Java – Полиморфизм

Полиморфизм – способность объекта принимать множество различных форм. Наиболее распространенное использование полиморфизма в ООП происходит, когда ссылка на родительский класс используется для ссылки на объект дочернего класса. Постараемся разобраться с понятием полиморфизма в Java простыми словами, так сказать для чайников.

Любой объект в Java, который может пройти более одного теста IS-A считается полиморфным. В Java все объекты полиморфны, так как любой объект пройдёт тест IS-A для своего собственного типа и для класса Object.

Важно знать, что получить доступ к объекту можно только через ссылочную переменную. Ссылочная переменная может быть только одного типа. Будучи объявленной, тип ссылочной переменной изменить нельзя.

Ссылочную переменную можно переназначить к другим объектам, которые не объявлены как final. Тип ссылочной переменной определяет методы, которые она может вызвать на объекте.

Ссылочная переменная может обратиться к любому объекту своего объявленного типа или любому подтипу своего объявленного типа. Ссылочную переменную можно объявить как класс или тип интерфейса.

Пример 1

Рассмотрим пример наследования полиморфизм в Java.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Теперь класс Deer (Олень) считается полиморфным, так как он имеет множественное наследование. Следующие утверждения верны для примера выше:

- A Deer IS-A Animal (олень - это животное);
- A Deer IS-A Vegetarian (олень - это вегетарианец);
- A Deer IS-A Deer (олень - это олень);
- A Deer IS-A Object (олень - это объект).

Когда мы применяем факты ссылочной переменной к ссылке на объект Deer (Олень), следующие утверждения верны:

Пример 2

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

Все переменные (d, a, v, o) ссылаются к тому же объекту Deer (Олень).

Виртуальные методы

В этом разделе рассмотрим, как поведение переопределённых методов в Java позволяет воспользоваться преимуществами полиморфизма при оформлении классов.

Мы уже рассмотрели переопредение методов, где дочерний класс может переопределить метод своего «родителя». Переопределённый метод же скрыт в родительском классе и не вызван, пока дочерний класс не использует ключевое слово super во время переопределения метода.

Пример

```
/* File name : Employee.java */
public class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number) {
      System.out.println("Собираем данные о работнике");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public void mailCheck() {
      System.out.println("Отправляем чек " + this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

Теперь предположим, что мы наследуем класс Employee следующим образом:

```
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary; // Годовая заработная плата

   public Salary(String name, String address, int number, double salary) {
      super(name, address, number);
      setSalary(salary);
   }

   public void mailCheck() {
      System.out.println("Внутри mailCheck класса Salary ");
      System.out.println("Отправляем чек " + getName()
      + " с зарплатой " + salary);
   }

   public double getSalary() {
      return salary;
   }

   public void setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      }
   }

   public double computePay() {
      System.out.println("Вычисляем заработную плату для " + getName());
      return salary/52;
   }
}
```

Теперь, внимательно изучите программу и попытайтесь предугадать её вывод:

```
/* File name : VirtualDemo.java */
public class VirtualDemo {

   public static void main(String [] args) {
      Salary s = new Salary("Олег Петров", "Минск, Беларусь", 3, 3600.00);
      Employee e = new Salary("Иван Иванов", "Москва, Россия", 2, 2400.00);
      System.out.println("Вызываем mailCheck, используя ссылку Salary --");
      s.mailCheck();
      System.out.println("Вызываем mailCheck, используя ссылку Employee --");
      e.mailCheck();
   }
}
```

После запуска программы будет выдан такой результат:

```
Собираем данные о работнике
Собираем данные о работнике

Вызываем mailCheck, используя ссылку Salary --
Внутри mailCheck класса Salary
Отправляем чек Олег Петров с зарплатой 3600.0

Вызываем mailCheck, используя ссылку Employee --
Внутри mailCheck класса Salary
Отправляем чек Иван Иванов с зарплатой 2400.0
```

Итак, мы создали два объекта Salary. Один использует ссылку Salary, то есть **s**, а другой использует ссылку Employee, то есть **e**.

Во время вызова s.mailCheck(), компилятор видит mailCheck() в классе Salary во время компиляции, а JVM вызывает mailCheck() в классе Salary при запуске программы.

mailCheck() в **e** совсем другое, потому что **e** является ссылкой Employee. Когда компилятор видит e.mailCheck(), компилятор видит метод mailCheck() в классе Employee.

Во время компиляции был использован mailCheck() в Employee, чтобы проверить это утверждение. Однако во время запуска программы JVM вызывает mailCheck() в классе Salary.

Это поведение называется вызовом виртуальных методов, а эти методы называются виртуальными. Переопределённый метод вызывается во время запуска программы вне зависимости от того, какой тип данных был использован в исходном коде во время компиляции.