Оглавление

[Методы 2](#_Toc120201928)

[Параметры 2](#_Toc120201929)

[Возвращение значения из метода 3](#_Toc120201930)

[Выход из метода 4](#_Toc120201931)

[Условия – стражники 4](#_Toc120201932)

[Рекурсия 6](#_Toc120201933)

[Рекурсия с кэшем 6](#_Toc120201934)

[Особенности работы с изменяемыми типами данных 7](#_Toc120201935)

[Классы и объекты 7](#_Toc120201936)

[Конструкторы 8](#_Toc120201937)

[Ключевое слово this 8](#_Toc120201938)

[Инициализаторы 9](#_Toc120201939)

[Ключевое слово final 10](#_Toc120201940)

[Ключевое слово instanceof 11](#_Toc120201941)

[Ключевое слово static 11](#_Toc120201942)

[Абстракция 12](#_Toc120201943)

[Модификаторы доступа 12](#_Toc120201944)

[Инкапсуляция 14](#_Toc120201945)

[Наследование 16](#_Toc120201946)

[Переопределение методов 17](#_Toc120201947)

[Слово @Override 17](#_Toc120201948)

[Динамический поиск метода 17](#_Toc120201949)

[Полиморфизм 18](#_Toc120201950)

[Интерфейс 19](#_Toc120201951)

[Связывание в java 20](#_Toc120201952)

[Различия инициализатора и конструктора. Выгода использования 22](#_Toc120201953)

[Области видимости переменных 22](#_Toc120201954)

[Определения: 24](#_Toc120201955)

**Защита по программированию**

# Методы

Переменные и константы хранят некоторые значения. Методы (функции) содержат набор операндов, которые выполняют определенные действия.

Общее определение методов:

*[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры]) {*

*// тело метода*

*}*

Пример:

*public static void main(String[] args) {*

*System.out.println(“Hello world”)*

*}*

Ключевое слово void означает, что метод ничего не возвращает.

Одно из преимуществ метода: мы можем вынести некоторые общие действия в отдельный метод и затем вызывать многократно их в различных местах программы.

* Методы прописываются внутри класса, а не внутри метода main. Нельзя прописывать методы внутри методов!
* Чтобы вызвать в методе main другие методы, которые определены в одном классе с методом main, они должны иметь модификатор static

# Параметры

С помощью параметров мы можем передать в методы различные данные, которые будут использоваться для вычислений.

Пример:

*public class Program{*

*public static void main (String args[]){*

*display("Tom", 34);*

*display("Bob", 28);*

*display("Sam", 23);*

*}*

*static void display(String name, int age){*

*System.out.println(name);*

*System.out.println(age);*

*}*

*}*

# Возвращение значения из метода

Для того, чтобы метод возвращал некоторое значение, должна применяться операция *return*. После оператора указывается возвращаемое значение, которое является результатом метода.

Пример:

*public class Main {*

*public static void main (String args[]){*

*int x = sum(1, 2, 3);*

*int y = sum(1, 4, 9);*

*System.out.println(x); // 6*

*System.out.println(y); // 14*

*}*

*static int sum(int a, int b, int c){*

*return a + b + c;*

*}*

*}*

Возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции.

Если в качестве возвращаемого типа для метода определен любой другой, отличный от *void*, то метод обязательно должен использовать оператор *return* для возвращения значения

Метод может использовать несколько вызовов оператора *return* для возвращения разных значений в зависимости от условий

# Выход из метода

Оператор *return* применяется не только для возвращения значения из метода, но и для *выхода из метода*

Сравнение break и return:

* Если вы используете вложенный цикл, то break сможет остановить только один цикл, но не оба сразу. А return сможет. Функция завершится после выполнения оператора return на какой бы степени вложенности он не был.

