Модуль 1. Основы программирования

Тема 1.9. Функции

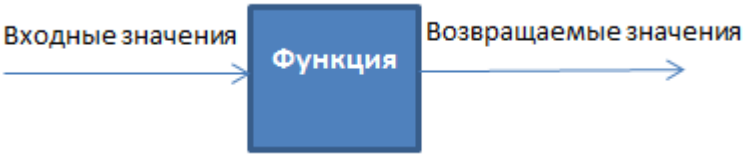
2 часа

Оглавление

1.9. Функции

1.9.1. Определение функции

Функцией называет именованный участок кода, к которому можно обратиться из другой части программы/проекта. Удобно рассматривать функцию, как черный ящик, на вход которого подаются входящие значения, а на выходе получаем результат - возвращаемые значения.



Функция, не имеющая возвращаемых значений в программировании называется ***процедурой.***
Примерами процедур могут являться функции ввода/вывода в языках Pascal и C, а примерами функций - любые математические операции в тех же языках, например sin(x).
Строго говоря, в Java нет процедур и функций. Все дело в том, что Java, как известно, объектно-ориентированный язык и потому в нем нет самостоятельных процедур и функций, они всегда определяются в некотором классе и потому получили другое название - ***методы***.
Более подробно о том, что такое объектно-ориентированный подход мы расскажем во втором модуле, а пока рассмотрим только методы.

Методы определяются в классе, потому так и говорят: "у этого класса есть следующие методы" или "эти методы находятся в данном классе" или "этот метод из данного класса, определен в данном классе". Так же могут сказать, что "метод принадлежит классу".

В определении функции сказано, что это участок кода. Любой ли участок кода можно оформить в виде метода? Конечно, нет. Код, который оформляется в виде отдельного метода должен решать некую отдельную задачу, выполнять операцию, т.е. иметь предназначение.
Рассмотрим пример. Представьте некий объект - книгу. Что с ней можно сделать? Какие элементарные операции можно объеденить в отдельную группу со смыслом и назвать действием? В данном случае это может быть:

1) открыть книгу
2) закрыть книгу
3) сменить страницу и т.д.
4) переместить книгу

Каждое такое действие при создании программы может быть оформлено в виде отдельного метода класса "Книга".
Перейдем к практическим примерам на языке Java - напишем свой метод, чтобы во всем разобраться. Чтобы определить метод разберемся с синтаксисом, принятым в Java для его описания:

```
[модификаторы] тип_возвращаемого_значения название_метода ([параметры]){
    // тело метода
}
```

Разберем части определения метода:

- о ***модификаторах*** мы поговорим позже, присутствуют они не всегда (об этом говорит то, что в определении они указаны в квадратных скобках), сейчас нам часто встречаются модификаторы ***public*** - это спецификация доступа и обозначает возможность доступа к методу из вне описанного класса и ***static*** - это ключевое слово и говорит о том, что метод можно вызвать отдельно от класса и он общий для всех объектов класса;
- ***тип возвращаемого значения*** - это тип результата, который возвращает метод;
- ***название метода*** - для того, чтобы к методу обращаться нужно дать ему имя, причем правилом хорошего тона считается давать методу такое имя, по которому можно догадаться в чем его назначение;
- ***параметры*** - здесь в круглых скобках указывается список входных значений, параметров может и не быть;
- все, что находится до { - это ***заголовок метода (сигнатура)***, то, что находится между фигурными скобками - это ***тело метода*** и туда мы и помещаем код, который будет выполнять предназначение метода.

Имя метода, также как и переменной:

- *должно начинаться только с буквы;*
- *может состоять из любой последовательности строчных и прописных букв, но принято использовать только латинские;*
- *может содержать символы подчеркивания (_) и знака доллара ($);*
- *учитывать регистр (например, имена MyMethod и myMethod компилятор посчитает разными именами);*
- *не должно совпадать с именами операторов и зарезервированных слов самого языка Java;*
- *не должно совпадать с зарезервированными именами методов, таких как clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait. Если только вы не хотите явно переопределить действия этих методов.*

Упражнение 1.9.1

Итак, давайте потренируемся.

