Лекція 3

ООП модель Java

Методи та члени класу

Реалізація інкапсуляції

Конструктори класів. Порядок ініціалізації полів

Перевантаження методів

Робота з посиланнями на обєкти

Ключове слово this

Статичні методи та поля

* Наслідування та поліморфізм
* Ключове слово super
* Абстрактний клас, інтерфейси
* Вкладені класи
* Pattern Immutable
* Модифікатор final
* Класи-огортки
* Класи-огортки примітивних типів
* Явне та неявне приведення типів, autoboxing
* Прості оболонки для типів. Number. Double та Float. Нескінченість та NaN. Integer і Long. Реалізація Unsigned. BigDecimal, BigNumber.
* Перерахування (enum)

Методи та члени класу

Java є обєктно-орієнтованою мовою, а це означає що ми можемо (та навіть повинні) створювати класи.

Клас містить описання змінних і констант, що характеризують обєкт. Вони називаються *полями класу* (class fields). Процедури, які описують поведінку обєкта, називаються *методами класу* (class methods). Всередині класу можна описати і *вкладені класи* (nested classes) і *вкладені інтерфейси(interfaces).* Поля, методи і вкладені класи першого рівня являються *членами класу* (class members). Різні школи обєктно-орієнтованного програмування пропонують різні терміни, ми використовуємо термінологію, прийняту в технології Java.

Ось зразок опису автомобіля:

**class Automobile{**

**int maxVelocity; // Поле, що містить найбільшу швидкість автомобіля**

**int speed; // Поле, що містить поточну швидкість автомобіля**

**int weight; // Поле, що містить вагу автомобіля**

**// Інші поля...**

**void moveTo(int x, int у){ // Метод, що моделює переміщення автомобіля. Параметри х і у — не поля**

**int а = 1; // Локальна змінна — не поле**

**// Тіло метода. Тут описується закон переміщення автомобіля в точку (х, у)**

**}**

**// Інші методи. . .**

**}**

Примітка: В Java немає вкладених процедур і функцій, в тілі метода не можна описати інший метод.

Після того, як описання класу закінчено, можна створювати конкретні обєкти, eкземпляри(instances) описаного класу. Створення экземплярів відбувається в три етапи, подібно описанню масивів. Спочатку оголошуютья посилання на обєкти: записується імя класу, і через пробіл перечисляються экземпляри класу, точніше, посилання на них.

**Automobile Iada2110, fordScorpio, oka;**

Потім операцією new визначаються самі обєкти, під них виділяється оперативна память, посилання отримує адресу цієї частини памяті в якості свого значення.

**lada2110 = new Automobile();**

**fordScorpio = new Automobile();**

**oka = new Automobile();**

На третьому этапі відбувається ініціалізація обєктів, задаються початкові значення. Цей етап, як правило, суміщується з другим, якраз для цього в операції new повторяється імя класа з дужками Automobile (). Це так званий *конструктор* (constructor) класу.

Оскільки імена полів, методів і вкладених класів у всіх об’єктах одинакові, вони задані в описанні класу, їх треба уточнювати імям посилання на обєкт:

**lada2110.maxVelocity = 150;**

**fordScorpio.maxVelocity = 180;**

**oka.maxVelocity = 350;// Чому б і ні?**

**oka.moveTo(35, 120);**

Нагадаємо, що текстовий рядок в лапках вважається в Java як обєкт класу String. Тому можна написати

**int strlen = "Це обєкт класу String".length();**

Обєкт "рядок" викликає метод length(), один з методів свого класу String, що підраховує кількість символів у рядку. В результаті одержимо значення strlen, рівне 21.

В багатьох ситуаціях створюють декілька моделів з різним степенем деталізації. Скажімо, для конструювання пальто і шуби потрібна менш точна модель контурів людського тіла і його рухів, а для конструювання фрака або вечірнього плаття — уже набагато точніша. При цьому більш точна модель, з меншим степенем абстракції, буде використовувати уже наявні методи менш точної моделі.

Для створення класу використовується ключове слово class. Ми вже бачили, що без створення класу компілятор у байт-код не може створити відповідний клас та запустити програму на мові Java.

Отже опис класу буде віглядати наступним чином:

class <Імя класу>{

// Опис полів класу

<Type0\_1> field1;

<Type0\_2> field2;

\*\*\*

// Опис методів класу

<Type1\_1> method1( <args1>) { \*\*\* };

<Type1\_2> method1( <args2>) {\*\*\*};

// Клас може включати й головну функцію

public static void main(String[] args){

// код головної функції

}

};

Примітка 1: Крім типів полів та методів класу там часто потрібно включати також їх специфікатори які будуть описані пізніше.

Примітка 2: Методи класу дозволяють нам створювати функції таким самим чином як і в процедурному програмуванні.

Приклад:

**Операція new**

Пора детальніше описати операцію з одним операндом, що позначається словом new. Вона застосовується для виділення памяті масивам і обєктам.

В першому випадку в якості операнда вказується тип елементів масиву і кількість його елементів у квадратних дужках, наприклад:

**double a[] = new double[100];**

В другому випадку операндом служить конструктор класу. Якщо конструктору в класі немає, то викликається конструктор за замовченням.

Числові поля класу ініціалізуються нульовими значеннями, логічні поля — значеннями false, посилання — значеннями null.

Результатом операції new буде посилання на створений обєкт. Це посилання може бути присвоєне змінній типу посилання на даний тип:

**Dog k9 = new Dog () ;**

але може використовуватися і безпосередньо

**new Dog().voice();**

Тут після створення безіменного обєкта зразу виконується його метод voice(). Такий дивний запис зустрічається в програмах, написаних на Java, на кожному кроці

Конструктори (Constructors)

Ви уже звернули увагу на те, що в операції new, що визначає екземпляри класу, повторюється імя клас з дужками. Це схоже на звертання до методу, але що це за "метод", імя котрого повністю співпадає з іменем класу?

Такий "метод" зветься *конструктором класу* (class constructor). Його своєрідність заключається не лише в імені. Перечислимо особливості конструктора.

* Конструктор є в будь-якому класі. Навіть якщо ви його не написали, компілятор Java сам створить *конструктор за замовченням* (default constructor), котрий, між іншим, порожній, він не робить нічого, кріме виклику конструктору суперкласу.
* Конструктор виконується автоматично при створенні екземпляру класу, після розподілу памяті і обнулення полів, але до початку використання створюваного обєкта.
* Конструктор не повертає жодного значення. Тому в його описі не пишеться навіть слово void, але можна задати один із трьох модифікаторів public, protected або private.
* Конструктор не є методом, він навіть не вважається членом класу. Тому його не можна наслідувати або перевизначати в підкласі.
* Тіло конструктора може починатися:
  + з виклику одного із конструкторів суперкласу, для цього записується слово super() з параметрами в дужках, якщо вони потрібні;
  + з виклику другого конструктора того ж класа, для цього записується слово this() з параметрами в дужках, якщо вони потрібні.
* Якщо ж super() на початку конструктора не вказаний, то спочатку виконується конструктор суперкласу без аргументів, потім відбувається ініціалізація полів значеннями, вказаними при їх оголошенні, а вже потім те, що записано в конструкторі.

У всьому останньому конструктор можна вважати звичайним методом, в ньому дозволяється записувати будь-які оператори, навіть оператор return, але тільки порожній (без значення).

В класі може бути декілька конструкторів. Оскільки в них одне і те ж імя, що співпадає з іменем класу, то вони повинні відрізнятися типом і/або кількістю параметрів.

В наших прикладах ми не розглядали конструктори класів, тому що при створенні екземплярів наших класів викликався конструктор класу Оbject.

Перевантаження

Перевантаження методу - це властивість ООП мови, яка дозволяє класу мати більше одного методу, що має однакове ім'я, якщо їх списки аргументів різні. Це схоже на перевантаження конструкторів в Java, що дозволяє класу мати більше одного конструктора, що має різні списки аргументів.

Список аргументів методу, це означає параметри, які має метод: Наприклад, список аргументів методу add (int a, int b), що має два параметри, відрізняється від списку аргументів методу add (int a, int b, int c), що має три параметри.

Три способи перевантажити метод

Щоб перевантажити метод, списки аргументів методів повинні відрізнятися в будь-якому з цих описів:

1. Кількість параметрів.

add(int, int)

add(int, int, int)

2. Типи параметрів.  
add(int, int)

add(int, float)

3. Послідовність параметрів

add(int, float)

add(float, int)

Некоректний випадок перевантаження методу:

Список аргументів, не включає тип повернення методу, наприклад, якщо два способи мають однакову назву, однакові параметри і мають різний тип повернення, то це не є коректним прикладом перевантаження методу. Це призведе до помилки компіляції.

int add(int, int)

float add(int, int)

Метод перевантаження є прикладом статичного поліморфізму.

Примітки:

1. Статичний поліморфізм також відомий як зв'язування часу збирання або раннє зв'язування.

2. Статичне зв'язування відбувається під час компіляції. Перевантаження методу - приклад статичного зв’язування, коли прив'язка виклику методу до його визначення відбувається в час компіляції.

Методи Приклади перевантаження

Перевантаження методу проводиться шляхом оголошення одного методу з різними параметрами. Параметри повинні бути різними в будь-якому з цих: число, послідовність або типи параметрів (або аргументів). Розглянемо приклади кожного з цих випадків.

Список аргументів також відомий як список параметрів

Приклад 1: Перевантаження - Різна кількість параметрів у списку аргументів

class DisplayOverloading

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(char c, int num)

{

System.out.println(c + " "+num);

}

}

class Sample

{

public static void main(String args[])

{

DisplayOverloading obj = new DisplayOverloading();

obj.disp('a');

obj.disp('a',10);

}

}

Output:

a

a 10

Приклад: Перевантаження по типу аргументу

class DisplayOverloading2

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(int c)

{

System.out.println(c );

}

}

class Sample2

{

public static void main(String args[])

{

DisplayOverloading2 obj = new DisplayOverloading2();

obj.disp('a');

obj.disp(5);

}

}

результат:

a

5

Приклад: перевнтаження по порядку параметрів

class DisplayOverloading3

{

public void disp(char c, int num)

{

System.out.println("I’m the first definition of method disp");

}

public void disp(int num, char c)

{

System.out.println("I’m the second definition of method disp" );

}

}

class Sample3

{

public static void main(String args[])

{

DisplayOverloading3 obj = new DisplayOverloading3();

obj.disp('x', 51 );

obj.disp(52, 'y');

}

}

Output:

I’m the first definition of method disp

I’m the second definition of method disp

Перевантаження методу та просування типу

Коли тип даних меншого розміру просувається до типу даних більшого розміру, це називається просування типу, наприклад: тип даних byte може бути просунутий до short, short може бути підвищений до int, long, double та ін.

