**ООП**

Прежде всего стоит ответить, зачем? Объектно-ориентированная идеология разрабатывалась как попытка связать поведение **сущности** с её данными и спроецировать объекты реального мира и бизнес-процессов в программный код.

**Классы и объекты**

Самое простое объяснение: класс — это чертеж трансформера, а экземпляры этого класса — конкретные трансформеры, например, Оптимус Прайм или Олег.

Состояние — это ряд меняющихся свойств. Поэтому у двух разных объектов одного класса мы можем наблюдать разное имя, возраст, местоположение, уровень заряда, количество боеприпасов и т. д. Само наличие этих свойств и их типы описываются в классе.  
  
**класс — это описание того, какими свойствами и поведением будет обладать объект.**

**объект — это экземпляр с собственным состоянием этих свойств.**  
  
Мы говорим **«свойства и поведение»**, но звучит это как-то абстрактно и непонятно. Привычнее для программиста будет звучать так: **«переменные и функции»**.

**«свойства»** — это такие же обычные переменные, просто они являются атрибутами какого-то объекта (их называют полями объекта).

**«поведение»** — это функции объекта (их называют методами), которые тоже являются атрибутами объекта.

Разница между методом объекта и обычной функцией лишь в том, что метод имеет доступ к собственному состоянию через поля.  
  
Итого, имеем **методы и свойства - атрибуты**.

**Ассоциация**

Традиционно в полях объекта могут храниться не только обычные переменные стандартных типов, но и другие объекты. Это отношение называется ассоциацией.  
  
Предположим, что наш трансформер оборудован пушкой. Хотя нет, лучше двумя пушками. В каждой руке. Пушки одинаковые (принадлежат к одному классу, или, если будет угодно, выполненные по одному чертежу), обе одинаково умеют стрелять и перезаряжаться, но в каждой есть свое хранилище боеприпасов (собственное состояние).

Итог: робота сделали, табельное оружие выдали, теперь разберемся, что тут происходит. В данном коде один объект стал составной частью другого объекта. Это и есть ассоциация. Она в свою очередь бывает двух видов:  
  
1. **Композиция** — случай, когда на фабрике трансформеров, собирая Оптимуса, обе пушки ему **намертво приколачивают** к рукам гвоздями, и после смерти Оптимуса, пушки умирают вместе с ним. Другими словами, жизненный цикл дочернего объекта совпадает с жизненным циклом родительского.  
  
2. **Агрегация** — случай, когда пушка выдается как пистолет в руку, и после смерти Оптимуса этот пистолет может подобрать его боевой товарищ Олег, а затем взять в свою руку, либо сдать в ломбард. То бишь **жизненный цикл дочернего объекта не зависит от жизненного цикла родительского**, и может использоваться другими объектами.

**Наследование**

**Наследование** — это механизм системы, который позволяет, наследовать одними классами **свойства и поведение** других классов для дальнейшего **расширения или модификации**.  
  
Что если, мы не хотим штамповать одинаковых трансформеров, а хотим сделать общий каркас, но с разным обвесом?

ООП позволяет нам такую шалость путем разделения логики на **сходства и различия** с последующим выносом сходств в родительский класс, а различий в классы-потомки. Как это выглядит?  
  
Оптимус Прайм и Мегатрон — оба трансформеры, но один является автоботом, а второй десептиконом. Допустим, что различия между автоботами и десептиконами будут заключаться только в том, что автоботы трансформируются в автомобили, а десептиконы — в авиацию. Все остальные свойства и поведение не будут иметь никакой разницы. В таком случае можно спроектировать систему наследования так: **общие черты (бег, стрельба)** будут описаны в **базовом классе «Трансформер»**, а различия (трансформация) в **двух дочерних классах «Автобот» и «Десептикон».**

### **Перегрузка**

Если же в классе-потомке переопределить уже существующий метод в классе-родителе, то сработает перегрузка. Это позволяет не дополнять поведение родительского класса, а модифицировать. В момент вызова метода или обращения к полю объекта, поиск атрибута происходит от потомка к самому корню — родителю. То есть, если у автобота вызвать метод fire(), сначала поиск метода производится в классе-потомке — Autobot, а поскольку его там нет, поиск поднимается на ступень выше — в класс Transformer, где и будет обнаружен и вызван.

### **Неуместное применение**

Как при описании отношений двух сущностей определить, когда уместно наследование, а когда — композиция? Можно воспользоваться популярной шпаргалкой: спросите себя, **сущность А является сущностью Б**? Если да, то скорее всего, тут подойдет наследование. Если же **сущность А является частью сущности Б**, то наш выбор — композиция.  
  
