УДК 681.3.06

М.М. Колодницький

ЕВОЛЮЦІЯ ТА "ПРИРОДНИЙ ВІДБІР" БАЗОВИХ КОНЦЕПЦІЙ МОВ ПРОГРАМУВАННЯ: НА ПРИКЛАДАХ МОВ C, C++, JAVA ТА C#

Коротко нагадуються деякі важливі історичні події, що мали суттєвий вплив на формування концепцій сучасних мов програмування загального призначення. Розглядається набір фундаментальних конструкцій та концепцій, які є свого роду будівельними цеглинками вказаних у назві чотирьох мов. Пропонується систематизація такого набору, яка показує, як відображаються компоненти архітектури комп'ютера у відповідні концепції мов програмування. Дану систематизацію зручно взяти за основу для проведення порівняльного аналізу цих мов. Наводяться систематизовані порівняльні таблиці базових імперативних концепцій чотирьох мов, а також порівнюються реалізації об'єктно-орієнтованих конструкцій в мовах C++, Java 2 та C#. На основі таких матеріалів проводиться аналіз еволюції базових концепцій цих мов.

Вступ

Мови програмування, без сумніву, є однією із основних областей комп'ютерних наук та інформаційних технологій. Для ефективного володіння певною мовою важливо добре розуміти її базові концепції, свого роду будівельні цеглинки мов, а для кращого їх розуміння корисно прослідкувати їх еволюцію та взаємозв'язок з різними мовами програмування. Особливо актуальною така задача є для нових мов програмування, таких, наприклад, як Java та C# (читається: "сі-шарп"). Дана стаття присвячена саме цій проблемі. Після короткого історичного екскурсу щодо розвитку мов програмування проаналізуємо модифікації їх базових концепцій на прикладі цих чотирьох мов та проведемо порівняння об'єктно-орієнтованих (ОО) конструкцій мов C++, Java та C#, звернемо увагу на ті елементи мови, що були відібрані в результаті еволюції цих мов.

1. Коротка історія мов програмування: від Plankalkül до наших днів

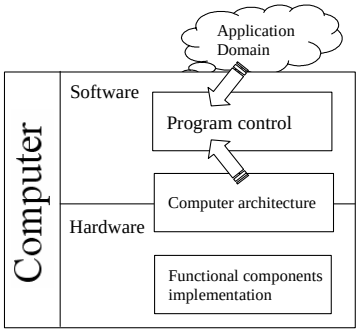

Рис. 1. Представлення комп'ютера як багаторівневої

Найголовнішою рисою комп'ютера як технічної системи є те, що він є програмно керованою системою. Більш детально цей факт можна представити рис. 1.

Те, які можливості надаються для проведення програмного керування (на рис. 1 — program control), цілком і повністю визначається архітектурою комп'ютера (на рис. 1 — computer architecture). Згідно означення [1], архітектура комп'ютера — це опис поведінки функціональних компонентів комп'ютера, який проводиться в термінах машинної мови комп'ютера, тобто мови, що є не природною, а мовою програмування, проте — програмування

найнижчого рівня. Власне, причиною створення мов програмування високого рівня (таких, як FORTRAN, C, Java) і була потреба у спрощенні процесу запису *алгоритму* керування комп'ютером при вирішенні прикладних задач з певної галузі застосування (на рис. 1 — application domain). Таким чином, на, так би мовити, природу мов програмування високого рівня, тобто на набір їх базових концепцій, чиниться вплив одночасно з двох протилежних напрямків — з боку архітектури комп'ютера та з боку прикладних задач (application domain) для вирішення яких та чи інша мова проектується.

Історично першими мовами програмування високого рівня були "Plankalkül" (Zuse, 1945), "Short Code" (Mauchky et al, 1949), "Intermediate PL" (Burks, 1950), "AUTOCODE" (Glennie, 1952), "A-2" (Hopper et al, 1953), "Algebraic Interpreter" (Lanin та Zierler, 1953), "FORTRAN" (Bacus et al, 1954-1957), "ПП-2" (Камынин и Любимский, 1954), "MATH-MATIC" (Katz et all, 1956-1958) та інші. Їх детальний огляд та аналіз проведено в [2]. Як бачимо, навіть за перше десятиріччя свого розвитку мови програмування вже відзначалися своєю різноманітністю як за їх кількістю, так і за змістом, тобто за тими концепціями програмування, що вони реалізовували.

Звичайно, задача проектування мови як такої виконувалась разом із задачею створення її компілятора, тобто програми, що переводила процедуру керування комп'ютером, описану мовою "високого рівня", в таку, що відповідала рівню архітектури комп'ютера — "машинну програму", яка могла безпосередньо виконуватися апаратними засобами комп'ютера. Розвиток технологій розробки компіляторів спричинював, у свою чергу, вплив на розвиток самих мов програмування, для яких такі компілятори створювались. Це породжувало свого роду ланцюгову реакцію.

Наступне десятиріччя характеризувалось "бумом" виникнення все нових та нових мов програмування. Ця ситуація у 1969 році в одній із публікацій була представлена у вигляді "Вавілонської вежі мов програмування" у порівнянні з відомим біблейським "прототипом" [3]. Наведений факт звертає нашу увагу на те, що вже на досить ранніх етапах історії розвитку мов у зв'язку із складністю процесу програмування була розпізнана важливість проблеми створення підходящого набору концепцій, який би допоміг зробити процес програмування мовами високого рівня більш легким.

Пройшло ще 10 років, і в 1978 році темі обговорення еволюції мов програмування вже була присвячена окрема конференція — Перша конференція з історії мов програмування, на якій обговорювалося 13 мов та їх сімейств: APL, APT, ALGOL 60, BASIC, COBOL 60, FORTRAN, GPSS, JOSS, JOVIAL, LISP, PL/1, SIMULA, SNOBOL [4]. Зауважимо також, що серед них не всі були мовами програмування загального призначення (general purpose languages), деякі з них — мови моделювання (specific domain-oriented), наприклад GPSS (discrete events system simulation), а деякі навіть мови керування певним обладнанням, як, наприклад, токарним чи іншим верстатами — APT (automatically programmed tools).

У даній статті надалі обмежимося розглядом лише мов програмування загального призначення і не будемо обговорювати спеціальні мови моделювання на зра-

зок GPSS, MatLab, Macsysma (так звані mathsoft та/або CAS – Computer Algebra System), а також не розглядатимемо мови логічного програмування типу Prolog або дуже спеціалізовані мови на зразок SNOBOL (string processing, text editing), RPG (Report Generators), мови CASE-засобів, лексичного (lex) та синтаксичного (syntax generators) аналізу, compiler compilers тощо.

На Першій конференції з історії мов програмування, зокрема, також зверталася увага на те, що в концепціях, які "винаходилися" в нових мовах програмування, багато що насправді вперше було запропоновано ще на початку 50-х років в найперших мовах програмування. Це ще раз свідчить про те, що питання вибору "раціонального" набору концепцій мови програмування все ще залишалось актуальним на той час.

У роботах Другої конференції з історії мов програмування (1993) до дискусії про еволюцію мов та їх концепцій вже були включені також сучасні мови, такі, як C, C++, ADA, та інші [5–18].

2. Базові концепції імперативних мов програмування

Як бачимо, проблема визначення набору базових концепцій мов програмування високого рівня та систематизація таких концепцій привертала до себе увагу протягом майже всього часу їх розвитку, і цій темі було присвячено немало публікацій як в періодиці, так і в навчальній літературі [19–24]. Так, в одному з історично перших підручників, присвячених цій темі, розглядається система концепцій мов програмування на прикладах мов Fortran IV, Algol-60, Cobol, SNOBOL, Lisp та PL/1 [20], пізніше до розгляду були включені також мови Pascal та Ada [21]. Деякі книги з цієї теми, наприклад [22], поповнювалися и перевидавалися декілька разів за останні 20 років, вводячи до розгляду все нові та нові сучасні мови програмування: C, C++, Prolog, ML, Smalltalk, Fortran-90, Ada-95. В [23–24] поруч з розглядом мов імперативного програмування почали розглядатися також об'єктно-орієнтовані конструкції, мови функціонального програмування ML, Lisp, Haskell та інші парадигми. Все це лише ще раз підкреслює актуальність проблеми, якій і присвячена дана стаття.

