МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Одеський національний політехнічний університет

Інститут Комп’ютерних Систем

Кафедра Інформаційних Систем та Технологій

Протокол лабораторної роботи №5

з дисципліни об’єктно-орієнтоване програмування

на тему: « РАЗРАБОТКА КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ »

Виконав студент групи

АД-201

Стицковський Н.Ю.

Прийняв

Рудніченко Н.Д.

Одеса, 2021

**Содержание**

Введение ……. 1

Теоретическая часть ….. 2

Практическая часть ……. 3

Вывод …….. 4

Литература …….. 5

ВВЕДЕНИЕ

**ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:**

* Ознакомиться с механизмом полиморфизма в ООП;
* Разобраться со статическим и динамическим связыванием;
* Научиться использовать восходящее преобразование.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Полиморфизм является третьей важной особенностью объектно-ориентированных языков, вместе с абстракцией и наследованием.

Он представляет еще одну степень отделения интерфейса от реализации, разъединения ***что*** от ***как***. Полиморфизм улучшает организацию кода и его читаемость, а также способствует созданию **расширяемых** программ, которые могут «расти» не только в процессе начальной разработки проекта, но и при добавлении новых возможностей.

Для начала, вернемся к наследованию. Самая важная особенность наследования заключается не в том, что оно позволяет новому классу использовать поля и методы базового, а в том, что наследование выражает отношение между новым и базовым классом. Это отношение можно выразить как «Новый класс **является разновидностью** базового класса», это же справедливо и для объектов этих классов. Например, класс **Student** является разновидностью **Person**, класс **Bear** является разновидностью **Animal** и т.д.

Данное отношение поддерживается языком программирования. Например, рассмотрим базовый класс **Instrument**, который представляет музыкальные инструменты, и класс **Guitar**, который представляет музыкальный инструмент гитару. Так как наследование означает, что все методы базового класса также доступны в производном классе, любое сообщение, которое мы можем отправить базовому классу, можно отправить и производному классу. Если в классе **Instrument** есть метод **play()**, то он будет присутствовать и в классе **Guitar**.

Исходя из этого, в Java такой код является корректным:

|  |
| --- |
| **class** Instrument {  **public void** play() {  ***// какой-то код*** } }  **class** Guitar **extends** Instrument {}  **public class** Main {  **public static void** main(String[] args) {  Guitar acoustic = **new** Guitar();  *tune*(acoustic); ***// <-- Как такое возможно?*** }   **public static void** tune (Instrument i) {  ***// настройка инструмента*** } } |

Обратите внимание, что метод tune() спокойно принимает тип Guitar вместо типа Instrument, хотя Java является языком строгой типизации.

Но мы говорили, что объект класса **Guitar** также является разновидностью объекта класса **Instrument**, и не существует метода, который можно вызвать в методе tune() для объектов **Instrument**, но нельзя для объектов класса **Guitar**. В методе tune() код работает для объектов **Instrument** и **объектов** любых **классов**, **производных** **от** **Instrument**.

Преобразование ссылки на объект Guitar в ссылку на объект Instrument называется **восходящим преобразованием типов** **(upcasting)**.

Восходящее преобразование всегда безопасно, так как это переход от конкретного типа к более общему. Иначе говоря, производный класс является надстройкой базового класса. Он может содержать больше методов, чем базовый класс, но **обязан** включать в себя все методы базового класса.

Рассмотрим на примере, какие преимущества дает нам восходящее преобразование. Представим себе, что у нас есть такой код:

|  |
| --- |
| **class** Instrument {  **public** String play() {  **return "метод play() класса Instrument"**;  } }  **class** Guitar **extends** Instrument {  ***// Переопределение метода play*** @Override  **public** String play() {  **return "метод play() класса Guitar"**;  } }  **public class** Main {  **public static void** main(String[] args) {  Guitar acoustic = **new** Guitar();  *tune*(acoustic);  }   **public static void** tune(Instrument i) {  i.play();  } } |

Метод **tune()** получает ссылку на объект класса **Instrument**, но мы можем передать объект любого класса, производного от **Instrument**. В методе **main()** ссылка на объект класса **Guitar** передается методу **tune()** без явных преобразований. Это нормально – интерфейс класса **Instrument** должен существовать и в классе **Guitar**, т.к. класс **Guitar** был унаследован от класса **Instrument**.

