

Лекция 1.

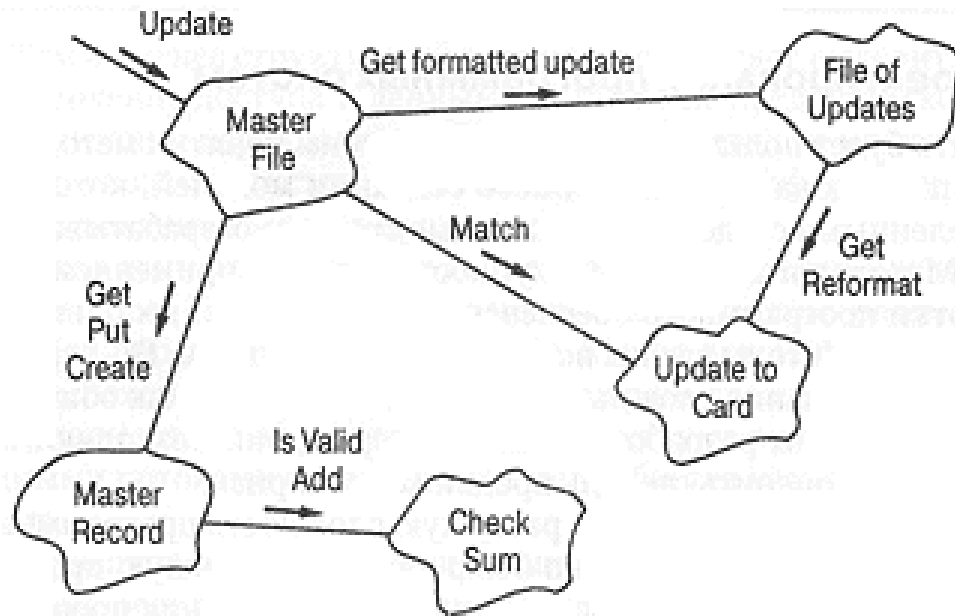
Основы ООП

Объектно-ориентированное программирование

- ООП - одна из самых распространенных «промышленных» парадигм программирования
- **Парадигма программирования** - это совокупность идей и понятий, определяющих стиль написания компьютерных программ
- Программы в ООП пишутся в терминах **объектов** и **классов**

Объектно-ориентированное программирование

- Java является **объектно-ориентированным языком**
- Это означает что программа на Java представляет собой набор взаимодействующих **объектов**



Объект

- **Объект** – это некоторая конкретная сущность (предмет, явление)
- **Примеры:**
Стол, кошка, гроза, ноутбук, человек и т.д.
- Каждый объект обладает **состоянием, поведением и уникальностью**

Состояние объекта

- **Состояние** – это набор характеристик объекта и их значений в данный момент времени
- В программировании состояние объектов задается при помощи переменных-полей
- **Пример для ноутбука:**
 - Высота, длина, ширина (вещественные числа)
 - Процент заряда (целое число)
 - Название модели (строка)
 - Заряжается сейчас или нет (**boolean**) и т.д.
- Состояние может меняться под внешним воздействием, либо сам объект может менять свое состояние
- Например, со временем процент заряда падает



Поведение объекта

- **Поведение** – это действия, которые может совершать объект и как объект может реагировать на воздействие со стороны других объектов
- Пример для ноутбука – его можно поставить на зарядку, и тогда процент будет расти
- Или выключатель – его можно включить или выключить
- В программировании поведение объекта задается при помощи функций-**методов**



Уникальность объекта

- **Уникальность** объекта – это то, что отличает его от других объектов
- Например, каждый человек уникален
- Или есть два одинаковых стула, но это все равно два стула, а не один. То есть стулья уникальны – это 2 отдельных объекта
- В программировании уникальность задается расположением объекта в памяти компьютера



Классы

- **Класс** – это вид **объектов** с одинаковой структурой и поведением
- Каждый объект обязательно **принадлежит** некоторому **классу**
- Если объект принадлежит некоторому классу, то говорят, что он является **экземпляром класса**
- То есть **объект** – это **экземпляр класса**

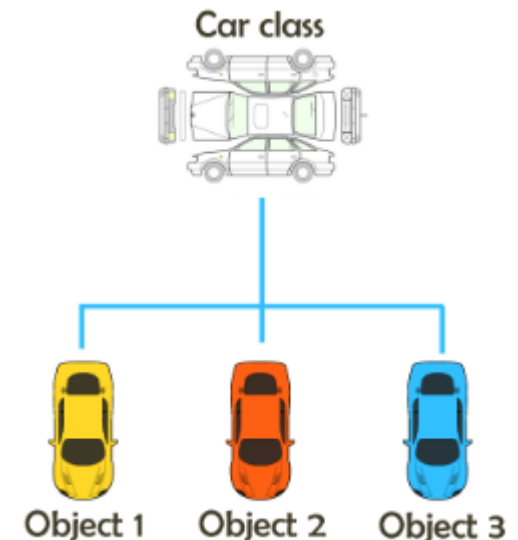
Пример класса

- Пусть, у нас есть кошка **Мурка**
- Она является объектом **класса Кошка**
- Все кошки (то есть объекты класса **Кошка**) устроены и ведут себя схожим образом
- У всех них есть свое состояние – цвет, положение в пространстве, размеры и др.
- У них есть свое поведение – они могут ходить, бегать, реагировать на воздействия и т.д.