Набір базових концепцій, що наводиться нижче, був сформований на основі вивчення зазначеної літератури, "офіційних" специфікацій мов [26–30], а також аналізу того факту, що, як вже згадувалося вище, на мову програмування чиниться вплив з двох протилежних сторін — прикладних задач, що мають вирішуватися, та архітектури комп'ютера; це є свого роду вісь з двома полюсами: "потреби – засоби" (needs – means).

Прикладом того, як впливала специфіка прикладних задач на конструкції мови програмування протягом їх історичного розвитку, може бути табл. 1, де показано, що виникнення потреби у таких специфічних задачах, як проведення обробки текстових даних (business data processing), списків (list processing), маніпуляцій з символами (symbolic (formula) manipulation), створення структур даних, визначених користувачем (user-defined data structures), тощо, приводило до впровадження відповідних конструкцій у різних мовах або навіть до виникнення нових мов програмування.

(Ще раз нагадаємо, що тут розглядаються тільки імперативні (imperative, або procedural, або command) мови програмування, а

Таблиця 1. **Вплив на базові конструкції мов програмування, спричинений *вимогами* прикладних задач**

	Вимоги прикладних задач (application domain requirements)								
	Numerical scientific computation	Business data processing	List processing	Symbolic (formula) manipulation	Multi-purpose	User-defined data structures	Dynamic storage allocation	Interactive	System programming
Мови, в яких вимоги були реалізованими вперше	FORTRAN-IV Family*, ALGOL-60	COBOL	LISP	FORMAC	JOVIAL, PL/1	ALGOL-68	APL, SNOBOL	JOSS, BASIC	CPL, BCPL, C (1969–73)
Рік	1954–	1960	1959	1962	1959,	1968	1960,	1964,	1962,

також об’єктно-орієнтований підхід і не розглядаються мови логічного або функціонального програмування, мови-сценаріїв (scripts) тощо.)

Для оцінки того, який зв’язок існує між конструкціями мови програмування та архітектурою комп’ютера, доцільно звернути увагу на те, що остання фактично може бути представлена наступним списком концепцій — табл. 2.

Таблиця 2. **Базові концепції для представлення архітектури комп'ютера**

	L *	Складові/компоненти концепції архітектури комп'ютера
Керування обчисленнями (computing control)	1	Дані (Data):
		– Імена (Name):
		– Пам'ять (Memory type, Address type, space (size), and organization)
		– Режим адресації (Addressing method)
		– Значення (Value, data format):
		– Числа десяткові (Fixed-Point Numbers)
		– Числа, представлені у форматі з плаваючою комою (Floating-Point Numbers)
		– Логічні значення (Logical Data)
		– Символи та рядки символів (Character Strings)
		– Агрегатні дані та масиви (Arrays)
	2	Операції (Operations):
		– Арифметика десяткових чисел (Fixed-Point Arithmetic)
		– Арифметика чисел у форматі з плаваючою комою (Floating-Point Arithmetic)
		– Логічні (Logic)
		– Бінарних відношень (Relational Operations)
		– Обробки символів та рядків символів (Data Handling)
		– Обробки масивів чисел (Numeric-Array Operations)
	3	Виконання інструкцій (Instruction Sequencing):
		– Лінійна послідовність (Linear Sequence)
		– Розгалуження (Decision)
		– Циклічна послідовність (Iteration)
		– Виклик функцій чи процедур (Delegation)
Керування ресурсами комп'ютера (computer resources management)	4	Більш високий рівень керування (Supervision and Operating System support):
		– "Одночасне" виконання певних дій (Concurrency and multiprocessor computation)
		– Перемикання між процесами (Control Switching and interruption)
		– Збереження стану обчислень (State-Saving)
		– Взаємодія процесів (Inter Process Communication, IPC) та синхронізація процессів
		– Цілісність даних (Data Integrity, memory protection, resource sharing, data access)
	5	Введення / виведення (Input / Output) та зв'язок по мережі (and communication):
		– На зовнішніх пристроях (Input / Output Devices)

Детально ці концепції розглядаються в [1] і в даній статті додатково не описуються. В даному контексті достатньо мати лише їх представлення у певному, систематизованому вигляді, тобто деякого структурованого списку, як у табл. 2.

Як побачимо далі, перша частина табл. 2 — керування обчисленнями (computing control) — відображається у відповідних концепціях та конструкціях мов програмування (див. нижче в табл. 3 розділи 3-7, тобто data (variables, constants), expressions, control flow operators, functions). Друга частина табл. 2 — керування ресурсами комп'ютера (computer resources management) — знаходить своє відображення у структурі бібліотек стандартних функцій (чи класів), що звичайно постачаються з компілятором мови програмування та "бібліотеки часу виконання" (run-time library).

Отже, об'єднавши зазначені вище ідеї, тобто розглянувши сумісний вплив на набір концепцій мови одночасно і з боку конструкцій задач предметної області і з боку різноманітності компонентів архітектури комп'ютера, отримуємо набір базових концепцій мов програмування високого рівня — табл. 3, друга колонка. В колонках 3-6 табл. 3 вказується на те, чи є така концепція або конструкція реалізованою в мовах C, C++, Java та C# відповідно (знаки "+" чи "-"), а також наводяться посилання на короткі коментарі (пронумеровані цифрами від 1 до 42) щодо особливостей *семантики* цих конструкцій.

Зауважимо тут також, що концепції, пов'язані з об'єктно-орієнтованим програмуванням (ООП), не включені до табл. 3, а розглядаються трохи нижче (див. табл. 4 та 5). Одночасно мова C вилучена із порівняння в табл. 5, оскільки не є об'єктно-орієнтованою.

В табл. 3 та 5, крім того, назви конструкцій та концепцій мови в основному наведені англійською мовою, тобто мовою їх першоджерела, з метою уникнення надлишковості, спричиненої транслітерацією термінів. Це сприяє також вживанню коректної та уніфікованої термінології, не спотвореної неточними перекладами. Самі терміни в табл. 3 та коментарі до них наводяться у досить стислому виді, що пов'язано з обмеженими розмірами даної публікації. З цією ж метою в тексті також опущені деякі посилання на літературу, з якої запозиченні загально прийняті означення термінів та певні "сталі вирази". Проте це не повинно спричинити труднощі для читача, обізнаного з основами теорії мов програмування або маючого досвід з програмування хоча б однією з мов, що тут розглядається. Додамо також, що табл. 3-5 ні в якому разі не можуть розглядатися як такі, що дають абсолютно вичерпну інформацію з усім можливим ступенем детальності про всі конструкції вказаних мов. Для цієї мети служать специфікації мов та додаткова документація до них. За мету створення даних таблиць автор бачив необхідність наведення стислого та систематизованого переліку концепцій та конструкцій мов, який був би корисним на практиці при вивченні мов та їх порівнянні.

Нижче наводять стислі коментарі до табл. 3, пронумеровані цифрами від 1 до 42.

[1] C/C++ програма складається з одного або декілька (source-) файлів, що називаються "translation unit"; source-файл є колекцією речень. В Java-програмі "translation unit" це або клас, або *package*, в якому містяться декілька класів; в C#

— це або клас, або *namespace*, в якому містяться декілька класів. Кожний клас в package / namespace складається із речень (sentence).