Как мы помним вывод в консоль у Java реализован достаточно сложно и содержит в себе пока не изученные нами конструкции. Упростим его и напишем свой метод вывода на экран, похожий на синстаксис Паскаля:

```
//-----------------------------------------------------
//Это метод вывода на экран
public static void writeln(int x){
   System.out.println(x);
}
```

Напишем метод таким образом, чтобы он принимал значение массива, перебирал его и выводил на экран:

```
//А это метод вывода значений массива
public static void writeArray(int[] arr){
   for(int i=0; i<=arr.length-1;i++){
```

```
        writeln(“ Значение массива:  ”+ arr[i]);
    }
}
```

1.9.2. Параметры функции

Параметры - это входящие значения, которые мы передаем в метод. В упражнении 1.9.1 в скобках мы указали в скобках после имени метода сначала int x, а во втором варианте int[] arr. И в том и другом случае это были параметры. мы их передали в метод для того, чтобы использовать их для выполнения операций внутри метода.

Как и при определении перменных, у параметров сначала пишем тип, а затем имя параметра. Если параметров несколько, то записываем эти пары через запятую.

В качестве примера определим метод Walk:

```
public static void Walk(int posx, int posy){
   // что-то делаем в теле метода
}
```

В скобках определены через запятую целые параметры: **int posx, int posy**. Параметров может быть сколько угодно много. Так же допустимо отсутствие параметров вовсе.

Упражнение 1.9.2

Давайте создадим метод, который будет выводить на экран максимум или минимум массива, который мы будем передавать в метод в качестве параметра. Причем будем задавать еще второй параметр `param`, по значению которого метод “узнает”, что ему находить максимум или минимум. Ведь алгоритм поиска максимума отличается от минимума только тем, что на каждом шаге цикла мы ищем элемент, который больше/меньше текущего значения, которое хранится в top.

Запустите программу:

```
public class MyProgram {
	//Метод для вывода целого на экран
	public static void writeln(int x) {
		System.out.println(x);
	}

	//Метод вычисления максимума или минимума
	public static void getTopOfArray(int[] arr, int param) {
		int top = arr[0];
		for (int i = 0; i <= arr.length - 1; i++) {
			if (param == 1) {
				if (arr[i] > top) {
					top = arr[i];
				}
			} else {
				if (arr[i] < top) {
					top = arr[i];
				}
```

```
                }
            }
            writeln(top); //вывод результата
        }

        public static void main(String[] args) {
            int[] a = { 1, 5, 3, 7, 2 }; //задаем массив
            getTopOfArray(a, 1); //вызываем метод

        }
}
```

1.9.3. Возвращаемое значение и вызов метода

В предыдущих упражнениях перед именем метода мы задавали тип возвращаемого значения ***void.*** В таком применении void - это обозначение того, что метод (функция) не возвращает значение (вспомните про процедуры!).

А если нам нужна функция, а не процедура? Например, подсчитать синус или взять квадратный корень? Тогда вместо *void* нужно поставить тип возвращаемого значения. Также это означает, что метод должен обязательно “вернуть” результат и он должен быть совместимым с указанным типом.

В **Java**, как и во многих языках, чтобы вернуть из метода значение, используют оператор ***return***. Например изменим предыдущий пример обработки массива так, чтобы результат не выводился на экран, а “возвращался”.

```
public class MyProgram {

        // Метод вычисления максимума или минимума
        public static int getTopOfArray(int[] arr, int param) {
            int top = arr[0];
            for (int i = 0; i <= arr.length - 1; i++) {
                if (param == 1) {
                    if (arr[i] > top) {
                        top = arr[i];
                    }
                } else {
                    if (arr[i] < top) {
                        top = arr[i];
                    }
                }
            }
            return top; // возвращение результата
        }

        public static void main(String[] args) {
            int[] a = { 1, 5, 3, 7, 2 }; // задаем массив
            //вызываем метод
```

```
            System.out.println("Результат: " + getTopOfArray(a, 1));
        }
}
```