Дуже важливо зрозуміти просування типу, інакше ви подумаєте, що програма видасть помилку компіляції, але насправді програма буде працювати нормально через просування типу.

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* I am passing float value as a second argument but

\* it got promoted to the type double, because there

\* wasn't any method having arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Результат:

Method A

Як вибачите було передано дійсне( float) значення коли викликаний метод disp() але він був просунутий до типу double оскільки метод був перевизначений лише для типів (int, float)

Але такий тип просування не завжди трапляється, дивимося наступний приклад:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

void disp(int a, float b){

System.out.println("Method C");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* This time promotion won't happen as there is

\* a method with arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Результат:

Method C

Такий резльтат відбувся без просування, тому що відповідний до типу перезавантажений метод вже існував  
Таблиця типів просувань (Type Promotion table):   
byte → short → int → long

short → int → long

int → long → float → double

float → double

long → float → double

коректні/некоректні випадки перевантаження

Випадок 1:

int mymethod(int a, int b, float c)

int mymethod(int var1, int var2, float var3)

Реультат: Ошибка підчас компіляції. Compile time error. Списки аргументів однакові.

Випадок 2:

int mymethod(int a, int b)

int mymethod(float var1, float var2)

Результат. Все коректно

Випадок 3:

int mymethod(int a, int b)

int mymethod(int num)

Результат. Все коректно

Випадок 4:

float mymethod(int a, float b)

float mymethod(float var1, int var2)

Результат. Все коректно

Випадок 5:

int mymethod(int a, int b)

float mymethod(int var1, int var2)

Реультат: Ошибка підчас компіляції. Compile time error. Списки аргументів однакові. Тип результату неважливий.

Робота з посиланнями на обєкти

Об'єкти класів, як і дані примітивних типів можуть передаватися в методи. Однак в даному випадку є одна особливість - при передачі об'єктів в якості значення передається копія посилання на область в пам'яті, де міститься цей об'єкт. Розглянемо невеликий приклад. Нехай у нас є наступний клас Person:

public class Program{

public static void main(String[] args) {

Person kate = new Person("Kate");

System.out.println(kate.getName()); // Kate

changeName(kate);

System.out.println(kate.getName()); // Alice

}

static void changeName(Person p){

p.setName("Alice");

}

}

class Person{

private String name;

Person(String name){

this.name = name;

}

public void setName(String name){

this.name = name;

}

public String getName(){

return this.name;

}

}

Тут в метод changeName передається об'єкт Person, у якого змінюється ім'я. Так як в метод буде передаватися копія посилання на область пам'яті, в якій знаходиться об'єкт Person, то змінна kate і параметр p методу changeName вказуватимуть на один і той же об'єкт в пам'яті. Тому після виконання методу у об'єкта kate, який передається в метод, буде змінено ім'я з "Kate" на "Alice".

Від цього випадку слід відрізняти інший випадок:

public class Program{

public static void main(String[] args) {

Person kate = new Person("Kate");

System.out.println(kate.getName()); // Kate

changePerson(kate);

System.out.println(kate.getName()); // Kate - змін не відбулося

// kate зберігає старе посилання

}

static void changePerson(Person p){

p = new Person("Alice"); // p вказує на новий обєкт

p.setName("Ann");

}

static void changeName(Person p){

p.setName("Alice");

}

}

class Person{

private String name;

Person(String name){

this.name = name;

}

public void setName(String name){

this.name = name;

}

public String getName(){

return this.name;

}

}

У метод changePerson також передається копія посилання на об'єкт Person. Однак в самому методі ми змінюємо не окремі значення об'єкта, а перетворюємо об'єкт за допомогою конструктора і оператора new. В результаті в пам'яті буде виділено нове місце для нового об'єкта Person, і посилання на цей об'єкт буде прищепи параметру p:

static void changePerson(Person p){

p = new Person("Alice"); // p створює новий обєкт

p.setName("Ann"); // зміна нового обєкту

}

Тобто після створення нового об'єкту Person параметр p і змінна kate в методі main зберігатимуть посилання на різні об'єкти. Змінна kate, яка передавалася в метод, продовжить зберігати посилання на старий об'єкт в пам'яті. Тому її значення не змінюється.

Перевантаження конструктору

Тут ми створюємо два обєкти класу StudentData. Один має конструктор за замовченням, а інший використовує парметризований конструктор. Оба конструктори мають різний ініціалізаціонний код. Взагалі кажучі можна створювати різну кількість перевантажених контрукторів з зовсім різними кодами.  
StudentData.java

class StudentData

{

private int stuID;

private String stuName;

private int stuAge;

StudentData()

{

//Default constructor

stuID = 100;

stuName = "New Student";

stuAge = 18;

}

StudentData(int num1, String str, int num2)

{

//Parameterized constructor

stuID = num1;

stuName = str;

stuAge = num2;

}

//Getter and setter methods

public int getStuID() {

return stuID;

}

public void setStuID(int stuID) {

this.stuID = stuID;

}

public String getStuName() {

return stuName;

}

public void setStuName(String stuName) {

this.stuName = stuName;

}

public int getStuAge() {

return stuAge;

}

public void setStuAge(int stuAge) {

this.stuAge = stuAge;

}

public static void main(String args[])

{

//This object creation would call the default constructor

StudentData myobj = new StudentData();

System.out.println("Student Name is: "+myobj.getStuName());

System.out.println("Student Age is: "+myobj.getStuAge());

System.out.println("Student ID is: "+myobj.getStuID());

/\*This object creation would call the parameterized

\* constructor StudentData(int, String, int)\*/

StudentData myobj2 = new StudentData(555, "Chaitanya", 25);

System.out.println("Student Name is: "+myobj2.getStuName());

System.out.println("Student Age is: "+myobj2.getStuAge());

System.out.println("Student ID is: "+myobj2.getStuID());

}

}

Output:

Student Name is: New Student

Student Age is: 18

Student ID is: 100

Student Name is: Chaitanya

Student Age is: 25

Student ID is: 555

Ключове слово this

this - ключове слово Java. Це слово може бути вокористане всередині методу або конструктору класу. Ц слово служить для посилання на поточний обєкт класу, тобто на клас чий метод або конструктор викликаний. За допомогою цього слова можна отримати доступ до інших методів або полів цього обєкту.

this з полем класу(Instance Variable)

this може бути корисним ри використанні приховування змінних (Variable Hiding). Ми не можемо сторити два екземпляри змінних з тим самим імям. Однак, ми можемо сторити одну лоокальну змінну та одне поле класу з тим самим іменем. Це зветься приховуаанням змінної Variable Hiding.

Приклад приховування змінної

class JBT {

int variable = 5;

public static void main(String args[]) {

JBT obj = new JBT();

obj.method(20);

obj.method();

}

void method(int variable) {

variable = 10;

System.out.println("Value of variable :" + variable);

}

void method() {

int variable = 40;

System.out.println("Value of variable :" + variable);

}

}

Результат роботи:

1. Value of variable :10
2. Value of variable :40

Як можна побачити в даному прикладі полек класу є прихованою змінної та її значення не виводиться. Для того, щоб отримати доступ до змінної можна використати ключове слово this та мати одночасний доступ і до поля і до локальної змінної.

Приклад використання this

class JBT {

int variable = 5;

public static void main(String args[]) {

JBT obj = new JBT();

obj.method(20);

obj.method();

}

void method(int variable) {

variable = 10;

System.out.println("Value of Instance variable :" + this.variable);

System.out.println("Value of Local variable :" + variable);

}

void method() {

int variable = 40;

System.out.println("Value of Instance variable :" + this.variable);

System.out.println("Value of Local variable :" + variable);

}

}

Результат роботи програми:

1. Value of Instance variable :5
2. Value of Local variable :10
3. Value of Instance variable :5
4. Value of Local variable :40

this у конструкторі

“this” може бути використаний всередині конструкторі для того щоб використати перегружений інший конструктор того ж класу. Це зветься виклик безпосереднього конструктору (Explicit Constructor Invocation). Це може бути корисним коли клас використовує два перегружених конструктори, наприклад, один з аргументом, а інший без. Тоді “this” може бути використане для виклику конструктору з аргументом всередині конструктору з параметром. Це потрібно коли конструктор не може бути викликаний безпосередньо.

Приклад

class JBT {

JBT() {

this("JBT");

System.out.println("Inside Constructor without parameter");

}

JBT(String str) {

System.out.println("Inside Constructor with String parameter as " + str);

}

public static void main(String[] args) {

JBT obj = new JBT();

}

}

Результат:

1. Inside Constructor with String parameter as JBT
2. Inside Constructor without parameter

Примітки:

* this може бути лише першим оператором в конструкторі.
* Конструктор може використати ключові слова this або super але не обидва одночасно.

Роль this () в перевантаженні конструктору

public class OverloadingExample2

{

private int rollNum;

OverloadingExample2()

{

rollNum =100;

}

OverloadingExample2(int rnum)

{

this();

/\*this() is used for calling the default

\* constructor from parameterized constructor.

\* It should always be the first statement

\* inside constructor body.

\*/

rollNum = rollNum+ rnum;

}

public int getRollNum() {

return rollNum;

}

public void setRollNum(int rollNum) {

this.rollNum = rollNum;

}

public static void main(String args[])

{

OverloadingExample2 obj = new OverloadingExample2(12);

System.out.println(obj.getRollNum());

}

}

результат:

112

As you can see in the above program that we called the parameterized constructor during object creation. Since we have this() placed in parameterized constructor, the default constructor got invoked from it and initialized the variable rollNum.