Додамо, що в традиційно (див., наприклад, [20]) класичних імперативних мовах, таких, як Fortran, Algol-60, PL/1, Pascal, розглядалася система ієрархічних рівнів програми, таких, як 1) програма (program) — одиниця виконання; 2) процедура (procedure) — одиниця компіляції, модульності; 3) блок (block) — одиниця області дії імен та обробки переривань; 4) оператор (statement, compound statement) —

Таблиця 3. **Базові концепції мов програмування C, C++, Java 2 та C#**

	Базові концепції та конструкції мов програмування	**C**	**C++**	**Java**	**C#**
1	**2**	**3**	**4**	**5**	**6**
1	Структура програми (program structure [1]):				
	– a primary translation unit (містить *main*() функцію):				
	– header file (optional)	+	+	–	–
	– source file (складається із речень (sentences) мови)	+ [2]	+	+	+ [2]
	– other translation units:				
	– header files	+	+	–	–
	– source files	+	+	+	+
2	Речення (sentences [3], іноді вживається також термін statements):				
	– directives and pragmas	див. нижче п. 3			
	– data declarations, definitions, and initialization; (див. також коментар № 19)	див. нижче п. 4			
	– expression and control flow	див. нижче п. 5, 6			
	– functions	див. нижче п. 7			
	– comments [4] and punctuators (separators)	+	+	+	+
3	Директиви, макроси та прагми компілятора (directives, macro and pragmas):				
	– compiler directives, macro definitions (#include, #define, etc.)	+	+	– [5]	+ [6]
4	Дані (data), тобто змінні величини (variables) чи константи:				
	4.1. Імена змінних (names, identifiers) [7a]:				
	4.1.1. Naming rules:				
	– only letters and digits starting with a letter or underscore symbol "_"	+	+	+ [7]	+ [7]
	– case sensitiveness; distinct from reserved words	+	+	+	+ [8]
	– the unlimited length of a name.	– [9]	– [9]	+	+
	4.1.2. Scope (or access) [10] (область дії або "доступ до…"):				
	– scoping rules	+ [11]	+ [12]	+ [12a]	+ [12a]
	4.1.3. Namespace (referencing environment) [13]:				
	– типи namespace	–	+ [14]	+ [14a]	+ [14a]
	4.1.4. Address of a storage area [15]:				
	– variable's address, address alias	+	+	+ [15a]	+ [15a]
	– address type: static memory, register, stack, and heap (див. також "Lifetime")	+	+	+	+
	4.1.5. Linkage [16] of names:				
	– external	+	+	– [16]	+ [16]
	– internal and none	+	+	+	+
	4.2. Значення (values, objects [17]):				
	4.2.1. Types [18] (або data format):				
	– різноманіття вбудованих типів даних (build-in types)	див. табл. 3.1			
	– data declaration and definition [19]; forward declaration	+	+	+ [19a]	+ [19a]
	– name type compatibility (anonymous type names, type name alias)	+ [20]	+ [20]	– [21]	– [21]
	– type checking, conversion (див. також п. 5. Expressions / type conversion)	+	+	+	+ [22]
	4.2.2. Lifetime [23] або storage classes (categories of variables according to their lifetime):				
	– static [24]; stack-dynamic (are allocated from run-time stack, local variables)	+	+	+	+
	– explicit or implicit heap-dynamic; fixed	+ [25]	+ [25]	+ [25]	+ [25a]
	4.2.3. Initialization:				
	– Initial values for different storage classes [26]	+	+	+ [26a]	+ [26a]
4a	Data; constants (вживається також термін literals [27]):				
	– 4.1.a. Named constants (manifest constants: const та final)	+ [28]	+	+ [28]	+
	– constants with scope and /or dynamic binding of values [29]	–	+	+	+
	– 4.2.a. Typed constants (build-in types of constants – див. табл. 3.2)	+	+	+	+
4b	Data; labels.	+	+	–	+
5	Вирази (Expressions: operations (operators) with typed variables or constants):				
	– general notions:				
	– operators:				
	– rules: precedence, associativity; side effect; "l-values" and "r-values" [30]	+	+	+	+

Продовження табл. 3

1	2	3	4	5	6
	– assignment statement:				
	– simple assignment (x = b;)	+	+	+	+
	– multiple assignment (x = y = b;)	+	+	+	+
	– conditional assignment (_ ?Op1:Op2_)	+	+	+	+
	– compound assignment (+= –= *= /= %= <<= >>= &= ^= |=)	+	+	+	+
	– arithmetic expressions:				
	– arithmetic operators: (+ – * / % Op++ Op––)	+	+	+	+
	– operands: numeric types as well as some others	+	+	+	+
	– relational expressions:				
	– relational operators: (> < >= <= == !=, а також is та as для C#)	+	+	+	+
	– operands: numeric types as well as some others	+	+	+	+
	– Boolean expressions:				
	– logical (Boolean) operations (short-circuit evaluation): (&& || !)	+	+	+ [31a]	+ [31a]
	– operands: Boolean type	+	+	+	+
	– Bitwise Boolean expressions: (& | ~ ^ << >>)	+	+	+ [31]	+ [31a]
	– операція взяття адреса & та значення за адресом (dereference) *Op; boxing / unboxing	+	+	+	+
	– coma operator: (,)	+	+	+	+
	– overloaded operators:				
	– overloaded operators for build-in primitive types (arithmetic +, / etc. for int, float, etc)	–	+	+	+
	– overloaded operators for user-defined types (string, arrays, etc.)	–	+	+	+
	– перетворення типів (type conversion; див. також вище п. 4.2.1. Types):				
	– явне (explicit), тобто casting: (int), (float), ... та as для C#	+	+ [32a]	+	+
	– неявне (implicit); issues [32]	+	+	+	+
	– typeof, sizeof, sizeof( ), instanceof, checked, unchecked оператори [33]	+	+	+	+
	– object creation / deletion operations	–	+	+	+
	– доступ до елементів об'єкта складного типу:				
	– елемент масиву: [ ] або [ ][ ]	+	+	+	+
	– член структури, union, об'єкту класового типу: . .* -> ->* : : base	+ [34]	+	+ [34a]	+ [34a]
	– елемент в межах namespace: with	–	+	+	+
	– exception handling (overflow, underflow, divide by 0): try-catch-throw оператори	–	+	+	+
6	Оператори керування обчисленнями (control flow operators або statements):				
	– compound statement [35]	+	+	+	+
	– selection statements:				
	– two-way selection statement (*if / then / else*)	+	+	+	+
	– multiple selections (*switch / case / break / default*)	+	+	+	+ [36]
	– iterative statements:				
	– counter-controlled loop [37]: *for* (expr_1; expr_2; expr_3) {statement}	+ [37a]	+ [37b]	+	+
	– logically controlled loops (*while / do*):				
	– *while* (expression) {statement}	+	+	+	+
	– *do* {statement} *while* (expression)	+	+	+	+
	– how a single loop or several nested loops can be exited?: *break / continue*	+	+	+	+
	– iterations based on the data structure: *iterator*; foreach / in	–	–	+	+ [38]
	– unconditional branch statement (*goto* label)	+	+	– [39]	+
7	Функції, визначені користувачем (user defined functions) [40]:				
	– function prototype (or signature):				
	– parameter passing: by value, by address, by name (reference); return value	– [40a]	+	+	+
	– variable number of parameters	+	+	–	+
	– parameter initial values	–	+	–	–
	– functions as parameters	+	+	–	+ [40b]
	– variety of function organization:				
	– global functions (local and non-local variables, side-effect)	+	+	– [41a]	– [41a]

одиниця послідовності обчислень;
5) вираз (expression) — конструкція, що створює значення; 6) посилання (reference) — засіб доступу до значень; 7) маркер (token (keyword, identifier)) — молекулярна синтаксична одиниця; 8) символ алфавіту (character) — атомарна синтаксична одиниця. В даному пункті таблиці розглядається найвищий рівень такої ієрархії — рівень "програмної одиниці компіляції". Інші ієрархічні рівні програмних елементів розглядаються в даній таблиці в інших відповідних розділах запропонованої тут систематизації.

[2] Головна функція (main-function) викликається на початку роботи програми (program start-up). В одному або декількох класах C# може бути декілька функцій *Main()*, хоча і різного прототипу, але лише одна з них викликається на початку.

