Обекно ориентирано програмиране. Основни принципи. Класове и обекти. Оператори. Шаблони на функции и класове. Наслдяване и полиморфизъм.

В днешно време най-разпространените езици за програмиране са т.нар. *езици от високо ниво*. При тях програмистът описва схематично основната идея за решаването на задачата, а специалният програмен транслатор (компилатор или интерпретатор) превежда това описание в машинни инструкции за конкретния процесор. Тези езици не зависят от хардуера. Програмирането на високо ниво се свързва с **обектно-ориентирания подход** – програмиране с помощта на **обекти**, създаването и използването на абстрактен тип данни, наследяването и полиморфизма. Основните характеристики на обектно-ориентираното програмиране са:

- Всичко е обект.
- Една програма е съвкупност от обекти, указващи си един на друг чрез изпращане на съобщения какво да правят.
- Обектите притежават собствена памет, съставена от други обекти.
- Всеки обект има тип.
- Всички обекти от определен тип могат да получават едни и същи съобщения.

Обектът представлява *екземпляр* на съответния клас, който служи за тип на обекта. Той се състои от множество компоненти (променливи (полета) и методи (функции)). Достъпът до компонентите на обектите се осъществява чрез задаване името на обекта и името на данните или метода, разделени с точка. Пример: **O1.a**(*списък от фактически параметри*) – тук **a** е услугата, а **O1** е обекта, който ще извърши услугата.

Класът представлява тип данни, дефинирани от програмиста. Той може да обогатява възможностите на вече съществуващ тип или да представя напълно нов тип данни. Дефинирането на един клас представлява *декларацията*, включваща *заглавие* и *тяло*. Заглавието представлява запазената дума **class**, следвана от името на класа. Тялото съдържа декларация на членовете на класа – полета и функции, заградени от фигурни скоби.
В някои езици за програмиране /C++/ класовете могат да се дефинират освен *глобално*, на ниво програма, *локално,* вътре във функция или в тялото на друг клас. Областта на действие на глобално деклариран клас започва от декларацията му и продължава до края на програмата. Областта на действие на клас, дефиниран в тялото на друг клас или функция, е класа/функцията. Обектите на такъв клас са видими само в тялото на класа/функцията.

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, наречени *инициализация* и изпълнявани от специален вид член-функции на класовете –

конструкторите. Конструкторът притежава повечето характеристики на другите член-функции, но особеностите му са, че:

- името му съвпада с името на класа
- типът на резултата е this и явно не се указва
- изпълнява се автоматично при създаване на обекти
- не може да се извиква явно

В даден клас може явно да е дефиниран един или няколко, но може и да не е дефиниран конструктор. Във втория случай автоматично се създава т.нар. *подразбиращ се конструктор*, който реализира действия като заделяне на памет за член-данните на обект, инициализиране на някои системни променливи и т.н. Инициализацията на новосъздаден обект от един клас може да зависи от вече съществуващ обект от класа. Тази реализация се реализира от специален конструктор, наречен *конструктор за присвояване*. Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас>const &. Ако в един клас е дефиниран констуктор за присвояване, компилаторът го използва. В противен случай компилаторът автоматично създава такъв в момента, когато новосъздадения обект се инициализира с обект, намиращ се от дясната страна на оператора за присвояване или в кръгли скоби. Този конструктор за присвояване се нарича *конструктор за копиране*.

Разрушаването на обекти на класове в някои случаи е свързано с извършване на определени действия, които се наричат *заключителни*. Най-често тези действия са свързани с освобождаване на заделената преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност тези действия да се вършат автоматично при разрушаването на обекта. В C++ това се осъществява от **деструкторите** на класовете. Деструкторът е член-функция, която се извиква при:

- разрушаването на обект чрез оператора delete
- излизане от блок, в който е бил създаден обект на класа

Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа и е предшествано от символа ‘~’ (тилда). Типът му е void и явно не се задава в заглавието. Деструкторът няма формални параметри. Java обаче използва различен, по-безпроблемен подход: **garbage collection** (събиране на боклука). Тази система автоматично се грижи за обектите. Действа по следния начин: когато не съществува референция към даден обект, този обект се счита за вече ненужен и заетата от него памет се освобождава. След това тази рециклирана памет може да се използва за следващо заделяне. Възможно е да се дефинира метод, който се извиква точно преди даден обект да бъде унищожен завинаги от системата за ”събиране на боклука”. Този метод се нарича **finalize**() и може да се използва, за да се гарантира чистото приключване на един обект. Много важно е обаче да се прави разлика между C++ и Java, защото в C++ обектите *винаги* се унищожават, докато в Java паметта, заделена за обектите, не винаги се възстановява от garbage collector. Или, казано по друг начин: Garbage collector не означава унищожаване на обектите!

Шаблоните са средства, дефинирани в езика C++, които позволяват създаването на класове, използващи неопределени типове данни за своите аргументи, като по такъв начин позволяват описването на „обобщени" типове данни. Използват се за изграждането на общоцелеви класове-контейнери, като стекове, опашки, списъци и други. При наличие на шаблони на класове възниква нуждата и съответно са създадени и шаблони на функции, използващи шаблони на класове.

Пример за декларация на шаблонен клас:

```
Template <class T, class S = int> // дефинира се стойност по подразбиране за S - int
Class CLASS {
        Public:
                T function1(T x, S y);
                S function2(T x, S y);
        Private:
                T a;
                S b;
};
```

Дефиницията на клас, използващ този шаблон би могла да бъде следната:
Typedef CLASS<int, double> CL1;
Дефинира се класа CL1, който е специализация на шаблона CLASS при T – int и S – double. Възможна е и следната дефиниция:
Typedef CLASS<int > CL2;
Която дефинира класа CL2, който е специализация на шаблона CLASS при T – int и S – int. Т.е. за S ако не се зададе явно тип, то се използва този по подразбиране.

Дефиниране на член-функции на шаблон (примерът е за дефиниране на вградена член-функция – function1 и невградена член-функция – function2):

```
Class CLASS {
        Public:
                T function1(T x, S y) {
                        Cout << "This is function 1 \n";
                S function2(T x, S y);
        Private:
                T a;
                S b;
};
Template <class T, class S>
S CLASS<T, S>::function2(T x, S y) {
        Return y;
}
```

Възможно е да предефинираме член-функция на даден шаблон ако се налага при конкретен тип входни данни да бъде изпълнявана различна функция от обикновено. Примерно ако сме дефинирали шаблон за stack и функция print(), която извежда елементите на конкретен стек, но искаме ако тези елементи са от тип char, да бъдат изведени ASCII кодовете им, то тогава трябва да предефинираме print(). Това става по следния начин:

```
Void stack<char>::print() { // конкретизираме типа, при който тази реализация на
        Char x;             // print се изпълнява
        While (pop(x))
                Cout << int(x) << " ";
        Cout << endl;
}
```

Един от основните принципи на обектно-ориентираното програмиране е наследяването. То е процес, при който един обект може да придобие свойствата на друг обект, като същевременно има свои отличителни и уникални полета. Класът, който се наследява се нарича надклас или родителски клас, а класът, който извършва наследяването – подклас или производен клас. Извършва се чрез ключовата дума ***extends***. В C++ се означава с <*име_на_производен_клас*> : <*режим_на_достъп*> <*име_на_основен_клас*>, където режим на достъп може да бъде public, private или protected. Ако режимът на достъп е пропуснат, по подразбиране той е private. Езици като C++ позволяват така нареченото множествено наследяване, т.е. един клас може да наследи полетата и методите на няколко надкласа. Java обаче не позволява множественото наследяване, там може да имаме само един родителски клас.

По принцип обект от един клас има достъп до компонентите на родителския клас, като достъпът е директен – с имената на съответния компонент. Това правило обаче не важи, когато поле или метод в родителския клас е дефиниран като private. В такъв случай подкласът няма никакъв достъп до този компонент. Добра практика е компоненти да се декларират като private, за да се избегне тяхното неоторизирано използване, но да се декларират методи за достъп до съответните компоненти, които да бъдат използвани при нужда от достигане или промяна на стойността им (get и set методи).

