен быть первым оператором в конструкторе». Так компилятор проверяет, что родительский класс был проинициализирован корректно ещё до создания дочернего класса.

Если не объявить в классе конструктор — Java сгенерирует конструктор по умолчанию. У него нет параметров, в коде его не видно, но именно он вызывается при создании объекта с помощью `new`.

Инициализировать поля при объявлении и обходиться только конструктором по умолчанию — можно, но не принято. Такой код неудобно читать — особенно если в классе много полей.

This

Конструктор с параметрами сохраняет переданное значение в поле класса. Поэтому для параметров конструктора удобнее всего использовать похожие имена. Например, такие, как `name` и `newName`. Именно так мы создавали конструкторы с параметрами до этого момента. Однако есть более удобный способ — при помощи ключевого слова `this` (англ. «этот»): `this.name = name;`

Слово `this` — это ссылка на объект текущего класса. Оно позволяет обратиться к полю, методу или конструктору класса внутри него самого. Обращаться к полям в конструкторе через `this` — общепринятое соглашение.

В теле конструктора не только инициализируются значения полей. В нём могут выполняться какие-то действия, например, подсчёт количества объектов или печать.

Через `this` так же, как и через `super`, можно обратиться к другим конструкторам, чтобы не дублировать код. Вызов конструктора через `this` должен быть на первом месте или сразу после вызова конструктора родительского класса через `super`.

Ещё один способ оптимизировать код — передавать в конструктор не поля по отдельности, а сразу объект. Когда параметр — объект, то внутри конструктора нужно обратиться к его полям через их названия и точечную нотацию. Точно также в конструктор подкласса можно передавать объект класса-родителя.

Автоматическачя генерация кода

Команда автогенерации позволяет добавить в код конструктор, `get-` и `set-`методы или переопределить метод. При нажатии комбинации перед вами должно появиться меню из девяти пунктов (доступ к нему можно также получить, щёлкнув по классу правой кнопкой и выбрав *Generate...*).

Для автоматической генерации кода в IDEA нужно запомнить следующую комбинацию:

`alt+Insert` — для Windows и Linux,

`⌘ + N`, `ctrl+Enter` — для Mac OS.

Чтобы переопределить один или несколько методов в подклассе при работе в IDEA, можно использовать комбинацию `Ctrl+O` (для Windows и Linux) или `^ (Control) + O` (для Mac OS)**.** При её использовании можно сразу выбрать все методы, поведение которых нужно изменить. Также IDEA сама добавит аннотацию `@Override` к этим методам.

Класс Object

Переменная типа Object

У всех классов в Java есть общий предок — **класс** `Object`. Он входит в пакет `java.lang`. Наследование от класса `Object` происходит по умолчанию.

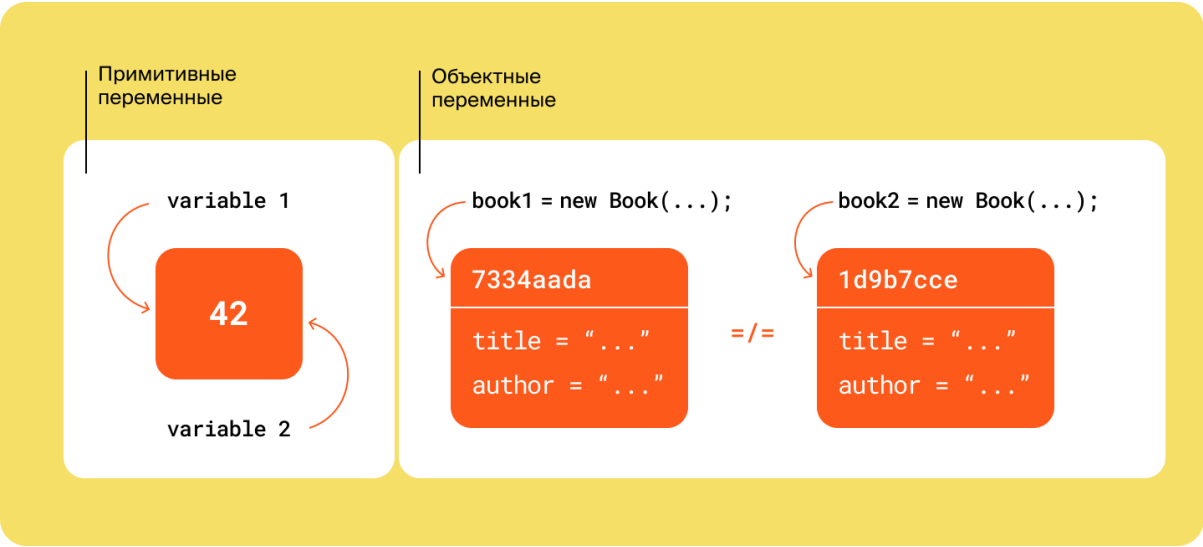
Переменной типа `Object` можно присвоить любое ссылочное значение. Это может быть объект любого класса, например, список, массив или ваш собственный, а также значение примитивного типа после автоупаковки в класс-обёртку.