Конструктор за замовченням

Важливим зауваженням є те, що якщо ми не створюємо жодного контруктору, то компілятор java сторює конструктор за замовченням, якщо ж стврюємо хоч один, він не створює коструктору без параметрів.

public class Demo

{

private int rollNum;

//We are not defining a no-arg constructor here

Demo(int rnum)

{

rollNum = rollNum+ rnum;

}

//Getter and Setter methods

public static void main(String args[])

{

//This statement would invoke no-arg constructor

Demo obj = new Demo();

}

}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation

problem:The constructor Demo() is undefined

this з методом

this можна використати для доступу до методу класу

Приклад

class JBT {

public static void main(String[] args) {

JBT obj = new JBT();

obj.methodTwo();

}

void methodOne(){

System.out.println("Inside Method ONE");

}

void methodTwo(){

System.out.println("Inside Method TWO");

this.methodOne();// same as calling methodOne()

}

}

Результат:

1. Inside Method TWO
2. Inside Method ONE

Використання this як параметру методу

public class JBTThisAsParameter {

public static void main(String[] args) {

JBT1 obj = new JBT1();

obj.i = 10;

obj.method();

}

}

class JBT1 extends JBTThisAsParameter {

int i;

void method() {

method1(this);

}

void method1(JBT1 t) {

System.out.println(t.i);

}

}

**Статичні члени класу**

Різні екземпляри одного класу мають повністю незалежні один від одного поля, що приймають різні значення. Зміна поля в одному екземплярі ніяк не впливає на те ж поле в другому екземплярі. В кожному екземплярі для таких полів виділяється своя комірка памяті. Тому такі поля називаються змінними е*кземпляра класа* (instance variables) або змінними *обєкта.*

Інколи треба визначити поле, спільне для всього класу, зміна котрого в одному екземплярі потягне зміну того ж поля у всіх екземплярах. Наприклад, ми хочемо в класі Automobile відзначити порядковий заводський номер автомобіля. Такі поля називаються *змінними класу* (class variables). Для змінних класу виділяється тільки одна комірка памяті, спільна для всіх екземплярів. Змінні класу створюються в Java модифікатором static. В прикладі 3.3 ми записуємо цей модифікатор при визначенні змінної number.

**Лістинг 3.3. Статична змінна**

**class Automobile {**

**private static int number;**

**Automobile(){**

**number++;**

**System.out.println("From Automobile constructor:"+ " number = "+number);**

**}**

**}**

**public class AutomobileTest{**

**public static void main(String[] args){**

**Automobile lada2105 = new Automobile(),**

**fordScorpio = new Automobile(),**

**oka = new Automobile();**

**}**

**}**

Цікаво, що до статичних змінних можна звертатися з іменем класа, Automobile.number, а не тільки з іменем eкземпляра, lada2105.number, причому це можна робити, навіть якщо не створено жодного екземпляра класу.

Для роботи з такими *статичними змінними* звичайно створюються *статичні методи,* помічені модифікатором static. Для методів слово static має зовсім інший зміст. Виконуюча система Java завжди створює в памяті тільки одну копію машинного коду метода, що розділяється всіма екземплярами, незалежно від того, статичний цей метод чи ні.

Основна особливість статичних методів — вони виконуються зразу у всіх екземплярах класу. Більше того, вони можуть виконуватися, навіть якщо не створений жоден екземпляр класа. Досить уточнити імя метода іменем класа (а не іменем обєкта), щоб метод міг працювати. Якраз так ми користувалися методами класу Math, не створюючи його екземпляри, а просто записуючи Math.abs(x), Math.sqrt(x). Точно так же ми використовували метод System.out.println(). Та і методом main() ми користуємося, взагалі не створюючи ніяких обєктів.

Тому статичні методи називаються *методами класу* (class methods), на відміну від нестатичних методів, що називаються *методами екземпляру* (instance methods).

Звідси витікають інші особливості статичних методів:

* в статичному методі не можна використовувати посилання this і super;
* в статичному методі не можна прямо, не створюючи екземплярів, посилатися на нестатичні поля і методи;
* статичні методи не можуть бути абстрактними;
* статичні методи перевизначаються в підкласах тільки як статичні.

Якраз тому в прикладі 1.5 ми помітили метод f() модифікатором static. Але в прикладі 3.1 ми працювали з экземпляром b2 класу Bisection2, і нам не знадобилося оголошувати метод f() статичним.

Статичні змінні ініціалізуються ще до початку роботи конструктора, але при ініціалізації можна використовувати тільки константні вирази. Якщо ж ініціалізація вимагає складних обчислень, наприклад, циклів для задання значень элементам статичних масивів або звернення до методів, то ці обчислення заключають в блок, помічений словом static, який теж буде виконаний до запуску конструктора:

**static int[] a = new a[10];**

**static {**

**for(int k = 0; k < a.length; k++)**

**a[k] = k \* k;**

**}**

Оператори, заключені в такий блок, виконуються лише один раз, при першому завантаженні класу, а не при створенні кожного екземпляра.

Наслідування в програмуванні Java з прикладами

Процес, за допомогою якого один клас набуває властивостей (членів даних) та функціональних можливостей (методів) іншого класу, називається наслідуванням. Мета наслідування - забезпечити повторне використання коду, щоб клас мав записувати лише унікальні особливості, а решта загальних властивостей та функціональних можливостей можна було б поширити з іншого класу.

Похідний клас:

Клас, який розширює особливості іншого класу, відомий як дочірній клас, підклас або похідний клас.

Клас батьків:

Клас, властивості та функціональність якого використовується (успадковується) іншим класом, називається батьківським класом, суперкласом або базовим класом.

Спадкування - це процес визначення нового класу на основі існуючого класу шляхом розширення його загальних членів даних та методів.

Спадкування дозволяє нам повторно використовувати код, це покращує повторне використання у вашому додатку Java.

Примітка. Найбільша перевага наслідування полягає в тому, що код, який вже є в базовому класі, не потрібно переписувати в дочірньому класі.

Це означає, що члени даних (змінні екземпляри) та методи батьківського класу можуть використовуватися в дочірньому класі.

Для успадкування класу ми використовуємо розширене ключове слово. Тут клас XYZ - це дочірній клас, а клас ABC - батьківський клас. Клас XYZ успадковує властивості та методи класу ABC.

class XYZ extends ABC

{

}

Приклад

У цьому прикладі ми маємо базовий клас Вчитель та підклас PhysicsTeacher. Оскільки клас PhysicsTeacher розширює позначення та властивості коледжу та метод work () з базового класу, нам не потрібно оголошувати ці властивості та метод у підкласі.

Тут ми маємо метод CollegeName, позначення та work (), який є загальним для всіх викладачів, тому ми оголосили їх у базовому класі. Таким чином, дошкільні класи типу MathTeacher, MusicTeacher та PhysicsTeacher не потребують написання цього коду та їх можна використовувати безпосередньо з базового класу.

class Teacher {

String designation = "Teacher";

String collegeName = "Beginnersbook";

void does(){

System.out.println("Teaching");

}

}

public class PhysicsTeacher extends Teacher{

String mainSubject = "Physics";

public static void main(String args[]){

PhysicsTeacher obj = new PhysicsTeacher();

System.out.println(obj.collegeName);

System.out.println(obj.designation);

System.out.println(obj.mainSubject);

obj.does();

}

}

Output:

Beginnersbook

Teacher

Physics

Teaching

можна сказати, що PhysicsTeacher IS-A Teacher. Тобто дочірній клас має IS-A відношення до батьківського класу. Наслідування відомо як IS-A відношення між батьківськи та дочірним класом

Note:  
The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

class Teacher {

private String designation = "Teacher";

private String collegeName = "Beginnersbook";

public String getDesignation() {

return designation;

}

protected void setDesignation(String designation) {

this.designation = designation;

}

protected String getCollegeName() {

return collegeName;

}

protected void setCollegeName(String collegeName) {

this.collegeName = collegeName;

}

void does(){

System.out.println("Teaching");

}

}

public class JavaExample extends Teacher{

String mainSubject = "Physics";

public static void main(String args[]){

JavaExample obj = new JavaExample();

/\* Note: we are not accessing the data members

\* directly we are using public getter method

\* to access the private members of parent class

\*/

System.out.println(obj.getCollegeName());

System.out.println(obj.getDesignation());

System.out.println(obj.mainSubject);

obj.does();

}

}

The output is:

Beginnersbook

Teacher

Physics

Teaching

The important point to note in the above example is that the child class is able to access the private members of parent class through protected methods of parent class. When we make a instance variable(data member) or method protected, this means that they are accessible only in the class itself and in child class. These public, protected, private etc. are all access specifiers and we will discuss them in the coming tutorials.

**Ієрархія**

Ієрархія обєктів давно використовується для їх класифікації. Особливо детально вона опрацьована в біології. Всі знайомі з сімействами, родами і видами. Ми можемо зробити описання своїх домашніх тварин (pets): кішок (cats), собак (dogs), корів (cows) і інших наступним чином:

**class Pet{ // Тут описуємо спільні властивості всіх домашніх улюбленців**

**Master person; // Хазяїн тварини**

**int weight, age, eatTime; // Вага, вік, час годування**

**int eat(int food, int drink, int time){ // Процес годування**

**// Початкові дії...**

**if (time == eatTime) person.getFood(food, drink);**

**// Метод приймання їди**

**}**

**void voice(); // Звуки, що видають тварини**

**// Інше...**

**}**

Потім створюємо класи, які описують більш конкретні обєкти, звязуючи іх із спільним класом:

**class Cat extends Pet{ // Описуються властивості, властиві лише кішкам:**

**int mouseCatched; // число спійманих мишей**

**void toMouse(); // процес ловіння мишей**

**// інші властивості**

**}**

**class Dog extends Pet{ // Властивості собак:**

**void preserve(); // охороняти**

**}**

Зверніть увагу, що ми не повторюємо спільні властивості, описані в класі Pet. Вони наслідуються автоматично. Ми можемо визначити обєкт класа Dog і використовувати в ньому всі властивості класа Pet так, як ніби то вони описані в класі Dog:

**Dog Tuzik = new Dog(), Sharik = new Dog();**

Після такого визначення можна буде написати

**Tuzik.age = 3;**

**int p = Sharik.eat (30, 10, 12);**

А класифікацію продовжити так:

**class Pointer extends Dog{ ... } // Властивості породи Пойнтер**

**class Setter extends Dog{ ... } // Властивості сеттерів**

Зверніть увагу, що на кожному наступному рівні ієрархії в клас додаються нові властивості, але жодна властивість не пропадє. Тому і використовується слово extends — "розширює" і говорять, що клас Dog — *розширення* (extension) класу Pet. З іншої сторони, кількість обєктів при цьому зменшується: собак менше, ніж всіх домашніх тварин. Тому часто говорять, що клас Dog — *підклас* (subclass) класу Pet, а клас Pet — *суперклас* (superclass) або надклас класу Dog.

