ППОИС – ОТВЕТЫ НА ВОПРОСЫ

***Мини-словарик терминов:***

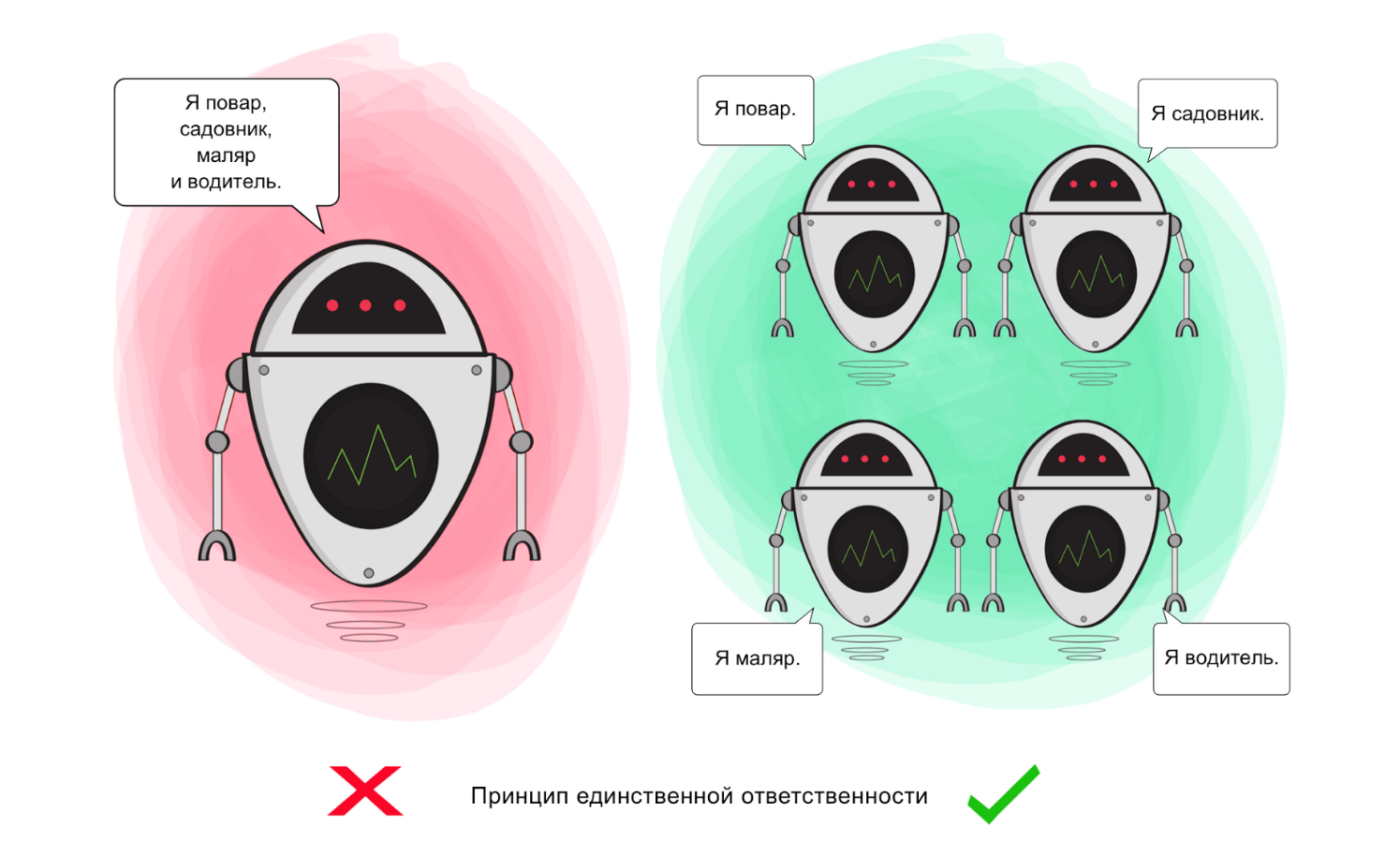
1. **Поведение (behavior)** – это перечень методов объекта и результатов их вызова. Другими словами – это задачи, которые выполняет объект.
2. **Модификация** – это процесс изменения существующего класса (его методов) с целью добавить новые свойства или поведения через изменение функционирования уже имеющихся элементов (или улучшение структуры кода).
3. **Расширение** – это процесс изменения существующего класса с целью добавить новые свойства или поведения без изменения функционирования и структуры уже имеющихся элементов. К тому, что уже было изначально, добавляется новое, не меняя старого (новые методы).
4. **God object** – “Божественный объект” – это антипаттерн в ООП, описывающий объект, который хранит в себе “слишком много” или делает “слишком много”.
5. **Операция** – это конструкция в языках программирования, аналогичная по записи математическим операциям, то есть специальный способ записи некоторых действий. (Действие, которое будет выполнено для данной задачи).
6. **Функция** – это подпрограмма (блок кода), выполняющая какие-либо операции и возвращающая значение.
7. **Процедура** – это подпрограмма (блок кода), которая только выполняет операции, без возврата значения.
8. **Интерфейс** – это абстрактный класс, содержащий чисто виртуальные функции (поведения). Они предоставляют абстрактный набор методов, которые классы должны реализовать.
9. **Абстракция** – использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе, без описания их конкретных/детальных реализаций (Выделение основных ключевых черт/характеристик объекта или группы объектов). Основная идея состоит в том, чтобы представить объект, обладающий набором методов, и при этом не предоставить конкретную логику этих методов (описать структуру).
10. **Парадигма** – это совокупность методов, концепций, принципов, техник и инструментов, которые определяют способ организации программы на языке программирования и ход её выполнения.
11. **Процедурное программирование**— это подход к написанию компьютерных программ, основанный на использовании процедур, функций и подпрограмм. Оно относится к структурному программированию и описывает программы в виде последовательности операций, которые выполняются по порядку.
12. В терминах объектно-ориентированного программирования **раннее связывание** означает, что объект и вызов функции связываются между собой на этапе компиляции. Это означает, что вся необходимая информация для того, чтобы определить, какая именно функция будет вызвана, известна на этапе компиляции программы. В качестве примеров раннего связывания можно указать стандартные вызовы функций, вызовы перегруженных функций и перегруженных операторов. Принципиальным достоинством раннего связывания является его эффективность — оно более быстрое и обычно требует меньше памяти, чем позднее связывание. Его недостатком служит невысокая гибкость.
13. **Позднее связывание** означает, что объект связывается с вызовом функции только во время исполнения программы, а не раньше. Позднее связывание достигается в С++ с помощью использования виртуальных функций и производных классов. Его достоинством является высокая гибкость. Оно может использоваться для поддержки общего интерфейса, позволяя при этом различным объектам иметь свою собственную реализацию этого интерфейса. Более того, оно помогает создавать библиотеки классов, допускающие повторное использование и расширение.
14. Метод **super()** используется в объектно-ориентированных языках программирования, таких как Java и C++, для вызова конструктора в родительском классе (также известном как суперкласс или базовый класс). Он используется для выполнения шагов инициализации, определенных в конструкторе родительского класса, до выполнения шагов инициализации конструктора текущего класса.
15. **Cohesion (сцепление/единство/сплочённость)** – это мера силы взаимосвязей элементов внутри модуля. Степень, в которой задачи, выполняемые некоторым программным модулем, связаны друг с другом. Связан с SRP из SOLID. (Это степень, в которой все элементы, направленные на выполнение одной задачи, содержатся в компоненте. По сути, сцепление – это внутренний клей, который удерживает модуль вместе. Хороший дизайн программного обеспечения будет иметь высокое сцепление.)
16. **Coupling (связность)** – это степень, в которой один блок независим от других. Невозможно добиться полного разделения (decoupling) без нарушения целостности (cohesion) и наоборот. (Связь является мерой степени взаимозависимости между модулями. Хорошее программное обеспечение будет иметь низкую связность.)

**ВОПРОСЫ И ОТВЕТЫ**

1. **SOLID**

***SOLID*** – аббревиатура пяти основных принципов проектирования в объектно- ориетированном программировании. Другими словами SOLID – это принципы разработки программного обеспечения, следуя которым Вы получите эффективный код, который в дальнейшем будет легко масштабироваться и поддерживаться в рабочем состоянии (гибкая и удобная архитектура; чистый, понятный и простой код).

Теперь разберём каждую букву и то, что она означает:

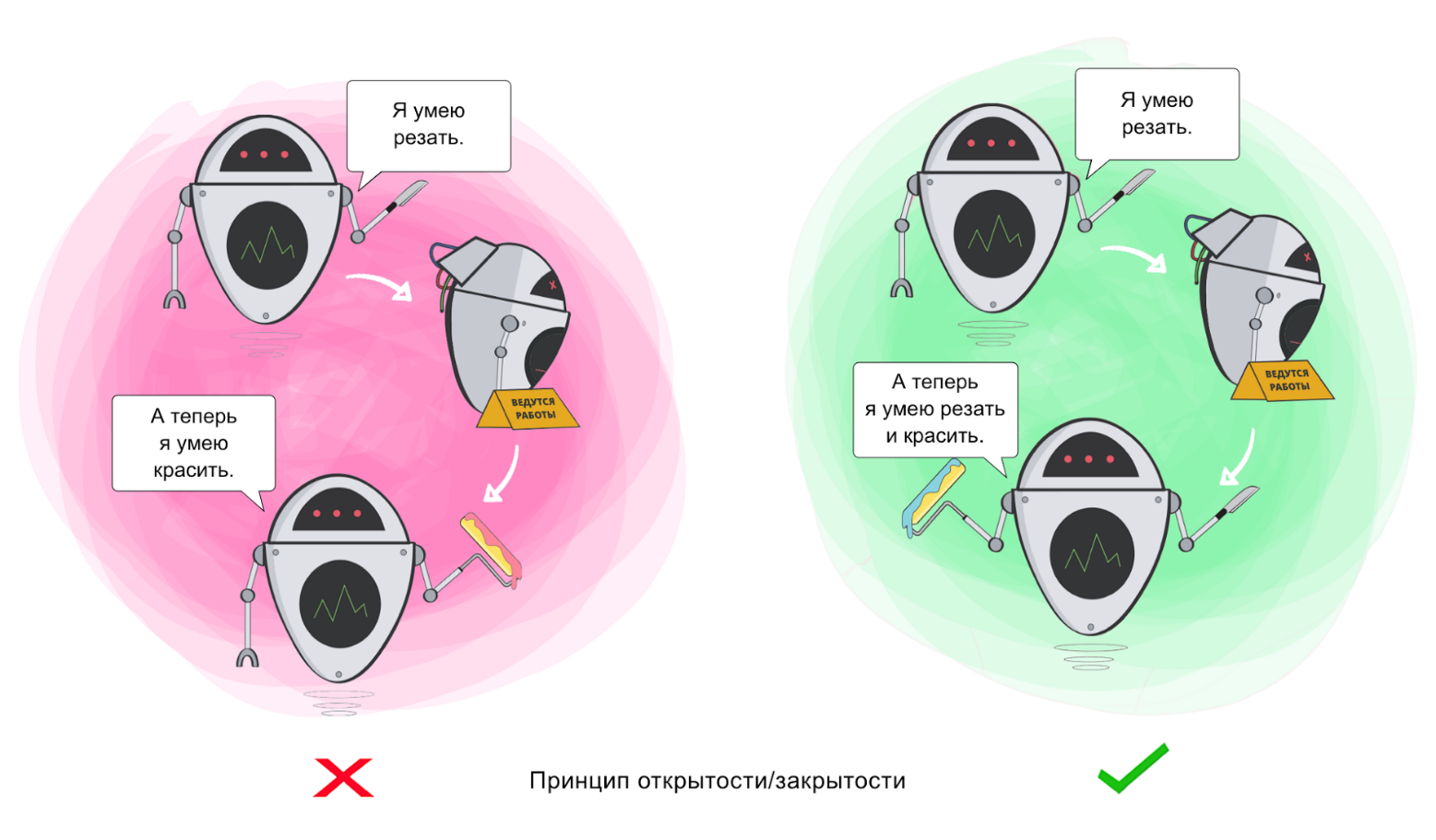
**S** – **Single Responsibility Principle** – **SRP** (Принцип единственной ответственности). Каждый класс должен отвечать за одну операцию (иметь только одну зону ответственности).