Благодаря тому что `Object` может хранить любой объект, его удобно использовать как параметр универсального метода, который должен принимать объекты разных классов. Это можно увидеть в некоторых классах стандартной библиотеки, например, в методах хеш-таблиц, таких как `get(Object key)`, `remove(Object key)` или `containsValue(Object value)`, где в качестве ключа и значения могут быть любые объекты.

`equals(Object)`

Метод `equals(Object)` по сути аналогичен оператору `==`. Разница в том, что при помощи `==` сравниваются значения примитивных типов, а при помощи `equals(Object)` — ссылочных. Для проверки объектов на равенство `==` не подходит.

В теме о типах в Java мы подробно разбирали, что в объектных переменных сохраняется не само значение, а ссылка на него — не кофемашина, а буклет с адресом, где её забрать. При помощи оператора `==` можно сравнить только ссылки — буклеты, а не содержание — саму технику. Из-за этого, даже если объекты будут идентичны, их сравнение при помощи `==` даст неверный результат:



Ссылки — только один из аспектов, по которому можно сравнить объекты между собой. Поэтому оператора `==` всегда будет недостаточно. Чтобы получить корректный результат, нужен метод `equals(Object)`.

Переопределение `equals`

Для сравнения объектов собственных классов метод `equals(Object)` нужно переопределить.

> 💡 Во многих классах стандартной библиотеки метод `equals(Object)` уже переопределён. Это даёт возможность применять его автоматически. Например, в классе `String` можно применять `equals(Object)` для сравнения строковых переменных без дополнительных действий.

- Начинаем с аннотации `@Override` и стандартной реализации метода. Так вы сразу вычислите, не имеете ли дело с одним и тем же объектом. Если это так, то нет смысла дальше проверять все поля на равенство, можно сразу вернуть положительный результат.
- Следующим шагом нужно проверить, не была ли передана в метод `equals(Object)` пустая ссылка `null` вместо объекта. Если аргумент равен `null` — можно сразу возвращать отрицательный результат. Если вовремя не отловить `null`, и продолжать дальнейшую проверку пустой ссылки, это приведёт к ошибке исключения `NullPointerException`.
- Поскольку базово метод `equals(Object)` принимает в качестве аргумента объекты любых классов, дальше требуется проверить, что в него передан экземпляр нужного. Первая часть переопределения метода завершена — исключено, что это один и тот же объект, экземпляры разных классов или передана пустая ссылка. Эти проверки нужно провести вне зависимости от того, объекты каких классов вы сравниваете между собой.
- Далее нужно привести переданный объект к тому классу, где переопределяется `equals(Object)`. Приведение любых типов, в том числе ссылочных, осуществляется с помощью круглых скобок. Приведение типов нужно, чтобы получить доступ к полям второго объекта. После этого можно обращаться к ним по выбранному имени (`otherBook`), используя точечную нотацию.
- Для сравнения полей удобно пользоваться методом `Objects.equals(Object, Object)`. Утилитарный класс `Objects` (с `s` на конце — подробнее о нём можно почитать здесь) содержит набор вспомогательных методов, в том числе `equals(Object, Object)`. Этот метод сначала проверяет, не равны ли переданные аргументы пустым ссылкам, и если нет — сравнивает их. Поля примитивных типов сравниваем через оператор `==`.

Контракт метода

1. **Правило рефлексивности** — объект должен быть равен самому себе. То есть вызов `x.equals(x)` должен всегда возвращать `true`.
2. **Правило симметричности** — «от перестановки мест слагаемых сумма не меняется». Результат сравнения объектов не зависит от того, в каком порядке они расположены. Вызов `x.equals(y)` должен возвращать `true` в то же время, когда вызов `y.equals(x)` возвращает `true`.
3. **Правило логической транзитивности —** если два объекта равны и один из них равен третьему, то все три объекта равны. Так, если вызов `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, то вызов `x.equals(z)` также должен вернуть `true`.
4. **Правило согласованности —** если не менять данные сравниваемых объектов, то и результат их сравнения должен быть всегда одинаков. То есть множественный вызов `x.equals(y)` должен возвращать один и тот же результат, пока данные полей объектов `x` и `y` неизменны.
5. **Правило «на ноль делить нельзя»** — ни один из сравниваемых объектов не может быть равен `null`. Это значит, что вызов `x.equals(null)` должен всегда возвращать `false`.

`hashCode()`

Каждый объект в программе можно представить в виде некоторого целого числа. Процесс вычисления такого числа называется **хешированием**, а его результат — **хешем**. Этим занимается метод `hashCode()` — генерирует хеш для объектов.

Базовая реализация метода `hashCode()` стремится создать уникальный хеш для каждого объекта, в том числе для идентичных. При переопределении нужно это исправить.

> 💡 У всех стандартных ссылочных типов данных в Java (`String`, `Integer`, `Double` и т.д.) методы `equals(Object)` и `hashCode()` уже корректно переопределены. Поэтому их можно спокойно использовать с коллекциями `HashMap`, `HashSet` и прочими.

