Содержание:

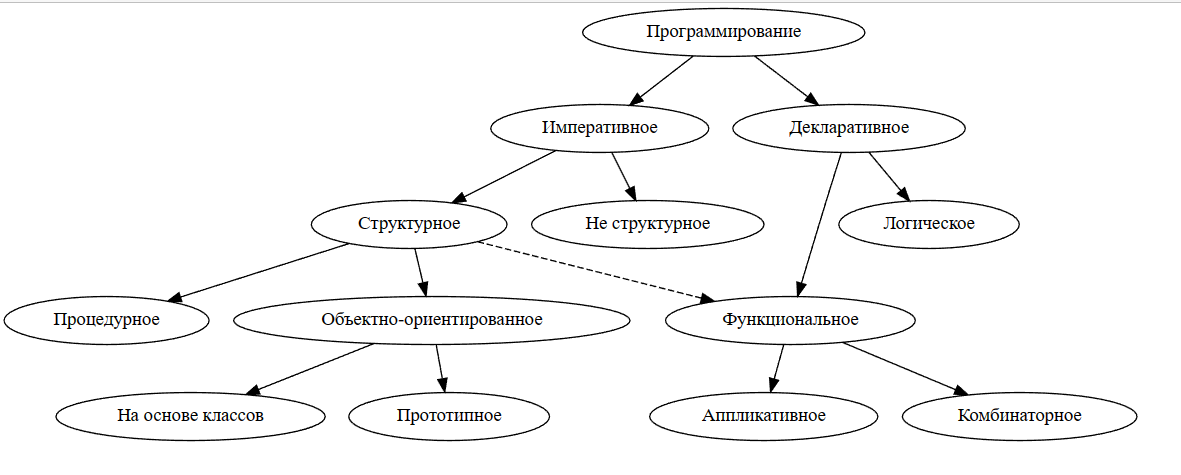
1. Парадигмы программирования.
2. ООП:

Инкапсуляция (модификаторы доступа).

Полиморфизм (Виды, upcasting/downcasting, разница instanceOf и getCalss, виртуальность полей и методов, overriding, overloading, сокрытие static методов).

Наследование (Отношения между классами).

Абстракция (Абстрактный класс и интерфейс, множественное наследование).

**Языковые парадигмы**

*Паради́гма программи́рования* — это совокупность идей и понятий, определяющих стиль написания компьютерных программ, подход к программированию.

Декларативное программирование: программирование в такой парадигме описывает ожидаемый результат, т.е. что нужно сделать, без описания того, как это нужно сделать (этот процесс автоматизирован, т.е. пути реализации запроса определяет система). В качестве примеров декларативных языков обычно приводят HTML и SQL.

Функциональное программирование:

Данные отделены от функций и являются immutable.

Функции рассматриваются как первоклассные объекты (могут быть переданы в другую функцию как переменная).

Принцип чистоты функций – Функция должна возвращать одинаковый результат, если вызывается с тем же набором аргументов – то есть результат не должен зависеть от внешнего состояния программы. Нет побочных эффектов – функция не изменяет состояния программы при ее вызове.

Рекурсия вместо циклов.

Императивное программирование: представляет собой последовательность инструкций, которая определяет, что нужно сделать и как.

* в исходном коде программы записываются инструкции (команды);
* инструкции должны выполняться последовательно;
* данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;
* данные, полученные при выполнении инструкции, могут записываться в память (Изменять свои состояния в ходе выполнения)

*Структурное программирование* – использует блоки кода, ветвления (if), циклы и код исполняется построчно.

*Процедурное программирование* – Нет классов, код пишется как сплошной блок кода, области видимости достигаются только за счет написания процедур – функций, все переменные и функции, объявленные не в других функциях, являются глобальными.

Из-за такого стиля кода, где нет никакого разделения есть проблема с масштабируемостью – чем больше проект, тем сильнее разрастается код и сложнее понять состояние программы.

**Объектно-ориентированное программирование – ООП** – организация программы в виде объектов, которые взаимодействуют между собой.

**Концепции ООП:**

**Инкапсуляция** – Под инкапсуляцией подразумевается 2 явления:

1) Объединение данных и инструкций в одну сущность (блок кода, метод, конструктор, класс), осуществляется за счет блоков кода – { }, обеспечивающие невозможность использования своей реализации и доступа к локальным переменным извне (т.е. невозможно воспользоваться переменной метода в другом методе или обратиться к части кода), что в свою очередь обеспечивает область видимости переменных - scope. Для всех сущностей, кроме класса, область видимости строгая, т.е. ограничена строго собственной сущностью.

