Шпаргалка по основам

Шаблон программы

```
/** App - имя класса. Файл с этим классом внутри должен называться App.java */
public class App {

  /** Точка входа в программу. у метода main всегда одна и та же сигнатура */
  public static void main(String[] args) {
    // Тут пишем код
  }
}
```

После того, как сохранили файл, как `App.java` , компилируем и выполняем командами

```
javac FirstApp.java
```

```
java FirstApp
```

Можно объеденить:

- Для unix/linux/mac

```
javac FirstApp.java; java FirstApp
```

- Для windows

```
javac FirstApp.java & java FirstApp
```

Зарезервированные слова

abstract continue for new switch
assert default goto package synchronized
boolean do if private this

break double implements protected throw
byte else import public throws
case enum instanceof return transient
catch extends int short try
char final interface static void
class finally long strictfp volatile
const float native super while

Примитивные типы

Тип	Размер	Default	min	max
byte	8 бит	0	-128	127
short	16 бит	0	-32768	32767
int	32 бита	0	-2_147_483_648	2_147_483_647
long	64 бита	0L	-9_223_372_036_854_775_808L	9_223_372_036_854_775_
float	32 бита	0.0F	1.40239846e-45f	3.40282347e+38f
double	64 бита	0.0	4.9406564584124654E-324	1.79769313486231570e+3
boolean	1 бит	false		
char	16 бит	false		

Операции

Операция	Синтаксис	Комментарий
Сложение	`a + b`	
Вычитание	`a - b`	
Умножение	`a * b`	
Деление	`a / b`	Не забываем приводить к `float` или `double`
Остаток от деления	`a % b`	

Операция	Синтаксис	Комментарий
Инкремент / Декремент	`a++` ; `a--` ; `++a` ; `--a`	
Присваивание с вычислением	`a += b` ; `a -= b` ; `a *= b` ; `a /= b` ; `a %= b` ; и т.д.	
лог И	`a && b`	
лог ИЛИ	`a || b`	
лог НЕ	`!a`	
Равно	`a == b`	
Не равно	`a != b`	
Больше	`a > b`	
Больше или равно	`a >= b`	
Меньше	`a < b`	
Меньше или равно	`a <= b`	
Конкатенация	`"aaa" + "bbb"`	
Привидение типа	`(нужныйТип) a`	

Полезные методы Math

- `Math.sqrt(a)` - корень из числа `a`
- `Math.pow(a, b)` - число `a` возводим в степень `b`
- `Math.random()` - случайное число типа `double`

Условный оператор

Вычисляет логическое выражение (boolean) и пускает выполнение по тому или иному пути

Может иметь одну ветку

```
// Не конвенционально. Принято со скобками
if (a > b) System.out.println("a больше b");

if (a > b) {
```

```
  System.out.println("a больше b");
}
```

Может иметь альтернативную ветку

```
if (a > b) {
  System.out.println("a больше b");
} else {
  System.out.println("b больше a");
}
```

Может иметь дополнительные ветки с условием

```
if (a > b) {
  System.out.println("a больше b");
} else if (a == b) { // блоков `else if` может быть много
  System.out.println("a и b равны");
} else {
  System.out.println("b больше a");
}
```

Причем, ветки `else` в этом случае может и не быть

```
if (a > b) {
  System.out.println("a больше b");
} else if (a == b) { // блоков `else if` может быть много
  System.out.println("a и b равны");
}
```

Тернарный оператор

```
String answer = (a > b) ? "a больше b" : "b больше a";
```

Полезные методы System

- `System.out.println(a)` / `System.out.print(a)` - выводим строку `a` в стандартный поток вывода (консоль по умолчанию) с переводом на новую строчку и без
- `System.err.println(a)` / `System.err.print(a)` - выводим строку `a` в стандартный поток вывода ошибок (консоль по умолчанию) с переводом на новую строчку и без

