Оглавление

3. Принцип подстановки Барбары Лисков
4. Принцип разделения интерфейса
5. Принцип инверсии зависимостей

ООП

· представление программы в виде совокупности объектов;
· каждый объект — экземпляр определенного класса;
· классы образуют иерархию наследования.

ООП программа — объекты, обменивающиеся сообщениями.

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D0%9E%D0%9E%D0%9F

Основные понятия ООП

Класс — способ описания сущности (определяющий состояние и поведение).

С точки зрения программирования — набор полей и методов, с точки зрения структуры программы — сложный тип данных.

Объект — экземпляр класса (с конкретным состоянием и поведением, определяемым классом).

Интерфейс — контракт класса (правила взаимодействия), описывающим возможности (набор доступных методов).

Состояние — значение полей, *поведение* — реализация методов.

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%A0%D0%B0%D1%81%D1%81%D0%BA%D0%B0%D0%B6%D0%B8%D1%82%D0%B5-%D0%BF%D1%80%D0%BE-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5-%D0%BF%D0%BE%D0%BD%D1%8F%D1%82%D0%B8%D1%8F-%D0%9E%D0%9E%D0%9F-%D0%BA%D0%BB%D0%B0%D1%81%D1%81-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81

Основные принципы ООП

- инкапсуляция;
- наследование;
- полиморфизм.

Инкапсуляция

- объединяем поля и методы в классы;
- скрываем детали реализации и открываем то, что необходимо при последующем использовании.

Цель инкапсуляции — уйти от зависимости внешнего интерфейса класса от реализации (малейшее изменение в классе не влечет за собой изменения внешнего поведения класса).

Наследование

· описываем новый класс на основе уже существующего с частичной или полностью заимствующей функциональностью.

Названия классов:
· базовый, родительский, предок;
· производный, наследник, потомок.

Вид отношения — *is a* (является).

Полиморфизм

· способность функции работать с полиморфными переменными (принимает значения разных типов).

ad-hoc (полиморфизм по запросу) — приведение данных и перегрузка методов.

Параметрический полиморфизм — одна и та же функция с одним и тем же телом может принимать в качестве параметра данные разных классов.

Все `if` в программе можно заменить на полиморфизм.

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%9D%D0%B0%D0%B7%D0%BE%D0%B2%D0%B8%D1%82%D0%B5-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5-

%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B-%D0%9E%D0%9E%D0%9F

Преимущества и недостатки ООП

Преимущества ООП

· *естественна* для человека;
· создавать *расширяемые системы*;
· абстрагироваться от *деталей реализации*;
· полиморфизм: работать с *разными типами* данных (полиморфная функция), изменять поведение на этапе выполнения (полиморфные функции) и работать с наследниками;
· *повторное использование* кода (время, меньше ошибок, одно улучшение положительное влияние на несколько участков программы);
· объединение полей и методов в классы — улучшают наглядность и удобства сопровождения ПО;
· модульность инкапсуляции — распараллеливание задачи между программистами и обновление версий отдельных компонентов.

Недостатки ООП

· *память* на этапе выполнения;
· излишняя универсальность (много методов);
· сложность *документирования классов* (переопределение методов);
· в сложных иерархиях классов не легко определить *принадлежность полей и методов*.

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%92-%D1%87%D0%B5%D0%BC-%D0%B7%D0%B0%D0%BA%D0%BB%D1%8E%D1%87%D0%B0%D1%8E%D1%82%D1%81%D1%8F-%D0%BF%D1%80%D0%B5%D0%B8%D0%BC%D1%83%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B0-%D0%B8-%D0%BD%D0%B5%D0%B4%D0%BE%D1%81%D1%82%D0%B0%D1%82%D0%BA%D0%B8-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4%D0%B0-%D0%B2-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B8

Виды связей

Является (is a) — наследование, ***имеет (has a)*** — ассоциация.

Наследование: *животные* и *слон* (*слон* не может наследоваться от *жирафа*).

Ассоциация (использование переменных определенного типа): *автомобиль* и *радио*.

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%A7%D1%82%D0%BE-%D0%BF%D0%BE%D0%B4%D1%80%D0%B0%D0%B7%D1%83%D0%BC%D0%B5%D0%B2%D0%B0%D1%8E%D1%82-%D0%B2-%D0%BF%D0%BB%D0%B0%D0%BD%D0%B5-%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D0%BE%D0%B2-%D0%9E%D0%9E%D0%9F-%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F-%D1%8F%D0%B2%D0%BB%D1%8F%D0%B5%D1%82%D1%81%D1%8F-%D0%B8-%D0%B8%D0%BC%D0%B5%D0%B5%D1%82

Ассоциация, композиция и агрегация

Ассоциация обозначает связь между объектами.

Композиция и ***агрегация*** — частные случаи ассоциации «часть-целое».

Авто и двигатель — композиция, авто и пассажиры — агрегация.

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%92-%D1%87%D0%B5%D0%BC-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%B7%D0%B8%D1%86%D0%B8%D0%B5%D0%B9-%D0%B8-

%D0%B0%D0%B3%D1%80%D0%B5%D0%B3%D0%B0%D1%86%D0%B8%D0%B5%D0%B9

Связывание

Связывание — присоединение вызова метода к телу метода.

Раннее связывание (статическое, early binding) — связывание компилятором перед запуском программы.

Позднее связывание (динамическое, late binding, связыванием на стадии выполнения, runtime binding) — связывание во время выполнения программы.

Методы в Java — механизм позднего связывания, если не `static` и не `final` (приватные методы по умолчанию `final`).

Источники:

https://github.com/enhorse/java-interview/blob/master/oop.md#%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5-%D0%B8-%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5-%D1%81%D0%B2%D1%8F%D0%B7%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5

SOLID

Роберт Мартин

Преимущества SOLID (при соблюдении его условий):

· программу легко дополнять, расширять, изменять и поддерживать;

· код легче читать (находить ошибки и переиспользовать).

1. Принцип единственной ответственности (SRP, The Single Responsibility Principle)

Роберт Мартин: каждый объект должен иметь *одну обязанность* и эта обязанность должна быть *полностью инкапсулирована* в класс.

2. Принцип открытости/закрытости (OCP, The Open Closed Principle)

Бертран Мейер: программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.

3. Принцип подстановки Барбары Лисков (LSP, The Liskov Substitution Principle)

Роберт Мартин: функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Поведение наследующих классов не должно противоречить поведению, заданного базовым классом.

4. Принцип разделения интерфейса (ISP, The Interface Segregation Principle)

Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения для всех.

5. Принцип инверсии зависимостей (DIP, The Dependency Inversion Principle)

Зависимость на абстракциях, нет зависимости на что-то конкретное. Абстрактность не должна зависеть на конкретное, конкретность должна завесить на абстракциях.

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракции.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

- посмотреть еще

Источники:

5 видео с канала Немчинского:

https://www.youtube.com/watch?v=O4uhPCEDzSo&t=345s&ab_channel=SergeyNemchinskiy

https://www.youtube.com/watch?v=x5OtQiKOG-Q&ab_channel=SergeyNemchinskiy

https://www.youtube.com/watch?v=NqvwYcjrwdw&ab_channel=SergeyNemchinskiy

https://www.youtube.com/watch?v=d9RJqf2o_Sw&t=23s&ab_channel=SergeyNemchinskiy

https://www.youtube.com/watch?v=Bw6RvCSsETI&ab_channel=SergeyNemchinskiy