Казалось бы, зачем нам указывать, что метод **tune()** принимает ссылку на объект класса **Instrument**? Если мы хотим, чтобы метод **tune()** принимал ссылку на объект класса **Guitar**, почему просто не написать tune(Guitar i)?

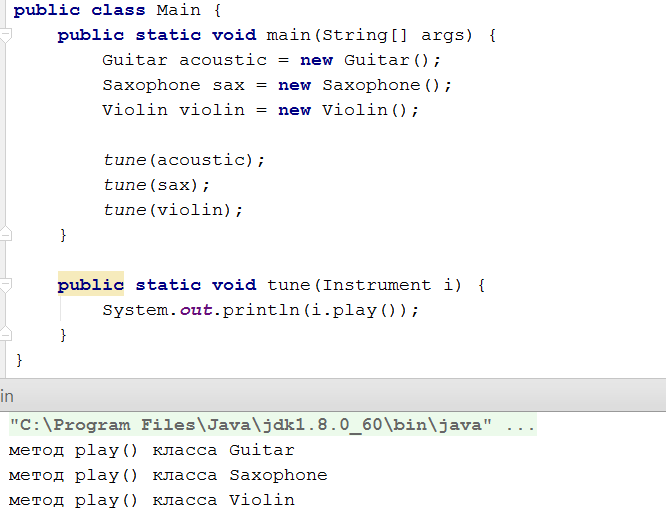
Если написать tune(Guitar i), то для объекта каждого класса, производного от Instrument придется писать свой метод **tune()**. Предположим, что в новой версии программы мы добавили классы **Saxophone** и **Violin** и в методе **tune()** в качестве входного аргумента указываем ссылку строго на объект подкласса:

|  |
| --- |
| **class** Instrument {  **public** String play() {  **return "метод play() класса Instrument"**;  } }  **class** Guitar **extends** Instrument {  ***// Переопределение метода play*** @Override  **public** String play() {  **return "метод play() класса Guitar"**;  } }  **public class** Saxophone **extends** Instrument {  @Override  **public** String play() {  **return "метод play() класса Saxophone"**;  } }  **public class** Violin **extends** Instrument{  @Override  **public** String play() {  **return "метод play() класса Violin"**;  } }  **public class** Main {  **public static void** main(String[] args) {  Guitar acoustic = **new** Guitar();  Saxophone sax = **new** Saxophone();  Violin violin = **new** Violin();   *tune*(acoustic);  *tune*(sax);  *tune*(violin);  }  **public static void** tune (Guitar i) {  i.play();  }  **public static void** tune (Saxophone i) {  i.play();  }  **public static void** tune (Violin i) {  i.play();  } } |

Программа будет работать, но у нее есть огромный недостаток: для каждого нового класса, производного от Instrument, необходимо будет писать новый метод tune(), который будет принимать объект этого производного класса. Объем программного кода увеличится, а при добавлении нового метода или нового класса придется выполнить много дополнительной работы.

Если же мы будем использовать в методе **tune()** ссылку на базовый класс, тогда необходимо будет написать всего один единственный метод **tune()**. Таким образом, код метода **tune()** будет, своего рода, обобщенным кодом, который будет выполняться не только для объектов базового класса, но и для объектов производного класса. В этом и заключается суть полиморфизма.

Однако, важно понимать, что в зависимости от разных объектов разных производных классов, метод **tune()** может работать по-разному, т.к. в каждом подклассе мы переопределяем метод **play()**.



В данном случае, методы **play()** различных подклассов возвращают различную строку, поэтому, хотя и вызывается один и тот же метод **tune()**, но результат его работы каждый раз разный, в зависимости от того – объект какого подкласса мы ему передали.

**Связывание «метод-вызов»**

Давайте еще раз взглянем на метод tune()

|  |
| --- |
| **public static void** tune(Instrument i) {  System.***out***.println(i.play()); } |

Мы уже разобрались, что в зависимости от ссылки на объект того или иного подкласса, Java вызовет тот или иной переопределенный метод **play()**. Но откуда компилятор знает – какой из методов **play()** необходимо будет вызвать, ведь в качестве входного аргумента у нас указана ссылка на объект класса **Instrument**? Ответ заключается в том, что компилятор этого не знает.

Присоединение вызова метода к телу метода называется **связыванием**. Если связывание производится перед запуском программы (на этапе компиляции или компоновки), оно называется ранним (early) или статическим (static) связыванием (binding).

