Преимущества ООП

- Сборка из готовых частей против изготовления их самому.
- Все рассматривается как объект.

Следствия:

- Упрощение разработки.
- Возможность создания расширяемых систем.
- Обработка разнородных структур данных, изменение поведения на этапе выполнения, наследование поведения и свойств родительских классов.
- Легкость модернизации с сохранением совместимости.

Недостатки ООП

- Неэффективность на этапе выполнения.
- Неэффективность при распределении памяти.
- Излишняя избыточность.
- Сложность проектирования.

Структурное программирование и ООП

Этапы структурного программирования:

- Формирование задачи.
- Разбивка на мелкие подзадачи.
- Создание набора функций (или алгоритмов) для решения задач.
- Задание подходящего способа хранения данных.

Алгоритмы стоят на первом месте, а структуры данных на втором. Сначала решают, как манипулировать данными; затем, какую структуру применить для организации этих данных, чтобы работать с ними было легче.

Структурное программирование и ООП

ООП:

- Формирование задачи
- Определение сущностей (объектов)
- Разработка классов (данные и алгоритмы для их обработки)

Существенное отличие заключается в том, что при объектно-ориентированном программировании в проекте сначала выделяются классы, и лишь затем определяются их методы.

Структурное программирование и ООП

ООП:

- Высокая производительность разработки: каждый объект предназначен для решения своей задачи.
- При разработке больших проектов классы имеют преимущество: предоставляют удобный механизм кластеризации методов.

Процедурное (структурное) программирование:

- Хорошо подходит для решения небольших задач.

Словарь ООП

Класс - это шаблон, по которому будет сделан объект. Он говорит виртуальной машине, как сделать объект заданного типа. Каждый объект, созданный из этого класса, может иметь собственные значения для переменных экземпляра класса

Создание экземпляра (instance) класса - конструирование объекта на его основе.

Все коды, которые создаются в языке Java, находятся внутри классов.

Словарь ООП

Инкапсуляция (сокрытие данных) - объединение данных и операций над ними в одном месте и сокрытие данных от пользователя объекта.

Данные в объекте называются **полями экземпляра**.

Функции и процедуры, выполняющие операции над данными, — **методами объекта**.

Текущее состояние объекта - множество значений полей экземпляра. Применение любого метода к какому-нибудь объекту может изменить его состояние.

Ключевые свойства объектов

Поведение объекта - что с ним можно делать и какие методы к нему можно применять.

Состояние объекта - как этот объект реагирует на применение методов.

Сущность объекта - чем данный объект отличается от других.

Ключевые свойства объектов

Все объекты, являющиеся экземплярами одного и того же класса, ведут себя одинаково. Поведение объекта определяется методами, которые можно вызвать.

- Каждый объект сохраняет информацию о своем состоянии.
- Состояние объекта может измениться, но только в результате вызовов методов.
- Если состояние объекта изменилось вследствие иных причин, значит, инкапсуляция нарушена.

С чего начинать?

- В традиционной процедурно-ориентированной программе выполнение начинается с функции **main**.

- При разработке объектно-ориентированной системы следует определить классы и добавить к ним методы

- Классы представляют собой аналоги имен существительных в описании решаемой задачи, а методы - используемых при этом глаголов.

Пример

При описании системы обработки заказов используются следующие имена существительные:

предмет, заказ, адрес доставки, оплата, счет.

Эти имена существительные соответствуют классам **Item**, **Order, Address, Payment, Bill**.

Используя глаголы, определяем действия, выполняемые объектами: предметы заказываются, заказы выполняются или отменяются и т.д.

