Лабораторная работа № 4

Тема: Динамическая диспетчеризация методов

Цель: Изучение принципов полиморфизма и динамической диспетчеризации методов

Переопределение методов служит основой для одного из наиболее эффективных принципов в Java — динамической диспетчеризация методов. Динамическая диспетчеризация методов - это механизм, с помощью которого вызов переопределенного метода разрешается во время выполнения, а не компиляции. Динамическая диспетчеризация методов важна потому, что благодаря ей полиморфизм в Java реализуется во время выполнения.

Ссылочная переменная из суперкласса может ссылаться на объект подкласса. Этот принцип используется в Java для разрешения вызовов переопределенных методов во время выполнения. Это происходит следующим образом: когда переопределенный метод вызывается по ссылке на суперкласс, нужный вариант этого метода выбирается в Java в зависимости от типа объекта, на который делается ссылка в момент вызова. Таким образом, этот выбор делается во время выполнения. По ссылке на разные типы объектов будут вызываться разные варианты переопределенного метода. Вариант переопределенного метода выбирается для выполнения в зависимости от типа объекта, на который делается ссылка, а не типа ссылочной переменной. Так, если суперкласс содержит метод, переопределяемый в подклассе, то по ссылке на разные типы объектов через ссылочную переменную из суперкласса будут выполняться разные варианты этого метода.

Пример 1

```
// Динамическая диспетчеризация методов
class A {
        void callme ( ) {
        System.out.println( " В методе callme ( ) из класса A " ) ;
        }
}

class B extends A {
// переопределение метода callme( )
        void callme( ) {
        System.out.println( " В методе callme ( ) из класса B " ) ;
        }
}
```

```
class C extends A {
// переопределение метода callme( )
        void callme( ) {
        System.out.println( " В методе callme ( ) из класса C " ) ;
        }
}

class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

В этой программе создаются один суперкласс A и два его подкласса B и C. В подклассах B и C переопределяется метод callme( ), объявляемый в классе A. В методе main( ) объявляются объекты классов A, B и C, а также переменная r ссылки на объект типа A. Затем переменной r присваивается по очереди ссылка на объект каждого из классов A, B и C , и по этой ссылке вызывается метод callme(). Выполняемый вариант метода callme() определяется исходя из типа объекта, на который делается ссылка в момент вызова.

Особое значение полиморфизма для ООП объясняется тем, что он позволяет определить в общем классе методы, которые станут общими для всех производных от него классов, а в подклассах - конкретные реализации некоторых или всех этих методов.

Переопределенные методы предоставляют еще один способ реализовать в Java принцип полиморфизма - один интерфейс, множество методов. Одним из основных условий успешного применения полиморфизма является понимание, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. Если суперкласс применяется правильно, он предоставляет все элементы, которые могут непосредственно использоваться в подклассе. В нем также определяются те методы, которые должны быть реализованы в производном классе. Это дает удобную

возможность определять в подклассе его собственные методы, сохраняя единообразие интерфейса. Таким образом, сочетая наследование с переопределенными методами , в суперклассе можно определить общую форму для методов, которые будут использоваться во всех его подклассах.

Динамический, реализуемый во время выполнения полиморфизм - один из самых эффективных механизмов объектно-ориентированной архитектуры, обеспечивающих повторное использование и надежность кода. Возможность вызывать из библиотек уже существующего кода методы для экземпляров новых классов, не прибегая к повторной компиляции и в то же время сохраняя ясность абстрактного интерфейса, является сильно действующим средством.

В приведенной ниже примере создается суперкласс Figure для хранения размеров двумерного объекта, а также определяется метод area( ) для расчета площади этого объекта. Кроме того, создаются два класса, Rectangle и Triangle, производные от класса Figure. Метод area( ) переопределяется в каждом из этих подклассов, чтобы возвращать площадь четырехугольника и треугольника соответственно.

Пример 2

```
// Using run-time polymorphism.
class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
```

```
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);

    Figure figref;