Если класс отвечает за несколько операций сразу, вероятность возникновения багов возрастает – внося изменения, касающиеся одной из операций, Вы, сами того не подозревая, можете затронуть и другие.

Также сами классы могут стать чрезмерно большими и громоздкими, а читаемость кода снизиться, так как придётся долго разбираться, для чего служат те или иные части класса. (Возникает риск создания *god object*)

Принцип служит для разделения типов поведения, благодаря которому ошибки, вызванные модификациями в одном поведении (методе), не распространялись на прочие, не связанные с ним типы.

**O** – **Open - Closed Principle** – **OCP** (Принцип открытости - закрытости). Классы должны быть открыты для расширения, но закрыты для модификации (изменения).



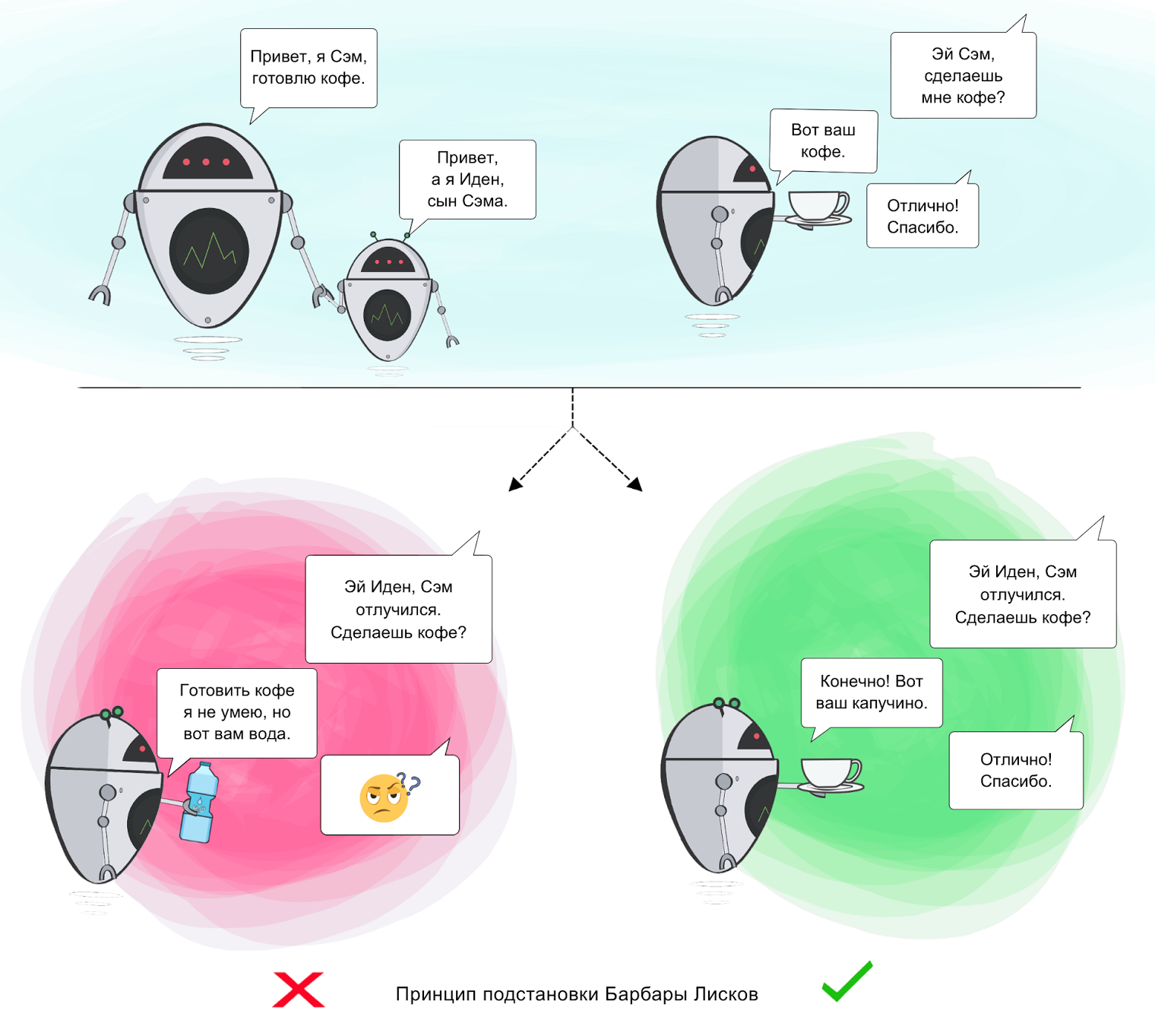
Когда вы меняете текущее поведение класса, эти изменения сказываются на всех системах, работающих с данным классом. Если хотите, чтобы класс выполнял больше операций, то идеальный вариант – не заменять старые на новые, а добавлять новые к уже существующим.

Также, изменяя уже созданные и проверенные операции, теряется гарантия их работоспособности и пригодности. Это приведёт к повторному переписыванию, проверкам, тестированию, на что придётся затратить дополнительное время.

Принцип служит для того, чтобы делать поведение класса более разнообразным, не вмешиваясь в текущие операции, которые он выполняет. Благодаря этому Вы избегаете ошибок в тех фрагментах кода, где задействован этот класс.

**L** – **Liskov Substitution Principle** – **LSP** (Принцип подстановки Барбары Лисков). “Если П является подтипом Т, то любые объекты типа T, присутствующие в программе, должны, могут заменяться объектами типа П без негативных последствий для функциональности программы” (+ \_ +).

А теперь на чуть более человеческом: должна быть возможность вместо базового типа (родительского класса) подставить любой его подтип (класс-наследник), и при этом работа программы не должна измениться.



В случаях, когда класс-потомок не способен выполнять те же действия, что и класс-родитель, возникает риск появления ошибок. Как правило, дочерний класс должен *расширять* родительский класс.

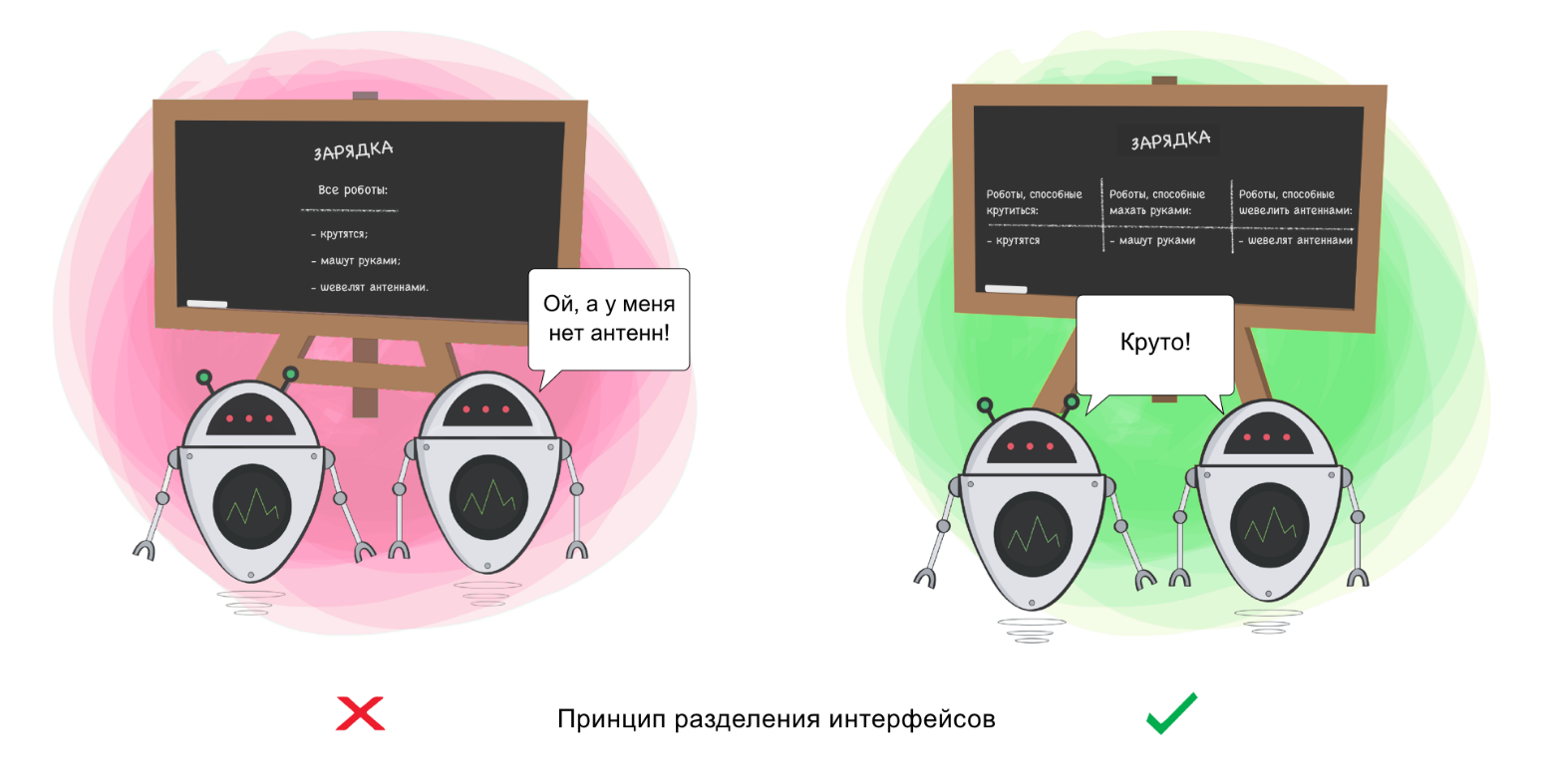
Если у Вас имеется класс и Вы создаете на его базе другой класс, исходный класс становится родителем, а новый – его потомком. Класс-потомок должен производить такие же операции, как и класс-родитель. Это называется ***наследственностью***.

Необходимо, чтобы класс-потомок был способен обрабатывать те же запросы, что и родитель, и выдавать тот же результат. Или же результат может отличаться, но при этом относиться к тому же типу. На картинке это показано так: класс-родитель подаёт кофе (в любых видах), значит, для класса-потомка приемлемо подавать капучино (разновидность кофе), но неприемлемо подавать воду

Принцип служит для того, чтобы обеспечить постоянство: класс-родитель и класс-потомок могут использоваться одинаковым образом без нарушения работы программы.

