Лекция 8

Основы ООП: основные принципы, наследование, инкапсуляция.

Абстракция

- **ООП** – объектно-ориентированное программирование.

- Хорошим примером абстракции в реальной жизни является описание должностей в компании или организации. Название должности – это одно, а обязанности каждой конкретной должности – это уже совсем другое.

- Представьте, что вы проектируете структуру своей будущей компании. Вы можете разделить обязанности секретаря: «раскидать» их по нескольким другим должностям. Можете разбить должность исполнительного директора на несколько независимых должностей: финансовый директор, технический директор, директор по маркетингу, директор по персоналу. Или, например, объединить должности офис-менеджера и рекрутера в одну.

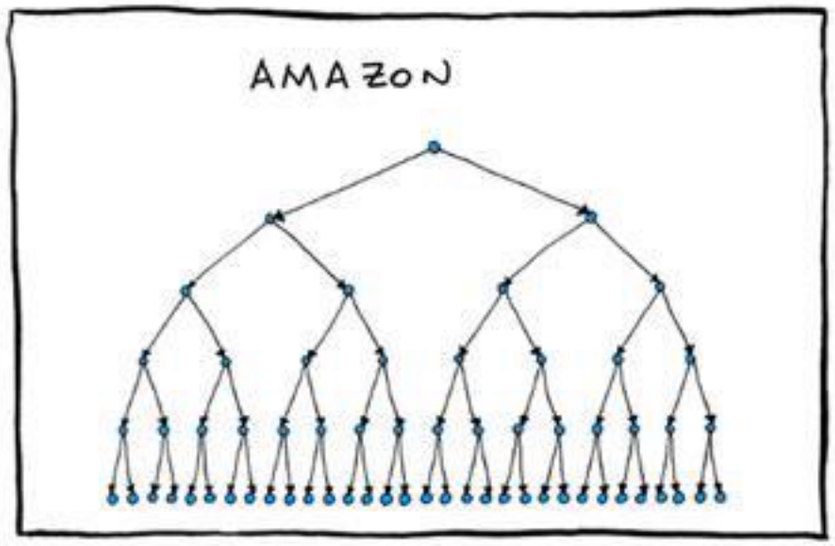
- Вы придумываете названия должностей в своей фирме, а потом «раскидываете» обязанности по этим должностям. Это и есть **абстракция** – разбиение чего-то большого, монолитного на множество маленьких составных частей.

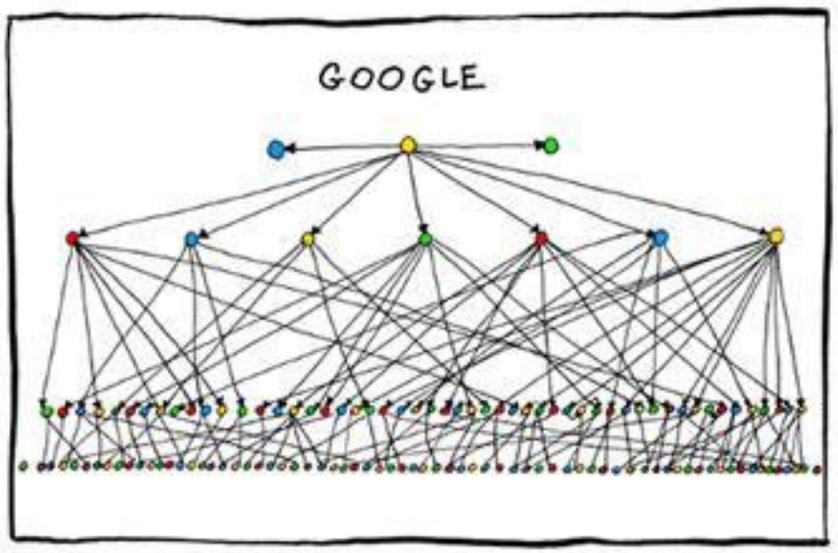
- С точки зрения программирования, абстракция — это, скажем так, **правильное разделение программы на объекты.**

