# Прикладне програмування

Лекція №2

Викладач: Мамчур Дмитро Григорович

# Java и объектно-ориентированное программирование

## Достоинства

- Конструирование из простых компонент (абстракция).
- Данные связаны с операциями обработки.
- Инкапсуляция делает код более безопасным.
- Повторное использование компонент.
- Обобщенные алгоритмы.
- Изменение поведения во время выполнения (полиморфизм).
- Создание полуфабрикатов или фреймворков.

## Недостатки

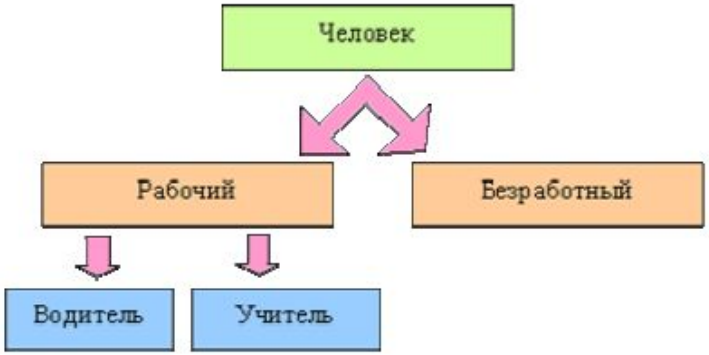
- Необходимость изучения концепций ООП
- Обилие библиотек компонентов повторного использования.
- Проектирование классов сложный процесс.
- Меньшее быстродействие
- Больший расход памяти
- Излишняя универсальность

# Java и объектно-ориентированное программирование

**Абстракция** позволяет акцентировать внимание на способах использования объекта и не вдаваться в подробности его реализации.

•**Инкапсуляция** — свойство языка программирования, позволяющее объединить и защитить данные и код в объекте и скрыть реализацию объекта от пользователя (прикладного программиста). При этом пользователю предоставляется только спецификация (интерфейс) объекта.

# Java и объектно-ориентированное программирование



- **Наследование** – описание нового класса на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.
  - Позволяет избавиться от дублирования кода.
  - Позволяет добавить новую функциональность в класс.
  - Позволяет описать отношения обобщения

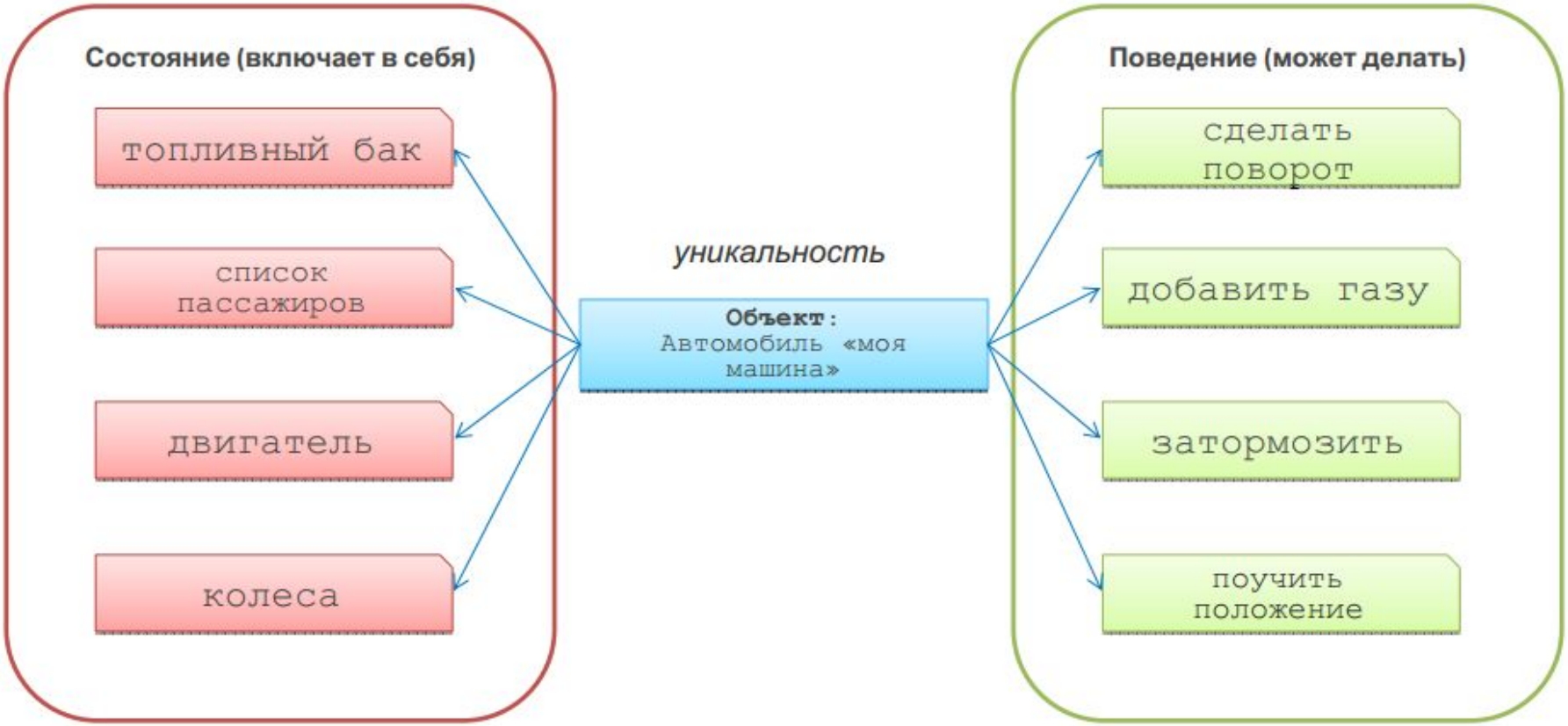
```
class Грузовик extends Автомобиль {
    Кузов кузов;
    процедура загрузить(груз) {
        кузов.загрузить(груз);
        кузов.проверитьПерегруз();
    }
}

Грузовик мойГрузовик = new Грузовик();
мойГрузовик.загрузить(помидоры);
мойГрузовик.добавитьГазу(немного);
```