Часто використовують генеалогічну термінологію: батьківський клас, дочірній клас, клас-нащадок, клас-предок, виникають племінники і внуки, вся неспокійна сімейка вступає у відносини, гідні мексіканського серіала.

В цій термінології говорять про *наслідування* (inheritance) класів, в нашому прикладі клас Dog наслідує клас Pet.

Ми ще не визначили щасливого хазяїна нашого домашнього зоопарка. Опишемо його в класі Master. Зробимо прикидку:

**class Master{ // Хазяїн тварини**

**String name; // Прізвище, імя**

**// Інші дані**

**void getFood(int food, int drink); // Годівля**

**// Інше**

**}**

Хазяїн і його домашні тварини постійно контактують. Їх взаємодія виражається дієсловами "гуляти", "годувати", "охороняти", "чиститиь", "лащитися", "проситися" і іншими. Для описаня взаємодії обєктів застосовується третій принцип обєктно-орієнтованого програмування — обовязок або відповідальність.

**3.2.3. Відповідальність**

В нашому прикладі розглядається тільки взаємодія в процесі годівлі, яка описується методом eat(). В цьому методі тварина звертається до хазяїна, умоляючи його застосувати метод getFood().

В англомовній літературі подібне звернення описується словом message. Це поняття невдало перекладено на українську мову ні до чого не зобовязуваним словом *"повідомлення".* Краще було б використати слово "послання", "доручення" або навіть "розпорядження". Але термін "*повідомлення*" устоявся і нам прийдеться його застосовувати. Чому ж не використовується словосполучення "виклик методу", адже говорять: "Виклик процедури"? Тому що між цими поняттями єсть, по крайній мірі, три відмінності.

* Повідомлення йде до конкретного обєкта, що знає метод розвязання задачі, в прикладі цей обєкт — поточне значення змінної person. У кожного обєкта є свій поточний стан, свої значення полів класу, і це може вплинути на виконання метода.
* Спосіб виконання доручення, що міститься в повідомленні, залежить від обєкта, якому воно послане. Один хазяїн поставить миску з "Chappi", другий кине кістку, третій вижене собаку на вулицю. Цю цікаву властивість називають *поліморфізмом* (polymorphism), її будемо обговорювати далі.
* Звернення до методу відбудеться лише на етапі виконання програми, компілятор нічого не знає про метод. Це називається *"пізнім звязуванням"* напротивагу *"ранньому звязуванню",* при якому процедура приєднується до програми на етапі компонування.

Отже, обєкт Sharik, виконуючи свій метод eat(), посилає повідомлення обєкту, посилання на який міститься в змінній person, с просьбою видати йому певну кількість їди і питва. Повідомлення записане в рядку person.getFood(food, drink).

Цим повідомленням заключається *контракт* (contract) між обєктами, суть якого в тому, що обєкт Sharik бере на себя *відповідальність* (responsibility) задати правильні параметри в повідомленні, а обєкт — поточне значення person — бере на себе *відповідальністьсть* застосувати метод годівлі getFood() , яким би він не був.

Для того щоб правильно реалізувати принцип відповідальності, застосовується четвертий принцип обєктно-орієнтованого програмування — *модульність* (modularity).

**3.2.4. Модульність**

Цей принцип стверджує — кожний клас повинен складати окремий модуль. Члени класу, до яких не планується звертання зовні, повинні бути інкапсульовані. В мові Java інкапсуляция досягається додаванням модифікатора private до описання члена класу. Наприклад:

**private int mouseCatched;**

**private String name;**

**private void preserve();**

Ці члени класів становляться *закритими,* ними можуть користуватися тільки екземпляри того ж самого класу, наприкладр, Тuzik може дати доручення

**Sharik.preserve().**

А якщо в класі Master ми напишемо

**private void getFood(int food, int drink);**

то метод getFood() не буде знайдено, і нещасний **S**harik не зможе отримати їжу.

В протилежність закритості ми можемо оголосити деякі члени класа *відкритими,* записавши замість слова private модифікатор public, наприклад:

public void getFood(int food, int drink);

До таких членів може зверннутися любий обєкт любого класа.

**Знавцям C++**

В мові Java словами private, public і protected відмічаються окремо кожний член класу.

Принцип модульності предбачає відкривати члени класу тільки у випадку необхідності. Згадайте напис: "Нормальне положення шлагбаума — закрите".

Якщо ж треба звернутися до поля класа, то рекомендується включить в клас спеціальні *методи доступу* (access methods), окремоо для читання цього поля (get method) і для запису в це поле (set method). Імена методів доступу рекомендується починати зі слів get і set, додаючи до цих слів імя поля. Для JavaBeans ці рекомендації введені в ранг закону.

В нашому прикладі класу Master методи доступу до поля Name в самому простому випадку можуть виглядіти так:

**public String getName(){**

**return name;**

**}**

**public void setName(String newName)**

**{**

**name = newName;**

**}**

В реальних ситуаціях доступ обмежується різними перевірками, особливо в *set-методах,* що змінюють значення полів. Можна перевіряти тип значения, що вводиться, задавати діапазон значень, зрівнювати зі списком допустимих значень.

Крім методів доступу рекомендується створювати провірочні *is-методи,* які повертають логічні значення true або false. Наприклад, в клас Master можна включити метод, перевіряючий, чи задано імя хазяїна:

public boolean isEmpty(){

return name == null ? true : false;

}

і використовувати цей метод для перевірки при доступі до поля Name, наприклад:

if (masterOl.isEmpty()) masterOl.setName(“Іванов");

Таким чином, ми залишаємо відкритими лише методи, необхідні для взаємодії обєктів. При цьому зручно спланувати класи так, щоб залежність між ними була найменшою, як прийнято говорит в теорії ООП, було найменше *зачеплення* (low coupling) міжду класами. Тоді структура програми сильно спрощується. Крім того, такі класи зручно використовувати як будівельні блоки для побудови інших програм.

Навпаки, члени класа повинні активно взаємодіяти один з одним, як говорять, мати тісну функціональну з*вязність* (high cohesion). Для цього в клас належить включати всі методи, що описують поведінку моделюючого обєкта, і тільки такі методи, нічого лишнього. Одно із правил досягнення сильної функціональної звязності, введенне Карлом Ліберхером (Karl J. Lieberherr), отримало назву *закон Деметра.* Закон гласить: "в методі m() класу А належить використовувати тільки методи класу А, методи класів, до яких належать аргументи метода m(), і методи класів, екземпляри яких створюються всередині метода m().

Обєкти, побудовні по цим правилам, схожі на кораблі, споряджені всім необходіним. Вони виходять в автономне плавання, готові виконати будь-яке доручення, на яке розрахована їх конструкція.

Чи будуть закриті члени класа доступні його наслідникам? Якщо в класі Pet написано

private Master person;

то чи можна використовувати Sharik.person? Розуміється, ні. Адже в противному випадку кожний, хто цікавиться закритими полями класу А, може розширити його класом B, і продивитися закриті поля класу А через екземпляри класу B.

Коли треба дозволити доступ наслідникам класу, але небажано віткривати його всьому світу, тоді в Java використовується *захищений* (protected) доступ, позначений модифікатором protected, наприклад, обєкт Sharik може звернутися до поля person батьківського класу Pet, якщо в класі Pet це поле описано так:

**protected Master person;**

Треба зразу сказати, що на доступ до члена класу впливає ще і пакет, в якому знаходиться клас, але про це поговоримо в наступній главі.

Із цього загального схематичного описання принципів обєктно-орієнтованого програмування видно, що мова Java дозволяє легко втілювати всі ці принципи. Ви уже зрозуміли, як записати клас, його поля і методи, як інкапсулювати члени класу, як зробити росширення класу і якими принципами належить при цьому користуватися. Розберемо тепер детальніше правила запису класів і розглянемо додаткові їх можливості.

Але, говорячи про принципи ООП, я не можу утриматися від того, щоб не нагадати основний принцип всякого програмування.

**3.3. Як описати клас і підклас**

Отже, описання класу починаеться зі слова class, після якого записується імя класу. "Code Conventions" рекомендує починать імя класу із заглавної літери.

Перед словом class можна записати модифікатори класа (class modifiers). Це одно із слів public, abstract, final, strictfp. Перед іменем вкладеного класу можна поставити, крім того, модифікатори protected, private, static. Модифікатори ми будем вводити по мірі вивчення мови.

Тіло класу, в якому в любому порядку перечисляються поля, методи, вкладені класи і інтерфейси, заключається в фігурні дужки.

При описанні поля вказується його тип, потім, через пробіл, імя і, може бути, початкове значення після знака рівності, яке можна записати константним виразом. Все це уже описано в уроці 2*.*

Описання поля може починатися з одного або декількох необовязкових модифікаторів public, protected, private, static, final, transient, volatile. Якщо треба поставити декілька модифікаторів, то перечисляти їх JLS рекомендує в указаному порядку, оскільки деякі компілятори вимагають певного порядку запису модифікаторів.

При описанні метода указується тип значення, який він повертає або слово void, потім, через пробіл, імя метода, потім, в дужках, список параметрів. Після цього в фігурних дужках розписується виконуваний метод.

Описання метода може починатися з модифікаторів public, protected, private, abstract, static, final, synchronized, native, strictfp.

В списку параметрів через кому перечисляються тип і імя кожного параметра. Перед типом якого-небудь параметра може стояти модифікатор final. Такий параметр не можна змінювати всередині метода. Список параметрів може бути відсутнім, але дужки зберігаються.

Перед початком роботи метода для кожного параметра виділяється ячейка оперативної памяті, в яку копіюється значення параметра, задане при зверненні до метода. Такий спосіб називається передачею параметрів *по значенню.*

В лістинзі 3.1 показано, як можна оформити метод ділення пополам для знаходження кореня нелінійного рівняння із лістинга 1.5.

