1. ООП, принципы ООП

Объектно-ориентированное программирование, или ООП — это одна из парадигм разработки. Парадигмой называют набор правил и критериев, которые соблюдают разработчики при написании кода. Суть понятия объектно-ориентированного программирования в том, что все программы, написанные с применением этой парадигмы, состоят из объектов. Каждый объект — это определённая сущность со своими данными и набором доступных действий.

Абстракция в объектно-ориентированном программировании (ООП) — это один из ключевых принципов, который заключается в выделении наиболее важных характеристик объекта или системы, скрывая при этом ненужные детали. Идея абстракции позволяет сосредоточиться на том, что объект делает, а не на том, как он это делает. Например, люди не представляют себе автомобиль как набор из десятков тысяч отдельных деталей. Они думают о нем, как о четко определенном объекте со своим уникальным поведением. Такая абстракция позволяет людям доехать на автомобиле до продуктового магазина, не перегружаясь сложностью индивидуальных деталей. Они могут игнорировать подробности работы двигателя, коробки передач и тормозной системы. Взамен они могут свободно использовать объект как единое целое. Эффективный способ управления абстракцией предусматривает применение иерархических классификаций. Они позволяют разбивать семантику сложных систем на более управляемые части. Снаружи автомобиль представляет собой единый объект. Но погрузившись внутрь, вы увидите, что автомобиль состоит из нескольких подсистем: рулевого управления, тормозов, аудиосистемы, ремней безопасности, обогрева, навигатора и т.д. Каждая подсистема в свою очередь содержит более специализированные узлы. Скажем, в состав аудиосистемы может входить радиоприемник, проигрыватель компакт-дисков и/или проигрыватель MP3. Суть в том, что вы управляете сложностью автомобиля (или любой другой сложной системы) за с чет использования иерархических абстракций. Иерархические абстракции сложных систем также можно применять к компьютерным программам. Данные из традиционной, ориентированной на процессы, программы могут быть трансформированы путем абстракции в составляющие ее объекты. Последовательность шагов процесса может стать совокупностью сообщений, передаваемых между этими объектами. Таким образом, каждый объект описывает свое уникальное поведение. Вы можете воспринимать такие объекты как конкретные сущности, которые реагируют на сообщения, указывающие им о необходимости делать что-то. В этом и заключается суть ООП.

Инкапсуляция представляет собой механизм, который связывает вместе код и обрабатываемые им данные, а также защищает их от внешнего вмешательства и неправильного использования. Основная идея инкапсуляции — скрыть внутренние детали реализации объекта, предоставляя только необходимые методы для взаимодействия с ним. Инкапсуляцию можно считать защитной оболочкой, которая предотвращает произвольный доступ к коду и данным из другого кода, определенного вне оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируется через четко определенный интерфейс. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует массу информации о вашем двигателе, такую как величина ускорения, наклон поверхности, по которой двигается автомобиль, и положение рычага переключения передач. Вы, как пользователь, располагаете только одним способом влияния на эту сложную инкапсуляцию: перемещение рычага переключения передач. Вы не можете воздействовать на коробку передач, скажем, с помощью сигнала поворота или дворников. Таким образом, рычаг переключения передач является четко определенным (и действительно уникальным) интерфейсом к коробке передач. Вдобавок то, что происходит внутри коробки передач, никак не влияет на объекты за ее пределами. Например, переключение передачи не включает фары! Поскольку автоматическая коробка передач инкапсулирована, десятки производителей автомобилей могут реализовать ее так, как им заблагорассудится. Однако с точки зрения водителя все они работают одинаково. Ту же самую идею можно применить и к программированию. Сила инкапсулированного кода в том, что каждый знает, как получить к нему доступ, и потому может использовать его независимо от деталей реализации и без каких-либо опасений столкнуться с неожиданными побочными эффектами. Основой инкапсуляции в Java является класс. Класс определяет структуру и поведение (данные и код), которые будут общими для набора объектов: каждый объект заданного класса содержит структуру и поведение, определенные классом, как если бы он был "отлит" в форме класса. По этой причине объекты иногда называют экземплярами класса. Таким образом, класс представляет собой логическую конструкцию, а объект имеет физическую реальность. Поскольку целью класса является инкапсуляция сложности, существуют механизмы для сокрытия сложности реализации внутри класса. Методы или переменные в классе могут быть помечены как закрытые или открытые. Открытый интерфейс класса представляет все, что должны или могут знать внешние пользователи класса. Доступ к закрытым методам и данным возможен только из кода, который является членом класса. Следовательно, любой другой код, не являющийся членом класса, не сможет получить доступ к закрытому методу или переменной. Так как доступ к закрытым членам класса другие части вашей программы могут получить только через открытые методы класса, вы можете гарантировать, что не будет совершено никаких неподходящих действий.

Представьте себе класс BankAccount (банковский счёт). У него есть данные, такие как баланс, и методы, которые позволяют изменять баланс, например, deposit() (внесение денег) и withdraw() (снятие денег). Чтобы защитить данные (баланс) от неправильного использования, мы скрываем его от прямого доступа, а для работы с балансом предоставляем методы. Это и есть инкапсуляция.

```
1 public class BankAccount {
2     private double balance;
3 
4     public void deposit(double amount) {
5         if (amount > 0) {
6             balance += amount;
7         }
```

```
8       }
9
10      public void withdraw(double amount) {
11          if (amount > 0 && amount <= balance) {
12              balance -= amount;
13          }
14      }
15
16      public double getBalance() {
17          return balance;
18      }
19  }
```