Възможно е предефинирането на някоя компонента от надкласа. Тогава при извикване на предефинирана компонента се изпълнява тази, която е дефинирана в подкласа, освен ако не е указано изрично, че става дума за компонента на надкласа. Указването става по следния начин:

Super.member

Тук *member* може да е както променлива, така и метод, който е бил предефиниран. В C++ достъпът се извършва по следния начин:

име_на_клас::име_на_компонента

Където *име_на_клас* е името на надкласа.

При създаване на обект от клас, който наследява друг клас, този обект реално се състои от две части – една част от надкласа и една от подкласа. Тези две части трябва да се създадат отделно, като се използват конструкторите на двата класа. Ако само подкласа има явно дефиниран конструктор процесът е следния: този конструктор просто създава обект от типа на подкласа, като частта от надкласа

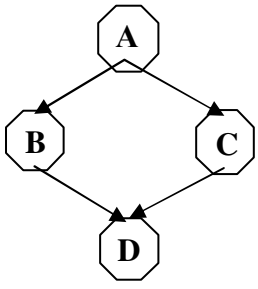
се създава автоматично от подразбиращия се конструктор. Ако обаче и двата класа имат дефинирани конструктори, тогава се използва ключовата дума **super**. Конструкторът на подкласа извиква този на надкласа по следния начин:

Super(списък_с_параметри)

списък_с_параметри е списък на параметрите, необходими на на конструктора на надкласа. Super задължително трябва да е първия изпълним код в конструктора на подкласа. Ако в надкласа има няколко дефинирани конструктора, то ще се изпълни този, който отговаря на подадените аргументи.
В C++ има т.нар. операторни функции за просвояване, при чието наследяване има особеност. Операторната функция за присвояване на производен клас трябва да указва как да стане присвояването както на собствените, така и на наследените член-данни. Тя прави това в тялото си (т.е. не поддържа инициализиращ списък с параметри). Ако в производния клас такава функция не е дефинирана, компилаторът създава такава (operator=(const <име_на_клас>&)) и тя се обръща към операторната функция за присвояване на основния клас. Тя инициализира наследената част, след което инициализира чрез присвояване и собствената част на производния клас. Ако в производния клас е дефинирана операторна функция за просвояване, тя трябва да се погрижи за наследените компоненти. Т.е. се налага в тялото ѝ да има обръщение към дефинирания оператор на основния клас, ако има такъв. (О учебника – Ако това не е направено явно, стандартът на езика не уточнява как ще стане присвояването на наследените компоненти. В случая операторът за присвояване на основния клас не се наследява.)
Деструкторите на производен клас и неговия родителски се изпълняват в ред, обратен на изпълнението на техните конструктори. Първо се изпълнява деструкторът на производния клас, след което деструкторът на основния клас.

Многократното наследяване на клас, което може да се получи при множественото наследяване в C++, води от една страна до затруднен достъп до многократно наследените членове, а от друга до поддържане на множество копия на член-данните на многократно наследения клас, което не е ефективно.

Фиг. 1



Фиг. 1 - В този случай класът D ще наследи 2 пъти член-данните на A – веднъж чрез B и веднъж чрез C

Преодоляването на тези недостатъци се осъществява чрез така наречените *виртуани основни класове*. Чрез тях се дава възможност да се „поделят" основни класове. Т.е. ако определим A като виртуален за класовете B и C, то класът D ще садържа само един „поделен" основен клас A. Синтакисът е следния:

Class <име_на_производен_клас> : virtual <режим_на_достъп><име_на_основен_клас>

В нашия случай:

Class B: virtual public A

Дефинирането и изпозлването на виртуални класове има две особености. Първата се отнася до дефиницията и използването на конструкторите на наследенитте класове. Нека A е виртуален основен клас за класа B, класът B е основен за класа D, който пък е основен за класа E. Ако класът A има

конструктор с параметри и няма подразбиращ се такъв, този конструктор трябва да бъде извикан не само в конструктора на класа B, но и в конструкторите на класовете D и E. Т.е. конструкторите с параметри на виртуалните класове трябва да се извикат от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на преките им наследници. Втората промяна се отнася за промяната на реда на инициализиране. Инициализирането на виртуални основни класове предхожда инициализирането на други основни класове в декларацията на производия клас. При няколко виртуални класа извикването на конструкторите става по реда им в декларацията на производния клас.