Чтобы хеш-коды разных объектов отличались, а одинаковых — совпадали, нужно вычислять хеш в связке с методом `equals(Object)`. Оба метода должны зависеть от одних и тех же полей. Отсюда и взялось правило, что при переопределении `equals(Object)` лучше сразу переопределять метод `hashCode()`.

Более простой и самый распространённый вариант переопределения `hashCode()` — через метод `hash(Object... values)` уже знакомого вам класса `Objects`. Этот метод генерирует хеш-код для последовательности переданных в него значений. Реализация с его использованием лаконична — вызываем метод и передаём в него нужные поля (те же, что и в методе `equals(Object)`).

Контракт

У метода `hashCode()` есть контракт, которым нужно руководствоваться при его ручном переопределении. Он включает три правила:

- Если при сравнении методом `equals(Object)` объекты оказались равны, то `hashCode()` должен возвращать у каждого из них одно и то же число.
- Метод `hashCode()` должен возвращать одно и то же целое число до тех пор, пока значения полей, используемых в методе `equals(Object)` того же класса, остаются прежними.
- Нужно стремиться к тому, чтобы у объектов, которые не равны при сравнении `equals(Object)`, были разные хеш-коды, но учитывать, что они могут совпасть. Поэтому если у двух объектов одинаковые

хеш-коды, нельзя утверждать, что объекты равны. Точный результат покажет только метод `equals(Object)`.

Если подытожить, переопределяя метод `hashCode()`, важно проверить, чтобы для равных объектов, всегда возвращался одинаковый хеш-код, а для разных по возможности разные.

Пара `equals(Object)` - `hashCode()` и поиск в хеш-таблицах

Правильно реализованная пара `equals(Object)` и `hashCode()` делает возможным поиск объектов в списках и хеш-таблицах.

Несмотря на то, что элементы добавлены — методы поиска в списках `contains()` и в таблицах `containsKey()` не могут их найти.

Если `equals(Object)` не переопределён — используется его базовая реализация, которая сравнивает только ссылки объектов и выдаёт некорректный результат. Если же переопределить метод `equals(Object)`, с его помощью метод `contains(Object)` класса `ArrayList` сможет один за другим сверить каждый элемент списка с искомым и легко найти нужный объект.

Для поиска в хеш-таблице нужно переопределить ещё и `hashCode()`, иначе будет вызываться его базовая реализация, когда для одинаковых объектов возвращается разный хеш-код.

`toString()`

Цель переопределения `toString()` — получить простую по форме и ёмкую по содержанию информацию о содержании объекта. Реализация метода при этом зависит от разработчика. Также как `equals(Object)` и `hashCode()`, метод `toString()` рекомендуется всегда переопределять. Его базовая реализация в классе `Object` не информативна — не учитывает поля каждого класса.

Контракт метода

Так как у `toString()` нет строгого контракта, при его переопределении принято руководствоваться такими рекомендациями:

1. **Единый формат.**

   Когда вывод `toString()` построен во всех классах по одной и той же логике — код удобно читать и воспринимать. Детали могут отличаться, но основа должна быть единой. В начале указывается имя класса, затем в фигурных скобках названия полей и их значения. Важно запомнить, что `toString()` не должен влиять на состояние объекта. Он работает в режиме «только чтение» — не меняет значения полей и не проводит с ними расчёты.

2. **Лаконичность и информативность.**

   - В реализацию `toString()` стоит включать только те поля, которые содержат ключевую или определяющую информацию. Статические или вспомогательные поля можно опустить.
   - Реализацию `toString()` важно поддерживать в актуальном состоянии. Добавлять поля или удалять их по мере необходимости.
   - Некоторые поля могут содержать объёмные данные. Нет практического смысла в том, чтобы выводить их полное или даже сокращённое содержание. Можно отобразить их длину.

3. **Профилактика исключений** `NullPointerException`**.**

Оператор конкатенации `+` умеет работать с пустыми ссылками `null`, поэтому при сложении строк с пустой ссылкой ошибки не будет. Исключение `NullPointerException` может возникнуть в том случае, если у одного из полей вызывается метод.

4. **Форматирование данных.**

Некоторые типы данных требуют дополнительного форматирования. Это актуально, к примеру, для массивов. Они наследуют базовую реализацию `toString()` и чтобы посмотреть их содержание, лучше дополнительно вызвать метод `toString(Object[] a)` класса `Arrays`. Этот метод проверяет массив на `null`, и, если всё в порядке, возвращает его текстовое представление.

`println()` и `toString()`

Через метод `println()` можно сразу получить текстовое представление объекта. Вызывать `toString()` при этом необязательно. Так как все классы наследуют `toString()` от `Object`, метод `println()` может преобразовать в строку объект любого класса.

Многие классы стандартной библиотеки `String`, `Integer`, `Double`, `Short` и другие уже имеют переопределённый `toString()`. При передаче их объектов в метод `println()` выводятся непосредственно значения, а не название класса с хеш-кодом. При конкатенации — оператор `+` также автоматически вызывает `toString()`.