Контроль доступа: инкапсуляция позволяет ограничить доступ к данным и разрешить их изменение только через определённые методы. Это предотвращает несанкционированное или ошибочное изменение данных.
Защита целостности данных: скрывая внутренние данные и предоставляя методы для работы с ними, можно гарантировать, что данные всегда будут находиться в корректном состоянии (например, проверяя условия перед изменением данных).
Гибкость в изменениях: внутреннюю реализацию класса можно изменять, не затрагивая остальной код. Внешний код будет работать через методы, которые могут не изменяться, даже если изменится способ хранения данных.
Снижение сложности: инкапсуляция помогает скрывать сложные детали реализации и предоставляет простой интерфейс для взаимодействия с объектом.
Наследование представляет собой процесс, посредством которого один объект приобретает свойства другого объекта. Оно важно, т.к. померживает концепцию иерархической классификации. Ранее уже упоминалось, что большинство знаний стало доступным за счет иерархической (т.е. нисходящей) классификации. Например, золотистый ретривер является частью класса "собаки"; который в свою очередь относится к классу "млекопитающие"; входящему в состав более крупного класса "животные": В отсутствие иерархий каждый объект должен был бы явно определять все свои характеристики. Тем не менее, используя наследование, объекту нужно определить только те качества, которые делают его уникальным внутри своего класса. Он может наследовать общие атрибуты от своего родителя. Таким образом, именно механизм наследования позволяет одному объекту быть специфическим экземпляром более общего случая. Давайте подробнее рассмотрим сам процесс. Большинство людей естественным образом воспринимают мир как состоящий из иерархически связанных друг с другом объектов, таких как животные, млекопитающие и собаки. При желании описать животных абстрактно вы бы сказали, что у них есть некоторые характерные признаки вроде размера, умственных способностей и типа костной системы. Животным также присущи определенные поведенческие аспекты: они едят, дышат и спят. Такое описание характерных признаков и поведения является определением класса для животных. Если бы требовалось описать более конкретный класс животных, скажем, млекопитающих, то они имели бы более конкретные характерные признаки вроде типа зубов и молочных желез. Такое определение известно как подкласс животных, а животные являются суперклассом млекопитающих. Поскольку млекопитающие - просто более точно определенные животные, они наследуют все характерные признаки животных. Находящийся глубоко в иерархии классов подкласс наследует все характерные признаки от каждого из своих предков. Наследование также взаимодействует с инкапсуляцией. Если заданный класс инкапсулирует некоторые характерные признаки, тогда любой подкласс будет иметь те же самые признаки плюс любые, которые он добавляет как часть своей специализации. Это ключевая концепция, обеспечивающая возрастание сложности объектно-ориентированных программ линейно, а не геометрически. Новый подкласс наследует все характерные признаки всех своих предков. У него нет непредсказуемых взаимодействий с большей частью остального кода в системе.
Полиморфизм (от греческого "много форм") представляет собой средство, которое позволяет использовать один интерфейс для общего класса действий. Конкретное действие определяется природой ситуации. Это означает возможность использования одного и того же имени метода для разных типов объектов, при этом каждый объект может реализовывать этот метод по-своему. Возьмем в качестве примера стек (т.е. список, работающий по принципу "последним пришел - первым обслужен"). У вас может быть программа, требующая стеки трех типов. Один стек используется для целых значений, другой - для значений с плавающей точкой и третий - для символов. Каждый стек реализуется по тому же самому алгоритму, даже если хранящиеся данные различаются. В языке, не являющемся объектно-ориентированным, вам придется создать три разных набора стековых процедур с отличающимися именами. Но благодаря полиморфизму в Java вы можете указать общий набор стековых процедур с одинаковыми именами. Как правило, концепция полиморфизма часто выражается фразой "один интерфейс, несколько методов": это означает возможность разработки общего интерфейса для группы связанных действий, что поможет уменьшить сложность, позволив использовать один и тот же интерфейс для указания общего класса действий. Задачей компилятора будет выбор конкретного действия (т.е. метода) применительно к каждой ситуации. Вам, как программисту, не придется делать такой выбор вручную. Вам понадобится только запомнить и задействовать общий интерфейс. Продолжая аналогию с собаками, можно отметить, что обоняние собаки полиморфно. Если собака почует кошку, то она залает и побежит за ней. Если собака почувствует запах еды, тогда у нее начнется слюноотделение, и она побежит к миске. В обеих ситуациях работает одно и то же обоняние. Разница в том, что именно издает запах, т.е. тип данных, с которыми имеет дело собачий нос! Та же общая концепция может быть реализована в Java, поскольку она применяется к методам внутри программы Java.
Виды полиморфизма:
Полиморфизм времени компиляции (статический полиморфизм):

Реализуется через перегрузку методов (method overloading) и перегрузку операторов.
В этом случае решение о том, какой метод вызвать, принимается во время компиляции.
Полиморфизм времени выполнения (динамический полиморфизм):
Реализуется через переопределение методов (method overriding) и наследование.
Выбор метода, который будет вызван, происходит во время выполнения программы в зависимости от типа объекта.
Пример полиморфизма времени выполнения:
Представим класс Animal, у которого есть метод sound(), и два подкласса: Dog и Cat, которые по-своему реализуют этот метод. Полиморфизм позволяет нам обращаться к объектам этих классов через общий тип (Animal), но вызывать разные реализации метода sound() в зависимости от конкретного объекта.

```java
class Animal {
    public void sound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Woof");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myAnimal.sound();  // Some generic animal sound
        myDog.sound();     // Woof
        myCat.sound();     // Meow
    }
}
```

У нас есть общий класс Animal с методом sound(), который может быть переопределен в подклассах. Классы Dog и Cat переопределяют метод sound() с реализацией, характерной для каждого животного. В методе main() переменные типа Animal ссылаются на объекты различных подклассов (Dog и Cat), но когда вызывается метод sound(), выполняется соответствующая версия метода для конкретного объекта.
Преимущества полиморфизма:
Упрощение кода: полиморфизм позволяет создавать более универсальные и гибкие программы, где можно использовать один и тот же код для работы с объектами разных классов.
Расширяемость: новые классы можно добавлять без изменения существующего кода. Например, можно создать новый класс Bird, который также будет наследовать Animal и реализовывать свой метод sound(), и существующий код продолжит работать с этим классом.
Поддержка динамического поведения: полиморфизм времени выполнения позволяет объектам разных классов вести себя по-разному, не нарушая общую структуру программы.
Пример полиморфизма времени компиляции (перегрузка методов):
Полиморфизм времени компиляции достигается через перегрузку методов — это когда методы с одним и тем же именем имеют разные сигнатуры (разные параметры или типы параметров).

```java
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));          // Result: 5
        System.out.println(calc.add(2.5, 3.5));      // Result: 6.0
    }
}
```

У метода add() есть две версии с разными типами параметров: одна принимает целые числа, другая — вещественные. Java во время компиляции определяет, какую версию метода вызывать, исходя из типа переданных аргументов.
Как итог, полиморфизм позволяет использовать один и тот же интерфейс (методы) для работы с объектами разных типов, что делает программы более гибкими, удобными для расширения и поддерживаемыми.
Полиморфизм важен для ООП по одной причине: он позволяет универсальному классу определять методы, которые будут общими для всех производных от него классов, одновременно разрешая подклассам определять индивидуальные реализации некоторых или всех общих методов.