Отношения между классами

Между классами существуют следующие отношения:

- Зависимость (“использует”)
- Агрегация (“содержит”, “является частью”)
- Композиция (“содержит”, “является частью”)
- Ассоциация
- Наследование (“является”)

UML

Для изображения диаграмм класса, описывающих отношения между классами, часто используют символы языка **UML** (Unified Modeling Language)

Зависимость

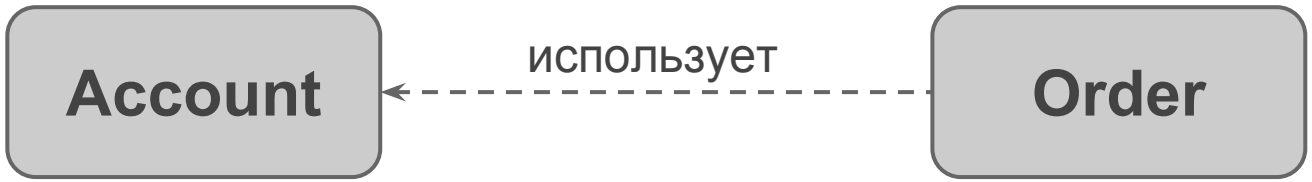
Наследование

Агрегирование

Композиция

Зависимость

- Класс зависит от другого класса, если его методы манипулируют объектами этого класса.
- Количество взаимозависимых классов следует минимизировать (ограничить количество связей).



Различия между агрегированием и композицией

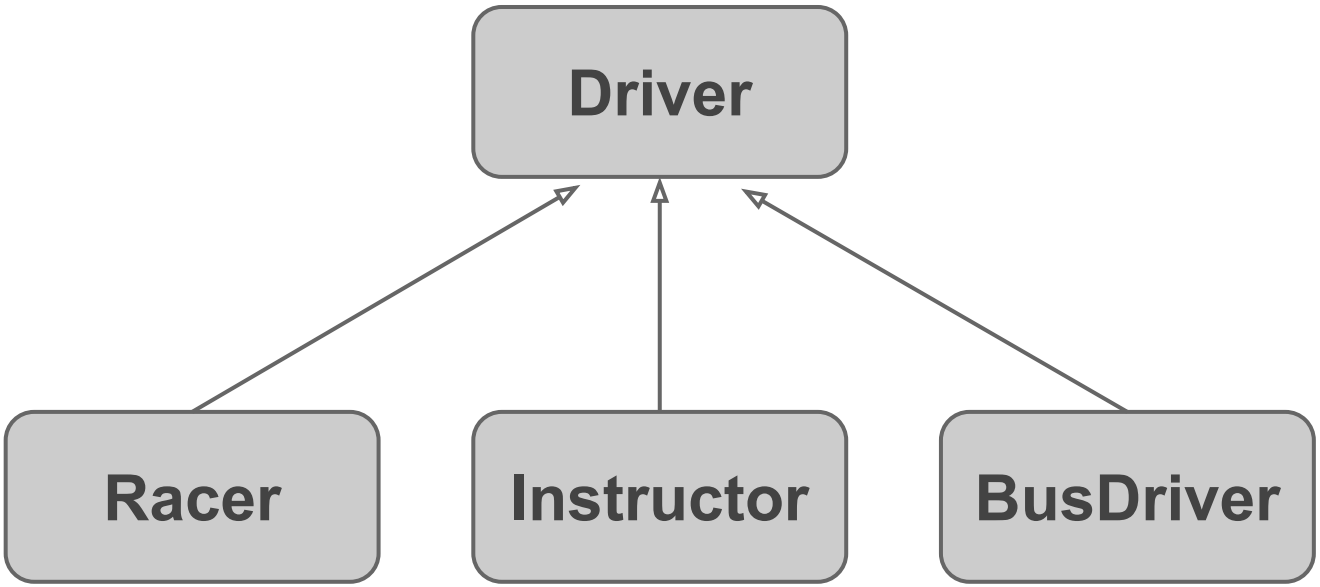
- **Агрегирование** - целое содержит свою составную часть, однако время их жизни не связано (например, составная часть передается через параметры конструктора)

- **Композиция** - целое явно контролирует время жизни своей составной части (часть не существует без целого)

Наследование

Выражает отношение между конкретным и более общим классом.

Если класс A расширяет класс B, то говорят, что класс A наследует свойства класса B, имея, кроме них, еще и дополнительные возможности.



Объекты и объектные переменные

Чтобы работать с объектами, их нужно сначала создать и задать исходное состояние. Затем у этих объектов можно вызывать методы.