Классы

- В программировании мы описываем **классы**
- В классе мы описываем, что в нем есть – какие поля, каких типов, какие есть методы, пишем их код
- А потом создаем сколько нам нужно **объектов (экземпляров)** этих классов и работаем с ними
- Т.е. класс – это как бы чертеж, описание, по которому потом можно создавать объекты, а объект – конкретная деталь, сделанная по этому чертежу

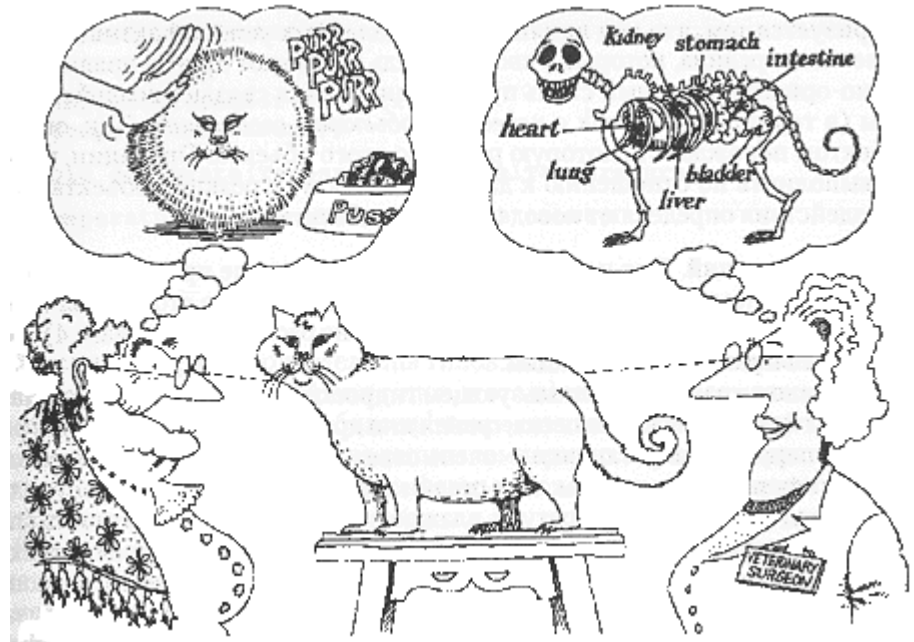


Принципы ООП

- В ООП программы пишутся в терминах классов и объектов
- Принципы ООП:
 - Абстракция
 - Инкапсуляция
 - Наследование
 - Полиморфизм

Абстракция

- **Абстракция** – выделение существенных характеристик объекта и существенного поведения, и отбрасывание несущественных характеристик и поведения
- Один и тот же объект реального мира для разных задач может быть представлен по-разному
- Важно выбирать абстракцию как можно более простую, но достаточную для задачи



Пример абстракции 1

- Допустим, мы деканат и у нас такая задача – хранить список всех студентов
- Для этого выделим класс **Студент**
- Понятно, что студент является человеком, то есть у него есть пол, размеры, вес, возраст, у каждого человека много присущих ему черт
- Но для нашей задачи нам достаточно знать о студентах только ФИО, дату рождения, контактные данные, номер зачетки, номер группы

Пример абстракции 2

- Допустим, мы военкомат и хотим хранить список всех студентов
- Для этого выделим класс **Студент**
- Для нашей задачи нам нужно знать ФИО, возраст, рост, состояние здоровья студента

Пример абстракции 3

- Допустим, мы бухгалтерия университета и хотим хранить список студентов
- Для этого выделим класс **Студент**
- Для нашей задачи нам нужно знать ФИО, номер стипендиальной карты, категорию учащегося (например, получает стипендию или нет; бюджетник или нет)

Абстракция

- Объект реального мира может быть одним и тем же
- Но в зависимости от задачи мы выбираем разные абстракции

