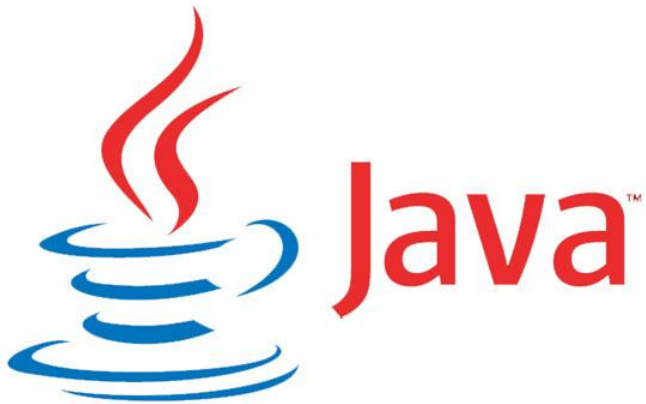
# Тема 3. Объектно-ориентированное программирование.



# Понятие класса и объекта


Java является объектно-ориентированным языком, поэтому такие понятия, как «класс» и «объект» играют в нем ключевую роль.

**Объект** – это любая конкретная сущность, с которой можно каким-либо образом взаимодействовать.

Более простыми словами – объект – это все, что нас окружает.

Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

# Понятие класса и объекта


Шаблоном или описанием объекта является **класс**, а объект представляет **экземпляр** этого класса.

***Проведем следующую аналогию:***

У нас у всех есть некоторое представление о человеке – наличие двух рук, двух ног, головы, пищеварительной и нервной системы, головного мозга, прямохождение и т.д.

Т.е. имеется некоторый *шаблон*, по которому можно описать конкретного человека. Этот шаблон можно назвать **КЛАССОМ**. А реально существующий человек (фактически – экземпляр данного класса) является **ОБЪЕКТОМ** данного класса.

# Понятие класса и объекта


## Класс - красивым языком

Класс – это абстрактный тип данных, описывающий присущие ему свойства, поведение и возможности взаимодействия.

**Красиво. Жаль, что непонятно ☺️**

# Понятие класса и объекта

## Класс - понятным языком

объект

жук
1

объект

жук
2

объект

жук
3

объект

жук
4

объект

жук
5

Класс объектов жук# – АВТОМОБИЛЬ

жук# -- объект класса АВТОМОБИЛЬ
жук# -- экземпляр класса АВТОМОБИЛЬ

# Понятие класса и объекта

## Класс - понятным языком

Класс АВТОМОБИЛЬ

# что присуще всем автомобилям

- Имеет двигатель
- Имеет колеса
- Имеет руль
- Умеет ездить

жук

Объект ЖУК

# что присуще данному жуку

- Имеет двигатель
- Имеет колеса
- Имеет руль
- Умеет ездить

Объект жук обладает всеми свойствами, которые присущи автомобилям.
Мы можем с чистой совестью утверждать, что ЖУК принадлежит классу АВТОМОБИЛЬ.

# Понятие класса и объекта


## Класс – путь через желудок

Представьте, что вы собираетесь печь булочки и у вас есть специальная *форма* для выпекания.

Вы можете рассматривать форму для выпекания – как класс, а сами булочки – как объекты.

Форма для выпекания определяет то, какие булочки у вас будут, то есть задает их свойства.
То же самое относится и к классам!

# Понятие класса и объекта


## Определение класса

Класс определяется с помощью ключевого слова **class**.

```
class Book{
}
```

Вся функциональность класса представлена его членами-полями (полями называются переменные класса) и методами.

# Понятие класса и объекта

## Определение класса

Например, класс Book мог бы иметь следующие определения:

```
class Book{

   public String name;
   public String author;
   public int year;

   public void info(){
      System.out.println(“The name of this book is ” + name);
   }

}
```

# Понятие класса и объекта


## Конструктор

Таким образом, в классе Book определены три переменных и один метод, который печатает значение переменной name.

Кроме обычных методов в классах используются также и специальные методы, которые называются ***конструкторами***. Конструкторы нужны для создания нового объекта данного класса и, как правило, выполняют начальную инициализацию объекта.

Название конструктора должно совпадать с названием класса:

# Понятие класса и объекта


```
class Book{

   public String name;
   public String author;
   public int year;

   Book(){
         name = “unknown”;
         author = “unknown”;
         year = 0;
   }

   Book(String name, String author, int year){
         this.name = name;
         this.author = author;
         this.year = year;
   }

   public void info(){
      System.out.println(“The name of this book is ” + name);
   }

 }
```

# Понятие класса и объекта


## Конструктор

- Здесь у класса Book определено два конструктора;
- Первый конструктор без параметров присваивает неопределенные начальные значения полям;
- Второй конструктор присваивает полям класса значения, которые передаются через его параметры;
- Так как имена параметров и имена полей класса в данном случае совпадают, то необходимо использовать ключевое слово **this**, которое представляет собой ссылку на текущий объект;

# Понятие класса и объекта


## Конструктор

- В выражении this.name = name; первая часть this.name означает, что name – это поле текущего класса, а не название параметра name;
- Если бы параметры конструктора и поля класса имели разное название, использовать ключевое слово this было бы необязательно;
- Можно определить несколько конструкторов для установки разного количества параметров и затем вызывать один конструктор из другого:

# Понятие класса и объекта


```
class Book{

   public String name;
   public String author;
   public int year;

   Book(String name, String author){
         this.name = name;
         this.author = author;
   }

   Book(String name, String author, int year){
         this(name, author);
         this.year = year;
   }

   public void info(){
      System.out.println("The name of this book is " + name);
   }

 }
```

# Понятие класса и объекта


## Конструктор

Как видно из примера выше, один конструктор класса может быть вызван из другого конструктора класса также через ключевое слово this, после которого в скобках перечисляются необходимые параметры.

# Создание объекта

Чтобы непосредственно использовать класс в программе, необходимо создать его объект.

*Процесс создания объекта двухступенчатый*:

- Вначале объявляется переменная данного класса;
- Затем, с помощью ключевого слова new и конструктора, непосредственно создается объект, на который и будет указывать объявленная переменная:

```
Book book;
b = new Book();
```

# Создание объекта


- После объявления переменной Book b; эта переменная еще не ссылается ни на какой объект и имеет значение null;
- Сам объект класса Book создается с помощью одного из конструкторов и ключевого слова new.

# Инициализаторы


- Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта;
- Так можно заменить конструктор без параметров следующим блоком:

# Инициализаторы

```
class Book{

   public String name;
   public String author;
   public int year;

   {
         name = “unknown”;
         author = “unknown”;
         year = 0;
   }

   Book(String name, String author, int year){
         this.name = name;
         this.author = author;
         this.year = year;
   }

   public void info(){
      System.out.println(“The name of this book is ” + name);
   }

 }
```

# Пакеты

- Как правило, Java классы объединяются в пакеты;
- Пакеты позволяют логически организовать классы в наборы;
- По умолчанию java уже имеет ряд встроенных пакетов (java.lang, java.util, java.io и др.);
- Кроме того, пакеты могут иметь вложенные пакеты;
- Организация классов в пакеты позволяет избежать конфликта имен между классами, ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами;
- Принадлежность к пакету позволяет гарантировать однозначность имен.

# Пакеты

- Чтобы указать, что класс принадлежит определенному пакету, необходимо использовать директиву package, после которой указывается имя пакета;

```
package bookstore;
public class BookStore{
	public static void main(String[] args){

	}
}
```

# Пакеты

- В данном случае класс BookStore находится в пакете bookstore;
- При определении классов в пакеты на жестком диске эти классы должны размещаться в подкаталогах, путь к которым соответствует названию пакета;
- Например, в данном случае файл BookStore.java будет находиться в каталоге bookstore.
- Классы необязательно определять в пакеты;
- Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

# Импорт пакетов и классов


- Если возникает необходимость использовать классы из других пакетов, нужно эти классы и пакеты подключить;
- Исключение составляют классы из пакета java.lang (например, String), которые подключаются в программу автоматически.
- Например, знакомый по прошлым темам класс Scanner находится в пакете java.util, поэтому мы можем получить к нему доступ следующим образом:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

# Импорт пакетов и классов


- В данном случае, необходимо указать полный путь к файлу в пакете при создании его объекта;
- Однако такое нагромождение имен пакетов не всегда удобно, и, в качестве альтернативы, можно импортировать пакеты и классы в проект с помощью директивы import, которая указывается после директивы package:

```
package bookstore;
import java.util.Scanner;
public class BookStore{
	public static void main(String[] args){
		Scanner in = new Scanner(System.in);
	}
}
```

# Импорт пакетов и классов


- Директива import указывается в самом начале кода, после чего идет имя подключаемого класса (в данном случае, класс Scanner);
- В примере выше был подключен только один класс, однако, пакет java.util содержит множество классов, и, чтобы не подключать по отдельности каждый класс, можно подключить

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

# Импорт пакетов и классов


- Теперь можно использовать любой класс из пакета java.util;
- Возможна ситуация, когда используются два класса с одинаковым именем из двух разных пакетов.

Например, класс Date имеется и в пакете java.util, и в пакете java.sql.

И если нужно одновременно использовать два этих класса, то понадобится указать полный путь к ним:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date();
```

# Статический импорт

В java есть особая форма импорта – статический импорт.

Для этого вместе с директивой import используется модификатор static.

```
package bookstore;
import static java.lang.System.*;
import static java.lang.Math.*;

public class BookStore{
	public static void main(String[] args){
		double result = sqrt(20);
		out.println(result);
	}
}
```

# Статический импорт

- Здесь происходит статический импорт классов System и Math, которые имеют статические методы;
- Благодаря операции статического импорта можно использовать эти методы без названия класса (т.е. писать не Math.sqrt(20), а sqrt(20), не System.out.println(result), а out.println(result));
- Статические члены класса будут рассмотрены позже.

# Модификаторы доступа


Все члены класса в языке Java – поля и методы, свойства – имеют модификаторы доступа.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть, контекст, в котором можно употреблять данную переменную или метод.

# Модификаторы доступа

***В Java используются следующие модификаторы доступа:***

- ***public***: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов;
- ***private***: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе;
- ***protected***: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах;
- ***Модификатор по умолчанию***. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию.
- Такие поля или методы видны всем классам только в текущем пакете.

# Статические члены и модификатор static

Кроме обычных методов и полей, класс может иметь статические поля, методы, константы и инициализаторы.

Например, главный класс программы имеет метод main, который является статическим:

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

}
```

Для объявления статических переменных, констант, методов и инициализаторов перед их объявлением указывается ключевое слово **static**.

Статические члены класса могут использоваться без создания объектов класса.

Например, в классе System содержится статическая переменная out, с помощью которой выводятся статические данные на консоль.

# Статические члены и модификатор static

```
public class Book {
    private int id;
    private static int counter = 1;

    public void display(){
        System.out.printf("Id: %d \n", id);
    }

    private String author;
    private int year;
    private String name;

    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
        id = counter++;
    }
}
```

# Статические члены и модификатор static

- Класс Book содержит статическую переменную counter, которая увеличивается в конструкторе, и ее значение присваивается переменной id.
- После этого возможно создать несколько объектов класса Book, и в каждом вызове конструктора переменная counter будет увеличиваться на единицу, так как она относится НЕ к конкретному объекту, а ко ВСЕМУ классу Book в целом или всем объектам Book сразу:

```
public static void main(String[] args) {
        Book book1 = new Book("Война и мир", "Л.Н. Толстой", 1863);
        book1.displayId(); //Print Id: 1
        Book book2 = new Book("Отцы и дети", "И. Тургенев", 1862);
        book2.displayId(); //Print Id: 2
    }
```

# Статические члены и модификатор static

- В примере выше статическая переменная инициализируется сразу, но не редко для инициализации статических полей применяется статический блок.
- Этот блок вызывается один раз в программе при создании первого объекта данного класса:

```
public class Book {
    private int id;
    private static int counter;

    static {
        counter = 1;
        System.out.println("Вызов статического
блока");
    }

    //Остальной код класса
}
```

# Статические члены и модификатор static

- Нередко константы на уровне класса объявляются как статические:

```
public final static double PI = 3.14;
```

- Статические методы, подобно статическим переменным, также относятся ко всему классу.
- Например, создадим новый класс Algorithm и добавим в него две функции для вычисления числа Фибоначчи и факториала:

# Статические члены и модификатор static


```
public class Algorithm {
   public final static double PI = 3.14;

   public static int factorial(int x){
      if(x == 1){
         return x;
      }else {
         return x * factorial(x - 1);
      }
   }

   public static int fibonachi(int x){
      if(x == 0){
         return 1;
      }

      if(x == 1){
         return 1;
      }else {
         return fibonachi(x - 1) + fibonachi(x - 2);
      }
   }
}
```

# Статические члены и модификатор static

Теперь используем их в программе.

```
public static void main(String[] args) {
        int num1 = Algorithm.factorial(5);
        int num2 = Algorithm.fibonachi(5);
        System.out.println(Algorithm.PI);
    }
```

- И поскольку методы factorial и fibonachi, а также поле PI являются статическими, то мы можем к ним обратиться напрямую без создания объекта класса: Algorithm.factorial(5);
- При использовании статических методов надо учитывать ограничения: в статических методах можно вызывать только другие статические методы и использовать только статические переменные.

# Объекты как параметры методов

Объекты классов, как и данные примитивных типов могут передаваться в методы.

Однако, в данном случае, есть одна особенность – объекты, в отличие от примитивных типов, передаются по ссылке.

Пример. Пусть имеется следующий класс Book:

```
public class Book {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public Book(String name) {
      this.name = name;
   }
}
```

# Объекты как параметры методов


В основном класса программы определим два дополнительных метода:

```
public class Book {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public Book(String name) {
      this.name = name;
   }

   private static void read(Book b){
      b.setName("Unknown book");
   }

   private static void read(int x){
      x = 20;
   }

   public static void main(String[] args) {
      Book book = new Book("War and peace");
      read(book);
      System.out.println(book.getName()); // Unknown book

      int n = 10;
      read(n);
      System.out.println(n); //10
   }
}
```

# Объекты как параметры методов


Здесь определены два метода с одним и тем же именем, но с разными параметрами – методы read(). Вызываются эти методы в методе main, в них передаются, соответственно объект Book и число n.

В случае с объектом Book в метод будет передаваться ссылка на область памяти, в котором объект Book находится. И при изменении объекта Book в методе read() соответственно поменяются данные и в этой области памяти. Поэтому на выходе объект Book будет называться не "War and peace", а "Unknown book". Поэтому и говорят, что объекты классов передаются по ссылке.

Однако, число n после передачи в метод read() не изменит своего значения, так как в методе read() будет создаваться копия этого числа в виде параметра x, и, затем, эта копия будет использоваться в методе. После завершения работы метода число x уничтожается.

# Объекты как параметры методов

Правда, среди классов есть и исключения – класс String.

Объекты данного класса являются неизменяемыми (immutable) – если понадобится присвоить объекту String новое значение, то для этого система создаст новый объект.

Например:

```
public class Book {
   private static void read(String title){
      title = "Unknown book";
   }

   public static void main(String[] args) {
      String title = "Отцы и дети";
      read(title);
      System.out.println(title); //"Отцы и
дети"
   }
}
```

# Вложенные и внутренние классы


- Если определение класса размещается внутри определения другого класса, то такие классы называются вложенными или внутренними.
- Область видимости вложенного класса ограничена областью видимости внешнего класса, поэтому, если создать класс B внутри класса A, то класс B не сможет существовать независимо от класса A.
- Вложенные классы позволяют группировать классы, логически принадлежащие друг другу, и управлять доступом к ним.