При правильном применении полиморфизм, инкапсуляция и наследование объединяются для создания программной среды, которая поддерживает разработку гораздо более надежных и масштабируемых программ, чем в случае использования модели, ориентированной на процессы. Хорошо спроектированная иерархия классов является основой для многократного использования кода, в разработку и тестирование которого вы вложили время и усилия. Инкапсуляция позволяет вам со временем переносить свои реализации, не нарушая код, который зависит от открытого интерфейса ваших классов. Полиморфизм дает возможность создавать чистый, понятный, читабельный и устойчивый код.
Если о собаках интересно думать с точки зрения наследования, то автомобили больше похожи на программы. Все водители полагаются на наследование для управления различными типами (подклассами) транспортных средств. Независимо от того, является транспортное средство школьным автобусом, седаном "Мерседес"; "Порше" или семейным минивэном, все водители могут более или менее находить и управлять рулем, педалью тормоза и педалью газа. Немного повозившись с рычагом переключения передач, большинство людей смогут даже отличить ручную коробку передач от автоматической, т.к. они в основном понимают их общий суперкласс - коробку передач. Пользуясь автомобилями, люди постоянно взаимодействуют с инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность, а интерфейс настолько прост, что ими можно управлять спомощью ног! Реализация двигателя, тип тормозов и размер шин не влияют на то, каким образом вы взаимодействуете с определением класса педалей. Последний принцип, полиморфизм, четко отражается в способности производителей автомобилей предлагать широкий спектр вариантов для одного и того же транспортного средства. Например, вы можете получить антиблокировочную или традиционную тормозную систему, рулевое управление с гидроусилителем или реечной передачей, а также 4-, 6- или 8-цилиндровый двигатель либо электромобиль. В любом случае вы по-прежнему будете нажимать на педаль тормоза для остановки, поворачивать руль для смены направления и нажимать на педаль газа, когда хотите двигаться. Один и тот же интерфейс может применяться для управления несколькими отличающимися реализациями. Как видите, благодаря применению инкапсуляции, наследования и полиморфизма отдельные части трансформируются в объект, известный как автомобиль. То же самое относится и к компьютерным программам. За счет применения принципов ООП различные части сложной программы могут быть объединены в единое, надежное, сопровождаемое целое. Каждая программа на Java является объектно-ориентированной или, говоря точнее, каждая программа на Java включает в себя инкапсуляцию, наследование и полиморфизм.

2. Классы и объекты

Класс лежит в самом центре Java. Он представляет собой логическую конструкцию, на которой построен весь язык Java, потому что она определяет форму и природу объекта. Таким образом, класс формирует основу для объектно-ориентированного программирования (ООП) на Java. Любая концепция, которую вы хотите реализовать в программе на Java, должна быть инкапсулирована внутри класса. Вероятно, наиболее важная характеристика класса заключается в том, что он определяет новый тип данных. После определения новый тип можно применять для создания объектов такого типа. Следовательно, класс - это шаблон для объекта, а объект - это экземпляр класса. Объект - сущность, которая объединяет данные и поведение, то есть свойства (атрибуты) и методы (функции), которые могут быть выполнены с этими данными. Так как объект является экземпляром класса, вы часто будете видеть, что слова объект и экземпляр используются взаимозаменяемо. Данные, или переменные, определенные в классе, называются переменными экземпляра. Код содержится внутри методов. В совокупности методы и переменные, определенные в классе, называются членами класса. В большинстве классов переменные экземпляра обрабатываются и доступны с помощью методов, определенных для этого класса. Таким образом, как правило, именно методы определяют, как можно использовать данные класса. Переменные, определенные внутри класса, называются переменными экземпляра из-за того, что каждый экземпляр класса (т.е. каждый объект класса) содержит собственную копию этих переменных.

3. Создание и инициализация объектов, вызов методов

Создание класса означает создание нового типа данных, который можно применять для объявления объектов этого типа. Однако получение объектов класса представляет собой двухэтапный процесс. Во-первых, потребуется объявить переменную типа класса. Такая переменная не определяет объект, а просто может ссылаться на объект. Во-вторых, необходимо получить физическую копию объекта и присвоить ее этой переменной, для чего служит операция new. Оператор new выделяет память для нового объекта на куче. Куча — это область памяти, в которой хранятся все объекты, созданные во время выполнения программы. Операция new динамически (т.е. во время выполнения) выделяет память для объекта и возвращает ссылку на нее, которая по существу является адресом в памяти объекта, выделенной new. Затем ссылка сохраняется в переменной. Таким образом, в Java все объекты класса должны размещаться динамически. Преимущество этого подхода состоит в том, что ваша программа может создать столько объектов, сколько требуется во время ее выполнения. Однако поскольку память конечна, возможно, что new не сможет выделить память под объект из-за нехватки памяти. В такой ситуации возникает исключение времени выполнения.
Переменные, которые ссылаются на объекты, не хранят сами объекты, а хранят ссылки на них. Ссылка — это "адрес" в памяти, где хранится объект. Когда вы присваиваете одну ссылочную переменную другой, вы копируете только ссылку, а не сам объект.
Когда вы создаёте объект с помощью оператора new, переменная типа объекта будет хранить ссылку на него. Эта переменная не содержит сам объект, а указывает на его место в памяти (куче).
Если вы присвоите одну ссылочную переменную другой, обе переменные будут указывать на один и тот же объект. Изменение объекта через одну переменную отразится на объекте, на который указывает другая переменная, так как они ссылаются на один и тот же объект. Если сравнивать две ссылочные переменные с помощью оператора ==, проверяется, указывают ли они на один и тот же объект в памяти. Для проверки содержимого объектов (например, полей) используется метод .equals(), который по умолчанию сравнивает ссылки, но может быть переопределён для сравнения полей объекта. Переменной-ссылке можно присвоить значение null, что означает, что она больше не ссылается на какой-либо объект. Когда все переменные-ссылки на объект становятся равны null или выходят из области видимости, объект становится недоступным, и Java может его удалить с помощью сборщика мусора (Garbage Collector).
Метод — это именованный блок кода, который выполняет определённую задачу. Методы позволяют разделить программу на логические части и многократно использовать код, избегая дублирования. Они также помогают структурировать программу, делая её более читаемой и управляемой. Метод может принимать входные данные (параметры), выполнять вычисления или действия и возвращать результат (или ничего не возвращать). В Java метод всегда принадлежит классу и вызывается на объекте (если это нестатический метод) или напрямую через класс (если это статический метод).

```
1 returnType methodName(parameters) {
2     // Body
3     return value;  // if not static
4 }
```

returnType — тип данных, который метод возвращает. Если метод не возвращает значение, используется void.
methodName — имя метода, через которое он будет вызываться.
parameters — список параметров (входные данные) метода. Если параметров нет, скобки остаются пустыми.
return value; — оператор, который возвращает значение из метода (если метод не void).
Чтобы вызвать метод, нужно использовать его имя, указав при этом необходимые параметры, если они есть. В Java вызов методов зависит от их типа: статические методы можно вызывать через имя класса, а нестатические — через объект класса. Оператор . (разделитель) используется для обращения к членам класса — это могут быть поля (переменные), методы или внутренние классы. Оператор точки позволяет получить доступ к свойствам объекта или класса и вызывать методы.