Задание 1.9.1

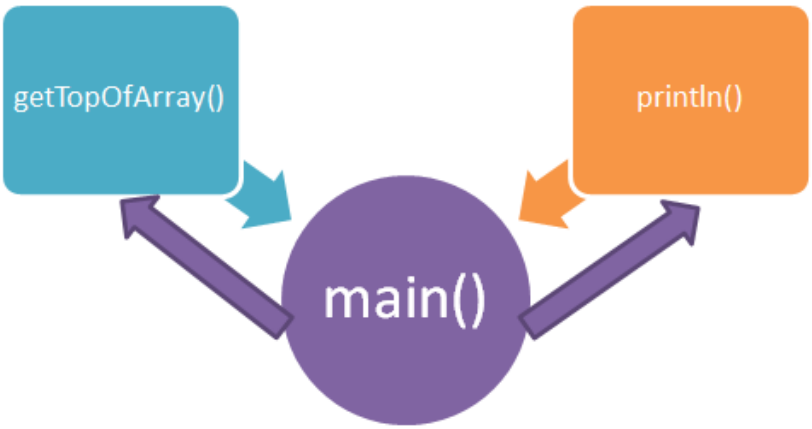
Измените предыдущую программу, сделайте так, чтобы при выводе пользователю было понятно, что было подсчитано, максимум или минимум массива.

Возникает вопрос, зачем возвращать результат из метода через оператор return?

Метод в такой форме позволит нам в головной функции самим решить, что делать с результатом: вывести на экран, использовать в арифметических выражениях и т.д. Надо сказать, такой способ реализации является также более правильным, это придает методу большую универсальность в использовании.

Что происходит в точке вызова метода в тексте программы? В исходном методе компиллятор доходит до вызова второго метода и выполняет его в новой выделенной области памяти, а когда там доходит до оператора return, то помещает укаазнный в операторе результат в место вызова в исходном методе. Второй метод завершает свою работу и выделенная под его выполнение область памяти освобождается.

К примеру для нашего примера цепочка вызовов функций, а потом возврат из них будет выглядеть, как на приведенном ниже рисунке.



*Для того, чтобы компиллятор знал куда возвращаться после выполнения очередной функции (метода) в программировании реализован механизм, который называется **стек вызовов**. Мы еще вернемся к этому вопросу при изучении материалов по теме “Рекурсия”.*

1.9.4. Область видимости переменной

Для того, чтобы разобраться с понятием и назначением области видимости переменных представим себе большую комнату на 500 человек, в которой сидят ученики разных школ. А затем

мы скажем, что у нас есть яблоко и пусть Вика и Сергей его возьмут. Что произойдет в данном случае? Выйдут 10 Викторий и 21 Сергей и попытаются присвоить себе это самое яблоко. Хорошо, если не дойдет до ссоры.

Здесь ситуация аналогичная! Для того, чтобы избежать конфликта имен в методах из разных частей кода мы используем разные области видимости. Но почему нельзя просто называть переменные разными именами, чтобы избежать путаницы? Дело в том, что проблема одинаковых имен - это одна из задач, которую решает разделение по областям видимости. Рассмотрим виды переменных на их уровнях:

1. Глобальные переменные. Это такие переменные, значение которых можно узнать или изменить в любом методе класса
2. Локальные переменные. Это переменные, значения которых могут быть прочитаны или изменены только в месте их инициализации

Сами области видимости можно классифицировать следующим образом:

1. ***Области класса***. Переменные видны во всех методах класса
2. ***Области метода***. Переменные видны только в отдельно взятом методе
3. ***Область блока***. Переменные видны только в отдельно взятом блоке. И вот тут немного подробнее, потому что конструкцию блока мы ранее не рассматривали

Блоком внутри метода будет являться конструкция ограничивающая часть метода фигурными скобками и имеющая свои собственные локальные переменные. К блоковым конструкциям будут относиться ветвления, циклы и сами блоки непосредственно. Отсюда можно определить правило - переменные определенные внутри блока фигурных скобок - видны всем на этом уровне и во всех вложенных блоках.

