Объектно-ориентированное программирование для самых маленьких

5 октября 312 185 10 мин

Вы когда-нибудь замечали, что на собеседованиях задают одни и те же клишированные вопросы?

Уверен, вы поняли, что я имею ввиду.

Ну вот, к примеру:

> ***Кем вы видите себя через 5 лет?***

Или еще «лучше»:

> ***Какой ваш главный недостаток?***

Боже, ответ на этот вопрос станет моим главным недостатком.

Конечно, эти вопросы до ужаса тривиальны и не вызывают ничего, кроме раздражения, но они помогают работодателям лучше вас понять. А именно — ваше текущее состояние, жизненную позицию, взгляд на будущее и т.д.

Отвечая на эти вопросы, вы должны быть очень осторожны, так как из-за невнимательность вы можете рассказать что-то такое, о чем позже пожалеете.

В сегодняшней статье я хочу разобрать похожий «клишированный» вопрос:

> *Каковы основные принципы объектно-ориентированного программирования?*

Я был по обе стороны стола, когда звучал этот вопрос. Он звучит настолько часто, что не знать ответа на него — просто невозможно.

В 90% случаев его задают новичкам, потому что он помогает разъяснить три вопроса:

- **Готовился ли кандидат к этому интервью?**
  Если ответ дается незамедлительно — кандидат ответственно подошел к делу.
- **Прошел ли кандидат начальный этап?**
  Понимание принципов объектно-ориентированного программирования (ООП) показывает, что кандидат прошел начальный этап — теперь он видит вещи с более продвинутой точки зрения.
- **Его понимание ООП находится на глубоком или поверхностном уровне?**
  Уровень компетентности по данному вопросу часто равен уровню компетентности по большинству других. Доверьтесь мне.

Пришло время озвучить 4 принципа ООП: **инкапсуляция, абстракция, наследование и полиморфизм.**

Для junior-разработчика слова могут показаться страшными и сложными

для junior-разработчика слова могут показаться страшными и сложными. Вдобавок, при поиске ответа в Интернете, чрезмерно длинные объяснения в Википедии удваивают путаницу.