Пример:

*static boolean find\_word(String[] lines, String prototype){*

*prototype = prototype.lower();*

*for (String line: lines){*

*String[] words\_in\_lines = line.split(" ");*

*for(String word: words\_in\_lines){*

*if (word.lower().equals(prototype))*

*return true;*

*}*

*}*

*return false;*

*}*

# Условия – стражники

Это условия, которые прописываются в начале функции и, если они не выполняются, то функция сразу же завершается с помощью *return*

Пример кода без условий-стражников:

*static void findRadius(int a, int b, int c){*

*if (a > c) {int d = a; a = c; c = d;}*

*if (b > c) {int d = b; b = c; c = d;}*

*//теперь в пересенной c лежит наибольшее значение из 3*

*//проверим, что такой треугольник существует*

*if (c < a + b){*

*//проверим, что это прямоугольный треугольник*

*if (c \* c == a \* a + b \* b){*

*//вот тут всё в порядке, находим радиус по формуле*

*double r = (a + b - c) / 2.0;*

*System.out.println("Радиус вписанной окружности равен: " + r);*

*}else{*

*System.out.println("Этот треугольник не прямоугольный");*

*}*

*}else{*

*System.out.println("Треугольника с такими сторонами не существует");*

*}*

*}*

Теперь перепишем этот код, но уже используя условия-стражники:

*static void findRadius(int a, int b, int c){*

*//для удобства сделаем так, чтобы в переменной c лежала длина самой длинной стороны.*

*if (a > c) {int d = a; a = c; c = d;}*

*if (b > c) {int d = b; b = c; c = d;}*

*//теперь в переменной c лежит наибольшее значение из 3*

*if (c >= a + b){ //первый стражник: проверяет, что такой треугольник существует*

*System.out.println("Треугольника с такими сторонами не существует");*

*return;*

*}*

*if (c \* c != a \* a + b \* b){ //второй стражник: проверяет, что треугольник прямоугольный*

*System.out.println("Этот треугольник не прямоугольный");*

*return;*

*}*

*//вот тут всё в порядке, находим радиус по формуле*

*double r = (a + b - c) / 2.0;*

*System.out.println("Радиус вписанной окружности равен: " + r);*

*}*

Полученный код выглядит более простым, структурированным и не «уходит» от начала строки. Наличие таких стражников говорит о хорошей масштабируемости кода (способности системы, сети или процесса справляться с увеличением рабочей нагрузки).

# Рекурсия

Рекурсия — это метод в программировании, использующий вызов какой-либо функции внутри самой этой функции (или её вызов через несколько других функций).

Пример:

*public class Main{*

*public static void main(String[] args) {*

*System.out.println(fib(10));*

*}*

*static int fib(int n){*

*if (n == 1 || n == 2)*

*return 1;*

*return fib(n-1) + fib(n-2);*

*}*

*}*

# Рекурсия с кэшем

Для ускорения работы рекурсии, можно заставить ее запоминать значения.

Пример улучшения предыдущей программы:

*import java.util.HashMap;*

*class Main{*

*static HashMap<Integer, Integer> cash;*

*public static void main(String[] args) {*

*cash = new HashMap<>();*

*cash.put(1, 1);*

*cash.put(2, 1);*

*System.out.println(fib(41));*

*}*

*static int fib(int n){*

*System.out. println("Запускается вычисление " + n + " числа Фиббоначи");*

*if (!cash.containsKey(n))*

*cash.put(n, fib(n-1) + fib(n-2));*

*return cash.get(n);*

*}*

*}*

# Особенности работы с изменяемыми типами данных

Иногда функция ничего не возвращает, но может изменить значение ссылочного типа, однако такое возможно только с изменяемыми типами данных: список, словарь, множества. С неизменяемыми типами данных: числа, строки, кортеж, булевые, None – такого не получится.

# Классы и объекты

Класс – описание характеристик \ описание объекта. Классы могут наследовать свойства от других классов. Родительский класс называется суперклассом

Объект – конкретный экземпляр класса. Обладает двумя основными состояниями: данные, которые хранит объект (поля, переменные класса) и действия, которые может совершать объект (методы)

Создание объекта:

*Person tom = new Person()*

Person tom — это объявление переменной, которое информирует компилятор о том, что tom будет ссылаться на объект типа person (переменная еще не определяет объект, а просто дает возможность ссылаться на него). Оператор new динамически (во время работы программы) выделяет память для объекта Person, а Person() инициализирует объект и присваивает значение ссылке (приводит программу или устройство в состояние готовности к использованию).

И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

# Конструкторы

Инициализация переменных объекта перед его использованием.

Вызываются при создании нового объекта данного класса и выполняют инициализацию объекта. Если в классе не определен ни один конструктор, то автоматически создается конструктор без параметров, но дефолтный конструктор исчезает из класса, когда ты создаешь какой-то конструктор с аргументами.