[3] З точку зору компілятора кожне речення (sentence) є послідовністю tokens (lexemes, terminals), розділених сепараторами (separators): blank character, tabs, carriage-return, line-feed, form-feed, comment. Подалі tokens поділяються за їх семантикою на групи: identifiers (names), constants, operations (operator), keywords (control flow statements, declarators, etc.), punctuators. Речення у всіх мовах закінчуються пунктуатором ";". (Нагадаємо, що в мові Pascal, на відміну, наприклад, від C, символ ";" вживався для розмежування операторів мови, тобто як "separators", а не як ознака кінця речення-statements, тобто не як "delimiter".) Кожний token складається із символів наперед визначеного алфавіту: стандартний набір ASCII символів для tokens, спеціальні послідовності символів для керування введення/виведення (escape sequences), стандартний та нестандартній набір ASCII символів для коментарів. В Java та C# підтримується також набір Unicode символів (Unicode character set).

[4] В мові ANSI C дозволяються багаторядкові, але не вкладені коментарі, що вводяться за допомогою пари символів /* та */. В мовах C++ / Java / C# додатково вводять однорядкові коментарі "//". І нарешті, в C#, крім цього, надається також можливість коментувати текст символом "////" для подальшого його використання в XML-файлах.

[5] В Java-програмі "директив" та "прагм" немає; проте ключове слово ***import*** (а далі слідує ім'я ***package***) можна розглядати як певний аналог директиви "#include" в C/C++.

[6] В C#-програмі, як і в C/C++, є багатий набір "директив", наприклад ряд директив для умовної компіляції, таких, як #define, #undef, #if, #elif, #else, #endif, а також для іншого призначення: #line, #error, #region, #endregion. Ключове слово ***using***, а далі слідує ім'я ***namespace***, можна також розглядати як певний аналог директиви "#include" в C/C++. Крім того, C# надає механізм для визначення декларативних тегів, що називаються атрибути (***attributes***), якими можна позначити необхідні фрагменти програми, щоб ввести опис певної бажаної додаткової інформації. Потім, під час виконання програми, ця інформація може бути використання за допомогою механізму, що називається ***reflection***. Так що концепцію attribute в C# можна також вважати певним різновидом директив компілятора.

[7a] Імена (names) використовуються для посилання на конструкції дані, декларованих у

програмі. В Java/C# такими конструкціями є пакет або область імен (package/namespace), класовий тип (class type), інтерфейс (interface type), члени такого типу даних, як посилання (reference type), параметри функцій або локальні змінні. Коротко кажучи, *імена* вводяться для *пакетів*, *типів* та *змінних*. В останньому випадку — для імен змінних — використовується також термін "*ідентифікатори*". В мові ANSI C ідентификатор позначає: 1) об'єкт, 2) функцію, або 3) одну з наступних конструкцій: 3.1) тег структури (structure), об'єднання (union), або перерахування (enumeration); 3.2) член структури, об'єднання, або перерахування; 3.3) ім'я нового типу (typedef name); 3.4) ім'я мітки (label name); 3.5) ім'я макросу (macro name); 3.6) параметр макросу (macro parameter). В ООП імена (ідентифікатори) вводяться також для класів та пов'язаних з ними конструкцій. Ідентифікатор називають простим, у випадку простих (не структурованих) змінних та кваліфікованим (qualified) — у випадку імен-членів складних об'єктів, як-то структури, класи тощо. Кваліфікований ідентифікатор є послідовністю ідентифікаторів, розділених символом *крапка* "."

Для того щоб компілятор міг "працювати" з ім'ям змінної (тобто ідентифікатором) в програмі, ім'я повинно бути доповнено інформацією про певні його атрибути та відповідне йому значення. Тобто повна інформація про ідентифікатор в компіляторі представляється як трійка-сукупність трьох відомостей: Name / Attributes / Value. При цьому атрибути характеризують такі властивості ідентифіксатора, як [21]: 1) тип (*type*), тобто дозволений діапазон значень даних для цього типу; 2) як і коли він створюється (*life-time*); 3) діапазон його видимості (*scope* of its name); 4) тип пам'яті, де зберігається його значення (*storage* class). Як правило, змінні вводяться в програму за допомогою механізму, відомого як декларація (*declaration*). Встановлення відповідності між ідентифікатором та його атрибутами та значенням називається ***binding*** ("прив'язка"). Процес ***binding*** для деяких атрибутів може відбуватися відразу при декларації змінних на етапі компіляції, для деяких інших атрибутів — на етапі виконання програми. Значення можуть бути надані змінним за допомогою процесів: 1) ініціалізації (*initialization*), тобто тоді, коли змінна створюється; 2) оператора присвоювання (*assignment*); 3) передачі (*transmission*) значення до змінної при операції введення (*input*).

[7] В Java/C#-програмі допустимі символи для ідентифікаторів вибираються не з ASCII, а з Unicode набору для букв (letters) та цифр (digits); В C# вони також не повинні включати такі спеціальні символи, як "#" чи "$".

[8] В C#-програмі, як і в Java, ідентифікатори не повинні співпадати з ключовими (зарезервованими) словами (див. відповідні таблиці ключових слів у документах по специфікації мов [26–30]), а також назвами булевих літералів (*true* та *false*) та *null*; проте в C# додатково дозволяються імена, що складаються із символу "@" на початку та наступного за ним ключового слова, наприклад "@if".

[9] В ANSI C-програмі максимальна довжина ідентифікатора складає 6 або 31 символів алфавіту, в MS Visual C++ — 247.

[10] Концепція *scope* означає діапазон операторів тексту програми, у якому змінна є видима, тобто може бути використана. Для того щоб змінна була видима в певному діапазоні, вона повинна бути в ньому задекларована.

[11] Існує чотири типи діапазону видимості: 1) блок, 2) функція, 3) прототип функції, та 4) файл. Діапазон видимості файлу, що містить main-функцію, називають також глобальною областю видимості (*global* scope).

[12] На доповнення до вказаних вище в мовах C++ / Java / C# вводиться також п'ятий вид "області дії" — класовий (class scope).

[12a] В мовах Java / C# змінні некласового типу або функції, що не є членами класу, не підтримуються, тобто не можуть мати глобальної області видимості (global scope level).

[13] Традиційно в різних мовах програмування виділяють різні набори програмних елементів, що можуть мати імена [22]: 1) змінна; 2) тип, визначений користувачем; 3) константи вбудованих типів; 4) константи типів, визначених користувачем; 5) базові операції; 6) мітки (labels) операторів (свого роду імена операторів); 7) формальні параметри; 8) підпрограми; 9) виняткові ситуації. В мовах C / C++ / Java / C#, якщо більше ніж одна декларація певного ідентифікатора є видимою в певній точці програми, така багатозначність розв'язується на основі синтаксичного контексту, в якому знаходиться посилання на даний ідентифікатор. Таким чином, існують відмінні області імен (*name spaces*) для різних категорій ідентифікаторів, наприклад: імена міток; імена тагів структур, об'єднань та перерахувань; імена членів структур чи об'єднань. Фактично, name space — це свого роду внутрішнє (для компілятора) ім'я масиву, в якому зберігаються імена ідентифікаторів. Тобто для різного виду ідентифікаторів використовуються різні масиви: один — для імен структур, один — для імен міток, один — для імен звичайних змінних. Для кожного з таких типів ідентифікаторів *name space* в C++ означає область декларації (declarative region).

[14] В C++/C# існує також ключове слово *namespace*. По суті, воно відображає подальший розвиток концепції scope, є наступним рівнем абстракції після class scope. Воно є аналогом ключового слова *package* в Java. Конструкція namespace створює область декларації (declarative region), яка приєднує додатковий ідентифікатор до кожного імені, об'явленого в межах цієї області.

[14a] В Java / C# немає глобальних змінних, але є global namespace; в Java йому відповідає unnamed package.

[15] Потрібно розрізняти поняття "адреси значення змінної" та "адреси ідентифікатора", оскільки можуть бути однакові імена (що мають різну область видимості — scope) і, значить, вони будуть мати різні адреси. В той же час деяка величина може зберігатися за певною адресою, на яку можуть вказувати також деякі інші змінні-вказівники (pointers або references). Таким чином, ідентифікатор змінної має синоніми (alias).

[15a] В Java / C# хоч і немає вказівників (pointers), використовуючи reference type, можливо зробити так, щоб дві змінні посилалися на той самий об'єкт.