Как мы уже знаем, в теле метода кроме входящих параметров можно инициализировать свои собственные премененные, которые называют **локальными**. Они существуют только в пределах своего метода, где были созданы и извне, напрямую, мы не можем получить их значения (вспомните, функция для нас непрозрачный ящик!).

В традиционном подходе для передачи/возвращения значений в/из функции нужно использовать механизм параметров и возврат через return, а еще с большой осторожностью глобальные переменные. Однако объектно-ориентированные языки, такие, как Java дают и другие способы, о которых мы будем говорить в следующем модуле.

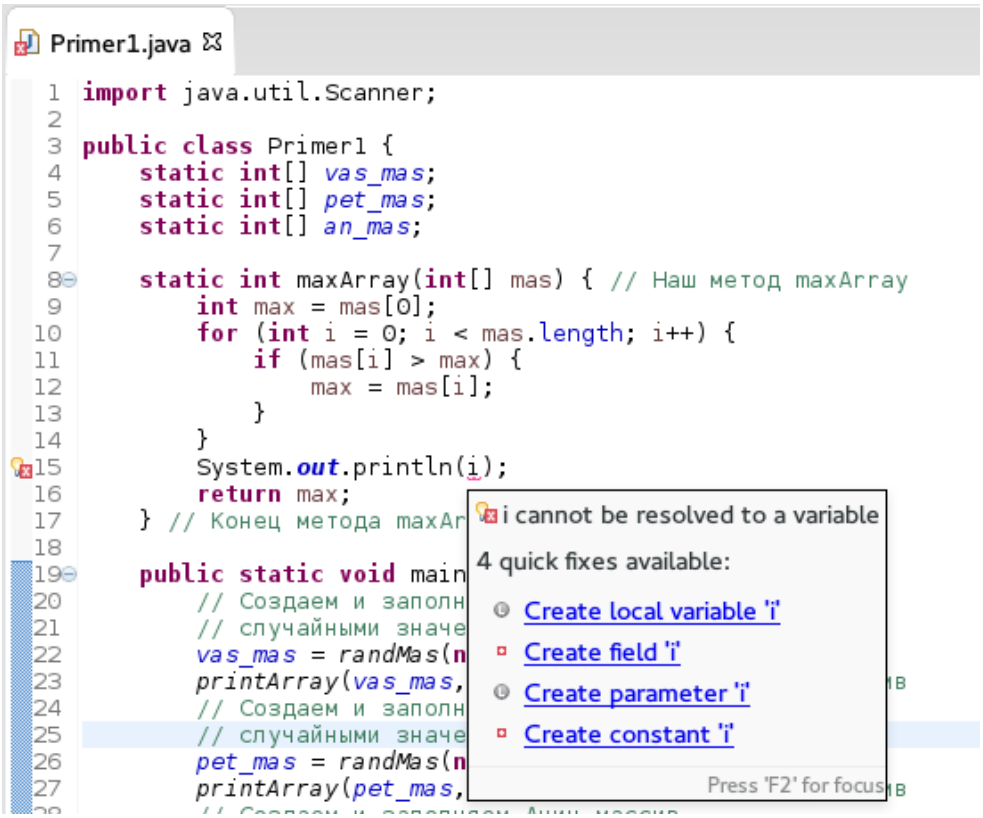
Как, мы уже сказали, переменная, определенная внутри блока не определена вне его. Например, переменная-итератор i не видна вне блока цикла, что приводит к тому, что в следующем цикле можно использовать это же имя (будет создана новая переменная).

Например следующий код не скомпилируется:

```
import java.util.Scanner;

public class Example1 {

      static int maxArray(int[] ar){ // Метод maxArray находит максимум
          int max = ar[0];
          for(int i = 0; i < ar.length; i++){
```

```
                if(ar[i] > max){
                    max = ar[i];
                }
            }
            System.out.println(i); //ОШИБКА! Переменной i нет вне цикла
            return max;
        }

        // … Остальные методы
}
```

Eclipse подобные ошибки отмечает слева лампочкой с красным крестиком, саму переменную подчеркивает красной волнистой линией, а при наведении курсора на переменную подсказывает ошибку и способы решения (так же можно навести курсором на лампочку с красным крестиком, чтобы увидеть только ошибку).



