Раздел 1 (Знакомство с языком программирования Java и освоение основных конструкций языка)

Особенности Java

Ключевой особенностью языка Java является то, что его код сначала транслируется в специальный байт-код, независимый от платформы. А затем этот байт-код выполняется виртуальной машиной JVM (Java Virtual Machine). В этом плане Java отличается от стандартных интерпретируемых языков как PHP или Perl, код которых сразу же выполняется интерпретатором. В то же время Java не является и чисто компилируемым языком, как C или C++.

Подобная архитектура обеспечивает кроссплатформенность и аппаратную переносимость программ на Java, благодаря чему подобные программы без перекомпиляции могут выполняться на различных платформах - Windows, Linux, Mac OS и т.д. Для каждой из платформ может быть своя реализация виртуальной машины JVM, но каждая из них может выполнять один и тот же код.

Java является языком с Си-подобным синтаксисом и близок в этом отношении к C/C++ и C#. Поэтому, если вы знакомы с одним из этих языков, то овладеть Java будет легче.

Еще одной ключевой особенностью Java является то, что она поддерживает автоматическую сборку мусора. А это значит, что вам не надо освобождать вручную память от ранее использовавшихся объектов, как в C++, так как сборщик мусора это сделает автоматически за вас.

Java является объектно-ориентированным языком. Он поддерживает полиморфизм, наследование, статическую типизацию. Объектно-ориентированный подход позволяет решить задачи по построению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений.

Структура программы

Основным строительным блоком программы на языке Java являются **инструкции** (statement). Каждая инструкция выполняет некоторое действие, например, вызовы методов, объявление переменных и присвоение им значений. После завершения инструкции в Java ставится точка с запятой (;). Данный знак указывает компилятору на конец инструкции. Например:

```
1 System.out.println("Hello Java!");
```

Данная строка представляет вызов метода `System.out.println`, который выводит на консоль строку "Hello Java!". В данном случае вызов метода является инструкцией и поэтому завершается точкой с запятой.

Кроме отдельных инструкций распространенной конструкцией является блок кода. Блок кода содержит набор инструкций, он заключается в фигурные скобки,

а инструкции помещаются между открывающей и закрывающей фигурными скобками:

```
{
    System.out.println("Hello!");
    System.out.println("Welcome to
Java!");
}
```

В этом блоке кода две инструкции, которые выводят на консоль определенную строку.

Выполнение программы. Метод main

Java является объектно-ориентированным языком, поэтому всю программу можно представить как набор взаимодействующих между собой классов и объектов. В первой главе при создании первого приложения программа была определена следующим образом:

```
public class Program{

    public static void main (String
args[]){

        System.out.println("Hello
Java!");
    }
}
```

То есть основу нашей программы составляет класс Program. При определении класса вначале идет модификатор доступа **public**, который указывает, что данный класс будет доступен всем, то есть мы сможем его запустить из командной строки. Далее идет ключевое слово **class**, а затем название класса. После названия класса идет блок кода, в котором расположено содержимое класса.

Входной точкой в программу на языке Java является метод **main**, который определен в классе Program. Именно с него начинается выполнение программы. Он обязательно должен присутствовать в программе. При этом его заголовок может быть только таким:

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

При запуске приложения виртуальная машина Java ищет в главном классе программы метод main с подобным заголовком, и после его обнаружения запускает его.

Вначале заголовка метода идет модификатор `public`, который указывает, что метод будет доступен извне. Слово `static` указывает, что метод main - статический, а слово `void` - что он не возвращает никакого значения. Далее в скобках у нас идут параметры метода - `String args[]` - это массив args, который

хранит значения типа String, то есть строки. При запуске программы через этот массив мы можем передать в программу различные данные.

После заголовка метода идет его блок, который содержит набор выполняемых инструкций.

Переменные и константы

Для хранения данных в программе предназначены **переменные**. Переменная представляет именованную область памяти, которая хранит значение определенного типа. Каждая переменная имеет тип, имя и значение. Тип определяет, какую информацию может хранить переменная или диапазон допустимых значений.

Переменные объявляются следующим образом:

```
тип_данных имя_переменной;
```

Например, определим переменную, которая будет называться x и будет иметь тип `int`:

```
int x;
```

В этом выражении мы объявляем переменную x типа `int`. То есть x будет хранить некоторое число не больше 4 байт.