Конструктор

- Специальный метод, предназначенный для создания и инициализации объектов.
- Имя конструктора всегда совпадает с именем класса.
- Не имеет возвращаемого типа.
- Может быть переопределен.
- Существует конструктор по-умолчанию.

Примеры

- Создание нового объекта, который инициализируется текущими датой и временем:

  ```
  new Date();
  ```

- Объект можно передать методу:

  ```
  System.out.println (new Date());
  ```

- Метод можно применить ко вновь созданному объекту:

  ```
  String dateStr = new Date().toString();
  ```

Примеры

Чтобы идентифицировать объект для дальнейшего использования, нужно его присвоить переменной:

```
Date birthday = new Date();
```

Объекты и объектные переменные

```
Date deadline;   // Переменная не ссылается ни на один объект
```

- Определяет **объектную переменную** *deadline*, которая может ссылаться на объекты типа *Date*.

- Переменная *deadline* объектом не является и не ссылается ни на один объект, поэтому ни один метод класса *Date* с помощью этой переменной вызывать пока нельзя:

```
s = deadline.toString();
```

Ошибка!

Инициализация объектных переменных

Два варианта:

- deadline = **new** Date();

- Ссылка на существующий объект:

  deadline = birthday;

  Теперь обе переменные ссылаются на один и тот же объект.

Объектная переменная фактически не содержит никакого объекта. Она лишь ссылается на объект.

“Уничтожение” объекта

Объектной переменной можно явно присвоить ссылку **null**, чтобы отметить тот факт, что эта переменная пока не ссылается ни на один объект:

deadline = **null**;

Сборка мусора

- При создании объекта для него выделяется область памяти под названием “куча” (heap).
- Куча управляется специальным процессом - сборщиком мусора.
- Если на объект нет ссылок, память, занятая им, освобождается.
- В случае нехватки памяти для создания очередного объекта виртуальная машина находит и удаляет недостижимые объекты.
- Момент запуска сборщика не определен, программист не знает, когда это произойдет.
- Процесс сборки мусора можно инициировать принудительно:

  System.gc();

Деструкторы

Если объект использует какие-либо ресурсы, например файлы, то при завершении работы с объектом, их нужно освобождать.

В C++ для этого существуют деструкторы - специальные методы, которые вызываются автоматически при удалении объекта.

В Java нет деструкторов, вместо них применяется механизм автоматической сборки мусора.

Еще про уничтожение объекта

Аналогом деструктора в Java иногда считают метод *finalize*.

Этот метод можно добавить в любой класс и он будет вызван при уничтожении объекта сборщиком мусора.

Однако, если нужно использовать ресурсы повторно, полагаться на метод *finalize* не следует.

Ключевое слово this

- Ссылка на текущий объект.

- Используется, чтобы отличить поля объекта от параметров метода.

```
public class Test {
    private int number;

    public Test(int number) {
        this.number = number;
    }
}
```

Ключевое слово this

- Может использоваться для вызова конструктора класса из другого конструктора этого же класса.

```
public class Test {
    public Test(int number) {
        this.number = number;
    }

    public Test() {
        this(10);
    }
}
```

Модификаторы доступа

- **public**

  Доступ отовсюду, из любых других классов.

- **private**

  Доступ только в контексте класса.

- **package, default, none**

  Доступ для самого класса и классов в том же пакете.

- **protected**

  Доступ в пределах класса, пакета и классов-наследников.

Getters and Setters

Специальные методы, позволяющие получить данные, доступ к которым напрямую ограничен.

```
public class Test {
    private int number;

    public void setNumber(int number) {
        this.number = number;
    }
    public int getNumber() {
        return number;   // или this.number
    }
```

Инициализация полей и переменных по умолчанию

Если значение поля не задано явно, то ему присваивается значение по умолчанию:

- числам - нули;
- boolean - false;
- ссылкам - null.

Локальные переменные всегда должны инициализироваться явно.