4. Члены класса, поля, методы, конструкторы, модификаторы доступа

Члены класса
В Java члены класса — это составляющие элементы, которые описывают структуру и поведение класса. В общем, они включают в себя:
Поля (переменные экземпляра) — это данные, которыми класс оперирует.
Методы — функции или процедуры, описывающие поведение класса.
Конструкторы — специальные методы, которые вызываются при создании объектов класса.
Статические блоки и инициализаторы — код, который выполняется при загрузке класса или при создании объекта.
Вложенные классы и интерфейсы — классы или интерфейсы, которые определены внутри другого класса.
Поля класса — это переменные, которые хранят данные о состоянии объекта или всего класса. В Java они бывают нескольких типов:
Статические переменные (переменные класса) — это переменные, которые принадлежат не объекту, а самому классу. Эти переменные создаются в единственном экземпляре при загрузке класса в память и общие для всех объектов

класса. Все объекты этого класса будут использовать одну и ту же статическую переменную, и изменения в одном объекте будут видны в других.
Методы — это функции, которые описывают поведение объекта. Они могут выполнять различные действия над данными объекта или над внешними параметрами. Параметр - это переменная, определенная методом, которая получает значение при вызове метода. Аргумент - это значение, которое передается методу при его вызове.
Методы можно разделить на:
Нестатические методы (методы экземпляра) — работают с конкретным объектом и могут обращаться к его переменным и методам.
Статические методы — принадлежат классу и могут быть вызваны без создания объекта.
Конструктор — это специальный метод, который вызывается при создании нового объекта класса. Конструктор не возвращает значений и имеет то же имя, что и класс. Конструктор используется для задания начального состояния объекта.
Статический блок инициализации — это блок кода, который выполняется один раз при загрузке класса в память.
Нестатический блок инициализации — выполняется каждый раз, когда создаётся объект класса.

```java
class Car {
    static int numberOfCars;

    // Static
    static {
        numberOfCars = 0;
        System.out.println("Class Car loaded.");
    }

    // Not static
    {
        System.out.println("New car object created.");
    }

    Car() {
        numberOfCars++;
    }
}

public class Check {
    public static void main(String[] args) {
        Car c1 = new Car();
        System.out.println(Car.numberOfCars);
        Car c2 = new Car();
        System.out.println(Car.numberOfCars);
    }
}
/*Res:
Class Car loaded.
New car object created.
1
New car object created.
2*/
```

Вложенные классы и интерфейсы — классы или интерфейсы, которые определены внутри другого класса. Вложенные классы (nested classes) — это классы, которые определены внутри другого класса. Они могут быть как статическими, так и нестатическими.
Статические вложенные классы (static nested classes) — это класс, который связан с внешним классом, но не имеет напрямую доступа к нестатическим членам внешнего класса.
Внутренние классы (inner classes) — это нестатические классы, которые могут обращаться к членам внешнего класса, включая нестатические переменные и методы.

Поля

Поля класса — это переменные, которые хранят данные или состояние объекта класса. Они играют ключевую роль, определяя, какие данные (свойства) будут храниться в каждом объекте класса. Поля класса позволяют каждому объекту хранить своё состояние и использовать его в методах. В Java поля класса можно разделить на несколько типов: переменные экземпляра (нестатические поля), статические поля (переменные класса), а также их различные комбинации с модификаторами доступа.
1. Переменные экземпляра (нестатические поля)
Переменные экземпляра — это поля, которые принадлежат каждому отдельному объекту класса. Каждый объект класса имеет свою копию этих переменных, что позволяет им хранить своё собственное состояние. Переменные экземпляра инициализируются при создании объекта и существуют, пока объект существует.

Характеристики переменных экземпляра:
Принадлежат объектам. У каждого объекта своя копия переменной.
Инициализируются при создании объекта.
Не могут быть вызваны без объекта.
Могут иметь модификаторы доступа (public, private, protected).
2. Статические блоки (переменные класса)
Статические переменные — это поля, которые принадлежат классу, а не отдельным объектам. Все объекты класса используют одну и ту же статическую переменную. Она создаётся при загрузке класса и уничтожается, когда программа завершает работу или когда класс выгружается.
Характеристики статических переменных:
Принадлежат классу и существуют в единственном экземпляре для всех объектов.
Инициализируются при загрузке класса и остаются в памяти до окончания программы.
Могут быть вызваны без создания объекта, через имя класса.
Общие для всех объектов данного класса. Изменения в статической переменной одного объекта видны всем остальным объектам.

Метод

Методы в Java — это функции, которые принадлежат классу и выполняют определённые действия. Они позволяют организовать код, делать его повторно используемым и структурированным. Методы могут принимать параметры, выполнять операции и возвращать результаты.
Основные элементы метода:
Тип возврата: указывает тип данных, который метод возвращает (например, int, String, void). Если метод ничего не возвращает, используется ключевое слово void.
Имя метода: это имя, через которое можно вызвать метод. По соглашению имена методов записываются в стиле camelCase.
Параметры: это входные данные, передаваемые методу. Параметры указываются в круглых скобках и могут быть одного или нескольких типов данных.
Тело метода: это блок кода, который выполняется при вызове метода. Он заключён в фигурные скобки {}.
Типы методов:
Статические методы: это методы, которые принадлежат классу, а не экземпляру объекта. Для вызова таких методов не нужно создавать объект класса, они вызываются через имя класса.
Нестатические методы (методы экземпляра): это методы, которые принадлежат объектам класса. Для их вызова нужно создать экземпляр класса.
Методы с возвращаемым значением: эти методы возвращают значение, и их тип возврата должен быть указан перед именем метода. Для возврата значения используется ключевое слово return.
Методы с параметрами: параметры передаются в метод через круглые скобки. Это позволяет методам работать с переданными данными.
В Java можно определять несколько методов с одинаковым именем, но с разными параметрами (или разным типом возвращаемых данных). Это называется перегрузкой методов. Она позволяет вызывать метод с различными наборами параметров.

