Таблица виртуальных методов

Статическое связывание

Допустим, у нас есть простой класс:

```
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
        doNothing();
    }

    private void doNothing() {  }

    public String getName() { return name; }

    @Override
    public String toString() { return name; }
}
```

Все его методы, а также методы, унаследованные от `Object`, лежат в памяти по каким-то адресам, и JVM должна знать, как вызвать каждый конкретный метод — ведь процессор ничего не знает о языке Java и его синтаксисе, и для вызова какого-либо блока кода ему нужен адрес в памяти. Как же выглядит вызов методов объекта под капотом?

Проще всего дело обстоит с методами, которые не могут быть переопределены. Это конструкторы, методы с доступом `private` и статические методы. Для них JVM всегда точно знает, по какому адресу в памяти они определены, и может вызвать их непосредственно, используя этот адрес. Так, вызов конструктора

```
Person musketeer = new Person("Д'Артаньян");
```

под капотом, в псевдокоде, выполняется примерно таким образом:

```
Person tmp = allocateMemory(sizeof(Person));
setClass(tmp, Person.class);
Person.constructor(tmp, "Д'Артаньян");
Person musketeer = tmp;
```

То есть сначала выделяется память, достаточная для хранения объекта класса `Person`, затем ей устанавливается признак того, что это именно объект класса `Person`, после чего для этой выделенной области памяти вызывается конструктор. Обратите внимание, что под капотом в конструктор передаётся ссылка на объект — это неявный параметр `this`. С его помощью конструкторы и методы объекта узнают, для какого объекта они вызываются.

И, соответственно, вызов `private`-метода внутри конструктора просто обратится к известному адресу

этого метода. Если расписать наш конструктор явно, в псевдокоде получится что-то вроде:

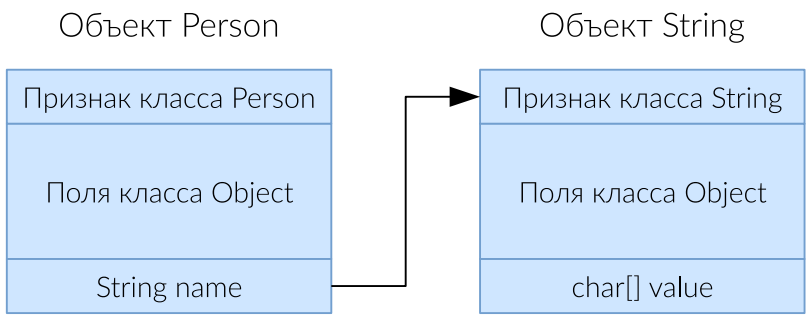
```
void Person.constructor(implicit Person this, String name) {
    Object.constructor(this);
    this.name = name;
    Person.doNothing(this);
}
```

(Не ищите в языке Java ключевые слова `constructor` или `implicit` (неявный). Это не Java-код.)

Как мы помним, любой конструктор в конце концов вызывает конструктор суперкласса. В нашем случае это конструктор класса `Object` без явных параметров, которому передаётся только неявный параметр `this`. Компилятор вставляет первой строкой конструктора вызов конструктора суперкласса без параметров, если не был явно вызван никакой другой конструктор. Это гарантирует, что и данные суперкласса, и данные подкласса будут правильно инициализированы.

Аналогичным образом вызываются и статические методы — с той лишь разницей, что они принадлежат классу, а не объектам этого класса, и неявный параметр `this` в них не передаётся.

Итак, JVM в этом случае ещё на этапе загрузки классов (и компиляции из байт-кода в машинный код) знает, как связать вызовы методов с конкретными адресами вызываемых методов. Узнавать эти адреса на этапе выполнения программы не нужно. Поэтому такой тип связывания называется *статическим связыванием*.