# Java и объектно-ориентированное программирование

- **Полиморфизм** — возможность объектов с одинаковой спецификацией иметь различную реализацию. При этом различные объекты могут быть использованы одинаковым образом.
  - Позволяет писать более абстрактные программы.
  - Позволяет усилить повторное использование кода.
- Реализуется с помощью наследования и интерфейсов.

```
class Автостоянка {
    СписокАвтомобилей автомобили;
    процедура добавить(автомобиль) {
        автомобиль.закрыть();
        автомобиль.включитьСигнализацию();
        автомобили.добавить(автомобиль);
    }
}
Автостоянка стоянка = дом.гдеБлижайшаяАвтостоянка()
стоянка.добавить(мойАвтомобиль);
стоянка.добавить(мойГрузовик);
```

# Java и объектно-ориентированное программирование



# Java и объектно-ориентированное программирование

```
class Автомобиль {
    ТопливныйБак бак;
    СписокПассажиров пассажиры;
    Колеса колеса;
    Двигатель двигатель;

    процедура сделатьПоворот(угол);
    процедура добавитьГазу(уровень);
    процедура затормозить();
    Координаты получитьПоложение();
}
```

```
class Двигатель {
    Свечи свечи;
    Цилиндры цилиндры;
    Карбюратор карбюратор;

    процедура увеличитьОбороты();
    процедура переключитьРежим(режим);
}
```

```
Автомобиль мояМашина = я.купитьМашину();
Двигатель двигательМоейМашины = мояМашина.двигатель;
двигательМоейМашины.переключитьРежим(форсированный);
```

# Java и объектно-ориентированное программирование

**Объект** –
*экземпляр класса.*

- состояние
- поведение
- уникальность

**Класс** –
*тип объекта.*

- определяет набор свойств и интерфейс взаимодействия.
- определяет поведение (реализацию)

```
Автомобиль мой = new Автомобиль();
Автомобиль жены = new Автомобиль();
Топливо топливо = жены.слитьТопливо();
мой.заправить(топливо);
мой.двигаться();
```

```
class Автомобиль {
    Двигатель двигатель;
    процедура двигаться() {
        двигатель.завести();
        двигатель.добавитьГазу();
    }
}
```

## Java и объектно-ориентированное программирование

Основная конструкция языка программирования Java – это **класс**. У каждого класса есть какие-то характеристики, называемые **полями** (другими словами – переменные) и умения что-то делать, называемые **методами** (другими словами - функции).

```
public class Wasinkremenchuk{
    public static void main(String[] args) {
    int n = 5;
    System.out.println("I was in Kremenchuk " + n + "
times!");
    }
}
```

В приведенной программе `Wasinkremenchuk` – это класс, `main` – это метод, `n` – поле.