Абстракция

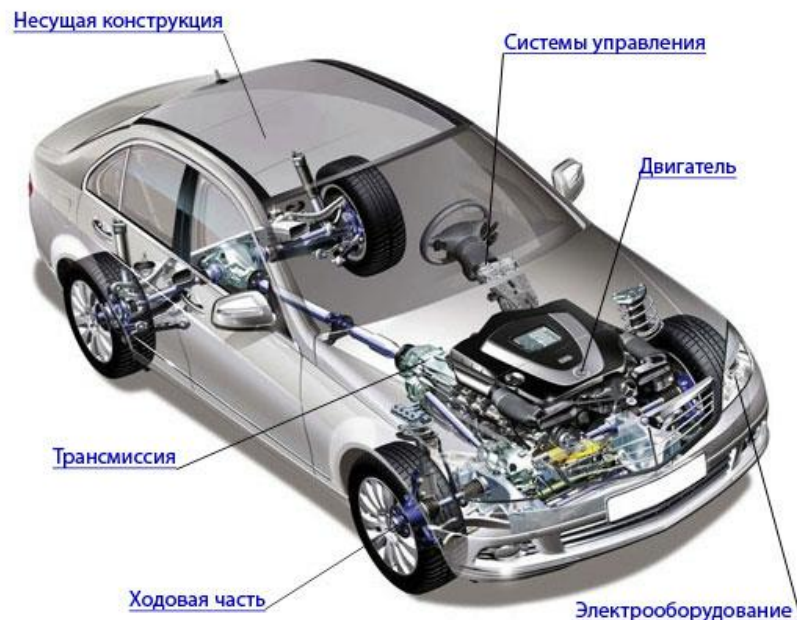
- Абстракция в основном охватывает внешние стороны объекта и не концентрируется на деталях реализации
- Пример абстракции – дверь
- В слове «дверь» не говорится из чего она сделана, ее размеры и так далее. Но мы уже понимаем что дверь можно открывать и закрывать. В этом и есть суть абстракции «дверь»

Инкапсуляция

- **Инкапсуляция** – механизм языка, позволяющий объединить данные и методы, работающие с этими данными, в единый объект, и скрыть детали реализации от пользователя кода
 - Здесь видно 2 роли инкапсуляции – объединение в объект и сокрытие реализации
- В соответствии с принципом инкапсуляции, мы хотим максимально скрыть от **пользователя кода (т.е. программиста)** реализацию классов, а дать им только возможность работать с публичным интерфейсом
- Это позволяет легко подменять одну реализацию другой (можно спокойно менять то, что скрыто)

Инкапсуляция

- Пример из жизни – устройство автомобиля
- Автомобиль состоит из огромного количества деталей, которые как-то друг с другом взаимодействуют
- Но чтобы водить автомобиль, не нужно знать многого – нужно только уметь работать с интерфейсом – руль, педали, коробка передач



Инкапсуляция

- Так же и в коде – класс может быть очень сложно устроен, иметь вспомогательные функции и поля, но наружу предоставлять только функции, нужные другим
- И другим программистам даже не нужно знать как этот класс устроен внутри
- Надо только знать как этим классом пользоваться
- Пример – класс `Scanner` в Java или класс `StreamReader` в C#

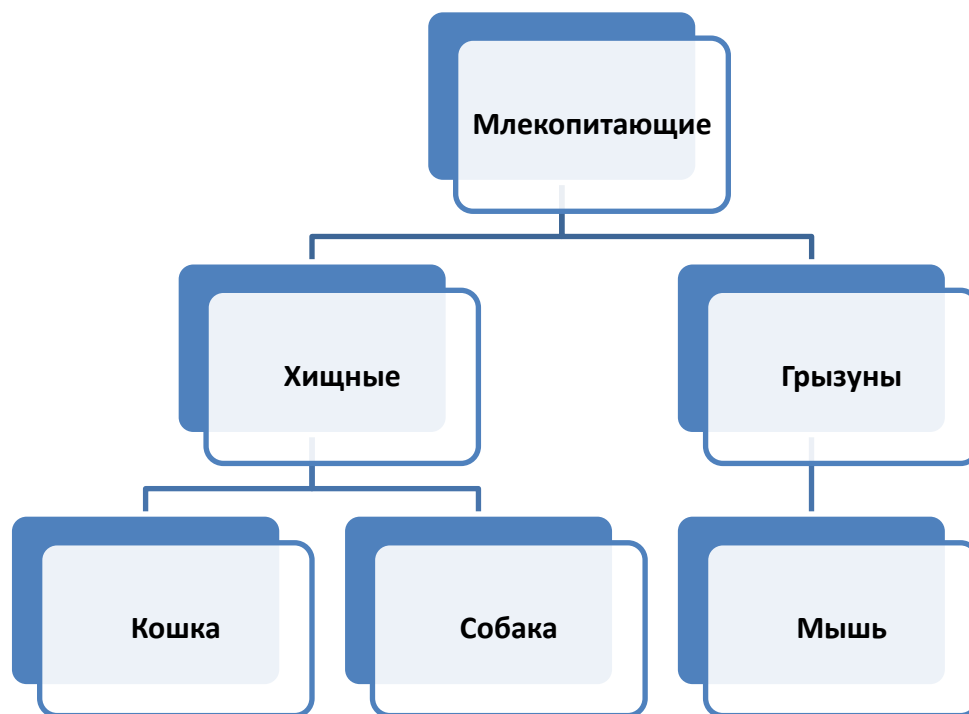
Путаница - инкапсуляция и сокрытие

- Есть другое определение от Гради Буча:
- **Инкапсуляция** – процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение
 - В этом определении идет акцент только на **сокрытие реализации**
- Считается, что нужно различать **инкапсуляцию** и **сокрытие**
- Есть разные мнения что такое инкапсуляция – кто-то считает, что это только сокрытие, кто-то считает, что это только объединение данных и методов в класс, кто-то, что это и то и другое вместе
- Мы будем считать, что это и объединение и сокрытие