Конструктор нельзя наследовать, но ему можно передать значения с помощью аргумента super

# Ключевое слово this

Представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы.

Пример:

*public class Main {*

*public static void main(String[] args) {*

*Example example = new Example("Example", 1);*

*System.out.print(example.name);*

*System.out.print(example.age);*

*}*

*}*

*class Example {*

*public String name;*

*public int age;*

*public Example(String name, int age) {*

*this.name = name;*

*this.age = age;*

*}*

*}*

# Инициализаторы

(Инициализация - это когда мы впервые задаем переменной какое-либо значение)

Инициализатор выполняется до любого конструктора.

Блоки инициализации - код, заключенный в фигурные скобки и размещаемый внутри класса вне объявления методов или конструкторов.

Виды блоков инициализации:

1. Статические (статический блок используется для инициализации статических переменных, а "обычный» - для всех остальных.)
2. Нестатические

Блоки инициализации экземпляра полезны, если вы хотите, чтобы некоторый код выполнялся независимо от того, какой конструктор используется.

Пример:

*public class Test {*

*static int staticVariable;*

*int nonStaticVariable;*

*// Статический блок инициализации:*

*// Запускается один раз (когда инициализирован класс)*

*static {*

*System.out.println("Static initalization.");*

*staticVariable = 5;*

*}*

*// Нестатический блок инициализации:*

*// Запускается каждый раз, когда инициализируется объект*

*{*

*System.out.println("Instance initialization.");*

*nonStaticVariable = 7;*

*}*

*public Test() {*

*System.out.println("Constructor.");*

*}*

*public static void main(String[] args) {*

*new Test();*

*new Test();*

*}*

*}*

# Ключевое слово final

Позволяет предотвратить изменение содержимого переменной, по сути, это становится константой. Финальное поле должно быть инициализировано во время его первого объявления.

Применять final можно к:

1. Параметрам метода (препятствует изменению в пределах метода)
2. Локальным переменным (препятствует присвоению значения больше одного раза)
3. Методам (предотвращает переопределение метода)

Когда класс объявлен как final, не может быть подклассом, т.е. никакой другой класс не может его расширить. (можно запретить наследование с помощью ключевого слова final)

Во время наследования мы должны объявлять методы с ключевым словом final, для которых мы должны следовать одной и той же реализации во всех производных классах.

# Ключевое слово instanceof

Иногда требуется проверить, к какому классу принадлежит объект. Это можно сделать при помощи ключевого слова instanceof. Это булев оператор, и выражение foo instanceof Foo истинно, если объект foo принадлежит классу Foo или его наследнику, или реализует интерфейс Foo (или, в общем виде, наследует класс, который реализует интерфейс, который наследует Foo).

Пример:

*SaltwaterFish nemo = new SaltwaterFish();*

*if(nemo instanceof Fish) {*

*// рыбка Немо относится к классу Fish*

*// это может быть класс Fish (родительский класс) или подкласс типа*

*// SaltwaterFish или FreshwaterFish.*

*if(nemo instanceof SaltwaterFish) {*

*// Немо - это морская рыбка!*

*}*

*}*

# Ключевое слово static

Static — модификатор, применяемый к полю, блоку, методу или внутреннему классу. Данный модификатор указывает на привязку субъекта к текущему классу.

1. Статические переменные (если переменная статическая, то это глобальное значение — одно для всех). К статическим переменным обычно обращаются не по ссылке на объект, а по имени класса
2. Статические блоки инициализации (если блок инициализации статический, то и все переменные внутри данного блока тоже статические)
3. Статические методы (методы привязаны к классу, а не к объекту). Статический метод может обратиться только к статическим переменным/методам. Переопределять статический метод нельзя!
4. Статический класс (может быть только внутренний класс; привязан к внешнему классу: если внешний класс будет наследован, то внутренний класс не будет). Внутренний класс получает доступ к приватным статическим переменным внешнего класса. Класс является набором только статических сущностей (полей, методов, свойств и т.п.), создавать объекты такого класса нельзя.