Классы

Шаблон поведения объекта с заданными параметрами, определяющими состояние.
Поля - для хранения состояния
Методы - для определения поведения
Классы неявно наследуются от класса `Object`
Класс определяется, как

```
[пакет, в котором лежит класс]

[импорты других классов]

[модификаторы] ИмяКласса {
  // Поля, конструкторы, методы
}
```

Поля класса

Могут иметь значения по умолчанию.
Их принято писать вначале класса
Внутри класса можно обращаться просто по имени. Если есть вероятность конфликта имен, то нужно дописать вначале `this.`
Поля классов не принято выставлять наружу. Их закрывают и добавляет getter/setter-ы для получения и установки их значений
Поле задается как
`[модификаторы] тип имя;` или
`[модификаторы] тип имя = значение;`

Конструктор класса

Вызывается, когда обращаемся к классу с ключевым словом `new`
Принято писать после полей класса и перед методами
Конструкторы могут вызывать друг друга через ключевое слово `this` и с нужными параметрами в скобках
Если в классе нет конструкторов, то Java создаст пустой конструктор по умолчанию
Конструктор определяется как

```
[модификаторы] ИмяКласса(){
  // тело
}
```

или с параметрами

```
[модификаторы] ИмяКласса(тип имяАргумента, тип имяДругогоАргумента){
  // тело
}
```

Методы класса

Определяют поведение
Могу вызывать друг друга и могут иметь одинаковые имена, если имеют разный набор аргументов
Методы обязаны иметь возвращаемый тип. Если метод ничего не должен возвращать, пишем `void`
Метод определяется, как

```
[модификаторы] возвращаемыйТип имяМетода(){
  // тело метода
}
```

или с параметрами

```
[модификаторы] возвращаемыйТип имяМетода(тип имяАргумента, тип имяДругогоАргумента){
  // тело метода
}
```

Перегрузка методов

В классе может быть несколько методов с одним и тем же именем, но разными параметрами. Это называется перегрузкой метода

Объекты

Конкретный экземпляр (интсанс), созданный по определению класса.

null

Значение по умолчанию у ссылочного типа данных – `null`
Если попытаться вызвать метод или обратить к полю, если объект `null`, получим `NullPointerException` – самую распространенную в мире ошибку.
Пример проверки на `null`

```
public boolean domrMoethod(SomeClass someObject) {
    if (someObject == null) {
        return false;
    }
    ... Дальнейшая логика ...
}
```

Обертки для примитивов

У примитивных классов (int, float, char и т.д.) есть классы-обертки: Integer, Double, Float, Character и т.д.
Для них Java поддерживает автоматическую упаковка и распаковку. То есть, в переменную Integer можно просто присвоить int и Java сама сделает объект. И наоборот.
Но это стоит ресурсов

Модификаторы доступа

Модификатор	Класс	Тот же пакет	Дочерний класс	Все
`public`	Да	Да	Да	Да
`protected`	Да	Да	Да	Нет
package-private	Да	Да	Нет	Нет
`private`	Да	Нет	Нет	Нет

Модификатор final

Далет поле класса (и переменную) неизменяемой.
Значение поля `final` обязано должно быть задано в конструкторе или при определении

Константы

В Java нет отдельного слова для определения константы, но ее можно определить, задав ключевые слова `static` и `final`
При этом, имя должно быть заглавными буквами, разделенными нижным подчеркиванием

```
private static final double THRESHOLD_VALUE = 75.0;
```

Некоторые методы класса Object

toString

Вызывается при преобразовании объекта в строку. При передачи в `System.out.println`, к примеру
По умолчанию, выводит строку вида `имяКласса@хэшКод`, где хэш-код берется от адреса в памяти (head)

equals

Используется для сравнения объектов между собой
Сравнение через `==` для объектов будет сравнивать их адреса в памяти
При переопределении `equals`, должен быть переопределен и `hashCode`

