1. Основные принципы парадигмы ООП?
2. Основные преимущества использования принципов модульности программного кода?
3. Понятие класса и интерфейса. В чем принципиальная разница между ними.
4. Дружественная функция. Как вызвать, способы передачи параметров из нескольких источников
5. Дружественный класс. Как вызвать, способы передачи параметров из нескольких источников
6. Принцип прозрачности ПО. Как он сочетается с использованием дружественных функций.
7. Соблюдается ли принцип инкапсуляции на всем времени жизни метода/класса?
8. Геттеры/Сеттеры. Эффективней ли они дружественных функций. Дайте полный сравнительный анализ.
9. Основные принципы парадигмы ООП?

SOLID   
**Принцип единственной обязанности**

Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

**Принцип открытости/закрытости**

Программные сущности должны быть *открыты* для расширения, но *закрыты* для изменений.

**Принцип подстановки Барбары Лисков**

Методы, использующие некий тип, должны иметь возможность использовать его подтипы, не зная об этом.

**Принцип разделения интерфейсов**

Предпочтительнее разделять интерфейсы на более мелкие тематические, чтобы реализующие их классы не были вынуждены определять методы, которые непосредственно в них не используются.

**Принцип инверсии зависимостей**

Система должна конструироваться на основе абстракций «сверху вниз»: не абстракции должны формироваться на основе деталей, а детали должны формироваться на основе абстракций.

Появление в ООП отдельного понятия **класса** закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП — это в чистом виде [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода — если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки [**полиморфизма**](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) — возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Отдельного пояснения требует понятие обмена [**сообщениями**](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B5). Первоначально (например, в том же [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk" \o "Smalltalk)) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, то есть пересылка от одного объекта другому специального объекта-сообщения. Такая модель является чрезвычайно общей. Она прекрасно подходит, например, для описания параллельных вычислений с помощью *активных объектов*, каждый из которых имеет собственный поток исполнения и работает одновременно с прочими. Такие объекты могут вести себя как отдельные, абсолютно автономные вычислительные единицы. Посылка сообщений естественным образом решает вопрос обработки сообщений объектами, присвоенными полиморфным переменным — независимо от того, как объявляется переменная, сообщение обрабатывает код класса, к которому относится присвоенный переменной объект. Данный подход реализован в языках программирования [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk" \o "Smalltalk), [Ruby](https://ru.wikipedia.org/wiki/Ruby), [Objective-C](https://ru.wikipedia.org/wiki/Objective-C" \o "Objective-C), [Python](https://ru.wikipedia.org/wiki/Python).

Однако общность механизма обмена сообщениями имеет и другую сторону — «полноценная» передача сообщений требует дополнительных накладных расходов, что не всегда приемлемо. Поэтому во многих современных объектно-ориентированных языках программирования используется концепция **«отправка сообщения как вызов метода»** — объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов. Данный подход реализован в огромном количестве языков программирования, в том числе [C++](https://ru.wikipedia.org/wiki/C%2B%2B), [Object Pascal](https://ru.wikipedia.org/wiki/Object_Pascal), [Java](https://ru.wikipedia.org/wiki/Java), [Oberon-2](https://ru.wikipedia.org/wiki/Oberon-2). Однако, это приводит к тому, что сообщения уже не являются самостоятельными объектами, и, как следствие, не имеют атрибутов, что сужает возможности программирования. Некоторые языки используют гибридное представление, демонстрируя преимущества одновременно обоих подходов — например, [CLOS](https://ru.wikipedia.org/wiki/CLOS), [Python](https://ru.wikipedia.org/wiki/Python).

Концепция **виртуальных методов**, поддерживаемая этими и другими современными языками, появилась как средство обеспечить выполнение нужных методов при использовании полиморфных переменных, то есть, по сути, как попытка расширить возможности вызова методов для реализации части функциональности, обеспечиваемой механизмом обработки сообщений.

Инкапсуляция (encapsulation) - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

Инкапсуляция нужна для того, что бы пользователь не мог использовать не предназначенный для него функционал.