[16] Можна зробити так, щоб один і той же ідентифікатор, об'явлений у різних областях видимості, міг посилатися на одне й теж саме значення змінної (чи

об'єкта) або функції. Процес, що це забезпечує, називається лінковка (*linkage*). В мовах Java ключове слово *"external"* відсутнє, але поняття, подібні до external linkage, використовуються (див. наприклад, "імпорт пакетів").

[17] Об'єкт (*object*) є іменована область зберігання даних в обчислювальному середовищі; зміст цієї області може представляти значення (див. також *l-values* нижче). За винятком бітових полів (bit-fields), об'єкти складаються з неперервної послідовності одного або декількох бітів, кількість, порядок та внутрішній формат яких є або явно описаним, або визначається неявно реалізацією [26]. Якщо ідентифікатор об'єкта чи змінної був об'явлений з локальною областю видимості (тобто в блоці або функції), тоді об'єкт чи змінна звичайно називаються "локальними"; у протилежному випадку вони "глобальні" в мові C/C++ та "нелокальні" в Java/C#. (Більш детальніше про об'єкти в C++/Java/C# див. нижче в табл. 4, 5.) Див. також коментар 1 до табл. 3.1.

[18] У кожній мові розглядається певна кількість вбудованих типів даних (build-in types — див. табл. 3.1) та надається можливість конструювати нові, визначені користувачем (user-defined) типи.

[19] Для кожної змінної величини (variables) або об'єкта (object) розглядається поняття *declaration* (див. також коментар 7a) та *definition*. *Декларація* вводить до розгляду імена змінних (тобто ідентифікатори) та їх типи і, як правило, не резервує для них пам'ять, значить, ніяк не визначає їх початкові значення. (Для уточнення див. поняття змінних з *global scope*, а також *static memory*.) *Дефініція* змінних резервує для них пам'ять визначеного (типом даних) формату. У всіх чотирьох мовах декларація є явною.

[19a] В Java/C# немає необхідності у forward declaration, оскільки порядок декларації є несуттєвим.

[20] Конструкція, наприклад "struct *tagStruct* {int i;} s1;", вводить до розгляду новий (структурний) тип *tagStruct*. Якщо в такій конструкції ім'я *tagStruct* буде опущено, то такий тип називається *анонімним*. В мові C за допомогою оператора *typedef* можуть бути введені до розгляду синоніми імені типу *tagStruct*. В C++ ім'я *tagStruct* та ім'я типу є синонімами по замовчуванню.

[21] В Java/C# немає оператора *typedef*. В той же час ідея *синонімів* імен типу (type name alias) також присутня, але в трохи іншому вигляді: імена вбудованих типів (int, float, і т.д.) є синонімами певних, наперед визначених в системі структур (наприклад, System.Int32 тощо). Нові типи ж вводяться за допомогою опису нового класу (class) — див. нижче розділ ООП, і таким чином потреба в конструкції *typedef* відпадає.

[22] В C# додатково до звичайних правил type conversion, що присутні у всіх чотирьох мовах, існує також конструкція "boxing/unboxing" для проведення перетворень між даними value type та reference type.

[23] Поняття часу життя (*lifetime*) змінної чи об'єкта означає той час, коли змінна чи об'єкт є асоційованими (bound) до певної області пам'яті.

[24] Поняття статичної (*static*) змінної означає, що її значення залишаються дійсними, тобто зберігаються в пам'яті, за час, що проходить між різними викликами функцій, в яких ця

змінна знаходиться. Іншими словами, статична змінна має глобальний час життя (іноді кажуть також history non-sensitive).

[25] У мові C для виділення пам'яті динамічним змінним використовуються функції alloc(), malloc(), realloc(). В мовах C++ / C# / Java введено нове ключове слово new та слово delete в мові C++. Динамічні змінні є асоційованими (bound) з областю динамічних даних, що часто називається "кучею" (heap), тільки тоді, коли цим змінним присвоюється деяке значення.

[25a] В C# будь-який динамічний об'єкт можна позначити за допомогою ключового слова fixed, для того щоб тимчасово перешкодити його вилучення із кучі засобом компілятора, що називається "збір сміття" (garbage collector).

[26] Для глобальних змінних класового типу порядок їх ініціалізації визначається порядком їх декларації (див. також коментар 12а). В мовах C++ / Java / C# для глобальних об'єктів, які мають в свою чергу члени-об'єкти, порядок ініціалізації не визначено мовою.

[26a] В Java / C# будь-яка змінна повинна мати початкове значення, перед тим як вона буде використана.

[27] Доцільно тут звернути увагу на походження слова "literal", яке (так само як і його синонім "literally") в англійській мові означає "буквально, дословно".

[28] У мові C конструкція manifest constants вводиться лише за допомогою директиви препроцесора #define і не існує ключового слова const, як в інших C-подібних мовах. В мові Java використовується ключове слово *final*, а також існує ключове слово const, але воно не використовується за призначенням і служить лише для контролю компілятором над вживанням цього зарезервованого слова як ідентифікатора.

[29] У мові C++ синтаксис для членів класу, що мають модифікатор const static, ускладнює декларацію символьних констант в середині цього класу. Зокрема, компілятор може не обрахувати (not bind) значення статичних констант-членів класу. Це означає, наприклад, що такі константи (тобто їх величини) не можуть бути використані для декларації розміру масиву. Щоб подолати такий недолік конструкції "статичні константи", можна використати тип enum. В C# константи по замовчуванню розглядаються як static члени і декларація констант не дозволяє використовувати static модифікатор.

[30] Терміни "l-values" (значення локатор виразу (locator expression)) та "r-values" (значення E2 виразу) походять від задачі опису виразу загального виду: E1 = E2. Конструкція l-value є вираз, що посилається на об'єкт (див. також концепцію "*об'єкт*" вище).

[30a] Позначення Op є скороченням слова "Operand".

[31] В мові Java додатково існують також операції >>>, >>>=, які означають *unsigned right shift* та *unsigned right shift assignment* відповідно.

[31a] В Java / C# операції *bitwise* operation називаються *logical* operation, а *logical* Boolean operation — *conditional* operation. Операндами *bitwise* operation можуть бути не тільки бульові, але і цілі типи: наприклад 0x03 | 0x05 як результат дасть 0x07.

[32] Перетворення *int* в *float* означає, що 9 десяткових цифр 4-х байт перетворюються в 7 десяткових цифр 4-х байт; перетворення *float* в *int* називають також

“coercion”. В Java ключове слово strictfp вказує, що вирази відповідають умові FP-strict.

[32a] В C++ дозволяється явне перетворення типів з використанням синтаксису, подібного до виклику функцій (function-style syntax): d = float (i); явне перетворення типів можна також задавати, використовуючи кастінг-синтаксис (casting). Попередній приклад, переписаний з використанням кастінгу, буде мати наступний вигляд: d = (float) i. Обидва випадки перетворень надають однаковий результат, коли перетворюються прості значення. Проте у випадку синтаксису функціонального стилю можна задавати більше ніж один аргумент для перетворення. Така відмінність є важливо для типів, заданих користувачем (user-defined types).

[33] В цих мовах існує різна комбінація набору таких операторів, але їх деталі тут не розглядаються.

[34] У мові C, оскільки вона не є мовою ООП, не підтримується доступ до об’єктів класового типу, статичних членів та scope resolution “: :”.

[34a] У мовах Java / C# існує тільки оператор “.”, оскільки в них немає типу вказівник (pointer).

[35] Складний оператор (*compound statement*) дозволяє представити (абстрагувати) набір операторів як один, більш абстрактний оператор; блок (*block*) є складним оператором, що, крім того, містить декларацію даних, тобто серед операторів якого є оператори декларації даних.

[36] В C# кожний оператор вибору (case) повинен включати оператор break. Крім того, в операторі switch керуючий параметр може бути не тільки типу int або enum, але і string.