**Лістинг 3.1. Знаходжение кореня нелінійного рівняння методом бісекцій**

**class Bisection2{**

**private static double EPS = 1e-8; // Константа**

**private double a = 0.0, b = 1.5, root; // Закриті поля**

**public double getRoot(){return root;}**

**// Метод доступа**

**public double f(double x)**

**{**

**return x\*x\*x - 3\*x\*x + 3; // Або щось інше**

**}**

**public void bisect(){ // Параметрів немає — метод працює з полями екземпляра**

**double y = 0.0; // Локальна змінна — не поле**

**do{**

**root = 0.5 \*(a + b); y = f(root);**

**if (Math.abs(y) < EPS) break;**

**// Корінь знайдено. Виходимо з циклу.**

**// Якщо на кінцях відрізка [a; root] функція має різні знаки:**

**if (f(a) \* y < 0.0) b = root;**

**// значить, корінь тут. Переносимо точку b в точку root**

**//В противному випадку:**

**else a = root;**

**// переносимо точку а в точку root**

**// Продовжуємо, доки [а; b] не стане малим**

**} while(Math.abs(b-a) >= EPS);**

**}**

**}**

**class Test{**

**public static void main(String[] args){**

**Bisection2 b2 = new Bisection2();**

**b2.bisect();**

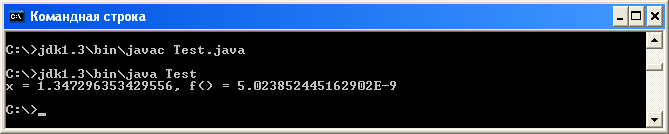
**System.out.println("x = " +**

**b2.getRoot() + // Звертаємося до кореня через метод доступа**

**", f() = " +b2.f(b2.getRoot()));**

**}**

**}**



В описанні метода f() збережений старий, процедурний стиль: метод одержує аргумент, опрацьовує його і повертає результат. Описання метода bisect() виконано в дусі ООП: метод активний, він сам звертається до полів экземпляра b2 і сам заносить результат в потрібне поле.

Імя метода, число і типи параметрів створюють *сигнатуру* (signature) метода. Компілятор розрізняє методи не по їх іменах, а по сигнатурах. Це дозволяє записувати різні методи з однаковими іменами, що відрізняються числом і/або типами параметрів.

**Зауваження**

Тип значення, що повертається, не входить в сигнатуру метода, значить, методи не можуть розрізнятися тільки типом результe та їх роботою.

Наприклад, в класі Automobile ми записали метод moveTo(int x, int у), позначивши пункт призначення його географічними координатами. Можна визначити ще метод moveTo (string destination) для вказівки географічної назви пункту призначення і звертатися до нього так:

**oka.moveTo("Полтава") ;**

Таке дублювання методів називається *перевантаженням* (overloading). Перевантаження методів дуже зручне у використанні. Згадайте, в уроці 2 ми виводили дані будь-якого типу на екран методом println() не турбуючись про те, дані якого іменно типу ми виводимо. На самім ділі ми використовували різні методи з одним и тим же іменем println, навіть не задумуючись про це. Звичайно, всі ці методи треба ретельно спланувати і заздалегідь описати в класі. Це і зроблено в класі PrintStream, де представлено близько двадцати методів print() і println(). Ось методи println().

|  |  |  |
| --- | --- | --- |
| void | println() | Terminate the current line by writing the line separator string. |
| void | println(boolean x) | Print a boolean and then terminate the line. |
| void | println(char x) | Print a character and then terminate the line. |
| void | println(char[] x) | Print an array of characters and then terminate the line. |
| void | println(double x) | Print a double and then terminate the line. |
| void | println(float x) | Print a float and then terminate the line. |
| void | println(int x) | Print an integer and then terminate the line. |
| void | println(long x) | Print a long and then terminate the line. |
| void | println(Object x) | Print an Object and then terminate the line. |
| void | println(String x) | Print a String and then terminate the line. |

Якщо ж записати метод з тим же іменем в підкласі, наприклад:

**class Truck extends Automobile{**

**void moveTo(int x, int y){**

**// Якісь дії**

**}**

**// Ще щось**

**}**

то він перекриє метод суперкласа. Визначивши экземпляр класа Truck, наприклад:

Truck gazel = new Truck();

і записавши gazel.moveTo(25, 150), ми звернемося до метода класу Truck. Відбудеться *перевизначення* (overriding) метода.

При перевизначенні права доступу до метода можна тільки розширити. Відкритий метод public повинен залишатися відкритим, захищений protected може стати відткритим.

Чи можна всередині підкласу звернутися до метода суперкласу? Так, можна, якщо уточнити імя метода, словом super, наприклад, super.moveTo(30, 40). Можна уточнити і імя метода, записаного в цьому ж класі, словом this, наприклад, this.moveTo (50, 70), але в даному випадку це вже зайве. Таким же способом можна уточняти і співпадаючі імена полів, а не тільки методів.

Дані уточнення подібні тому, як ми говорим про себя "я", а не "Іван Петрович", і говорим "батько", а не "Петро Сидорович".

Перевизначення методів приводить до цікавих результатів. В класі Pet ми описали метод voice(). Перевизначимо його в підкласах і використаємо в класі chorus, як показано в лістинзі 3.2.

**Лістинг 3.2. Приклад поліморфного метода**

**abstract class Pet{**

**abstract void voice();**

**}**

**class Dog extends Pet{**

**int k = 10;**

**void voice(){**

**System.out.println("Gav-gav!");**

**}**

**}**

**class Cat extends Pet{**

**void voice () {**

**System.out.println("Miaou!");**

**}**

**}**

**class Cow extends Pet{**

**void voice(){**

**System.out.println("Mu-u-u!");**

**}**

**}**

**public class Chorus{**

**public static void main(String[] args){**

**Pet[] singer = new Pet[3];**

**singer[0] = new Dog();**

**singer[1] = new Cat();**

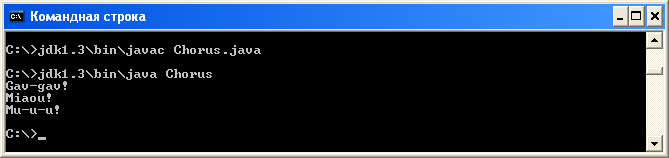
**singer[2] = new Cow();**

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

**singer[i].voice();**

**}**

**}**



Вся справа тут у визначенні поля singer[]. Хоча масив singer [] має тип Pet, кожний його елемент посилається на обєкт свого типу Dog, Cat, Сow. При виконанні програми викликається метод конкретного обєкта, а не метод класу, яким визначалося імя посилання. Так в Java реалізується поліморфізм.

**Знавцям C++**

В мові Java всі методи являються віртуальними функціями.

Уважний читач помітив у описанні класу Pet нове слово abstract. Клас Pet і метод voice() являються абстрактними.

При описаннні класу Pet ми не можемо задати в методі voice () ніякого корисного алгоритму, оскільки у всіх тварин різні голоси. В таких випадках ми записуємо тільки заголовок метода і ставимо після закриваючої список параметрів дужки крапку з комою. Цей метод буде *абстрактним* (abstract), що необхідно вказати компілятору модифікатором abstract.

Якщо клас містить хоча один абстрактний метод, то створити його екземпляри, а тим більше використовувати їх не удасться. Такий класс становиться *абстрактним,* що обовязково треба вказати модифікатором abstract.

Як же використовувати абстрактні класи? Тільки породжуючи від них підкласи, в яких перевизначені абстрактні методи.

Для чого ж потрібні абстрактні класи? Чи не краще зараз же написати потрібні класи з повністю визначеними методами, а не наслідувати їх від абстрактного класа? Для відповіді знову звернемося до лістингу 3.2.

Хоча елементи масива singer [] посилаються на підкласи Dog, Cat, Cow, але все-таки це змінні типу Pet і посилатися вони можуть тільки на поля і методи, описані в суперкласі Pet. Додаткові поля підкласу для них недоступні. Попробуйте звернутися, наприклад, до поля k класу Dog, написавши singer[0].k. Компілятор "скаже", що він не може реалізувати таку посилку. Тому метод, який реалізується в декількох підкласах, приходиться виносити в суперклас, а якщо там його неможливо реалізувати, то оголосити абстрактним. Таким чином, абстрактні класи групуються на вершині ієрархії класів.

До речі, можна задати пусту реалізацію метода, просто поставивши пару фігурних дужок, нічого не написавши між ними, наприклад:

void voice(){}

Получиться повноцінний метод. Але це штучне рішення, яке заплутає структуру класу. Замкнути ж ієрархію можна остаточними класами.

**3.6. Остаточні члени і класи**

В java є ключове слово – final. Воно може застосовуватись до класів, методів, змінних (в тому числі до аргументів-змінних методов).

Змінні з final

Для змінних модифікатор final має сенс схожий на слово const для Сі++. Якщо помітити модифікатором final опис змінної, то її значення (а воно повинно бути обовязково задано або тут же, або в блоці ініціалізації чи в конструкторі) не можна змінити ані в підкласах, ані в самому класі. Змінна перетворюється в константу. Якраз так в мові Java визначаються константи:

public final int MIN\_VALUE = -1, MAX\_VALUE = 9999;

Згідно "Code Conventions" константи записуються прописними літерами, слова в них розділяються знаком підкреслювання.

Таким чином, для змінних примітивного типу це означає, що одного разу присвоєне значення не може бути змінено. Для змінних-посилань це означає, що після присвоєння об'єкту, не можна змінити посилання на даний об'єкт. Отже, посилання змінити не можна, але стан об'єкта змінювати можна.

Примітка: З java 8 з'явилося поняття - effectively final. Застосовується воно лише до змінних (в тому числі аргументів методів). Суть в тому, що не дивлячись на явну відсутність ключового слова final, значення змінної не змінюється після ініціалізації. Іншими словами, до такої змінної можна підставити слово final без помилки компіляції. effectively final змінні можуть бути використані всередині локальних класів (Local Inner Classes), анонімних класів (Anonymous Inner Classes), Стрім (Stream API).

public void someMethod(){

// В прикладі нижче a та b - effectively final, так як значення встановлюютьcя один раз:

int a = 1;

int b;

if (a == 2) b = 3;

else b = 4;

// с НЕ є effectively final, так як значення змінилося

int c = 10;

c++;

Stream.of(1, 2).forEach(s-> System.out.println(s + a)); //Ок

Stream.of(1, 2).forEach(s-> System.out.println(s + c)); //Помилка компіляції

}

Методи з final

Помітивши метод модифікатором final, можна заборонити його перевизначення в підкласах. Це зручно в цілях безпеки. Ви можете бути впевненими, що метод виконує ті дії, які ви задали. Якраз так визначені математичні функції sin(), cos() інші в класі Math. Ми впенені, що метод Math.cos (x) обчислює якраз косинус числа х. Розуміється, такий метод не може бути абстрактним.

Для повної безпеки, поля, що обробляються остаточними методами, належить зробити закритими (private).