Неоднозначность в работе метода **tune()** связана именно с ранним связыванием: компилятор не может знать заранее, какой вариант метода **play()** нужно будет вызвать, когда у него есть только ссылка на объект класса **Instrument**.

Проблема решается благодаря **позднему связыванию**, то есть связыванию, проводимому во время выполнения программы, в зависимости от типа объекта. Позднее (late) связывание (binding) называют также динамическим (dynamic) или связыванием на стадии выполнения (runtime binding).

В Java существует механизм для фактического определения типа объекта во время работы программы для вызова подходящего метода. Иначе говоря, компилятор не знает типа объекта, но механизм вызова методов определяет фактический тип объекта и вызывает соответствующее тело метода.

Для всех методов Java используется механизм позднего связывания, если только метод не был объявлен как **final** (приватные методы являются final по умолчанию).

Итак, подведем итоги:

1) Статическое связывание в Java происходит на этапе компиляции, тогда как динамическое связывание происходит во время выполнения программы (в runtime (рантайме));

2) для **private**, **final** и **static** методов, а также для всех полей используется статическое связывание, тогда как для остальных методов (такие методы в некоторых языках программирования называются виртуальными (virtual)) используется динамическое связывание;

3) в статическом связывании используется тип ссылки, тогда как в динамическом связывании используется фактический тип объекта;

4) перегруженные методы используют статическое связывание, тогда как переопределенные методы используют статическое связывание.

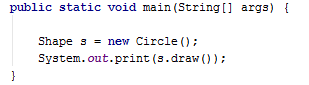
**Рассмотрим популярный пример с геометрическими фигурами.**

Создадим базовый класс **Shape** и различные производные классы: **Circle**, **Square**, **Triangle**. Восходящее преобразование имеет место даже в команде:

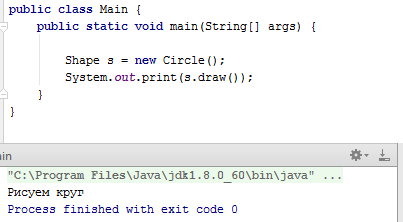


Здесь создается объект **Circle**, и полученная ссылка немедленно присваивается переменный типа **Shape**. На первый взгляд это ошибка, мы не можем присвоить ссылочной переменной одного типа ссылку на объект другого типа, но, на самом деле, все правильно, потому что тип **Circle** (окружность) является типом **Shape** (фигура) посредством наследования.

Если мы вызовем метод **draw()**



то можно подумать, что вызывается метод **draw()** из класса **Shape**, раз у нас ссылочная переменная типа **Shape**. Но на самом деле будет вызван правильный метод **Circle.draw()**, так как в программе используется позднее связывание (полиморфизм).



Рассмотрим код классов фигур.

|  |
| --- |
| **public class** Shape {  **public** String draw(){  **return null**;  };  **public** String erase(){  **return null**;  }; }  **public class** Circle **extends** Shape {  @Override  **public** String draw() {  **return "Рисуем круг"**;  }  @Override  **public** String erase() {  **return "Стираем круг"**;  } }  **public class** Triangle **extends** Shape {  @Override  **public** String draw() {  **return "Рисуем треугольник"**;  }  @Override  **public** String erase() {  **return "Стираем треугольник"**;  } }  **public class** Square **extends** Shape {  @Override  **public** String draw() {  **return "Рисуем квадрат"**;  }  @Override  **public** String erase() {  **return "Стираем квадрат"**;  } }  **public class** Main {  **public static void** main(String[] args) {   Shape s = **new** Circle();  System.***out***.println(s.draw());   Shape s2 = **new** Triangle();  System.***out***.println(s2.draw());   Shape s3 = **new** Square();  System.***out***.println(s3.draw());  } }  \*\*\*\* CONSOLE \*\*\*\*  "C:\Program Files\Java\jdk1.8.0\_60\bin\java" ...  Рисуем круг  Рисуем треугольник  Рисуем квадрат  Process finished with exit code 0 |

Базовый класс **Shape** устанавливает для всех классов, производных от S**h**ape, общий интерфейс – набор публичных методов (действий, которые может совершить над объектом). То есть любую фигуру можно нарисовать (**draw()**) и стереть (**erase()**). Производные классы переопределяют этот набор методов, чтобы реализовать свое уникальное поведение для этой фигуры.