Инициализация полей и переменных по умолчанию

```
public class Test {
    private int number;

    public void doSome() {
        int undefinedNumber;

        System.out.println(number);
        System.out.println(undefinedNumber);
    }
```

Создание собственных классов

Чтобы написать полностью законченную программу, нужно объединить классы, один из которых имеет метод **main**.

Проектируя класс, думайте об объектах, которые будут созданы на его основе. Думайте:

- о вещах, которые объект знает;
- о вещах, которые объект делает.

Ключевые моменты

- Объектно ориентированное программирование позволяет расширять приложение, не затрагивая проверенный ранее и работающий код.

- Весь код в Java находится внутри классов.

- Класс описывает, как создавать объект определенного типа. Класс — это что-то вроде шаблона.

- Объект кое-что знает и умеет делать.

- Сведения об объекте называются переменными (полями) экземпляра. Они определяют состояние объекта.

Ключевые моменты

- Действия объекта называются методами. Они обуславливают поведение объекта.

- Для каждого класса можно предусмотреть отдельный проверочный класс, который будет использоваться для создания объектов нового класса.

- Класс может наследовать поля и методы от более абстрактного родительского класса.

- Работающая программа на языке Java — это не что иное, как набор объектов, которые общаются между собой.

Что хорошего в ООП

- Помогает разрабатывать программы более естественным путем. Предметы имеют возможность развиваться.

- Если нужно добавить функциональность, не приходится возвращаться к коду, который уже протестировал.

- Данные и методы, которые оперируют этими данными, находятся в одном месте.

- Возможность повторного использования кода в других приложениях. Создавая новый класс, можно сделать его достаточно гибким, чтобы применять и в других проектах.

Пакеты

- Объединяют классы в коллекции, в зависимости от функционала.
- Облегчают организацию работы и позволяют отделить вашу собственную работу от классов, разработанных другими программистами.
- Предотвращают конфликты имен за счет создания локальных областей видимости.
- Предоставляют еще один уровень безопасности: можно ограничить доступ к своему коду только теми классами, которые находятся в одном пакете.

Пакеты

Чтобы обеспечить абсолютную уникальность имени пакета, рекомендуется использовать доменное имя вашей компании, записанное в обратном порядке:

com.netcracker

com.netcracker.ejb.ipplanner.IpRange

Вложенные пакеты

- Пакеты можно делить на подпакеты.

- Единственная цель вложенных пакетов - гарантия уникальности имен. С точки зрения компилятора между вложенными пакетами нет никакой связи.

- Например, пакеты **java.util** и **java.util.jar** никак не связаны друг с другом. Каждый из них представляет собой независимую коллекцию классов.

Пакеты

Членами пакетов являются:

- классы;
- интерфейсы;
- вложенные пакеты;
- дополнительные файлы ресурсов.

Пакеты

Чтобы использовать класс из пакета (если это не *java.lang*), нужно указать его полное имя. Существуют два варианта использования:

- Поместить оператор *import* в начале исходного файла:

```
import java.util.ArrayList;
ArrayList numbers = new ArrayList();
```

- Указать полное имя класса в любом месте своего кода, каждый раз, когда вы его используете:

```
java.util.ArrayList numbers = new java.util.ArrayList();
```

Пакеты

Можно импортировать как один конкретный класс, так и весь пакет:

- импортировать все классы из пакета java.util:

  import java.util.* ;

- импортировать отдельный класс из пакета:

  import java.util.Date;

Импорт всех классов из пакета не сказывается на размере кода.

Оператор import со звездочкой * можно применять для импортирования только одного пакета.

Добавление класса в пакет

1. Создать каталог в файловой системе, имя которого соответствует полному имени пакета:

   com.netcracker.test : com\netcracker\test

2. Указать имя пакета в начале исходного файла:

   package com.netcracker.test;

Если оператор package в исходном файле не указан, то классы, описанные в этом файле, принадлежат пакету по умолчанию (default package). Пакет по умолчанию не имеет имени.

Область видимости пакета

Если ни один модификатор доступа не указан, то класс, метод или поле класса являются доступными всем методам других классов в том же самом пакете.