[37] В кожній мові програмування при описі оператора циклу (loop statement) розглядається набір його характеристик (common features): 1) специфікація оператора циклу: ключове слово та наступна за ним синтаксична структура; 2) позиція тестування; 3) тип змінної керування (змінної циклу); 4) початкове значення: константне, змінне або задане виразом; 5) крок (прирощення) циклу; 6) верхня границя (граничне значення); 7) те, чи можна переходити граничне значення протягом виконання циклу; 8) спосіб виходу з циклу та/або спосіб пропуску кроку циклу; 9) те, чи зберігається значення контрольного параметру при виході із циклу та чому воно дорівнює.

[37a] У мові C тіло оператора циклу for виконується нуль або більше разів, доки виконується умова. Можна також, використовуючи додаткові вирази, ініціювати та змінювати значення змінних протягом виконання циклу for.

[37b] В мові C++ специфікація оператора циклу for може включати також дефініцію змінних; крім того, термінальний вираз може бути типу Boolean. Наступний оператор циклу for:

```
for (for-init-statement;
     expression1; expression2)
{
 // Statements
}
```

є еквівалентним оператору циклу while:

```
for-init-statement;
while( expression1 )
{
 // Statements
 expression2;
}
```

Якщо керуюча змінна циклу була продекларована у середині виразу for-init-statement, тобто з локальною областю видимості, тоді її значення не буде збере-

женим за межами циклу, тобто при виході із нього.

[38] В C # оператор foreach / in може застосовуватися не тільки до масивів, але і до будь-яких класів, що реалізують інтерфейс IEnumerable.

[39] В Java ключове слово *goto* є, але воно забороняється компілятором до використання.

[40] Використання процедур є важливими з наступних причин [23]:
1) економічність; 2) захист даних;
3) абстракція; 4) приховування реалізації (implementation hiding); 5) можливість нарощувати програму (extensibility); 6) модульність; 7) робота з бібліотеками.

1. *Економічність*: процедури дозволяють використовувати одну й ту ж частину коду декілька разів. Процедури є конструкціями, що надають ім'я для такої частини коду. Самі частини коду називають *тілом процедури*.

2. Захист даних (*protection*): в межах кожної нової процедури можна тепер організувати: новий діапазом видимості для роботи з іменами; новий режим виділення пам'яті для даних; нову локалізацію для обробки помилок.

3. *Абстракція*: дескриптивні імена процедур, такі, наприклад, як sort, дозволяють абстрагуватися від деталей реалізації та проектувати програму в термінах операцій предметної області чи задачі, що вирішується. Такий тип абстрагування отримав назву *process abstraction*.

4. Приховування реалізації, або *encapsulation*: алгоритм, ізольований в процедурі, може бути пізніше модифікованим, визвавши тим самим зміни лише в одному модулі процедури, а не в усій програмі.

5. Можливість нарощувати програму (*extensibility*): концепція процедур дозволяє нарощувати можливості мови за рахунок нарощення функціональних можливостей операторів та вбудованих функцій.

6. Модульність (*modularity*): процедури дозволяють розбити велику програму на менші модулі, надаючи базис для модульного програмування (яке іноді також називали "*structural programming*").

7. Робота з *бібліотеками*: стандартна колекція процедур може бути організована у бібліотеку.

Декларація процедури явно вводить до розгляду елементи або частини процедури:

1) *ім'я* процедури;

2) *тіло* процедури, яке складається з локальних декларацій та операторів мови;

3) *формальні параметри*, які є заготовленими місцями для підстановки в них *актуальних значень* (що іноді називають *аргументами*);

4) необов'язковий опис типу значення, що повертається (*result type*).

Історично так склалося, що процедуру, яка не повертає ніякого значення, називали підпрограмою (*subroutine*), а ту, що повертає певне значення, функцією (*function*). В мовах C / C++ / Java / C# в усіх випадках використовується термін функція. Тоді, коли функція не повертає ніякого значення, в цих мовах прийнято вказувати функцію, що повертає значення *void* ("пустий"). В Java / C# функції називаються *методами* (класу).

Сукупність імені функції та її формальних параметрів прийнято називати прототипом функції (*prototype* or *signature*).

Кожна функція має єдину "точку входу", тобто місце, з

якого починається виконання операторів функції — тіла функції. Кожне виконання тіла функції називають активізацією функції. Функція називається рекурсивною (*recursive*), якщо вона може бути активізована із середини її власного тіла, безпосередньо викликаючи саму себе, або опосередковано, через виклики інших функцій. Обчислення в тілі функції, що викликала іншу, тимчасово припиняються протягом виконання функції, що була викликана; по завершенні її роботи контроль завжди повертається в тіло функції, що викликала іншу. Це відрізняє підпрограми та функції від так званих копрограм (*co-routines*).

Якщо функція має параметри, розглядається декілька методів їх передачі:

1) за значенням (call-by-value);

2) за ссилкою (call-by-reference);

3) вхідні/вихідні параметри (call-by-value-result: in/out/in-out parameters);

4) за ім'ям (call-by-name), яке на сьогодні має лише історичне значення і було подібне до макро-підстановки).

[40a] У мові C передача параметрів за ссилкою реалізована з використанням вказівників (*pointer*).

[40b] У мові C# передача функцій як параметрів реалізована з використанням ключового слова *delegate*.

[41a] У Java/C# немає глобальних функцій (див. також коментарі 11, 12 та 12[a]).

[41] У мові C, оскільки вона не є мовою ООП, не підтримується class-member, inline та overloaded functions.

[42] У Java/C# generic та template functions не підтримується, у всякому разі у той спосіб, у який вони були задумані початково у C++.

Нижче наводяться стислі коментарі до табл. 3.1, пронумеровані цифрами від 1 до 13:

[1] У різних мовах програмування класифікація типів даних проводиться по-різному, використовуючи при цьому різну термінологію. Так, часто всі типи даних прийнято поділяти на *прості* та *складні*, або basic (fundamental) та derived, або primitive (embedded) та composed, чи scalar та aggregate, чи predefined та user-defined. В ANSI C концепція "type" поділяється на три підвиди: object types, function types та incomplete types. (Object type фактично означає value type, тобто це тип даних — змінних величин чи констант, оскільки в даному контексті термін object є синонімом до слова "значення" (value), тобто значення змінної величини. Див. також коментар 17 до табл. 3.) У мовах Java/C# введена інша класифікація типів даних: value type та reference type. Змінні reference type фактично відповідають таким змінним, пам'ять для яких повинна виділятися *динамічно*. Тому змінні такого типу "мають діло" скоріше з адресою пам'яті, де їх значення буде зберігатися, а не з самим значенням. Проте вони відрізняються від звичайних вказівників мови C (pointers) тим, що "робота з адресою", а тим самим і "динамічний характер" цих змінних, так би мовити, завуальовується, оскільки врешті-решт через ці адреси надається можливість працювати безпосередньо із значеннями величин цих змінних (подібно до reference type в мові C++).

Таблиця 3.1. **Типи даних в мовах програмування C, C++, Java 2 та C#**

	Типи даних (змінних величин) [1]	**C**	**C++**	**Java**	**C#**
1	Primitive data types (i.e. which are not defined in terms of other types):				
	– numeric (or arithmetic) supported directly by hardware:				
	– byte (with modifiers *signed* or *unsigned*)	– [2]	– [2]	1 [4]	1
	– integer (a string of bits) with modifiers *signed* or *unsigned*:				
	– short int (or ushort)	2 [3]	2	2 [4]	2
	– int (or uint)	4	4	4 [4]	4
	– long int (or ulong)	4	4	8 [4]	8
	– float (4 bytes)	4	4	4	4
	– double (8 bytes)	8	8	8	8
	– long double (8 bytes, but depends on the compiler implementation)	8	8	–	–
	– decimal (16 bytes; provides 28 significant digits)	–	–	–	16
	– char	1	1	2	2
	– bool (boolean)	– [5]	+	+	+
	– void	+	+	+	+
2	Predefined reference types:				
	– object	–	–	+	+
	– string (див. також нижче п. 3.3)	– [6]	– [6]	+ [6]	+ [6]
3	User defined types, i.e. derived types [7]:				
	3.1. Scalar types:				
	– pointer	+	+	–	+ [7a]
	– reference (а також delegate type в C#)	–	+	+	+
	3.2. Ordinal types:				
	– enumerations, допомогає уникнути помилок, пов'язаних з константами [8]	+	+	+ [8a]	+
	– interval type	–	–	–	+ [9]
	3.3. Aggregate types:				
	– array type [10]:				
	– багатовимірні масиви: [,]	+	+	+ [10a]	+ [10a]
	– "зубчаті" масиви (jagged arrays): [ ][ ]	– [11]	– [11]	+	+
	– string type (див. також вище п. 2)	– [6]	– [6]	+ [6]	+ [6]
	– structure type	+	+	–	+ [12]

В даній таблиці використовується систематизація, яка дозволяє провести порівняння типів даних всіх чотирьох мов, тобто як C/C++, так і Java та C#.