При обикновените функции, тъй като процесът на реализиране на обръщението към функцията приключва по време на компилация и не може да бъде променян по време на изпълнение на програмата, се казва, че има *статично разрешаване на връзката* или *статично свързване*. При статичното свързване по време на създаването на класа трябва да се предвидят възможните обекти, чрез които ще се викат член-функциите му. При сложни йерархии от класове това е не само трудно, но и понякога невъзможно. Езикът C++ поддържа още един механизъм, прилаган върху специален вид член-функции, наречен *късно* или *динамично свързване*. При него изборът на функцията, която трябва да се изпълни, става по време на изпълнение на програмата. Този механизъм за късно свързване се прилага върху специални член-функции на класове, наречени виртуални функции. Те се декларират по следния начин:

Virtual <*тип_на_резултат*> <*име_на_метод*>(<*параметри*>);

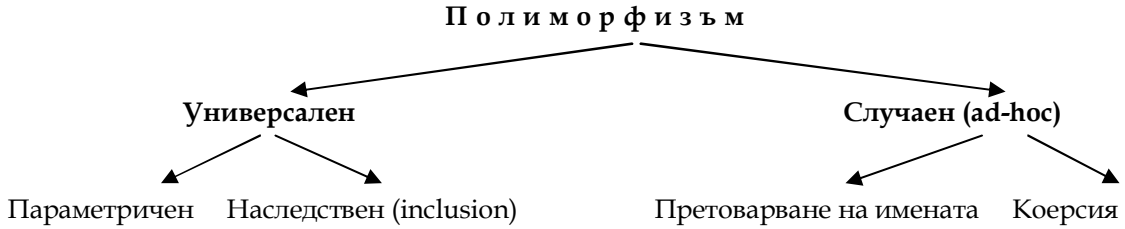
В класовата йерархия, ако метод в подкласа има същото име и сигнатура като друг, дефиниран в надкласа метод, се казва, че методът в подкласа *преопределя* (override) метода в надкласа. Когато преопределеният метод се извика от подкласа, винаги се изпълнява дефинираният в подкласа метод, а методът, дефиниран в надкласа, е скрит. Ако искаме да извикаме метода от надкласа, трябва да използваме **super**. Преопределяне на метод настъпва само, когато имената и сигнатурата на двата метода са еднакви. Ако това не е така и двата метода са различни по типова сигнатура, то те се наричат *претоварени* (overloaded).

Претоварването на методи оформя една от основните и най-мощни концепции в Java: *управлението на динамични методи*. Това е механизъм, при който извикването на претоварения метод се определя по време на изпълняване на програмата, а не по време на компилацията. Тази концепция е много важна, защото чрез нея в Java се осъществява полиморфизъм по време на изпълнение. Използва се фактът, че референцията на надклас може да се отнася към обект от подклас, за обръщение към претоварени методи по време на изпълнение на програмата. Когато един претоварен метод се извика чрез референция за надклас, се определя коя версия на метода да се изпълни на базата на типа на обекта, към който се отнася референцията по време на изпълнение на кода. Следователно методът се определя по време на изпълнение. Когато референцията се отнася за обекти от различни типове, ще се изпълнят съответно и различни версии на метода. С други думи, типът на метода

определя кой вариант ще се извика, а не типът на референцията. Ако един надклас има метод, който е предефиниран в негов подклас, то, когато с референция за надклас се посочат различни по тип обекти, то ще се изпълняват различни версии на метода.
Забележка: Предефинираните методи в Java са еквивалентни по предназначение и имат сходно действие с това на виртуалните функции в C++.