(Но! Потомок не может быть заменяем родителем (только в случае, если их используемые и необходимые методы совпадают)).

**I** – **Interface Segregation Principle** – **ISP** (Принцип разделения интерфейсов). Не следует ставить клиент (класс) в зависимость от методов, которые он не использует.

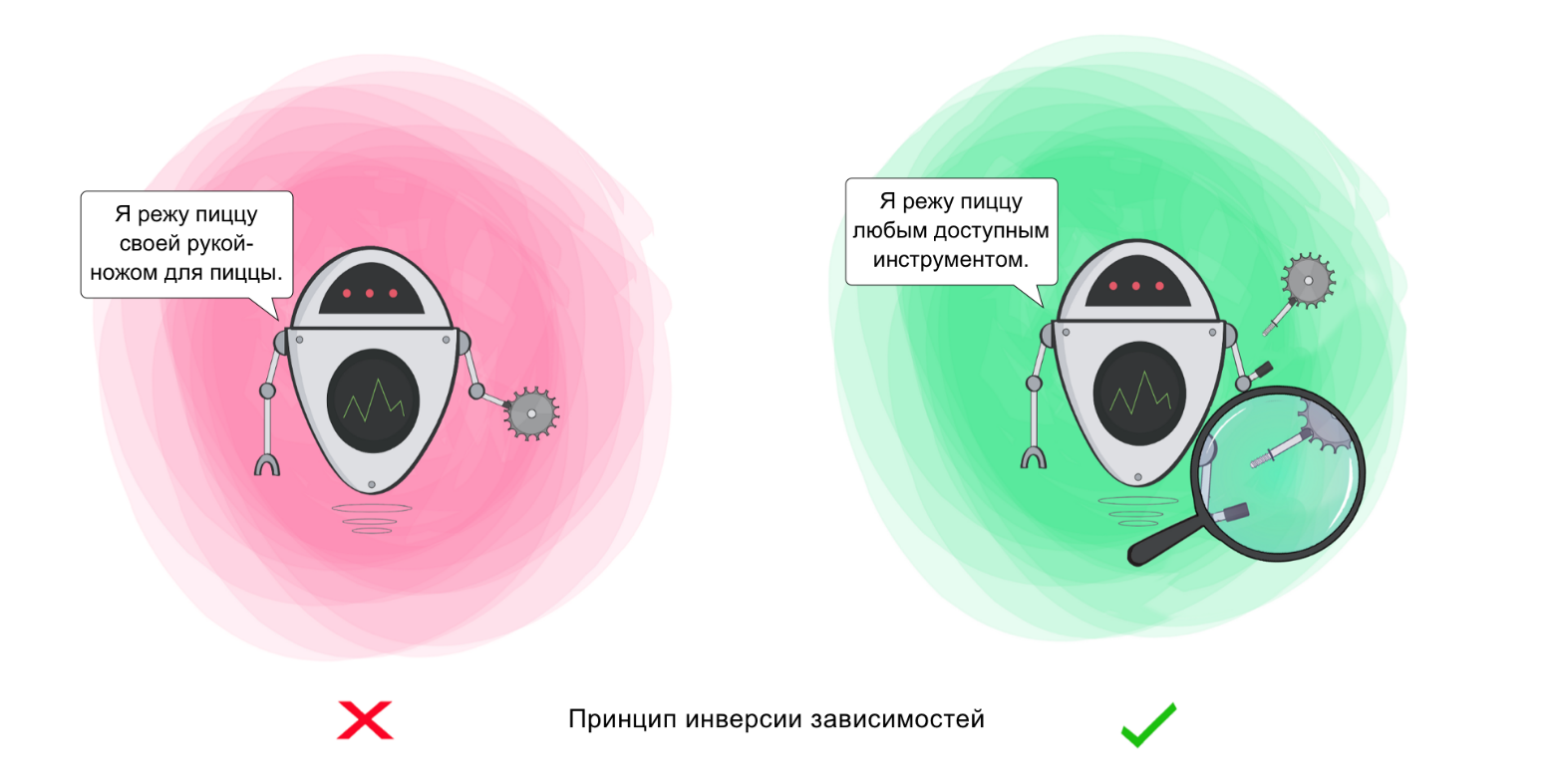


Когда классу приходится производить действия, не несущие никакой реальной пользы, это выливается в пустую трату ресурса, а в случае, если класс выполнять эти действия не способен, ведёт к возникновению багов.

Пример: пусть у нас есть интерфейс Weapon, в котором есть методы Attack() и Reload(). От интерфейса поведения наследуют классы Shotgun и Minigun. Вдруг появляется класс Knife(), который тоже наследует поведения от Weapon. Но из-за этого, в классе Knife появляется метод Reload(), который тот обязан реализовать (хотя он его не будет использовать и он ему не нужен) - чётко наблюдается нарушение **ISP**. Для разрешения этой ошибки в конкретно данном примере следовало бы разбить интерфейс Weapon на два: для оружия с перезарядкой и без неё или в целом по типу оружия.

Класс должен производить только те операции, которые необходимы для осуществления его функций. Все другие действия следует либо удалить совсем, либо переместить, если есть вероятность, что они понадобятся другому классу в будущем.

Принцип служит для того, чтобы раздробить единый набор действий на ряд наборов поменьше – таким образом, каждый класс делает то, что от него действительно требуется, и ничего больше.

**D – Dependency Inversion Principle – DIP** (Принцип инверсии зависимостей). Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Согласно данному принципу, класс не должен соединяться с инструментом, который применяет для выполнения операции. Вместо этого он должен быть соединён с интерфейсом, который поможет установить связь между инструментом и классом.

Кроме того, принцип гласит, что ни интерфейс, ни класс, не обязаны вникать в специфику работы инструмента. Напротив, это инструмент должен подходить под требования интерфейса.

Этот принцип служит для того, чтобы устранить зависимость классов верхнего уровня от классов нижнего уровня за счёт введения интерфейсов.

**2. YAGNI, DRY, KISS, BDUF, APO**

**YAGNI / You Aren't Gonna Need It / Вам это не понадобится**

* Если пишете код, то будьте уверены, что он вам понадобится. Не пишите код, если думаете, что он пригодится позже.

**DRY / Don't Repeat Yourself / Не повторяйтесь**

* Следует избегать дублирования кода, ради экономии времени и ресурсов.

**KISS / Keep It Simple, Stupid / Будь проще 😊**

* Не придумывайте к задаче более сложного решения, чем ей требуется.

**BDUF / Big Design Up Front / Глобальное проектирование прежде всего**

* Подход к разработке ПО, при котором проектирование программы завершается и совершенствуется до начала её реализации.

**APО / Avoid Premature Optimization / Избегайте преждевременной оптимизации**

* Не следует тратить время на оптимизацию кода на ранних этапах разработки, пока не станет ясно.

**3. Что такое объектно-ориентированное программирование (ООП)?**

**Объектно-ориентированное программирование (ООП)** — это подход (парадигма программирования), при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение. Если постараться объяснить простыми словами, то ООП ускоряет написание кода и делает его более читаемым.

Идеология объектно-ориентированного программирования (ООП) разрабатывалась, чтобы связать поведение определенного объекта с его классом. Людям проще воспринимать окружающий мир как объекты, которые поддаются определенной классификации (например, разделение на живую и неживую природу). В ООП выделяют четыре основных элемента: класс, объекты, методы и атрибуты (свойства).

**4. Каковы основные принципы ООП?**

*(Без упоминания абстракции (абстрагирования – см. вопрос 10), с краткой характеристикой)*

Объектно-ориентированный подход к программированию строится на трёх основных принципах:

* **Наследование** – концепция ООП, согласно которой одни классы (производные) могут наследовать свойства и поведения от других классов (базовых / родительских), а также расширять их;
* **Инкапсуляция** – механизм, позволяющий объединить данные и методы, работающие с этими данными в единый объект и скрыть детали реализации от пользователя;
* **Полиморфизм** – механизм ООП, который позволяет разным сущностям выполнять одни и те же действия. При этом неважно, как эти сущности устроены внутри и чем они различаются.

**5. Объясните концепцию инкапсуляции в ООП.**

**Инкапсуляция** – механизм ООП, позволяющий объединить данные и методы, работающие с этими данными в единый объект и скрыть детали реализации от пользователя. Доступ к объектам возможен через специальные открытые методы (или интерфейсы), а напрямую обратиться к их содержимому нельзя.

Преимущества принципа инкапсуляции:

– Позволяет защитить данные объекта от некорректного доступа и изменения извне.

– Сокрытие реализации.

– Упрощение интерфейса.

– Способствует соблюдению принципа единственной ответственности (SOP).

– Позволяет установить контроль доступа к данным.

Применение процесса инкапсуляции в практике программирования помогает создавать более структурированный, безопасный и расширяемый код. Она способствует лучшему управлению сложностью программы и облегчает сотрудничество между разработчиками при разработке больших проектов.

**6. Как наследование способствует повторному использованию кода в ООП?**

Наследование позволяет создавать иерархии классов, где общая функциональность реализуется в базовом (родительском) классе, и все подклассы автоматически наследуют этот код. Это способствует повторному использованию кода и соблюдения правила **DRY (Don’t Repeat Yourself)** и принципов **SOLID**, что уменьшает дублирование и облегчает его поддержку, и в совокупности способствует созданию качественного и эффективного кода.

**7. Какова цель полиморфизма в ООП?**

Полиморфизм в контексте ООП означает, что разные объекты могут реагировать на один и тот же запрос, проявляя разное поведение в зависимости от своего типа. Это позволяет сократить дублирование кода, улучшить читаемость и облегчить расширение программы.

**Цель**: позволить выполнять разным сущностям выполнять одни и те же действия, причём неважно, как эти сущности устроены внутри и чем они различаются. За счёт этого повышается гибкость и читаемость кода, становится легче внедрять новые классы и создавать более модульные приложения, а также создавать общие интерфейсы, которые могут быть реализованы различными классами, что способствует соблюдению принципов инкапсуляции и абстракции.

**8. Опишите разницу между классом и объектом.**

**Класс** – является планом / шаблоном некоторого объекта, содержащий описание его свойств, полей и методов (структура). (+ Второе определение, которое можно встретить: пользовательский тип данных, который может содержать другие типы).

**Объект** – конкретное воплощение класса (шаблона); конкретный экземпляр класса. Объекты создаются с использованием некоторых определённых данных, которыми инициализируются его поля (поля объекта класса инициализированы некоторыми значениями). Обычно, объекты используются для представления сущностей реального мира. Методы класса, как правило, вызываются относительно какого-либо объекта.

**9. Что такое конструктор в ООП и какова его роль.**

**Конструкторы** представляют специальную функцию (метод), которая имеет то же имя, что и класс, которая не возвращает никакого значения и которая позволяет инициализировать объект класса во время его создания и таким образом гарантировать, что поля класса будут иметь определённые значения. При каждом создании нового объекта класса вызывается конструктор класса. Без вызова конструктора невозможно создать экземпляр класса.

(+ Выделяют следующие виды конструкторов: с параметрами, по умолчанию, именованный, копирования, преобразования, перемещения, виртуальный).