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());

    figref = f;
    System.out.println("Area is " + figref.area());
  }
}
```

Таким образом, двойной механизм - наследование и полиморфизм, во время выполнения позволяет определить единый интерфейс, используемый разнотипными, хотя и связанными вместе классами объектов. Так, если объект относится к классу, производному от класса Figure, его площадь можно рассчитать, вызвав метод area(). Интерфейс для выполнения этой операции остается неизменным независимо от вида фигуры.

Иногда суперкласс требуется определить таким образом, чтобы объявить в нем структуру заданной абстракции, не предоставляя полную реализацию каждого метода. Это означает создать суперкласс, определяющий только обобщенную форму для совместного использования всеми его подклассами, в каждом из которых могут быть добавлены

требующиеся детали. В таком классе определяется характер методов, которые должны быть реализованы в подклассах. Подобная ситуация может возникнуть, когда в суперклассе не удается полностью реализовать метод. Так в предыдущем примере в классе Figure определение метода area() служит лишь в качестве шаблона, не позволяя рассчитать и вывести площадь объекта какого-нибудь типа. В случае, если требуется убедиться, что в подклассе действительно переопределяются необходимые методы, для этой цели в Java служит абстрактный тип метода, для чего используется модификатор типа abstract. Иногда такие методы называются методами под ответственностью подкласса, поскольку в суперклассе для них никакой реализации не предусмотрено. Следовательно, эти методы должны быть переопределены в подклассе, где нельзя просто воспользоваться их вариантом, определенным в суперклассе. Для объявления абстрактного метода используется следующая общая форма:

abstract тип имя (список параметров) ;

Любой класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно указать ключевое слово abstract перед ключевым словом class в начале объявления класса. У абстрактного класса не может быть никаких объектов. Экземпляр абстрактного класса не может быть получен непосредственно с помощью оператора new. Также нельзя объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс, производный от абстрактного класса, должен реализовать все абстрактные методы из своего суперкласса или же сам быть объявлен абстрактным.

Ниже приведен простой пример класса, содержащего абстрактный метод:

Пример 3

```
// Пример демонстрации использования абстрактного класса
abstract class A {
  abstract void callme();

// конкретный метод может оставаться доступным в абстрактном классе
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
```

```
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

В этой программе объекты класса A не объявляются, так как это абстрактный класс. В абстрактном классе могут реализовываться конкретные методы, например, в классе A реализуется метод callmetoo( ). В абстрактные классы может быть включена реализация любого количества конкретных методов.

Несмотря на то что абстрактные классы не позволяют получать экземпляры объектов, их можно применять для создания ссылок на объекты, так как в Java полиморфизм во время выполнения реализован с помощью ссылок на суперкласс.

В приведенном ниже примере показано, как создать ссылку на абстрактный класс Figure, после чего делаются и используются указания на объект подкласса.

Пример 4

```
// Абстрактный класс с абстрактным методом вычисления площади
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // Это абстрактный метод
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }
```

```
  // Реализация абстрактного метода для четырехугольника
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // Реализация абстрактного метода для прямоугольного треугольника
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class DemoAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(2, 2); // недопустимое объявление Figure
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);

    Figure figref; // допустимое объявление, но объект не создается

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());
  }
}
```

Задания к работе

1. Изучите и повторите примеры 1-4 теоретической части лабораторной работы. Определите результаты работы примеров, сделайте их скрин-шот и дайте пояснения.

2. Ответьте на вопрос: Какой был бы результат в примере 1, если бы выбор делался по типу ссылочной переменной r ?
3. В одной из задач прикладной механики необходимо использовать значения определенных интегралов от *a* до *b* для различных функций *f(x)*. Предполагается, что множество используемых функций будет расширяться по мере необходимости. Разработайте абстрактный класс FunctionIntegral с тремя абстрактными методами численного интегрирования (например, методом прямоугольников, трапеций и метод Симпсона), где *a* и *b* используются в качестве входных параметров. Разработайте 3 класса, наследующие абстрактный класс FunctionIntegral и реализующие его абстрактные методы для функций своего варианта.

4. Создайте класс, демонстрирующий возможности разработанных классов, определите результаты работы примеров, сделайте их скрин-шот и дайте пояснения.

5. Подготовьте отчет.