Конструктор

Инициализировать все переменные в классе при каждом создании его экземпляра может быть утомительно. Конструктор инициализирует объект немедленно после создания. Он имеет такое же имя, как у класса, где находится, и синтаксически похож на метод (не является им, тк конструктор не имеет возвращаемого типа, даже void. Это главное отличие от методов, которые всегда имеют тип возврата (или void, если ничего не возвращают)). После определения конструктор автоматически вызывается при создании объекта до завершения операции new. Конструкторы выглядят немного странно, потому что у них нет возвращаемого типа, даже void. Причина в том, что неявным возвращаемым типом конструктора класса является сам класс. Задача конструктора - инициализировать внутреннее состояние объекта, чтобы код, создающий экземпляр, немедленно получил в свое распоряжение полностью инициализированный и пригодный для использования объект.
переменная-класса = new имя-класса ( );
Скобочки после имени класса нужны, ибо на самом деле происходит вызов конструктора класса. Если конструктор для класса не определяется явно, тогда компилятор Java создает стандартный конструктор. При использовании стандартного конструктора все неинициализированные переменные экземпляра будут иметь стандартные значения, которые для числовых типов, ссылочных типов и логических значений равны соответственно нулю, null и false.
Конструкторы также можно перегружать.

```
1 class Box {
2     double width;
3     double height;
4     double depth;
5     Box (Box ob) {
6         width = ob.width;
7         height = ob.height;
8         depth = ob.depth;
9     Box (double w, double h, double d) {
```

```
        width = w;
        height = h;
        depth = d;
    Box () {
        width = -1;
        height = -1;
        depth = -1;
    Box (double len) {
    width = height = depth = len;
    }
    double volume() {
    return width * height * depth;
    }
class OverloadCons2 {
    public static void main (String [] args ) {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        Box myclone = new Box(mybox1);
    {
}
```

this

Ключевое слово this используется для ссылки на текущий объект, в контексте которого выполняется метод или блок кода (те если у нас есть что-то типо объектА.метод(аргументы), а в теле указанного метода используется this, то под this подразумевается объектА).

Основные случаи использования this

Для различения полей объекта и параметров метода или конструктора.

Для вызова конструктора из другого конструктора (chaining).

Для передачи текущего объекта в метод или конструктор.

Для возврата текущего объекта из метода.

1. Различение полей объекта и параметров

Когда имя параметра метода или конструктора совпадает с именем поля класса, возникает проблема "теневой переменной" — локальная переменная (параметр) "затеняет" поле класса. В таких случаях для доступа к полям объекта используется ключевое слово this.

```
class Car {
    String model;
    int year;

    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void displayInfo() {
        System.out.println("Model: " + this.model + ", Year: " + this.year);
    }
}
```

this.model и this.year указывают на поля класса. Без ключевого слова this Java будет использовать локальные переменные model и year (параметры конструктора), а не поля объекта, что приведет к ошибке.

2. Вызов одного конструктора из другого (Constructor Chaining)

Можно вызвать один конструктор из другого в том же классе с помощью this(). Это помогает избежать дублирования кода, когда нужно выполнить общие задачи при создании объекта в нескольких конструкторах.

```
class Car {
    String model;
    int year;

    Car(String model) {
        this(model, 2020);
    }

    Car(String model, int year) {
        this.model = model;
```

```
        this.year = year;
    }

    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        car1.displayInfo();  // Res: Model: Tesla, Year: 2020
    }
}
```

Конструктор с одним параметром вызывает другой конструктор с двумя параметрами с помощью this(model, 2020). Это сокращает код и предотвращает дублирование инициализации полей, таких как model и year.
3. Передача текущего объекта в качестве аргумента
Ключевое слово this может использоваться для передачи текущего объекта в методы других классов. Это полезно, если необходимо передать ссылку на сам объект в другой метод или конструктор.

```
class Car {
    String model;
    int year;
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void displayInfo() {
        System.out.println("Model: " + this.model + ", year: " + this.year);
    }

    void printCarInfo(PrintHelper helper) {
        helper.print(this);
    }
}

class PrintHelper {
    void print(Car car) {
        car.displayInfo();
        }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla", 2023);
        PrintHelper helper = new PrintHelper();
        car1.printCarInfo(helper);
    }
}
```

Создание объекта класса Car: в методе main создается объект класса Car с моделью "Tesla" и годом выпуска 2023. В этот момент вызывается конструктор Car(String model, int year), который принимает значения "Tesla" и 2023. Поля объекта car1 инициализируются: this.model принимает значение "Tesla", а this.year — значение 2023. Объект car1 теперь содержит информацию о машине с моделью "Tesla" и годом выпуска 2023. Далее создается объект класса PrintHelper:
PrintHelper helper = new PrintHelper();
Этот объект будет использоваться для того, чтобы вызвать метод print и вывести информацию об автомобиле.
Вызов метода printCarInfo():
Теперь для объекта car1 вызывается метод printCarInfo() с передачей объекта helper в качестве аргумента:
car1.printCarInfo(helper);
В этом методе происходит следующее:
void printCarInfo(PrintHelper helper) {
helper.print(this); // Передача текущего объекта 'car1' через 'this'
}

Ключевое слово this в данном контексте ссылается на текущий объект car1. То есть, через this мы передаем объект car1 в метод print класса PrintHelper. Метод helper.print(this) вызывает метод print объекта helper и передает туда ссылку на объект car1. Теперь в классе PrintHelper вызывается метод print() с объектом car1 в качестве аргумента. Аргументом метода print является объект класса Car, который был передан в качестве аргумента, в нашем случае — это car1. Далее вызывается метод car.displayInfo(). В этом вызове метод displayInfo() будет вызываться для объекта car1, который был передан через this. Метод displayInfo() выводит информацию об автомобиле.
4. Возврат текущего объекта из метода
Ключевое слово this также можно использовать для возвращения текущего объекта из метода. Это может быть полезно для реализации паттерна "цепочки вызовов" (method chaining).

```
class Car {
    String model;
    int year;

    Car setModel(String model) {
        this.model = model;
        return this;
    }

    Car setYear(int year) {
        this.year = year;
        return this;
    }

    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.setModel("Tesla").setYear(2023).displayInfo();
        // Res: Model: Tesla, Year: 2023
    }
}
```

В этом примере методы setModel и setYear возвращают текущий объект с помощью this. Это позволяет вызывать методы цепочкой:
car.setModel().setYear().displayInfo().
Модификаторы доступа
Доступ к члену определяется модификатором доступа, присоединенным к его объявлению. Язык Java предлагает богатый набор модификаторов доступа. Некоторые аспекты управления доступом в основном связаны с наследованием или пакетами (пакет по существу представляет собой группу классов). Модификаторами доступа Java являются pubic (открытый), private (закрытый) и protected (защищенный). В Java также определен стандартный уровень доступа. Модификатор доступа protected применяется, только когда задействовано наследование. Когда член класса изменяется с помощью pubic, доступ к нему может получать любой другой код. Когда член класса указан как private, доступ к нему могут получать только другие члены этого класса. Теперь понятно, почему объявлению метода main() всегда предшествовал модификатор public. Он вызывается кодом, находящимся вне программы, т.е. исполняющей средой Java. Если модификатор доступа не задействован, то по умолчанию член класса является открытым в своем пакете, но к нему нельзя получить доступ за пределами пакета.