**10. Объясните концепцию абстракции в ООП.**

**Абстрагирование** – выделение общих характеристик объектов, их свойств и методов, при игнорировании деталей реализации (иногда упоминается в качестве одного из принципов ООП).

**Абстракция** – использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе, без описания их конкретных/детальных реализаций (Выделение основных ключевых черт/характеристик объекта или группы объектов). Основная идея состоит в том, чтобы представить объект, обладающий набором методов, и при этом не предоставить конкретную логику этих методов (описать структуру).

**Абстракция** позволяет разрабатывать программы на различных языках программирования, скрывая сложность и детали нижележащего кода. Это делается для упрощения сложных систем и концепций, чтобы разработчики могли фокусироваться на основных аспектах проблемы и легче понимали код.

В объектно-ориентированном программировании абстракция играет важную роль. Она позволяет создавать абстрактные классы и интерфейсы, которые определяют общие свойства и методы, не зависящие от конкретной реализации. Преимущества абстракции ООП включают:

- Упрощение сложности;

- Модульность;

- Повышение безопасности.

**11. Как классы и объекты связаны с объектами реального мира?**

ООП основывается на концепции классов и объектов, которые тесно связаны с реальными объектами и понятиями. Проявление этой связи:

1. Классы как абстракция реальных объектов.

Класс можно рассматривать как шаблон или абстракцию, представляющую групповую сущность (группу объектов) реального мира. Например, класс “Автомобиль” может описывать все автомобили, включая их характеристики и поведение. Вместо того, чтобы описывать каждый автомобиль отдельно, мы создаём общий класс, который содержит общие для всех автомобилей свойства и методы.

1. Объекты как конкретные экземпляры.

Объекты – это конкретные экземпляры классов, которые моделируют реальные объекты. Объекты позволяют создавать уникальные реализации абстракций, что помогает в управлении сложностью.

1. Связь через свойства и методы.

Свойства класса соответствуют характеристикам реальных объектов (+ не забываем об абстрагировании и абстракциях). Методы представляют действия или поведения этих объектов. Это соответствует тому, как объекты взаимодействуют с окружающим миром.

1. Наследование и полиморфизм как реализация вариативности.

Наследование позволяет создавать новые классы на основе существующих, что отражает иерархию в реальном мире. Полиморфизм позволяет одному и тому же методу вести себя по-разному в зависимости от типа объекта, с которым он взаимодействует. Это отражает разнообразие объектов в реальном мире, которые могут реагировать по-своему на одни и те же действия.

1. Инкапсуляция как защита данных.

Это аналогично тому, как в реальном мире мы не всегда имеем доступ к механизмам объектов, а взаимодействуем с ними только через определённые интерфейсы.

**12. Каковы преимущества использования ООП перед процедурным программированием?**

ООП имеет ряд преимуществ перед процедурным программированием:

* **Инкапсуляция**: объекты в ООП скрывают свои детали реализации от других объектов, что уменьшает сложность кода и делает его более понятным. Это также обеспечивает более легкое тестирование и модификацию кода.
* **Наследование**: наследование позволяет создавать новые классы, которые могут наследовать свойства и методы от родительских классов. Это позволяет избежать дублирования кода и уменьшить количество ошибок при изменении кода.
* **Полиморфизм**: полиморфизм позволяет использовать один и тот же интерфейс для работы с разными типами объектов. Это увеличивает гибкость кода и позволяет повторно использовать уже написанный код.
* **Безопасность**: ООП позволяет контролировать доступ к свойствам и методам объекта. Таким образом, возможность ошибки в программе сокращается, а ее безопасность увеличивается.
* **Модульность**: ООП позволяет разбить программу на модули (объекты и классы), каждый из которых может быть независимо разработан и протестирован. Это позволяет повысить эффективность разработки и сопровождения программного обеспечения.

В целом, ООП предоставляет ряд методов и инструментов для создания более гибких, масштабируемых и безопасных приложений. Однако, в зависимости от конкретной задачи, процедурное программирование также может быть достаточным и эффективным способом разработки.

**13. Определите класс и приведите пример.**

Надеюсь, ни у кого не возникнет сложности с тем, чтобы определить и привести пример любого класса… Возьмите там человека, машину, ну или яблоко например… Ваше задание из лабораторных…

**14. Что такое объект и чем он отличается от класса?**

Это снова повтор вопроса… (8. Опишите разницу между классом и объектом).

**15. Как создать объект из класса в ООП?**

Для начала объявим и опишем некоторый класс интересующего нас объекта (и не забудем прописать конструктор класса). А затем у нас есть два способа создания объекта: на стеке (статическое создание объекта) или в куче (динамическое создание объекта).

Рассмотрим на примере. Пусть у нас есть класс “Книга”. Тогда:

1. Создание на стеке:

int main() {

Книга мояКнига("1984", "Джордж Оруэлл");

// Объект создается на стеке

мояКнига.показатьИнформацию(); // Вызов метода

return 0;

}

1. Создание в куче (для динамического создания объекта используется оператор “new”):

int main() {

Книга\* мояКнига = new Книга("Война и мир", "Лев Толстой");

// Объект создается в куче

мояКнига->показатьИнформацию(); // Вызов метода через указатель

delete мояКнига; // Не забываем освобождать память, когда объект больше не нужен

return 0;

}

**16. Указатель this.**

Ключевое слово this представляет собой неявно определенный указатель на сам объект. С его помощью метод класса определяет, с данными какого объекта ему предстоит работать. Каждый метод класса неявно содержит в качестве поля данных указатель.

При компиляции обычного метода, компилятор неявно добавляет к нему параметр \*this. **Указатель \*this** — это скрытый **константный указатель**, содержащий адрес объекта, который вызывает метод класса (относительно которого вызывается метод класса).

**17. Как ООП помогает моделировать объекты реального мира с помощью классов?**

По сути на этот вопрос уже был дан ответ ранее… (11. Как классы и объекты связаны с объектами реального мира).

**18. Каково назначение ключевого слова “new” при создании объекта?**

Ключевое слово **“new”** используется для динамического выделения памяти в куче (heap) во время выполнения программы.

При помощи оператора **new** вы можете создавать объекты, когда размер данных или количество создаваемых объектов неизвестны во время компиляции. Также это важно для работы со структурами данных (такими как списки, деревья и т.д.), размеры которых могут изменяться в процессе выполнения программы.

Объекты, созданные при помощи **new**, будут существовать до тех пор, пока Вы сами не освободите их, используя оператор **delete**. Это позволяет гибко управлять временем жизни объектов, что важно в больших и сложных приложениях.

Объекты, созданные на стеке (например, обычные локальные переменные), имеют ограниченный размер, и, если они становятся слишком большими, это может привести к переполнению стека. Использование **new** позволяет избежать этой проблемы, выделяя память в куче.

Главное следить за возможными утечками памяти и вовремя очищать память. 😉

**19. Как вы определяете и используете статические методы в ООП?**

Статические методы в объектно-ориентированном программировании (ООП) — это функции, которые принадлежат классу, а не конкретному объекту (экземпляру) этого класса, то есть являются общими для всех объектов класса. Они определяются с использованием ключевого слова **static** и могут быть вызваны без создания экземпляра класса. Как правило статические методы предназначены и используются для работы со статическими полями.

Особенности статических методов:

* Статические методы не имеют доступа к указателю this, так как они не связаны с конкретным объектом класса.
* Статические методы могут обращаться только к статическим членам класса. Для обращения к нестатическим членам необходимо использовать указатель на объект.
* Статические методы могут вызываться непосредственно через имя класса, что позволяет избежать необходимости создавать экземпляр, если это не требуется.

**Static** поля и методы могут использоваться для различных целей. Остаётся только придумать пример (можно даже из реальной жизни))).

Например, счётчики объектов, идентификация значениями, static id generator, вывод общей графики, для утилитарных функций, которые не зависят от состояния экземпляра, когда требуется общая функциональность, не требующая создания объекта и т.д.

**20. Что такое конструктор и почему он важен в ООП?**

Это же повторение вопроса… (9. Что такое конструктор в ООП и какова его роль?).

**21. Объясните разницу между полями / методами объекта и класса.**

Разница между полями / методами объекта и класса заключается в их контексте, цели использования и области действия.

* Поля объекта и класса.

Поля класса (статические поля (*остальные поля класса являются частью шаблона для создания объекта и инициализируются при создании объекта для соответственного объекта*)) это переменные, которые принадлежат самому классу, а не конкретному объекту.

- Они могут быть доступны без создания экземпляра класса. Для доступа можно использовать имя класса.

- Поля класса обычно используются для хранения информации, которая должна быть общей для всех объектов этого класса.

Поля объекта (не статические поля):

- Это переменные, которые принадлежат экземпляру класса, то есть конкретному объекту.

- Каждое изменение поля объекта затрагивает только этот экземпляр, а не другие.

- Поля объекта используются для хранения состояния данного объекта.

* Методы объекта и класса.

Методы класса (статические методы).

- Это функции, которые также принадлежат классу и могут быть вызваны без создания экземпляра класса.

- Они могут оперировать только статическими полями и не имеют доступа к полям объекта.

- Используются, например, для создания вспомогательных функций, которые не зависят от состояния отдельных объектов.

Методы объекта (не статические методы).

- Это функции, которые зависят от конкретного экземпляра класса. Они могут обращаться к полям объекта и изменять их состояние.

- Методы объекта обычно используются для выполнения операций, связанных с конкретным состоянием объекта.

Основная разница между полями и методами объекта и класса заключается в том, что поля и методы класса являются статическими и принадлежат самому классу, а поля и методы объекта относятся к конкретному экземпляру класса. Это определяет их область действия и то, как они могут использоваться в коде.

**22. Как ООП поддерживает концепцию модульности и организации кода?**

Объектно-ориентированное программирование (ООП) поддерживает концепцию модульности и организации кода несколькими способами. Основными из них являются:

-- Инкапсуляция

Инкапсуляция позволяет скрыть внутреннюю реализацию объекта от внешнего мира. Это означает, что поля (состояние) объекта защищены от прямого доступа и могут изменяться только через методы класса (интерфейс). (+ Объединение данных и методов в единый объект).

Преимущества:

- Упрощает изменение внутренней реализации без риска поломать код, который использует этот объект.

- Уменьшает зависимость между различными частями кода, что способствует поддерживаемости.

-- Наследование

Наследование позволяет создавать новые классы на основе существующих, унаследовав их поля и методы. Это упрощает повторное использование кода и помогает организовывать иерархии объектов.

Преимущества:

- Способствует созданию более абстрактных и обобщенных классов.

- Позволяет избежать дублирования кода, так как общая логика может быть реализована в базовом классе.

-- Полиморфизм

Полиморфизм позволяет объектам разных классов обрабатывать данные по единому интерфейсу. Это достигается через использование методов с одинаковым именем, но с различной реализацией.

Преимущества:

- Упрощает код, предоставляя единый интерфейс для взаимодействия с объектами разных классов.

- Способы взаимодействия могут определяться во время выполнения, что увеличивает гибкость системы.

-- Абстракция

Абстракция позволяет сосредоточиться на важной информации, скрывая менее значимые детали. Это осуществляется через создание абстрактных классов и интерфейсов, которые задают общие характеристики и поведение класса.

Преимущества:

- Упрощает понимание сложных систем, позволяя сосредоточиться на высоком уровне.

- Способствует созданию модулей, которые могут быть легко адаптированы или заменены.