Степень изоляции для класса более гибкая, нежели у остальных сущностей, что предоставляет возможность выбора – ограничить реализацию класса или нет.

2) Сокрытие реализации на уровне класса – сокрытие доступа к компонентам класса извне, что осуществляется за счет модификаторов доступа.

|  |  |
| --- | --- |
| public | Доступ из любого места программы |
| protected | Доступ в пакете или наследникам |
| default – package-private | Доступ в том же пакете |
| private | Поля и методы доступны только внутри объявленного класса |

И, при необходимости, предоставление интерфейсов для взаимодействия с данными этого класса.

*Что дает инкапсуляция:*

Контроль за корректным состоянием объекта – сокрытие доступа к полям объекта и предоставление интерфейсов – геттер, сеттер (предоставляют возможность заключить в них некоторую логику – проверить корректность устанавливаемых данных в коде сеттера).

Удобство для пользователя за счет интерфейса. Мы оставляем «снаружи» для доступа пользователя только методы. Ему достаточно вызвать их, чтобы получить результат, и совсем не нужно вникать в детали их работы.

Изменения в коде не отражаются на пользователях. Все изменения мы проводим внутри методов. На пользователя это не повлияет.

**Полиморфизм** – Использование объектов с одинаковым интерфейсом без информации о его внутреннем устройстве.

*Виды полиморфизма:*

*Полиморфизм подтипов* – возможность алгоритма работать со всеми подтипами одного типа, опираясь на свойства общего типа, достигается за счет наследования и реализации интерфейса (каждый предок имеет методы родителей и каждый кто имплементирует интерфейс должен иметь его методы). Позволяет определять, как одинаковое поведение для разных объектов, так и разное – за счет overriding методов в дочерних классах. Это возможно за счет наличия виртуальных методов и динамической диспетчеризации – определение и вызов нужной реализации метода в рантайме на основе типа объекта на котором вызват метод (определяется за счет заголовка объекта).

Построен на 3х явлениях – наследовании, имплементации и owerriding.

Т.е. если метод не переопределен, то наследник получить реализацию предка, если переопределен, то реализация может быть отличной. Выбор метода, который будет вызван происходит в рантайме на основании объекта на котором метод был вызван (на соновании ссылки).

К данному виду полиморфизма относится приведение типов (связан с ним) – up-casting (сужение типа) – процесс приведения объекта к базовому классу, происходит неявно (не требует дополнительного синтаксиса), при этом методы доступные к исполнению будут зависеть от типа переменной - предка, к которой привели ссылку.

Animal animal = new Cat();

При этом подразумевается:

Animal animal = (Animal) new Cat();

down-casting (расширение типа) – процесс приведения типа от базового к более конкретному в иерархии типов. Требует явного указания оператора приведения типа, в случае несоблюдения иерархии классов вылетит ClassCastException.

Чтобы избежать этого используется ключ слово instanceof или метод getClass() для проверки принадлежности к типу. Отличие в том, что instanceof проверяет принадлежность к иерархии классов (возвращает true/false), а getClass() возвращает точный объект класса class.

Cat cat = (Cat) new Animal();

Предок не может быть приведен к потомку, так как объект родитель не знает о методах и переменных, которые содержатся в наследнике.

Данный вид полиморфизма контролируется компилятором, то есть он ограничивает методы, доступные к исполнению на основании переменной, в которую помещается ссылка.

*Ad-hoc полиморфизм* - Мнимый полиморфизм. Это когда только создается видимость, что алгоритм полиморфный, но на самом деле в момент компиляции программы либо происходит подстановка другого алгоритма, в зависимости от типа/ов переданных аргументов, либо происходит неявное преобразование типа/ов аргумента/ов. Важно то, что в результате есть один или больше алгоритмов, каждый из которых мономорфный -- т.е. работает с данными строго одного типа. К нему относятся - автоматическое расширение типов, overloading.

*Параметрический полиморфизм* – Дженерики. Возможность алгоритма работать с данными неназванного типа (параметризированного), которым присвоены абстрактные имена.

Параметрический полиморфизм и ad-hoc полиморфизм осуществляются за счет компилятора (не в рантайме).