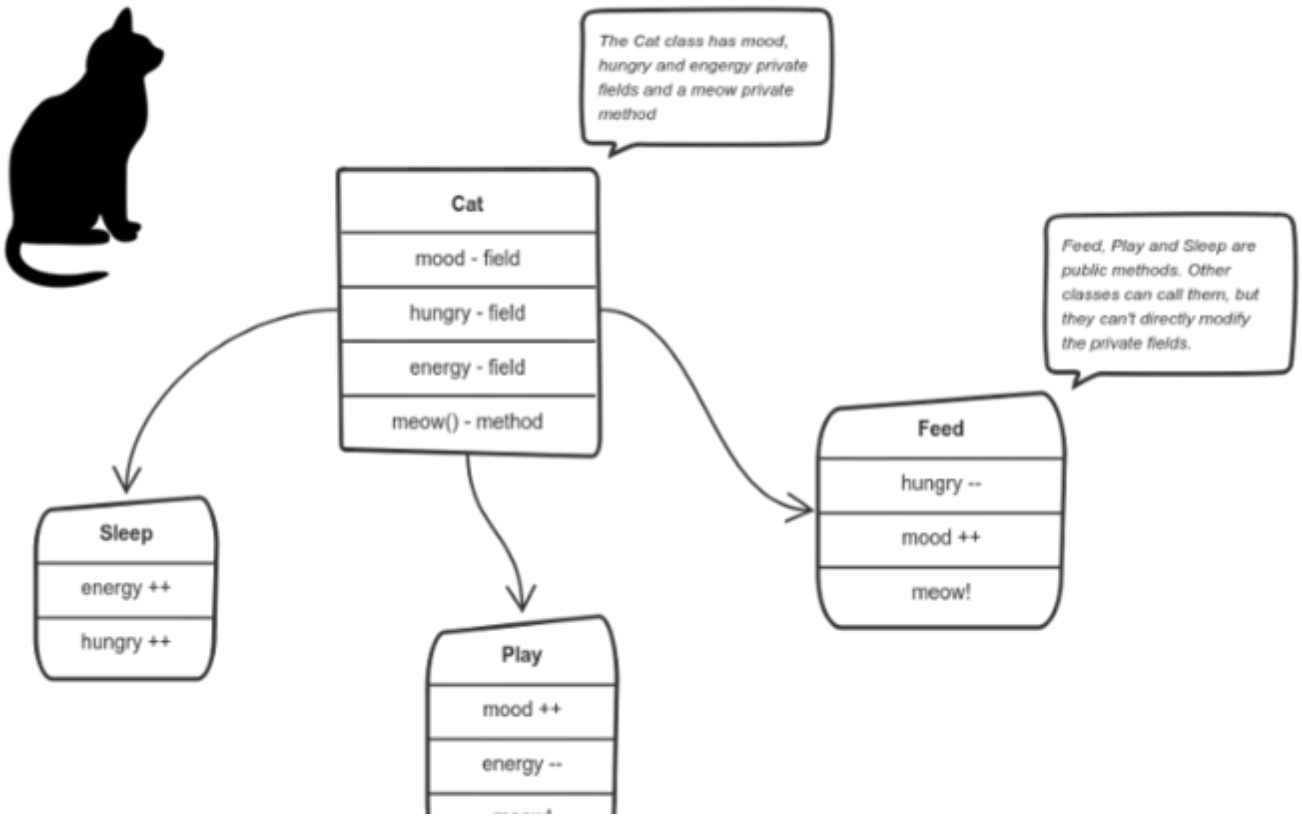
Вот почему я хочу дать простое, короткое и четкое объяснение для каждого из приведенных принципов. Может показаться, что они звучат чересчур просто, будто бы я разговариваю с маленьким ребенком, но именно такую формулировку я бы хотел услышать от кандидата на собеседовании.

Инкапсуляция

Допустим, у нас есть некая программа. У нее есть объекты, которые общаются между собой, в соответствии с правилами, установленными в программе.

Объект самостоятельно управляет своим внутренним состоянием, с помощью методов — и никто другой не может трогать его, если на это нет особого разрешения. Если другой захочет с ним взаимодействовать, ему нужно будет использовать разрешенные методы. Но (по умолчанию) менять внутреннее состояние нельзя.

Теперь допустим, что мы играем в Sims. У нас есть семья из нескольких людей и кошки. Они общаются друг с другом. Мы хотим применить инкапсуляцию, поэтому инкапсулируем всю логику «cat» в класс Cat. Это выглядит следующим образом:



Здесь внутренним состоянием кошки являются частные(private) переменные: настроение(mood), голод(hungry) и энергия(energy). Она также имеет частный(private) метод meow (). Кошка может вызвать его в любой момент, когда захочет, другие классы не могут говорить кошке, когда ей можно мяукать.

То, что им можно делать, определяется в публичных(public) методах sleep (), play () и feed (). Каждый из них каким-то образом влияет на внутреннее состояние кошки и может вызвать meow (). Таким образом, устанавливается связь между внутренним состоянием объекта и публичными методами.

Вот что такое инкапсуляция.

Абстракция

В объектно-ориентированном программировании, зачастую, у программ очень большой объем, и объекты много обuyjhgщаются между собой. Как вы понимаете, поддерживать такую кодовую базу в течение долгих лет — с постоянными изменениями — трудно.