-- Организация кода в классы и модули

В ООП программа разбивается на объекты, которые инкапсулируют данные и поведение. Каждый класс представляет собой отдельный модуль со своей логикой и зависит только от своих интерфейсов.

Преимущества:

- Легкость в управлении кодом: классы могут разрабатываться и тестироваться независимо.

- Улучшает читаемость и упрощает поиск ошибок, так как функциональность сгруппирована по смыслу.

(+ есть способ скрыть реализацию / детали, модель чёрного ящика -> пространства имён, статические и динамические библиотеки, разделение на .h и .cpp файлы)

Таким образом, ООП поддерживает концепцию модульности и организации кода с помощью различных механизмов, которые ограничивают зависимости, уменьшают связанность и делают код более структурированным. Это, в свою очередь, способствует лучшей поддерживаемости, расширяемости и повторному использованию. Благодаря этим принципам разработчики могут создавать более сложные и стабильные системы, которые легче развивать и поддерживать.

**23. Определить наследование и его роль в ООП.**

Чем-то напоминает вопрос номер *6. Как наследование способствует повторному использованию кода в ООП?* Но здесь подробнее).

Принцип программирования **наследование** является одним из ключевых понятий в ООП. Он позволяет создавать иерархии классов, где один класс (подкласс) наследует свойства и методы другого класса (базового класса). Это позволяет сокращать дублирование кода, упрощать структуру программы и создавать более логичные иерархии объектов.

Принцип наследования является одним из фундаментальных понятий объектно-ориентированного программирования и предоставляет несколько преимуществ, которые делают его важным и полезным инструментом при проектировании программных систем:

* **Повторное использование кода**. Наследование позволяет создавать иерархии классов, где общая функциональность реализуется в родительском классе, и все подклассы автоматически наследуют этот код. Это способствует повторному использованию кода, что уменьшает дублирование и облегчает его поддержку.
* **Расширяемость**. Принцип наследования позволяет создавать новые классы, расширяющие функциональность существующих классов. Подклассы могут добавлять новые свойства и методы, а также переопределять поведение унаследованных методов. Это делает код более гибким и позволяет легко вносить изменения.
* **Упрощение кода**. Использование наследования позволяет разбивать большие и сложные классы на более мелкие и управляемые части. Каждый подкласс специализируется на определенном аспекте функциональности, что упрощает понимание и поддержку кода.
* **Полиморфизм**. Наследование поддерживает концепцию полиморфизма, которая позволяет обращаться к объектам подклассов через ссылки на родительские классы. Это облегчает обработку групп объектов с различными типами, что упрощает написание общего и универсального кода.
* **Абстракция**. Наследование позволяет выделить общие характеристики объектов и создать абстрактные классы, которые определяют интерфейс для группы связанных классов. Абстрактные классы предоставляют общую сущность без необходимости определения всех деталей реализации.
* **Структурирование кода**. Наследование помогает упорядочить классы в логические иерархии, что улучшает структуру программы. Каждый класс наследует функциональность от одного или нескольких родительских классов, что улучшает организацию кода и делает его более понятным и легко поддерживаемым.

В целом, принцип наследования позволяет создавать более гибкие, модульные и расширяемые программы, что упрощает разработку и сопровождение сложных проектов. Он способствует повторному использованию кода и помогает соблюдать принципы DRY (Don’t Repeat Yourself) и SOLID, что в свою очередь способствует созданию качественного и эффективного кода.

**24. Как подкласс наследует свойства и поведение суперкласса?**

Вот несколько ключевых аспектов, которые описывают, как происходит наследование (его процесс):

1. Определение класса:

Суперкласс (или родительский класс) определяет общие характеристики и поведения для своих подклассов. Это могут быть как свойства (атрибуты), так и методы (функции), которые реализуют определенные действия.

2. Наследование свойств:

Подкласс автоматически получает все публичные и защищенные (protected) атрибуты и методы суперкласса. Это значит, что экземпляры подкласса могут использовать эти свойства и методы так же, как и экземпляры суперкласса.

3. Переопределение методов:

Подкласс имеет возможность переопределить методы суперкласса (virtual / override). Это позволяет изменять или дополнить поведение унаследованного метода.

5. Расширение функциональности:

Подкласс может добавлять свои собственные атрибуты и методы, что позволяет создать более специфичный функционал, не изменяя суперкласс.

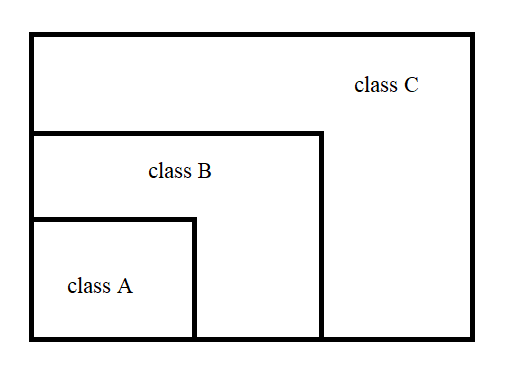
6. Множественное наследование:

В некоторых языках программирования подкласс может наследовать от нескольких суперклассов, что позволяет комбинировать свойства и методы различных классов. Однако это может привести к сложности, такой как "алмазная проблема" (“проблема ромба”, проблема ромбовидного наследования), когда возникает неоднозначности из-за различных иерархий.

7. Инкапсуляция и доступность:

Важно помнить, что принципы инкапсуляции подсказывают, какие атрибуты и методы доступны для подкласса. В большинстве языков программирования существуют различные уровни доступа (например, public, protected, private), что влияет на то, как суперкласс взаимодействует с подклассом.

Таким образом, механизм наследования позволяет создавать гибкие и переиспользуемые структуры кода, обеспечивая возможность более эффективного проектирования программного обеспечения.



*Class B наследует от class A. Class C наследует от class B.*

**25. Объясните термины “суперкласс”, “подкласс”, “базовый класс” и “производный класс”.**

Класс, который наследует данные, называется подклассом (subclass), производным классом (derived class) или дочерним классом (child). Класс, от которого наследуются данные или методы, называется суперклассом (super class), базовым классом (base class) или родительским классом (parent). Термины “родительский” и “дочерний” чрезвычайно полезны для понимания наследования. Как ребенок получает характеристики своих родителей, производный класс получает методы и переменные базового класса.

А теперь давайте уточним разницу (*да, она правда существует*) между этими понятиями >:)):

**Суперкласс**, родительский класс, предок, родитель или надкласс — класс, производящий наследование в подклассах, т. е. класс, от которого наследуются другие классы. Суперклассом может быть подкласс, базовый класс, абстрактный класс и интерфейс.

**Подкласс, производный класс,** дочерний класс, класс потомок, класс наследник или класс-реализатор — класс, наследуемый от суперкласса или интерфейса, т. е. класс, определённый через наследование от другого класса или нескольких таких классов. Подклассом может быть суперкласс.

**Базовый класс** — это класс, находящийся на вершине иерархии наследования классов и в основании дерева подклассов, т. е. не являющийся подклассом и не имеющий наследований от других суперклассов или интерфейсов. Базовым классом может быть абстрактный класс и интерфейс. Любой не базовый класс является подклассом.

**26.Что такое переопределение метода и почему оно важно при наследовании?**

**Переопределение метода** (*overriding*) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или в одном из родительских классов. Реализация метода в подклассе *переопределяет* (заменяет) его реализацию в суперклассе, описывая метод с тем же названием, что и у метода суперкласса, а также у нового метода подкласса должны быть те же параметры или сигнатура, тип возвращаемого результата, что и у метода родительского класса.

Версия метода, которая будет исполняться, определяется объектом, используемым для его вызова. Если вызов метода происходит от объекта родительского класса, то выполняется версия метода родительского класса, если же объект подкласса вызывает метод, то выполняется версия дочернего класса. Некоторые языки программирования позволяют программисту защищать методы от переопределения.

**27. Опишите концепцию отношения “is-a” при наследовании.**

Между классами возможны два типа отношений:

1. Отношение типа **is-a** (есть, является), при котором один класс является подвидом другого класса. При таком отношении один класс расширяет (детализирует и уточняет) возможности другого класса. Расширение возможностей класса осуществляется благодаря использованию наследования.

2. Отношение, при котором существует взаимосвязь между двумя классами. Здесь выделяют два подвида взаимосвязи между классами:

2.1. Отношение типа **has-a** (класс содержит другой класс). В этом случае в классе объявляется один или несколько экземпляров другого класса. При данном отношении возможны два случая взаимодействия. Первый случай, это когда объект (экземпляр), который объявлен в классе, не является составной частью класса (***агрегация***) и его использование не влияет на функциональную работу класса. Второй случай, когда объект, объявленный в классе, является составной частью этого класса (***композиция***).

2.2. Отношение типа **uses** (класс “использует” другой класс). В этом случае класс содержит программный код другого вложенного класса, к которому он имеет доступ.

Пример простейшего типа отношения **is-a** (наследование):

Суть отношения типа is-a состоит в том, что класс есть подвидом другого класса. Например, есть базовый класс Circle, который расширяется классом CircleColor. Класс CircleColor является подвидом класса Circle и добавляет к нему поле цвета.

Примеры отношения между классами типа **has-a:**

При отношении has-a класс содержит один или несколько объектов (экземпляров) другого класса. Существуют два вида отношения has-a:

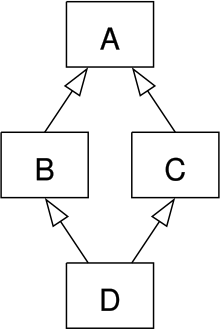
* Агрегация. Это случай, когда один или несколько вложенных объектов не являются частью класса. Класс может содержать любое количество таких объектов (даже 0).
* Композиция. В этом случае один или несколько вложенных объектов являются частью класса, то есть без этих объектов невозможно логическое существование самого класса.

В случае агрегации класс содержит множество (один или несколько) объектов других классов, которые не являются составной частью этого класса (не содержат код, который дополняет работу самого класса).

**28. Как ООП предотвращает “проблему ромба” с помощью виртуального наследования?**

**Ромбовидное наследование** (*diamond* inheritance) — ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. При этой схеме наследования может возникнуть неоднозначность: если объект класса D вызывает метод, определенный в классе A (и этот метод не был переопределен в классе D), а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: от B или от C?

Проблема ромба (*diamond problem*) получила своё название благодаря очертаниям диаграммы наследования классов в этой ситуации.



**Виртуальное наследование** предотвращает появление множественных объектов базового класса в иерархии наследования. Таким образом, конструктор базового класса будет вызван только единожды, а обращения к методам без их переопределения в дочернем классе не будут вызывать ошибку при компиляции.

Виртуальное наследование применяется в тех случаях, когда множественное наследование вместо предполагаемой полной композиции свойств классов-предков приводит к ограничению доступных наследуемых свойств вследствие неоднозначности. Базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слова virtual.

C++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно, в результате чего объект D будет на самом деле содержать два разных подобъекта A, и при использовании членов A потребуется указать путь наследования (B::A или C::A). Чтобы сгенерировать ромбовидную структуру наследования, необходимо воспользоваться виртуальным наследованием класса A на нескольких путях наследования: если оба наследования от A к B и от A к C помечаются спецификатором virtual (например, class B : virtual public A), C++ специальным образом проследит за созданием только одного подобъекта A, и использование членов A будет работать корректно. Если виртуальное и невиртуальное наследования смешиваются, то получается *один виртуальный подобъект* A и *по одному невиртуальному подобъекту A для каждого пути невиртуального наследования* к A.