Динамическое связывание

Теперь усложним задачу. В классе `Person` также переопределяется метод `toString`, определение которого было унаследовано от класса `Object`. Мощь переопределяемых методов, известная в терминологии ООП как *полиморфизм*, заключается в том, что в итоге будет вызван правильный метод `Person.toString` даже в случае, если работа с объектом класса `Person` ведётся через ссылку на тип `Object`!

```
Object obj = new Object();
Object wreckIt = new Person("Ральф");
System.out.println(obj);      // java.lang.Object@7d4991ad
System.out.println(wreckIt);  // Ральф
```

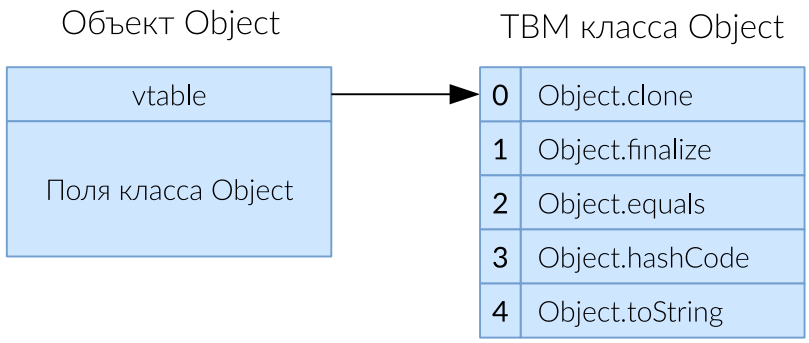
Как мы помним, метод `println` с параметром типа `Object` под капотом вызывает `toString`, то есть вышеуказанный код эквивалентен:

```
System.out.println(obj.toString());
System.out.println(wreckIt.toString());
```

Но метод `println` ничего не знает о нашем классе `Person`. Он является частью стандартной библиотеки Java и был написан за много лет до того, как мы решили написать класс `Person`. Метод `println` просто принимает ссылку на объект типа `Object` и вызывает у неё метод `toString` — и каким-то образом вызывается метод, соответствующий фактическому типу объекта, на который эта ссылка указывает. Так и достигается полиморфизм — способность одного и того же кода корректно работать с данными разных типов, даже типов, которые ещё не существовали на момент написания этого кода.

Как же реализовано *динамическое связывание* вызова метода с самим вызываемым методом?

На самом деле скрытое поле, которое на предыдущей диаграмме называлось "признак класса", само является ссылкой. Эта ссылка указывает на структуру данных, существующую в единственном экземпляре для каждого класса и называемую *таблицей виртуальных методов* (ТВМ; в англоязычной литературе также распространено сокращение `vtable`).



Термин "виртуальный метод" пришёл из языка C++, где переопределяемые методы обозначаются ключевым словом `virtual`. Несмотря на название, эти методы вполне себе реальны и загружаются в память так же, как любые другие методы. В языке Java нет ключевого слова `virtual`, и в документации по языку Java термин "виртуальный метод" обычно не используется, хотя его можно встретить в коде самой JVM и в дизассемблированном байт-коде классов Java.

Каждый класс наследует от класса `Object` пять переопределяемых методов: `clone`, `finalize`, `equals`, `hashCode` и `toString`. Поэтому первые пять элементов ТВМ (которая, по сути, является массивом ссылок на методы) будут ссылаться на реализации этих пяти методов класса `Object` — хотя и необязательно именно в таком порядке.

В нашей воображаемой реализации JVM метод `toString` имеет индекс 4 в ТВМ, и поскольку все классы наследуют от `Object`, то индекс 4 во всех ТВМ всех классов будет ссылаться на реализацию метода `toString` именно в этом классе. И тогда вызов

```
obj.toString();
```

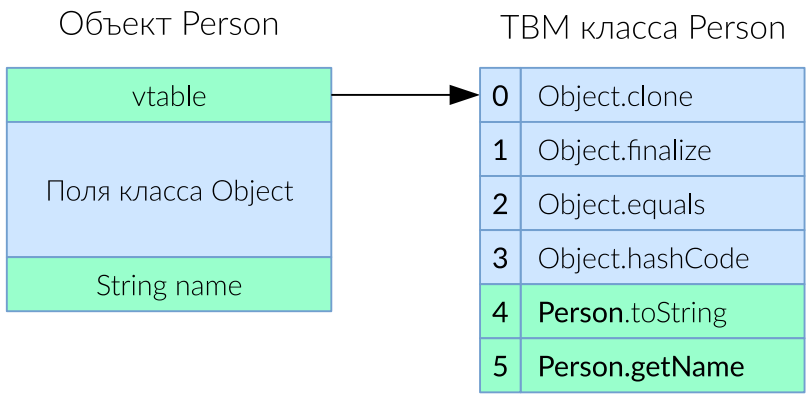
под капотом реализуется как:

```
obj.vtable[4](obj);
```

Естественно, это опять-таки псевдокод. У объектов Java нет свойства, называемого `vtable`, ведь ТВМ — это деталь реализации JVM. И опять-таки в метод передаётся ссылка на сам объект — как неявный параметр `this`. Поскольку в начале каждого объекта располагаются поля класса `Object`, методы класса `Object` могут работать со ссылкой на любой объект так, как если бы это был объект класса `Object`,

игнорируя любые дополнительные поля, с которыми они работать не умеют.