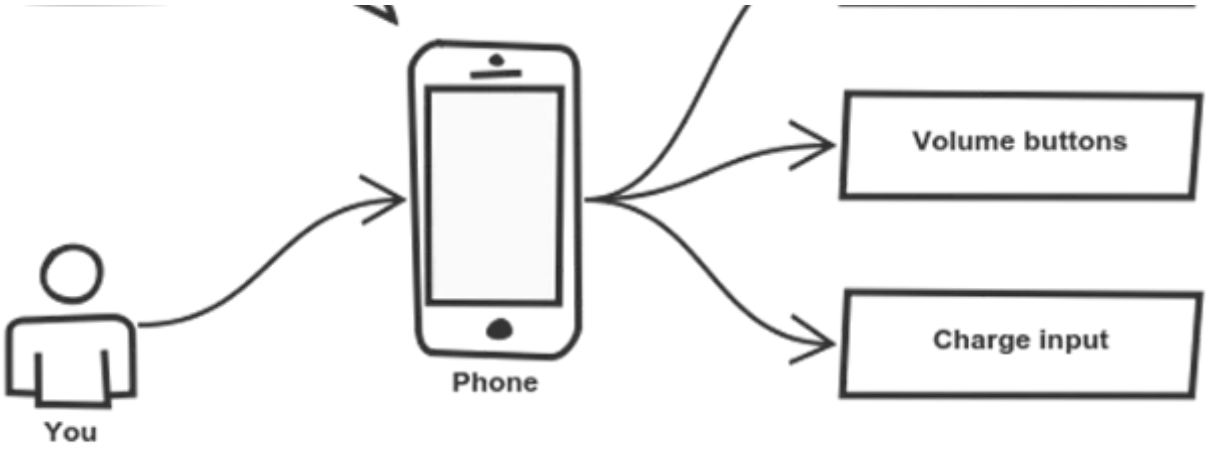
Абстракция нацелена на решение этой проблемы.

Применение абстракции означает, что нужно отбросить незначимые характеристики объекта и оставить только значимые. Данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня.

Представьте кофеварку. Когда она варит кофе, под ее корпусом творится много вещей. Но от вас требуется лишь насыпать кофе и нажать кнопку.

Данный механизм должен быть неприхотлив в использовании, а также редко меняться со временем. Подумайте об этом, как о наборе публичных(public) методов, которые любой другой класс может вызывать, не “зная”, как они работают.

Хотите еще один пример? Возьмите свой смартфон.



Каждый день вы бесчисленное количество раз проводите пальцами по экрану. Что в этот момент происходит внутри смартфона? Рядовому пользователю не нужно это знать, так как ему дан определенный перечень действий для работы с гаджетом.