При виртуальном вызове метода виртуального базового класса используется так называемое правило доминирования: компилятор запрещает виртуальный вызов метода, который был перегружен на нескольких путях наследования.

**29. Каковы преимущества и потенциальные недостатки использования наследования?**

Как и любая другая концепция программирования, наследование имеет свои преимущества и недостатки.

**Преимущества наследования:**

1. Повторное использование кода.

Наследование обеспечивает возможность повторного использования кода, позволяя подклассу наследовать свойства и поведение родительского класса. Это означает, что подкласс может основываться на существующем коде родительского класса, уменьшая объем кода, который необходимо писать и поддерживать. Возможность повторного использования кода — ключевой принцип ООП, а наследование — мощный инструмент для его достижения.

2. Улучшенная модульность.

Наследование способствует модульности проектирования программного обеспечения, позволяя создавать иерархию классов. Каждый класс в иерархии может сосредоточиться на определенном аспекте предметной области, что упрощает понимание, обслуживание и расширение системы. Модульность также упрощает модификацию или замену отдельных компонентов, не затрагивая всю систему.

3. Более простое обслуживание.

Наследование упрощает обслуживание и обновление программных систем. Когда в родительский класс вносятся изменения, все подклассы автоматически наследуют эти изменения, сокращая усилия, необходимые для обслуживания системы. Это особенно полезно в больших и сложных системах, где внесение изменений в отдельные классы может иметь волновой эффект во всей системе.

4. Более быстрое развитие.

Наследование обеспечивает более быструю разработку, предоставляя основу для создания новых классов. Наследуя свойства и поведение родительского класса, разработчики могут сосредоточиться на добавлении новых функций и возможностей в подкласс, а не начинать с нуля.

5. Улучшенная читабельность.

Наследование может улучшить читаемость кода, предоставляя ясный и краткий способ выражения отношений между классами. Используя наследование, разработчики могут создавать иерархию классов, отражающую реальные отношения между объектами, что упрощает понимание и поддержку кода.

**Недостатки наследования:**

1. Плотная связь.

Наследование может привести к тесной связи между классами, что затрудняет изменение одного класса, не затрагивая другие. Это может привести к созданию жесткой и негибкой системы, склонной к ошибкам и головной боли при обслуживании.

2. Проблема хрупкого базового класса.

Проблема хрупкого базового класса возникает, когда подкласс тесно связан со своим родительским классом, что затрудняет изменение родительского класса без разрушения подкласса. Это может привести к каскаду ошибок и проблемам с обслуживанием.

3. Проблемы переопределения.

Наследование может привести к проблемам переопределения, когда подкласс переопределяет метод своего родительского класса, но родительский класс не знает о переопределении. Это может привести к неожиданному поведению и ошибкам.

4. Проблемы множественного наследования.

В языках, поддерживающих множественное наследование, может возникнуть «проблема ромба», когда класс наследует конфликтующие методы от нескольких родительских классов. Это может привести к неясностям и ошибкам.

5. Чрезмерная инженерия.

Наследование может привести к чрезмерному проектированию, когда для моделирования простой проблемной области создается сложная иерархия классов. Это может привести к ненужной сложности, затрудняя понимание и обслуживание системы.

**30. Приведите пример одиночного и множественного наследования.**

Думаю, каждому по силам справиться с этим. Достаточно привести любые примеры из жизни (или из игры, например… Множественно наследуемся от самолёта и автомобиля в один класс и получаем на выходе новый класс “Летающий автомобиль” 😊).

**31. Как ООП поддерживает создание иерархий с использованием наследования?**

Объектно-ориентированная программа состоит из классов. В процессе работы программы создаются экземпляры классов, взаимодействуют с другими экземплярами и завершают своё существование. Взаимодействуя с себе-подобными, классы образуют два типа иерархий:

— Иерархия отношения часть -> целое;

— Иерархия отношений общее -> частное.

**Иерархия** – это упорядочение абстракций, расположение их по уровням. Абстракция – вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. (Набор классов, связанных отношением наследования, называют **иерархией**.)

Объектно-ориентированное программирование (ООП) предоставляет мощные средства для создания иерархий классов с использованием наследования, что позволяет разработчикам организовывать код и повторно использовать его более эффективно.

Наследование — это механизм ООП, позволяющий одному классу (наследнику) получать свойства и методы другого класса (родителя). Это позволяет создавать новые классы на основе уже существующих, добавляя или модифицируя функциональность, что является одним из ключевых принципов ООП.

Одним из больших преимуществ ООП является возможность создания многоуровневых иерархий. Например, мы можем создать класс Млекопитающее, от которого будут наследоваться классы Животное, а уже от Животного — классы Собака и Кошка и т.п.

Наследование в ООП также поддерживает полиморфизм — возможность работы с объектами различных классов через один интерфейс. Это означает, что разные классы могут иметь методы с одинаковыми именами, которые ведут себя по-разному.

Наличие иерархий классов значительно упрощает поддержку и изменение кода. Если требуется изменить поведение или свойства родительского класса, это автоматически повлияет на все дочерние классы.

Ограничивая доступ к внутренним данным и методам с помощью модификаторов доступа (например, public, protected, private), ООП позволяет создавать защищенные интерфейсы для взаимодействия с объектами. Это приводит к повышению безопасности кода и уменьшает вероятность ошибок.

В общем и целом, ООП и наследование позволяют организовать код в иерархические структуры, упрощая его поддержку и расширение, добавляя возможность полиморфизма, обеспечивая инкапсуляцию и защиту данных. Это делает объектно-ориентированное программирование одним из самых популярных подходов в разработке программного обеспечения.

**32. Когда в ООП следует отдавать предпочтение композиции, а не наследованию?**

Композиция: если поле имеет тип Класс, то оно может содержать ссылку на другой объект этого класса, создавая таким образом связь между двумя объектами. Не влезая в дебри различий между простой ассоциацией, агрегированием и композицией, лучше давайте "на пальцах" определим: композиция — это когда один объект предоставляет другому свою функциональность частично или полностью (т.е. является составной частью класса (внутри другого класса)).

Наследование полезнее всего для группировки сходных сущностей и понятий, определения семейств классов, и вообще для организации терминов и понятий, описывающих предметную область. Зачастую, когда значительная часть предметной логики уже реализована, исходно выбранные иерархии наследования перестают работать. Если всё к тому идет, не бойтесь разобрать и заново сложить эти иерархии так, чтобы они лучше соответствовали и работали друг с другом.

В ситуации, когда вроде бы подходят сразу и наследование, и композиция, стоит взглянуть на дизайн в двух плоскостях:

1. Структура и механическое исполнение объектов.
2. Что они обозначают по смыслу и как взаимодействуют.

Пока наследование остается внутри одной плоскости, все нормально. Но если иерархия проходит через две плоскости сразу, это плохой симптом.

Например, у вас есть один объект внутри другого. Внутренний объект реализует значительную часть поведения внешнего. У внешнего объекта много методов, которые просто передают параметры во внутренний объект и возвращают от него результат. В этом случае посмотрите, а не стоит ли унаследоваться от внутреннего объекта, хотя бы частично.

Чуть-чуть инструкций: наследуем, если:

- Оба класса из одной предметной области;

- Наследник является корректным подтипом предка;

- Код предка необходим либо хорошо подходит для наследника;

- Наследник в основном добавляет логику;

Иногда все эти условия выполняются одновременно:

- В случае моделирования высокоуровневой логики из предметной области.

- При разработке библиотек и расширений для них.

Если это не ваш случай, то и наследование вам, скорее всего, будет нужно не часто. Но не потому, что надо "предпочитать" композицию наследованию, и не потому, что она "лучше". Нужно выбирать то, что подходит должным образом для конкретно вашей задачи.

**33. Дайте определение полиморфизму и его значению в ООП.**

**Полиморфизм** – механизм ООП, который позволяет разным сущностям выполнять одни и те же действия. При этом не важно, как эту сущности устроены внутри и чем они различаются.

Полиморфизм в контексте ООП означает, что разные объекты могут реагировать на один и тот же запрос, проявляя разное поведение в зависимости от своего типа. Это позволяет сократить дублирование кода, улучшить читаемость и облегчить расширение программы.

**Преимущества принципа полиморфизма:**

* Гибкость и расширяемость.

Полиморфизм позволяет добавлять новые типы объектов и операций без изменения существующего кода. Новые классы, реализующие общий интерфейс, могут быть легко интегрированы в существующую систему.

* Упрощение кода.

Полиморфизм способствует уменьшению дублирования кода. Общий интерфейс или абстрактный базовый класс позволяют описать общее поведение, и каждый конкретный класс реализует только свою специфичную логику.

* Читаемость кода.

Полиморфизм делает код более интуитивно понимаемым, так как работа с различными объектами происходит через общий интерфейс. Это упрощает восприятие кода другими разработчиками и способствует поддержке программы.

* Расширение функциональности.

Добавление новых функций или операций для существующих классов становится проще. Достаточно реализовать необходимые методы в новых классах, которые наследуют общий интерфейс.

* Повторное использование кода.

Полиморфизм позволяет использовать одни и те же методы для разных типов данных. Это устраняет необходимость создания аналогичных функций для разных классов.

* Улучшение тестирования.

Тестирование становится более удобным, так как можно создать общие тестовые сценарии для всех классов, реализующих один интерфейс. Это способствует повышению качества и надежности программы.

* Абстракция и инкапсуляция.

Полиморфизм позволяет абстрагироваться от конкретных реализаций и сосредоточиться на общем поведении объектов. Также он способствует инкапсуляции, разделяя интерфейс от деталей реализации.

* Облегчение командной разработки.

Когда разработчики работают над разными частями программы, полиморфизм позволяет им взаимодействовать через общие интерфейсы без необходимости глубокого понимания внутренней реализации друг друга.

**34. Объясните разницу между полиморфизмом времени компиляции и полиморфизмом времени выполнения.**

**Полиморфизм времени компиляции.**

Этот тип полиморфизма, также называемый *статическим полиморфизмом*, достигается путем создания нескольких методов с одинаковыми именами в одном классе, но каждый из которых имеет разное количество параметров или параметров разных типов данных.

**Полиморфизм времени выполнения.**

Этот тип полиморфизма, также называемый *динамическим полиморфизмом*, возникает, когда дочерний класс имеет собственное определение одного из методов-членов родительского класса. Это называется переопределением метода.

**35. Что такое перегрузка метода и как она обеспечивает полиморфизм?**

При определении методов какого-нибудь класса в программах необходимо указать тип возвращаемого методом значения, а также количество параметров и тип каждого из них.

Чтобы избежать дублирования функций, ряд языков программирования позволяют определять несколько функций с одним и тем же именем. В процессе компиляции принимаются во внимание количество аргументов, используемых каждой функцией, а затем вызывается именно требуемая функция. Предоставление компилятору выбора среди нескольких функций называется *перегрузкой*. Перегрузка является одним из способов реализации *полиморфизма*.

Перегрузка методов позволяет использовать одно и то же имя для нескольких функций с разным количеством или разным типом параметров.