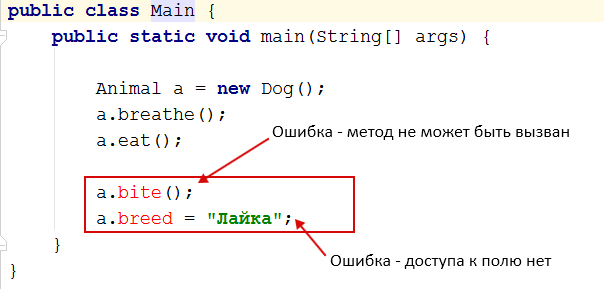
Благодаря полиморфизму вы можете добавлять сколько угодно новых типов, внося в программу минимальные изменения или не внося изменений вовсе. В хорошо спроектированной программе, большая часть методов (или даже все методы) переопределяют интерфейс базового класса. Такая программа является **расширяемой**, поскольку в нее можно добавлять дополнительную функциональность, создавая новые типы данных от общего базового класса.

**Потеря информации при восходящем преобразовании**

Важно понимать, что если вы используете восходящее преобразование, то вы теряете информацию о полях и методах, которых нет в базовом классе. Рассмотрим простой пример

|  |  |
| --- | --- |
|  |  |

Попробуем вызвать различные методы и установить поля



Мы видим большой недостаток восходящего преобразования – мы не можем использовать поля и вызывать методы, которые есть в производном классе Dog. Вы можете вызывать только те методы, которые есть в базовом классе. Это же касается и public-полей.

А что, если после восходящего преобразования, нам необходимо восстановить утраченный доступ к полям и методам производного класса? Для этого можно использовать такой прием, который называется **нисходящее преобразование** (downcasting, «даункастинг»).

Восходящее преобразование абсолютно безопасно: базовый класс не может иметь «бОльший» интерфейс, чем производный класс, и поэтому любой вызов метода гарантированно выполнится. Но при использовании нисходящего преобразования, мы не знаем достоверно, что, например, фигура в действительности является окружностью. С такой же вероятностью она может оказаться треугольником, прямоугольником или другим производным типом.

В Java существует специальный механизм, который называется динамическим определением типов (run-time type identification, RTTI). При попытке сделать нисходящее преобразование, этот механизм позволяет пройти проверку на фактическое соответствие типа. Если типы не совпадают, происходит исключение ClassCastException и программа будет закрыта (программисты в таком случае говорят, что программа «вылетела» или «покрашилась»).

Для того, чтобы избежать ошибок при нисходящем преобразовании, используйте ключевое слово **instanceof**, которое позволяет узнать фактический тип объекта, который скрывается за ссылочной переменной типа базового класса.

|  |
| --- |
| **public static void** main(String[] args) {   Animal a = **new** Dog(); *// Восходящее преобразование* a.bite(); *// <-- Ошибка, мы теперь не можем  // вызвать метод, которого нет в классе Animal   // Сделать обратную операцию нельзя* Dog dog1 = a; *// <-- Ошибка, нельзя приравнять   // Делаем нисходящее преобразование* Dog dog2 = (Dog) a; *// <--- Это называется нисходящее преобразование* dog2.bite(); *// <--- Теперь нам доступны методы класса Dog   // Но мы можем сделать ошибку или просто не знать - какой фактический тип  // скрывается за ссылочной переменной* Animal a2 = **new** Dog();  Cat cat = (Cat) a2; *// <-- Программа скомпилируется, т.к. компилятор* cat.beCute(); *// не знает, куда будет ссылаться a2. Но если вы запустите программу,  // то она закроется с ошибкой, т.к. мы ошиблись при нисходящем преобразовании   // Разберем такой пример* Animal aa;  **int** i = **new** Random().nextInt(2);  **if** (i == 0) {  aa = **new** Dog();  } **else** {  aa = **new** Cat();  }  *// Мы не знаем, на объект какогт типа будет ссылаться aa  // Поэтому мы используем ключевое слово instanceof  // которое позволяет нам узнать фактический тип объекта* **if** (aa **instanceof** Dog) {  Dog dog = (Dog) aa;  } **else if** (aa **instanceof** Cat) {  Cat cat2 = (Cat) aa;  }  *// Всегда делайте такую проверку, чтобы ваша программа  // не вылетала с ошибками* |



|  |
| --- |
| *// Разберем такой пример* Animal aa;  **int** i = **new** Random().nextInt(2);  **if** (i == 0) {  aa = **new** Dog();  } **else** {  aa = **new** Cat();  }  *// Мы не знаем, на объект какого типа будет ссылаться aa  // Поэтому мы используем ключевое слово instanceof  // которое позволяет нам узнать фактический тип объекта* **if** (aa **instanceof** Dog) {  Dog dog = (Dog) aa;  } **else if** (aa **instanceof** Cat) {  Cat cat2 = (Cat) aa;  }  *// Всегда делайте такую проверку, чтобы ваша программа  // не вылетала с ошибками* |