В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

- имя может содержать любые алфавитно-цифровые символы, а также знак подчеркивания, при этом первый символ в имени не должен быть цифрой
- в имени не должно быть знаков пунктуации и пробелов
- имя не может быть ключевым словом языка Java

Кроме того, при объявлении и последующем использовании надо учитывать, что Java - регистрозависимый язык, поэтому следующие объявления `int num;` и `int NUM;` будут представлять две разных переменных.

Объявив переменную, мы можем присвоить ей значение:

```
int x;       // объявление переменной
x = 10;      // присвоение значения
System.out.println(x);  // 10
```

Также можно присвоить значение переменной при ее объявлении. Этот процесс называется **инициализацией**:

```
int x = 10;     // объявление и инициализация переменной
System.out.println(x);  // 10
```

Если мы не присвоим переменной значение до ее использования, то мы можем получить ошибку, например, в следующем случае:

```
int x;
System.out.println(x);
```

Через запятую можно объявить сразу несколько переменных одного типа:

```
int x, y;
x = 10;
y = 25;
System.out.println(x);  // 10
System.out.println(y);  // 25
```

Также можно их сразу инициализировать:

```
int x = 8, y = 15;
System.out.println(x);  // 8
System.out.println(y);  // 15
```

Отличительной особенностью переменных является то, что мы можем в процессе работы программы изменять их значение:

```
int x = 10;
System.out.println(x);  // 10
x = 25;
System.out.println(x);  // 25
```

Ключевое слово var

Начиная с Java 10 в язык было добавлено ключевое слово **var**, которое также позволяет определять переменную:

```
var x = 10;
System.out.println(x);  // 10
```

Слово `var` ставится вместо типа данных, а сам тип переменной выводится из того значения, которое ей присваивается. Например, переменной x присваивается число 10, значит, переменная будет представлять тип int.

Но если переменная объявляется с помощью var, то мы обязательно должны инициализировать ее, то есть предоставить ей начальное значение, иначе мы получим ошибку, как, например, в следующем случае:

```
var x;      // ! Ошибка, переменная
не инициализирована
x = 10;
```

Константы

Кроме переменных, в Java для хранения данных можно использовать **константы**. В отличие от переменных константам можно присвоить значение только один раз. Константа объявляется также, как и переменная, только вначале идет ключевое слово **final**:

```
final int LIMIT = 5;
System.out.println(LIMIT);  // 5
// LIMIT=57; // так мы уже не можем
написать, так как LIMIT - константа
```

Как правило, константы имеют имена в верхнем регистре.

Константы позволяют задать такие переменные, которые не должны больше изменяться. Например, если у нас есть переменная для хранения числа pi, то мы можем объявить ее константой, так как ее значение постоянно.

Типы данных

Одной из основных особенностей Java является то, что данный язык является строго типизированным. А это значит, что каждая переменная и константа представляет определенный тип и данный тип строго определен. Тип данных определяет диапазон значений, которые может хранить переменная или константа.

Итак, рассмотрим систему встроенных базовых типов данных, которая используется для создания переменных в Java. А она представлена следующими типами.

- **boolean**: хранит значение `true` или `false`

```
boolean isActive = false;
boolean isAlive = true;
```

- **byte**: хранит целое число от `-128` до `127` и занимает 1 байт

```
byte a = 3;
byte b = 8;
```

- **short**: хранит целое число от `-32768` до `32767` и занимает 2 байта

```
short a = 3;
short b = 8;
```

- **int**: хранит целое число от `-2147483648` до `2147483647` и занимает 4 байта

```
int a = 4;
int b = 9;
```

- **long**: хранит целое число от `–9 223 372 036 854 775 808` до `9 223 372 036 854 775 807` и занимает 8 байт

```
long a = 5;
long b = 10;
```

- **double**: хранит число с плавающей точкой от $\pm 4.9*10^{-324}$ до $\pm 1.7976931348623157*10^{308}$ и занимает 8 байт

```
double x = 8.5;
double y = 2.7;
```

В качестве разделителя целой и дробной части в дробных литералах используется точка.

- **float**: хранит число с плавающей точкой от $-3.4*10^{38}$ до $3.4*10^{38}$ и занимает 4 байта

```
float x = 8.5F;
float y = 2.7F;
```

- **char**: хранит одиночный символ в кодировке UTF-16 и занимает 2 байта, поэтому диапазон хранимых значений от `0` до `65535`
  При этом переменная может принимать только те значения, которые соответствуют ее типу. Если переменная представляет целочисленный тип, то она не может хранить дробные числа.