**Виртуальность полей и методов** – Методы, поддерживающие overriding – виртуальные (все, кроме final – нельзя переопределеить, static – принадлежат классу, private –наследуются, но нет доступа). Виртуальность методов обеспечивает выбор реализации метода на основе ссылки, находящейся в переменной.

Поля в Java не являются виртуальными - компилятор Java использует тип переменной для определения, к какому полю следует обращаться во время компиляции, и это определение остается неизменным во время выполнения.

public class A {  
 public static void main(String[] args) {  
 C c = new C();  
 B b = c;  
 c.field = 1; //   
 b.field = 2; // Не имеет значения, что в b лежит ссылка new C, к какому полю обратиться определяет компилятор  
 System.*out*.println(c.field);  
 System.*out*.println(b.field);  
 }  
}//Вывод 1 и 2  
class B {  
 int field; //Собственное поле класса B

}  
class C extends B {  
 int field; //Собственное поле класса C

//то есть класс содержит как бы 2 поля field, одно из них принадлежит B  
}

Демонстрация:

public class A {  
 public static void main(String[] args) {  
 C c = new C();  
 B b = c; // ссылка 1 и таже  
 c.field = 1;  
 b.field = 2;  
 System.*out*.println(c.field);  
 System.*out*.println(b.field);  
 //Выводом будет 2 и 2 поскольку С не имеет собственного поля field, но имеет унаследованное от класса B поле field, при этом ссылка на 1 и тот же объект.

}  
}  
class B {  
 int field;  
}  
class C extends B {

}

Что означает, что поля привязаны к переменной, а не к ссылке и инициализация происходит статически, а не в рантайме.

**Overriding** – механизм, позволяющий изменить реализацию дочернего метода, унаследованного от суперкласса. Если метод не переопределен, то будет использована раелизация близжайшего предка.

Правила переопределения:

1)Нельзя изменять тип аргументов, их кол-во и имя метода – это будет считаться другим методом, а не переопределенным – компилятор смотрит именно на 3 этих параметра функции, когда определяет их идентичность (актуально для overloading)

2)К переопределению доступны только методы подклассов

3) static (принадлежат классу, не виртуальны), final (явный запрет на переопределение) и private (являются внутренними методами и не могут быть использованы за пределами своего класса) методы не могут быть переопределены.

4) У переопределенных методов могут быть изменены доступ и возвращаемое значение:

а) Сужать доступ нельзя, расширять можно – так как сужение нарушает принципы полиморфизма – если сузить доступ в потомке, то возможно нарушение типобезопасности из-за того, что может быть закрыт доступ к нужному методу.

public static void main(String[] args) {  
 A obj = new B();  
 obj.doSomething();  
//метод будет невидим из класса Main, т.к. он приватный  
//соответственно он просто не может быть вызван и не имеет смысла  
}  
static class A {  
 public void doSomething() {  
 System.*out*.println("Doing something in A");  
 }  
}  
static class B extends A {  
 private void doSomething() { //ошибка  
 System.*out*.println("Doing something in B");  
 }  
}

б) Возвращать можно либо тот же тип, либо наследников возвращаемого типа – ковариантное поведение

public class Main {  
 public static void main(String[] args) {  
 A a = new A();  
 B b = new B();  
 A c = new B();  
 Number one = a.foo();//вызывается метод класса А, ожидаемый результат Number  
 Integer two = b.foo();//вызывается метод класса B, ожидаемый результат Integer, который может быть сужен до Number  
 Number three = c.foo();//вызывается метод класса B, но результат кастится только к Number, поскольку переменная A  
 //если дочерний метод возвращал бы Object, то невозможно было бы положить его в переменную Number, т.к. это сломает типобезопасность  
 A d = new B();  
 Number fo = d.foo();//Допустим тут лежит ссылка на Object  
 fo.byteValue();//Вызываем метод класса Number на объекте Object, в котором его нет и получаем ошибку  
 }  
 public static class A {  
 public Number foo() {  
 return 1;  
 }  
 }  
 public static class B extends A {  
 @Override  
 public Integer foo() {  
 return 1;  
 }  
 }  
}

**Overloading** – Создание методов в одном классе с одинаковым именем, но разными входными параметрами. Под входными параметрами подразумевается кол-во аргументов в методе и их тип. Остальные элементы сигнатуры, такие как: модификатор доступа, возвращаемый тип, статичность не являются критерием для перегрузки (При изменении этих параметров компилятор посчитает, что это тот же самый метод).

public void foo(int a) {}  
private static String foo(int b) {return "";}//Ошибка, тот же метод  
public void foo(double a){}

Благодаря перегрузке и автоматическому расширению типов возможен ad-hoc полиморфизм (методов на самом деле много и каждый работает с определенным набором аргументов и под них уже подгоняются аргументы см. выше).