# Абстракция

Выделение главных, наиболее значимых характеристик предмета, и наоборот — отбрасывание второстепенных, незначительных. Просто чертеж для будущих «нормальных» классов

Экземпляр (объект) абстрактного класса создать нельзя

Базовый класс, который не предполагает создания экземпляров

# Модификаторы доступа

Все члены класса (поля, методы) имеют модификаторы доступа.

Модификаторы задают допустимую область видимости для членов класса, контекст, в котором метод или переменную можно употреблять.

1. Public. Публичный, общедоступный класс или член класса, поля и методы видны другим классам
2. Private. Закрытый класс или член класса, противоположность модификатору public. Класс или член класса доступен только из кода в том же классе.
3. Protected. Поля и методы видны в пределах всех классов, находящихся в том же пакете, что и наш, а также в пределах всех классов наследников. (то же самое, что default, но доступ имеют еще классы-наследники)
4. Default (package visible). Поля или методы видны всем классам в текущем пакете. (классы-наследники не имеют доступ)

Пример использования модификаторов доступа:

*public class Program{*

*public static void main(String[] args) {*

*Person kate = new Person("Kate", 32, "Baker Street", "+12334567");*

*kate.displayName(); // норм, метод public*

*kate.displayAge(); // норм, метод имеет модификатор по умолчанию*

*kate.displayPhone(); // норм, метод protected*

*//kate.displayAddress(); // ! Ошибка, метод private*

*System.out.println(kate.name); // норм, модификатор по умолчанию*

*System.out.println(kate.address); // норм, модификатор public*

*System.out.println(kate.age); // норм, модификатор protected*

*//System.out.println(kate.phone); // ! Ошибка, модификатор private*

*}*

*}*

*class Person{*

*String name;*

*protected int age;*

*public String address;*

*private String phone;*

*public Person(String name, int age, String address, String phone){*

*this.name = name;*

*this.age = age;*

*this.address = address;*

*this.phone = phone;*

*}*

*public void displayName(){*

*System.out.printf("Name: %s \n", name);*

*}*

*void displayAge(){*

*System.out.printf("Age: %d \n", age);*

*}*

*private void displayAddress(){*

*System.out.printf("Address: %s \n", address);*

*}*

*protected void displayPhone(){*

*System.out.printf("Phone: %s \n", phone);*

*}}*

# Пакет — это способ объединить группу классов, интерфейсов и подпакетов. С помощью пакетов создаются группы связанных классов, интерфейсов, перечислений и так далее

# Инкапсуляция

Инкапсуляция – объединение данных и методов для работы с этими данными в одной упаковке («капсуле») В Java в роли капсулы выступает класс.

Сокрытие – разграничение доступа различных частей программы ко внутренним компонентам друг друга.

Пример сокрытия:

*public class Auto {*

*public void gas() {*

*/\*внутри автомобиля происходят какие-то сложные вещи*

*в результате которых он едет вперед\*/*

*}*

*public void brake() {*

*/\*внутри автомобиля происходят какие-то сложные вещи*

*в результате которых он тормозит\*/*

*}*

*public static void main(String[] args) {*

*Auto auto = new Auto();*

*//Как все выглядит для пользователя*

*//нажал одну педаль - поехал*

*auto.gas();*

*//нажал другую педаль - затормозил*

*auto.brake();*

*}*

*}*

\*Пользователю предоставлен интерфейс (методы). Если ему нужно, чтобы автомобиль в программе выполнил действие, достаточно вызвать нужный метод. А уж что там происходит внутри этих методов — информация лишняя, главное, чтобы все работало как надо\*

Преимущества использования инкапсуляции:

* Контроль за корректным состоянием объекта
* Удобство для пользователя за счёт интерфейса. Для доступа пользователя мы оставляем только методы, ему достаточно вызвать их, чтобы получить результаты, и не обязательно вникать в суть работы кода.
* Изменения в коде не отображаются для пользователя.

Для достижения инкапсуляции в Java:

* Объявите переменные класса как private.
* Предоставьте public к методам установки и получения (сеттеру и геттеру) для изменения и просмотра значений переменных.

# Наследование

Наследование - расширение функционала уже имеющихся классов за счет добавления нового функционала или изменения старого. При наследовании класс наследует все те же поля и методы, которые есть в родительском, кроме тех, которые имеют модификатор private.