**Перегрузка методов** – это один из способов, которым достигается полиморфизм в языках программирования. Две и более функции могут иметь одно и то же имя, а отличаться набором аргументов в интерфейсе (описании).

**Полиморфизм** – позволяет использовать один и тот же интерфейс при реализации целого круга различных действий.

Наиболее распространенным видом перегрузки методов является перегрузка конструкторов в классе.

**36. Опишите понятие “интерфейс” в контексте ООП.**

**Интерфейс** – это абстрактный класс, содержащий чисто виртуальные функции (поведения). Они предоставляют абстрактный набор методов, которые классы должны реализовать.

Интерфейс позволяет вам не знать деталей реализации, а просто взаимодействовать с объектом. Можно сказать, что интерфейс определяет то, как мы можем использовать объект.

Другими словами:

Интерфейс – это определение функциональности, в виде определения методов и свойств, без каких-либо привязок к особенностям класса.

Класс может реализовывать более одного интерфейса. Интерфейс не содержит реализации методов.

Метод класса реализующий метод интерфейса должен быть не статическим, объявленным со спецификатором доступа public, иметь ту же сигнатуру.

Реализующий интерфейс класс может не придерживаться политики реализации аксессоров свойств интерфейса, за исключением ситуации, когда интерфейс объявлен как explicit.

Интерфейс не может быть инициализирован оператором new.

Интерфейсы могут содержать методы, свойства, индексаторы и события.

**37. Как интерфейсы способствуют множественному наследованию и полиморфизму?**

Интерфейсы играют ключевую роль в реализации множественного наследования и полиморфизма в объектно-ориентированном программировании.

1) **Множественное наследование**: В языках, где оно не поддерживается непосредственно, классы имеют возможность реализовывать сразу несколько контрактов (интерфейсов). Это означает, что класс может унаследовать поведение (интерфейсы) от разных источников, что дает ему возможность комбинировать различные функциональности без конфликтов, связанных с множественным наследованием классов.

2) **Полиморфизм**: Интерфейсы обеспечивают возможность использования одного интерфейса с различными реализациями в разных классах. Это значит, что метод, принимающий интерфейс в качестве параметра, может работать с любым объектом, который этот интерфейс реализует. Таким образом, можно создавать обобщенные коды, которые могут работать с различными типами данных, повышая гибкость и переиспользуемость кода.

В общем и целом, интерфейсы позволяют разработчикам создавать более структурированные и масштабируемые решения, обеспечивая чистоту архитектуры и удобство в работе с различными типами объектов.

**38. Приведите пример полиморфизма с использование переопределения метода.**

#include <iostream>

#include <vector>

// Базовый класс

class Animal {

public:

// Виртуальный метод

virtual void speak() const {

std::cout << "Animal speaks" << std::endl;

}

// Виртуальный деструктор

virtual ~Animal() {}

};

// Производный класс Dog

class Dog : public Animal {

public:

void speak() const override { // Переопределение метода

std::cout << "Woof!" << std::endl;

}

};

// Производный класс Cat

class Cat : public Animal {

public:

void speak() const override { // Переопределение метода

std::cout << "Meow!" << std::endl;

}

}; 😊😊😊

Вы справитесь с придумыванием своего примера по аналогии (или, например, что-нибудь из своей лабораторной (да всё что-угодно)).

**39.Что такое абстрактные классы и как они связаны с полиморфизмом?**

**Абстрактный класс** в объектно-ориентированном программировании — базовый класс, который не предполагает создания экземпляров. (Ещё одно определение: **Абстрактный класс** – это класс, который содержит или наследует без переопределения хотя бы одну чисто виртуальную функцию).

Абстрактные классы реализуют на практике один из принципов ООП — полиморфизм. Абстрактный класс может содержать (или не содержать) абстрактные методы и свойства (определяет интерфейс для переопределения производным классом). Абстрактный метод *не реализуется* для класса, в котором описан, однако *должен быть реализован для его неабстрактных потомков, что тесно связано с полиморфизмом*. Абстрактные классы представляют собой наиболее общие абстракции, то есть имеющие наибольший объём и наименьшее содержание. (Не стоит путать с полиморфным классом – это такой класс, который содержит виртуальный метод, который в свою очередь может иметь некоторую реализацию / функционал).

Одна из основных целей использования абстракции в ООП – повышение гибкости и упрощение разработки. Абстрактный подход помогает создавать интерфейсы и классы, которые определяют только те свойства и методы, которые необходимы для выполнения определенной задачи. Это позволяет создавать более гибкие и масштабируемые приложения, которые легко поддаются изменению и расширению.

Абстракция позволяет создавать общие модели объектов, которые могут использоваться для создания конкретных объектов. Упрощает работу со сложными системами, которые включают множество взаимодействующих компонентов, и позволяет создавать расширяемые, модульные приложения.

Полиморфизм же проявляется в том, что каждый подкласс абстрактного класса реализует абстрактные (чисто виртуальные методы) по-своему. Абстракция позволяет определить общий интерфейс для работы с объектами, а полиморфизм позволяет использовать этот интерфейс для работы с различными объектами, которые могут иметь различную реализацию.

(+ Абстракция позволяет скрыть детали реализации объекта и предоставить только необходимый интерфейс для работы с ним. Это помогает упростить код, сделать его более понятным и гибким.)

**40. Как полиморфизм повышает гибкость и расширяемость кода?**

Полиморфизм позволяет использовать различные реализации методов в зависимости от типа объекта, что делает код более универсальным и удобным для использования. Он также уменьшает дублирование кода – можно написать одну функцию для работы с несколькими типами объектов. Полиморфизм позволяет использовать общие интерфейсы и абстракции для работы с объектами разных типов; обеспечивает гибкость и расширяемость – можно добавлять новые типы объектов без необходимости изменять существующий код. Это дает возможность разработчикам встраивать новые функции в программу, не нарушая ее существующую функциональность.

Полиморфизм тесно связан с абстракцией:

* Абстракция позволяет скрыть детали реализации объекта и предоставить только необходимый интерфейс для работы с ним. Это помогает упростить код, сделать его более понятным и гибким.
* Полиморфизм предоставляет возможность использовать один и тот же интерфейс для работы с разными объектами, которые могут иметь различную реализацию. Этот подход значительно упрощает расширение функциональности ПО.

**41. Что такое множественное наследование?**

**Множественное наследование** – это свойство, поддерживаемое частью объектно-ориентированных языков программирования, когда класс может иметь более одного суперкласса (непосредственного класса-родителя) и тем самым наследовать поведения, свойства сразу от нескольких классов-родителей; интерфейсы поддерживают множественное наследование во многих языках программирования. Эта концепция является расширением «простого (или одиночного) наследования», при котором класс может наследоваться только от одного суперкласса.

**42. Как ООП позволяет вам писать более общий и многократно используемый код?**

Объектно-ориентированное программирование (ООП) предоставляет ряд концепций, которые способствуют написанию более общего и многократно используемого кода:

1. Инкапсуляция.

Инкапсуляция позволяет скрывать внутреннее состояние объектов и предоставлять доступ к данным только через публичные методы, что помогает избежать непреднамеренных изменений и упрощает использование объектов. Разделяя интерфейс и реализацию, мы можем изменять внутренность класса, не затрагивая код, который его использует.

2. Наследование.

Наследование позволяет создавать новые классы на основе существующих. Это означает, что мы можем создавать специализированные версии классов, которые наследуют свойства и методы родительского класса. Благодаря этому можно избежать дублирования кода и облегчить его поддержку.

3. Полиморфизм.

Полиморфизм позволяет объектам разных классов быть использованными в одном и том же контексте. Это достигается через переопределение методов в дочерних классах и использование интерфейсов. Таким образом, можно писать более гибкий код, который может работать с различными типами объектов.

+ 4. Абстракция.

Абстракция позволяет сосредоточиться на основных характеристиках, игнорируя детали реализации. Это достигается через создание абстрактных классов и интерфейсов, которые определяют, что должно делать поведение, но не как это реализовано. Это позволяет разрабатывать более общие решения, которые могут быть использованы в различных контекстах.

+ 5. Композиция

Композиция основана на использовании объектов внутри других объектов. Это позволяет строить более сложные системы, не полагаясь только на иерархию классов. Вместо наследования функциональности можно комбинировать различные объекты, что делает код более гибким и удобным для повторного использования.

Комбинируя эти концепции, а также соблюдая принципы программирования и SOLID, разработчики могут создавать более структурированный, управляемый и повторно используемый код. ООП упрощает поддержку и расширение программного обеспечения, что особенно важно в масштабных проектах. Такой подход позволяет адаптировать код к новым требованиям, не нарушая его работоспособности.

**43. Определить инкапсуляцию и её роль в ООП.**

Очередной (в этот раз частичный) повтор вопроса… (*см .5. Объясните концепцию инкапсуляции в ООП*).

**Инкапсуляция в программировании** – это принцип (концепция), согласно которому внутреннее устройство сущностей нужно объединять в специальной “оболочке” и скрывать от вмешательства извне. Доступ к объектам возможен через специальные открытые методы, а напрямую обратиться к их содержимому нельзя.

**Инкапсуляция** – механизм ООП, позволяющий объединить данные и методы, работающие с этими данными в единый объект и скрыть детали реализации от пользователя. Доступ к объектам возможен через специальные открытые методы (или интерфейсы), а напрямую обратиться к их содержимому нельзя.

**44. Как инкапсуляция помогает обеспечить сокрытие данных и контроль доступа?**

**Как инкапсуляция помогает обеспечить сокрытие данных:**

1. Частные и защищенные члены: в классе можно объявлять атрибуты (переменные) как частные (private) или защищенные (protected). Это ограничивает доступ к ним из внешнего кода.

2. Защита от некорректных данных: путем сокрытия данных и предоставления методов для их изменения мы можем гарантировать, что данные всегда будут находиться в корректном состоянии.

**Как инкапсуляция помогает обеспечить контроль доступа:**

1. Ограничение доступа: быстрое ограничение доступа к критическим данным и методам позволяет защитить внутреннее состояние объекта от непреднамеренных изменений. Например, если метод является частным, его не сможет вызвать никакой код, находящийся вне класса.

2. Разделение ответственности: инкапсуляция помогает разделить обязанности между различными частями программы. Один класс может отвечать за хранение данных, а другой — за их обработку. Это упрощает поддержку и развитие кода, так как изменения в одном классе не влияют на другие.

3. Инкапсуляция поведения: методы класса могут включать логику проверки прав доступа.

4. Интерфейс: инкапсуляция позволяет создавать четкий интерфейс для классов. Вместо того чтобы работать напрямую с данными, пользователи класса взаимодействуют с ним через методы (функции), что позволяет контролировать доступ к данным и защитить их от некорректного использования.

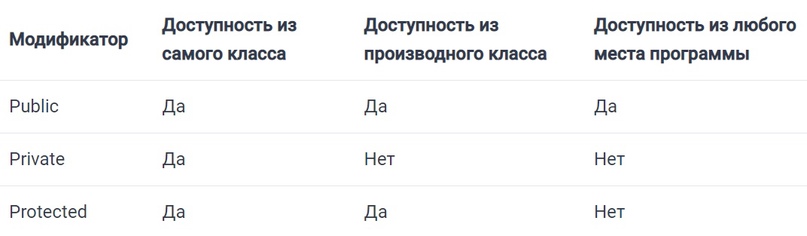
Инкапсуляция обеспечивает сокрытие данных и контроль доступа, позволяя разработчикам создавать более безопасные и устойчивые к ошибкам программы. Она создает границы между внутренним состоянием объекта и внешним кодом, что способствует поддержанию целостности данных и уменьшает риск их повреждения из-за неправильного доступа.