Класи з final

Якщо ж помітить модифікатором final увесь клас, то його взагалі не можна будет розширювати. Так визначений, наприклад, класс Math:

public final class Math{ . . . }

Для класу це означає, що клас не зможе мати підкласів, тобто заборонено успадкування. Це корисно при створенні immutable (незмінних) об'єктів, наприклад, клас String оголошений, як final.

public final class String{

}

class SubString extends String{ //Помилка компіляції

}

Слід також зазначити, що до абстрактних класів (з ключовим словом abstarct), не можна застосувати модифікатор final, тому що це взаємовиключні поняття. Для методу final означає, що він не може бути перевизначений в підкласах. Це корисно, коли ми хочемо, щоб вихідну реалізацію не можна було нехтувати.

public class SuperClass{

public final void printReport(){

System.out.println("Report");

}

}

class SubClass extends SuperClass{

public void printReport(){ // Помилка компіляції

System.out.println("MyReport");

}

}

Конструктори та наслідування

Конструктор підкласу викликається, коли ми створюємо об'єкт підкласу, він за замовчуванням викликає конструктор за замовчуванням суперкласу. Отже, у наслідувані об'єкти будуються зверху вниз.

Конструктор суперкласу може бути викликаний явно за допомогою ключового слова super, але це має бути першим оператором у конструкторі. Ключове слово super відноситься до суперкласу, безпосередньо над класом виклику в ієрархії. Використання декількох ключових слів super для доступу до класу предків, окрім прямого батька, не дозволяється.

class ParentClass{

//Parent class constructor

ParentClass(){

System.out.println("Constructor of Parent");

}

}

class JavaExample extends ParentClass{

JavaExample(){

/\* It by default invokes the constructor of parent class

\* You can use super() to call the constructor of parent.

\* It should be the first statement in the child class

\* constructor, you can also call the parameterized constructor

\* of parent class by using super like this: super(10), now

\* this will invoke the parameterized constructor of int arg

\*/

System.out.println("Constructor of Child");

}

public static void main(String args[]){

//Creating the object of child class

new JavaExample();

}

}

Output:

Constructor of Parent

Constructor of Child

Використання Super

1) super зі змінними: Цей сценарій виникає, коли похідний клас та базовий клас мають однакові члени даних. У цьому випадку існує можливість неоднозначності для JVM.

|  |
| --- |
| /\* Base class vehicle \*/  class Vehicle  {      int maxSpeed = 120;  }    /\* sub class Car extending vehicle \*/  class Car extends Vehicle  {      int maxSpeed = 180;        void display()      {          /\* print maxSpeed of base class (vehicle) \*/          System.out.println("Maximum Speed: " + super.maxSpeed);      }  }    /\* Driver program to test \*/  class Test  {      public static void main(String[] args)      {          Car small = new Car();          small.display();      }  } |

Output:

Maximum Speed: 120

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

2) super з методами:

Це використовується, коли ми хочемо викликати метод батьківського класу. Отже, коли батьки та дочірні класи мають однакові методи, то для вирішення неоднозначності ми використовуємо ключове слово super.

|  |
| --- |
| /\* Base class Person \*/  class Person  {      void message()      {          System.out.println("This is person class");      }  }    /\* Subclass Student \*/  class Student extends Person  {      void message()      {          System.out.println("This is student class");      }        // Note that display() is only in Student class      void display()      {          // will invoke or call current class message() method          message();            // will invoke or call parent class message() method          super.message();      }  }    /\* Driver program to test \*/  class Test  {      public static void main(String args[])      {          Student s = new Student();            // calling display() of Student          s.display();      }  } |

Output:

This is student class

This is person class

3)  super з конструктором

Ключове слово super також може використовуватися для доступу до конструктора батьківського класу. Ще одна важлива річ полягає в тому, що " super " може називати як параметричні, так і непараметричні конструктори в залежності від ситуації

|  |
| --- |
| /\* superclass Person \*/  class Person  {      Person()      {          System.out.println("Person class Constructor");      }  }    /\* subclass Student extending the Person class \*/  class Student extends Person  {      Student()      {          // invoke or call parent class constructor          super();            System.out.println("Student class Constructor");      }  }    /\* Driver program to test\*/  class Test  {      public static void main(String[] args)      {          Student s = new Student();      }  } |

Output:

Person class Constructor

Student class Constructor

Примітки:

Ключове слово super також може використовуватися для доступу до конструктора батьківського класу. Ще одна важлива річ полягає в тому, що " super " може викликати і параметричні конструктори:  
1. Виклик до super () повинен бути першим твердженням у конструкторі класу Derived (Student).  
2. Якщо конструктор явно не викликає конструктор класу-батька, компілятор Java автоматично вставляє виклик в конструктор без аргументів надкласу. Якщо в суперкласі немає конструктора без аргументів, ви отримаєте помилку часу компіляції. У Object є такий конструктор, тому якщо Object є єдиним суперкласом, то проблем немає.  
Якщо конструктор підкласу викликає конструктор свого надкласу явно або неявно, ви можете подумати, що цілий ланцюжок конструкторів викликається, аж до конструктора Об'єкта. Це, власне, так і є. Це називається ланцюгом конструктора ..

Спадкування і перевизначення методу

Dispatcherisation (virtual methods)

Поліморфізм на Java

Слово поліморфізм означає мати багато форм. Простими словами, ми можемо визначити поліморфізм як здатність повідомлення відображатися у більш ніж одній формі.  
Поліморфізм вважається однією з важливих особливостей об'єктно-орієнтованого програмування. Поліморфізм дозволяє виконувати одну дію різними способами. Іншими словами, поліморфізм дозволяє визначити один інтерфейс і мати кілька реалізацій. Слово "полі" означає багато, а "морфи" означає форми, тому воно означає безліч форм.  
У Java поліморфізм в основному поділяється на два типи:  
• Поліморфізм часу компіляції  
• Поліморфізм під час виконання

Поліморфізм часу компіляції:

Він також відомий як статичний поліморфізм. Цей тип поліморфізму досягається перевантаженням функцій або перевантаженням оператора.  
• Перевантаження методом: коли є кілька функцій з однаковою назвою, але різні параметри, то, як кажуть, ці функції перевантажені. Функції можуть бути перевантажені зміною кількості аргументів або / та зміною типу аргументів.

|  |
| --- |
| // Java program for Method overloading    class MultiplyFun {        // Method with 2 parameter      static int Multiply(int a, int b)      {          return a \* b;      }        // Method with the same name but 3 parameter      static int Multiply(int a, int b, int c)      {          return a \* b \* c;      }  }    class Main {      public static void main(String[] args)      {          System.out.println(MultiplyFun.Multiply(2, 4));            System.out.println(MultiplyFun.Multiply(2, 7, 3));      }  } |

**Output:**

8

42

2)

|  |
| --- |
| // Java program for Method overloading    class MultiplyFun {        // Method with 2 parameter      static int Multiply(int a, int b)      {          return a \* b;      }        // Method with the same name but 2 double parameter      static double Multiply(double a, double b)      {          return a \* b;      }  }    class Main {      public static void main(String[] args)      {            System.out.println(MultiplyFun.Multiply(2, 4));            System.out.println(MultiplyFun.Multiply(5.5, 6.3));      }  } |

**Output:**

8

34.65

* + Перевантаження оператору
  + В Java це можливо лише для оператору додавання (для числових типів та для рядків)

|  |
| --- |
| // Java program for Operator overloading    class OperatorOVERDDN {        void operator(String str1, String str2)      {          String s = str1 + str2;          System.out.println("Concatinated String - "                             + s);      }        void operator(int a, int b)      {          int c = a + b;          System.out.println("Sum = " + c);      }  }    class Main {      public static void main(String[] args)      {          OperatorOVERDDN obj = new OperatorOVERDDN();          obj.operator(2, 3);          obj.operator("joe", "now");      }  } |

**Output:**

Sum = 5

Concatinated String - joenow

1. Поліморфізм під час виконання ([Runtime polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/): also Dynamic Method Dispatch).

|  |
| --- |
| // Java program for Method overrinding    class Parent {        void Print()      {          System.out.println("parent class");      }  }    class subclass1 extends Parent {        void Print()      {          System.out.println("subclass1");      }  }    class subclass2 extends Parent {        void Print()      {          System.out.println("subclass2");      }  }    class TestPolymorphism3 {      public static void main(String[] args)      {            Parent a;            a = new subclass1();          a.Print();            a = new subclass2();          a.Print();      }  } |

**Output:**

subclass1

subclass2

Коли ми оголошуємо той самий метод у дочірньому класі, який вже присутній у батьківському класі, це називається методом перевизначення. У цьому випадку, коли ми викликаємо метод з об'єкта дочірнього класу, викликається версія дочірнього класу. Однак ми можемо викликати метод батьківського класу за допомогою ключового слова супер, як я показав у прикладі нижче:

class ParentClass{

//Parent class constructor

ParentClass(){

System.out.println("Constructor of Parent");

}

void disp(){

System.out.println("Parent Method");

}

}

class JavaExample extends ParentClass{

JavaExample(){

System.out.println("Constructor of Child");

}

void disp(){

System.out.println("Child Method");

//Calling the disp() method of parent class

super.disp();

}

public static void main(String args[]){

//Creating the object of child class

JavaExample obj = new JavaExample();

obj.disp();

}

}

The output is :

Constructor of Parent

Constructor of Child

Child Method

Parent Method

**Клас Object**

На самій вершині ієрархії класів Java стоїть класс Object.

Якщо при описанні класу ми не вказуємо ніяке розширення, тобто не пишемо слово extends і імя класу за ним, як при описанні класу Pet, то Java вважає цей клас розширенням класу Оbject, і компілятор дописує це за нас:

**class Pet extends Object{ . . . }**

Можна записати це розширення і явно.

Сам же класс Оbject не являється нічиїм наслідником, від нього починається ієрархія любих класів Java. Зокрема, всі масиви — прямі наслідники класу Оbject.

Оскільки такий клас може містити тільки загальні властивості всіх класів, в нього включено лише декілька самих загальних методів, наприклад, метод equals(), що порівнює даний обєкт на рівність з обєктом, заданим в аргументі, і повертаючий логічне значення. Його можна використати так:

**Object obj1 = new Dog(), obj 2 = new Cat();**

**if (obj1.equals(obj2)) ...**

Оцініть обєктно-орієнтований дух цього запису: обєкт obj1 активний, він сам порівнює себя с другим обєктом. Можна, звичайно, записати і obj2.equals(obj1), зробивши активним обєкт obj2, з тим же результатом.