А как выглядит ТВМ для класса Person?



Класс `Person` переопределяет метод `toString`, поэтому в его ТВМ по индексу 4 находится ссылка не на конкретный метод `Object.toString`, а на метод `Person.toString`. Вышеприведённый код, однако, по-прежнему работает и со ссылкой на объект типа `Person`, просто вызывая метод по индексу 4. Кроме того, класс `Person` объявляет новый метод `getName`, которого нет в классе `Object`. Этот метод не помечен как `final`, поэтому его тоже может переопределить какой-нибудь подкласс класса `Person`, и ссылка на него добавляется в конец ТВМ — после ссылок на методы класса `Object`.

И именно поэтому — поскольку в начале ТВМ каждого класса находится копия ТВМ суперкласса, в которой, возможно, изменены некоторые ссылки для переопределённых методов — вызов `obj.toString()` вызывает метод нужного класса (`Object.toString` или `Person.toString`) согласно фактическому типу объекта, даже если ссылка `obj` имеет тип `Object`.

Допустим, мы объявили ещё какой-нибудь класс, наследующий от `Person`:

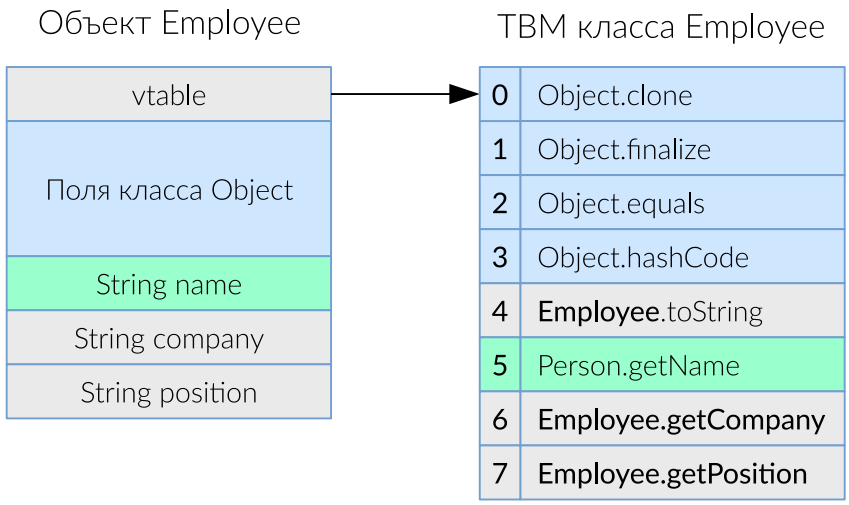
```
public class Employee extends Person {
    private final String company;
    private final String position;

    public Employee(String name, String company, String position) {
        super(name);  // под капотом: Person.constructor(this, name)
        this.company = company;
        this.position = position;
    }

    public String getCompany() { return company; }
    public String getPosition() { return position; }

    @Override
    public String toString() {
        return String.format("%s, %s %s", getName(), position, company);
    }
}
```

Класс `Employee` добавил два новых метода и ещё раз переопределил `toString`. Его структура в памяти и ТВМ будут выглядеть так:



И тогда следующий код выполнится именно так, как мы ожидаем:

```
Person p1 = new Person("Король Артур");
Person p2 = new Employee("Марк Цукерберг", "Facebook", "президент");

System.out.println(p1.getName());  // Король Артур
// под капотом: p1.vtable[5](p1) вызывает Person.getName(p1)

System.out.println(p2.getName());  // Марк Цукерберг
// под капотом: p2.vtable[5](p2) вызывает Person.getName(p2)

System.out.println(p1);            // Король Артур
// под капотом: p1.vtable[4](p1) вызывает Person.toString(p1)

System.out.println(p2);            // Марк Цукерберг, президент Facebook
// под капотом: p2.vtable[4](p2) вызывает Employee.toString(p2)
```