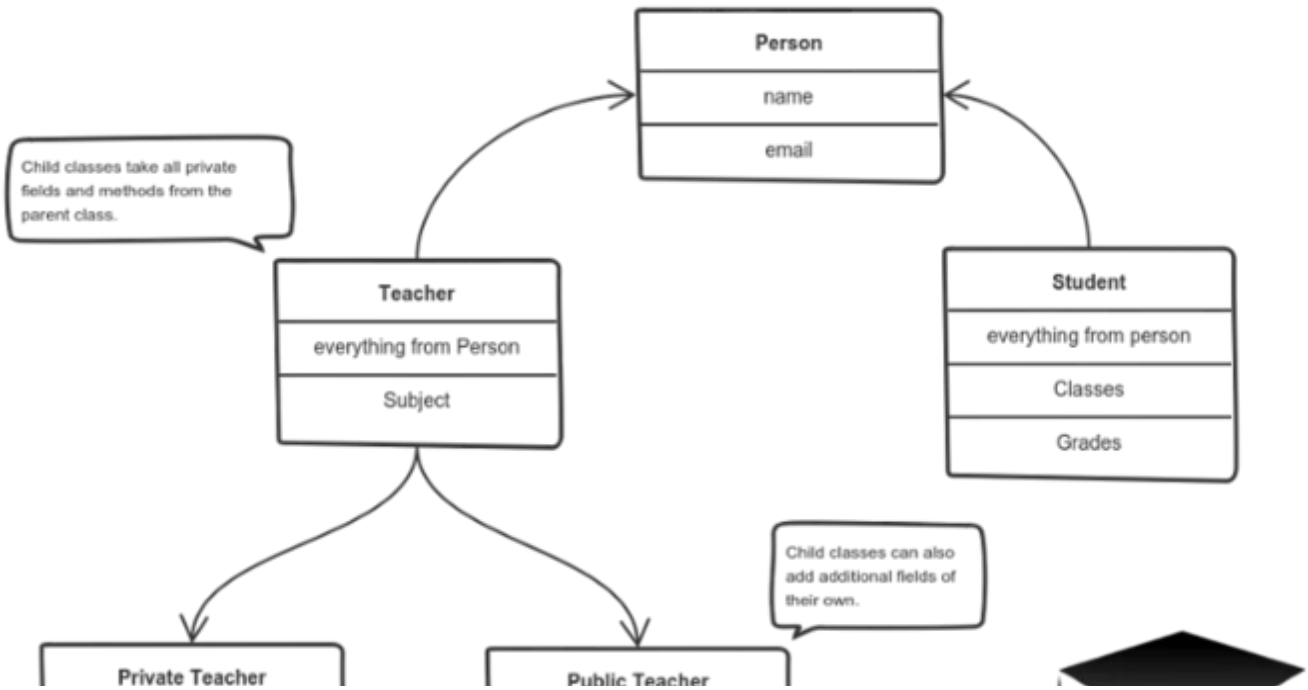
Наследование

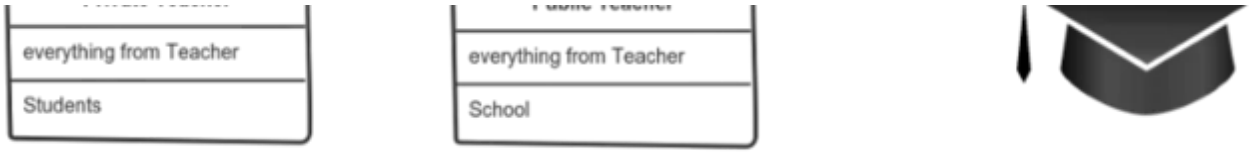
Итак, мы разобрались с двумя принципами ООП, осталось еще два.

Наследование позволяет создать новый класс (производный класс) на основе уже существующего (родительский класс).

Производный класс берет все от родительского, но имеет дополнительные характеристики. Это нужно для того, чтобы просто немного улучшить старый класс, а не создавать с нуля новый.

Например:





На данном примере, «обычный» преподаватель делится на «частного» и «публичного». «Частный» работает индивидуально с каждым студентом, а «публичный» со всеми в одно время. Эти классы имеют новые уникальные характеристики, не присущие родительскому классу.

Полиморфизм

На греческом, полиморфизм означает «многообразие форм».

Мы с вами узнали, что такое наследование, и успешно применяем его на практике. Но возникает одна проблема.

Представим, что у нас есть родительский класс и несколько производных. Мы хотим использовать коллекцию или, по-другому, список, который содержит «микс» всех этих классов. Или у нас есть метод, осуществляемый для родительского класса, но мы бы хотели использовать его и для производного.

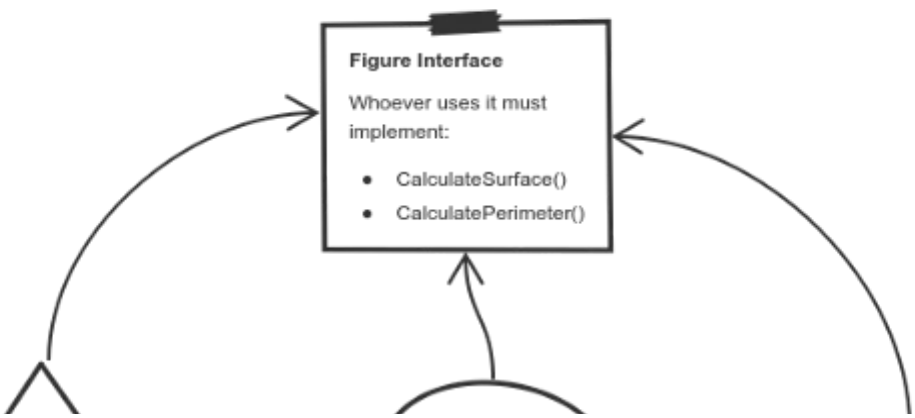
Эту проблему можно решить, используя полиморфизм.