Пример наследования членов суперкласса:

*class A {*

*int i;*

*}*

*// наследуемся от класса A*

*class B extends A {*

*int i; // имя переменной совпадает и скрывает переменную i в классе A*

*B(int a, int b) {*

*super.i = a; // обращаемся к переменной i из класса A*

*i = b; // обращаемся к переменной i из класса B*

*}*

*void show() {*

*System.out.println("i из суперкласса: " + super.i);*

*System.out.println("i в подклассе: " + i);*

*}*

*}*

*class MainActivity {*

*B subClass = new B(1, 2);*

*subClass.show();*

*}*

Правила наследования:

* Наследуем только один класс
* Наследуется все кроме приватных переменных и методов

# Переопределение методов

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса. Перед переопределяемым методом указывается аннотация @Override (проверяет переопределен ли метод, вызывает ошибку если метод не найден в родительском классе).

# Слово @Override

Аннотация @Override указывает, что далее мы собираемся переопределять метод базового класса. При этом, если в базовом классе не окажется метода с аналогичной сигнатурой, то мы получим предупреждение компилятора о том, что хотя мы и собирались что-то переопределить, по факту этого не произошло.

# Динамический поиск метода

Пример:

*public class Program{*

*public static void main(String[] args) {*

*Person tom = new Person("Tom");*

*tom.display();*

*Person sam = new Employee("Sam", "Oracle");*

*sam.display();*

*}*

*}*

*class Person {*

*String name;*

*public String getName() { return name; }*

*public Person(String name){*

*this.name=name;*

*}*

*public void display(){*

*System.out.printf("Person %s \n", name);*

*}*

*}*

*class Employee extends Person{*

*String company;*

*public Employee(String name, String company) {*

*super(name);*

*this.company = company;*

*}*

*@Override*

*public void display(){*

*System.out.printf("Employee %s works in %s \n", super.getName(), company);*

*}*

*}*

При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе. Данный процесс еще называется dynamic method lookup или динамический поиск метода или динамическая диспетчеризация методов.

# Полиморфизм

Полиморфизм — это возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе («перегрузка метода)» или в группе классов («переопределение метода»), связанных отношением наследования.

Перегрузка метода – создание нескольких методов с одним названием, но с разными параметрами.

Пример полиморфизма (перегрузка):

*public class Main {*

*public static void main(String[] args) {*

*Animal dog1 = new Animal();*

*dog1.voice("hello"); //hello*

*dog1.voice(5) //5*

*}*

*}*

*class Animal {*

*void voice (String string) {*

*System.out.print(string);*

*}*

*void voice(int ing) {*

*System.out.print(ing);*

*}*

*}*

Класс считается полиморфным, если имеет множественное наследование:

*public interface Vegetarian{}*

*public class Animal{}*

*public class Deer extends Animal implements Vegetarian{}*

# Интерфейс

Абстрактный тип, который используется для указания поведения, которое должны реализовывать классы.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Если класс применяет интерфейс, то он должен реализовать все методы интерфейса. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

Пример использования интерфейса и implements:

*public class Program{*

*public static void main(String[] args) {*

*Book b1 = new Book("Java. Complete Referense.", "H. Shildt");*

*b1.print();*

*}*

*}*

*interface Printable{*

*void print();*

*}*

*class Book implements Printable{*

*String name;*

*String author;*

*Book(String name, String author){*

*this.name = name;*

*this.author = author;*

*}*

*public void print() {*

*System.out.printf("%s (%s) \n", name, author);*

*}*

*}*

# Связывание в java

Наличие связи между ссылкой и кодом. Переменная, на которую вы ссылаетесь, привязана к коду, в котором она определена. Аналогично, вызываемый метод привязан к месту в коде, где он определен.

Виды связывания:

* Раннее связывание (статическое). Происходит во время компиляции, то есть код «знает», какой метод вызывать после компиляции исходного кода на Java в файлы классов. Относится к ранней стадии жизненного цикла программы.
* Позднее связывание (динамическое). Происходит во время выполнения, после запуска программы виртуальной машиной Java. В этом случае то, какой метод вызвать, определяется конкретным объектом, так что в момент компиляции информация недоступна, ведь объекты создаются во время выполнения. Относится к поздней стадии жизненного цикла программы.

Фундаментально различие: статическое связывание происходит во время компиляции на основе типа ссылочной переменной, а динамическое во время выполнения, с использованием конкретных объектов.