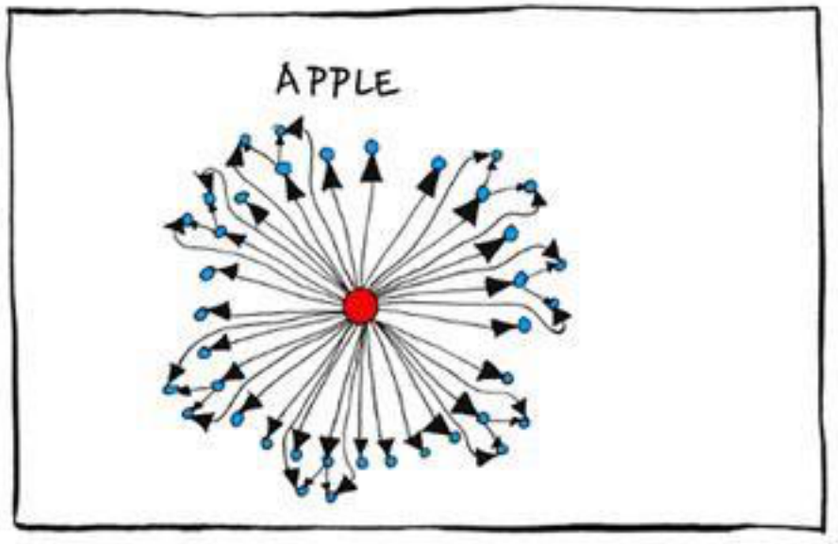
- Обычно любую большую программу можно десятками способов представить в виде взаимодействующих объектов. Абстракция позволяет отобрать главные характеристики и опустить второстепенные.

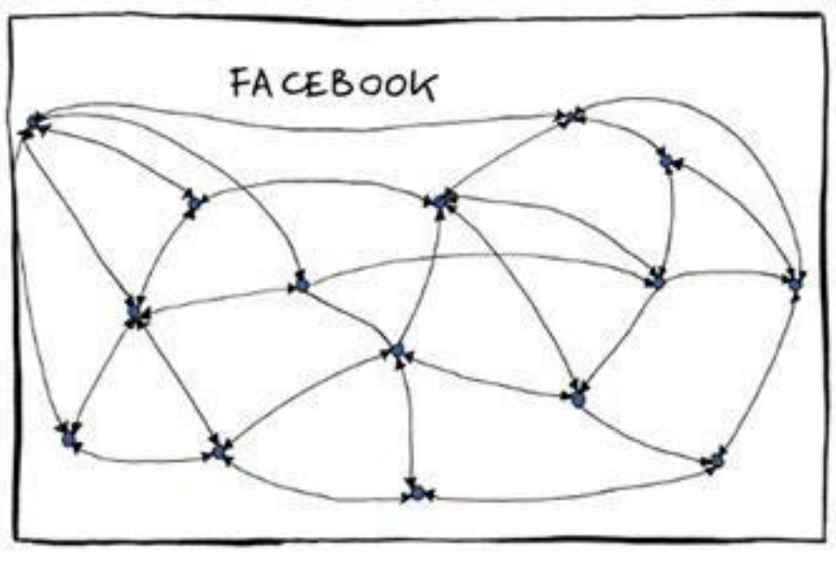
- Абстракция – это как стратегия в военном деле. Плохая стратегия – и никакой гениальной тактикой ситуацию уже не исправить.