  Целые числа
  Все целочисленные литералы, например, числа 10, 4, -5, воспринимаются как значения типа **int**, однако мы можем присваивать целочисленные литералы другим целочисленным типам: **byte**, **long**, **short**. В этом случае Java автоматически осуществляет соответствующие преобразования:

```
byte a = 1;
short b = 2;
long c = 2121;
```

  Однако если мы захотим присвоить переменной типа long очень большое число, которое выходит за пределы допустимых значений для типа int, то мы столкнемся с ошибкой во время компиляции:

```
long num = 2147483649;
```

  Здесь число 2147483649 является допустимым для типа long, но выходит за предельные значения для типа int. И так как все целочисленные значения по умолчанию расцениваются как значения типа int, то компилятор укажет нам на ошибку. Чтобы решить проблему, надо добавить к числу суффикс **l** или **L**, который указывает, что число представляет тип long:

```
long num = 2147483649L;
```

  Как правило, значения для целочисленных переменных задаются в десятичной системе счисления, однако мы можем применять и другие системы счисления. Например:

```
int num111 = 0x6F; // 16-теричная
система, число 111
int num8 = 010; // 8-ричная система,
число 8
int num13 = 0b1101; // 2-ичная
система, число 13
```

  Для задания шестнадцатеричного значения после символов **0x** указывается число в шестнадцатеричном формате. Таким же образом восьмеричное значение указывается после символа **0**, а двоичное значение - после символов **0b**.

  Также целые числа поддерживают разделение разрядов числа с помощью знака подчеркивания:

```
int x = 123_456;
int y = 234_567__789;
System.out.println(x);  // 123456

```

```
System.out.println(y);  //
234567789
```

Числа с плавающей точкой

При присвоении переменной типа float дробного литерала с плавающей точкой, например, 3.1, 4.5 и т.д., Java автоматически рассматривает этот литерал как значение типа `double`. И чтобы указать, что данное значение должно рассматриваться как `float`, нам надо использовать суффикс f:

```
1  float fl = 30.6f;
2  double db = 30.6;
```

И хотя в данном случае обе переменных имеют практически одно значения, но эти значения будут по-разному рассматриваться и будут занимать разное место в памяти.

Символы и строки

В качестве значения переменная символьного типа получает одиночный символ, заключенный в одинарные кавычки: `char ch='e';`. Кроме того, переменной символьного типа также можно присвоить целочисленное значение от `0` до `65535`. В этом случае переменная опять же будет хранить символ, а целочисленное значение будет указывать на номер символа в таблице символов Unicode (UTF-16). Например:

```
1  char ch=102; // символ 'f'
2  System.out.println(ch);
```

Еще одной формой задания символьных переменных является шестнадцатеричная форма: переменная получает значение в шестнадцатеричной форме, которое следует после символов "\u". Например, `char ch='\u0066';` опять же будет хранить символ 'f'.

Символьные переменные не стоит путать со строковыми, 'a' не идентично "a". Строковые переменные представляют объект `String`, который в отличие от char или int не является примитивным типом в Java:

```
1  String hello = "Hello...";
2  System.out.println(hello);
```

Кроме собственно символов, которые представляют буквы, цифры, знаки препинания, прочие символы, есть специальные наборы символов, которые называют управляющими последовательностями. Например, самая популярная последовательность - "\n". Она выполняет перенос на следующую строку. Например:

```
1  String text = "Hello \nworld";
2  System.out.println(text);
```

Результат выполнения данного кода:

```
Hello
world
```

В данном случае последовательность \n будет сигналом, что необходимо сделать перевод на следующую строку.

Начиная с версии 15 Java поддерживает тестовые блоки (text blocks) - многострочный текст, облеченный в тройные кавычки. Рассмотрим, в чем их практическая польза. Например, выведем большой многострочный текст:

```
String text = "Вот мысль, которой
весь я предан,\n"+
                "Итог всего, что ум
скопил.\n"+
                "Лишь тот, кем бой
за жизнь изведан,\n"+
                "Жизнь и свободу
заслужил.";
System.out.println(text);
```

С помощью операции + мы можем присоединить к одному тексту другой, причем продолжение текста может располагаться на следующей строке. Чтобы при выводе текста происходил перенос на следующую строку, применяется последовательность \n.

Результат выполнения данного кода:

```
Вот мысль, которой весь я предан,
Итог всего, что ум скопил.
Лишь тот, кем бой за жизнь изведан,
Жизнь и свободу заслужил.
```

Текстовые блоки, которые появились в JDK15, позволяют упростить написание многострочного текста:

```
String text = """
                Вот мысль, которой
весь я предан,
                Итог всего, что ум
скопил.
                Лишь тот, кем бой
за жизнь изведан,
                Жизнь и свободу
заслужил.
                """;
```

```
System.out.println(text);
```

Весь текстовый блок оборачивается в тройные кавычки, при этом не надо использовать соединение строк или последовательность \n для их переноса. Результат выполнения программы будет тем же, что и в примере выше.