Як указувалось в *главі 1,* посилання можна порівнювати на рівність і нерівність:

**obj1 == obj2; obj1 != obj 2;**

В цьому випадку співставляються адреси обєктів, ми можемо узнати, чи не указують обидва посилання на один і той же обєкт.

Метод equals() порівнює зміст обєктів в їх поточному стані, фактично він реалізований у класі Оbject як тотожність: обєкт рівний тільки самому собі. Тому його часто перевизначають в підкласах, більше того, правильно спроектовані, "добре виховані", класи повинні перевизначити методи класа Оbject, якщо їх не влаштовує стандартна реалізація.

Другий метод класу Оbject, який належить перевизначати в підкласах, — метод toString (). Цей метод без параметрів, який намагається зміст обєкта претворити в рядок символів і повертає обєкт класe String.

До цього методу виконуюча система Java звертається кожен раз, коли потрібно представити обєкт у вигляді рядка, наприклад, в методі printing.

Розглянемо методи класу Оbject

toString

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

System.out.println(tom.toString()); // Будет выводить что-то наподобие Person@7960847b

}

}

class Person {

private String name;

public Person(String name){

this.name=name;

}

}

Після ревизначення:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

System.out.println(tom.toString()); // Person Tom

}

}

class Person {

private String name;

public Person(String name){

this.name=name;

}

@Override

public String toString(){

return "Person " + name;

}

}

Метод hashCode

Метод hashCode дозволяє задати деякий числове значення, яке буде відповідати даному об'єкту або його хеш-код. За даним числа, наприклад, можна порівнювати об'єкти.  
Наприклад, виведемо представлення нашого об'єкта:

Person tom = new Person("Tom");

System.out.println(tom.hashCode()); // 2036368507

Або після перевантаження власним методом:

class Person {

private String name;

public Person(String name){

this.name=name;

}

@Override

public int hashCode(){

return 10 \* name.hashCode() + 20456;

}

}

Тип обєкту й метод getClass

Метод getClass дозволяє отримати тип даного обєкту:

Person tom = new Person("Tom");

System.out.println(tom.getClass()); // class Person

Метод equals

Метод equals порівнює 2 обєкти на рівність:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

Person bob = new Person("Bob");

System.out.println(tom.equals(bob)); // false

Person tom2 = new Person("Tom");

System.out.println(tom.equals(tom2)); // true

}

}

class Person {

private String name;

public Person(String name){

this.name=name;

}

@Override

public boolean equals(Object obj){

if (!(obj instanceof Person)) return false;

Person p = (Person)obj;

return this.name.equals(p.name);

}

}

Метод equals пріймає в якості параметру об'єкт будь-якого типу, який ми потім наводимо до поточного, якщо вони є об'єктами одного класу.  
Оператор instanceof дозволяє з'ясувати, чи є переданий в якості параметра об'єкт об'єктом певного класу, в даному випадку класу Person. Бо якщо об'єкти належать до різних класів, то їх порівняння не має сенсу, і повертається значення false.  
Потім порівнюємо по іменах. Якщо вони збігаються, повертаємо true, що буде говорити, що об'єкти рівні.

Внутрішні (вкладені) класи

Створення класів.

Поки що ми створювали клас та використовували його в середині методу main() цього самого класу. Але насправды класи можна створювати в тому ж файлі, де основний клас, але не в самому класі додатковим класом.

Class A{

int field1;

\*\*\*

int method1(){

return 0;

}

};

class Base{

public static void main(String[] args){

}

}

Коли ми скомпілюємо файл Base.java за допомогою javac ми отримаємо два фали-класів

A.class Base.class

Для того щоб використати обидва класи при інтерпретації байт-коду Base.class потрібно тоді викликати всередині public static void main класу Base створення обєкту класу А та його методи:

public static void main(String[] args){

A obj = new A();

A.method1();

}

Однак java дозволяє ще один шлях стоврення класів у базовому файлі – створення вкладеного класу(Nested class). Ми можемо обявити новий клас всередині базового, наприклад

class Base{

int field0;

class Nestedbase{

int field1;

void method1(){

System.out.println(“Nested”);

}

}

int method0(){

Nestedbase n2= Nestedbase();

}

public static void main(String[] args){

Nestedbase n1= new Nestedbase();

n1.method1();

Base b1 = new Base();

b1.method0();

}

}

*зауважимо, що в цьому випадку при створенны класу Base стврюється і клас Nestedbase, а для створення Nestedbase навпакі потрібен клас Base, бо він його частина.*

Класи можуть бути вкладеними (nested), тобто можуть бути визначені внурь інших класів. Окремим випадком вкладених класів є внутрішні класи (inner class). Наприклад, є клас Person, всередині якого визначено клас Account:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom", "qwerty");

tom.displayPerson();

tom.account.displayAccount();

}

}

class Person{

private String name;

Account account;

Person(String name, String password){

this.name = name;

account = new Account(password);

}

public void displayPerson(){

System.out.printf("Person \t Name: %s \t Password: %s \n", name, account.password);

}

public class Account{

private String password;

Account(String pass){

this.password = pass;

}

void displayAccount(){

System.out.printf("Account Login: %s \t Password: %s \n", Person.this.name, password);

}

}

}

Внутрішній клас поводиться як звичайний клас за тим винятком, що його об'єкти можуть бути створені тільки всередині зовнішнього класу.  
Внутрішній клас має доступ до всіх полів зовнішнього класу, в тому числі закритим за допомогою модифікатора private. Аналогічно зовнішній клас має доступ до всіх членів внутрішнього класу, в тому числі до полів і методів з модифікатором private.  
Посилання на об'єкт зовнішнього класу з внутрішнього класу можна отримати за допомогою виразу Внешній\_класс.this, наприклад, Person.this.  
Об'єкти внутрішніх класів можуть бути створені тільки в тому класі, в якому внутрішні класи опеределить. В інших зовнішніх класах об'єкти внутрішнього класу створити не можна.  
Ще однією особливостей внутрішніх класів є те, що їх можна оголосити всередині будь-якого контексту, в тому числі всередині методу і навіть в циклі:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

tom.setAccount("qwerty");

}

}

class Person{

private String name;

Person(String name){

this.name = name;

}

public void setAccount (String password){

class Account{

void display(){

System.out.printf("Account Login: %s \t Password: %s \n", name, password);

}

}

Account account = new Account();

account.display();

}

}

Статичні вкладені класи  
Крім внутрішніх класів також можуть статичні вкладені класи. Статичні вкладені класи дозволяють приховати деяку комплексну інформацію всередині зовнішнього класу:

class Math{

public static class Factorial{

private int result;

private int key;

public Factorial(int number, int x){

result=number;

key = x;

}

public int getResult(){

return result;

}

public int getKey(){

return key;

}

}

public static Factorial getFactorial(int x){

int result=1;

for (int i = 1; i <= x; i++){

result \*= i;

}

return new Factorial(result, x);

}

}

Здесь определен вложенный класс для хранения данных о вычислении факториала. Основные действия выполняет метод getFactorial, который возвращает объект вложенного класса. И теперь используем классы в методе main:

public static void main(String[] args) {

Math.Factorial fact = Math.getFactorial(6);

System.out.printf("Факториал числа %d равен %d \n", fact.getKey(), fact.getResult());

}

Перетворення класів

У минулому розділі говорилося про перетвореннях об'єктів простих типів. Однак з об'єктами класів все відбувається трохи по-іншому. Припустимо, у нас є наступна ієрархія класів:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

tom.display();

Person sam = new Employee("Sam", "Oracle");

sam.display();

Person bob = new Client("Bob", "DeutscheBank", 3000);

bob.display();

}

}

// класс человека

class Person {

private String name;

public String getName() { return name; }

public Person(String name){

this.name=name;

}

public void display(){

System.out.printf("Person %s \n", name);

}

}

// служащий некоторой компании

class Employee extends Person{

private String company;

public Employee(String name, String company) {

super(name);

this.company = company;

}

public String getCompany(){ return company; }

public void display(){

System.out.printf("Employee %s works in %s \n", super.getName(), company);

}

}

// класс клиента банка

class Client extends Person{

private int sum; // Переменная для хранения суммы на счете

private String bank;

public Client(String name, String bank, int sum) {

super(name);

this.bank=bank;

this.sum=sum;

}

public void display(){

System.out.printf("Client %s has account in %s \n", super.getName(), bank);

}

public String getBank(){ return bank; }

public int getSum(){ return sum; }

}

У цій ієрархії класів можна простежити наступну ланцюг успадкування: Object (всі класи неявно успадковуються від типу Object) -> Person -> Employee | Client.  
  
Суперкласу зазвичай розміщуються вище підкласів, тому на вершині успадкування знаходиться клас Object, а в самому низу Employee і Client.  
Об'єкт підкласу також представляє об'єкт суперкласу. Тому в програмі ми можемо написати наступним чином:

Object tom = new Person("Tom");

Object sam = new Employee("Sam", "Oracle");

Object kate = new Client("Kate", "DeutscheBank", 2000);

Person bob = new Client("Bob", "DeutscheBank", 3000);

Person alice = new Employee("Alice", "Google");

Це так зване восходяче перетворення (від підкласу внизу до суперкласу вгорі ієрархії) або upcasting. Таке перетворення здійснюється автоматично.  
Зворотне не завжди вірно. Наприклад, об'єкт Person не завжди є об'єктом Employee або Client. Тому спадне перетворення або downcasting від суперкласу до підкласу автоматично не виконується. У цьому випадку нам треба використовувати операцію перетворення типів.

Object sam = new Employee("Sam", "Oracle");

// нисходящее преобразование от Object к типу Employee

Employee emp = (Employee)sam;

emp.display();

System.out.println(emp.getCompany());

Змінна sam приводиться до типу Employee. Потім від обєкту emp ми звераємося до функціоналу Employee.

Обєкт Employee перетворюється по всій ліній від Object до Employee.