Т.нар. **интерфейс** дава представа за това как обектите си комуникират. В него методите не се реализират. Той служи да информира клиента за действията, които извършва програмата, но не и за тяхната реализация. Интерфейсът описва чистия екстракт на тип – множество от стойности, множество от операции, но не и реализацията. Той съдържа сигнатура на метод. Един клас може да реализира много интерфейси.

Полиморфизмът е важна характеристика на ООП. Изразява се в това, че едни и същи действия се реализират по различен начин в зависимост от обектите, върху които се прилагат, т.е. действията са полиморфни (с много форми). Той е свойството, което позволява на един интерфейс да извършва достъп до общ клас от действия. Един елементарен пример за полиморфизъм може да се открие при кормилната уредба на един автомобил. Кормилната уредба (т.е. интерфейсът) е една и съща, независимо от механизма на управление. Следователно, ако знаем как да я използваме, можем да караме всякакъв вид коли. По-общо казано, принципът на полиморфизма най-често се изразява с фразата "един интерфейс, множество методи".



Универсален полиморфизъм:
- Изпълнява се един и същи код за аргументи от всякакви допустими типове
- Държи се по еднороден начин за всички съответни типове

Случаен полиморфизъм:
- Изпълнява се различен код за всеки отделен тип на аргументите
- Държи се по различен начин с обекти от различни типове
- Позволява на дадена стойност да има само краен брой различни типове, а не безкраен
- Просто малка група от мономорфни функции

Параметричен полиморфизъм: Функцията работи по един и същи начин с различни типове, които обаче обикновено имат сходна структура. Функцията е с обширно приложение – например length. Има параметър за тип.

Наследствен полиморфизъм: Включва йерархия от подтипове. Опростява замяната на типове. В тази група влиза и предефинирането на компоненти в базовия клас (override)
Полиморфизъм с претоварване на имената (overload): Това представлява удобни синтактични съкращения. Повторната обработка на програмата би премахнала претоварването чрез даване на различни имена на различните функции. Позволява на един и същи оператор да изпълнява различни функции в зависимост от контекста, например: събиране на числа (3+4) и конкатенация (“абв”+”где”).
Полиморфизъм с коерсия: Позволява на типа на аргумента да бъде превърнат в такъв, какъвто функцията очаква; така се избягват грешки, свързани с типовете; Смислени операции, които позволяват намаляване на размера на кода; Разликата между претоварване и коерсия не е ясна в няколко ситуации: например, ако имаме 3+4, 3+4.0, 3.0+4, 3.0+4.0, може да става въпрос и за претоварване, и за коерсия, или за смесица от двете, в зависимост от конкретния код.

Понякога е необходимо да се създаде надклас, който дефинира единствено основната форма на подкласовете си, като оставя на тях да попълнят детайлите. Подобен клас определя природата на методите, които подкласовете трябва да разработят, но самият той не разработва един или няколко от тях. Подобна нужда може да възникне, когато надкласът не може да попълни даден метод със смислено съдържание. В такива случаи се използват *абстрактните методи*. Абстрактен метод се създава чрез типовия модификатор ***abstract***. Този метод няма тяло и не се разработва в надкласа. Затова подкласът трябва да го предефинира – не може просто да използва версията от надкласа. Основната форма на декларацията на абстрактен метод е следната:

abstract *type name(списък-с-парам);*

Тяло на метода не съществува. Модификаторът **abstract** може да се използва само с обикновени методи. Не е разрешено прилагането му върху **static** методи или конструктори.
Клас, който съдържа един или повече абстрактни методи, също трябва да се дефинира като абстрактен. Това става като пред **class** се постави модификаторът **abstract**. Тъй като имплементацията на абстрактния клас не е пълна, от него не могат да се създават обекти. Всеки опит за създаване на обект от абстрактен клас чрез оператора **new** ще предизвика грешка при компилиране.
Когато един подклас наследява абстрактен надклас, той трябва да имплементира всички абстрактни методи. Ако той не имплементира дори само един от тях, подкласът също трябва да се обяви като абстрактен. Атрибутът **abstract** се наследява до тогава, докато се завърши цялата имплементация.