5. Модификаторы final и static, метод main, аргумент переменной длины

Поле может быть объявлено как **final** (финальное), что предотвращает изменение его содержимого, делая его по существу константой. Это означает, что поле final должно быть инициализировано при его объявлении. Существуют два способа инициализации такого поля. Во-первых, полю final можно присвоить значение при его объявлении. Во-вторых, полю final можно присвоить значение в конструкторе. Ключевое слово final также может применяться к методам, но его смысл существенно отличается от того, когда оно применяется к переменным.
Временами вам понадобится определять член класса, который будет применяться независимо от любого объекта данного класса. Обычно доступ к члену класса должен осуществляться только в сочетании с объектом его класса. Однако можно создать член, который можно использовать сам по себе, без привязки к конкретному экземпляру.

Чтобы создать такой элемент, перед его объявлением следует указать ключевое слово **static** (статический). Когда член объявляется статическим, к нему можно получать доступ до того, как будут созданы какие-либо объекты его класса, и без ссылки на какой-либо объект. Объявить статическими можно как методы, так и переменные. Наиболее распространенным примером статического члена является метод main (), который объявлен как static, потому что он должен быть вызван до того, как будут созданы любые объекты. Переменные экземпляра, объявленные как static, по существу являются глобальными переменными. При объявлении объектов такого класса копия статической переменной не создается. Взамен все экземпляры класса имеют дело с одной и той же статической переменной.
С методами, объявленными как static, связано несколько ограничений:
Они могут напрямую вызывать только другие статические методы своего класса (поскольку статические методы не принадлежат конкретным объектам, они не могут напрямую обращаться к нестатическим методам или полям, но через объект это возможно).
Они могут напрямую получать доступ только к статическим переменным своего класса (поскольку статические методы не связаны с конкретным объектом, они не могут обращаться к нестатическим полям, так как в момент вызова статического метода может просто не существовать никакого объекта, и, соответственно, нестатические поля объекта не будут доступны).
Они никоим образом не могут ссылаться на this или super.

В языке Java метод main является точкой входа для выполнения программы. Он необходим, чтобы JVM знала, с какого места начинать выполнение кода. main — это метод, который запускается первым при выполнении программы. Он указывает JVM, с чего начать выполнение программы. Без метода main программа не может быть запущена, так как JVM не будет знать, с какого метода начинать.
Метод main имеет строго определённую сигнатуру, которая должна быть следующей:
public static void main(String[] args)
public — делает метод доступным для JVM из любого места (требуется для выполнения программы).
static — позволяет JVM вызывать метод без создания объекта класса. Это важно, так как программа начинается до того, как какие-либо объекты были созданы.
void — указывает, что метод не возвращает никакого значения.
String[] args — параметр, который позволяет передавать аргументы командной строки в программу.

	private	Без модификатора	protected	public
Тот же класс	Да	Да	Да	Да
Подкласс из того же пакета	Нет	Да	Да	Да
Не подкласс из того же пакета	Нет	Да	Да	Да
Подкласс из другого пакета	Нет	Нет	Да	Да
Не подкласс из другого пакета	Нет	Нет	Нет	Да

Аргумент переменной длины
В состав современных версий Java входит средство, упрощающее создание методов, которые должны принимать произвольное количество аргументов. Оно называется аргументами переменной длины (variable-length arguments - varargs). Метод, принимающий произвольное число аргументов, называется методом с переменной арностью или методом с аргументами переменной длины. Аргумент переменной длины определяется с помощью трех точек (. . .). Синтаксис . . . просто сообщает компилятору, что будет использоваться переменное число аргументов, причем аргументы будут храниться в массиве, на который ссылается аргумент. Наряду с параметром переменной длины метод может иметь и "обычные" параметры. Тем не менее, параметр переменной дл ины должен объявляться в методе последним. Существует еще одно ограничение, о котором следует помнить: должен быть только один параметр переменной длины.
Метод, принимающий аргумент переменной длины, можно перегружать двумя способами. Первый способ предусматривает применение отличающегося типа для параметра переменной длины: Test (int . . .) и Test (boolean . . .). . . . приводит к тому, что параметр интерпретируется в виде массива заданного типа. Следовательно, точно так же, как обычные методы можно перегружать с использованием отличающихся параметров типа массивов, методы с параметрами переменной длины разрешено перегружать, указывая разные типы для параметров переменной длины. В этом случае компилятор Java вызывает надлежащий перегруженный метод на основе отличия между типами. Второй способ перегрузки метода с аргументом переменной длины предполагает добавление одного или нескольких обычных параметров.

6. Внутренние классы и области видимости переменных

Класс можно определять внутри другого класса; такой класс из вестен как вложенный класс. Область действия вложенного класса ограничена областью действия его объемлющего класса. Таким образом, если класс B определен внутри класса A, то B не существует независимо от A. Вложенный класс имеет доступ к членам, в том числе закрытым, класса, в который он вложен. Тем не менее, объемлющий класс не имеет доступа к членам вложенного класса. Вложенный класс, объявленный непосредственно в области действия его объемлющего класса, будет членом объемлющего класса. Также можно объявлять вложенный класс, локальный для блока.
Существуют два типа вложенных классов: статические и нестатические. Статический вложенный класс - это класс, к которому применяется модификатор static. Поскольку класс статический, он должен обращаться к нестатическим членам объемлющего класса через объект. То есть статический вложенный класс не может напрямую ссылаться на нестатические члены объемлющего класса. Вторым типом вложенного класса является внутренний класс. Внутренний класс - это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может ссылаться на них напрямую так же, как поступают другие нестатические члены внешнего класса.

```
1  class Outer {
2      int outer x = 100;
3      void test() {
4          Inner.inner = new Inner();
5          inner.display();
6      }
7      class Inner {
8          void display() {
9              System.out.printls("display( ) : outer_x = " + outer_x);
10         }
11     }
12 }
13 class InnerClassDemo {
14     pubic static void main (String[] args) {
15         Outer outer = new Outer();
16         outer.test();
17     }
18 }
```

Вот вывод, генерируемый программой:
display( ): outer_x = 100
В программе внутренний класс по имени Inner определен в рамках области действия класса Outer, поэтому любой код класса Inner может напрямую обращаться к переменной outer_x. В классе Inner определен метод экземпляра display (), который отображает outer_x в стандартном потоке вывода. Метод main() объекта InnerClassDemo создает экземпляр класса Outer и вызывает его метод test(), который создает экземпляр класса Inner и вызывает метод display(). Важно понимать, что экземпляр Inner может быть создан только в контексте класса Outer.