**Сокрытие static методов** (hiding) – Проблема, появляющаяся при нахождении в одной иерархии классов статических методов с одинаковыми именами.

Обычно статический метод вызывается путем обращения к классу метода – в таком случае конфликта имен не возникнет. Но Java позволяет обращаться к static методам класса через объекты классов. При таком обращении и конфликте имен методов возникает ситуация, при которой ссылка на один и тот же объект вызывает разные статические методы. Это происходит из-за того, что компилятор выводит класс, у которого нужно вызвать статический метод не на основании ссылки, а на основании переменной.

public class Main {  
 public static void main(String[] args) {  
 B b = new B();  
 A a = b;  
 a.*foo*();  
 b.*foo*();  
 }  
 public static class A {  
 static void foo() {  
 System.*out*.println("A");  
 }  
 }  
 public static class B extends A {  
 static void foo() {  
 System.*out*.println("B");  
 }  
 }}

//A

//B

Когда дочерний класс хранит в себе ссылку на родительский это вроде как называется полиморфной переменной.

**Наследование –** Описание (конструирование) класса на основе уже существующего класса, в Java монотонное наследование, то есть нет возможности исключить у класса-потомка какое-либо поле или метод родительского класса – не наследуются только конструкторы – Java требует, при создании объекта-наследника сначала обратиться к конструктору предка. Это сделано по причине возможности наличия у класса родителя приватных полей, которые могут быть проинициализированы только родителем и возможность наличия у класса потомка новых полей, которые не могут быть проинициализированы конструктором родителем.

Приватные поля и методы наследуются, но получить к ним доступ компилятор не позволяет (можно проверить, если попробовать обратиться к age из класса B), то есть в коде ниже унаследованный метод setAge не переопределен и вызывается реализация родителя.

public static void main(String[] args) {  
 B b = new B();  
 b.setAge(22); /\*с помощью публичных методов унаследованных от класса A, получаем доступ  
 к полю age класса B \*/  
 System.*out*.println(b.getAge());  
 }  
 public static class A {  
 private int age;  
 public void setAge(int age) {  
 this.age = age;  
 }  
 public int getAge() {  
 return age;  
 }  
 }  
 public static class B extends A {  
 //Имеется унаследованное поле age, но к нему нет доступа  
 //Так же мы уснаследовали get и set  
 }  
}

Java не поддерживает множественного наследования состояний из-за diamond problem, при этом поддерживается множественное наследование реализаций и множественное наследование типов за счет интерфейсов (см. ниже)

*Отношение между классами*:

*Наследование* (является – is a) – наследник имеет поля и методы класса родителя, при этом возможно добавляется некий новый функционал.

*Ассоциация* (имеет – has a) – Обозначает некоторую связь между классами, основанную на их взаимодействии (Преподаватель – Студент. Студент учится у какого-то преподавателя). В программировании – один класс содержит другой в качестве поля.

*Агрегация и композиция* – частные случаи ассоциации, более конкретизированные отношения между этими объектами.

*Агрегация* – объект является частью другого объекта, но существовать первый может и без второго, то есть допускается некоторая вариативность в отношениях. При агрегации студент может входить в другие группы. При Агрегации объект Двигатель создается в другом месте программы и передается в конструктор Машина в качестве параметра. При уничтожении главного объекта, его составные части могут существовать.

(Студент входит в группу любителей физики.)

*Композиция* – Еще более жесткое отношение между объектами, при котором один объект является частью другого, при этом не может принадлежать кому-либо еще и в целом не существует отдельно, при этом первый объект тоже не должен существовать без второго (Машина и Двигатель). Обычно такой объект создается прямо в конструкторе, т.е. при создании Машины, в конструкторе будет создан объект двигатель.

Т.е. основным отличием агрегации и композиции является наличие зависимости существования его составных частей от главного класса. Композиция: уничтожен главный – уничтожены все его части. Агрегация: уничтожен главный – составные части могут не быть уничтожены.

*Делегирование* – Отношение между классами или внутри них, при которых один объект или часть одного класса передает выполнение задачи другому.