Пример: 3

*public class Main {*

*public static void main(String[] args) {*

*// Пример статического и динамического связывания в Java*

*Insurance current = new CarInsurance();*

*// Динамическое связывание на основе объекта*

*int premium = current.premium();*

*// Статическое связывание на основе класса*

*String category = current.category();*

*System.out.println("premium : " + premium);*

*System.out.println("category : " + category);*

*}*

*}*

*class Insurance{*

*public static final int LOW = 100;*

*public int premium(){*

*return LOW;*

*public static String category(){*

*return "Insurance";*

*}*

*}*

*class CarInsurance extends Insurance{*

*public static final int HIGH = 200;*

*public int premium(){*

*return HIGH;*

*}*

*public static String category(){*

*return "Car Insurance";*

*}*

*}*

# Различия инициализатора и конструктора. Выгода использования

В блоке инициализации указываем конкретные значения для присвоения значений элементам, а в интерфейсе мы указываем какие значения нужно будет применить исходя из того, что написано при создании объекта.

В конструктор мы можем передать параметры, а в блок инициализации нет.

# Области видимости переменных

Свойство, которое определяет, можно ли использовать переменную из разных областей класса. Места в коде, где к этой переменной можно обратиться

1. Переменная, объявленная в методе, существует/видна с начала объявления до конца метода.

2. Переменная, объявленная в блоке кода, существует до конца этого блока кода.

3. Переменные — аргументы метода — существуют везде внутри метода.

4. Переменные класса/объекта существуют все время жизни содержащего их объекта. Их видимость дополнительно регулируется специальными модификаторами доступа: public, private.

5. Статические переменные классов существуют все время работы программы. Их видимость также определяется модификаторами доступа.

# Класс enum

Набор именованных констант, который помогает в определении своих собственных типов данных

# Код, который могут попросить объяснить на защите:

Первый вариант:

*class A {*

*{*

*System.out.println("A1");*

*}*

*A() {*

*System.out.println("A2");*

*}*

*}*

*class B extends A {*

*{*

*System.out.println("B1");*

*}*

*B() {*

*System.out.println("B2");*

*}*

*}*

*public class Main {*

*public static void main(String[] args) {*

*B b = new B();*

*}*

*}*

Второй вариант:

*class A {*

*int test1() {System.out.println("A1"); return 1;}*

*static int test2() {System.out.println("A2"); return 2;}*

*}*

*class B extends A {*

*int test1() {System.out.println("B1"); return 1;}*

*static int test2() {System.out.println("B2"); return 2;}*

*}*

*public class Main {*

*public static void main(String[] args) {*

*A b = new B();*

*((A)b).test1();*

*((B)b).test2();*

*}*

*}*

Что спрашивают: «Что выведется в результате работы программы и почему?»

# Определения:

ООП (Объектно-Ориентированное Программирование) – совокупность объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Наследование – расширение функционала уже имеющихся классов за счет добавления нового функционала или изменения старого.

Полиморфизм – применение одноименных методов с одинаковыми или различными наборами параметров в одном классе («перегрузка метода)» или в группе классов («переопределение метода»), связанных отношением наследования.

Абстракция – набор значимых характеристик объекта, доступный остальной программе.

Инкапсуляция – объединение данных и методов для работы с этими данными в одной упаковке («капсуле») В Java в роли капсулы выступает класс.

This – ссылка на объект текущего класса.

Super – обозначает суперкласс, т.е. класс, производным от которого является текущий класс.

Final – определение сущности, которая может быть назначена только один раз.

Static – указывает на привязку субъекта к текущему классу.

Void – метод не возвращает никаких данных.

@Override – указывает, что мы собираемся переопределять метод базового класса.

Инициализация – присваивания начального значения, выделение памяти под объект (при инициализации объекта).

Статическое связывание – происходит во время компиляции на основе типа ссылочной переменной.

Динамическое связывание – происходит во время выполнения после запуска JVM, с использованием конкретных объектов

Переопределение метода – реализация метода так, чтобы он имел идентичную сигнатуру с методом класса-предка, но предоставлял иное поведение.

Перегрузка метода – использование одинакового имени метода с разными параметрами

Сигнатура метода = имя метода + параметры

Сокрытие – разграничение доступа различных частей программы ко внутренним компонентам