Области видимости переменных в Java определяют, где можно использовать переменные и как долго они "живут" в программе. Эти области контролируют доступ к переменным и помогают избежать ошибок, связанных с именами переменных, их видимостью и временем жизни. Существует несколько основных областей видимости (или сфер видимости) переменных:
1. Локальные переменные (Local Variables)
Локальные переменные объявляются внутри методов, конструкторов или блоков кода (например, циклов или условных операторов). Они видимы только в пределах этого блока кода и недоступны за его пределами.
Где объявляются: внутри метода, конструктора или блока.
Где видимы: только внутри того метода, конструктора или блока, в котором они были объявлены.
Время жизни: с момента их объявления до завершения работы метода или блока, в котором они были созданы.
Инициализация: локальные переменные не инициализируются по умолчанию, их необходимо явно инициализировать перед использованием.
2. Переменные экземпляра (Instance Variables)
Переменные экземпляра — это нестатические поля класса, которые принадлежат каждому объекту этого класса. Они хранят данные, уникальные для каждого экземпляра класса.
Где объявляются: внутри класса, но вне любых методов, конструкторов или блоков.
Где видимы: видимы во всех методах и конструкторах этого класса (и могут быть доступны другим классам, если модификатор доступа позволяет).
Время жизни: живут с момента создания объекта до уничтожения этого объекта (статические переменные живут до завершения программы).
Инициализация: инициализируются по умолчанию значениями: 0 для чисел, false для булевых типов, null для ссылочных типов.
3. Параметры методов (Method Parameters)
Параметры методов — это переменные, которые передаются в метод при его вызове. Они действуют как локальные

переменные внутри метода.
Где объявляются: в объявлении метода (в круглых скобках после имени метода).
Где видимы: только внутри того метода, в котором они объявлены.
Время жизни: с момента вызова метода до завершения выполнения этого метода.
Инициализация: инициализируются значениями, которые передаются при вызове метода.
4. Переменные цикла (Loop Variables)
Переменные, объявленные в циклах (например, в цикле for), видимы только внутри тела цикла.

7. Наследование

В терминологии Java унаследованный класс называется суперклассом, а класс, выполняющий наследование - подклассом. Следовательно, подкласс представляет собой специализированную версию суперкласса. Он наследует все члены, определенные суперклассом, и добавляет собственные уникальные элементы. Чтобы наследовать класс, вы просто включаете определение одного класса в другой с применением ключевого слова extends. Для любого создаваемого подкласса разрешено указывать только один суперкласс. Однако ни один класс не может быть суперклассом для самого себя. Хотя подкласс включает в себя все члены своего суперкласса, он не может получить доступ к тем членам суперкласса, которые были объявлены как закрытые.
Основное преимущество наследования связано с тем, что после создания суперкласса, который определяет характерные черты, общие для набора объектов, его можно применять для создания любого количества более конкретных подклассов. Каждый подкласс может точно настраивать собственное предназначение. Скажем, следующий класс наследуется от Box и добавляет свойство цвета:

```
1 class ColorBox extends Box {
2     int color;
3     ColorBox(double w, double h, double d, int c) {
4         width = w;
5         height = h;
6         depth = d;
7         color = c;
8     }
9 }
```

Теперь можно вызывать Box, если цвет не нужен, а в противном случае - ColorBox.
Когда ссылочной переменной типа суперкласса присваивается ссылка на объект подкласса, то доступ имеется только к тем частям объекта, которые определены в суперклассе. Если подумать, то в этом есть смысл, потому что суперклассу ничего не известно о том, что к нему добавляет подкласс.
Будут возникать ситуации, когда желательно создавать суперкласс, который держит детали своей реализации при себе (т.е. хранит свои элементы данных закрытыми). В таком случае у подкласса не было бы возможности напрямую обращаться к этим переменным либо инициализировать их самостоятельно. Поскольку инкапсуляция является основным атрибутом ООП, совершенно не удивительно, что в Java предлагается решение описанной проблемы. Всякий раз, когда подклассу необходимо сослаться на свой непосредственный суперкласс, он может воспользоваться ключевым словом **super**. Ключевое слово super имеет две основные формы.
Первая вызывает конструктор суперкласса, а вторая служит для доступа к члену суперкласса, который был сокрыт членом подкласса. Чтобы увидеть, как используется super(), рассмотрим показанную ниже усовершенствованную версию класса BoxWeight:

```
1 class BoxWeight extends Box {
2     double weight;
3     BoxWeight (double w, double h, double d, double m) {
4         super(w, h, d);
5         weight = m;
6     }
7 }
```

Конструктор BoxWeight() вызывает super() с аргументами w, h и d, что приводит к вызову конструктора Box, который инициализирует поля width, height и depth с применением этих значений. Класс BoxWeight больше не инициализирует указанные поля самостоятельно.
Когда подкласс вызывает super(), он вызывает конструктор своего непосредственного суперкласса. Таким образом, super() всегда ссылается на суперкласс непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии. Кроме того, вызов super() всегда должен быть первым оператором, выполняемым внутри конструктора подкласса. super() всегда ссылается на конструктор в ближайшем суперклассе. В рамках иерархии классов, когда конструктору суперкласса требуются аргументы, то все подклассы должны передавать их "вверх по цепочке наследования".
Вторая форма ключевого слова super действует примерно так же, за исключением того, что всегда относится к суперклассу подкласса, в котором задействована. Вот как она выглядит:
super.член

Здесь член может быть либо методом, либо переменной экземпляра. Вторая форма super наиболее применима в ситуациях, когда имена членов подкласса скрывают члены с тем же именем в суперклассе.

```
class A {
    int i;
}
class B extends A {
    int i;
    B (int a, int b)
        super.i = a;
        i = b;
    void show() {
        System.out.println("i in superclass : " + super.i);
        System.out.println("i in sublass : " + i);
    }
}
class UseSuper {
    public static void main (String [] args) {
    B subOb = new B(1, 2);
    subOb.show();
    }
}
```