hashCode

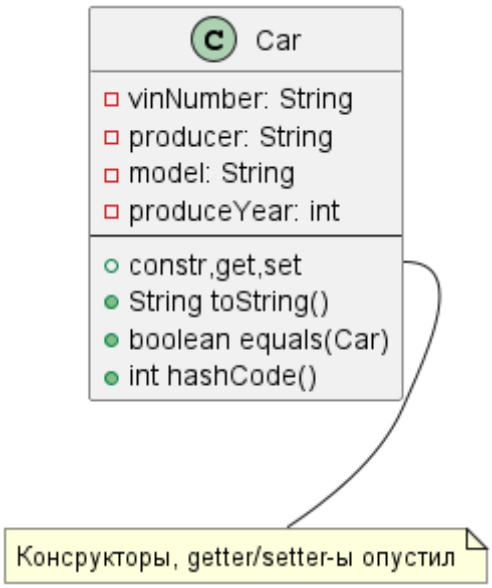
Вычисляет числовой код объекта
Используется, к примеру, при определении бакета в `HashMap`

Пример класса

```
// Определение пакета
package lesson_2.car_example.ex_6;

// Определяем импорты вначале файла
import java.util.Objects;

// класс имеет доступ package-private
class Car {
  // вначале принято определять поля класса
  // final переменная инициализируется в конструкторе, но уже не может изменяться
  private final String vinNumber;
  private String producer;
  private String model;
  private int produceYear;

  // Конструктор, который вызывает другой конструктор с меньшим числом параметров
  public Car(String vinNumber, String producer, String model, int produceYear) {
    this(vinNumber, producer, model);
    this.produceYear = produceYear;
  }

  public Car(String vinNumber, String producer, String model) {
```

```
  this.vinNumber = vinNumber;
  this.producer = producer;
  this.model = model;
}

public String getVinNumber() {
  return vinNumber;
}

public String getProducer() {
  return producer;
}

public void setProducer(String producer) {
  this.producer = producer;
}

// ...
// И прочие getter/setter-ы
// ...

/**
 * Переопределяем, чтобы иметь понятный вывод в консоле
 *
 * @return Строковое представление объекта
 */
@Override
public String toString() {
  return String.format("Автомобиль (vin номер %s, производства %s, модель %s, %s года выпуска
      vinNumber, producer, model, produceYear);
}

/**
 * Определяем для сравнения по значениям полей
 *
 * @param o объект, с которым будем сравнивать
 * @return равны объекты или не равны
 */
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Car car = (Car) o;
  return produceYear == car.produceYear && Objects.equals(vinNumber, car.vinNumber) && Object
}

/**
 * По контракту обязаны переопределить с тем же набором полей, которые учавствуют в сравнении
 *
 * @return Числовое представление объекта
 */
@Override
public int hashCode() {
```

```
        return Objects.hash(vinNumber, producer, model, produceYear);
    }
}
```



Наследование

Чтобы наследовать один класс от другого, нужно использовать ключевое слово `extends` сразу после имени класса в определении

Переопределение

Чтобы переопределить метод родительского класса, нужно сделать метод с той же сигнатурой в дочернем классе
Переопределенные методы нужно помечать аннотацией `@Override`

Абстрактные классы

Чтобы сделать класс абстрактным, нужно в его определении использовать ключевое слово `abstract`
Нельзя создать экземпляр абстрактного класса
Абстрактные классы могут иметь абстрактные методы

