

Множественное наследование в Java. Противоречия и способы их решения

Александр Поточкин

Хороший программист знает не только сильные стороны языка, на котором он программирует, но и не забывает о его недостатках. В этой статье мы рассмотрим некоторые слабые стороны Java и посмотрим как эти проблемы решены в других языках.

Как известно, язык Java не поддерживает множественное наследование реализаций (классов). Это объясняется тем, что такое наследование порождает некоторые проблемы. Чаще всего указываются неоднозначности, возникающие при так называемом «ромбовидном» наследовании, когда один класс “А” является наследником двух других классов “В” и ”С”, причем и тот и другой наследуют класс “D”.

Вместо множественного наследования классов в Java введено множественное наследование интерфейсов, при котором, как утверждается, никаких проблем не возникает.

Кроме того автор языка Java Джеймс Гослинг считает, что одиночное наследование реализаций способствует правильному проектированию. На этот счет есть разные мнения, но в данной статье мы не будем заострять на них внимание, мы попытаемся определить, действительно ли язык Java свободен от недостатков возникающих при множественном наследовании.

Для начала попробуем унаследовать класс от двух интерфейсов, имеющих поля с одинаковой сигнатурой:

Множественное наследование в Java. Противоречия и способы их решения

```
interface Interfacel {
    String commonString =
        "commonString from Interfacel";
}

interface Interface2 {
    String commonString =
        "commonString from Interface2";
}

class MixedClass1 implements Interfacel,Interface2 {
}

public class Test1 {
    public static void main(String[] args) {

        System.out.println("Test N1");
        System.out.println(
            "Two interfaces have property
            with the same signature");

        MixedClass1 mixedClass = new MixedClass1();

        /* reference to commonString is ambiguous,
        both variable commonString in Interfacel
        and variable commonString in Interface2 match */

        //System.out.println(mixedClass.commonString);

        Interfacel Interface = mixedClass;
        System.out.println(Interface.commonString);
        System.out.println(((Interface2) mixedClass).commonString);
    }
}

/*
Output:
    Test N1
    Two interfaces have property with the same signature
    commonString from Interfacel
*/
```

Множественное наследование в Java. Противоречия и способы их решения

```
commonString from Interface2
*/
```

Так как непонятно из какого интерфейса унаследовано поле `mixedClass.commonString`, то попытка обратиться к нему совершенно законно вызывает ошибку компиляции (указана в комментариях). Для того чтобы устранить неоднозначность, нам необходимо преобразовать `mixedClass` к одному из родительских интерфейсов. Таким образом в этом случае Java ведет себя совершенно естественно.

Теперь проверим что произойдет, если мы вместо полей с одинаковой сигнатурой возьмем методы:

```
interface FirstInterface {
    String getInfo();
}

interface SecondInterface {
    String getInfo();
}

class MixedClass2
    implements FirstInterface, SecondInterface {

    /*
    It is impossible to have two
    implementation getInfo() function from both
    interfaces it could be support something like this:

        public String FirstInterface.getInfo() {
            return "getInfo() from FirstInterface";
        }

        public String SecondInterface.getInfo() {
            return "getInfo() from SecondInterface";
        }
    */

    public String getInfo() {
        return "MixedClass2.getInfo()"
    }
}
```

Множественное наследование в Java. Противоречия и способы их решения

```
        belongs to both interfaces";
    }
}

public class Test2 {
    public static void main(String arg[]) {
        System.out.println("Test N2");
        System.out.println("Two interfaces
have methods with the same signature");
        MixedClass2 testClass = new MixedClass2();
        System.out.println(testClass.getInfo());
    }
}

/*
Output:
    Test N2
    Two interfaces have methods with the same signature
    MixedClass2.getInfo() belongs to both interfaces
*/
```

Вот здесь происходит очень интересная вещь, совершенно очевидный прием — уточнение имени интерфейса, который прекрасно работает для полей, для функций почему-то не срабатывает! Вместо того чтобы предоставить программисту право решать, что делать в этом случае, компилятор предлагает нам только один вариант — сделать общую реализацию двух разных методов.

И это при том, что существуют случаи, когда он совершенно не подходит: если взять гипотетические интерфейсы `Display` и `File`, то между `Display.print()` и `File.print()` есть очень большая разница, и объединять только по причине совпадающих имен просто неверно.

Почему в Java выбран такой вариант, действительно непонятно, тем более что никаких сложностей в его реализации нет. Вот как это выглядит, к примеру, на новомодном C#:

```
using System;

namespace test2
{
```

Множественное наследование в Java. Противоречия и способы их решения

```
interface FirstInterface {
    String getInfo();
};
interface SecondInterface {
    String getInfo();
};

class MixedClass2 : FirstInterface, SecondInterface {
    String FirstInterface.getInfo() {
        return "getInfo() from FirstInterface";
    }
    String SecondInterface.getInfo() {
        return "getInfo() from SecondInterface";
    }
};

class Root
{
    [STAThread]
    static void Main(string[] args)
    {
        System.Console.WriteLine ( "Test N2" );
        System.Console.WriteLine ( "Two interfaces have methods with the same
signature" );
        MixedClass2 mixedClass = new MixedClass2();
        System.Console.WriteLine ( ((FirstInterface)mixedClass).getInfo ( ) );
        System.Console.WriteLine ( ((SecondInterface)mixedClass).getInfo ( ) );
    }
};
}
/* Test N2
    Two interfaces have methods with the same signature
    getInfo() from FirstInterface
    getInfo() from SecondInterface
*/
```

Ну что ж, надо признать, что в этом случае C# ведет себя корректнее.