Хотя переменная экземпляра i в B скрывает i в A, ключевое слово super делает возможным доступ к члену i, определенному в суперклассе. Как вы увидите, super можно также использовать для вызова методов, сокрытых подклассом. В иерархии классов конструкторы завершают свое выполнение в порядке наследования от суперкласса к подклассу. Если хорошо подумать, то имеет смысл, что конструкторы завершают свое выполнение в порядке наследования. Поскольку суперклассу ничего не известно о каких-либо подклассах, любая инициализация, которую должен выполнить суперкласс, является отдельной и возможно обязательной для любой инициализации, выполняемой подклассом. Следовательно, она должна быть завершена первой. В иерархии классов, когда метод в подклассе имеет то же имя и сигнатуру типа, что и метод в его суперклассе, то говорят, что метод в подкласс переопределяет метод в суперклассе. При вызове переопределенного метода через его подкласс всегда будет вызываться версия метода, определенная в подклассе. Версия метода, определенная в супер классе, будет сокрыта. Метод переопределяется только в случае, если и мена и сигнатуры типов двух методов идентичны, а иначе два метода будут просто перегруженными.
Переопределение методов лежит в основе одной из самых мощных концепций Java - диспетчеризации динамических методов. Диспетчеризация динамических методов представляет собой механизм, с помощью которого вызов переопределенного метода распознается во время выполнения, а не на этапе компиляции. Динамическая диспетчеризация методов важна, потому что именно так в Java обеспечивается полиморфизм во время выполнения. Давайте начнем с повторения важного принципа: **ссылочная переменная типа суперкласса может ссылаться на объект подкласса**. Данный факт используется в Java для распознавания вызовов переопределенных методов во время выполнения. А каким образом? Когда переопределенный метод вызывается через ссылку на суперкласс, версия метода, подлежащая выполнению, выясняется на основе типа объекта, на который производится ссылка в момент вызова. Соответственно, такое выяснение происходит во время выполнения. При ссылке на разные типы объектов будут вызываться разные версии переопределенного метода. Другими словами, именно тип объекта, на который делается ссылка (а не тип ссылочной переменной), определяет, какая версия переопределенного метода будет выполняться. Таким образом, если суперкласс содержит метод, который переопределяется в подклассе, то при ссылке на разные типы объектов через ссылочную переменную типа суперкласса выполняются разные версии метода.

```
class A {
    void callme() {
    System.out.println("callme() inside A");
    }
    class B extends A {
        void callme() {
            System.out.println("callme() inside B");
        }
    }
    class C extends a {
        void callme() {
            System.out.println("callme() inside C");
        }
    }
}
class Dispatch {
    public static void main (String[] args) {
        A a = new A();
```

```
19          B b = new B();
20          C c = new C();
21          A r;
22          r = a;
23          r.callme();
24          r = b;
25          r.callme();
26          r = c;
27          r.callme()
28      }
29 /*Result:
30 callme () inside A
31 callme () inside B
32 callme () inside C*/
33 }
```

Наряду с тем, что переопределение методов является одной из самых мощных функциональных средств Java, иногда его желательно предотвращать. Чтобы запретить переопределение метода, в начале его объявления понадобится указать ключевое **final** в качестве модификатора. Методы, объявленные как final, не могут быть переопределены. Иногда нужно предотвратить наследование класса. Для этого перед объявлением класса укажите ключевое слово final. Объявление класса как final также неявно объявляет все его методы как final.

Аннотация @Override в Java используется для указания того, что метод в подклассе (наследнике) переопределяет метод суперкласса (родительского класса). Эта аннотация не является обязательной, но ее использование имеет несколько важных преимуществ.
Проверка компилятором: аннотация @Override заставляет компилятор проверять, действительно ли метод переопределяет метод суперкласса. Если вы допустите ошибку, например, неверно напишете имя метода или его сигнатуру (например, параметры), компилятор выдаст ошибку. Без этой аннотации ошибка могла бы остаться незамеченной, и метод в реальности не переопределился бы, а стал бы новым методом в подклассе.
Повышение читаемости кода: когда другой разработчик (или вы сами через какое-то время) читаете код, аннотация @Override помогает сразу понять, что данный метод переопределяет поведение метода из родительского класса. Это делает код более понятным.
Предотвращение ошибок: в случае, если суперкласс изменил имя метода или его сигнатуру, но вы забыли обновить соответствующий метод в подклассе, аннотация @Override снова вызовет ошибку на этапе компиляции, помогая избежать потенциальных багов.

Бывают ситуации, когда желател ьно определить суперкласс, который объявляет структуру заданной абстракции, не предоставляя полные реализации методов. То есть иногда нужно создать суперкласс, определяющий только обобщенную форму, которая будет применяться всеми его подклассами, оставляя каждому подклассу возможность заполнить детали. Такой класс определяет природу методов, подлежащих реализации в подклассах. Ситуация подобного рода может возникнуть, когда суперкласс не способен создать осмысленную реализацию метода. Вы можете потребовать, чтобы некоторые методы были переопределены в подклассах, указав модификатор abstract. Иногда их называют методами, подпадающими под ответственность подкласса, потому что они не имеют реализаации, указанной в суперклассе. Таким образом, подкласс обязан их переопределить - он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода применяется следующая общая форма:
abstract тип имя(список-параметров);
Любой класс, содержащий один или несколько абстрактных методов, тоже должен быть объявлен абстрактным. Чтобы объявить класс абстрактным, перед ключевым словом class в начале объявления класса просто используется ключевое слово abstract. Объектов абстрактного класса не бывает, т.е. экземпляр абстрактного класса нельзя создать напрямую с помощью операции new. Кроме того, не допускается объявлять абстрактные конструкторы или абстрактные статические методы.

8. Пакеты

Пакеты представляют собой контейнеры для классов. Они используются для отделения пространства имен класса. Например, создав класс по имени List и сохранив его в собственном пакете, можно не беспокоиться о том, что он будет конфликтовать с другим классом по имени List, который находится где-то в другом месте. Пакеты хранятся в иерархическом порядке и явно импортируются в определения новых классов. Создать пакет довольно легко: понадобится просто поместить в начало файла с исходным кодом Java оператор package. Любые классы, объявленные в данном файле, будут принадлежать указанному пакету. Оператор package определяет пространство имен, в котором хранятся классы. Если оператор package отсутствует, тогда имена классов помещаются в стандартный пакет, не имеющий имени. Общая форма оператора многоуровневого пакета выглядит следующим образом:
package пакетl [. пакет2 [. пакет]] J;
Иерархия пакетов должна быть отражена в файловой системе на машине для разработки приложений Java. Например, объявленный ниже пакет должен храниться в папке a\b\c в среде Windows: package a.b.c ;