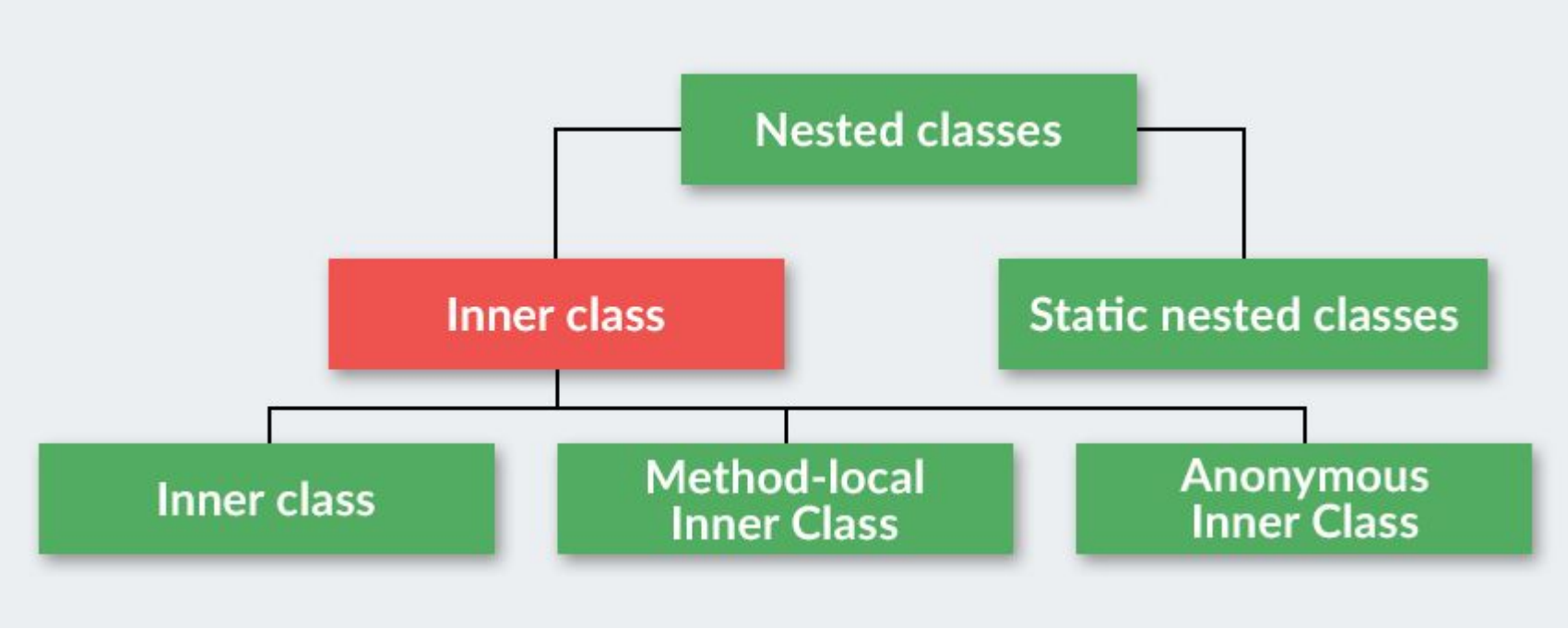
**Существуют два типа вложенных классов:**

- Non-static nested classes — нестатические вложенные классы или внутренние классы (inner classes);
- Static nested classes — статические вложенные классы.

# Вложенные и внутренние классы

В свою очередь, внутренние классы (inner classes) имеют два особых подвида:

- локальный класс (local class)
- анонимный класс (anonymous class)



# Вложенные и внутренние классы

## Для чего нужен внутренний класс

Для понимания сути использования внутреннего класса рассмотрим следующий пример:

```
public class Bicycle {

   private String model;
   private int weight;

   public Bicycle(String model, int weight) {
      this.model = model;
      this.weight = weight;
   }

   public void start() {
      System.out.println("Поехали!");
   }

   public class  SteeringWheel {
      public void right() {
         System.out.println("Руль вправо!");
      }

      public void left() {
         System.out.println("Руль влево!");
      }
   }

   public class  Seat {
      public void up() {
         System.out.println("сиденье поднято выше!");
      }

      public void down() {
         System.out.println("сиденье опущено ниже!");
      }
   }
}
```

# Вложенные и внутренние классы


- Создадим класс Bicycle (велосипед);
- У класса Bicycle есть два поля – модель и вес и один метод start();
- Отличие класса Bicycle от обычного класса в том, что внутри него имеется два других класса SteeringWheel (руль) и Seat (сиденье);
- Классы SteeringWheel и Seat – это полноценные классы, имеющие собственные методы, поля, конструкторы и т.д. (в данном случае, только методы);
- И вот тут может возникнуть очень хороший вопрос: **а зачем мы вообще засунули одни классы внутрь другого? Зачем делать их внутренними, если можно просто создать два отдельных класса для руля и сиденья?**
- Ответ прост: технических ограничений у нас нет — можно сделать и так, но дело здесь скорее в правильном проектировании классов с точки зрения конкретной программы и в смысле этой программы: руль, сиденье, педали — это составные части велосипеда, и отдельно от него они не имеют смысла.

# Вложенные и внутренние классы


**Внутренние классы** — *это классы для выделения в программе некой сущности, которая неразрывно связана с другой сущностью*.

Если сделать все вышеперечисленные классы отдельными публичными классами, в нашей программе мог бы появиться, к примеру, такой код, смысл которого сложно объяснить:

```
public class Main {
  public static void main(String[] args) {
     SteeringWheel steeringWheel = new SteeringWheel();
     steeringWheel.right();
  }
}
```

Есть какой-то непонятный велосипедный руль (зачем он нужен?), который поворачивается вправо...сам по себе, без велосипеда...зачем-то.

**Отделив сущность руля от сущности велосипеда, мы потеряли логику нашей программы**.

# Вложенные и внутренние классы


**С использованием внутреннего класса код смотрится совсем иначе:**

```
public class Main {
  public static void main(String[] args) {
     Bicycle peugeot = new Bicycle("Peugeot", 120);
     Bicycle.SteeringWheel wheel = peugeot.new SteeringWheel();
     Bicycle.Seat seat = peugeot.new Seat();

     seat.up();
     peugeot.start();
     wheel.left();
     wheel.right();
  }
}
```

# Вложенные и внутренние классы


**Однако, давайте рассмотрим другую ситуацию:**

*необходимо создать программу, моделирующую магазин велосипедов и их запчастей*.

В этой ситуации предыдущее решение будет неудачным.

В рамках магазина запчастей каждая отдельная часть велосипеда имеет смысл даже отдельно от сущности велосипеда.

Например,  могут понадобиться методы типа «продать покупателю педали», «купить новое сидение» и т.д.

*Здесь использовать внутренние классы было бы* ***ошибкой*** *— каждая отдельная часть велосипеда в рамках нашей новой программы имеет собственный смысл:* ***она отделима от сущности велосипеда, никак не привязана к нему***.

# Вложенные и внутренние классы

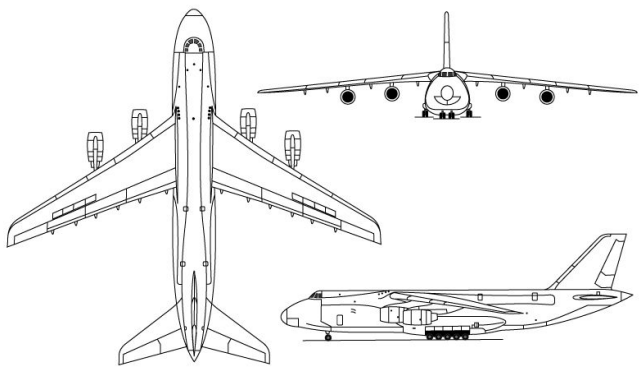

## Статические вложенные классы

При объявлении такого класса используется ключевое слово **static.**

```
public class Boeing737 {

    private int productionYear;
    private static int maxPassengersCount = 300;

    public Boeing737(int productionYear) {
        this.productionYear = productionYear;
    }

    public int getProductionYear() {
        return productionYear;
    }

    public static class Drawing {
        public static int getMaxPassengersCount() {
            return maxPassengersCount;
        }
    }
}
```

# Вложенные и внутренние классы

- В данном примере имеется внешний класс Boeing737, который создает самолет этой модели;
- В классе объявлен конструктор с одним параметром - годом выпуска (int productionYear);
- Имеется одна статическая переменная int maxPassengersCount - максимальное число пассажиров;
- Оно будет одинаковым у всех самолетов одной модели, так что достаточно одного экземпляра этой переменной;
- Описан статический внутренний класс Drawing — чертеж самолета, в который может быть помещена вся служ[illegible]те . В нашем случае она содержит количество пас[illegible] содержать много другой информации.



# Вложенные и внутренние классы


## Отличия между статическим и нестатическим вложенными классами

1. *Объект статического класса не хранит ссылку на конкретный экземпляр внешнего класса*

Если в примере с внутренним классом речь шла о том, что в каждый экземпляр внутреннего класса SteeringWheel (руль) незаметно для нас передается ссылка на объект внешнего класса Bicycle (велосипед) и, без объекта внешнего класса объект внутреннего просто не может существовать, то для статических вложенных классов это не так - объект статического вложенного класса вполне может существовать сам по себе.

Это делает статические классы более независимыми. Единственный момент — при создании такого объекта нужно указывать название внешнего класса:

```
public class Main {
   public static void main(String[] args) {
      Boeing737.Drawing drawing1 = new Boeing737.Drawing();
      Boeing737.Drawing drawing2 = new Boeing737.Drawing();
   }
}
```

# Вложенные и внутренние классы


## Отличия между статическим и нестатическим вложенными классами

*2. Разный доступ к переменным и методам внешнего класса*

Статический вложенный класс может обращаться только к статическим полям внешнего класса.

В нашем примере в классе Drawing есть метод getMaxPassengersCount(), который возвращает значение статической переменной maxPassengersCount из внешнего класса.

Однако невозможно создать метод getProductionYear() в Drawing для возврата значения productionYear. Ведь переменная productionYear— нестатическая, а значит, должна принадлежать конкретному экземпляру Boeing737. А как мы уже выяснили, в случае со статическими вложенными классами, объект внешнего класса запросто может отсутствовать.

Отсюда и ограничение :)

# Вложенные и внутренние классы


## Снова немного «философии»

Почему мы сделали класс Drawing статическим, а в прошлой лекции класс Seat (сиденье велосипеда) был нестатическим?

В отличие от сиденья велосипеда, сущность чертежа не привязана так жестко к сущности самолета.

Отдельный объект сиденья, без велосипеда, чаще всего будет бессмысленным (хотя и не всегда).

*Сущность чертежа имеет смысл сама по себе*.

Например, он может пригодиться инженерам, планирующим ремонт самолета. Сам самолет для планирования им не нужен, и может находиться где угодно — достаточно просто чертежа.

Кроме того,  для всех самолетов одной модели чертеж все равно будет **одинаковым**, так что такой жесткой связи, как у сиденья с велосипедом, нет. Поэтому и ссылка на конкретный объект самолета объекту Drawing не нужна.

# Внутренние классы в локальном методе


**Локальный класс** – это класс, который объявляется только в блоке кода (чаще всего — внутри какого-то метода внешнего класса).

```
public class PhoneNumberValidator {

  public void validatePhoneNumber(String number) {

      class PhoneNumber {

        private String phoneNumber;

        public PhoneNumber() {
           this.phoneNumber = number;
        }

        public String getPhoneNumber() {
           return phoneNumber;
        }

        public void setPhoneNumber(String phoneNumber) {
           this.phoneNumber = phoneNumber;
        }
      }

      //...код валидации номера
  }
}
```

# Внутренние классы в локальном методе


- Это небольшая программа – валидатор телефонных номеров;
- Ее метод validatePhoneNumber() принимает на вход строку и определяет, является ли она номером телефона;
- Внутри этого метода объявлен локальный класс PhoneNumber.

*Снова возникает вполне логичный вопрос: а зачем*? ☺

Как  и в случае с велосипедом, все зависит от структуры и предназначения создаваемой программы.

- В большинстве случаев PhoneNumberValidator будет даже не самостоятельной программой, а просто частью в логике авторизации для основной программы;
- Например, на разных сайтах при регистрации часто просят ввести номер телефона, и, если напечатать какую-нибудь чушь вместо цифр, сайт выдаст ошибку о некорректных данных;
- Для работы такого сайта (а точнее, механизма авторизации пользователя) его разработчики могут включить в код аналог PhoneNumberValidator.

# Внутренние классы в локальном методе


- Иными словами, имеется один внешний класс с одним методом, который будет использован в одном месте программы и больше нигде;
- А если и будет, то в нем ничего не изменится, так как один метод делает свою работу;
- В этом случае, раз уж вся логика работы собрана в одном методе, будет гораздо удобнее и правильнее инкапсулировать там и дополнительный класс;
- Своих методов, кроме геттера и сеттера, у него нет и, по сути, нужны только данные из его конструктора, а в других местах он не задействован;
- Следовательно нет причин выносить информацию о нем за пределы единственного метода, где он и используется.

# Внутренние классы в локальном методе


Локальный класс можно объявить также просто в блоке кода или даже в цикле:

```
public class PhoneNumberValidator {
  {
     class PhoneNumber {
        private String phoneNumber;
        public PhoneNumber(String phoneNumber) {
           this.phoneNumber = phoneNumber;
        }
     }
  }

  public void validatePhoneNumber(String
phoneNumber) {
     //...код валидации номера
  }
}
```

```
public class PhoneNumberValidator {
  public void validatePhoneNumber(String phoneNumber) {
     for (int i = 0; i < 10; i++) {
        class PhoneNumber {
           private String phoneNumber;
           public PhoneNumber(String phoneNumber) {
              this.phoneNumber = phoneNumber;
           }
        }
        //...какая-то логика
     }
     //...код валидации номера
  }
}
```

Но такие случаи – редкость

☺️

# Внутренние классы в локальном методе


## Особенности локальных классов

- Объект локального класса не может создаваться за пределами метода или блока, в котором его объявили;
- Имеет доступ к локальным переменным и параметрам метода;
- В Java 7 локальный класс может получить доступ к локальной переменной или параметру метода, только если они объявлены в методе как final;
- Однако в Java 8 поведение локальных классов было изменено: в этой версии языка локальный класс имеет доступ не только к final-локальным переменным и параметрам, но и к effective-final;

***Effective-final называют переменную, значение которой не менялось после инициализации.***

- У локального класса есть доступ ко всем (даже private) полям и методам внешнего класса: и к статическим, и к нестатическим.

# Внутренние классы в локальном методе


## Особенности локальных классов

- Нельзя объявить интерфейс внутри блока, так как интерфейсы по своей природе статичны;
- Но если интерфейс объявлен внутри внешнего класса, локальный класс может его реализовать;
- В локальных классах нельзя объявлять статические инициализаторы (блоки инициализации), но у локальных классов могут быть статические члены при условии, что они постоянные переменные (static final).

# Анонимные классы

## Почему «анонимные»? ☺️

Предположим, что имеется основная программа, которая постоянно работает и что-то делает. Перед разработчиком стоит задача создать для этой программы систему мониторинга из нескольких модулей:

- Модуль для отслеживания общих показателей работы и ведения лога;
- Модуль для фиксации и регистрации ошибки в журнале ошибок;
- Модуль для отслеживания подозрительной активности (например, попытки несанкционированного доступа и прочие связанные с безопасностью вещи).

Поскольку все три модуля должны, по сути, просто стартовать в начале программы и работать в фоновом режиме, будет хорошей идеей создать для них общий интерфейс:

```
public interface MonitoringSystem {

    public void startMonitoring();
}
```

# Анонимные классы


## Почему «анонимные»? ☺️

Данный интерфейс будут реализовывать 3 конкретных класса-модуля:

```
public class GeneralIndicatorsMonitoringModule implements MonitoringSystem {

@Override
    public void startMonitoring() {
        System.out.println("Мониторинг общих показателей стартовал!");
    }
}


public class ErrorMonitoringModule implements MonitoringSystem {

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг отслеживания ошибок стартовал!");
    }
}


public class SecurityModule implements MonitoringSystem {

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг безопасности стартовал!");
    }
}
```

# Анонимные классы

## Почему «анонимные»? ☺️

В данном примере все выглядит логично:

- Есть система, состоящая из нескольких модулей;
- У каждого модуля есть собственное поведение;
- Можно легко добавлять новые модули, ведь имеется интерфейс, который просто реализовать.

Все логично, но: чтобы система заработала, нужно просто создать 3 объекта - GeneralIndicatorsMonitoringModule, ErrorMonitoringModule, SecurityModule – и вызвать метод startMonitoring() у каждого из них.

```
public class Main {

    public static void main(String[] args) {

        GeneralIndicatorsMonitoringModule generalModule = new GeneralIndicatorsMonitoringModule();
        ErrorMonitoringModule errorModule = new ErrorMonitoringModule();
        SecurityModule securityModule = new SecurityModule();

        generalModule.startMonitoring();
        errorModule.startMonitoring();
        securityModule.startMonitoring();
    }
}
```

# Анонимные классы


## Почему «анонимные»? ☺️

Вывод в консоль:
Мониторинг общих показателей стартовал!
Мониторинг отслеживания ошибок стартовал!
Мониторинг безопасности стартовал!

И для такой небольшой работы была написана целая система аж с 3, по сути, одноразовыми классами и целым интерфейсом! ☺️

Вопрос ☺️

Что можно изменить?

# Анонимные классы

## Почему «анонимные»? ☺️

Здесь на помощь приходят анонимные внутренние классы.

```
public class Main {

    public static void main(String[] args) {

        MonitoringSystem generalModule = new MonitoringSystem() {
            @Override
            public void startMonitoring() {
                System.out.println("Мониторинг общих показателей стартовал!");
            }
        };



MonitoringSystem errorModule = new MonitoringSystem() {
            @Override
            public void startMonitoring() {
                System.out.println("Мониторинг отслеживания ошибок стартовал!");
            }
        };

        MonitoringSystem securityModule = new MonitoringSystem() {
            @Override
            public void startMonitoring() {
                System.out.println("Мониторинг безопасности стартовал!");
            }
        };

        generalModule.startMonitoring();
        errorModule.startMonitoring();
        securityModule.startMonitoring();
    }
}
```

# Анонимные классы


## Что же тут происходит? ☺️

Выглядит все так, как будто создается объект интерфейса, но его объект создать нельзя.

В тот момент, когда пишется

```
MonitoringSystem generalModule = new MonitoringSystem() {

};
```

внутри Java-машины происходит следующее:

1. Создается безымянный Java-класс, реализующий интерфейс MonitoringSystem;
2. Компилятор, увидев такой класс, начинает требовать от разработчика реализовать все методы интерфейса MonitoringSystem (что и происходит 3 раза);
3. Создается один объект этого класса.

# Анонимные классы


### Что же тут происходит? ☺️

Обратите внимание на код – в конце стоит точка с запятой. И стоит не просто так: здесь одновременно объявляется класс с помощью **{}** и создается его объект с помощью **();**.

Каждый из наших трех объектов переопределил метод startMonitoring() по-своему.

И осталось лишь вызвать этот метод у каждого из них:

```
generalModule.startMonitoring();
errorModule.startMonitoring();
securityModule.startMonitoring();
```

Задача выполнена!

Все три модуля успешно запущены и работают, а структура программы стала намного проще.

# Анонимные классы


**Что же тут происходит?** ☺️

Если каждому из анонимных классов-модулей понадобится какое-то отличающееся поведение, свои специфические методы, которых нет у других, их можно легко дописать:

```
MonitoringSystem generalModule = new MonitoringSystem() {

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг общих показателей стартовал!");
    }

    public void someSpecificMethod() {

        System.out.println("Специфический метод только для первого модуля");
    }
};
```

В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».

# Анонимные классы

## Особенности анонимных классов

- Анонимный класс — это полноценный внутренний класс, поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и private:

```
public class Main {

    private static int currentErrorsCount = 23;

    public static void main(String[] args) {

        MonitoringSystem errorModule = new MonitoringSystem() {

            @Override
            public void startMonitoring() {
                System.out.println("Мониторинг отслеживания ошибок стартовал!");
            }

            public int getCurrentErrorsCount() {

                return currentErrorsCount;
            }
        };
    }
}
```

# Анонимные классы

## Особенности анонимных классов

- Анонимные классы видны только внутри того метода, в котором определены: любые попытки обратиться к объекту errorModule за пределами метода main() будут неудачными;
- Анонимный к

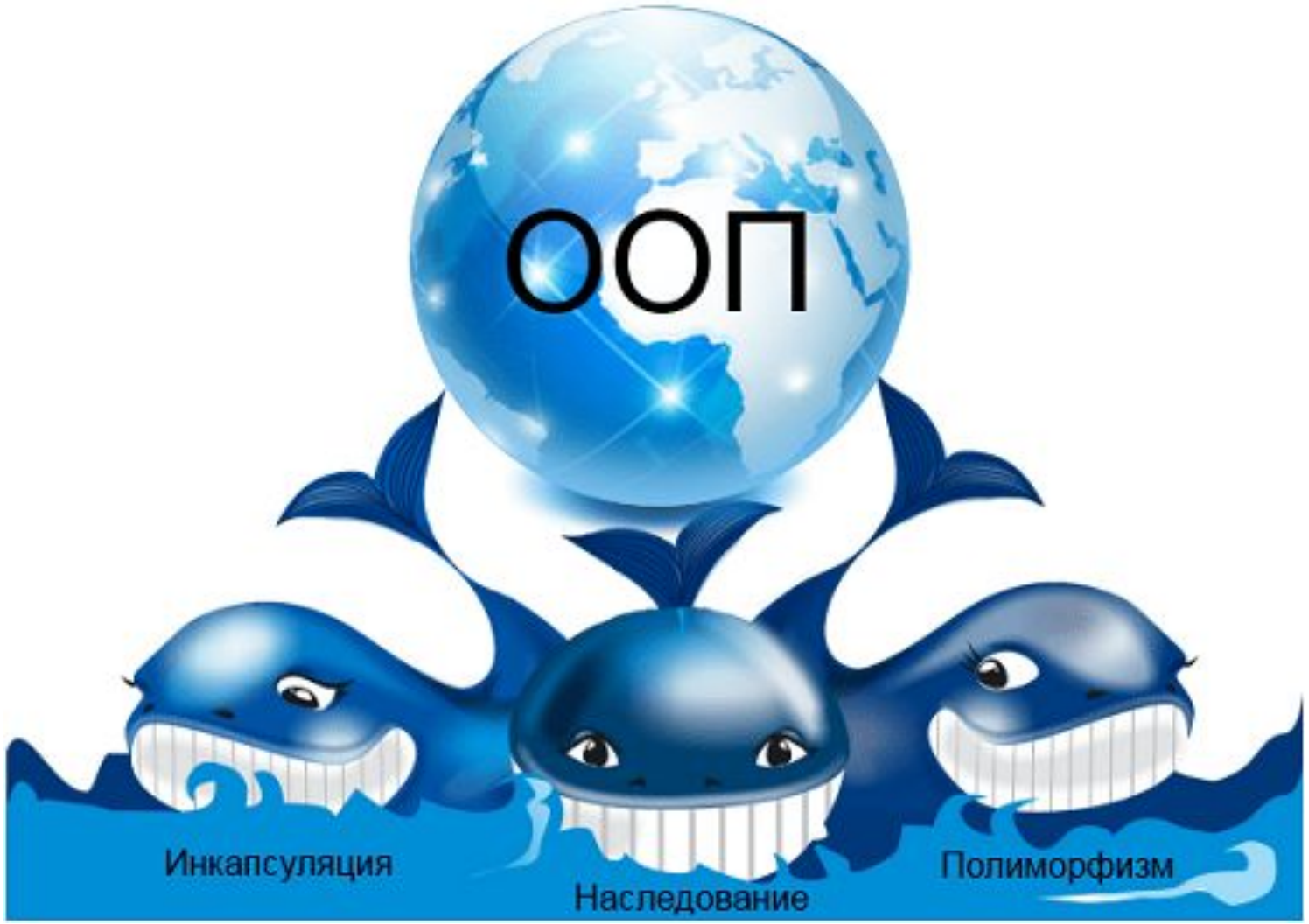
```
//ошибка! Inner classes cannot have static declarations
public static int getCurrentErrorsCount() {

    return currentErrorsCount;
}
```

ленные и методы:

- Тот же результат будет, если попробовать объявить статическую переменную:

```
MonitoringSystem errorModule = new MonitoringSystem() {

    //ошибка! Inner classes cannot have static declarations!
    static int staticInt = 10;

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг отслеживания ошибок стартовал!");
    }

};
```

# Парадигмы ООП

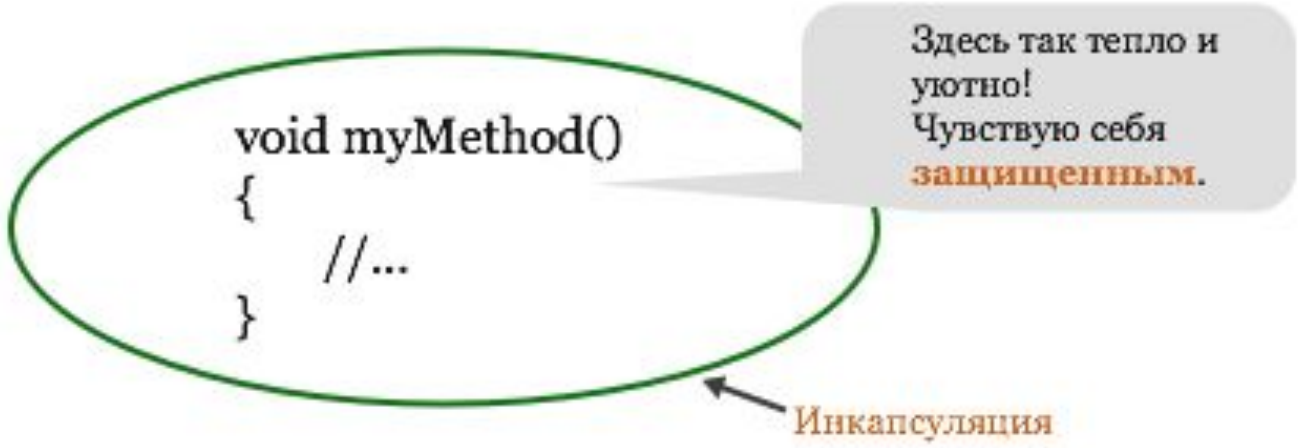


# Парадигмы ООП

## Инкапсуляция

***Инкапсуляция*** – это «упаковка», «сокрытие» данных и каких то методов их обработки в один компонент.

В программировании инкапсуляция реализуется при помощи механизма классов.



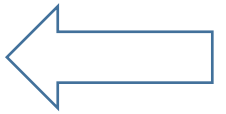
# Парадигмы ООП

## Инкапсуляция

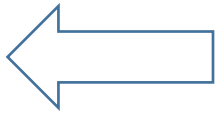
***Возьмем нашего жука и опишем класс***

класс ЖУК:
- ✔ 4 колеса
- ✔ МКПП
- ✔ ДВС

- ✔ Увеличить скорость
- ✔ Снизить скорость
- ✔ Повернуть



Данные/знания



Методы/функции

***Класс ЖУК объединяет данные о движущей части и методы работы с ней в единое целое!***

***Класс ЖУК ИНКАПСУЛИРУЕТ их в себе.***

# Парадигмы ООП

## Инкапсуляция. Другая сторона силы. Сокрытие.

***Возьмем нашего многострадального жука***

- Жук – объект класса ЖУК, который мы описали ранее, а значит, как мы помним, умеет разгоняться, тормозить и поворачивать.
- Проще говоря – умеет ездить.
- Но большинство людей понятия не имеют, как он это делает и почему. Да и вообще, что происходит, когда они едут в авто.
- Однако им и не надо. Им достаточно того, что авто может их отвезти куда нужно.
- Таким образом наш жук скрывает в себе *реализацию процесса движения*.

***Жук хранит в себе знание о том, как ехать, но не раскрывает его!***

***Жук инкапсулирует в себе это знание.***

# Парадигмы ООП

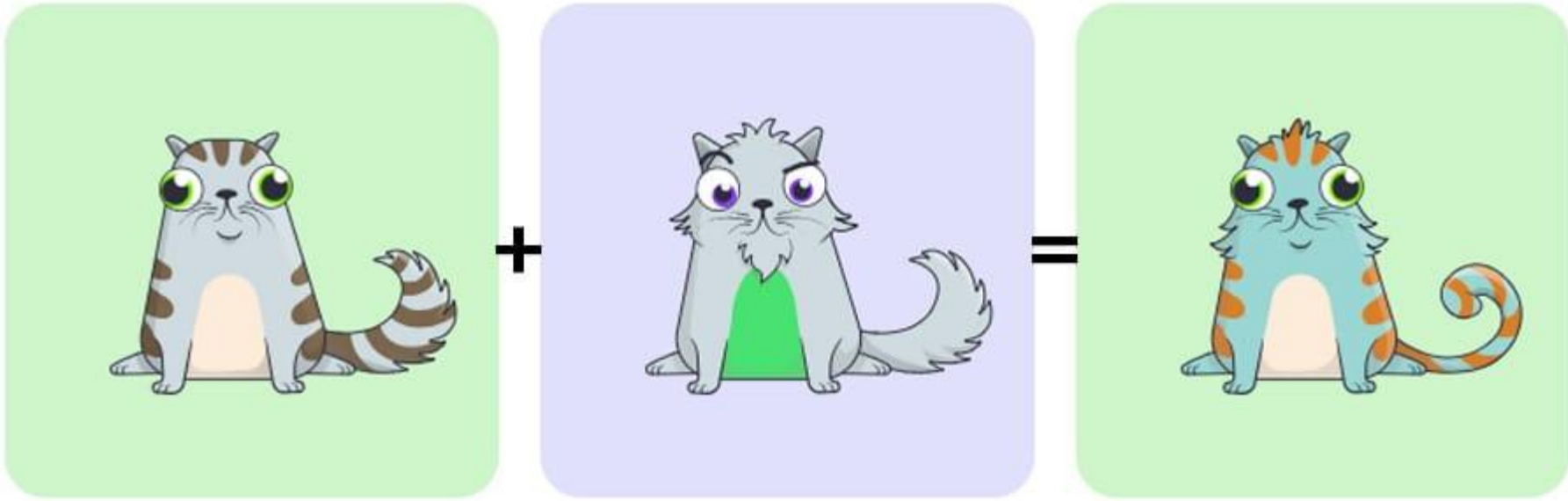
## Наследование

***Наследование –*** механизм, позволяющий описать класс на основе уже существующего.

***Родитель*** – класс, от которого происходит наследование.

***Потомок/наследник*** – класс, который произошел (был унаследован) от какого-то базового класса (класса-родителя).

Между родителем и потомком устанавливается отношение ***«является»***.



# Парадигмы ООП

## Наследование

Рассмотрим автомобиль и нашего жука.

Все мы знаем, что, вообще говоря, ***наш жук – это автомобиль***.
Как и VW Passat, как и BMW X6 и все остальные.

Жук ЯВЛЯЕТСЯ автомобилем.

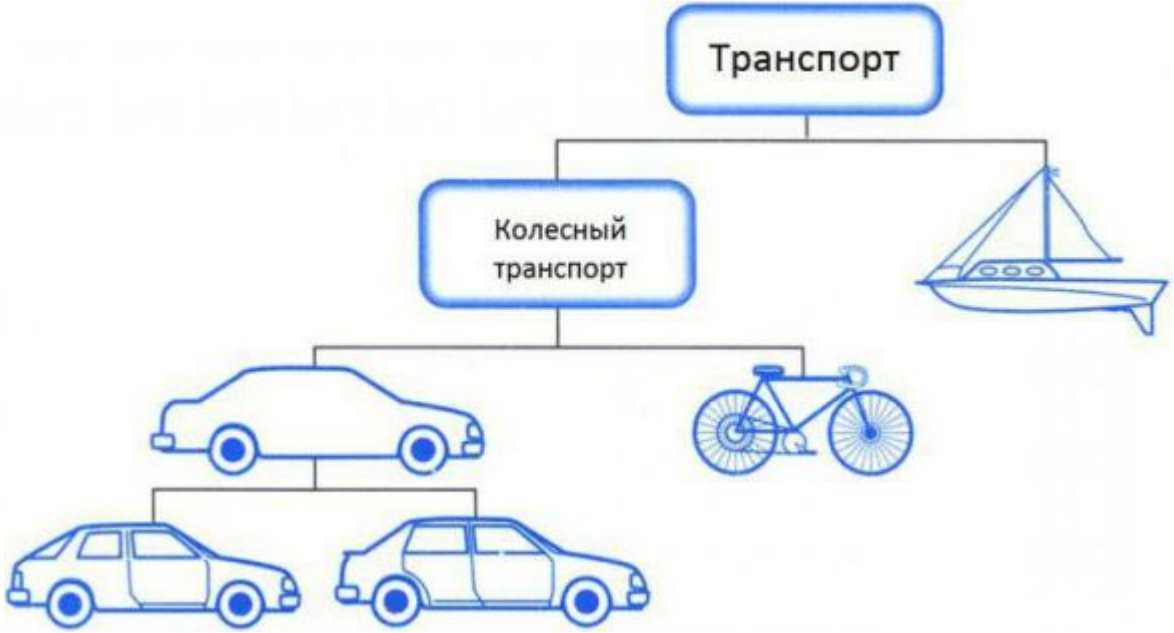
BMW X5 ЯВЛЯЕТСЯ автомобилем.
Ferrari F40 ЯВЛЯЕТСЯ автомобилем.

«Класс-потомок» ЯВЛЯЕТСЯ «Класс-родитель»

# Парадигмы ООП

## А в чем смысл?

- ***Класс-потомок может пользоваться данными и методами класса-родителя!***
- ***Класс-потомок может добавлять новые данные и методы!***
- ***Класс потомок может переопределять методы класса-родителя!***
- **А ЗНАЧИТ мы можем строить иерархии классов!**
- **Следовательно, нам нужно писать меньше кода!**

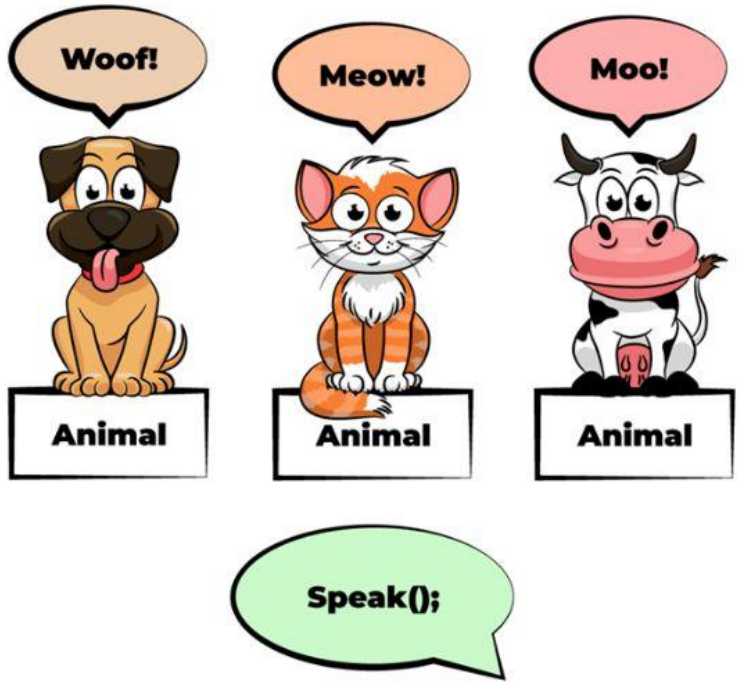


# Парадигмы ООП

## Полиморфизм

***Полиморфизм*** – способность функции обрабатывать различные типы входных данных.

Полиморфизмом называется 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.



# Парадигмы ООП

## Наглядный пример

Давайте опишем класс Person, представляющий отдельного человека:

```
1  public class Person {
2
3      private String name;
4      private String surname;
5
6      public Person(String name, String surname){
7
8          this.name=name;
9          this.surname=surname;
10     }
11
12     public String getName() {
13         return name;
14     }
15
16     public String getSurname() {
17         return surname;
18     }
19
20     public void displayInfo(){
21
22         System.out.println("Имя: " + name + " Фамилия: " + surname);
23     }
24 }
```

# Парадигмы ООП

## Наглядный пример

- В последствии, мы хотели бы расширить имеющуюся систему классов, добавив в нее класс, описывающий сотрудника предприятия – класс Employee;
- Сотрудник предприятия является человеком и, следовательно, имеет тот же функционал, что и класс Person;
- Поэтому, ничто не мешает нам сделать класс Employee производным от класса Person:

```
1 ▼ class Employee extends Person{
2
3 ▲ }
```

- Чтобы объявить один класс наследником другого, нужно использовать после имени класса-наследника ключевое слово extends, после которого указывается имя базового класса.

# Парадигмы ООП

## Наглядный пример

- В классе Employee могут быть определены свои поля, методы и конструктор:

```
1  class Employee extends Person{
2
3      private String company;
4
5      public Employee(String name, String surname, String company) {
6
7          super(name, surname);
8          this.company=company;
9      }
10 }
```

# Парадигмы ООП

## Наглядный пример

- Еще одно определение полиморфизма: полиморфизм – это способность к изменению функциональности, унаследованной от базового класса.
- Переопределим метод displayInfo() класса Person в классе Employee:

```
1  class Employee extends Person{
2
3      private String company;
4
5      public Employee(String name, String surname, String company) {
6
7          super(name, surname);
8          this.company=company;
9      }
10
11     public void displayInfo(){
12         super.displayInfo();
13         System.out.println("Компания: " + company);
14     }
15 }
```

# Парадигмы ООП

## Наглядный пример

- Класс Employee определяет дополнительное поле для хранения компании, в которой работает сотрудник. Кроме того, это поле устанавливается в конструкторе.
- Так как поля name и surname в базовом классе Person объявлены с модификатором доступа private, к ним нельзя обратиться из класса Employee напрямую. Однако, в данном случае, в этом нет необходимости, так как, чтобы их установить, нужно обратиться к конструктору базового класса с помощью ключевого слова super, после которого в скобках идет перечисление передаваемых аргументов.
- С помощью ключевого слова super можно обратиться к любом члену базового класса – методу или полю, если они не определены с модификатором доступа private.
- Также в классе Employee переопределяется метод displayInfo() базового класса.
- В нем, с помощью ключевого слова super, также идет обращение к методу displayInfo(), но уже БАЗОВОГО класса, а затем выводится дополнительная информация, относящаяся только к Employee.

# Парадигмы ООП

## Наглядный пример

- Используя обращение к методам базового класса, можно было бы переопределить метод displayInfo() следующим образом:

```
public void displayInfo(){
    System.out.println("Имя: " + super.getName() + " Фамилия: "
        + super.getSurname() + " Компания: " + company);
}
```

- При этом совершенно не обязательно переопределять все методы базового класса. В данном случае, не переопределяются методы getName() и getSurname(), поэтому для этих методов класс-наследник будет использовать реализацию из базового класса.
- Можно использовать эти методы в основной программе:

```
public static void main(String[] args) {
    Employee empl = new Employee("Tom", "Simpson", "Oracle");
    empl.displayInfo();
    String firstName = empl.getName();
    System.out.println(firstName);
}
```

# Парадигмы ООП

## Запрет наследования

- Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным.
- В этом случае можно запретить наследование с помощью ключевого слова final:

```
1  public final class Person {
2
3  }
```

- Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как наследование было запрещено:

```
1  class Employee extends Person{ {
2
3  }
```

# Парадигмы ООП

## Запрет наследования

- Кроме запрета наследования можно также запретить переопределение отдельных методов.
- Запретим переопределение метода displayInfo():

```
1  public class Person {
2
3      //.......................
4
5      public final void displayInfo(){
6
7          System.out.println("Имя: " + name + " Фамилия: " + surname);
8      }
9  }
```

# Абстрактные классы

- Кроме обычных классов в Java есть абстрактные классы.
- Абстрактный класс похож на обычный класс, в нем также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса.
- Абстрактные классы призваны предоставлять базовый функционал для классов-наследников, а производные классы, в свою очередь, реализуют этот функционал.
- При определении абстрактных классов используется ключевое слово abstract:

```
public abstract class Human{

    private String name;

    public String getName() { return name; }
}
```

- Но главное отличие абстрактного класса состоит в невозможности использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

**Human h = new Human()**;

# Абстрактные классы


- Кроме обычных методов абстрактный класс может содержать **абстрактные методы**.
- Такие методы определяются с помощью ключевого слова **abstract** и не имеют никакого функционала: **public abstract void display()**;
- Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе.
- Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе.
- Следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как **абстрактный**.

# Абстрактные классы

## Зачем нужны абстрактные классы

Предположим, что перед разработчиком стоит задача написать программу для обслуживания банковских операций.

В программе будут определены 3 класса:

- Person, который описывает человека;
- Employee, который описывает банковского служащего;
- Client, который представляет клиента банка.

Очевидно, что классы Employee и Client будут производными от класса Person, так как оба класса имеют некоторые общие поля и методы.

И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую от класса Person создаваться объекты не будут, поэтому имеет смысл сделать его абстрактным.

# Абстрактные классы

## Зачем нужны абстрактные классы

```java
public abstract class Person {

    private String name;
    private String surname;

    public String getName() { return name; }
    public String getSurname() { return surname; }

    public Person(String name, String surname){

        this.name=name;
        this.surname=surname;
    }

    public abstract void displayInfo();
}
```

```java
class Employee extends Person{

    private String bank;

    public Employee(String name, String surname, String company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Работает в банке: " + bank);
    }
}
```

```java
class Client extends Person
{
    private String bank;

    public Client(String name, String surname, String company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Клиент банка: " + bank);
    }
}
```

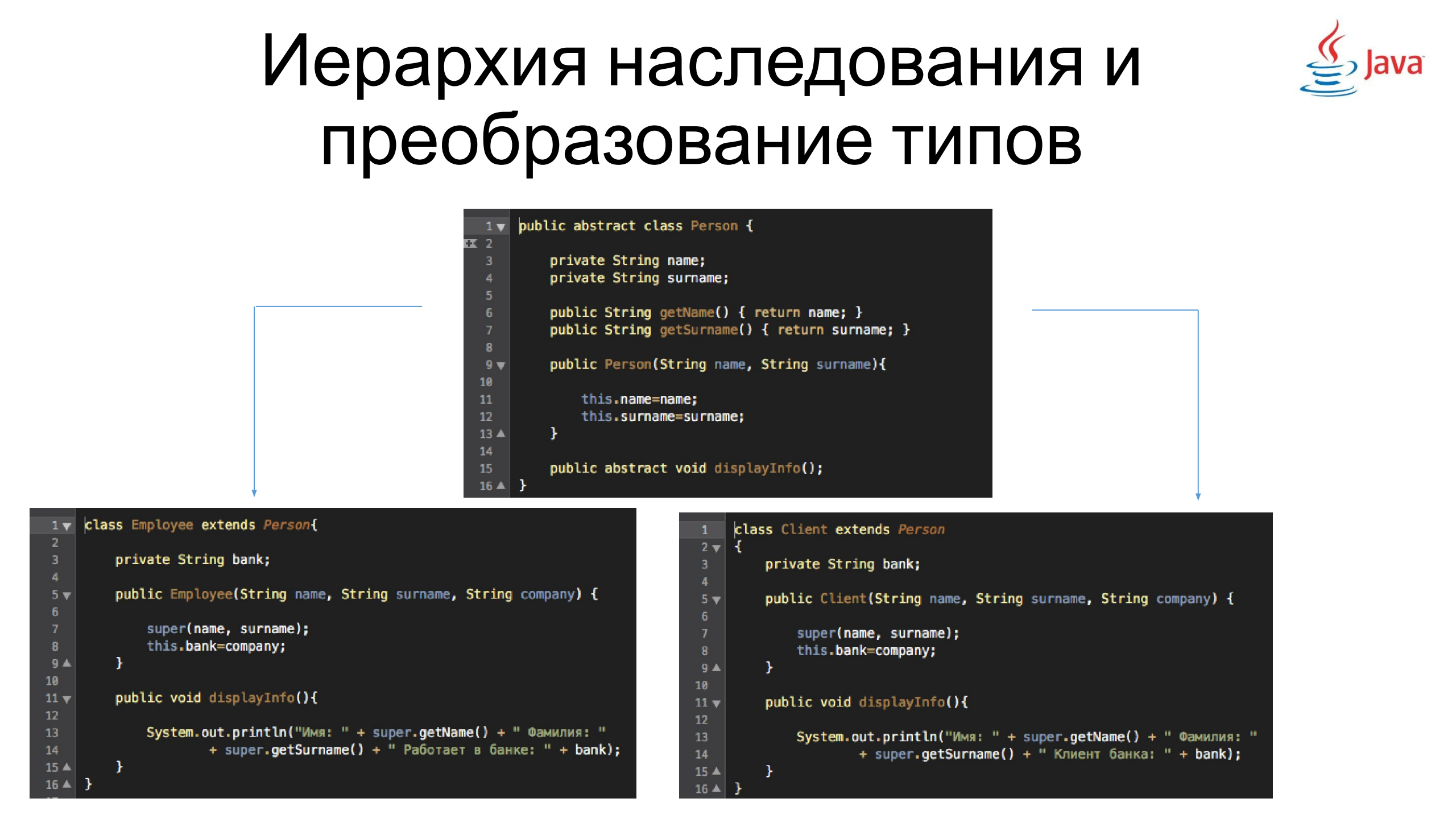
# Иерархия наследования и преобразование типов


Ранее говорилось о преобразовании объектов простых типов.

Однако с объектами классов все происходит немного по другому.

Вернемся к иерархии классов в предыдущем примере:

# Иерархия наследования и преобразование типов



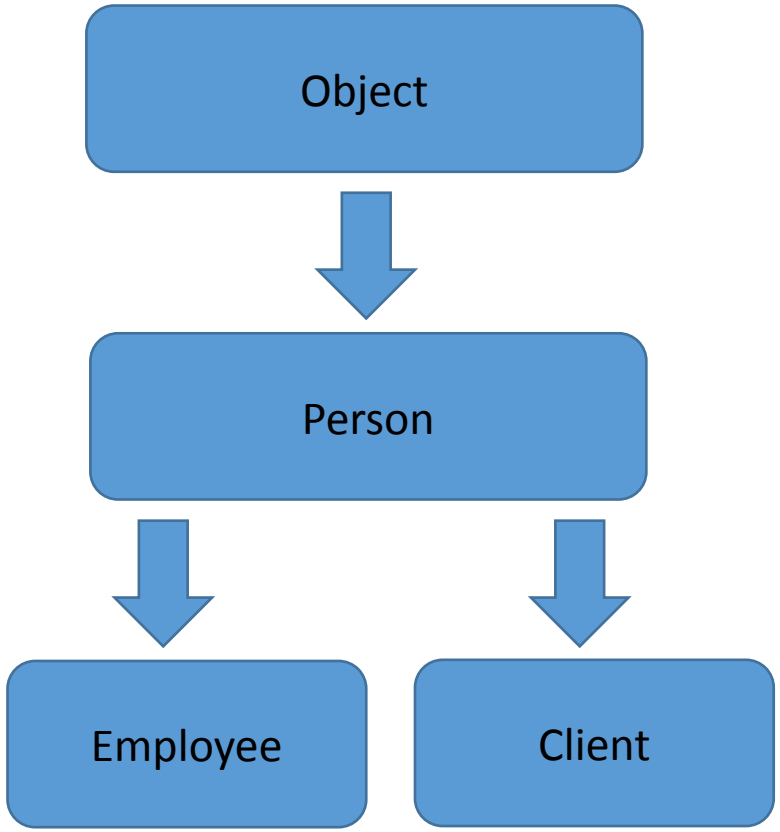
```java
public abstract class Person {

    private String name;
    private String surname;

    public String getName() { return name; }
    public String getSurname() { return surname; }

    public Person(String name, String surname){

        this.name=name;
        this.surname=surname;
    }

    public abstract void displayInfo();
}
```

```java
class Employee extends Person{

    private String bank;

    public Employee(String name, String surname, String company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Работает в банке: " + bank);
    }
}
```

```java
class Client extends Person
{
    private String bank;

    public Client(String name, String surname, String company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Клиент банка: " + bank);
    }
}
```

# Иерархия наследования и преобразование типов

В такой иерархии можно проследить следующую цепь наследования:



# Иерархия наследования и преобразование типов


Используем классы в программе и проведем несколько преобразований:

```
1   Object emp = new Employee("Bill", "Microsoft");
2
3   Person cl = new Client("Tom", "UnitBank", 200, 20);
4
5   //у класса Object нет метода displayInfo, поэтому приводим к классу Employee
6   ((Employee)emp).displayInfo();
7
8   // у класса Person есть метод displayInfo
9   cl.displayInfo();
10
11  // у класса Person нет метода getBank(), поэтому приводим к классу Client
12  String bank = ((Client)cl).getBank();
```

# Иерархия наследования и преобразование типов


- Здесь вначале создаются две переменные типов Object и Person, которые хранят ссылки на объекты Employee и Client соответственно.
- В данном случае работает неявное преобразование, так как наши переменные представляют классы Object и Person, поэтому допустимо неявное восходящее преобразование – преобразование к типам, которые находятся вверху иерархии классов.
- Однако, при **применении** этих переменных нам придется использовать явное преобразование.
- Поскольку переменная emp хранит ссылку на объект типа Employee, то можно выполнить преобразование к данному типу:

  ((Employee)emp).displayInfo();
- То же самое и с переменной cl.
- В то же время невозможно привести переменную emp к объекту типа Client – ((Client)emp).displayInfo(); так как класс Client не находится в иерархии классов между Employee и Object.

# Иерархия наследования и преобразование типов


- Добавим новый класс Manager, который будет наследоваться от класса Employee и поэтому будет находиться в самом низу иерархии классов:

```
class Manager extends Employee{

    public Manager(String name, String comp) {

        super(name,comp);
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Менеджер банка " + super.getCompany());
    }
}
```

- И поскольку объект класса Manager в то же время является и сотрудником банка (то есть, объектом Employee), можно использовать этот класс следующим образом:

```
Employee man = new Manager("John", "City Bank");
man.displayInfo(); // преобразование не нужно, так как в классе Employee есть метод displayInfo
```

# Иерархия наследования и преобразование типов


- Здесь снова видно восходящее преобразование Manager к Employee.
- И, так как метод displayInfo() есть у класса Employee, нет необходимости выполнять преобразование переменной к типу Manager.
- Однако, попробуем применить нисходящее преобразование неявно:

```
1   Manager man = new Employee("John", "City Bank");
2   man.displayInfo();
```

- В этом случае во время выполнения программы произойдет ошибка, ведь происходит попытка неявно преобразовать объект Employee к типу Manager, и, если Manager является объектом типа Employee, то объект Employee НЕ является типом Manager.
- Перед тем, как провести преобразование типов, можно проверить, возможно ли приведение типов с помощью оператора instanceof:

```
1   Employee emp = new Employee("John", "City Bank");
2   if(emp instanceof Manager){
3       ((Manager)emp).displayInfo();
4   }
5    else{
6       System.out.println("Преобразование не допустимо");
7   }
```

# Интерфейсы

- Механизм наследования очень удобен, но он имеет свои ограничения – наследование возможно только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.
- В языке Java подобную проблему частично позволяют решить интерфейсы.
- Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы.
- Один класс может применить ***множество*** интерфейсов.
- Чтобы определить интерфейс, используется ключевое слово interface:

```
interface Printable{

    void print();
}
```

# Интерфейсы


- В IDE интерфейс создается так же, как и обычный класс.
- В Intelij IDEA это можно сделать следующим образом: **File -> New -> Java Class** и в открывшемся окне выбирать тип **Interface;**
- Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации.
- Методы без реализации похожи на абстрактные методы абстрактных классов.
- В примере выше, в интерфейсе Printable объявлен один метод, который не имеет реализации.
- Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ у них public, так как цель интерфейса – определение функционала для реализации его классом, и весь функционал должен быть открыт для реализации.
- При объявлении интерфейса надо учитывать, что только один интерфейс в файле может иметь тип доступа public, а его название должно совпадать с именем файла.
- Остальные интерфейсы (если такие имеются в файле java) НЕ должны иметь модификаторов доступа.

# Интерфейсы

- Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**.

```
class Book implements Printable{
    String name;
    String author;
    int year;

    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }

    @Override
    public void print() {
        System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name, author, year);
    }
}
```

- При этом надо учитывать - если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print().

# Интерфейсы

- Потом, в главном классе мы можем использовать данный класс и его метод print():

```
1  Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
2  b1.print();
```

- В то же время нельзя напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

```
1  Printable pr = new Printable();
2  pr.print();
```

# Интерфейсы


## Для чего нужны интерфейсы

- Простой пример интерфейса из повседневной жизни — пульт от телевизора.
- Он связывает два объекта, *человека* и *телевизор*, и выполняет разные задачи: *прибавить или убавить звук*, *переключить каналы*, *включить или выключить телевизор*.
- Одной стороне (*человеку*) нужно обратиться к интерфейсу (*нажать на кнопку пульта*), чтобы вторая сторона выполнила действие, например, чтобы телевизор переключил канал на следующий.
- При этом пользователю не обязательно знать устройство телевизора и то, **как** внутри него реализован процесс смены канала.
- Все, к чему пользователь имеет доступ — это ***интерфейс***.
- Главная задача — получить нужный результат.

# Интерфейсы

## Для чего нужны интерфейсы

- Интерфейс описывает поведение, которым должны обладать классы, реализующие этот интерфейс.
- «Поведение» — это совокупность методов.

Предположим, что мы хотим создать несколько мессенджеров.

*Что должен уметь любой мессенджер?*

*В упрощенном виде, принимать и отправлять сообщения*.

Давайте создадим интерфейс Messenger:

```
public interface Messenger{

    public void sendMessage();

    public void getMessage();
}
```

# Интерфейсы

## Для чего нужны интерфейсы

- И теперь мы можем просто создавать наши классы-мессенджеры, имплементируя этот интерфейс.
- Компилятор сам «заставит» нас реализовать их внутри классов.

Telegram

```
public class Telegram implements Messenger {

    public void sendMessage() {

        System.out.println("Отправляем сообщение в Telegram!");
    }

     public void getMessage() {
         System.out.println("Читаем сообщение в Telegram!");
     }
}
```

# Интерфейсы

## Для чего нужны интерфейсы

WhatsApp

```
public class WhatsApp implements Messenger {

    public void sendMessage() {

        System.out.println("Отправляем сообщение в WhatsApp!");
    }

     public void getMessage() {
         System.out.println("Читаем сообщение в WhatsApp!");
     }
}
```

Viber

```
public class Viber implements Messenger {

    public void sendMessage() {

        System.out.println("Отправляем сообщение в Viber!");
    }

     public void getMessage() {
         System.out.println("Читаем сообщение в Viber!");
     }
}
```

Какие преимущества это дает? Самое главное из них — **слабая связанность**.

# Интерфейсы

## Для чего нужны интерфейсы

- Давайте представим, что мы проектируем программу, в которой у нас будут собраны данные клиентов.
- В классе Client обязательно нужно поле, указывающее, каким именно мессенджером клиент пользуется.

*Без интерфейсов это выглядело бы следующим образом*:

```
public class Client {

    private WhatsApp whatsApp;
    private Telegram telegram;
    private Viber viber;
}
```

# Интерфейсы


## Для чего нужны интерфейсы

- В данном случае, у клиента мы создали три поля для трех разных мессенджеров.
- Но у клиента может быть только один мессенджер, просто мы не знаем какой именно.
- Получается, один или два из этих полей будут всегда равны null, да и вообще не нужны для работы программы.
- Так не лучше ли использовать наш интерфейс?

```
public interface Messenger{

    public void sendMessage();

    public void getMessage();
}
```

Это и есть пример «слабой связанности»! Вместо того, чтобы указывать конкретный класс мессенджера в классе Client, мы просто упоминаем, что у клиента есть мессенджер, а какой именно — определится в ходе работы программы.

# Интерфейсы

## Для чего нужны интерфейсы

Но зачем нам для этого именно интерфейсы? Зачем их вообще добавили в язык?

Давайте разберемся.

Предположим, класс Messenger — родительский, а Viber, Telegram и WhatsApp — наследники.

Но есть одна загвоздка - множественного наследования в Java нет, а вот множественная реализация интерфейсов — есть.

Т.е. класс может реализовывать сколько угодно интерфейсов.

Представьте, что у нас есть класс ***Smartphone***, у которого есть поле ***Application*** — установленное на смартфоне приложение.

```
public class Smartphone {

    private Application application;
}
```

# Интерфейсы

## Для чего нужны интерфейсы

Приложение и мессенджер, конечно, похожи, но все-таки это разные вещи.

Мессенджер может быть и мобильным, и десктопным, в то время как Application — это именно *мобильное* приложение.

Если бы мы использовали наследование, то не смогли бы добавить объект Telegram в класс Smartphone.

Ведь класс Telegram не может наследоваться одновременно от Application и от Messenger!

А мы уже успели унаследовать его от Messenger, и в таком виде добавить в класс Client.

Но вот реализовать оба интерфейса класс Telegram запросто может!

Поэтому в классе Client мы сможем внедрить объект Telegram как Messenger, а в класс Smartphone — как Application.

Вот как это делается:

# Интерфейсы

## Для чего нужны интерфейсы

```
public class Telegram implements Application, Messenger {

    //...методы
}

public class Client {

    private Messenger messenger;

    public Client() {
        this.messenger = new Telegram();
    }
}


public class Smartphone {

    private Application application;

    public Smartphone() {
        this.application = new Telegram();
    }
}
```

# Интерфейсы

## Для чего нужны интерфейсы

Теперь можно использовать класс Telegram и в роли Application, и в роли Messenger.

Стоит еще раз обратить внимание, что методы в интерфейсах всегда пустые, т.е., не имеют реализации.

Причина этого проста - интерфейс ***описывает*** поведение, а не реализует его.

# Интерфейсы – методы по умолчанию

- Ранее до JDK 8 при реализации интерфейса необходимо было обязательно реализовать все его методы в классе, а сам интерфейс мог содержать только определения методов без конкретной реализации.
- В JDK 8 была добавлена такая функциональность как методы по умолчанию.
- Начиная с этой версии интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод.

Пример:

```
interface Printable {

    default void print(){

        System.out.println("Undefined printable");
    }
}
```

```
class Journal implements Printable {

    private String name;

    String getName(){
        return name;
    }
    Journal(String name){

        this.name = name;
    }
}
```

# Интерфейсы – статические методы


- Начиная с JDK 8 в интерфейсах доступны также статические методы - они аналогичны методам класса:

```
interface Printable {

    void print();

    static void read(){

        System.out.println("Read printable");
    }
}
```

- Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

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

    Printable.read();
}
```

# Интерфейсы – private методы

- По умолчанию все методы в интерфейсе фактически имеют модификатор public.
- Однако начиная с Java 9 стало возможным определять в интерфейсе методы с модификатором private.
- Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию и должны использоваться только внутри самого интерфейса:

```
interface Calculatable{

    default int sum(int a, int b){
        return sumAll(a, b);
    }
    default int sum(int a, int b, int c){
        return sumAll(a, b, c);
    }

    private int sumAll(int... values){
         int result = 0;
         for(int n : values){
             result += n;
         }
         return result;
    }
}
```

```
class Calculation implements Calculatable{

}
```

```
public class Program{

    public static void main(String[] args) {

        Calculatable c = new Calculation();
        System.out.println(c.sum(1, 2));
        System.out.println(c.sum(1, 2, 4));
    }
}
```

# Константы в интерфейсах

- Кроме методов в интерфейсах могут быть определены статические константы:

```
class WaterPipe implements Stateable{

    public void printState(int n){
        if(n==OPEN)
            System.out.println("Water is opened");
        else if(n==CLOSED)
            System.out.println("Water is closed");
        else
            System.out.println("State is invalid");
    }
}
```

```
interface Stateable{

    int OPEN = 1;
    int CLOSED = 0;

    void printState(int n);
}
```

```
public class Program{

    public static void main(String[] args) {

        WaterPipe pipe = new WaterPipe();
        pipe.printState(1);
    }
}
```

- Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа ***public static final***, и поэтому их значение доступно из любого места программы.

# Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

```
interface BookPrintable extends Printable{

    void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

# Вложенные интерфейсы

```
class Printer{
    interface Printable {

        void print();
    }
}
```

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах.

```
public class Journal implements Printer.Printable {

    String name;

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

При применении такого интерфейса нам надо указывать его полное имя вместе с именем класса.

```
Printer.Printable p =new Journal("Foreign Affairs");
p.print();
```

Использование интерфейса будет аналогично предыдущим случаям.

# Интерфейсы как параметры и результаты методов

Интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа:

```java
public class Program {
    public static void main(String[] args) {

        Printable printable = createPrintable( name: "Foreign Affairs", option: false);
        printable.print();

        read(new Book( name: "Java for impatients",  author: "Cay Horstmann"));
        read(new Journal( name: "Java Dayly News"));
    }

    static void read(Printable p){

        p.print();
    }

    static Printable createPrintable(String name, boolean option){

        if(option)
            return new Book(name,  author: "Undefined");
        else
            return new Journal(name);
    }
}
```

```java
public class Journal implements Printable {
    private String name;

    String getName(){
        return name;
    }

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

```java
public interface Printable {
    void print();
}
```

```java
public class Book implements Printable {
    String name;
    String author;

    Book(String name, String author){

        this.name = name;
        this.author = author;
    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);
    }
}
```

# Интерфейсы как параметры и результаты методов

- Метод read() в качестве параметра принимает объект интерфейса Printable, поэтому в этот метод мы можем передать как объект Book, так и объект Journal.

- Метод createPrintable() возвращает объект Printable, поэтому мы можем возвратить как объект Book, так и Journal.

Консольный вывод:

```
Foreign Affairs
Java for impatients (Cay Horstmann)
Java Dayly News
```

# Интерфейсы в механизме обратного вызова


Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова.

**Суть** обратного вызова состоит в том, что программист создает действия, которые вызываются при других действиях.

*Стандартный пример - нажатие на кнопку*:

- когда пользователь нажимает на кнопку - он производит действие, но, в ответ на это нажатие, запускаются другие действия.
- Например, нажатие на значок принтера запускает печать документа на принтере и т.д.

# Интерфейсы в механизме обратного вызова

Давайте рассмотрим пример.

```java
public class EventsApp {
    public static void main(String[] args) {
        Button button = new Button(new ButtonClickHandler());
        button.click();
        button.click();
        button.click();
    }
}

class ButtonClickHandler implements EventHandler{
    public void execute(){
        System.out.println("Кнопка нажата!");
    }
}

interface EventHandler{
    void execute();
}

class Button{
    EventHandler handler;

    Button(EventHandler action){
        this.handler = action;
    }

    public void click(){
        handler.execute();
    }
}
```

# Интерфейсы в механизме обратного вызова

- Итак, определим класс Button, который в конструкторе принимает объект интерфейса EventHandler и в методе click() (имитация нажатия) вызывает метод execute() этого объекта.
- Далее определяется реализация EventHandler в виде класса ButtonClickHandler.
- В основной программе объект этого класса передается в конструктор класса Button.
- Таким образом, через конструктор устанавливается обработчик нажатия кнопки, который будет вызываться при каждом вызове метода button.click().
- В итоге программа выведет на консоль следующий результат:

```
Кнопка нажата!
Кнопка нажата!
Кнопка нажата!
```

# Интерфейсы в механизме обратного вызова


Но, казалось бы, зачем выносить все действия в интерфейс, а потом его реализовывать? Почему бы не написать класс Button с методом execute() и вызвать этот метод у объекта класса Button?

```
1 ▼ class Button{
2 ▼     public void click(){
3           System.out.println("Кнопка нажата!");
4 ▲     }
5 ▲ }
```

Дело в том, что на момент определения класса не всегда бывают точно известны те действия, которые должны производиться.
Особенно если класс Button и класс основной программы находятся в разных пакетах, библиотеках и могут проектироваться разными разработчиками.
К тому же может быть несколько кнопок - объектов Button - для каждого из которых надо определить свое действие.

# Интерфейсы в механизме обратного вызова

Давайте изменим главный класс программы.

```
public class EventsApp {
    public static void main(String[] args) {
        Button tvButton = new Button(new EventHandler(){
            private boolean on = false;
            public void execute(){
                if(on) {
                    System.out.println("Телевизор выключен..");
                    on=false;
                } else {
                    System.out.println("Телевизор включен!");
                    on=true;
                }
            }
        });

        Button printButton = new Button(new EventHandler(){
            public void execute(){
                System.out.println("Запущена печать на принтере...");
            }
        });
        tvButton.click();
        printButton.click();
        tvButton.click();
    }
}
```

# Интерфейсы в механизме обратного вызова


- Здесь имеется две кнопки - одна для включения-выключения телевизора, а другая для печати на принтере.
- Вместо того, чтобы создавать отдельные классы, реализующие интерфейс EventHandler, обработчики задаются в виде анонимных объектов, которые реализуют интерфейс EventHandler.
- Причем обработчик кнопки телевизора хранит дополнительное состояние в виде логической переменной on.
- В итоге консоль выведет нам следующий результат:В итоге консоль выведет нам следующий результат:

```
Телевизор включен!
Запущена печать на принтере...
Телевизор выключен..
```

- Интерфейсы в данном качестве особенно широко используются в различных графических API - AWT, Swing, JavaFX, где обработка событий объектов - элементов графического интерфейса – особенно актуальна.

# Перечисления (enum) в Java

Представим, что перед разработчиком поставили задачу создать класс, представляющий дни недели.
На первый взгляд, ничего сложного в этом нет, и код будет выглядеть следующим образом:

```
public class DayOfWeek {

    private String title;

    public DayOfWeek(String title) {
        this.title = title;
    }

    public static void main(String[] args) {
        DayOfWeek dayOfWeek = new DayOfWeek("Суббота");
        System.out.println(dayOfWeek);
    }

    @Override
    public String toString() {
        return "DayOfWeek{" +
                "title='" + title + '\'' +
                '}';
    }
}
```

# Перечисления (enum) в Java


Вроде бы, все в порядке, НО в конструктор класса DayOfWeek можно передать любой текст. Таким образом, кто-то сможет создать день недели «Лягушка», «Облачко» или «azaza322». А это далеко не то поведение, которое ожидает от класса разработчик и пользователи, ведь реальных дней недели существует всего 7, и у каждого из них есть название.

Перед разработчиком появляется еще одна важная задача - как-то ограничить круг возможных значений для класса «день недели».

До появления Java 1.5 разработчики были вынуждены самостоятельно придумывать решение этой проблемы, поскольку готового решения в самом языке не существовало.

В те времена, если ситуация требовала ограниченного числа значений, делали так:

# Перечисления (enum) в Java


```
public class DayOfWeek {

    private String title;

    private DayOfWeek(String title) {
        this.title = title;
    }

    public static DayOfWeek SUNDAY = new DayOfWeek("Воскресенье");
    public static DayOfWeek MONDAY = new DayOfWeek("Понедельник");
    public static DayOfWeek TUESDAY = new DayOfWeek("Вторник");
    public static DayOfWeek WEDNESDAY = new DayOfWeek("Среда");
    public static DayOfWeek THURSDAY = new DayOfWeek("Четверг");
    public static DayOfWeek FRIDAY = new DayOfWeek("Пятница");
    public static DayOfWeek SATURDAY = new DayOfWeek("Суббота");

    @Override
    public String toString() {
        return "DayOfWeek{" +
                "title='" + title + '\'' +
                '}';
    }
}
```

# Перечисления (enum) в Java

В чем здесь особенность:

- **Закрытый (private) конструктор.** Если конструктор помечен модификатором private, объект класса нельзя создать с помощью этого конструктора. А поскольку в этом классе конструктор всего один, объект DayOfWeek нельзя создать вообще.

```
public class Main {

        public static void main(String[] args) {

                DayOfWeek sunday = new DayOfWeek();//ошибка!
        }
}
```

- Нужное количество public static объектов, представляющих правильные названия дней недели, что позволяло использовать объекты в других классах.

```
public class Man {

        public static void main(String[] args) {

                DayOfWeek sunday = DayOfWeek.SUNDAY;

                System.out.println(sunday);
         }
}
```

# Перечисления (enum) в Java


Такой подход во многом позволял решить задачу. В распоряжении пользователя были 7 дней недели, и при этом никто не мог создать новые.

С выходом Java 1.5 в языке появилось готовое решение для таких ситуаций — перечисление (Enum).

Enum — тоже класс, но он специально «заточен» на решение задач, похожих на пример выше - создание некоторого ограниченного круга значений.

Поскольку у создателей Java уже были готовые примеры (скажем, язык C, в котором Enum уже существовал), они смогли создать оптимальный вариант.

Итак, что же из себя представляет Enum в Java? Снова начнем с примера.

# Перечисления (enum) в Java


Давайте опишем с помощью enum тип данных для хранения времени года:

```
enum Season { WINTER, SPRING, SUMMER, AUTUMN }
```

Ну и простой пример его использования:

```
Season season = Season.SPRING;
if (season == Season.SPRING) season = Season.SUMMER;
System.out.println(season);
```

В результате выполнения которого на консоль будет выведено ***SUMMER***.

# Перечисления (enum) в Java


**Перечисление — это класс.**

- Объявляя enum мы неявно создаем класс, производный от *java.lang.Enum*.
- Условно, конструкция enum Season { ... } эквивалентна ***class Season extends java.lang.Enum { ... }***.
- И, хотя, явным образом наследоваться от java.lang.Enum не позволяет компилятор, все же в том, что enum наследуется, легко убедиться с помощью reflection:

```
System.out.println(Season.class.getSuperclass());
```

- В консоль будет выведен: `class java.lang.Enum`
- Наследование за разработчика автоматически выполняет компилятор Java.

# Перечисления (enum) в Java


## Элементы перечисления — экземпляры enum-класса, доступные статически.

- Элементы enum Season (WINTER, SPRING и т.д.) — это статически доступные экземпляры enum-класса Season.
- Их статическая доступность позволяет выполнять сравнение с помощью оператора сравнения ссылок ==

```
Season season = Season.SUMMER;
if (season == Season.AUTUMN) season = Season.WINTER;
```

## Методы перечислений.

- name() – возвращает имя константы, определенной в перечислении.
- toString() - возвращает имя константы, определенной в перечислении, и является наиболее используемым при работе с перечислениями.
- ordinal() - возвращает порядковый номер определенной константы (нумерация начинается с 0).

# Перечисления (enum) в Java

```java
package enumerations;

public class EnumClass {
    public static void main(String[] args) {
        Season season = Season.WINTER;
        System.out.println("season.name()=" + season.name()
                + " season.toString()=" + season.toString()
                + " season.ordinal()=" + season.ordinal());
    }
}
```

```
Run:  EnumClass
"C:\Program Files\java8\bin\java.exe" ...
season.name()=WINTER season.toString()=WINTER season.ordinal()=0

Process finished with exit code 0
```

Пример использования методов name(), toString() и ordinal().

# Перечисления (enum) в Java

## Методы перечислений.

- valueOf(String name) – получение элемента enum по строковому представлению его имени.

```
String name = "WINTER";
Season season = Season.valueOf(name);
```

В результате выполнения кода переменная season будет равна Season.WINTER.

- Следует обратить внимание, что если элемент не будет найден, то будет выброшен IllegalArgumentException, а в случае, если name равен null — NullPointerException.
- values() – получение всех элементов перечисления.

```
System.out.println(Arrays.toString(Season.values()));
```

```
[WINTER, SPRING, SUMMER, AUTUMN]
```

# Перечисления (enum) в Java

## Пользовательские методы в перечислениях.

Существует возможность добавлять собственные методы как в enum-классы, так и в их элементы:

```
package enumerations;

public enum Direction {
    UP, DOWN;

    public Direction opposite(){
        return this ==UP ? DOWN : UP;
    }
}
```

```
package enumerations;

public class EnumClass {
    public static void main(String[] args) {
        Direction direction = Direction.UP;
        System.out.println(direction.opposite());
    }
}
```

# Перечисления (enum) в Java


## Наследование в enum. Полиморфный подход к перечислениям.

С помощью enum в Java можно реализовать иерархию классов, объекты которой создаются в единственном экземпляре и доступны статически.

При этом элементы enum могут содержать собственные конструкторы.

Давайте рассмотрим пример:

# Перечисления (enum) в Java


```java
package enumerations;

public enum Type {
    INT(true){
        public Object parse(String value){
            return Integer.valueOf(value);
        }
    },
    INTEGER(false){
        public Object parse(String value){
            return Integer.valueOf(value);
        }
    },
    STRING(false){
        public Object parse(String value){
            return value;
        }
    };

    boolean primitive;
    Type(boolean primitive){
        this.primitive = primitive;
    }

    public boolean isPrimitive(){
        return  primitive;
    }

    public abstract Object parse(String value);
}
```

```java
package enumerations;

public class EnumClass {
    public static void main(String[] args) {
        Type type = Type.INTEGER;
        System.out.println(type.parse( value: "1"));
    }
}
```

```
"C:\Program Files\java8\bin\java.exe" ...
1

Process finished with exit code 0
```

# Перечисления (enum) в Java


## Наследование в enum. Полиморфный подход к перечислениям.

Здесь объявляется перечисление Type с тремя элементами INT, INTEGER и STRING.

Компилятор создаст следующие классы и объекты:

- Type — класс производный от java.lang.Enum;
- INT — объект 1-го класса производного от Type;
- INTEGER — объект 2-го класса производного от Type;
- STRING — объект 3-го класса производного от Type.

Три производных класса будут созданы с полиморфным методом Object parse(String) и конструктором Type(boolean primitive).

При этом объекты классов INT, INTEGER и STRING существуют в единственном экземпляре и доступны статически.

В этом можно убедится:

# Перечисления (enum) в Java

## Наследование в enum. Полиморфный подход к перечислениям.

```
System.out.println(Type.class);
System.out.println(Type.INT.getClass() + " " + Type.INT.getClass().getSuperclass());
System.out.println(Type.INTEGER.getClass() + " " + Type.INTEGER.getClass().getSuperclass());
System.out.println(Type.STRING.getClass()  + " " + Type.STRING.getClass().getSuperclass());
```

Вывод:

```
class Type
class Type$1 class Type
class Type$2 class Type
class Type$3 class Type
```

Видно, что компилятор создал класс Type и 3 nested класса, производных от Type.

# Класс Object и его методы


- В Java есть специальный суперкласс Object, и все классы являются его подклассами.
- Ссылочная переменная класса Object может ссылаться на объект любого другого класса.
- Так как массивы также являются классами, то переменная класса Object может ссылаться и на любой массив.
- К классу Object можно привести любой класс:

  *Object obj = new Cat("Barsik");*

- Правда, в таком виде объект, обычно, не используют.
- Чтобы с объектом что-то сделать, нужно выполнить приведение типов:

  *Cat cat = (Cat) obj;*

# Класс Object и его методы

| Метод | Описание |
|---|---|
| public String toString() | Возвращает строковое представление объекта. |
| public native int hashCode()<br>public boolean equals(Object obj) | Пара методов, которые используются для сравнения объектов. |
| public final native Class getClass() | Возвращает специальный объект, который описывает текущий класс. |
| public final native void notify()<br>public final native void notifyAll()<br>public final native void wait(long timeout)<br>public final void wait(long timeout, intnanos)<br>public final void wait() | Методы для контроля доступа к объекту из различных потоков. Управление синхронизацией потоков. |
| protected void finalize() | Метод позволяет «освободить» родные не-Java ресурсы: закрыть файлы, потоки и т.д. |
| protected native Object clone() | Метод позволяет клонировать объект: создает дубликат объекта. |

# Класс Object и его методы


### *toString() метод*

- Этот метод позволяет получить текстовое описание любого объекта или его *строковое представление*:

```
return getClass().getName() + "@" + Integer.toHexString(hashCode());
```

- Стандартный результат вызова такого метода:

```
java.lang.Object@12F456
```

- *Строковое представление объекта* состоит из полного имени объекта с именем пакета, знака @ и хэш-кода объекта в шестнадцатеричном виде.
- Ценность этого метода в том, что его можно переопределить в любом классе и возвращать более нужное или более детальное описание объекта.
- Более того, благодаря тому, что для каждого объекта можно получить его текстовое представление, в Java можно реализовать поддержку «сложения» строк с объектами.

# Класс Object и его методы


***toString() метод***

| Код | Что происходит на самом деле |
|---|---|
| int age = 18;<br>System.out.println("Age is " + age); | String s = String.valueOf(18);<br>String result = "Age is " + s;<br>System.out.println(result); |
| Student st = new Student("Vasya");<br>System.out.println("Student is " + st); | Student st = new Student("Vasya");<br>String result = "Student is " + st.toString();<br>System.out.println(result); |
| Car car = new Porsche();<br>System.out.println("My car is " + car); | Car car = new Porsche();<br>String result = "My car is " + car.toString();<br>System.out.println(result); |

***В классах Student и Car метод toString() должен быть переопределен, если мы желаем увидеть информацию о полях класса.***

# Класс Object и его методы


***toString() метод***

| Код | Что происходит на самом деле |
|---|---|
| int age = 18;<br>System.out.println("Age is " + age); | String s = String.valueOf(18);<br>String result = "Age is " + s;<br>System.out.println(result); |
| Student st = new Student("Vasya");<br>System.out.println("Student is " + st); | Student st = new Student("Vasya");<br>String result = "Student is " + st.toString();<br>System.out.println(result); |
| Car car = new Porsche();<br>System.out.println("My car is " + car); | Car car = new Porsche();<br>String result = "My car is " + car.toString();<br>System.out.println(result); |

***В классах Student и Car метод toString() должен быть переопределен, если мы желаем увидеть информацию о полях класса.***

# Обобщенные типы (Generics)


- На протяжении всей истории развития языка Java, он претерпевал изменения.

- Иногда изменения носили косметический характер, иногда это было просто исправление уязвимостей, а иногда переход на новую версию языка знаменовал поистине значительные, а в некоторых случаях и революционные, изменения. Одним из таких изменений стали обобщения (generics).

- Обобщения в Java были представлены в версии 5.0 - это было результатом реализации самых первых требований к спецификации Java, которые были сформулированы еще в 1999 году.

- Они позволили создавать более безопасный и легче читаемый код, который не перегружен переменными типа Object и приведением классов

# Обобщенные типы (Generics)


Наверняка, большинство уже сталкивалось с подобного рода объявлениями:

```
ArrayList<String> list = new ArrayList<String>();
```

В качестве обобщенного типа, в данном случае, выступает тип String, указанный в треугольных скобках.

Начиная с версии Java 7 SE обобщенный тип можно опускать в объявлении конструктора класса:

```
ArrayList<String> list = new ArrayList<>();
```

# Обобщенные типы (Generics)

## Причины появления обобщенных типов

Основное назначение обобщенных типов – более строгая проверка типов во время компиляции и устранение необходимости явного приведения.

Рассмотрим пример:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args){
        List list = new ArrayList();
        list.add("Hello");
        String text = list.get(0) + ", world!";
        System.out.print(text);
    }
}
```

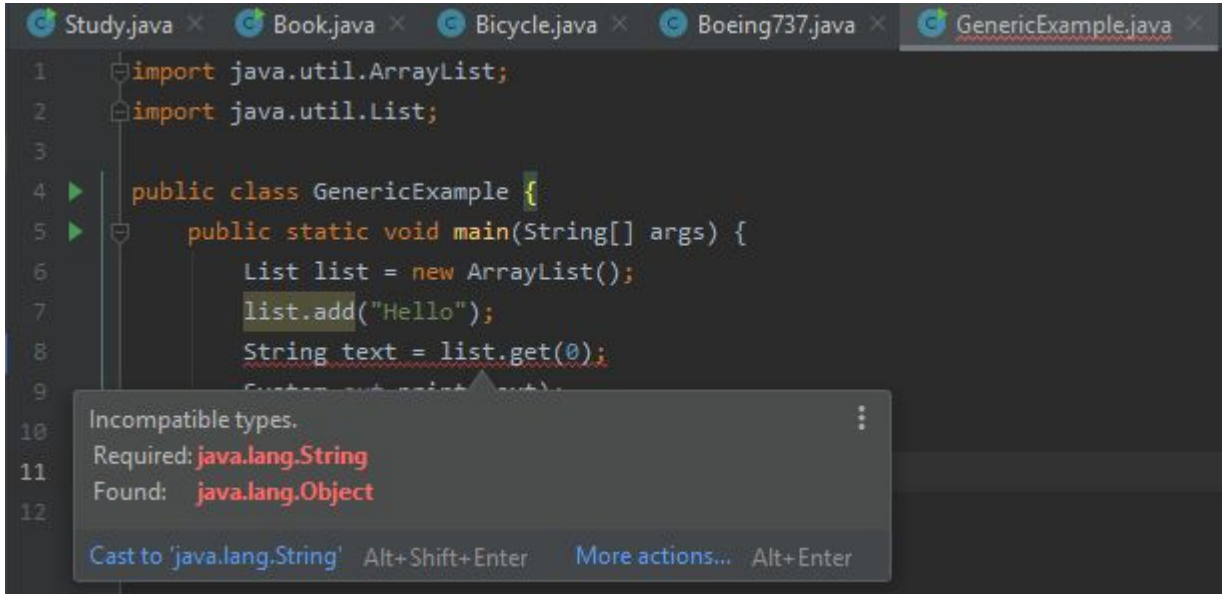
# Обобщенные типы (Generics)

## Причины появления обобщенных типов

Проблем с компиляцией данного кода не возникнет.

Однако, что если заказчик продукта скажет, что фраза "Hello, world!" избита и нужно вернуть только Hello?

Уберем ", world!"  часть и получим:



# Обобщенные типы (Generics)

## Причины появления обобщенных типов

Всё дело в том, что коллекция List хранит список объектов типа Object.

Так как тип String наследуется от класса Object (как и все классы в Java), то требует явного приведения типов, чего по факту не происходит.

Однако, в первом случае, при конкатенации строк для объекта будет вызван статический метод String.valueOf(list.get(0)), который в итоге вызовет метод toString() для Object, и, следовательно, вот эта часть list.get(0) преобразуется к типу String.

Во втором случае происходит попытка присвоить переменной типа String объект типа Object без какой-либо конкатенации, и, поскольку в данном случае требуется выполнить явное преобразование, компилятор выдает ошибку:

***incompatible types: Object cannot be converted to String***

# Обобщенные типы (Generics)

## Причины появления обобщенных типов

Выходит, там где нам нужен конкретный тип, а не Object, необходимо делать приведение типов:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args){
        List list = new ArrayList();
        list.add("Hello!");
        list.add(123);
        for (Object str : list) {
            System.out.println("-" + (String)str);
        }
    }
}
```

Однако, в данном случае, List хранит не только String, но и Integer.

# Обобщенные типы (Generics)


## Причины появления обобщенных типов

И самое плохое в этом случае – компилятор не увидит ничего подозрительного. Ошибка будет брошена уже ВО ВРЕМЯ ВЫПОЛНЕНИЯ (на Runtime):

***java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String***

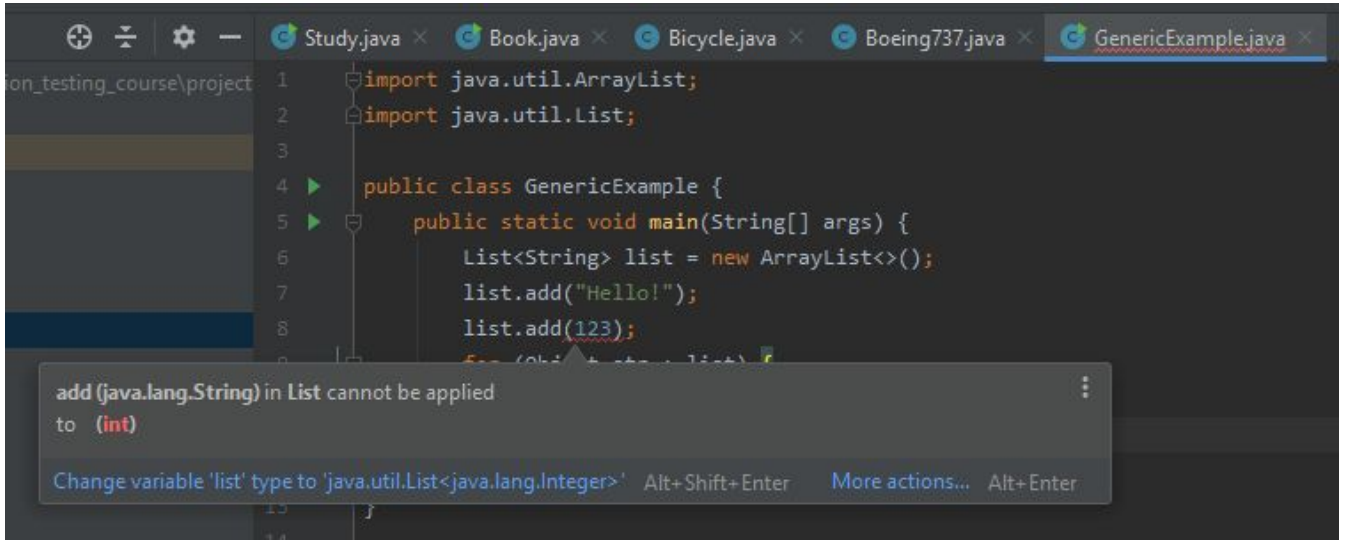
Компилятор — не искусственный интеллект и не может угадать всё, что подразумевает программист.

А чтобы рассказать ему подробнее о том, какие типы планируется использовать, в Java SE 5 ввели дженерики.

# Обобщенные типы (Generics)

## Причины появления обобщенных типов

```
import java.util.*;
public class HelloWorld {
    public static void main(String []args){
        List<String> list = new ArrayList<>();
        list.add("Hello!");
        list.add(123);
        for (Object str : list) {
            System.out.println("-" + str);
        }
    }
}
```



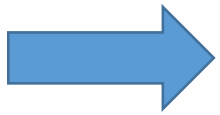

Теперь в приведении к String больше нет необходимости. Нужный тип указывается в угловых скобках (angle brackets), которые обрамляют дженерики. Теперь компилятор не даст скомпилировать класс, пока добавление 123 в список не будет удалено, т.к. это тип Integer.

# Обобщенные типы (Generics)

## Причины появления обобщенных типов

Многие называют дженерики “синтаксическим сахаром”, что небезосновательно, так как дженерики действительно при компиляции прячут в себе преобразование типов:

```
public void testAppHasAGreeting() {
    List list = new ArrayList();
    list.add("hello");
    String test = (String) list.get(0);
}
```



```
public void testAppHasAGreeting() {
    List<String> list = new ArrayList<>();
    list.add("hello");
    String test = list.get(0);
}
```

После компиляции какая-либо информация о дженериках стирается.

Это называется "Стирание типов" или "Type Erasure".

Стирание типов гарантирует, что новые классы для параметризованных типов не создаются.

Следовательно, дженерики не расходуют ресурсы на этапе выполнения программы.

# Обобщенные типы (Generics)

Существуют две категории дженериков:

- Сырые типы (Raw Types)
- Типизированные типы (Generic Types)

Сырые типы — это типы без указания "уточнения" в фигурных скобках.

```
 LinkedList list = new LinkedList();
list.add(new MyObject());
MyObject myObject = (MyObject)list.get(0);
```

Типизированные типы — наоборот, с указанием "уточнения":

```
 LinkedList<MyObject> list = new LinkedList<MyObject>();
list.add(new MyObject());
MyObject myObject = list.get(0);
```

# Обобщенные типы (Generics)


- При создании объекта обобщенного типа с помощью конструктора в последних версиях Java второй раз тип можно не указывать, например:

```
java.util.ArrayList<Integer> array = new java.util.ArrayList<>();
```

- Такой синтаксис называется алмазным (the diamond), так как форма фигурных скобок напоминает алмаз.
- Diamond синтаксис связан с понятием «выведения типов»: компилятор, видя справа <> смотрит на левую часть, где расположено объявление типа переменной, в которую присваивается значение, и, по этой части понимает, каким типом типизируется значение справа.

# Обобщенные типы (Generics)

- Если в левой части указан дженерик, а справа не указан, компилятор сможет вывести тип:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args) {
        List<String> list = new ArrayList();
        list.add("Hello World");
        String data = list.get(0);
        System.out.println(data);
    }
}
```

- Однако, так лучше не делать, так как запись List<String> list = new ArrayList(); - это смешение старого и нового стилей, и среда разработки выведет предупреждающее сообщение о некорректности такого объявления:

# Обобщенные типы (Generics)

```java
package generics;

import java.util.ArrayList;
import java.util.List;

public class HelloWorld {
    public static void main(String []args) {
        List<String> list = new ArrayList<>();
        list.add("Hello World");
        String data = list.get(0);
        System.out.println(data);
    }
}
```

```java
package generics;

import java.util.ArrayList;
import java.util.List;

public class HelloWorld {
    public static void main(String []args) {
        List<String> list = new ArrayList();
        list.a
        String data = list.get(0);
        System.out.println(data);
    }
}
```

Unchecked assignment: 'java.util.ArrayList' to 'java.util.List<java.lang.String>' more... (Ctrl+F1)

Однако, если в объявление добавить <>, сообщение тут же исчезнет.
Т.е. без diamond синтаксиса компилятор не понимает, что его обманывают, а вот с diamond — понимает.

# Обобщенные типы (Generics)

Еще один пример:

```
package generics;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class HelloWorld {
    public static void main(String []args) {
        List<String> list = Arrays.asList("Hello", "World");
        List<Integer> data = new ArrayList(list);
        Integer intNumber = data.get(0); //java.lang.ClassCastException
        System.out.println(data);
    }
}
```

В третьей строке будет выброшена ошибка, так как невозможно переменной типа Integer присвоить значение типа String. При этом ни компилятор, ни среда разработки никак не предупреждают разработчика о некорректности объявления List<Integer> data = new ArrayList(list); так как «алмазный» синтаксис не используется.

# Обобщенные типы (Generics)

Однако после добавления <> скобок сразу же появляется сообщение об ошибке:

```
package generics;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class HelloWorld {
    public static void main(String []args) {
        List<String> list = Arrays.asList("Hello", "World");
        List<Integer> data = new ArrayList<>(list);
        Integer intNumber = data.ge                    ang.ClassCastException
        System.out.println(data);
    }
}
```

Cannot infer arguments

Отсюда вытекает основное правило - всегда использовать diamond синтаксис с типизированными типами. В противном случае есть риск пропустить, где используется raw type.

# Обобщенные типы (Generics)

## Обобщенные методы (Generic Methods)

Дженерики позволяют типизировать методы.
Рассмотрим пример:

```
import java.util.*;
public class HelloWorld{

    public static class Util {
        public static <T> T getValue(Object obj, Class<T> clazz) {
            return (T) obj;
        }
        public static <T> T getValue(Object obj) {
            return (T) obj;
        }
    }

    public static void main(String []args) {
        List list = Arrays.asList("Author", "Book");
        for (Object element : list) {
            String data = Util.getValue(element, String.class);
            System.out.println(data);
            System.out.println(Util.<String>getValue(element));
        }
    }
}
```

# Обобщенные типы (Generics)

## Обобщенные методы (Generic Methods)

- В классе Util объявлено два типизированных метода.
- Благодаря возможности выведения типов можно предоставить определение типа непосредственно компилятору или можно сделать это самостоятельно.
- При типизировании метода дженерик указывается ДО имени метода, потому что если использовать дженерик после имени метода, Java не сможет понять, с каким типом работать.
- Поэтому сначала объявляем дженерик, а затем говорим, что этот дженерик будем возвращать.

# Обобщенные типы (Generics)

## Обобщенные методы (Generic Methods)

- В примере выше в классе HelloWorld создан вложенный статический класс Util, возвращающий значения, соответствующие определенным типам (тип String).
- Естественно, Util.<Integer>getValue(element, String.class) упадёт с ошибкой incompatible types: Class<String> cannot be converted to Class<Integer>

При использовании типизированных методов стоит всегда помнить про **стирание типов**.

Посмотрим на пример:

# Обобщенные типы (Generics)

## Обобщенные методы (Generic Methods)

```
import java.util.*;
public class HelloWorld {

    public static class Util {
        public static <T> T getValue(Object obj) {
            return (T) obj;
        }
    }

    public static void main(String []args) {
        List list = Arrays.asList(2, 3);
        for (Object element : list) {
            System.out.println(Util.<Integer>getValue(element) + 1);
        }
    }
}
```

# Обобщенные типы (Generics)

## Обобщенные методы (Generic Methods)

Все будет прекрасно работать до тех пор, пока компилятор будет понимать, что у вызываемого метода тип Integer.

Заменим вывод на консоль на следующую строку:

```
System.out.println(Util.getValue(element) + 1);
```

Среда разработки сразу же выдаст ошибку:

# Обобщенные типы (Generics)

## Обобщенные методы (Generic Methods)

```
package generics;

import java.util.Arrays;
import java.util.List;

public class HelloWorld {
    public static class Util {
        public static <T> T getValue(Object obj) {
            return (T) obj;
        }
    }

    public static void main(String []args) {
        List list = Arrays.asList(2, 3);
        for (Object element : list) {
            System.out.println(Util.getValue(element) + 1);
        }
    }
}
```

Operator '+' cannot be applied to 'java.lang.Object', 'int'

Компилятор видит, что тип никто не указал, следовательно, тип указывается как Object, и выполнение кода становится невозможным.
Произошло стирание типов.

# Обобщенные типы (Generics)

## Обобщенные классы (Generic Types)

Типизировать можно не только методы, но и сами классы.

Давайте создадим и используем обобщенный класс Bank.

```java
package generics;

import java.util.Arrays;

public class Bank<T> {
    T[] accounts;

    public Bank() {
    }

    public Bank(T[] accounts) {
        this.accounts = accounts;
    }

    @Override
    public String toString() {
        return "Bank{" +
                "accounts=" + Arrays.toString(accounts) +
                '}';
    }
}
```

```java
package generics;

public class Main {
    public static void main(String[] args) {
        Bank<Integer> bank = new Bank<>(new Integer[]{1, 2, 3, 4, 5, 6});
        Bank<String> bank2 = new Bank<>(new String[]{"123456", "321654", "654789"});
        System.out.println(bank.toString());
        System.out.println(bank2.toString());
    }
}
```

# Обобщенные типы (Generics)

## Обобщенные классы (Generic Types)

- Буква T в определении класса class Bank<T> указывает, что данный тип T будет использоваться этим классом.
- Параметр T называется универсальным параметром, и вместо него можно подставить любой тип.
- При этом заранее не известно, какой это будет тип, будет ли это класс или интерфейс.
- Буква T выбрана условно, и вместо нее может быть любая другая буква.
- После объявления класса можно применить универсальный параметр T: далее в классе объявляется переменная этого типа, которой затем присваивается значение в конструкторе.

# Обобщенные типы (Generics)

## Обобщенные классы (Generic Types)

- При использовании обобщенных классов необходимо учитывать, что ***они работают только с объектами, но не работают с примитивными типами***.
- Можно написать ArrayList<Employee>, но нельзя использовать тип int или double, например: ArrayList<int>.
- Вместо примитивных типов необходимо использовать классы-обертки: Integer вместо int, Double вместо double.
- Также нельзя использовать статические переменные универсальных параметров: нельзя написать *static T account;*.
- Кроме того, нельзя создавать экземпляры универсальных классов следующим образом: *account = new T();*.

# Обобщенные типы (Generics)

## Ограничения

- Представим, что у нас имеется интерфейс ITicket, который представляет билет на одну поездку.
- В автомате для покупке билетов установлена система, позволяющая покупать их разные виды: в одну сторону, в обе стороны и др.
- Разные типы билетов покупаются клиентами в различное время, и нет возможности узнать, какой именно билет будет куплен в тот или иной момент.
- Давайте установим ограничение в виде интерфейса ITicket на покупку билетов.
- Ограничение типа ITicket  говорит о том, что система будет покупать только те билеты, которые реализуют этот интерфейс:

# Обобщенные типы (Generics)

## Ограничения

```java
package generics.tickets;

public interface ITicket {
    String getType();
}
```

```java
package generics.tickets;

public class Ticket implements ITicket {
    private int id;
    private String type;
    private String destination;
    private String purchaseDate;

    public Ticket(int id, String type, String destination, String purchaseDate) {
        this.id = id;
        this.type = type;
        this.destination = destination;
        this.purchaseDate = purchaseDate;
    }

    @Override
    public String getType() {
        return type;
    }
}
```

```java
package generics.tickets;

public class PurchaseTicketSystem<T extends ITicket> {
    T[] tickets;

    public PurchaseTicketSystem(T[] tickets) {
        this.tickets = tickets;
    }

    public void purchasedTicketInfo(){
        for(ITicket ticket : tickets){
            System.out.println(ticket.getType());
        }
    }
}
```

```java
package generics.tickets;

public class Main {
    public static void main(String[] args) {
        Ticket[] tickets = new Ticket[]{
                new Ticket( id: 1,  type: "One-way-ticket",  destination: "USA",  purchaseDate: "05.10.2019"),
                new Ticket( id: 2,  type: "Return-ticket",  destination: "England",  purchaseDate: "05.11.2019")
        };
        PurchaseTicketSystem<Ticket> purchaseTicketSystem = new PurchaseTicketSystem<>(tickets);
        purchaseTicketSystem.purchasedTicketInfo();
    }
}
```

# Обобщенные типы (Generics)


## Ограничения

- В качестве ограничения можно задать не только интерфейс, но и класс.
- Также можно установить сразу несколько ограничений.
- Например, мы хотим создать гараж, в котором будут стоят различные транспортные средства.
- Все транспортные средства умеют двигаться.
- Создадим родительский класс Vehicle (транспортное средство), и его наследников Car и Train.
- Создадим интерфейс IMove с методом move() и реализуем его в наших классах-транспортных средствах:

# Обобщенные типы (Generics)

## Ограничения

```java
package generics.restrictions;

public interface IMove {
    void move();
}
```

```java
package generics.restrictions;

public class Vehicle {
    String name;

    public Vehicle(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```java
package generics.restrictions;

public class Car extends Vehicle implements IMove {
    public Car(String name) {
        super(name);
    }

    @Override
    public void move() {
        System.out.println("Car " + name + " is moving!");
    }
}
```

```java
package generics.restrictions;

public class Train extends Vehicle implements IMove {

    public Train(String name) {
        super(name);
    }

    @Override
    public void move() {
        System.out.println("Train " + name + " is moving");
    }
}
```

```java
package generics.restrictions;

public class Garage<T extends Vehicle & IMove> {
    private T[] memebers;

    public Garage(T[] memebers) {
        this.memebers = memebers;
    }

    public void info(){
        for (T member : memebers){
            member.move();
        }
    }
}
```

# Обобщенные типы (Generics)

## Ограничения

```
package generics.restrictions;

public class Main {
    public static void main(String[] args) {
        Car[] cars = new Car[]{new Car( name: "Audi"), new Car( name: "BMW")};
        Train[] trains = new Train[]{new Train( name: "Минск-Варшава"), new Train( name: "Минск-Гродно")};

        Garage<Car> carGarage = new Garage<>(cars);
        carGarage.info();

        Garage<Train> trainGarage = new Garage<>(trains);
        trainGarage.info();
    }
}
```

```
Car Audi is moving!
Car BMW is moving!
Train Минск-Варшава is moving
Train Минск-Гродно is moving

Process finished with exit code 0
```

# Обобщенные типы (Generics)

## Использование нескольких универсальных параметров

Можно задать сразу несколько универсальных параметров и ограничения к каждому из них:

```
class Operation<A extends IAccount, S>{
    A account;
    S sum;

    public Operation(A acc, S money){
        this.account = acc;
        this.sum = money;
    }

    void getInfo(){
        System.out.printf("Клиент %s вывел %s рублей \n", account.getId(), String.valueOf(sum));
    }
}
```

И затем ис

```
public static void main(String[] args) {
    Account account = new Account(21);
    Operation<Account, Integer> op = new Operation(account, 100);
    op.getInfo();
}
```

# Обобщенные типы (Generics)

## Подстановки (wildcards)

В обобщённом коде знак вопроса (?), называемый подстановочным символом, означает любой тип.

Рассмотрим пример: программу, рисующую фигуры.

1

```
package generics.wildcards;

public abstract class Shape {
    public abstract void draw(Canvas c);
}
```

2

```
package generics.wildcards;

import java.util.List;

public class Canvas {
    public void drawAll(List<Shape> shapes) {
        for (Shape s: shapes) {
            s.draw( c: this);
        }
    }
}
```

3

```
package generics.wildcards;

public class Circle extends Shape {
    private int x, y, radius;
    @Override
    public void draw(Canvas c) {

    }
}
```

4

```
package generics.wildcards;

public class Rectangle extends Shape {
    private int x, y, width, height;
    @Override
    public void draw(Canvas c) {

    }
}
```

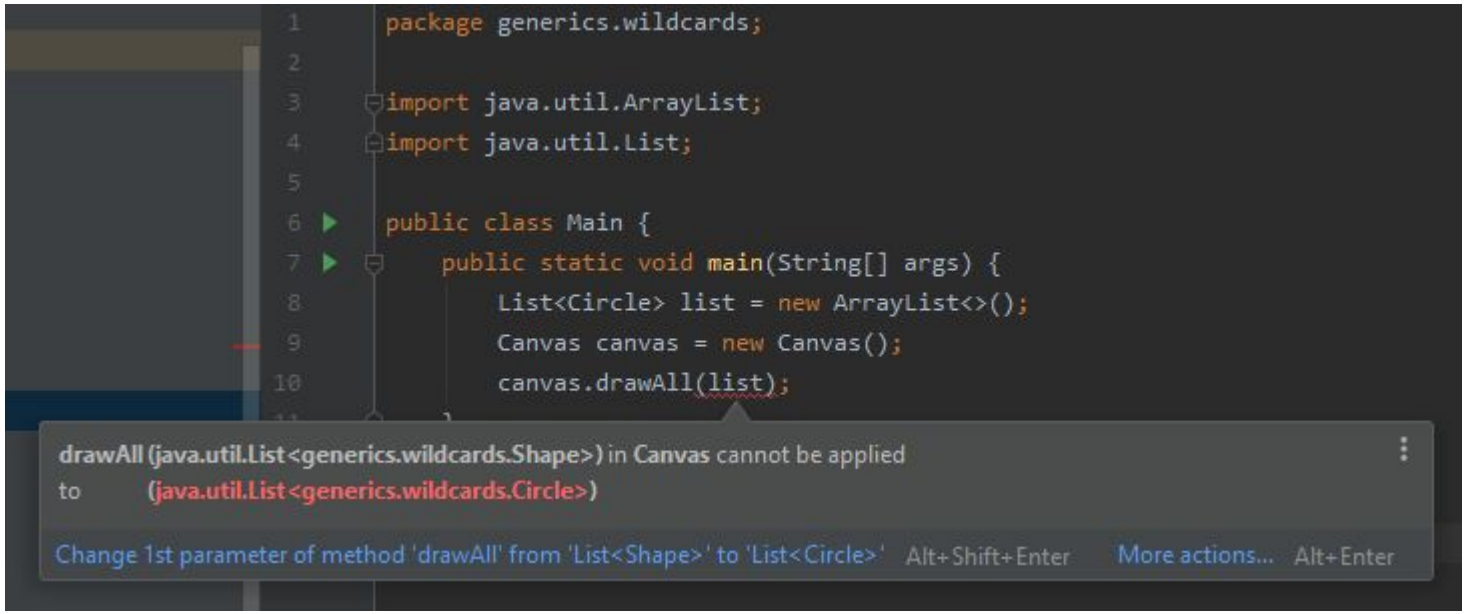
# Обобщенные типы (Generics)

## Подстановки (wildcards)

- Для организации иерархии фигур создадим абстрактный класс Shape.
- Создадим конкретные классы-фигуры – круг и прямоугольник.
- Фигуры будут рисоваться на полотне – класс Canvas.
- В классе Canvas опишем метод для отрисовки списка фигур - drawAll(List<Shape> shapes).
- Метод принимает в качестве параметра обобщенный список, в который может хранить только объекты типа Shape.
- И если попытаться передать в данный метод список объектов Circle, компилятор тут же выдаст ошибку:

# Обобщенные типы (Generics)

## Подстановки (wildcards)




- Но если немного изменить сигнатуру метода на drawAll(List<? extends Shape> shapes), ошибка тут же исчезнет:

# Обобщенные типы (Generics)

## Подстановки (wildcards)

```java
package generics.wildcards;

import java.util.List;

public class Canvas {
    public void drawAll(List<? extends Shape> shapes) {
        for (Shape s: shapes) {
            s.draw( c: this);
        }
    }
}
```

```java
package generics.wildcards;

import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Circle> list = new ArrayList<>();
        Canvas canvas = new Canvas();
        canvas.drawAll(list);
    }
}
```

- Теперь в метод можно передать список объектов, наследующихся от класса Shape.
- Подстановка ? выставляет верхнюю границу для используемых типов – тип может быть как Shape непосредственно, так подтипом, расширяющим Shape.

# Обобщенные типы (Generics)

## Обобщенные конструкторы

```
1  public class GenericsApp {
2      public static void main(String[] args) {
3          Operation op = new Operation(12.6, 30.4);
4          System.out.println(op.getSum());
5      }
6  }
```

```
1  class Operation{
2      double x1;
3      double x2;
4
5      <T extends Number> Operation(T d1, T d2){
6          this.x1 = d1.doubleValue();
7          this.x2 = d2.doubleValue();
8      }
9
10     double getSum(){
11         return x1 + x2;
12     }
13 }
```

Здесь конструктор класса Operation типизируется типом T, который должен быть унаследован от класса Number.
При использовании конструктора в качестве параметров передаются два объекта, которые представляют числа – то есть тип Number.

# Обобщенные типы (Generics)

## Обобщенные интерфейсы

Интерфейсы, как и классы, тоже могут быть обобщенными.

Например:

```
1  interface Accountable<T>{
2      T getAccount();
3  }
```

```
1  class Operation implements Accountable<IAccount>{
2      IAccount account;
3
4      public Operation(Account acc){
5          this.account = acc;
6      }
7
8      public IAccount getAccount(){
9          return account;
10     }
11 }
```

```
1  public class GenericsApp {
2
3      public static void main(String[] args) {
4          Operation op = new Operation(new Account(21));
5          IAccount account = op.getAccount();
6          System.out.println(account.getId());
7      }
8  }
```

# Обобщенные типы (Generics)

## Наследование и обобщение

Обобщенные классы могут участвовать в иерархии наследования: могут наследоваться от других, либо выполнять роль базовых классов.

При наследовании от обобщенного класса, класс-наследник должен передавать данные о типе в конструкторе базового класса.

```
1  class Account<T> {
2      private T _id;
3
4      T getId(){
5          return _id;
6      }
7
8      Account(T id){
9          _id = id;
10     }
11 }
```

```
1  class DepositAccount<T> extends Account<T>{
2
3      DepositAccount(T id){
4          super(id);
5      }
6  }
```

В конструкторе класса DepositAccount идет обращение к конструктору базового класса, в который передаются данные о типе.

# Обобщенные типы (Generics)

## Наследование и обобщение

Варианты использования классов

```
DepositAccount dAccount1 = new DepositAccount(20);
System.out.println(dAccount1.getId());

DepositAccount dAccount2 = new DepositAccount("12345");
System.out.println(dAccount2.getId());
```

При этом класс-наследник может добавлять и использовать какие-то свои параметры типов:

```
class Account<T>{
    private T _id;

    T getId(){
        return _id;
    }

    Account(T id){
        _id = id;
    }
}
```

```
class DepositAccount<T, S> extends Account<T>{
    private S _name;
    S getName(){
        return _name;
    }

    DepositAccount(T id, S name){
        super(id);
        this._name=name;
    }
}
```

```
DepositAccount<Integer, String> dAccount1 = new DepositAccount(20, "Tom");
System.out.println(dAccount1.getId() + " : " + dAccount1.getName());

DepositAccount<String, Integer> dAccount2 = new DepositAccount("12345", 23456);
System.out.println(dAccount2.getId() + " : " + dAccount2.getName());
```

# Обобщенные типы (Generics)

## Наследование и обобщение

Класс-наследник вообще может не быть обобщенным:

```
1  class Account<T> {
2      private T _id;
3
4      T getId(){
5          return _id;
6      }
7
8      Account(T id){
9          _id = id;
10     }
11 }
```

```
1  class DepositAccount extends Account<Integer>{
2
3      DepositAccount(){
4          super(5);
5      }
6  }
```

Здесь при наследовании явным образом указывается тип, который будет использоваться конструкторами базового класса, то есть тип Integer. Соответственно, в конструктор базового класса передается значение именно этого типа – число 5.

```
1  DepositAccount dAccount1 = new DepositAccount();
2  System.out.println(dAccount1.getId());
```

# Обобщенные типы (Generics)

## Наследование и обобщение

Также может быть ситуация, когда базовый класс является обычным необобщенным классом:

```
class Account{
    private String _name;

    String getName(){
        return _name;
    }

    Account(String name){
        _name=name;
    }
}
```

```
class DepositAccount<T> extends Account{
    private T _id;

    T getId(){
        return _id;
    }

    DepositAccount(String name, T id){
        super(name);
    }
}
```

В этом случае данные в конструктор базового класса передаются как обычно.

# Обобщенные типы (Generics)

## Преобразование обобщенных типов

Объект одного обобщенного типа можно привести к другому типу, если они используют один и тот же тип:

```
1  class Account<T>{
2      private T _id;
3
4      T getId(){
5          return _id;
6      }
7
8      Account(T id){
9          _id = id;
10     }
11 }
```

```
1  class DepositAccount<T> extends Account<T>{
2
3      DepositAccount(T id){
4          super(id);
5      }
6  }
```

```
1  DepositAccount<Integer> depAccount = new DepositAccount(10);
2  Account<Integer> account = (Account<Integer>)depAccount;
3  System.out.println(account.getId());
```

Объект DepositAccount<Integer> можно привести к Account<Integer>

или

DepositAccount<String> к Account<String>:

```
1  DepositAccount<Integer> depAccount = new DepositAccount(10);
2  Account<String> account = (Account<String>)depAccount;
```

# Ссылочные типы и клонирование объектов


## Ссылочные типы

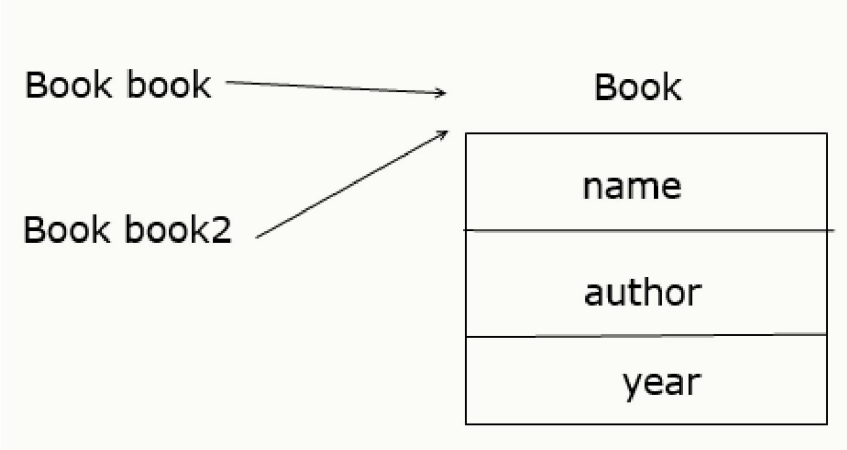
- При работе с объектами классов надо учитывать, что они все представляют ссылочные типы, то есть указывают на какой-то объект, расположенный в памяти.
- Чтобы понять возможные трудности, с которыми можно столкнуться при использовании ссылочных типов, рассмотрим пример:

```
1  Book book = new Book("Война и мир", "Л. Толстой", 1863);
2  Book book2 = book;
3  book2.setName("Отцы и дети");
4  System.out.println(book.getName());
```

# Ссылочные типы и клонирование объектов

## Ссылочные типы

- Здесь создаются два объекта Book, и один объект присваивается другому.
- Но, несмотря на то, что изменяется только объект book2, вместе с ним изменяется и объект book.
- Это происходит потому, что после присвоения ссылки в переменных book и book2 указывают на одну и ту же область памяти, где, собственно, данные об объекте Book и его полях и хранятся.



# Ссылочные типы и клонирование объектов

## Клонирование объектов

- Чтобы избежать этой проблемы, необходимо создать отдельный объект для переменной book2, например, с помощью метода clone().

```
1  class Book implements Cloneable{
2
3      //остльной код класса
4
5      public Book clone() throws CloneNotSupportedException{
6
7          return (Book) super.clone();
8      }
9  }
```

- Для реализации клонирования класс Book должен применить интерфейс Cloneable, который определяет метод clone().
- Реализация этого метода просто возвращает вызов метода clone для родительского класса – класса Object – с преобразованием к типу Book.
- Кроме того, на случай, если класс не поддерживает клонирование, метод должен выбрасывать исключение CloneNotSupportedException, что определяется с помощью оператора throws.

# Ссылочные типы и клонирование объектов

## Клонирование объектов

- Затем с помощью вызова этого метода можно осуществить клонирование:

```
try{
    Book book = new Book("Война и мир", "Л. Толстой", 1863);
    Book book2 = book.clone();
} catch(CloneNotSupportedException ex){
    System.out.println("Не поддерживается клонирование");
}
```

- Однако, данный способ осуществляет неполное клонирование и подойдет только если клонируемый объект не содержит в себе других сложных объектов.
- Например, пусть класс Book имеет следующее определение:

# Ссылочные типы и клонирование объектов

## Клонирование объектов

```java
class Book implements Cloneable{
    private String name;
    private Author author;

    public void setName(String n){
        name=n;
    }

    public String getName(){
        return name;
    }

    public void setAuthor(String n){
        author.setName(n);
    }

    public String getAuthor(){
        return author.getName();
    }

    Book(String name, String author){
        this.name = name;
        this.author = new Author(author);
    }

    public String toString(){
        return "Книга '" + name + "' (автор " +  author + ")";
    }

    public Book clone() throws CloneNotSupportedException{
        return (Book) super.clone();
    }
}
```

```java
class Author{
    private String name;

    public void setName(String n){
        name=n;
    }

    public String getName(){
        return name;
    }

    public Author(String name){
        this.name=name;
    }
}
```

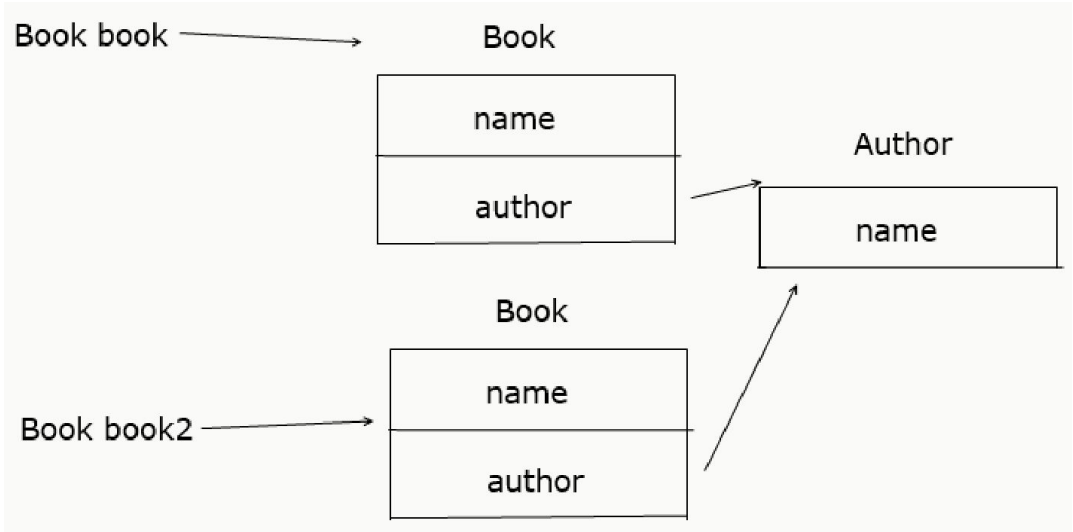
# Ссылочные типы и клонирование объектов

## Клонирование объектов

Если мы попробуем изменить автора книги, нас последует неудача:

```
try{
    Book book = new Book("Война и мир", "Л. Толстой");
    Book book2 = book.clone();
    book2.setAuthor("И. Тургенев");
    System.out.println(book.getAuthor());
} catch(CloneNotSupportedException ex){
    System.out.println("Не поддерживается клонирование");
}
```

Потому что, хотя переменные book и book2 будут указывать на разные объекты в памяти, но эти объекты будут указывать на один и тот же объект Author.



# Ссылочные типы и клонирование объектов

## Клонирование объектов

- В этом случае нам необходимо выполнить полное клонирование.
- Для этого нужно определить метод клонирования у класса Author:

```
class Author implements Cloneable{

    // остальной код класса

    public Author clone() throws CloneNotSupportedException{
        return (Author) super.clone();
    }
}
```

- И исправить метод clone() в классе Book следующим образом:

```
public Book clone() throws CloneNotSupportedException{
    Book newBook = (Book) super.clone();
    newBook.author=(Author) author.clone();

    return newBook;
}
```