Наследование

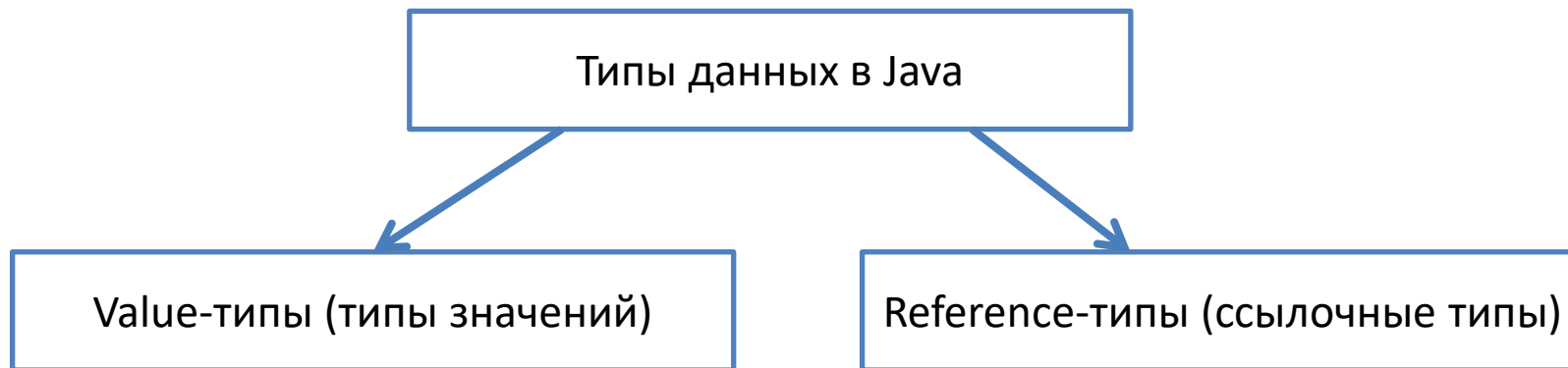
- Классы могут образовывать иерархию **наследования**
- Класс-наследник получает все свойства класса-родителя, может переопределять его черты, либо добавлять новые черты

Наследование



- Пример: биологическая классификация, рассмотрим кошку
- Так как кошка принадлежит классу млекопитающих, то она наследует свойства, присущие этому классу – кормление детей молоком
- Так как принадлежит классу хищников, то ест мясо и т.д.

Наследование



Числовые целые:

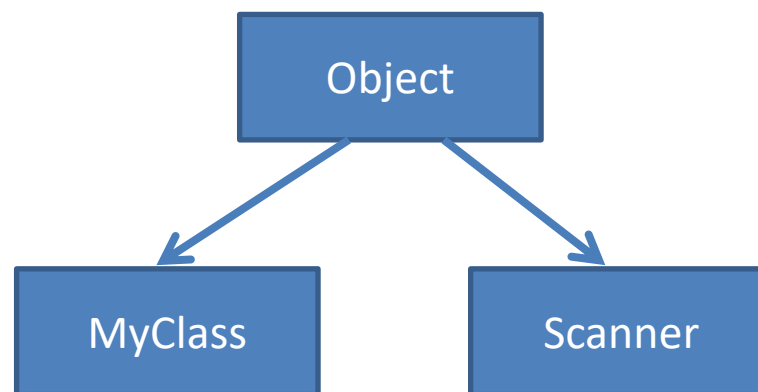
byte, short, int, long

Вещественные:

float, double

Логический: boolean

Символьный: char



**Все классы наследуются
от класса Object**

Подход без ООП со структурами

- Пусть у нас есть программа, в которой мы работаем с геометрическими точками на плоскости
- У нас есть сущность - точка с двумя координатами x , y
- В Java такого понятия нет, но в не ООП языках было такое понятие, как **структура**
- **Структура** – это тип данных, который внутри себя хранит несколько переменных
- ```
struct Point {
 double x;
 double y;
}
```

# Структуры

- `struct Point {  
 double x;  
 double y;  
}`
- Объявив такой тип, можно создавать его переменные и работать с ними
- `Point p = new Point();  
p.x = 4;  
p.y = 12;`
- В общем-то, похоже на класс, но у структур нет поведения. Они могут только хранить состояние

# Пример программы со структурами

- ```
public static void printPoint(Point p) {  
    System.out.printf("(%.f, %.f)", p.x, p.y);  
}
```
- ```
public static double getDistance(Point p1, Point p2) {
 return Math.sqrt(Math.pow(p1.x - p2.x, 2)
 + Math.pow(p1.y - p2.y, 2));
}
```
- ```
public static void main(String[] args) {  
    Point p = new Point();  
    p.x = 3;  
    p.y = 5;  
    printPoint(p);  
}
```