Приклади:

Object kate = new Client("Kate", "DeutscheBank", 2000);

((Person)kate).display();

Object sam = new Employee("Sam", "Oracle");

((Employee)sam).display();

Но рассмотрим еще одну ситуацию:

Object kate = new Client("Kate", "DeutscheBank", 2000);

Employee emp = (Employee) kate;

emp.display();

// или так

((Employee)kate).display();

В даному випадку змінна типу Object зберігає посилання на об'єкт Client. Ми можемо без помилок привести цей об'єкт до типам Person або Client. Але при спробі перетворення до типу Employee ми отримаємо помилку під час виконання. Так як kate не представляє об'єкт типу Employee.  
Тут ми явно бачимо, що змінна kate - це посилання на об'єкт Client, а не Employee. Однак нерідко дані приходять ззовні, і ми можемо точно не знати, який саме об'єкт ці дані представляють. Відповідно виникає велика ймовірна зіткнутися з помилкою. І перед тим, як провести перетворення типів, ми можемо перевірити, а чи можемо ми виконати приведення за допомогою оператора instanceof:

Object kate = new Client("Kate", "DeutscheBank", 2000);

if(kate instanceof Employee){

((Employee)kate).display();

}

else{

System.out.println("Conversion is invalid");

}

Выражение kate instanceof Employee проверяет, является ли переменная kate объектом типа Employee. Но так как в данном случае явно не является, то такая проверка вернет значение false, и преобразование не сработает.

Abstract

Крім звичайних класів в Java є абстрактні класи. Абстрактний клас схожий на звичайний клас. В абстрактному класі також можна визначити поля і методи, в той же час не можна створити об'єкт або екземпляр абстрактного класу. Абстрактні класи покликані надавати базовий функціонал для класів-спадкоємців. А похідні класи вже реалізують цей функціонал.  
При визначенні абстрактних класів використовується ключове слово abstract:  
public abstract class Human {  
  
private String name;  
  
public String getName () {return name; }  
}  
Але головна відмінність полягає в тому, що ми не можемо використовувати конструктор абстрактного класу для створення його об'єкта. Наприклад, у такий спосіб:  
Human h = new Human ();  
Крім звичайних методів абстрактний клас може містити абстрактні методи. Такі методи визначаються за допомогою ключового слова abstract і не мають ніякого функціоналу:  
public abstract void display ();  
Похідний клас зобов'язаний перевизначити і реалізувати всі абстрактні методи, які є в базовому абстрактному класі. Також слід враховувати, що якщо клас має хоча б один абстрактний метод, то даний клас повинен бути визначений як абстрактний.  
Навіщо потрібні абстрактні класи? Припустимо, ми робимо програму для обсулжіванія банківських операцій і визначаємо в ній три класи: Person, який описує людини, Employee, який описує банківського службовця, і клас Client, який представляє клієнта банку. Очевидно, що класи Employee і Client будуть похідними від класу Person, так як обидва класи мають деякі загальні поля і методи. І так як всі об'єкти будуть представляти або співробітника, або клієнта банку, то безпосередньо ми від класу Person створювати об'єкти не будемо. Тому має сенс зробити його абстрактним.

public class Program{

public static void main(String[] args) {

Employee sam = new Employee("Sam", "Leman Brothers");

sam.display();

Client bob = new Client("Bob", "Leman Brothers");

bob.display();

}

}

abstract class Person {

private String name;

public String getName() { return name; }

public Person(String name){

this.name=name;

}

public abstract void display();

}

class Employee extends Person{

private String bank;

public Employee(String name, String company) {

super(name);

this.bank = company;

}

public void display(){

System.out.printf("Employee Name: %s \t Bank: %s \n", super.getName(), bank);

}

}

class Client extends Person

{

private String bank;

public Client(String name, String company) {

super(name);

this.bank = company;

}

public void display(){

System.out.printf("Client Name: %s \t Bank: %s \n", super.getName(), bank);

}

}

Другим хрестоматийным примером является системы фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

// абстрактный класс фигуры

abstract class Figure{

float x; // x-координата точки

float y; // y-координата точки

Figure(float x, float y){

this.x=x;

this.y=y;

}

// абстрактный метод для получения периметра

public abstract float getPerimeter();

// абстрактный метод для получения площади

public abstract float getArea();

}

// производный класс прямоугольника

class Rectangle extends Figure

{

private float width;

private float height;

// конструктор с обращением к конструктору класса Figure

Rectangle(float x, float y, float width, float height){

super(x,y);

this.width = width;

this.height = height;

}

public float getPerimeter(){

return width \* 2 + height \* 2;

}

public float getArea(){

return width \* height;

}

}

Перерахування

роме отдельных примитивных типов данных и классов в Java есть такой тип как enum или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

|  |  |
| --- | --- |
|  | enum Day{        MONDAY,      TUESDAY,      WEDNESDAY,      THURSDAY,      FRIDAY,      SATURDAY,      SUNDAY  } |

Перечисление фактически представляет новый тип, поэтому мы можем определить переменную данного типа и использовать ее:

|  |  |
| --- | --- |
|  | public class Program{        public static void main(String[] args) {            Day current = Day.MONDAY;          System.out.println(current);    // MONDAY      }  }  enum Day{        MONDAY,      TUESDAY,      WEDNESDAY,      THURSDAY,      FRIDAY,      SATURDAY,      SUNDAY  } |

Перечисления могут использоваться в классах для хранения данных:

|  |  |
| --- | --- |
|  | public class Program{        public static void main(String[] args) {            Book b1 = new Book("War and Peace", "L. Tolstoy", Type.BELLETRE);          System.out.printf("Book '%s' has a type %s", b1.name, b1.bookType);            switch(b1.bookType){              case BELLETRE:                  System.out.println("Belletre");                  break;              case SCIENCE:                  System.out.println("Science");                  break;              case SCIENCE\_FICTION:                  System.out.println("Science fiction");                  break;              case PHANTASY:                  System.out.println("Phantasy");                  break;          }      }  }  class Book{        String name;      Type bookType;      String author;        Book(String name, String author, Type type){            bookType = type;          this.name = name;          this.author = author;      }  }    enum Type  {      SCIENCE,      BELLETRE,      PHANTASY,      SCIENCE\_FICTION  } |

Само перечисление объявлено вне класса, оно содержит четыре жанра книг. Класс Book кроме обычных переменных содержит также переменную типа нашего перечисления. В конструкторе мы ее также можем присвоить, как и обычные поля класса.

С помощью конструкции switch..case можно проверить принадлежность значения bookType определенной константе перечисления.

### Методы перечислений

Каждое перечисление имеет статический метод values(). Он возвращает массив всех констант перечисления:

|  |  |
| --- | --- |
|  | public class Program{        public static void main(String[] args) {            Type[] types = Type.values();          for (Type s : types) { System.out.println(s); }      }  }  enum Type  {      SCIENCE,      BELLETRE,      PHANTASY,      SCIENCE\_FICTION  } |

Метод ordinal() возвращает порядковый номер определенной константы (нумерация начинается с 0):

|  |  |
| --- | --- |
|  | System.out.println(Type.BELLETRE.ordinal());    // 1 |

### Конструкторы, поля и методы перечисления

Перечисления, как и обычные классы, могут определять конструкторы, поля и методы. Например:

|  |  |
| --- | --- |
|  | public class Program{        public static void main(String[] args) {            System.out.println(Color.RED.getCode());        // #FF0000          System.out.println(Color.GREEN.getCode());      // #00FF00          }  }  enum Color{      RED("#FF0000"), BLUE("#0000FF"), GREEN("#00FF00");      private String code;      Color(String code){          this.code = code;      }      public String getCode(){ return code;}  } |

Перечисление Color определяет приватное поле code для хранения кода цвета, а с помощью метода getCode оно возвращается. Через конструктор передается для него значение. Следует отметить, что конструктор по умолчанию приватный, то есть имеет модификатор private. Любой другой модификатор будет считаться ошибкой. Поэтому создать константы перечисления с помощью конструктора мы можем только внутри перечисления.

Также можно определять методы для отдельных констант:

|  |  |
| --- | --- |
|  | public class Program{        public static void main(String[] args) {            Operation op = Operation.SUM;          System.out.println(op.action(10, 4));   // 14          op = Operation.MULTIPLY;          System.out.println(op.action(6, 4));    // 24      }  }  enum Operation{      SUM{          public int action(int x, int y){ return x + y;}      },      SUBTRACT{          public int action(int x, int y){ return x - y;}      },      MULTIPLY{          public int action(int x, int y){ return x \* y;}      };      public abstract int action(int x, int y);  } |

Хотя мы можем создать обычный класс, который не является наследником, но фактически все классы наследуют от класса **Object**. Все остальные классы, даже те, которые мы добавляем в свой проект, являются неявно производными от класса Object. Поэтому все типы и классы могут реализовать те методы, которые определены в классе Object. Рассмотрим эти методы.

### toString

Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса. Например:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

System.out.println(tom.toString()); // Будет выводить что-то наподобие Person@7960847b

}

}

class Person {

private String name;

public Person(String name){

this.name=name;

}

}

Полученное мной значение (в данном случае Person@7960847b) вряд ли может служить хорошим строковым описанием объекта. Поэтому метод toString() нередко переопределяют. Например:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

System.out.println(tom.toString()); // Person Tom

}

}

class Person {

private String name;

public Person(String name){

this.name=name;

}

@Override

public String toString(){

return "Person " + name;

}

}

### Метод hashCode

Метод hashCode позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.

Например, выведем представление вышеопределенного объекта:

Person tom = new Person("Tom");

System.out.println(tom.hashCode()); // 2036368507

Но мы можем задать свой алгоритм определения хэш-кода объекта:

class Person {

private String name;

public Person(String name){

this.name=name;

}

@Override

public int hashCode(){

return 10 \* name.hashCode() + 20456;

}

}

### Получение типа объекта и метод getClass

Метод getClass позволяет получить тип данного объекта:

Person tom = new Person("Tom");

System.out.println(tom.getClass()); // class Person

### Метод equals

Метод equals сравнивает два объекта на равенство:

public class Program{

public static void main(String[] args) {

Person tom = new Person("Tom");

Person bob = new Person("Bob");

System.out.println(tom.equals(bob)); // false

Person tom2 = new Person("Tom");

System.out.println(tom.equals(tom2)); // true

}

}

class Person {

private String name;

public Person(String name){

this.name=name;

}

@Override

public boolean equals(Object obj){

if (!(obj instanceof Person)) return false;

Person p = (Person)obj;

return this.name.equals(p.name);

}

}

Метод equals принимает в качестве параметр объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса.

Оператор instanceof позволяет выяснить, является ли переданный в качестве параметра объект объектом определенного класса, в данном случае класса Person. Так как если объекты принадлежат к разным классам, то их сравнение не имеет смысла, и возвращается значение false.

Затем сравниваем по именам. Если они совпадают, возвращаем true, что будет говорить, что объекты равны.