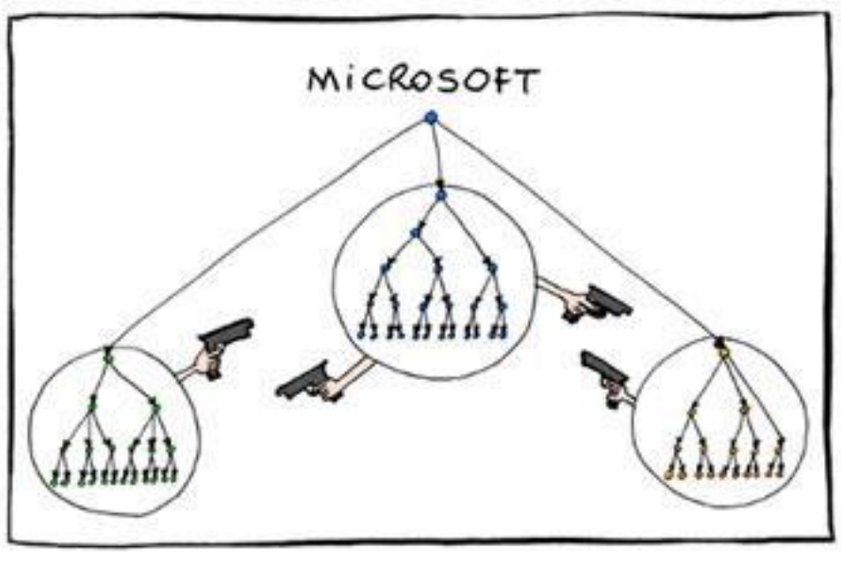
Абстракция

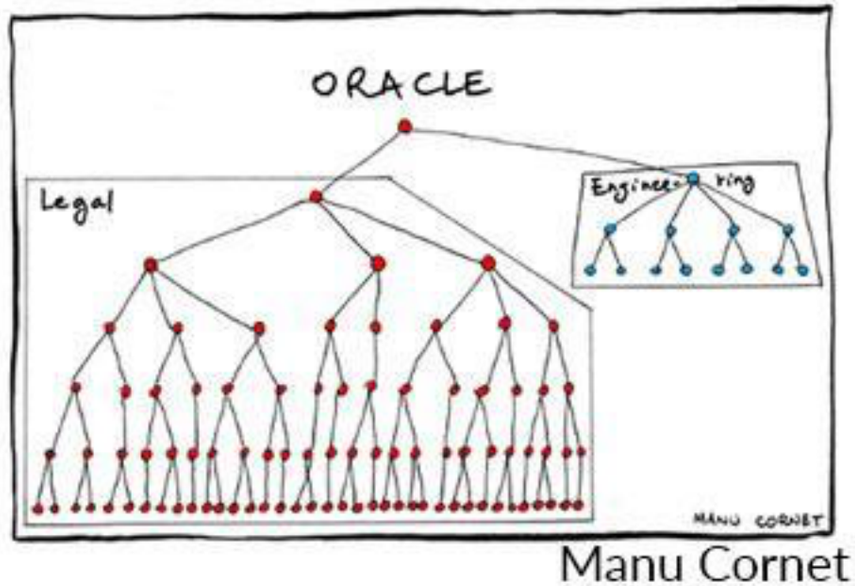








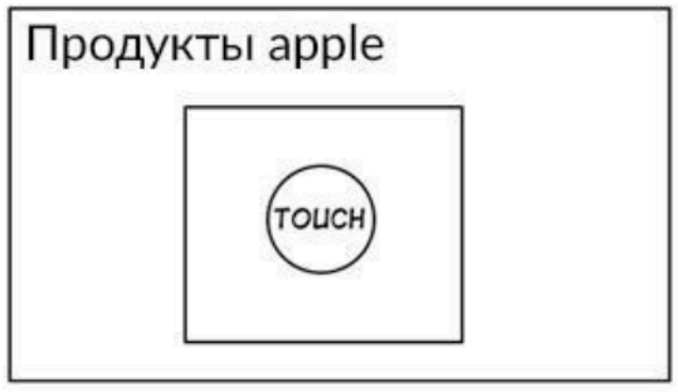


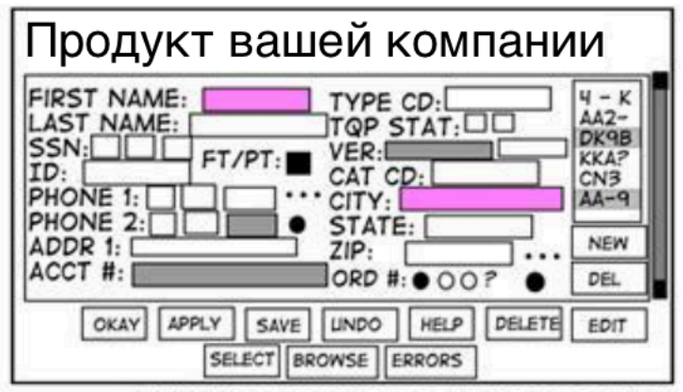


Manu Cornet

Инкапсуляция

- Цель инкапсуляции – улучшить качество взаимодействия вещей за счет упрощения их.





- Лучший способ упростить что-то – это скрыть все сложное от посторонних глаз. Например, если вас посадят в кабину Боинга, вы не сразу разберетесь, как им управлять:

- С другой стороны, для пассажиров самолета все выглядит проще: купил билет, сел в самолет, взлетели и приземлились. Вы можете с легкостью перелететь с континента на континент, обладая только навыками «купить билет» и «сесть на самолет». Все сложности в виде подготовки самолета к полету, взлету, посадки и различных внештатных ситуаций скрыты от нас. Не говоря уже о спутниковой навигации, автопилоте и диспетчерских центрах в аэропортах. И это упрощает нам жизнь.