# ПРАКТИЧЕСКАЯ ЧАСТЬ

**ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ:**

1. Создайте класс Person, который должен содержать следующие поля и геттеры\сеттеры для этих полей:

* Фамилия;
* Имя;
* Возраст.

Создайте метод printInfo(), который бы возвращал строку следующего формата:

|  |
| --- |
| Человек <фамилия> <имя>, возраст: <возраст> |

2. Создайте класс Student, который должен наследоваться от класса Person. Добавьте дополнительные поля, и геттеры\сеттеры для этих полей:

* Группа;
* Номер студенческого билета.

Переопределите метод printInfo(), который должен выводить строку следующего формата:

|  |
| --- |
| Студент группы <группа> <фамилия> <имя>, возраст: <возраст>. Номер студенческого билета: <номер> |

3. Создайте класс Lecturer, который должен наследоваться от класса Person. Добавьте дополнительные поля, и геттеры\сеттеры для этих полей:

* Кафедра;
* Зарплата.

Переопределите метод printInfo(), который должен выводить строку следующего формата:

|  |
| --- |
| Преподаватель кафедры <кафедра> <фамилия> <имя>, возраст: <возраст>. Зарплата: <зарплата> |

4. Используя восходящее преобразование, создайте в классе Main несколько объектов классов Student и Lecturer, после чего создайте массив, который бы мог включать объекты классов Person, Student, Lecturer. Заполните массив объектами этих классов.

5. Используя цикл, обратитесь к элементам массива и выведите в консоль, с помощью метода printInfo(), информацию от каждого объекта.

Код программы:

ackage lab.pkg5.stytskovskyi;

/\*\*

\*

\* @author USER

\*/

public class Lab5Stytskovskyi {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args)

{

// TODO code application logic here

Person a = new Student("Mathew", "Callahan", 20, "AL", 189);

Person b = new Student("Joey", "Rottham", 19, "AD", 169);

Person c = new Lecturer("Alexander", "Boskey", 30, "Programming", 14000);

Person[] array = { a, b , c };

for (int i = 0; i < array.length; i++)

array[i].printInfo();

}

}

public class Lecturer extends Person

{

private String faculty;

private int salary;

public Lecturer(String n, String l, int a, String f, int s)

{

super(n, l, a);

setFaculty(f);

setSalary(s);

}

@Override

public void printInfo()

{

System.out.println("Преподаватель кафедры " + faculty + " " + lastName + " " + name

+ ", возраст: " + age + ". Зарплата: " + salary);

}

public void setFaculty(String f) { faculty = f; }

public String getFaculty() { return faculty; }

public void setSalary(int s) { salary = s; }

public int getSalary() { return salary; }

}

public class Person

{

protected String name, lastName;

protected int age;

public Person(String n, String l, int a)

{

setName(n);

setLastName(l);

setAge(a);

}

public void printInfo() // В джаве нет cout для того чтобы его оверлоуднуть, да?

{

System.out.println("Человек " + name + " " + lastName + ", возраст: " + age);

}

final public void setName(String n) { name = n; }

public String getName() { return name; }

final public void setLastName(String l) { lastName = l; }

public String getLastName() { return lastName; }

final public void setAge(int a) { age = a; }

public int getAge() { return age; }

}

public class Student extends Person

{

private String group;

private int number;

public Student (String n, String l, int a, String g, int num)

{

super(n,l,a);

setGtoup(g);

setNumber(num);

}

@Override

public void printInfo()

{

System.out.println("Студент группы " + group + " " + lastName + " " + name + ", возраст: "

+ age + "\nНомер студенческого билета: " + number);

}

final public void setGtoup(String g) { group = g; }

public String getGroup() { return group; }

final public void setNumber(int n) { number = n; }

public int getNumber() { return number; }

}

Результат работы программы:

