Конспект. Введение в ООП. Инкапсуляция

В конспекте мы собрали всю теорию из урока. Скачайте файл или распечатайте, чтобы повторять материал офлайн в любом месте, где вам удобно.

Объектно-ориентированное программирование

Java — **объектно-ориентированный язык программирования.** В Java всё строится вокруг объектов. Объект — это экземпляр класса. Именно у объектов есть свойства (данные) и методы (функции), которые могут быть использованы для взаимодействия с объектами.

Без классов и объектов невозможно написать программу на Java.

- У вас всегда будет основной класс приложения, в котором находится метод `public static void main`.
- Для печати результата расчета простого выражения нужен класс `System` и его поле `out`, чтобы написать `System.out.println("Результат = " + result)`.

В рамках ООП программы представлены как множество объектов, которые взаимодействуют между собой. Объекты описываются через их свойства, взаимодействуют друг с другом с помощью методов, а шаблоны, по которым строятся эти объекты, представлены в классах.

Объектно-ориентированное программирование (сокращенно ООП) — стиль программирования, основанный на описании элементов программы в виде объектов.

Классы в Java

Объекты и классы можно сравнить с реальными предметами и их чертежами.

Каждая машина *(объект)* в каком-то смысле уникальна. Двух идентичных машин не найдешь. Однако собраны они могут быть по одному **чертежу**, то есть представлять собой один **класс** объектов. Эти две машины могут объединяться с автомобилями, собранными по другим чертежам, на том основании, что они одной марки. А машины разных марок могут относиться к одной группе легковых автомобилей.

Класс — это шаблон объектов, определяющий переменные и методы, которые могут быть использованы объектами этого класса.

Чтобы создать Java-класс, нужно нажать правой кнопкой мыши на папку (пакет), в которой планируется создание класса, и выбрать **New → Java Class**.

Объекты в Java

Один объект (*машина*) может реализовывать не только один конкретный класс (*Audi Q3*), а несколько (*все машины марки Audi, спортивные автомобили, легковые автомобили*). От каждого из этих классов объект «забирает» какие-то признаки.

У нескольких машин может быть общее свойство и конкретные реализации этого свойства. У каждой машины есть цвет, но не каждая машина желтая — есть синие машины, зеленые, красные и т. д. Некоторые свойства могут быть уникальны: не у каждого авто есть прокачанная звуковая система.

> **Объект** — это экземпляр класса, которому можно посылать команды и который может на эти команды реагировать, используя свои данные.

Чтобы создать новый объект, используйте оператор `new` при инициализации переменной определенного класса.

```
// Создаем объект car1 класса Car
Car car1 = new Car();
```

Методы в Java

> **Метод** — блок кода, который выполняет определенную функцию и позволяет использовать себя *снова и снова* в нескольких местах — без необходимости писать один и тот же код.

У нескольких машин могут быть как общие методы («*ехать*»), так и методы, которые не присущи всем объектам класса (например, не все машины в классе легковых автомобилей имеют метод «*сложить крышу*»).

Если бы мы писали программу, которая просто хранит информацию об автомобилях, у объектов-автомобилей мог быть один общий метод — «*сообщить информацию об автомобиле*».

Давайте напишем такой метод:

```
public class Car {
   public String model; // Поле для хранения модели автомобиля
   public String colour; // Поле для хранения цвета автомобиля

   // Метод для печати информации об автомобиле
   public void printInfo() {
      System.out.println("Модель авто " + model + " цвет " + colour);
      // Вывод информации о машине в консоль
   }
}
```

Создадим два объекта-автомобиля и присвоим им разные значения свойств `model` и `colour`:

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

      // Создаем объект car1 класса Car
      Car car1 = new Car();

      // Задаем значения полям объекта car1
      car1.model = "Audi";
      car1.colour = "Grey";

      // Вызываем метод printInfo() у объекта car1
      // для печати информации о машине
      car1.printInfo();

      // Создаем объект car2 класса Car
      Car car2 = new Car();

      // Задаем значения полям объекта car2
      car2.model = "Lada";
      car2.colour = "Blue";

      // Вызываем метод printInfo() у объекта car2
      // для печати информации о машине
      car2.printInfo();
   }
}
```

В консоли будет:

```
Модель авто Audi цвет Grey
Модель авто Lada цвет Blue
```

Метод — реализация поведения объекта. Это классическое определение — скорее рекомендация, чем требование. В примере выше поведение класса `Car` — возможность вывести информацию о себе в консоль, а метод `printInfo` реализует это.

Конечно, реальный автомобиль не может назвать свой цвет. Но в ООП мы и не моделируем реальность, а реализуем такое поведение объектов, которое решит конкретную задачу.

Данный метод будет работать идентично для каждого экземпляра класса и выполнять одни и те же действия, разница будет только в значениях полей объектов.

Каждый метод отвечает за решение одной определенной задачи. Метод может возвращать какое-то значение определенного типа. Или может быть помечен модификатором `void`, который означает, что метод только совершает какое-то действие, ничего при этом не возвращая.

Конструктор

Объекты в Java выполняют ключевую роль. Но для того чтобы начать использовать объекты, их нужно создать. В Java объекты создаются с помощью ключевого слова `new`. Сам процесс создания объекта происходит автоматически, но мы можем влиять на инициализацию объекта после создания, и в этом нам помогают конструкторы.

Что такое конструктор

Конструктор — это специальный метод класса. Он может задавать атрибуты объектов класса при их создании и конкретные значения этих атрибутов.

Задание атрибутов объекта или их конкретного значения также называется его **инициализацией.** То есть **конструктор класса инициализирует объект**, принадлежащий тому же классу.

Конструктор всегда вызывается при использовании ключевого слова `new`. Например, так выглядит вызов конструктора при создании объекта:

```
Car myCar = new Car();
// Создаем новый объект класса Car;
// Присваиваем его ссылке myCar.
// То, что находится после new, — по сути и есть конструктор.
```

Выше мы создали новый объект класса `Car`. При этом мы не инициализировали его, то есть не задали определенные атрибуты в скобках. То, что мы их не задали, не означает, что при создании объекта не использовался конструктор. Создать объект в Java без конструктора **невозможно.**

В данном случае Java сама создала **конструктор по умолчанию**. Этот тип конструктора всегда вызывается автоматически, если мы создаем объект и не прописываем ему атрибуты.

Конструктор, который **сразу** присваивает определенные значения полям, называется **конструктором с параметрами.**

Это единственные два типа конструкторов, которые существуют в Java.

Конструктор по умолчанию

Конструктор по умолчанию — это специальный метод класса, который создается компилятором автоматически, если в классе явным образом не определен конструктор.

Такой конструктор не отслеживает, какие поля у объекта должны быть заполнены. Посмотрим на пример работы с конструктором по умолчанию:

```
public class Car {
   private String model;
   private String colour;

   // Создаем конструктор по умолчанию
   public Car() {
      // Попробуем вернуть значения атрибутов Car
      System.out.println("Model: " + model);
      System.out.println("Colour: " + colour);
   }

   public static void main(String[] args) {
      // Создаем новый объект Car
      Car myCar = new Car();
   }
}

// Результат в консоли: [Model: null / Colour: null]
```

В примере выше мы создали класс `Car` с двумя приватными атрибутами — `model` и `colour`. Конструктор по умолчанию создает объект класса `Car` без параметров.

Если попробуем вернуть значения атрибутов `model` и `colour` — получим `null`. По умолчанию атрибуты объектов класса инициализируются значением `null` для ссылочных типов и `0` для числовых типов.

Теперь создадим конструктор по умолчанию, который задаст конкретные значения атрибутам при создании объекта:

```
public class Car {
   public String model;
   public String colour;

   // Создаем конструктор по умолчанию
   public Car() {
      this.model = "Стандартная";
      this.colour = "Любой";
   }

   public static void main(String[] args) {
      // Создаем новый объект Car
      Car myCar = new Car();

      //Выводим значения полей которые мы задали в конструкторе
      System.out.println("Model: " + myCar.model);
      System.out.println("Colour: " + myCar.colour);
    }
}

// Результат в консоли: [Model: Стандартная / Colour: Любой]
```

Такой прием будет полезен, если создаваемые вами атрибуты:

- не имеют конкретного значения,
- имеют общее значение по умолчанию для всех объектов класса.

Конструктор с параметрами

Часто при создании объектов нужно **сразу** присваивать им конкретные значения полей. Для этого используется **конструктор с параметрами**. Такие конструкторы называются **параметризованными.**

Конструктор с параметрами — это конструктор, который принимает один или несколько параметров. Он позволяет задавать начальные значения атрибутов при создании объекта.

Чтобы задать объектам класса конкретные атрибуты, просто поместите их в скобки `()`. Помните, что порядок следования аргументов важен.

```
public class Car {
   public String model;
   public String colour;

   // Создаем параметризованный конструктор
   public Car(String model, String colour) {
      this.model = model;
      this.colour = colour;
   }

   public static void main(String[] args) {

      // Создаем новый объект Car;
      // Обязательно задаем ему значения атрибутов в том же порядке,
      // в котором они заданы в конструкторе
      Car myCar = new Car("Toyota", "Red");

      // Печатаем значения атрибутов Car
      System.out.println("Model: " + this.model);
      System.out.println("Colour: " + this.colour);
   }
}

// Результат в консоли: [Model: Toyota / Colour: Red]
```

Этот конструктор используется для установки необходимых значений прямо во время создания объекта. Так как объект обычно не может существовать без определенных свойств и ожидать, когда ему их присвоят, — это самый правильный путь.

Обратите внимание

1. Если у класса есть хотя бы один заданный конструктор, компилятор не будет создавать конструктор по умолчанию.
2. Если вам нужны два конструктора, один из которых не должен содержать аргументов (т. е. конструктор по умолчанию), — создавайте его явно.

Код конструктора отвечает только за создание объекта. Не стоит добавлять в конструктор вызовы методов, которые не относятся к инициализации объекта. Конструктор используется, только чтобы задать начальное состояние объекта.

Ключевое слово this

В примерах этого урока использовалось ключевое слово `this` для создания параметризованного конструктора, а также конструктора по умолчанию со стандартными значениями.

> Переменная `this` в Java используется внутри конструктора для обращения к текущему объекту, который создается при вызове конструктора.

То есть эта переменная нужна, чтобы указать Java: следующий атрибут нужно привязать к объектам, которые создаются именно этим конструктором.

Также переменная `this` может использоваться в обычных методах объекта, чтобы сослаться на самого себя в определенных методах.

Принципы ООП

Выделяют три **основных** принципа объектно-ориентированного программирования (ООП):

1. Данные объекта должны быть скрыты от остальной программы.
2. Классы могут наследовать свойства и методы других классов.
3. Объекты одного и того же класса могут иметь разное поведение.

Сокрытие данных называется **инкапсуляцией.**

> **Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя.

Инкапсуляция помогает упростить код, уменьшить ошибки и облегчить сопровождение программы.

То, что классы могут перенимать свойства и методы других классов, называется **наследованием.**

> **Наследование** — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется **базовым, родительским** или **суперклассом**. Новый класс — **потомком, наследником** или **производным классом.**

Реализация этого принципа позволяет повторно использовать код и создавать иерархии классов.

Основные преимущества ООП проявляются, только если в нем реализован **полиморфизм.**

Полиморфизм — это возможность объектов с одинаковой спецификацией иметь различную реализацию и по-разному себя вести.

Благодаря этому принципу Java может использовать объекты с одинаковыми интерфейсами без информации о типе и внутренней структуре объекта.

Принципы **инкапсуляции, наследования и полиморфизма** широко используются в языке Java. С принципом инкапсуляции как с одним из самых простых вы познакомитесь в следующем шаге урока, с остальными принципами — в следующих уроках.

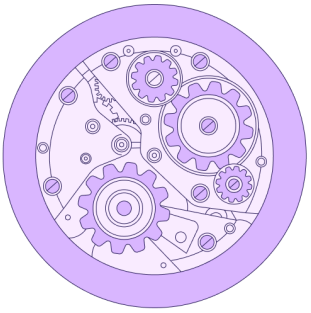
Инкапсуляция

Одним из важнейших принципов программирования является **безопасность**. В ООП безопасность обеспечивается механизмом **инкапсуляции функций и данных.**

Инкапсуляция скрывает внутреннюю реализацию сложной концепции. Что вы должны видеть — то и видите. Не больше и не меньше.

Внутренняя реализация скрыта, шанс что-то сломать — минимальный

Внутренняя реализация открыта, любое вмешательство опасно



Мы скрываем данные по той же логике, что и в жизни. Какие-то данные мы разглашаем легко, например свое имя. Другие данные мы или вовсе не разглашаем, или предоставляем лишь немногим по необходимости, например номер банковской карты.

Более того, такие данные мы стараемся дополнительно обезопасить с помощью разных методов: паролей, сейфов, замков.

Для обеспечения безопасности в Java используются методы инкапсуляции. Они помогают скрыть информацию:

- которая слишком сложна для пользователя нашего класса;
- которую может **неправильно использовать** другой разработчик, поменяв таким образом логику работы методов из-за неправильного изменения внутреннего состояния объекта.

Как применяется инкапсуляция

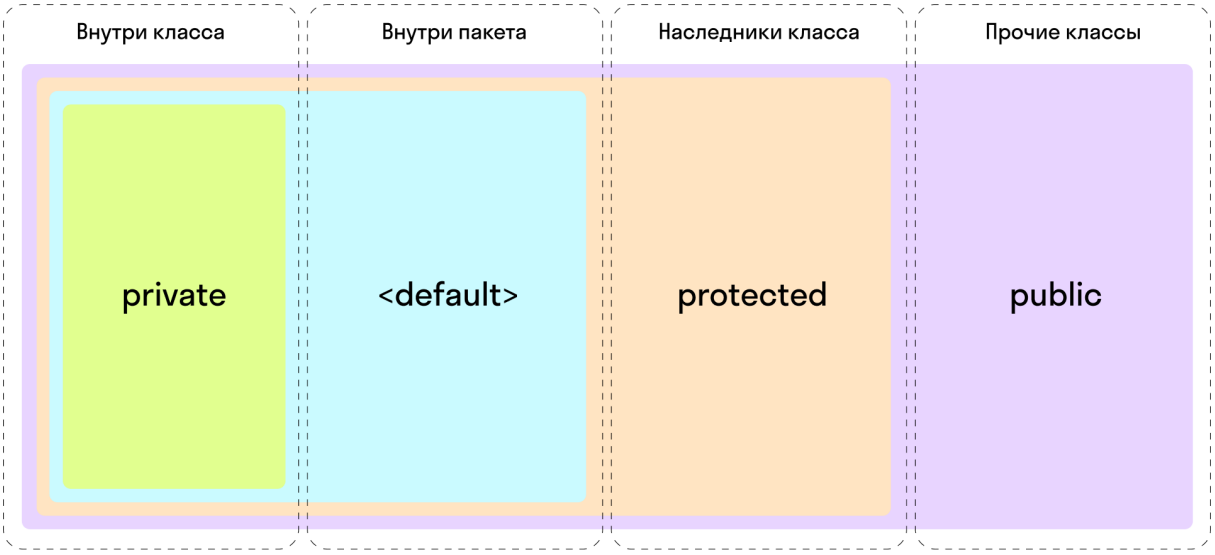
Инкапсуляция не просто скрывает данные ото всех — она **регулирует доступ** к этим данным.

- Чтобы указывать, кому должны быть доступны данные, используются **модификаторы доступа**.
- Чтобы запрашивать и изменять значения защищенных полей, используются **геттеры и сеттеры.**

Модификаторы доступа

Модификаторы доступа (или видимости) — это специальные ключевые слова, которые показывают, кому нельзя, а кому можно пользоваться данными.

Модификаторы видимости



В Java есть четыре модификатора доступа:

- `public` — публичный доступ. Означает, что данные доступны в любой точке программы.
- `default` — доступ по умолчанию. Выставляется сам, если мы не указываем модификатор доступа вручную. Данные с этим модификатором видны в пределах папки (пакета).
- `protected` — защищенный доступ. Устанавливает видимость, аналогичную модификатору `default`, только доступ распространяется на классы-наследники.
- `private` — приватный доступ. Данные, помеченные этим модификатором доступа, видны только внутри класса.

Модификатор доступа нужно указывать в самом начале объявления переменной, метода или класса.

Разберем на примере, как модификаторы доступа влияют на безопасность наших данных. Для начала создадим класс с одной публичной переменной:

```
public class SomeClass {
   public String str = "Hello World!!!";
   // public не ограничивает доступ к объекту никаким классом
}
```

Далее выведем на экран значение этой переменной:

```
public class Test {
   public static void main(String[] args) {
      SomeClass object = new SomeClass();
      System.out.println(object.str);
   }
}
```

На выходе мы получим фразу `«Hello World!!!»`.

Так как переменная публичная, ничто не мешает нам изменить переменную:

```
public class Test {
   public static void main(String[] args) {
      SomeClass object = new SomeClass();
      System.out.println(object.str);

      // Вызвали строку
      object.str = " It's modified!!!";

      // Беспрепятственно добавили к ней что-то
      System.out.println(object.str);
   }
}
```

В результате в консоль будет выведена измененная фраза:

```
Hello World!!!
It's modified!!!
```

Строку может изменить любой, у кого есть доступ к коду нашей программы, причем изменить в любой части программы.

Чтобы этого избежать, поменяем модификатор с `public` на `private`:

```
class SomeClass {
   private String str = "Hello World!!!";
   // private ограничивает доступ к str извне класса SomeClass
}
```

Теперь попытка получить доступ к нашей переменной обернется ошибкой:

```
java: s has private access in MyClass
```

Это значит, что наша переменная защищена от изменений.

Геттеры и сеттеры

Вы уже знаете, что модификатор `private` не дает доступа к данным за пределами класса. Если мы сами захотим обратиться к переменной за пределами класса, то также получим ошибку.

Чтобы этого избежать и иметь доступ к этой переменной, нужно:

- либо изменить модификатор (чего мы не хотим);
- либо использовать инструментарий **геттеров и сеттеров.**

Это обычные методы, использование которых стало настолько повсеместным, что разработчики языка Java вынесли их в отдельную категорию методов. Их можно генерировать автоматически с помощью среды разработки.

Геттер (от англ. get — «получать») — это метод, с помощью которого мы получаем значение свойства объекта (переменной, поля).

Пример использования геттера:

```
class SomeClass {
    private String str = "Hello World!!!";
    public String getStr() {
        return str;
    }
}
```

Сеттер (от англ. set — «устанавливать») — это метод, с помощью которого мы меняем или задаем значение свойства объекта.

Добавим в наш класс сеттер:

```
class SomeClass {
   private String str = "Hello World!!!";
   public String getStr() {
      return str;
   }

   public void setStr(String newStr) {
      this.str = newStr;
   }
}
```

Теперь реализуем наш класс `Test` в соответствии с изменениями:

```
public class Test {
   public static void main(String[] args) {
      SomeClass object = new SomeClass();
      System.out.println(object.getStr());
      object.setStr("It's modified!!!");
      System.out.println(object.getStr());
   }
}
```

Мы получим в качестве результата уже знакомую фразу:

```
Hello World!!!
It's modified!!!
```

Мы не обращались к переменной напрямую — мы использовали геттер, чтобы получить значение, и сеттер, чтобы его изменить.

С помощью **геттера** и **сеттера** мы добились точно такого же результата, как и при изменении переменной **напрямую**. Зачем же тогда эти дополнительные методы?

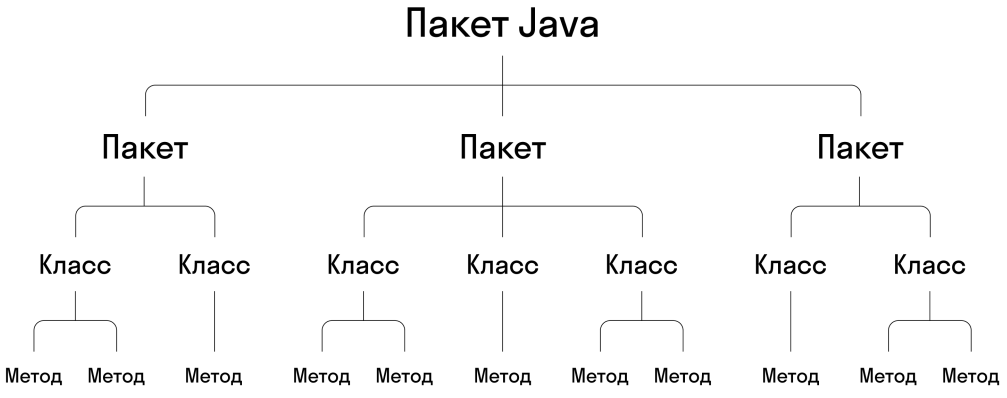
Сейчас мы рассмотрели самый простой случай применения геттеров и сеттеров. Но если мы решим установить какой-то **фильтр для изменения переменной** или добавить **условие для получения ее значения,** то без данных методов это будет сделать невозможно.

Пакеты

При обсуждении модификаторов доступа мы узнали, что у `protected` и `default` видимость определяется пакетом (папкой). Давайте разберемся, что это такое.

Что такое пакет

Пакеты очень похожи на папки в файловой системе. По сути, это и есть папки для хранения классов или файлов в программе.



Даже в маленьких проектах есть много классов и различных файлов. Хранить их в одном месте без какой-либо структуры — неудобно. И нефункционально: разные программисты могут создавать классы с одинаковыми названиями в пределах одной программы.

Для этого и существуют пакеты.

Добавление класса в пакет

Сначала посмотрим, как добавить класс в пакет. Разберем это на примере добавления класса `MyClass` в пакет `lessons`.

Чтобы добавить класс в пакет, необходимо использовать оператор `package`, который всегда прописан в первой строке файла:

```
package lessons;
// Создаем пакет lessons - все классы ниже будут входить в него

public class MyClass {
   public static void main(String[] args) {
      System.out.print("Hello world");
   }
}
```

В Java есть возможность создавать многоуровневые пакеты. Для этого всю иерархию пакетов необходимо прописать через точки:

```
package com.skypro.javacourse.lessons;

public class MyClass {
   public static void main(String[] args) {
      System.out.print("Hello world");
   }
}
```

При создании пакетов нужно придерживаться некоторых правил.

1. В коммерческой разработке именование пакетов начинается с `com`**.** После этого идет название компании и имя проекта.
2. Далее названия пакетов формируются по функциональному признаку.
3. Полное название класса включает в себя название пакета.
   В первом примере полное название класса — `lessons.MyClass`, а во втором примере — `com.skypro.javacourse.lessons.MyClass`.

Пакеты необходимы не только **для логического разделения классов и файлов проекта**, но и **для управления доступами**. Мы можем определить, какие классы внутри пакета будут недоступны за его пределами.

Импорт классов и пакетов

Теперь мы знаем, как добавлять классы в пакеты, как правильно называть пакеты и, главное, зачем это делать: объединение классов в пакеты упрощает жизнь.

Но и есть недостаток: **полное имя класса становится громоздким,** так как включает в себя название пакета.

Для решения этой проблемы в Java есть **оператор** `import`. С его помощью можно *импортировать* класс (или даже целый пакет) в файл программы.

> Мы упоминаем длинный, сложно названный класс один раз, чтобы в дальнейшем упоминать его по имени.

Использование класса из другого пакета

Посмотрим на использование класса из другого пакета на примере.

У нас есть классы из разных пакетов — `one.Ex1` и `two.Ex2`:

```
package one;

public class Ex1 {
   public static void main(String[] args) {
            // Прописываем класс Ex2 из пакета two.Ex2
      // в качестве объекта класса Ex1
      two.Ex2 ex2 = new two.Ex2();
      // Вызываем метод объекта ex2, который описывается классом Ex2
      ex2.doWork();
      System.out.print("Done!!!");
   }
}
```

```
package two;

public class Ex2 {
  public void doWork(){
    System.out.println("Method from Ex2");
  }
}
```

В классе `one.Ex1` создается объект `two.Ex2` с указанием его полного имени. Это неудобно, особенно при использовании многоуровневых пакетов.

Использование оператора import

Теперь реализуем класс `one.Ex1`, используя оператор `import`:

```
package one;

// Импортируем класс Ex2
import two.Ex2;

public class Ex1 {
   public static void main(String[] args) {
      // Прописываем класс Ex2 из пакета two.Ex2
      // в качестве объекта класса Ex1,
      // в этот раз без использования полной аннотации класса
      Ex2 ex2 = new Ex2();
      // Вызываем метод объекта ex2, который описывается классом Ex2
      ex2.doWork();
      System.out.print("Done!!!");
   }
}
```

Оператор `import` нужен для удобства программистов, так как уменьшает объем вводимого кода. И хотя он не обязательный, у него есть правила использования:

1. В исходном файле программы операторы `import` должны следовать непосредственно за операторами `package` (если он есть) перед любыми определениями классов.
2. В класс можно импортировать сразу несколько пакетов, в которых есть классы с одинаковыми названиями. В этом случае придется обращаться к ним по полному имени, иначе компилятор не поймет, о каком именно классе идет речь.

В языке Java есть пакет `java.lang`, классы которого уже импортированы в программу по умолчанию. Он содержит в себе самые часто используемые классы.