Перейдем к самому интересному тесту: на этот раз унаследуемся от класса и реализуем интерфейс, причем и класс и интерфейс будут содержать методы с одинаковой сигнатурой.

Множественное наследование в Java. Противоречия и способы их решения

Как вы думаете скомпилируется ли приведенный ниже код ?

```
class Class3 {
    public void testMethod() {
        System.out.println(
            "testMethod () from parent class");
    }
}

interface Interface3 {
    public void testMethod ();
}

class MixedClass3 extends Class3 implements Interface3 {
}

public class Test3 {
    public static void main(String[] args) {

        MixedClass3 mixedClass = new MixedClass3();
    }
}
```

Компиляция проходит, причем даже без единого предупреждения. Таким образом у нас есть возможность создать экземпляр класса в котором не определен метод реализуемого интерфейса! Компилятор считает, что `Interface3.testMethod()` реализовывать не надо, так как в родительском классе уже есть метод с такой же сигнатурой.

Следующий пример демонстрирует эту особенность более явно:

```
class Class4 {
    public void run() {
        System.out.println("method run() from parent class");
    }
}

// Runnable is standard java
// interface that has only one method - public void
```

Множественное наследование в Java. Противоречия и способы их решения

```
run();

class MixedClass4 extends Class4 implements Runnable {
    // We can don't implement run()
    // here because it is already implemented in the parent
class
}

public class Test4 {
    public static void main(String[] args) {

        System.out.println("Test N4");
        System.out.println(
            "Class and interface have
            methods with the same signature");

        MixedClass4 mixedClass = new MixedClass4();
        Thread thread = new Thread(mixedClass);
        thread.start();
    }
}

/*
Output:
    Test N4
    Class and interface have methods with the same signature
    method run() from parent class
*/
```

Таким образом мы смогли породить и запустить новый Thread от класса, который возможно не создавался для таких целей. А так как при этом компилятор не выдал никаких предупреждений, то при реализации какого-либо интерфейса программист должен следить, не совпадает ли какой-нибудь метод из интерфейса с методом из базового класса, то есть выполнять лишнюю работу. Ведь если сигнатуры методов совпадают, то интерфейс можно случайно реализовать не полностью и тогда для пропущенных методов будет использоваться реализация из базового класса, что верно далеко не всегда.

Причем C# в этом случае ведет себя точно также.

Множественное наследование в Java. Противоречия и способы их решения

А есть ли вообще язык, свободный от проблем такого рода? И вообще, почему эти проблемы возникают? А возникают они всего лишь потому, что методы из разных классов и интерфейсов могут иметь одно и то же имя. Что же делать с такими функциями? — надо их переименовать! Позвольте представить вам язык Eiffel, язык в котором проблема одноименных функций решена совершенно очевидным способом: В Eiffel нет интерфейсов, и разрешено множественное наследование классов

```
deffered class INTERFACE1
  feature
  get_info is deferred end
end

deferred class INTERFACE2
  get_info is deferred end
end

class MIXED_CLASS
  inherit INTERFACE1 rename get_info as get_infoFromInterface1
  redefine get_infoFromInterface1
end
end
inherit INTERFACE2 rename get_info as get_infoFromInterface2
redefine get_infoFromInterface2
end

feature
  get_infoFromInterface1 is do
io.putstring ( "getInfo from INTERFACE1" )
  end

  get_infoFromInterface2 is do
    io.putstring ( "getInfo() from INTERFACE2" )
  end
end

class
  ROOT_CLASS
```

Множественное наследование в Java. Противоречия и способы их решения

```
a_obj : MIXED_CLASS
-- Creation procedure.
do
create a_obj

a_obj.get_info_from_interface1
io.put_new_line
a_obj.get_info_from_interface2

end

end -- class ROOT_CLASS

-- Output:
--
-- getInfo from INTERFACE1
-- getInfo from INTERFACE2
--
```

Обратите внимание на строку `inherit INTERFACE1 rename get_info as get_infoFromInterface1`. Здесь мы меняем имя метода `get_info` на `get_infoFromInterface1`, для метода из второго интерфейса действуем также. Таким образом, после переименования одноименных методов, экземпляр класса `MIXED_CLASS` больше не содержит проблемного метода `get_info` (попытка обратиться к нему вызовет ошибку компиляции), а содержит два метода `get_infoFromInterface1` и `get_infoFromInterface2`.

Причем, (используя синтаксис языка Java), вызов метода `a_class.get_infoFromInterface1()` получается эквивалентным вызову `((INTERFACE1) a_class).get_info()`, а `a_class.get_infoFromInterface2()` равен `((INTERFACE2) a_class).get_info()`. То есть переименование функций для программиста совершенно прозрачно, при приведении объекта к типу базового класса, все методы будут вызываться корректно и никакой путаницы не возникнет. Изящное решение! Здесь Eiffel бесспорно выигрывает и у Java и у C#, похоже Eiffel имеет только один серьезный недостаток — Паскалеподобный синтаксис :-).

Я надеюсь, что данная статья поможет начинающим (а возможно и более опытным)

Множественное наследование в Java. Противоречия и способы их решения

Java программистам разобраться с некоторыми недостатками языка Java и познакомиться с решениями, принятыми в других языках. А также позволит избежать «подводных камней» при использовании множественного наследования.

Приятного программирования.

Особая благодарность специалисту по Microsoft .Net технологиям Алексею Лапшину <http://www.star.spb.ru/~alexey/>