Здесь среди вариантов мы видим следующее:

1. Create local variable - создать локальную переменную. Это может решить проблему но может создать новую. Смотри дальше
2. Create field - создать свойство класса (или еще называют поле)
3. Create parameter - создать входящий параметр
4. Create constant - создать свойство как константу. Тот же вариант что и свойство, только переменная будет финализирована и явщяться уже не переменной, а константой

Чем же чревато создание локальной переменной в данном случае? Если вы попытаетесь создать локальную переменную метода, а в этом методе уже есть такая локальная переменная, то компилятор опять же выдаст ошибку “Duplicate local variable i” (дублирование локальной переменной i). Если нам все же нужен результат, который будет сохранен в i после звершения цикла, то нужно определить эту переменную вне его:

```
import java.util.Scanner;

public class Example1 {

	static int maxArray(int[] ar){ // Метод maxArray находит максимум
	    int max = ar[0];
	    int i;
	    for(i = 0; i < ar.length; i++){
	        if(ar[i] > max){
	            max = ar[i];
	        }
	    }
	    System.out.println(i); //ОШИБКА! Переменной i нет вне цикла
	    return max;
	}

	// … Остальные методы
}
```

Упражнение 1.9.3

Потренируемся с видимостью переменных: создадим переменные для класса, метода и блока с одним именем и выведем их на экран. Увидим: 10 20 40 50

```
public class GlobalPractic {
	private static int globalInt = 10;

	public static void main(String[] args) {
		writeln(globalInt);
		int globalInt = 20;
		writeln(globalInt);
		{
			int IntInBlock = 40;
			writeln(IntInBlock);
		}
		int IntInBlock = 50;
		writeln(IntInBlock);
	}

	public static void writeln(int x) {
		System.out.println(x);
	}
}
```

1.9.5. Не глупые вопросы

Вопрос: А что если у меня есть глобальная переменная star и локальная переменная star. Что тогда произойдет ? Ведь физически это 2 разных переменных и никаких конфликтов быть не может

> **Ответ**: Все верно. Конфликтов не будет, но внутри метода если ты будешь использовать одно и тоже имя то предпочтение будет отдано локальной. Но лучше избегать таких ситуаций - это не лучший стиль программировния. Например, в случае полей класса явно указывают о каком классе идет речь.

Вопрос: А может ли быть в одном классе 2 и более метода с одним именем ?

> **Ответ**: Может быть, но тогда входящие данные должны координально различаться. Например:
> public static void func() { }
> public static void func(int n) { }
> public static void func(float[] arr, int x, int y) { }
> В таких случаях при вызове метода компиллятор найдет и запустит метод с совпадающей сигнатурой. Как называется такой процесс мы рассмотрим на уроках по объектно ориентированному программированию.

Заключение

Мы рассмотрели важнейшие вопросы, связанные с функциями. Давайте подведем итог: зачем в программировании нужны функции?

Во первых, для того, чтобы структурировать код. Для решения больших задах человек всегда делит их на более мелкие. В этом смысле функции - это способ правильного разделения большой задачи на более мелкие, и в результате код становится более понятным.

Во-вторых, функции позволяют использовать выделенные участки кода повторно. Например, каждый раз, где нам нужно вывести что-то на экран, мы не реализуем все с нуля, а просто вызываем уже готовую функцию.

В-третьих, функции значительно облегчают внесение в него изменений. А это очень важно, потому что во время жизненного цикла программы она не раз потребует изменений. Например, функция подсчитывала размер налога, но законодательство изменилось и формула подсчета стала другой, тогда разработчику достаточно внести изменения только в одной функции, а не по всему коду, где этот подсчет применялся.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Киселеву Павлу Георгиевичу и Тазетдинову Андрею Мансуровичу.