Применительно к нашей ситуации это будет звучать так:

1. Автобот является Трансформером? Да, значит выбираем наследование.
2. Пушка является частью Трансформера? Да, значит — композиция.

Для самопроверки попробуйте обратную комбинацию, получится фигня.

### **Множественное наследование**

Мы рассмотрели ситуацию, когда два класса унаследованы от общего потомка. Но в некоторых языках можно сделать и наоборот — унаследовать **один класс от двух и более родителей**, объединив их свойства и поведение. Возможность наследоваться от нескольких классов вместо одного — это множественное наследование.

### **Абстрактные классы**

От обычных классов они отличаются тем, что нельзя создать объект такого класса. Зачем же нужен такой класс? Он нужен для того, чтобы от него могли **наследоваться потомки — обычные классы**, объекты которых уже можно создавать.  
  
Абстрактный класс наряду с обычными методами содержит в себе абстрактные методы **без имплементации (с сигнатурой, но без кода)**, которые обязан имплементировать программист, задумавший создать класс-потомок.

Предположим, на сайте нужны **три вида публикаций** — **новости, объявления и статьи**. В чем-то они похожи — у всех них есть **заголовок и текст**, у **новостей и объявлений есть дата**. В чем-то они разные — **у статей есть авторы**, у **новостей — источники**, а у **объявлений — дата**, после которой оно становится не актуальным.

Грубо говоря, это класс-шаблон. Он реализует функциональность только на том уровне, на котором она известна на данный момент. Производные же классы ее дополняют.

**Инкапсуляция**

Инкапсуляция — это контроль доступа к полям и методам объекта. Под контролем доступа подразумевается не только можно/неможно, но и различные валидации, подгрузки, вычисления и прочее динамическое поведение.  
  
Во многих языках частью инкапсуляции является сокрытие данных. Для этого существуют модификаторы доступа (опишем те, которые есть почти во всех ООП языках):

* publiс — к атрибуту может получить доступ любой желающий
* private — к атрибуту могут обращаться только методы данного класса
* protected — то же, что и private, только доступ получают и наследники класса в том числе

## Интерфейсы

**Задача интерфейса** — снизить уровень зависимости сущностей друг от друга, добавив больше абстракции.

Не во всех языках присутствует этот механизм, но в ООП языках со статической типизацией без них было бы совсем худо. Выше мы рассматривали абстрактные классы, затрагивая тему контрактов, обязующих имплементировать какие-то абстрактные методы.

Так вот **интерфейс очень смахивает на абстрактный класс**, но является не классом, а просто **пустышкой с перечислением абстрактных методов** (без имплементации).

Другими словами, интерфейс имеет декларативную природу, то есть, **чистый контракт без капельки кода.**  
  
  
Классы с интерфейсами состоят в отношении «многие ко многим»: **один класс может имплементировать множество интерфейсов**, и каждый **интерфейс, в свою очередь, может имплементироваться многими классами**.  
  
У интерфейса двустороннее применение:

1. По одну сторону интерфейса — **классы, имплементирующие данный интерфейс.**
2. По другую сторону — потребители, которые используют этот интерфейс в качестве **описания типа данных**, с которым они (потребители) работают.

Например, если какой-то объект помимо основного поведения, может быть сериализован, то пускай он имплементирует интерфейс «Сериализуемый».

А если объект можно склонировать, то пусть он имплементирует еще один интерфейс — «Клонируемый».

И если у нас есть какой-то транспортный модуль, который передает объекты по сети, он будет принимать любые объекты, имплементирующие интерфейс «Сериализуемый».  
  
Представим, что каркас трансформера оборудован **тремя слотами**: слот для оружия, для генератора энергии и для какого-нибудь сканера.

Эти слоты обладают определенными интерфейсами: в каждый слот можно установить только подходящее оборудование. В слот для оружия можно установить ракетную установку или лазерную пушку, в слот для генератора энергии — ядерный реактор или РИТЭГ (радиоизотопный термоэлектрический генератор), а в слот для сканера — радар или лидар. Суть в том, что каждый слот имеет универсальный интерфейс подключения, а уже конкретные устройства должны соответствовать этому интерфейсу. К примеру, на материнских платах используется несколько типов слотов: слот для процессора позволяет подключать различные процессоры, подходящие под данный сокет, а слот SATA — любой SSD или HDD накопитель или даже CD/DVD.  
  
Обращаю внимание, что получившаяся система слотов у трансформеров — это пример использования композиции. Если же оборудование в слотах будет сменным в ходе жизни трансформера, то тогда это уже агрегация. Для наглядности, мы будем называть интерфейсы, как принято в некоторых языках, добавляя заглавную «И» перед именем: IWeapon, IEnergyGenerator, IScanner.