## Java и объектно-ориентированное программирование

В каждой программе, которую мы собираемся запускать на выполнение, должен быть метод main. Этот метод будет выполнен при запуске программы.

```
Объявление класса {
    Объявление полей...
    Описание методов...
    Описание метода main
}
```

## Java и объектно-ориентированное программирование

Покажем теперь, как класс из одной программы можно использовать в другой программе. Ниже приведен текст программы krem.java.

```
public class Krem {
    public void reklama(String napravlenieObuchenija, int
    chisloBudzhetnyhmest) {
        System.out.println("В КрНУ около 5 000 студентов
        учатся по более чем 30 направлениям обучения.");
        System.out.println("Одним из этих направлений
        является направление " + napravlenieObuchenija);
        System.out.println("Число бюджетных мест по этому
        направлению: " + chisloBudzhetnyhmest);
    }
}


public class Telek{
    public static void main(String[] args) {
        Krem k = new Krem();
        k.reklama("Компьютерная инженерия",30);
    }
}
```

## Java и объектно-ориентированное программирование

Чтобы использовать методы класса krem, нужно создать **экземпляр** этого класса. Это делается в строке:

```
Krem k = new Krem();
```

При этом создается экземпляр `k` класса `Krem`.

В общем виде создание экземпляра заданного класса выглядит так:

```
Имя_класса имя_переменной = new Имя_класса()
```

Ключевое слово `new` как раз и говорит виртуальной машине Java о том, что в памяти выделяется место под новый экземпляр класса.

Теперь с помощью `k` можно вызывать метод `Reklama`: `k.reklama`

## Наследование и изменение класса. Полиморфизм

Создадим файл krnubest.java со следующим текстом

```
public class Krnubest extends Krnu {
    public void reklama(String napravlenieObuchenija, int
chisloBudzhetnyhmest) {
    super.reklama(napravlenieObuchenija, chisloBudzhetnyhmest);
    System.out.println(napravlenieObuchenija + " – это лучшее
направление обучения в КрНУ!");
    }
}
```

Ключевое слово extends означает, что класс krnubest наследует(копирует) класс Krnu:

```
class Krnubest extends Krnu
```

Класс, который наследует еще называется классом-потомком, а класс, которого наследуют - классом-родителем. Наследовав от класса krnu его поля и методы, в классе krnubest мы имеем право оставить любой из них без изменения, либо дополнить, либо полностью изменить, переписав заново. В этом состоит полиморфизм объектного программирования на Java.

## Наследование и изменение класса. Полиморфизм

Если мы хотим дополнить какой-то метод, для запуска его старой версии предусмотрено ключевое слово super. Формат его использования:

```
super.имя_родительского_метода
```

В данном примере метод super позволяет вызвать неизмененный метод родительского класса r`eklama`. Таким образом, `super.reclama` означает вызов метода `reclama` наследуемого класса `Krnu` из файла Krnu.java.

## Наследование и изменение класса. Полиморфизм

Приведем более сложный пример, однако, позволяющий нам до конца разобраться с полиморфизмом. На рис. приведен текст класса punktir.java.

```
public class Punktir {

    public void line1() {
        System.out.print("---");
    }

    public void line2() {
        System.out.print("=");
    }

    public void punktirLine(int dlina) {
        for (int i=0; i < dlina; i++) {
            line1();
            line2();
        }
    }
}
```

## Наследование и изменение класса. Полиморфизм

На рисунке приведен текст программы, которая использует класс `punktir`.

```
public class Tst{
    public static void main(String[] args) {
        Punktir p = new Punktir();
        p.PunktirLine(5);
    }
}
```

При запуске программы tst.class мы увидим на экране вот такое подобие пунктира:

```
---=---=---=---=---=
```

## Наследование и изменение класса. Полиморфизм

Теперь создадим класс `punktir2`, который наследует `punktir` и заменим текст одного из методов, например, `line2`.

```
public class Punktir2 extends Punktir {
    int num=1;
    public void line2() {
        for (int i=0; i<num; i++)
            System.out.print("o");
        num++;
    }
}
```

Обратите внимание, что в классе-потомке `Punktir2` описывается только метод `line2`, а методы `line1` и `punktirLine` вообще не упоминаются. Это значит, что эти неупомянутые методы остались неизменными, такими же, как и в родительском классе. Метод `line2` полностью изменен в классе-потомке. Тем не менее он точно так же, как и раньше будет без проблем вызываться из метода `punktirLine`. На рис. 1.11 приведен текст программы Tst2.java, использующий новый класс Punktir2.java.

# Наследование и изменение класса. Полиморфизм

```
public class Tst2{
    public static void main(String[] args) {
        Punktir2 p = new Punktir2();
        p.punktirLine(5);
    }
}
```

При запуске программы Tst2.class мы увидим следующую строку:

```
---o---oo---ooo---oooo---ooooo
```

## Модификаторы public, private, protected

Как мы уже заметили, перед именами классов, методов и переменных у нас часто стоит служебное слово `public`. Так вот, это служебное слово сообщает компилятору Java, что помеченные им метод или поле можно без ограничений использовать в других классах (в других программах). Кроме служебного слова `public`, есть еще другие служебные слова, в частности `private` и `protected`. Вот что означают эти слова:

`public` – методы и поля видно и можно использовать где угодно;
`private` – методы и поля видно и можно использовать только в этом классе;
`protected` – методы и поля видно и можно использовать только в этом классе или в классах, наследующих его с помощью extends.

Крайне рекомендуется защищать все поля классов модификатором `private`, а также защищать этим модификатором большинство методов, которые не предполагается в дальнейшем использовать из других классов. Причем даже начинающий программист на Java, который пишет небольшие программы просто для тренировки, с самого начала должен приучать себя к этому правилу.

# Модификаторы public, private, protected

Рассмотрим на примере использование модификатора `private`.
На рис. приведен текст класса `factorial`, и класса `test`, который его использует.

```
public class Factorial {
    private int limit=10;

    public void demo() {
        int r=1;
        for (int i=2;i<=limit;i++)
            r=r*i;
        System.out.println("Факториал от значения " + limit +
" равен " + r);
    }
}

public class Test{
    public static void main(String[] args) {
        Factorial k = new Factorial();
        k.demo();
    }
}
```

## Модификаторы public, private, protected

Метод `demo()` выводит на экран результат вычисления факториала от числа 10. Это число хранится в поле`limit`, защищенном модификатором `private`. Ни прочитать, ни изменить значения этого поля из класса `test` невозможно. При попытке вставить в текст метода `main` класса `test`, например, строку `k.limit=6;` компилятор javac выдаст ошибку. Каким же образом, не нарушая принципа надежного программирования (все поля должны быть помечены `private`) тем не менее разрешить из других классов изменять значения полей? Очень просто – написать метод, который их изменяет. На рис. Приведена модификация класса `factorial`, позволяющая менять значение поля `limit`.

```
public class Factorial {
    private int limit=10;

    public void setLimit(int value) {
        limit=value;
    }
    public void demo() {
        int r=1;
        for (int i=2; i<=limit; i++)
            r=r*i;
        System.out.println("Факториал от значения " + limit +
" равен " + r);
    }
}
```

## Модификаторы public, private, protected

Здесь прописан новый метод `setLimit`, который меняет значение поля `limit` на `value` (входное данное метода). На рис приведена модифмкация текста программы test.java, в которой используется метод `setLimit`.

```
public class Test{
    public static void main(String[] args) {
        Factorial k = new Factorial();
        k.setLimit(6);
        k.demo();
    }
}
```

В этом случае компилятор не выдаст никакойо шибки, так как прямого обращения к полю `limit` в классе `test` нет, а изменение этого поля происходит внутри одного из методов класса `factoria`l, что разрешается модификатором `private`, так как это поле декларировано именно в этом классе. В результате модифицированная программа Test.java выводит на экран значение факториала для числа 6.