**45. Объясните термины “public”, “private” и “protected”, модификатор доступа в ООП.**

1) Модификаторы доступа – это ключевые слова в ООП, которые задают параметры доступа для классов, методов и иных объектов кода. Модификаторы доступа используются для реализации важного аспекта объектно-ориентированного программирования, известного как сокрытие данных.

2) Модификаторы доступа или спецификаторы доступа – это ключевые слова в объектно-ориентированных языках. Они помогают установить доступность классов, методов и других членов.

* **public**: публичный, общедоступный класс или член класса. К полям и методам, объявленным с этим ключевым словом, можно обратиться извне.
* **private**: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе. Нет доступа извне.
* **protected**: такой класс или член класса доступен из любого места в текущем классе или в производных классах, но не доступен извне.



(+ Если мы не укажем никаких модификаторов доступа для членов внутри класса, то по умолчанию модификатор доступа для членов будет private).

**46. Приведите пример инкапсуляции в классе.**

С этим все должны справиться, используя свой опыт взаимодействия с ООП. Стоит лишь помнить, что как правило в private секцию мы помещаем поля, хранящие некоторые данные, а также реализацию служебных функций (методов).

**47. Что такое методы getters и setters и почему они используются?**

**Функции доступа обычно бывают двух типов**:

**Геттеры**— это функции, которые возвращают значения закрытых переменных-членов класса.

**Сеттеры** — это функции, которые позволяют присваивать значения закрытым переменным-членам класса.

Задача геттера — получить данные о чём-то внутри объекта и передать их наружу. Например, водитель не может постоянно заглядывать в бензобак и смотреть, сколько бензина. Вместо этого водитель смотрит на индикатор топлива — тот показывает, сколько бензина осталось в баке. Индикатор топлива — это геттер: он берёт данные из объекта (автомобиль) и отдаёт их тому, кто их попросил (водитель).

Сеттер работает иначе: он меняет значения внутри объекта. В примере с автомобилем педаль газа — это сеттер: в зависимости от силы нажатия она изменяет объём впрыска топлива в двигатель. Так производители защищают двигатель от нештатных ситуаций: даже при нажатии педали газа в пол в двигатель попадёт безопасное количество топлива.

Чаще всего в программировании эти инструменты используются как метод или интерфейс объекта.

**48. Описать преимущества инкапсуляции с точки зрения удобства сопровождения и безопасности.**

**Преимущества инкапсуляции:**

* **Удобство сопровождения:**
* Упрощение изменения кода: инкапсуляция позволяет изменять внутреннюю реализацию класса, не затрагивая код, который использует этот класс. Это значит, что разработчик может улучшать или исправлять ошибки, не беспокоясь о том, что изменения повлияют на другие части программы.
* Локализация ошибок: поскольку внутренние детали класса скрыты, ошибки и их причины легче изолировать. Это упрощает диагностику и исправление проблем, что существенно ускоряет процесс сопровождения.
* Чистота интерфейса: инкапсуляция обеспечивает наличие четко определенного интерфейса для взаимодействия с объектом. Это делает код более читабельным и понятным, так как пользователю класса не нужно разбираться во всех внутренних нюансах, чтобы использовать его функциональность.
* Упрощение тестирования: инкапсуляция упрощает процесс тестирования классов и модулей, так как внутренняя логика может быть протестирована отдельно от других компонентов. Это способствует более надежной и быстрой отладке.
* **Безопасность:**

- Защита данных: Инкапсуляция обеспечивает защиту состояния объекта. Доступ к данным осуществляется только через публичные методы, что позволяет контролировать и ограничивать изменения, предотвращая непредвиденные ошибки и неправомерное использование.

- Контроль доступа: Инкапсуляция позволяет установить разные уровни доступа к методам и данным (например, с помощью модификаторов доступа, таких как private, protected и public). Это обеспечивает дополнительный уровень безопасности, так как внутренние детали класса недоступны для внешнего взаимодействия.

- Снижение риска некорректного использования: поскольку внутренние детали сокрыты, вероятность некорректного использования объекта или его методов минимизируется. Пользователи класса вынуждены следовать заданному интерфейсу, что уменьшает вероятность ошибок.

- Инкапсуляция логики защиты: Выделение логики в отдельные методы или классы способствует упрощению управления безопасностью данных и легкому внедрению механизма авторизации или валидации ввода. Можно централизовать проверку и защиту данных, что улучшает общее качество кода.

**49. Как инкапсуляция способствует снижению сложности кода?**

Инкапсуляция играет важную роль в снижении сложности кода и повышении его читаемости и поддерживаемости.

1. Скрытие внутренней реализации:

Инкапсуляция позволяет скрывать внутренние детали реализации объектов от других частей программы. Такой подход снижает количество зависимостей между различными частями программы и позволяет легко изменять внутреннюю логику, не затрагивая код, который использует этот класс.

2. Облегчение понимания кода.

Когда детали реализации скрыты, пользователям класса нужно сосредоточиться только на том, как использовать его интерфейс. Это упрощает понимание кода и делает его более доступным для других разработчиков. Четко определенный интерфейс позволяет быстро понять, что конкретный объект может делать, не вникая в его внутренние дела 😊.

3. Улучшение возможностей повторного использования

Инкапсуляция способствует созданию модульного кода, который можно легко повторно использовать в других частях приложения или в других проектах. Каждая часть кода представляет собой самостоятельный модуль с четкими границами, что позволяет использовать ее в различных контекстах без необходимости вносить изменения.

4. Поддержка изменения и расширяемости.

Благодаря инкапсуляции, изменения в внутренней реализации класса не затрагивают код, который использует его. Это дает возможность изменять, улучшать и расширять функциональность классов, не опасаясь, что эти изменения повлияют на большинство других частей кода. Таким образом, степень зависимости между модулями программы снижается, что упрощает процесс поддержки и модификации кода.

5. Упрощение отладки и тестирования.

Изолированные компоненты легче тестировать и отлаживать. Поскольку инкапсуляция подразумевает, что класс управляет своим состоянием и поведением, разработчику не нужно беспокоиться о том, как изменения в одном классе могут повлиять на другие. Это делает тестирование более целенаправленным и позволяет легче находить и устранять ошибки.

6. Упрощение управления состоянием.

Инкапсуляция позволяет управлять доступом к внутреннему состоянию объекта через методы доступа (геттеры и сеттеры). Это значит, что разработчики могут контролировать, как и когда данные изменяются, предотвращая некорректное использование или повреждение внутреннего состояния объекта. Тщательное управление состоянием также помогает предотвратить сложные ошибки, связанные с многопоточностью и одновременным доступом.

**50. Какова связь между инкапсуляцией и сокрытием данных?**

Основная цель инкапсуляции — организовать код таким образом, чтобы он был более понятным, структурированным и удобным для использования (посредством объединения данных и методов в единый объект).

**Сокрытие данных** — это аспект инкапсуляции, который относительно тщательно контролирует доступ к внутренним состояниям объекта. Оно подразумевает, что некоторые данные или методы могут быть недоступны из внешнего кода. Сокрытие данных достигается с помощью модификаторов.

Инкапсуляция включает в себя сокрытие данных, но это не одно и то же. Инкапсуляция — это более широкий концепт, который фокусируется на объединении данных и поведения, тогда как сокрытие данных — это конкретный прием, обеспечивающий защиту данных внутри инкапсулированного объекта.

Можно сказать, что сокрытие данных является одним из аспектов инкапсуляции, который помогает защитить данные и управлять доступом к ним, в то время как инкапсуляция в целом касается структуры и организации кода.

**51. Можно ли добиться инкапсуляции без использования классов и объектов? Почему да или почему нет?**

Инкапсуляции можно добиться и без использования классов и объектов, хотя в программировании на современных языках, таких как Python, Java или C++, этот подход часто осуществляется именно через объектно-ориентированные концепции.

Способы инкапсуляции без классов:

1. Использование функций и модулей (а также учитывая области видимости):

- Вы можете организовать код в модули (например, в Python) или в группы функций, которые предоставляют общий интерфейс. Внутренние переменные и функции могут оставаться невидимыми для пользователя, обеспечивая доступ только через публичные функции.

2. Замыкания:

- В языках, поддерживающих функции первого класса, можно использовать замыкания, чтобы скрыть состояние внутри функции. Это позволяет создать закрытое пространство имен, недоступное извне.

3. Использование модульного программирования:

- В некоторых языках (например, в JavaScript) можно создавать модули, используя IIFE (Immediately Invoked Function Expressions), которые содержат внутренние переменные и функции.

4. Пространства имен:

- В некоторых языках можно использовать пространства имен для хранения переменных и функций, делая их доступными только через определенные пути.

Хотя классы и объекты делают инкапсуляцию более удобной и структурированной, в отсутствии таких механизмов, как классы, всё равно можно добиться хорошего уровня инкапсуляции через использование функций, замыканий и организованного кода. Главное — создать четкий и ограниченный интерфейс, который будет взаимодействовать с внутренними данными и логикой.

**52. Как инкапсуляция улучшает сотрудничество между несколькими программистами, работающими над проектом?**

Инкапсуляция играет важную роль в сотрудничестве между программистами, работающими над проектом, и в этом контексте она предлагает несколько существенных преимуществ:

- Сокрытие деталей реализации: инкапсуляция позволяет скрыть сложные подробности реализации компонентов. При делении задач на модули каждый программист может сосредоточиться на своей части, не беспокоясь о внутренних аспектах работы других модулей.

- Четкий интерфейс: каждый модуль или компонент предоставляет четко определённый интерфейс для взаимодействия. Это облегчает понимание и использование другого кода, так как программисты могут работать с заранее определёнными методами и свойствами без необходимости углубляться в детали реализации.

- Уменьшение взаимозависимости: когда программисты работают над инкапсулированными модулями, они могут изменять и улучшать свой код независимо от других компонентов. Это уменьшает вероятность возникновения ошибок и позволяет разработчикам быстрее адаптироваться к изменениям.

- Повышение качества кода: инкапсуляция способствует созданию кода более высокого качества, так как программисты придерживаются принципов хорошего дизайна и структурирования кода. Это также делает код более удобочитаемым и понятным для других.

- Упрощение тестирования и отладки: изолированные модули легче тестировать и отлаживать. Если отдельный компонент имеет баг, его можно протестировать в отрыве от остальной системы, что значительно ускоряет процесс исправления ошибок.

- Сокращение времени на обучение: при работе с инкапсуляцией новые члены команды могут быстрее вникнуть в проект, так как им нужно изучить только интерфейсы, а не детальную реализацию каждого модуля.

- Поддержка изменений и расширений: когда требуется внести изменения в функциональность, программисты могут модифицировать или заменять внутренние элементы инкапсулированного модуля, не затрагивая другие части системы. Это способствует более легкому поддержанию и расширению проекта.

- Повышение безопасности: сокрытие данных и методов обеспечивает защиту внутреннего состояния объектов от некорректного использования. Это снижает вероятность ошибок при совместной работе и увеличивает надежность программы.