Циклы

Еще одним видом управляющих конструкций являются циклы. Циклы позволяют в зависимости от определенных условий выполнять определенное действие множество раз. В языке Java есть следующие виды циклов:

- for
- while
- do...while

Цикл for

Цикл for имеет следующее формальное определение:

```
for ([инициализация счетчика];
[условие]; [изменение счетчика])
{
    // действия
}
```

Рассмотрим стандартный цикл for:

```
for (int i = 1; i < 9; i++){
    System.out.printf("Квадрат
числа %d равен %d \n", i, i * i);
}
```

Первая часть объявления цикла - `int i = 1` создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип `int`. Это может быть и любой другой числовой тип, например, float. Перед выполнением цикла значение счетчика будет равно 1. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока i не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: `i--`.

В итоге блок цикла сработает 8 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

```
int i = 1;
for (; ;){
```

```
    System.out.printf("Квадрат
числа %d равен %d \n", i, i * i);
}
```

Определение цикла осталось тем же, только теперь блоки в определении у нас пустые: `for (; ;)`. Теперь нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Либо можно опустить ряд блоков:

```
int i = 1;
for (; i<9;){
    System.out.printf("Квадрат
числа %d равен %d \n", i, i * i);
    i++;
}
```

Этот пример эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

Цикл for может определять сразу несколько переменных и управлять ими:

```
int n = 10;
for(int i=0, j = n - 1; i < j; i++,
j--){

    System.out.println(i * j);
}
```

Цикл do

Цикл do сначала выполняет код цикла, а потом проверяет условие в инструкции while. И пока это условие истинно, цикл повторяется. Например:

```
int j = 7;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

В данном случае код цикла сработает 7 раз, пока j не окажется равным нулю. Важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно. Так, мы можем написать:

```
int j = -1;
do{
    System.out.println(j);
    j--;
}
```

```
6 while (j > 0);
```

Хотя переменная j изначально меньше 0, цикл все равно один раз выполнится.

Цикл while

Цикл `while` сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
1 int j = 6;
2 while (j > 0){
3 
4     System.out.println(j);
5     j--;
6 }
```

Операторы continue и break

Оператор **break** позволяет выйти из цикла в любой его момент, даже если цикл не закончил свою работу:

Например:

```
1 for (int i = 0; i < 10; i++){
2     if (i == 5)
3         break;
4     System.out.println(i);
5 }
```

Когда счетчик станет равным 5, сработает оператор break, и цикл завершится.

Теперь сделаем так, чтобы если число равно 5, цикл не завершался, а просто переходил к следующей итерации. Для этого используем оператор **continue**:

```
1 for (int i = 0; i < 10; i++){
2     if (i == 5)
3         continue;
4     System.out.println(i);
5 }
```

В этом случае, когда выполнение цикла дойдет до числа 5, программа просто пропустит это число и перейдет к следующему.

Классы и объекты

Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

Шаблоном или описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, туловища и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом.

Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

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

```
class Person{

}
```

В данном случае класс называется Person. После названия класса идут фигурные скобки, между которыми помещается тело класса - то есть его поля и методы.

Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.

Для хранения состояния объекта в классе применяются поля или переменные класса. Для определения поведения объекта в классе применяются методы. Например, класс Person, который представляет человека, мог бы иметь следующее определение:

```
class Person{

    String name;         // имя
    int age;             // возраст
    void displayInfo(){
        System.out.printf("Name: %s
\tAge: %d\n", name, age);
    }
}
```

В классе Person определены два поля: name представляет имя человека, а age - его возраст. И также определен метод displayInfo, который ничего не возвращает и просто выводит эти данные на консоль.