Абстрактный метод

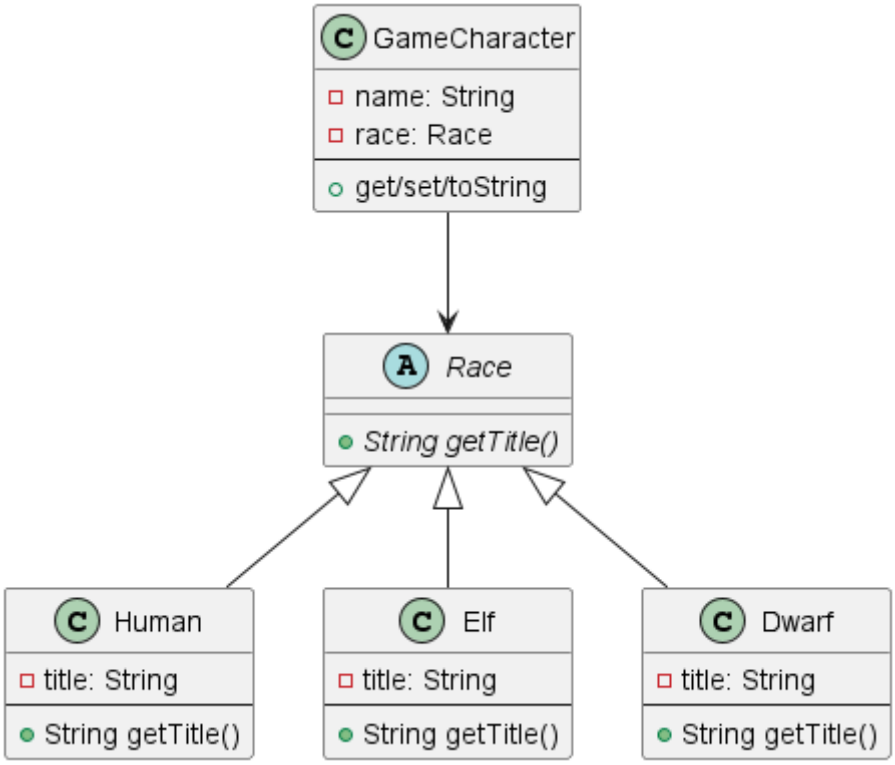
В абстрактном классе может быть абстрактный метод, у которого определена сигнатура, но не определено тело.
Этот метод должен быть реализован в потомке, если потомок тоже не является абстрактным

Абстрактный класс и конструктор с параметрами

Если в классе, от которого мы наследуемся есть только конструктор с параметрами, мы обязаны его вызвать в своем конструкторе первой строкой используя ключевое слово `super`

```
public abstract class Measurement {
  ...
  public Measurement(Random randomGenerator) {
        this.randomGenerator = randomGenerator;
    }
}



public class CpuMeasurement extends Measurement {
  ...
  public CpuMeasurement() {
    super(new Random());
  }
}
```

Пример наследования



Абстрактный класс `Race` с абстрактным методом `getTitle()`

```
public abstract class Race {
  public abstract String getTitle();
}
```

Наследник класса `Race` класс `Elf` с реализацией метода `getTitle()`

```
public class Elf extends Race {
  private final String title = "Эльф";

  @Override
  public String getTitle() {
    return title;
  }
}
```

Класс, ожидающий на вход `Race`

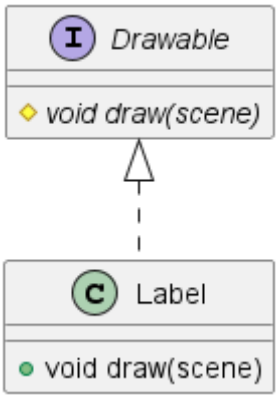
```
public class GameCharacter {
    private final Race race;
    private String name;

    public GameCharacter(Race race, String name) {
```

```
        this.race = race;
        this.name = name;
    }

    // getter/setter/toString
}
```

Использование одного из наследников `Race` вместо `Race`

```
public class Game {
    public static void main(String[] args) {
        GameCharacter character = new GameCharacter(new Elf(), "Глорфиндеил");
        System.out.println(character);
    }
}
```

Интерфейс



Определяется ключевым словом `interface` вместо `class` и может содержать методы без реализации, при этом методы интерфейса по умолчанию уже являются публичными, а поля - константами
Нельзя сделать экземпляр интерфейса. Чтобы реализовать интерфейс, в классе используется слово `implements` .
Если класс наследуется и реализует одновременно, то вначале идет `extends` , и класс может реализовывать много интерфейсов
Класс обязан реализовать методы интерфейса

Проверка на наличие интерфейса у объекта

Оператор `isntaceOf` позволяет определить, реализует ли класс объекта нужный интерфейс . В `instanceOf` неявно проводится проверка на `null`