Краткий синтаксис создания структуры

- Есть еще такой синтаксис создания структуры:
- `Point p = { 3, 5 };`
- Мы указываем значения для полей через запятую по порядку
- Можно указывать не все поля

Недостатки структур

- Полный доступ на чтение и запись ко всем полям структуры – любой код может читать и писать поля, что может приводить к ошибкам в коде
- Код функций, работающих с полями структуры, находится отдельно
- Небезопасная логика создания экземпляров структуры:
 - Нет возможности сделать валидацию входных данных
 - При использовании краткого синтаксиса заполнения полей структуры можно ошибиться – заполнить не все поля
 - Или в будущем изменится порядок полей, или добавятся новые поля, а мы забудем поменять код

От структур к классам

- `class Point {`

`private double x;`

`private double y;`

В классе 2 переменных (поля)

`public Point(double x, double y) {`

`this.x = x;`

`this.y = y;`

`}`

Конструктор – функция
инициализации нового объекта

`public void print() {`

`System.out.printf("(%.2f, %.2f)", x, y);`

`}`

Функции могут обращаться
к полям, и они не static

`public double getDistance(Point p) {`

`return Math.sqrt(Math.pow(x - p.x, 2) + Math.pow(y - p.y, 2));`

`}`

`}`

От структур к классам

- ```
public static void main(String[] args) {
 Point p = new Point(3, 5);
 p.print();
}
```
- Сравним со структурами:
- ```
public static void main(String[] args) {  
    Point p = { 3, 5 };  
    printPoint(p);  
}
```
- ООП вариант проще для понимания, т.к. мы думаем в терминах объектов - мы просим точку распечататься
- А в варианте со структурами мы передаем точку в функцию, это более «машинный» подход

Выгода использования классов

- Классы серьезно упрощают понимание кода
- Код легче модифицируется так как данные и функции находятся вместе – в коде класса
- Класс может скрывать то, чего другим знать не нужно (при помощи модификаторов видимости, например, `private`)
- Есть надежная логика инициализации новых объектов при помощи конструкторов

Синтаксис класа

Классы в Java

- `class Point {`
 `// члены класса: поля и методы`
}
- Каждый класс в Java может содержать **поля (переменные)** и **методы (функции)**
- **Поля** определяют структуру класса, а **методы** – поведение класса

Классы в Java

- `class Point {` **Имя класса**

`private double x;`
`private double y;` **Поля (переменные)**

`public Point(double x, double y) {`
`this.x = x;`
`this.y = y;`
`}`

Конструктор
(специальная функция),
вызываемая при
создании объекта

`public void print() {`
`System.out.printf("(%.2f, %.2f)", x, y);`
`}`
`}`

Метод (функция)

Порядок объявления членов класса

- ```
class Point {
 private double x;
 private double y;

 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }
 public void print() {
 System.out.printf("(%f, %f)", x, y);
 }
}
```
- Порядок членов класса неважен, но обычно поля пишут вместе вверху, ниже пишут вместе конструкторы, а ниже - методы

# Классы в Java

- class Point {  
 private double x;  
 private double y;

- public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
}

- public void print() {  
 System.out.printf("(%.2f, %.2f)", x, y);  
}

Если имя поля конфликтует  
с именем переменной, то  
обращаемся к нему через  
this

this – ключевое слово,  
обозначающего текущий  
объект (для которого  
вызвана функция)

Можем всегда обращаться  
к полям и методам через  
this

# Конструкторы

- ```
class Main {  
    public static void main(String[] args) {  
        Point point = new Point(3, 2);  
        point.print();  
    }  
}
```
- **Конструктор** – специальная функция, которая позволяет создать и инициализировать экземпляр класса
- Конструктор нельзя вызвать явно, но он вызывается если создавать объект при помощи оператора **new**

Конструкторы

- ```
class Point {
 private double x;
 private double y;

 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }
}
```
- При объявлении функции-конструктора не указывается возвращаемый тип. Конструктор ничего не возвращает
- Имя конструктора всегда совпадает с именем класса

# Конструкторы

- Класс может иметь несколько конструкторов
- Это будет перегрузка, как для обычных методов

- ```
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Point() {  
    }  
}
```


Вызов конструкторов друг из друга

- Чтобы не дублировать код конструкторов, в конструкторе есть возможность вызывать другие конструкторы через ключевое слово `this`

- ```
public class Point {
 private double x;
 private double y;
```

```
 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }
```

```
 public Point() {
 this(0, 0);
 }
}
```

Вызов через `this` должен быть первой командой в конструкторе.  
Дальше может идти другой код