Наследование

- Основная идея: новые классы создаются из существующих.

- Методы и поля существующего класса используются повторно (наследуются) вновь создаваемым классом.

- В новый класс добавляются дополнительные поля и методы для изменения его параметров или поведения.

Признак наследования одного класса другим - отношение типа “является” (IS-A).

Наследование

Ключевое слово **extends** означает, что создается новый класс, производный от существующего.

- Существующий класс - суперкласс, базовый или родительский.
- Вновь создаваемый класс - подкласс, производный или дочерний.

Суперкласс не имеет превосходства над своим подклассом и не обладает более широкими функциональными возможностями. Наоборот, подкласс шире, чем суперкласс.

Отметьте отношения, которые имеют смысл

- ❑ Печь extends Кухня
- ❑ Гитара extends Инструмент
- ❑ Человек extends Служащий
- ❑ Феррари extends Двигатель
- ❑ Яичница extends Еда
- ❑ Гончая extends Домашнее животное
- ❑ Контейнер extends Ёмкость
- ❑ Металл extends Титан.
- ❑ Grateful Dead extends Музыкальная группа
- ❑ Блондинка extends Ум
- ❑ Напиток extends Мартини

Ключевые моменты

- Дочерний класс **расширяет** класс-родитель.

- Дочерний класс наследует все *public* и *protected* переменные и методы своего родителя, но не наследует его приватные переменные и методы.

- Унаследованные методы могут быть переопределены. Переменные экземпляра не могут быть переопределены (хотя их можно заново объявить в классе, но этого почти никогда не требуется).

- Используйте проверку на соответствие (IS-A), чтобы подтвердить правильность своей иерархии наследования. Если X расширяет Y, значит, X — это Y.

Ключевые моменты

- Отношение IS-A работает только в одну сторону. Пиво — это напиток, но не все напитки — пиво.

- Когда метод, переопределенный в дочернем классе, вызывается из экземпляра дочернего класса, в ответ будет выполнена переопределенная версия (которая находится в самом низу иерархии).

- Если класс B расширяет A и C расширяет B, то класс B — это A, класс C — это B, а класс C — это A.

Правильно

- Использовать наследование для класса, который представляет собой более специфичный вид родительского класса.

  Пример: *Кот - это Животное.*

- Применяйте наследование, если у вас есть поведение (реализованный код), которое должно быть общим для разных классов одного типа.

  Пример: *вычисление площади Квадрата и Круга имеет смысл добавить в родительский класс Фигура.*

Неправильно

- Не применяйте наследование только для того, чтобы иметь возможность повторно использовать код из других классов, если отношения между родительским и дочерним классами нарушают хотя бы одно из двух правил, приведенных ранее.

- Не используйте наследование, если дочерний и родительский классы не проходят проверку на соответствие.

  Пример: *Кот - это Животное (верно)*

  Животное - это Кот (не имеет смысла)

Ключевое слово super

- Сообщает компилятору, что нужно вызвать метод суперкласса.
- Не ссылается на объект. Его нельзя присвоить другой объектной переменной.