Тут цікаво також звернути увагу на те, що в Java та C# всі (базові) типи, включаючи value type, є похідними від класу *object* системної бібліотеки класів. В цьому розумінні вони, строго кажучи, не є базовими (basic, fundamental або primitive). Проте з точки зору прикладних задач їх все ж доцільно певною мірою вважати базовими типами. (Те, що всі такі "базові" типи тут є похідними від типу/класу *object*, забезпечує підняття рівня мови на ступінь вище, тобто далі від архітектури комп'ютера, що робить можливим більш легке перенесення (portability) програм на інші платформи.) Зауважимо також, що назва класу "object" є, на думку автора, трохи такою, що може "збити з пантелику", нав'язуючи занадто строгі асоціації з терміном "об'єктно-орієнтоване програмування" (хоч, без сумніву, ця асоціація тут має місце). Та все ж замість назви "object" для цього базового класу може було б більш доцільно вжити назву "value type", або "instance type", або навіть просто "value" чи "instance". В доказ достатньо лише розглянути декларацію цього класу, тобто його methods-members, і ми побачимо, що його конструктор та методи (public: Equals, GetHashCode, GetType, ReferenceEquals, ToString; protected: Finalize, MemberwiseClone) виконують звичайні, так би мовити, досить рутинні операції, які програмістам на C++ доводилось реалізовувати власноручно кожного разу, коли вони вводили до розгляду новий тип (type) даних — клас чи структуру і надалі мали працювати із динамічними значеннями (values, instances) цього типу. Тобто ніякої магії в класі "object" немає, проте така назва класу подекуди приводить читачів-програмістів до конфузів, оскільки вона певною мірою розмиває границю між поняттями *типу* даних (введенного класом) та *значенням* змінної (об'єкта) цього типу (хоч і створеним динамічно та з посиланням на нього за допомогою адреси). Проте, звичайно, якби автори C# вибрали за назву цього базового класу термін, наприклад, "value" (або навпаки — "class") замість "object", тоді замість терміну "object-oriented programming" прийшлось би вживати термін "value-oriented programming" чи "class-oriented programming", що навряд чи визвало б захоплення у прихильників ООП.

[2] У мовах C/C++ немає вбудованого типу даних byte, але він може бути легко змодельований з використанням *unsigned char*.

[3] У відповідній клітинці таблиці вказано кількість байт, що відводиться для представлення (зберігання) вказаного типу даних, наприклад 1 байт чи 8.

[4] У Java немає типу беззнакові цілі (*unsigned int, short або long*), а також знакового байта (*sbyte*).

[5] У мові Cнемає вбудованого типу даних bool, замість нього використовується тип int; змінна цього типу має приймати тільки два значення: 0 та 1.

[6] У мовах C/C++/Java немає вбудованого типу даних string; він моделюється як масив символів в C/C++ (і тоді для обробки такої змінної-string використовується вказівник на цей масив) або як class — в C++/Java. У мові C# тип string підтримується

і як базовий тип (синонім системного класу), і як class.

[7] При опису типів даних, заданих користувачем, звичайно розглядаються наступні питання: як зконструювати новий тип? (наприклад: [ ] – для масивів, ( ) – для функцій, * – для вказівників); які операції будуть надаватися для змінних такого типу (наприклад, операції доступу до членів структурованих даних: ".", "->", " : : ", тощо); як ці операції визначаються.

[7a] У мові C# тип pointer підтримується в режимі unsafety mode.

[8] Декларація типу enum визначає ім'я для групи логічно пов'язаних один з одним констант цілого типу, виражених символьними назвами. Тип enum часто використовується у випадку, коли потрібно організувати багаточисельний вибір серед кількості варіантів, наперед відомих на етапі компіляції.

[8a] У Java тип enum безпосередньо не підтримуються. Проте він може бути змодельованим з використанням класів.

[9] У мові C# тип interval наближено можна змоделювати, використовуючи тип enum, оскільки останній дозволяє задавати значення констант і тим самим їх діапазон.

[10] При описі масивів розглядають такі поняття, як type of array elements, subscripts (or indexes) та range checking. Зауважимо також, що іноді слово "масив" вживається в двох значеннях: ім'я типу і назва об'єкта такого типу. Наприклад, у виразі "int A[2][10]" мови C ім'я масиву A вживається у двох значеннях.

[10a] У мовах Java/C#, оскільки масиви є змінними типу reference, тобто динамічними змінними, то, як і для будь-яких динамічних змінних, їх дефініція складається із двох частин: виділення пам'яті певного розміру та визначення/ініціалізація значень елементів масиву у виділеній пам'яті. Проте синтаксис мови дозволяє певні скорочення у запису. Так, наступні конструкції є еквівалентними:

```
1) float [] Arr = new
   float [3]; Arr [0] = 1.0; Arr
   [1] = 2.0;
   Arr [2] = 3.0;
```

Таблиця 3.2. **Типізовані константи в мовах програмування C, C++, Java та C#**

Типи констант	C	C++	Java	C#
	Приклади:			
– numeric (supported directly by hardware):				
– integer with or without suffix l or L: and with or without suffix u or U	+	+	+	+
	+	+	–	+
– decimal	28 | 28**uL**	28 | 28**uL**	28 | 28**L**	28 | 28**L**
– hexadecimal	0x1C | 0x1C**uL**	0x1C | 0x1C**uL**	0x1C | 0x1C**L**	0x1C | 0x1C**L**
– octal	034 | 034**uL**	034 | 034**uL**	034 | 034**L**	034 | 034**L**
– float (4 bytes) with suffix f or F	28.0 | 28.0**F**	28.0 | 28.0**F**	28.0 | 28.0**F**	28.0 | 28.0**F**
– double (8 bytes) – unsuffixed constant	28.0	28.0	28.0 | 28.0**D**	28.0 | 28.0**D**
– long double with suffix l or L	28.0, 28.0**L**	28.0, 28.0**L**	–	–
– decimal (16 bytes)	–	–	–	28.0 |

2) float [] Arr = new float [3] {1.0; 2.0; 3.0};
3) float [] Arr = {1.0; 2.0; 3.0}.

[11] У мовах C/C++ "зубчаті" (jagged) масиви безпосередньо не підтримуються. Проте їх можна змоделювати, використовуючи масиви вказівників та динамічне виділення пам'яті для їх елементів.

[12] Хоч структури і класи багато в чому подібні — structs можуть реалізовувати інтерфейси та можуть мати такі самі види членів, як і класи, — потрібно пам'ятати, що в мові C# структури є value type, а класи — reference type, звідки і випливає відповідна відмінність у їх реалізації. Наслідування також не підтримується для структур.

[13] Більш детально про класи в мовах C++/Java/C# дивись нижче, в наступному розділі статті.

Як бачимо, наведений перелік імперативних концепцій та конструкцій мов дозволяє провести їх критичну оцінку та порівняльний аналіз, що, без сумніву, має бути корисним для кращого володіння певною мовою програмування. Крім того, він може бути корисним для тих, хто у своїй практиці програмування, в силу певних причин, володіє лише деякою підмножиною можливостей мови. У цьому випадку розуміння решти конструкцій та концепцій мови може надати негаданих переваг таким практикам.