# Конструктор по умолчанию

- Если при объявлении класса не создавать конструктор, то компилятор Java сам генерирует **конструктор по умолчанию** (он без аргументов), который ничего не делает, а только вызывает конструктор класса-родителя
- Если в классе создать любой конструктор с аргументами, то компилятор не создает конструктор по умолчанию
- В этом случае если понадобится создать конструктор без аргументов, то его нужно будет объявить самим

# Обращение к полям и методам классов

- ```
class Main {  
    public static void main(String[] args) {  
        Point point = new Point(3, 2);  
        point.print();  
    }  
}
```
- Обращение к полям и методам объекта осуществляется через оператор точка
- Для членов класса могут иметься разные **права доступа**. Они задаются при объявлении класса при помощи **модификаторов видимости**, например, `public` и `private`. Еще есть `protected` и package видимость
- Если прав недостаточно, то обращение к члену класса приведет к ошибке компиляции


Классы в Java

- class Point {
 private double x;
 private double y;

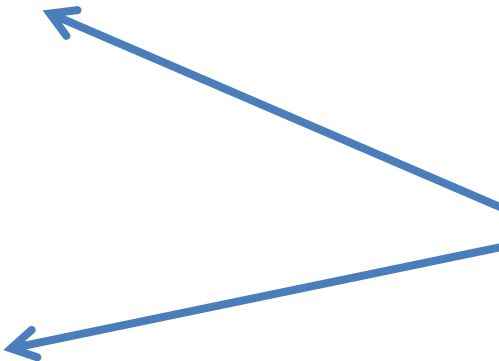
 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }

 public void print() {
 System.out.printf("(%.f, %.f)", x, y);
 }
}

private члены класса
видны только
функциям внутри
класса



public члены класса
видны всем



Обращение к полям и методам классов

- Модификаторы видимости и есть средство инкапсуляции в Java – они позволяют скрыть реализацию класса, а наружу выставлять только то, что должны использовать пользователи класса
- **Поля всегда должны быть `private`!! Если к ним все же нужен доступ, то для этого должны использоваться методы**

Обращение к полям и методам классов

- Поля всегда должны быть **private**!! Если к ним все же нужен доступ, то для этого должны использоваться методы

- ```
class Point {
 private double x;

 public double getX() {
 return x;
 }

 public void setX(double x) {
 this.x = x;
 }
}
```

Соглашение именования –  
методы для получения  
значений должны  
начинаться с **get**, а для  
установки значения – с **set**

Методы **get**  
называют геттерами,  
методы **set** -  
сеттерами

Не обязательно  
иметь оба

# Зачем поля private?

- **Достоинства:**

- Пользователи кода теперь не могут вмешиваться во внутренние дела класса, например, присвоить полю недопустимое значение
- Если имя поля изменится, или поле вообще исчезнет, то метод можно оставить с прежним именем, и тогда это изменение не затронет код, который использовал этот метод
- Метод может выполнять дополнительную работу: проверять корректность данных, сохранять сообщения в лог и т.д.

- **Недостатки:**

- Некоторое падение производительности т.к. получить значение поля дешевле, чем вызвать метод. Но производительность часто не важна

# Примеры

- Класс прямоугольника, который хранит площадь в поле
- Класс человека с возрастом – отрицательное значение



# Нестатические члены класса

- ```
class Person {  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

Поле name будет
свое у каждого
экземпляра класса
Person

Методы класса
могут работать с
полями объекта. На
сам объект можно
сослаться при
помощи слова this

Статические члены

Статические члены относятся не к конкретным экземплярам, а к классу в целом

- class Person {
 private String name;
 public static final int MAX_NAME_LENGTH = 100;
 public Person(String name) {
 this.name = name;
 }
 public String getName() {
 return name;
 }
 public static String formatName() {
 // код
 }
}

Статические поля существуют в единственном экземпляре

Чтобы работать со статическими членами не нужно создавать объекты класса

Статические члены класса

- ```
class Person {
 public static final int MAX_NAME_LENGTH = 100;
 public static String formatName(String name) {
 // возвращает имя с инициалами
 }
}
```
- Как обращаться к статическим методам и полям:
- ```
public static void main(String[] args) {  
    int maxLength = Person.MAX_NAME_LENGTH;  
    String formattedName  
        = Person.formatName("Ivan Ivanovich");  
}
```

Статические члены класса

- Мы уже много работали со статическими методами и полями
- Например, мы использовали класс `Math` и его статические члены:
 - `Math.PI` – статическое поле-константа
 - `Math.random()` – получение случайного числа
 - `Math.abs(x)` – получение модуля числа и т.д.
- Такие классы, как `Math`, которые содержат только статические методы и статические константы, называются **классами-утилитами**

Static и не-static

	He static	Static
В целом	Относится к объекту	Относится к классу в целом
Поля	Это поле будет у каждого объекта свое	Поле будет одно на весь класс. Оно хранится не в объектах, а отдельно в программе
Методы	Метод вызывается только от объекта	Метод вызывается от класса в целом

Задачи на дом «Range», «Range*»

- См. файл со списком задач

Структура программ

Структура программ на Java

- Программы на Java обычно состоят из многих файлов
- В каждом файле находится один или более классов
- Классы можно группировать по так называемым **пакетам**
- В них стараются помещать близкие друг к другу типы
- Например, в один пакет можно поместить классы GUI – графического интерфейса, а в другой пакет – классы логики программы

Пакеты

- Указать к какому пакету относится файл, можно при помощи команды **package**
- **Файл Main.java:**
- `package ru.academits.java;`

Все классы из данного файла будут лежать в пакете ru.academits.java