Теперь используем данный класс. Для этого определим следующую программу:

```
public class Program{

    public static void main(String[]
args) {

        Person tom;
    }
}
class Person{

    String name;     // имя
    int age;         // возраст
    void displayInfo(){
```

```
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Как правило, классы определяются в разных файлах. В данном случае для простоты мы определяем два класса в одном файле. Стоит отметить, что в этом случае только один класс может иметь модификатор public (в данном случае это класс Program), а сам файл кода должен называться по имени этого класса, то есть в данном случае файл должен называться Program.java.

Класс представляет новый тип, поэтому мы можем определять переменные, которые представляют данный тип. Так, здесь в методе main определена переменная `tom`, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение **null**. По большому счету мы ее пока не можем использовать, поэтому вначале необходимо создать объект класса Person.

Конструкторы

Кроме обычных методов классы могут определять специальные методы, которые называются **конструкторами**. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров.

Выше определенный класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию, который мы можем использовать для создания объекта Person. В частности, создадим один объект:

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person();
// создание объекта
        tom.displayInfo();

        // изменяем имя и возраст
        tom.name = "Tom";
        tom.age = 34;
        tom.displayInfo();
    }
}
class Person{

    String name;    // имя
```

```
    int age;        // возраст
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Для создания объекта Person используется выражение `new Person()`. Оператор **new** выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная `tom` получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, `tom.name = "Tom"`.

В итоге мы увидим на консоли:

```
Name: null      Age: 0
Name: Tom       Age: 34
```

Если необходимо, чтобы при создании объекта производилась какая-то логика, например, чтобы поля класса получали какие-то определенные значения, то можно определить в классе свои конструкторы. Например:

```
public class Program{

    public static void main(String[] args) {

        Person bob = new Person(); // вызов первого конструктора без параметров
        bob.displayInfo();

        Person tom = new Person("Tom"); // вызов второго конструктора с одним параметром
        tom.displayInfo();

```

```
        Person sam = new
Person("Sam", 25); // вызов
третьего конструктора с двумя
параметрами
        sam.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст
    Person()
    {
        name = "Undefined";
        age = 18;
    }
    Person(String n)
    {
        name = n;
        age = 18;
    }
    Person(String n, int a)
    {
        name = n;
        age = a;
    }
    void displayInfo(){
        System.out.printf("Name: %s
\tAge: %d\n", name, age);
    }
}
```

Теперь в классе определено три коструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса.

Консольный вывод программы:

```
Name: Undefined      Age: 18
Name: Tom            Age: 18
Name: Sam            Age: 25
```

Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы. Например:

```
public class Program{

    public static void main(String[]
args) {

        Person undef = new Person();
        undef.displayInfo();

        Person tom = new
Person("Tom");
        tom.displayInfo();

        Person sam = new
Person("Sam", 25);
        sam.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст
    Person()
    {
        this("Undefined", 18);
    }
    Person(String name)
    {
        this(name, 18);
    }
    Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s
\tAge: %d\n", name, age);
    }
}
```

В третьем конструкторе параметры называются так же, как и поля класса. И чтобы разграничить поля и параметры, применяется ключевое слово this:

```
this.name = name;
```

Так, в данном случае указываем, что значение параметра name присваивается полю name.

Кроме того, у нас три конструктора, которые выполняют идентичные действия: устанавливают поля name и age. Чтобы избежать повторов, с помощью this можно вызвать один из конструкторов класса и передать для его параметров необходимые значения:

```
Person(String name)
{
    this(name, 18);
}
```

В итоге результат программы будет тот же, что и в предыдущем примере.

Инициализаторы

Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта. Инициализатор выполняется до любого конструктора. То есть в инициализатор мы можем поместить код, общий для всех конструкторов:

```
public class Program{

    public static void main(String[]
args) {

        Person undef = new Person();
        undef.displayInfo();

        Person tom = new
Person("Tom");
        tom.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст

    /*начало блока инициализатора*/
    {
        name = "Undefined";
        age = 18;
    }
    /*конец блока инициализатора*/
    Person(){

    }
    Person(String name){

```

```
        this.name = name;
    }
    Person(String name, int age){

        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s
\tAge: %d\n", name, age);
    }
}
```

ООП

Наследование

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Например, имеется следующий класс Person, описывающий отдельного человека:

```
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: "
+ name);
    }
}
```

И, возможно, впоследствии мы захотим добавить еще один класс, который описывает сотрудника предприятия - класс Employee. Так как этот класс реализует тот же функционал, что и класс Person, поскольку сотрудник - это также и человек, то было бы рационально сделать класс Employee производным

(наследником, подклассом) от класса Person, который, в свою очередь, называется базовым классом, родителем или суперклассом:

```
class Employee extends Person{
    public Employee(String name){
        super(name);    // если базовый класс определяет конструктор
                        //  то производный класс должен его вызвать
    }
}
```

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идет имя базового класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

Если в базовом классе определены конструкторы, то в конструкторе производного классы необходимо вызвать один из конструкторов базового класса с помощью ключевого слова **super**. Например, класс Person имеет конструктор, который принимает один параметр. Поэтому в классе Employee в конструкторе нужно вызвать конструктор класса Person. То есть вызов `super(name)` будет представлять вызов конструктора класса Person.

При вызове конструктора после слова `super` в скобках идет перечисление передаваемых аргументов. При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса. Таким образом, установка имени сотрудника делегируется конструктору базового класса.

Причем даже если производный класс никакой другой работы не производит в конструкторе, как в примере выше, все равно необходимо вызвать конструктор базового класса.

Использование классов:

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");
        tom.display();
        Employee sam = new Employee("Sam");
        sam.display();
```

```
    }
}
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: "
+ name);
    }
}
class Employee  extends Person{
    public Employee(String name){
        super(name);    // если
базовый класс определяет
конструктор
                        //  то
производный класс должен его
вызвать
    }
}
```

Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором **private**. При этом производный класс также может добавлять свои поля и методы:

```
public class Program{

    public static void main(String[]
args) {

        Employee sam = new
Employee("Sam", "Microsoft");
        sam.display();  // Sam
        sam.work();     // Sam
works in Microsoft
```

```
    }
}
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: "
+ name);
    }
}
class Employee  extends Person{

    String company;

    public Employee(String name,
String company) {

        super(name);
        this.company=company;
    }
    public void work(){
        System.out.printf("%s works
in %s \n", getName(), company);
    }
}
```

В данном случае класс Employee добавляет поле company, которое хранит место работы сотрудника, а также метод work.

Переопределение методов

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса. Например, переопределим в классе Employee метод display:

```
public class Program{


```

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

        Employee sam = new Employee("Sam", "Microsoft");
        sam.display();  // Sam
                        // Works in Microsoft
    }
}
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee  extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    @Override
    public void display(){

        System.out.printf("Name: %s \n", getName());
        System.out.printf("Works in %s \n", company);
```

```
    }
}
```

Перед переопределяемым методом указывается аннотация **@Override**. Данная аннотация в принципе необязательна.

При переопределении метода он должен иметь уровень доступа не меньше, чем уровень доступа в базовом класса. Например, если в базовом классе метод имеет модификатор public, то и в производном классе метод должен иметь модификатор public.

Однако в данном случае мы видим, что часть метода display в Employee повторяет действия из метода display базового класса. Поэтому мы можем сократить класс Employee:

```
class Employee extends Person{

    String company;

    public Employee(String name,
String company) {

        super(name);
        this.company=company;
    }
    @Override
    public void display(){

        super.display();
        System.out.printf("Works in
%s \n", company);
    }
}
```

С помощью ключевого слова super мы также можем обратиться к реализации методов базового класса.

Запрет наследования

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

```
public final class Person {
}
```

Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как мы тем самым запретили наследование:

```
class Employee extends Person{ {
}
```

Кроме запрета наследования можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод `display()`, запретим его переопределение:

```
public class Person {

    //.....................

    public final void display(){

        System.out.println("Имя: " +
name);
    }
}
```

В этом случае класс Employee не сможет переопределить метод display.

Динамическая диспетчеризация методов

Наследование и возможность переопределения методов открывают нам большие возможности. Прежде всего мы можем передать переменной суперкласса ссылку на объект подкласса:

```
Person sam = new Employee("Sam",
"Oracle");
```

Так как Employee наследуется от Person, то объект Employee является в то же время и объектом Person. Грубо говоря, любой работник предприятия одновременно является человеком.

Однако несмотря на то, что переменная представляет объект Person, виртуальная машина видит, что в реальности она указывает на объект Employee. Поэтому при вызове методов у этого объекта будет вызываться та версия метода, которая определена в классе Employee, а не в Person. Например:

```
public class Program{

    public static void main(String[]
args) {

        Person tom = new
Person("Tom");
        tom.display();
        Person sam = new
Employee("Sam", "Oracle");
        sam.display();
    }
}
class Person {

```

```
    String name;

    public String getName() { return
name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

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

class Employee extends Person{

    String company;

    public Employee(String name,
String company) {

        super(name);
        this.company = company;
    }
    @Override
    public void display(){

        System.out.printf("Employee
%s works in %s \n",
super.getName(), company);
    }
}
```

Консольный вывод данной программы:

```
Person Tom
Employee Sam works in Oracle
```

При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе.

Данный процесс еще называется **dynamic method lookup** или динамический поиск метода или динамическая диспетчеризация методов.