**Абстракция –** Выделение главных характеристик класса или отбрасывание второстепенных, вынесение характерных для всей иерархии классов свойств в виде базовых, универсальных концепций. Так же абстракция направлена на описание поведения, не фокусируясь на реализации этого поведения (что делает, но все-равно как делает).

Выражается через использование абстрактных классов и интерфейсов.

**Абстрактные классы и интерфейсы –** абстрактные классы могут иметь конструкторы, задавая тем самым конструирование всех своих наследников (создание объекта происходит с помощью обращения к конструктору предка) Что относит предмет к категории этих предметов.

Интерфейсом задается какое-то свойство или способность класса, оно может быть присуще двум классам, никоим образом не относящимся друг к другу. Абстрактный класс же реализует отношение is a по отношению к своим наследникам.

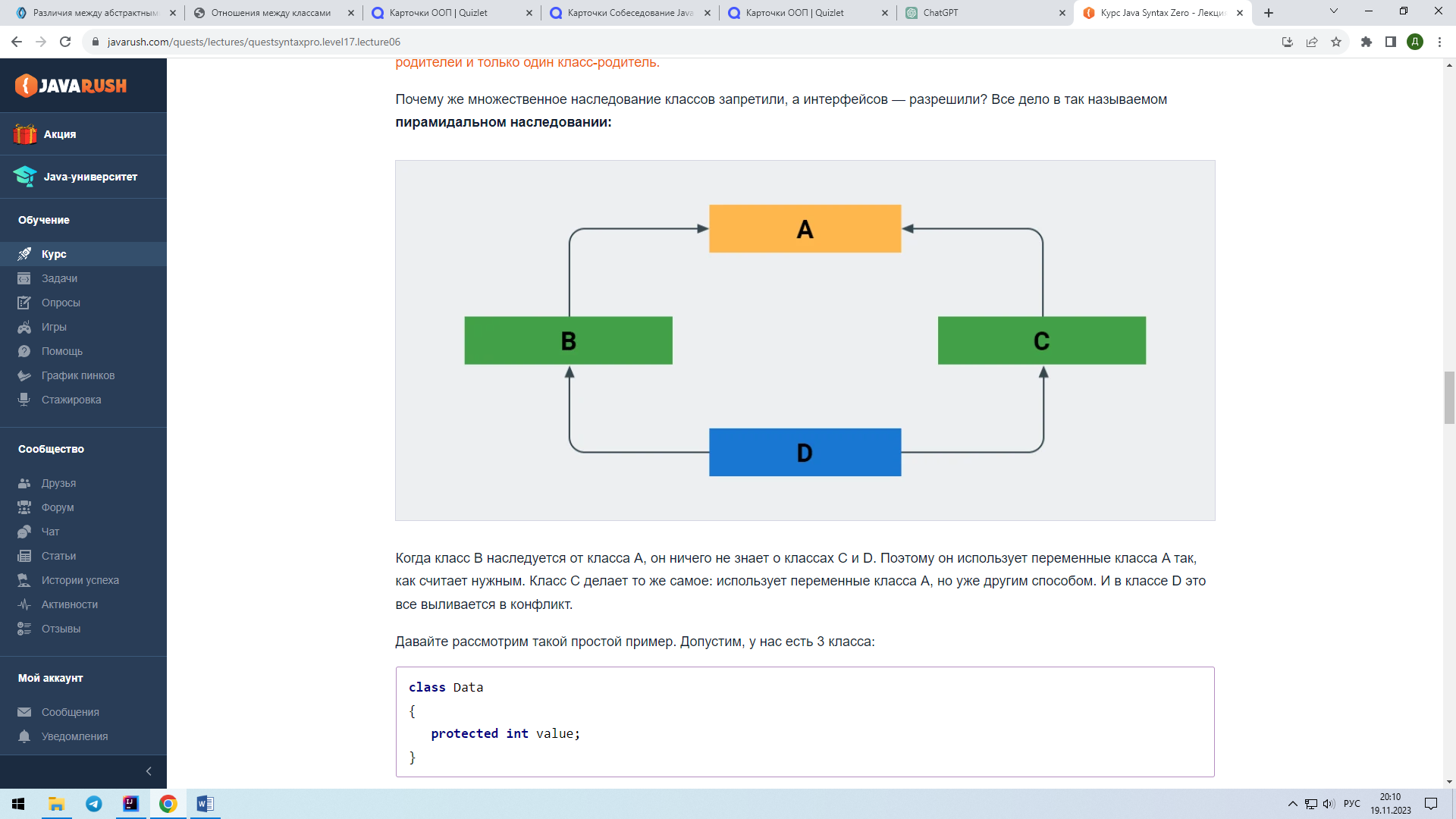
**Абстрактный класс и интерфейс**:

Главные свойства абстрактного класса – реализует отношение is a, может хранить в себе состояние объекта(поля). Интерфейс – не может хранить в себе состояния (нет полей) и описывает поведение или свойства каких-либо классов. *Эти отличия являются самыми главными*, остальные либо вытекают из этих утверждений, либо являются техническими отличиями.