- С точки зрения программирования, инкапсуляция – это «сокрытие реализации». Ваш класс может содержать сотни методов и реализовывать очень сложное поведение в различных ситуациях. Но мы можем скрыть от посторонних глаз все его методы (пометить модификатором private), а для взаимодействия с другими классами оставить всего пару-тройку методов (пометить их модификатором public). Тогда все остальные классы нашей программы будут видеть в этом классе всего три метода, и будут вызывать именно их. А все сложности будут скрыты внутри класса, как кабина пилотов от счастливых пассажиров.

Наследование

- У наследования есть две стороны. Сторона программирования и сторона реальной жизни. С точки зрения программирования, наследование — это специальное отношение между двумя классами. Но гораздо интереснее, что же такое наследование с точки зрения реальной жизни.

- Если бы нам понадобилось что-то создать в реальной жизни, то у нас есть два решения:
  1) создать нужную нам вещь с нуля, потратив кучу времени и сил.
  2) создать нужную нам вещь на основе уже существующей.

- Наиболее оптимальная стратегия выглядит так: берем существующее хорошее решение, немного его дорабатываем, подгоняем под свои нужды и используем.

- Если мы проследим историю возникновения человека, то окажется, что с момента зарождения жизни на планете прошли миллиарды лет. А если представить, что человек возник из обезьяны (на основе обезьяны), то прошла всего пара миллионов лет. Создание с нуля — дольше. Гораздо дольше.

- В программировании тоже есть возможность создавать один класс на основе другого. Новый класс становится потомком (наследником) уже существующего. Это очень выгодно, когда есть класс, который содержит 80%-90% нужных нам данных и методов. Мы просто объявляем подходящий класс родителем нашего нового класса, тогда в новом классе автоматически появляются все данные и методы класса-родителя.

Полиморфизм

- Полиморфизм — это понятие из области программирования. Оно описывает ситуацию, когда за одним интерфейсом скрываются разные реализации. Если постараться поискать его аналоги в реальной жизни, то одним из таких аналогов будет процесс управления машиной.

- Если человек может управлять грузовиком, то его можно посадить и за руль скорой, и за руль спорткара. Человек может управлять машиной вне зависимости от того, что это за машина, потому что все они имеют одинаковый интерфейс управления: руль, педали и рычаг коробки передач. Внутреннее устройство машин разное, но все они имеют одинаковый интерфейс управления.

- Если вернуться к программированию, то полиморфизм позволяет единообразно обращаться к объектам различных классов (обычно имеющих общего предка) — вещь, которую трудно переоценить. Ценность его тем выше, чем больше программа.

- **ООП** — это принципы. Внутренние законы. Каждый из них нас в чем-то ограничивает, давая взамен **большие преимущества**, когда программа вырастает до больших размеров.

Причины появления ООП

- История:

- Была небольшая компания, которая занималась доставкой товаров. В ней работало 5 человек. Один занимался финансами, второй работал на складе, третий выполнял доставку, четвертый руководил рекламой, а пятый управлял всем этим.

- Они были очень старательными, и все у них получалось. Компания имела хорошую репутацию и зарабатывала много денег. Но заказов с каждым годом было все больше, так что директору пришлось нанимать дополнительных сотрудников. Несколько на склад, несколько на доставку, еще одного кассира и рекламщика для расширения рынка.

- И тут начались проблемы. Людей стало больше, и они начали друг другу мешать.

- Маркетолог тратит все деньги на новую рекламную кампанию, и в кассе нет денег на закупку товара, который надо срочно отправлять.

- На складе есть 10 коробок с новенькими гипердвигателями, которые поставляют раз в месяц. Курьер поехал отвозить один гипердвигатель, и заказ на 10 гипердвигателей от другого клиента вынужден ждать еще месяц. Первый курьер просто не знал о другом заказе, который выполняет второй курьер.

- Новый помощник директора отправляет курьера на вертолете для закупки товара, и все остальные ждут, пока появится доступный вертолет. Есть куча срочных доставок, но этот помощник заведует только закупками и **старается хорошо выполнять свою работу.** Чем лучше человек выполнял свои обязанности, тем больше он мешал остальным.

- Пытаясь проанализировать ситуацию директор понял, что такие важные ресурсы, как вертолет, наличность и товар расходуются не оптимально, а по принципу «кто первый встал — того и тапки». Любой мог взять нужный всем ресурс для своей работы, поставив при этом под удар остальных сотрудников, да и всю компанию в целом.

- Нужно было что-то делать, и директор решил разделить монолитную компанию на несколько отделов. Появился отдел доставки, отдел маркетинга, отдел закупок, финансовый отдел и отдел запасов. Теперь уже никто не мог просто так взять вертолет. Директор отдела доставки получал всю информацию о доставках и выдавал вертолет тому курьеру, чья доставка была выгоднее для компании. Склад тоже не разрешал любому курьеру взять любой товар, а контролировал этот процесс. Финансовый отдел мог не дать денег на маркетинг, если знал, что скоро будет закупка. У каждого отдела было одно публичное лицо — его начальник. Внутреннее устройство каждого отдела было его внутренним делом. Если курьер хотел получить товар, он шел к начальнику склада, а не на склад. Если появлялась новая заявка, ее получал директор отдела доставки (**public** person), а не курьер (**private** person).

- Другими словами, директор объединил в группы (отделы) ресурсы и действия над ними, а также запретил другим вмешиваться во внутреннюю структуру отделов. Контактировать можно было строго с определенным лицом.

- С точки зрения ООП, это не что иное, как разбиение программы на объекты. Монолитная программа, состоящая из функций и переменных, превращается в программу, состоящую из объектов. А объекты содержат в себе переменные и функции.

- А проблема была в том, что любой сотрудник мог бесконтрольно работать с любым ресурсом и отдавать команды любому человеку.

- Ввели небольшое ограничение, но получили больше порядка. А также смогли лучше контролировать все это.

Преимущества ООП

- Программы больше напоминают не строения, а животных. **Их не строят, их выращивают.** Разработка — это постоянные изменения. В строительстве вы можете иметь хороший план и четко ему следовать. В случае с разработкой программ – это не так.

- Очень часто что-то нельзя сделать тем способом, который вы себе наметили, и приходится многое переделывать. Еще чаще меняются требования заказчика.

- Если заказчик проекта дал очень точную его спецификацию, тогда нужно взглянуть на ситуацию во времени. Успех продукта приведет к тому, что заказчик захочет выпустить его новую версию, а затем еще и еще. И, конечно, нужно будет всего лишь добавить «**небольшие изменения**» в уже существующий продукт. Поэтому разработка продукта – это последовательность постоянных изменений. Только масштаб времени разный. Каждая новая версия может выходить раз в неделю, раз в месяц или раз в полгода.

- Внутреннюю структуру продукта нужно поддерживать в таком состоянии, которое позволит внести значительные (и не очень) изменения с минимальными переделками.

- Мы уже говорили, что программа состоит из объектов, которые взаимодействуют между собой. Давайте нанесем на доску все объекты нашей программы, обозначив их жирными точками. И проведем от каждого объекта (точки) стрелочки ко всем объектам (точкам), с которыми он взаимодействует.

- Теперь мы будем объединять объекты (точки) в группы. Точки должны быть объединены в группу, если связи между ними гораздо интенсивнее, чем с остальными точками. Если большинство стрелочек от точки идет к точкам ее же группы, тогда разбиение на группы произошло правильно. Точки внутри одной группы мы будем называть сильно связанными, а точки из разных групп — слабо связанными.

- Это называется «**принцип слабой связности**». Программа разбивается на несколько частей, часто слоев, логика которых сильно завязана на их внутреннее устройство и очень слабо на другие слои/части. Обычно взаимодействие слоев очень регламентировано. Один слой может обращаться ко второму и использовать только небольшую часть его классов.

- Это приводит к тому, что мы можем реорганизовать отдел, повысить его эффективность, нанять в него еще больше людей, но если мы не изменим протокол взаимодействия других отделов с нашим, то все сделанные изменения останутся локальными. Никому не придется переучиваться. Не придется переделывать всю систему. Каждый отдел может заниматься такой внутренней оптимизацией, если общие механизмы взаимодействия выбраны удачно.

- Если они выбраны неудачно, то «**запас изменений**» быстро иссякнет и придется переделывать всю систему. Такое приходится делать время от времени. Нельзя предугадать, что будет в будущем, но можно свести количество таких переделок к минимуму.