```
public class Main {  
    public static void main(String[] args) {  
        //...  
    }  
}
```

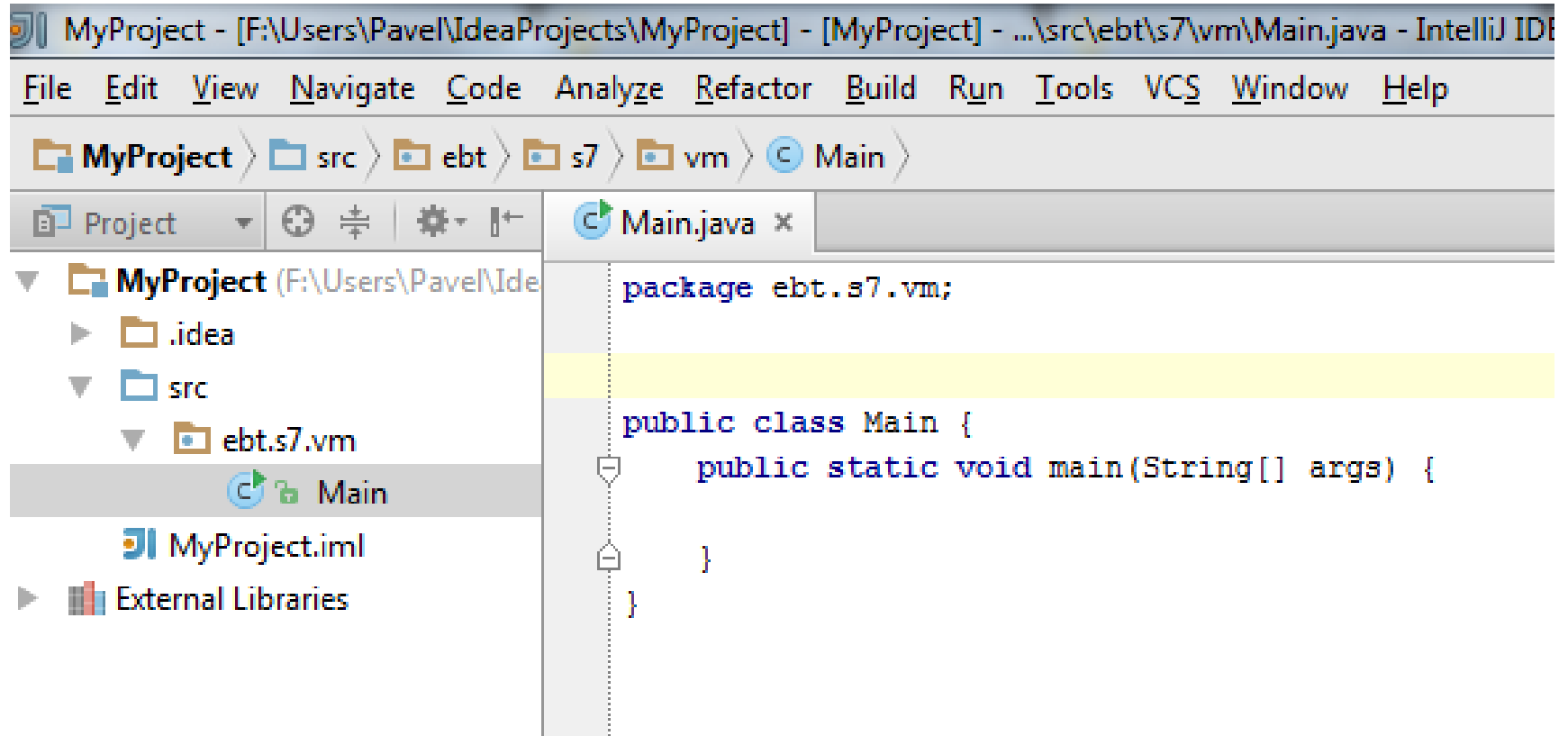
Слово `public` при объявлении класса означает что класс «виден» из других пакетов. Если `public` не написать, то класс не виден снаружи пакета

Внутри одного java файла может находиться только один публичный класс и его имя должно совпадать с именем файла

Пакеты

- `package ru.academits.java;`
- Пакеты могут вкладываться друг в друга. Запись **ru.academits.java** означает что есть пакет **ru**, в нем есть вложенный пакет **academits**, а в нем есть вложенный пакет **java**
- Структура папок проекта должна повторять структуру пакетов. В противном случае будет ошибка компиляции
- Класс **Main** находится в пакете `ru.academits.java`. Это означает, что в проекте на верхнем уровне должна быть папка `ru` (имя совпадает с именем пакета), внутри неё – папка `academits`, внутри неё – папка `java`, внутри неё – файл `Main.java`, в котором обязательно есть класс с именем `Main` и в этом файле указан `package ru.academits.java`.

Пакеты



Зачем нужны пакеты?

- Пакеты позволяют:
 - лучше структурировать файлы проекта
 - не делать вспомогательные классы доступными вне пакета, внутри которого они объявлены. То есть пакеты также являются средством инкапсуляции
 - избежать конфликтов имен. Благодаря пакетам можно давать разным классам одинаковые имена, если эти классы лежат в разных пакетах
- В Java именем класса является не просто имя, которое мы указываем при объявлении класса, а имя пакета + имя класса
- Наш класс Main на самом деле: **ru.academits.java.Main**

Какие имена давать пакетам?

- Имена пакетов следует делать уникальными для всего мира, чтобы никогда не возникало конфликтов имен с чужим кодом
- Поэтому для уникальности, обычно, компании используют url своего сайта в качестве имен пакетов
- Например, компания у нас Academ IT School (можно сократить до **academits**), сайт **academ-it-school.ru**
- Поэтому пакет будет: **ru.academits**
- Заметьте, что части имени сайта идут в обратном порядке, от общего к частному: **ru** означает Россию, **academits** – более конкретная часть России
- Для каждого проекта делается свой пакет. Например, для проекта Virtual Manager: **ru.academits.vm**

Какие имена давать пакетам?

- Для студентов можно порекомендовать что-то такое:
 - ru.nsu.mmf.g12202.ivanov

Пакеты стандартной библиотеки

- Стандартная библиотека Java тоже поделена на множество пакетов
- Например:
 - Основные классы находятся в пакете **java.lang**, например, класс **Object**
 - Классы для работы со вводом и выводом находятся в пакете **java.io**
Например, это классы потоков ввода и вывода, файлы и т.д.
 - Пакет **java.net** содержит классы для работы с сетью и т.д.
 - Пакет **java.util** содержит много полезных классов, например, **Scanner**

Пакет по умолчанию

- Если в файле с кодом не указать инструкцию **package**, то класс помещается в так называемый пакет по умолчанию, который не имеет имени
- Следует этого избегать и использовать только в самых простых программах

Import

- Любой класс может использовать любые публичные классы из других пакетов и любые классы из своего пакета
- Но обращаться к классам других пакетов можно только по квалифицированному (полному имени) – имя пакета + имя класса
- Например, хотим использовать `java.util.Scanner`, тогда придется писать:
`java.util.Scanner s = new java.util.Scanner(System.in);`
- Чтобы все время не писать полные имена, а использовать только имя класса, существует инструкция **import**

Import конкретного класса