Техническое описание:

|  |  |
| --- | --- |
| Абстрактный класс | Интерфейс |
| Есть поля объекта | Нет полей объекта, все поля, объявленные в интерфейсе неявно public static final – то есть константы, принадлежащие к классу, а не объекту, если написать int a, то компилятор потребует проинициализировать это поле, как для final переменной  public static final int *a* = 1; |
| Реализует отношение is a – класс является абстракцией над конкретными представителями этого класса | Описывает свойства каких-либо классов, они могут быть абсолютно не связаны между собой |
| Нет множественного наследования (extends) | Есть множественная имплементация (implements) |
| Есть конструктор – так как абстрактный класс – некий абстрактный представитель класса, то у него должен быть конструктор | Может связывать абсолютно разные сущности, поэтому не может иметь конструктора |
| Может иметь любые модификаторы доступа для методов. Все методы, кроме абстрактных должны иметь реализацию. Абстрактные не могут быть приватными – это не имеет смысла. | Методы могут быть приватными – т.е. внутренние, для сокращения объема кода с Java 9 (должны быть реализованы) – логично, что могут быть использованы только в методах, имеющих реализацию в интерфейсе, то есть в дефолтных, публичными – необходимые к реализации, дефолтные – наследуемые методы, имеющие поведение по умолчанию (могут быть переопределены) – методы не могут быть приватными и дефолтными одновременно – противоречит концепции приватности и дефолтности. Помимо этого так же могут быть статические методы, которые public по умолчанию, но могут быть и private |
| Может как наследовать другие классы, так и имплементить интерфейсы | Иерархия наследования может строится только из других интерфейсов (extends another interface) |
| Не может быть final, т.к. подразумевает, что от него будут наследоваться | Не может быть final, т.к. подразумевает, что от него будут имплемениться |

Проблема множественного наследования – Java не поддерживает *множественного наследования состояния* (то есть проблема в полях объектов). Есть классы B и C с одинаковыми полями int age, которым мы хотим задать общего наследника A, так как родителя имеют одинаковые поля, то непонятно, каким полям пользоваться, то ест возникает конфликтная ситуация.



При этом в Java поддерживается *множественное наследование реализаций* – за счет интерфейсов и наличия у них default методов, которые должны иметь реализацию.

Для решения diamond-problem в случае с реализацией одинаковых по сигнатуре default методов одним классом компилятор потребует решить конфликт реализаций (написать собственную реализацию метода)

interface Readable {  
 default void print(){  
 System.*out*.println("1");  
 }  
}

interface Readable2 {  
 default void print(){  
 System.*out*.println("2");  
 }  
}

public class Main implements Readable2, Readable {  
 public static void main(String[] args) {  
 Main main = new Main();  
 main.print();  
 }  
 @Override  
 public void print() {  
 //можно написать свою реализацию, главное переопределить

Readable2.super.print();//явное обращение к реализации Readable2

}  
}

еще можно просто перекрыть один метод другим через наследование

interface Readable {  
 default void print(){  
 System.*out*.println("1");  
 }  
}  
interface Readable2 extends Readable { //метод print() Readable2 перекроет метод своего предка, т.к. вызывается близжайшая реализация  
 default void print(){  
 System.*out*.println("2");  
 }  
}  
public class Main implements Readable2, Readable {  
 public static void main(String[] args) {  
 Main main = new Main();  
 main.print();  
 }  
 //не нужно переопределять метод, т.к. понятно, какой метод вызывать  
}

В такой ситуации компилятор сначала требует переопределить default метод print() – что является необязательным условием для них при отсутствии конфликтов имен, а затем явно указать в нем к какому методу интерфейса необходимо обратиться, либо написать свою реализацию.\*\*

*Множественное наследование типа –* достигается так же за счет множественной имплементации интерфейсов. То есть каждый интерфейс класса по сути определяет его новый тип – что так же играет важную роль в полиморфизме.