Преимущества наследования

- Предположим вам нужно написать очень сложный класс. Писать с нуля долго, потом еще долго все тестировать и искать ошибки. Зачем идти самым сложным путем? Лучше поискать – а нет ли уже такого класса?

- Предположим, вы нашли класс, который своими методами реализует 80% нужной вам функциональности. Вы можете просто скопировать его код в свой класс. Но у такого решения есть несколько минусов:
  1) Найденный класс уже может быть скомпилирован в байт-код, а доступа к его исходному коду у вас нет.
  2) Исходный код класса есть, но вы работаете в компании, которую могут засудить на пару миллиардов за использование даже 6 строчек чужого кода. А потом она засудит вас.
  3) Ненужное дублирование большого объема кода. Кроме того, если автор чужого класса найдет в нем ошибку и исправит ее, у вас эта ошибка останется.

- Есть решение потоньше, и без необходимости получать легальный доступ к коду оригинального класса. В Java вы можете просто объявить тот класс родителем вашего класса. Это будет эквивалентно тому, что вы добавил код того класса в код своего. В вашем классе появятся все данные и все методы класса-родителя.

- Наследование можно использовать и для других целей. Допустим, у вас есть десять классов, которые очень похожи, имеют совпадающие данные и методы. Вы можете создать специальный базовый класс, вынести эти данные (и работающие с ними методы) в этот базовый класс и объявить те десять классов его наследниками. Т.е. указать в каждом классе, что у него есть класс-родитель — данный базовый класс.

- Также как преимущества абстракции раскрываются только рядом с инкапсуляцией, так и преимущества наследования гораздо сильнее при использовании полиморфизма.

- Предположим, мы пишем программу, которая играет в шахматы с пользователем, тогда нам понадобятся классы для фигур:
  Король, Ферзь, Слон, Конь, Ладья и Пешка.

- Данные, которые необходимо хранить в этих классах:
  Координаты x и y, а также ее ценность (worth). Ведь некоторые фигуры ценнее других.

- Отличия в том, как они ходят, фигуры. В поведении.

- Вот как можно было бы описать их в виде классов:

```
class King
{
int x;
int y;
int worth;
void kingMove()
{
//код, решающий,
//как пойдет король
}
}
```

```
class Queen
{
int x;
int y;
int worth;
void queenMove()
{
//код, решающий,
//как пойдет ферзь
}
}
```

```
class Rook
{
int x;
int y;
int worth;
void rookMove()
{
//код, решающий,
//как пойдет ладья
}
}
```

```
class Knight
{
int x;
int y;
int worth;
void knightMove()
{
//код, решающий,
//как пойдет конь
}
}
```

```
class Bishop
{
int x;
int y;
int worth;
void bishopMove()
{
//код, решающий,
//как пойдет слон
}
}
```

```
class Pawn
{
int x;
int y;
int worth;
void pawnMove()
{
//код, решающий,
//как пойдет пешка
}
}
```

- А вот, как можно было бы сократить код с помощью наследования. Мы могли бы вынести одинаковые методы и данные в общий класс. Назовем его ChessItem. Объекты класса ChessItem не имеет смысла создавать, так как ему не соответствует ни одна шахматная фигура, но от него было бы много пользы:

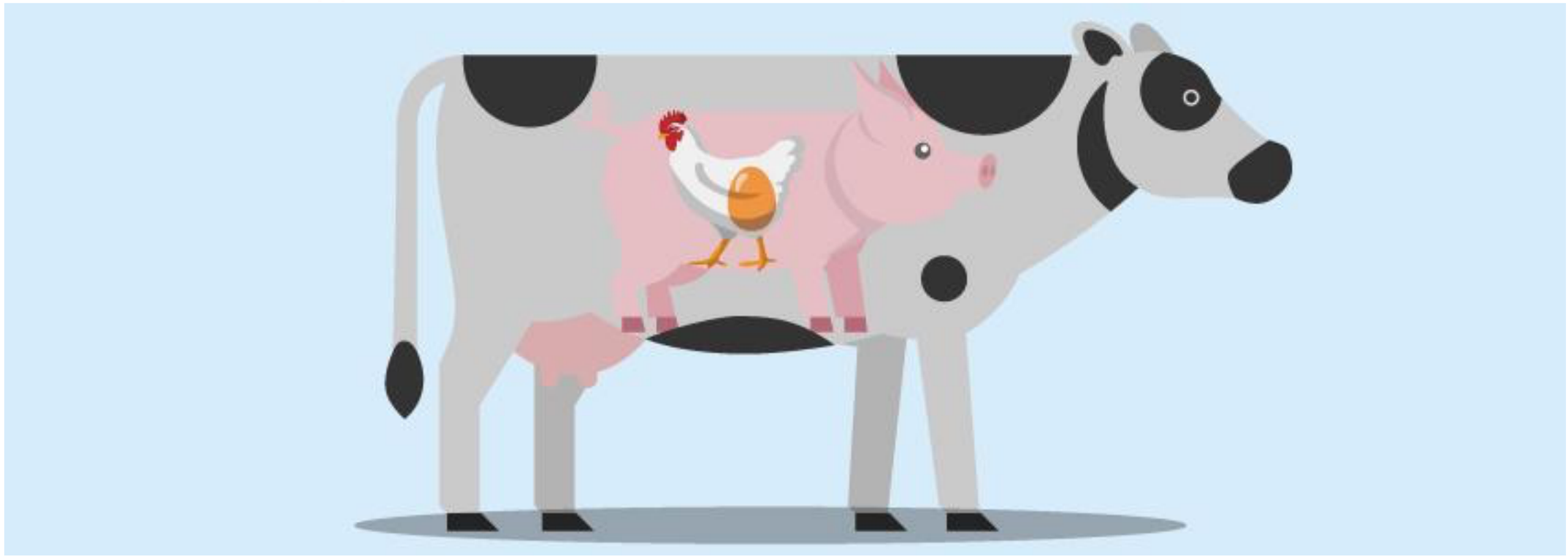
class King **extends** ChessItem
{
void kingMove()
{
//код, решающий,
//как пойдет король
}
}
	class Queen **extends** ChessItem
	{
	void queenMove()
	{
	//код, решающий,
	//как пойдет ферзь
	}
	}
		class Rook **extends** ChessItem
		{
		void rookMove()
		{
		//код, решающий,
		//как пойдет ладья
		}
		}

	class ChessItem
	{
	int x;
	int y;
	int worth;
	}	

class Knight **extends** ChessItem
{
void knightMove()
{
//код, решающий,
//как пойдет конь
}
}
	class Bishop **extends** ChessItem
	{
	void bishopMove()
	{
	//код, решающий,
	//как пойдет слон
	}
	}
		class Pawn **extends** ChessItem
		{
		void pawnMove()
		{
		//код, решающий,
		//как пойдет пешка
		}
		}

- Особенно много преимуществ мы получаем, когда в проекте тысячи различных объектов и сотни классов. Тогда правильно подобранными классами можно не только существенно упростить логику, но и сократить код в десятки раз.

- Для унаследования класса нужно после объявления нашего класса указать ключевое слово extends и написать имя родительского класса. Унаследоваться можно только от одного класса.



- На картинке видно «корову», унаследованную от «свиньи». «Свинья» унаследована от «курицы», «курица» от «яйца». Только один родитель! Такое наследование не всегда логично. Но если есть только свинья, а очень нужна корова, программист зачастую не может устоять перед желанием сделать «корову» из «свиньи».

- Множественного наследования классов в Java нет: класс может иметь только одного класса-родителя.

Инкапсуляция

Внутренняя реализация **скрыта**,
шанс что-то сломать – минимальный

Внутренняя реализация **открыта**,
любое вмешательство опасно

Преимущества инкапсуляции

- **1) Валидное внутреннее состояние.**

- В программах часто возникают ситуации, когда несколько классов, взаимодействуют с одним и тем же объектом. В результате их совместной работы нарушается целостность данных внутри объекта — объект уже не может продолжить нормально работать.

- Поэтому объект должен следить за изменениями своих внутренних данных, а еще лучше – проводить их сам.

- Если мы не хотим, чтобы какая-то переменная класса менялась другими классами, мы объявляем ее **private**, и тогда только методы её же класса смогут получить к ней доступ. Если мы хотим, чтобы значения переменных можно было только читать, но не изменять, тогда нужно добавить **public getter** для нужных переменных.

- Например, мы хотим, чтобы все могли узнать количество элементов в нашей коллекции, но никто не мог его поменять без нашего разрешения. Тогда мы объявляем переменную **private int count** и метод **public getCount()**.