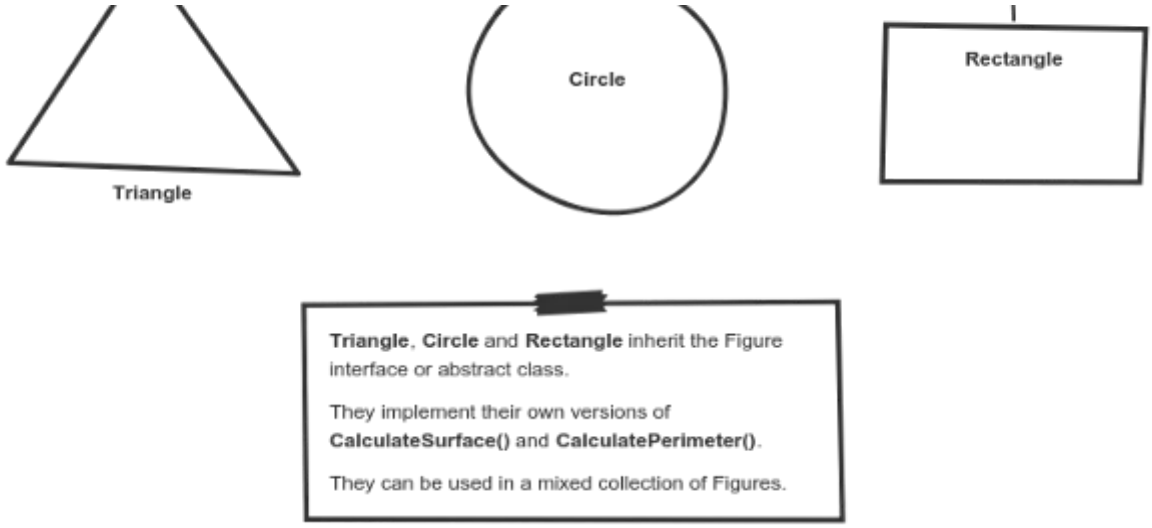
Полиморфизм позволяет использовать объекты с одинаковыми характеристиками в разных направлениях.

Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс.

Данный принцип позволяет повысить коэффициент повторного использования кода.

Взгляните на эскиз с геометрическими фигурами. Они повторно используют общий интерфейс для расчета площади поверхности и периметра:





Имея в распоряжении эти фигуры, наследующие интерфейс родительского класса, мы можем создать коллекцию (список) смешанных треугольников, окружностей и прямоугольников. И относиться к ним, как к одному и тому же типу объектов.

Затем, если потребуется вычислить площадь какого-либо элемента, эта коллекция (список) найдет и выполнит правильный метод. Если элемент является треугольником, выполняется метод CalculateSurface (). Если это круг, выполняется метод CalculateSurface (). И так далее.

Вместо того, чтобы писать класс для каждого конкретного типа следует создать типы, которые будут реализованы во время выполнения программы.

Надеюсь, мое «детское» объяснение вам помогло. Используйте его при прохождении собеседования и должность, фактически, вам обеспечена!

Источник: nuancesprog.ru - Объектно-ориентированное программирование для самых маленьких

Подписывайтесь и ставьте лайк. Этим Вы очень поможете в развитии блога!

Рекомендую также прочитать статью SOLID в объектно-ориентированном программировании

CODE BLOG / Программирование

Статьи для программистов от программиста, а также новости IT-индустрии, компьютеры, гаджеты, достижения науки и технологий, гик фильмы и сериалы. Все самое интересное для вас!

Вы подписаны

Комментарии

Ваш комментарий...

Другие публикации этого автора

Изучите эти основы JavaScript и станьте лучшим разработчиком

Топ действительно полезных ресурсов для разработчиков-самоучек

Топ 3 самых популярных языка программирования в

2018 году

Поделиться