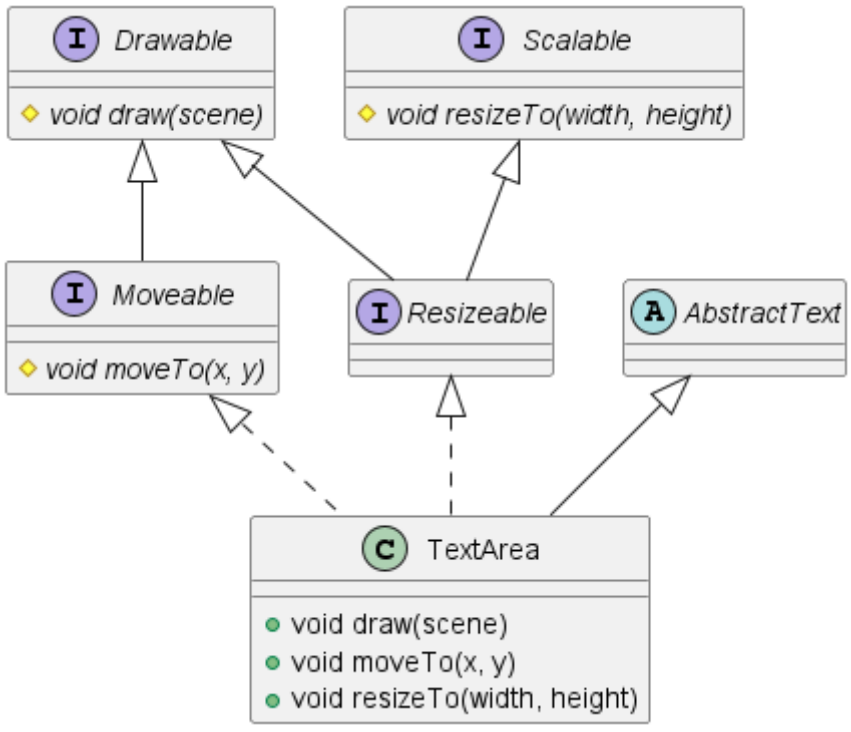
Привидение типа к интерфейсу

```
    private void addToListeners(Drawable drawable) {
    if (drawable instanceof Moveable) {
      ((Moveable) drawable).moveTo(1, 2);
    }
    ...
  }
```

Наследование интерфейса от других интерфейсов

Интерфейсы могут наследовать от других интерфейсов, при этом, у интерфейса может быть несколько предков
Класс, который реализует такой интерфейс обязан реализовывать все методы, определенные во всей цепочке интерфейсов

Пример с интерфейсами



Главный родительский интерфейс

```
public interface Drawable {
  void draw(Object scene);
}
```

Еще один родительский интерфейс с константами

```
public interface Scalable {
  Integer MIN_HEIGHT = 10;
  Integer MIN_WIDTH = 10;
  Integer MAX_HEIGHT = 600;
  Integer MAX_WIDTH = 800;
  void resizeTo(Integer width, Integer height);
}
```

Интерфейс, наследующий от двух родительских

```
public interface Resizeable extends Drawable, Scalable {
}
```

Класс, реализующий несколько интерфейсов и наследующий от абстрактного класса
Тут же использование констант
Класс реализует все методы, определенные во всех интерфейсах по цепочке наследования

```
public class TextArea extends AbstractText implements Moveable, Resizeable {
  ...
  @Override
  public void draw(Object scene) { ... }

  @Override
  public void moveTo(Integer x, Integer y) { ... }

  @Override
  public void resizeTo(Integer width, Integer height) {
    if (width  < Scalable.MIN_WIDTH || width > Scalable.MAX_WIDTH
        || height < Scalable.MIN_HEIGHT || height > Scalable.MAX_HEIGHT) {
      System.out.println("Некорректный размер");
      return;
    }
    System.out.println("Рахмер изменился и теперь " + width + " " + height);
  }
  ...
}
```