- Правильное использование инкапсуляции гарантирует, что ни один класс не может получить прямой доступ к внутренним данным нашего класса и, следовательно, изменить их без контроля с нашей стороны. Только через вызов методов того же класса, что и изменяемые переменные.

- Лучше исходить из того, что другие программисты всегда будут использовать ваши классы самым удобным для них образом, а не самым безопасным для вас (для вашего класса). Отсюда и ошибки, и попытки заранее избавиться от них.

- **2) Контроль передаваемых аргументов.**

- Иногда нужно контролировать аргументы, передаваемые в методы нашего класса. Например, наш класс описывает объект «человек» и позволяет задать дату его рождения. Мы должны проверять все передаваемые данные на их соответствие логике программы и логике нашего класса. Например, не допускать 13-й месяц, дату рождения 30 февраля и так далее. (Во-первых — это может быть ошибка ввода данных от пользователя. Во-вторых, прежде чем программа будет работать как часы, в ней будет много ошибок. Например, возможна такая ситуация.)

- Программист пишет программу для определения людей, у которых день рождения послезавтра. Например, сегодня 3 марта. Программа добавляет к текущему дню месяца число 2 и ищет всех, кто родился 5 марта. Вроде бы все верно.

- Вот только, когда наступит 30 марта программа не найдет никого, т.к. в календаре нет 32 марта. В программе становится гораздо меньше ошибок, когда в методы добавляют проверку переданных данных.

- **3) Минимизация ошибок при изменении кода классов.**

- Представим, что мы написали один очень полезный класс, когда участвовали в большом проекте. Он так всем понравился, что другие программисты начали использовать его в сотнях мест в своем коде.

- Класс оказался настолько полезен, что вы решили его улучшить. Но если вы удалите какие-то методы этого класса, то код десятков людей перестанет компилироваться. Им придется срочно все переделывать. И чем больше переделок, тем больше ошибок. Вы поломаете кучу сборок, и вас будут ненавидеть. :(

- А когда мы меняем методы, объявленные как private, мы знаем, что нигде нет ни одного класса, который вызывал бы эти методы. Мы можем их переделать, поменять количество параметров и их типы, и зависимый код будет работать дальше. Ну, или как минимум, компилироваться.

- **4) Задаем способ взаимодействия нашего объекта со сторонними объектами.**

- Мы можем ограничить некоторые действия, допустимые с нашим объектом. Например, мы хотим, чтобы объект можно было создать только в одном экземпляре. Даже если его создание происходит в нескольких местах проекта одновременно. И мы можем сделать это благодаря инкапсуляции.

- Инкапсуляция позволяет добавлять дополнительные ограничения, которые можно превратить в дополнительные преимущества. Например, класс String реализован как immutable (неизменяемый) объект. Объект класса String неизменяем с момента создания и до момента смерти. Все методы класса String (remove, substring, ...), возвращают новую строку, абсолютно не изменяя объект, у которого они были вызваны.

Задачи

1. Отредактировать **два класса**: **Horse** (**лошадь**) и **Pegasus** (**пегас**). Унаследовать пегаса от лошади.

```
public class Solution {
   public static void main(String[] args) {
   }

   public class Horse {

   }

   public class Pegasus {

   }
}
```

2. Отредактировать три класса: Pet (домашнее животное), Cat (кот) и Dog (собака). Унаследовать кота и собаку от животного.

```
public class Solution {
   public static void main(String[] args) {
   }

   public class Pet {

   }

   public class Cat {

   }

   public class Dog {

   }
}
```

3. Скрыть все внутренние переменные **класса Cat**.

```
public class Solution {
   public static void main(String[] args) {
   }

   public class Cat {
      public String name;
      public int age;
      public int weight;

      public Cat() {
      }

      public Cat(String name, int age, int weight) {
         this.name = name;
         this.age = age;
         this.weight = weight;
      }
   }
}
```

4. Написать шесть классов: Animal (животное), Cow (корова), Pig (свинья), Sheep (овца), Bull (бык) и Goat (козел). Унаследовать корову, свинью, овцу, быка и козла от животного.

```
public class Solution {
   public static void main(String[] args) {
   }

   public class Animal {
   }

   public class Cow {
   }

   public class Pig {
   }

   public class Sheep {
   }

   public class Bull {
   }

   public class Goat {
   }

}
```