В конструкторе:

```
super(name, salary, hireDay);
```

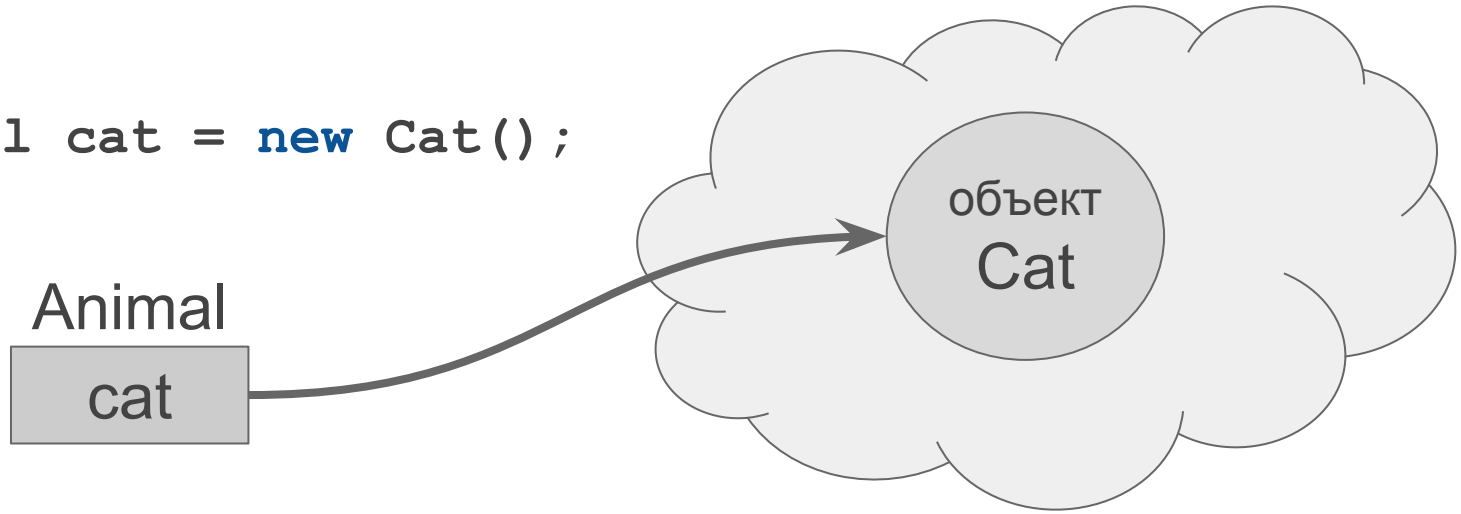
Вызов конструктора суперкласса с параметрами должен быть первым оператором.

Полиморфизм

- Возможность работать с несколькими типами так, как будто это один и тот же тип и в то же время поведение каждого типа будет уникальным в зависимости от его реализации.

- Объектная переменная ссылается на объекты, имеющие разные фактические типы. То есть тип ссылки может быть родительским для типа самого объекта.

```
Animal cat = new Cat();
```



Полиморфизм

- При объявлении ссылочной переменной ей можно присвоить любой объект, который проходит проверку на соответствие для типа этой ссылки. Иными словами, все, что расширяет тип объявленной ссылочной переменной, может быть ей присвоено.

- Аргументы методов также могут быть полиморфными:

```
public void moveAnimal(Animal animal) {
    animal.move(); }

Cat murzik = new Cat();
animal.move(murzik);
```

Связывание

- Статическое связывание - выбор вызываемого метода во время компиляции.

- Автоматический выбор нужного метода во время выполнения программы называется динамическим связыванием.

Полиморфизм

- Благодаря полиморфизму можно писать код, который не придется менять с появлением в программе новых типов дочерних классов.

Предотвращение наследования

Иногда наследование является нежелательным. Классы, которые нельзя расширить, отмечаются модификатором **final**

```
final class BigBoss extends Manager {
    ...
}
```

Метод с модификатором **final** не может замещаться никакими методами подклассов. Все методы финального класса автоматически являются финальными.

Константные поля класса

С помощью модификатора **final** описываются константные поля класса:

```
final String QUERY = “select * from objects”;
```

Содержимое поля нельзя изменять после создания объекта.

Заметка: если класс объявлен финальным, его поля не становятся константными.

Модификатор static

- Означает принадлежность поля или метода к классу, а не к экземпляру класса. Для доступа к таким полям или методам создание объекта **не требуется**.

- Статические поля существуют в единственном экземпляре. Часто применяются для хранения констант.

- Необходимо различать *методы класса* и *методы экземпляра*. Статические методы не работают с объектами и не могут получить доступ к полям экземпляра.

- Для вызова статических методов необходимо использовать имена классов вместо имен объектов.

Модификатор static (пример)

```
public class Employee {
    // Счетчик экземпляров класса
    private static int counter = 0;

    public Employee() {
        counter++;
    }

    public static int getCounter() {
        return counter;
    }
}
```

Порядок вызовов конструкторов