```
package ru.academits.vm;
```

```
import java.util.Scanner;
```

```
public class Main {  
    public static void main(String[] args) {  
        Scanner s = new Scanner(System.in);  
    }  
}
```

Теперь в этом файле имя
Scanner означает класс
java.util.Scanner

Это импорт имени
конкретного класса

Import всего пакета

```
package ru.academits.vm;
```

```
import java.util.*;
```

```
public class Main {  
    public static void main(String[] args) {  
        Scanner s = new Scanner(System.in);  
    }  
}
```

Теперь если в этом файле встретится имя класса, который есть в пакете `java.util`, то компилятор будет считать что мы используем этот класс

Рекомендуется импортировать классы по одному, а не целым пакетом

Import всего пакета

```
package ru.academits.vm;
```

```
import java.util.*;  
import ru.academits.Scanner;
```

Import конкретного класса
«перебивает» import всего
пакета

```
public class Main {  
    public static void main(String[] args) {  
        Scanner s = new Scanner(System.in);  
    }  
}
```

Будет использоваться
ru.academits.Scanner

Разрешение неоднозначности

```
package ru.academits.vm;
```

```
import java.util.*;
```

```
public class Main {  
    public static void main(String[] args) {  
        Scanner s = new Scanner(System.in);  
        ru.academits.Scanner s1 = new ru.academits.Scanner();  
    }  
}
```

java.util.Scanner импортирован

Для своего класса Scanner
придется использовать полное
квалифицированное имя

Класс во внешнем пакете

- Пусть не публичный класс A объявлен в пакете ru
- `package ru;`

```
class A {  
}
```

- Тогда класс A не виден **во всех** других пакетах, **даже во вложенных** в ru

Чтение на дом

- Рекомендую дома читать этот курс:
- <http://www.intuit.ru/studies/courses/16/16/lecture/27105>
- И любые другие материалы, какие хочется
- **Обязательно к следующему разу прочитать вторую лекцию из курса intuit**
- В свободное время читайте этот курс, задавайте вопросы

Задачи на курс

- См. Shapes, Vector, Matrix, CSV