У сучасній літературі та в Internet-джерелах (див., наприклад, MSDN) починають наводитися порівняльні таблиці конструкцій для ряду нових мов, таких, як C, C++, Visual Basic, Java, JavaScript, Visual FoxPro, C#, проте там такі порівняння проводяться в основному з позицій огляду граматичних (в основному — синтаксичних) конструкцій мов, тобто з позицій, коли мова представляється так би мовити з точки зору компілятора мови, але не так, як вона повинна представлятися з урахуванням того, що ця мова є інструментом опису прикладної задачі обробки інформації за допомогою комп'ютера, як програмно керованої системи. Тобто, у відомих з літератури порівняннях наводяться окремі таблиці для операторів керування (control flow operators), типів даних, виразів чи інших конструкцій, але не наводиться упорядковане подання та аналіз системи концепцій та конструкцій мов, як це зроблено в даній статті.

Зауважимо також, що в офіційних специфікаціях цих мов (див. [29, 30]) порядок викладення набору конструкцій навіть для таких споріднених мов, як Java та C#, є суттєво різним. Це наводить на думку, що на сьогодні все ще не існує загально прийнятого порядку розгляду переліку конструкцій мов програмування, і означає, в свою чергу, що ця проблема все ще не є остаточно вирішеною. В той же час у ряді випадків, особливо при вивченні нової мови, доцільно мати такий упорядкований набір конструкцій мови.

3. Еволюція об'єктно-орієнтованих конструкцій мов програмування

Як ми могли побачити з попереднього розділу, концепції імперативного програмування надають багаті можливості по програмному керуванню комп'ютером для вирішення широкого кола прикладних задач. Проводячи декомпозицію задачі на менш складні функціональні модулі, можна створювати програми, що вирішують складні задачі. В той же час підхід імперативного, функціонально-модульного програмування не надає ефективних засобів для об-

робки складних даних. Серед таких засобів, в першу чергу, необхідним є створення типів даних, визначених користувачем, — ***user defined types***, а також змінних таких типів, за допомогою яких (типів та відповідних їм змінних) можна було б набагато легше моделювати складну прикладну задачу, ніж використовувати вбудовані в мову програмування типи. В таких сферах застосування, як, наприклад, інформаційні системи для бізнесу, системи "штучного інтелекту" та деякі інші, складність програмної системи в основному пов'язана з даними, а не алгоритмами їх обчислення. Тут дані мають складну структуру і характеризуються складними взаємозв'язками між їх компонентами.

Концепція user defined types була розвинута далі й привела до поняття "*абстрактних типів даних*" (ADT), тобто таких, які, по-перше, визначені користувачем, а по-друге, для яких користувач встанавив також набір операцій чи функцій обробки змінних такого типу. (Нагадаємо, що, за означенням, тип даних — це множина значень та операції, визначені над нею.) Причому безпосередній доступ до компонентів таких типів тепер забороняється, і він стає можливим лише за допомогою наперед заданих функцій. Деталі реалізації самих даних виявляються схованими від користувача (data hiding); це дозволяє змінювати внутрішнє представлення даних без негативного впливу на модулі, що користуються такими даними (data encapsulation; інкапсулювати частину алгоритму означає локалізувати її в одній секції програми). Опис такого абстрактного типу даних може бути виконано як окремий модуль або компонент. Тоді програма проектується так, щоб вона складалась із модулів, які репрезентують абстрактні типи даних, тобто модель даних початкової задачі.

Звернемо ще раз увагу на те, що механізм абстрактних типів даних був націлений на вирішення проблеми складності не алгоритмів обробки даних, а саме *структур даних* в прикладних задачах.

Наступна проблема складності вже не вирішувалася ефективно навіть з використанням абстрактних типів даних. Виявилося, що існує велика кількість прикладних задач, в яких потрібно обробляти певну кількість *об'єктів* (змінних) абстрактного типу даних при виконанні додаткових умов, таких, наприклад, як наступні: об'єкти повинні створюватися та знищуватися динамічно; можуть існувати об'єкти, що мають підмножину характеристик існуючих об'єктів та певні власні відмінні риси; повинна забезпечуватись можливість різного виду ініціалізації значень об'єктів; при модуляризації програми на абстрактні типи даних потрібно забезпечити вирішення проблем "доступу" (scope) до компонентів, "лінковки" функцій-операцій, що оброблюють дані, тощо.

Для вирішення такого роду проблем у мовах програмування була запропонована концепція "клас", а програмування з використанням класів та об'єктів (екземплярів, змінних) заданого класу стали називати *об'єктно-орієнтованим програмуванням* (ООП). Таким чином, відбулося зміщення акценту уваги від process-oriented до data-oriented, а потім і до object-oriented програмування. Такий розвиток відбувся як розвиток концепції "абстрагування" обчислень, а точніше — абстрагування

Таблиця 4. **Об'єктно-орієнтовані конструкції мови C++, їх попередники та аналогії в інших мовах**

Об'єктно-орієнтовані конструкції мови C++	SIMULA-67 1964, 67	Modula, Euclid 1977	CLU 1973	ADA (1975, 83)	C++ 1982 -1985	SMALL-TALK-72 / 80
1. User defined data types, data abstraction, and encapsulation	class	module	cluster	package	class	object
2. Objects creation and initialization	+	+	+	+	+	+
3. Inheritance	+	+	–	–	+	+
4. Dynamical binding and polymorphism	–	+	–	–	+	+
5. Parameterized types	–	–	+	+	+	+
6. Exception handling	–	–	+	+	+	+

послідовності програмного керування. На шляху такого розвитку були створенні такі концепції абстрагування:

1) складний оператор та блок (compound statement and block) — для абстрагування послідовності обчислень та пов'язаних з ним декларацій змінних;

2) функція (a function) — абстрагування процесу обчислень (process abstraction);

3) абстрактний тип даних (abstract data type, ADT) — абстрагування реалізації форматів даних;

4) клас (class) в Simula 67.

В табл. 4. наводиться основний перелік об'єктно-орієнтованих конструкцій мов програмування та їх наявність в історичних попередниках мови C++. Більш детальний розгляд ООП концепцій та порівняння їх реалізацій в мовах C++ / Java / C# наводиться нижче, в табл. 5.

Нижче наводять стислі коментарі до табл. 5, пронумеровані цифрами від 1 до 13:

[1] Згідно означення, абстракція даних є специфікацією: 1) множини об'єктів даних; 2) множини абстрактних операцій над цими об'єктами; 3) інкапсуляція цього всього у такий спосіб, що користувач нового типу не може маніпулювати об'єктами даних напряму, а лише використовувати введені операції. Як бачимо, абстрагування даних означає те, що ми не звертаємо уваги на деталі реалізації даних в новому типі даних, нас цікавить більш абстрактний (менш детальний) рівень їх використання, тобто те, як використовувати ці данні, іншими словами, як проводити над цими даними операції (і нас не хвилює те, як ці операції були реалізовані).

[2] У мовах C++ / Java / C# члени класу можуть включати: 1) константи (тобто значення, що можуть бути вичислені на етапі компіляції);
2) змінні, які тут називаються полями (*fields*): static або non-static з або без атрибуту read-only в Java / C#);
3) функції, які тут називаються методами (*methods*); 4) оператори (члени, що визначають значення виразу-оператора (expression operator), який може бути застосованим до екземпляру класу, тобто змінної класового типу; 5) конструктори екземпляру класу (instance constructors), тобто члени, що реалізують дії, необхідні для ініціалізації екземплярів класу; 6) деструктори (за винятком мови Java; в C# деструктори екземплярів класу викликаються автоматично при "збиранні сміття" — garbage collection); 7) вкладені декларації типів (nested types declarations).

[2a] У C# члени класу можуть, крім того, включати: 8) статичний конструктор, тобто член, що реалізує дії, необхідні для ініціалізації класу як типу даних; 9) властивості (properties), тобто члени, що забезпечують до-

ступ до даних об’єкта або класу; 10) індексер (indexer), тобто член, що дозволяє організувати індексацію об’єкта у такий же спосіб, як це робиться у масивах; 11) події (events), тобто члени, що дозволяють передавати повідомлення (notifications) для об’єкта або класу. Методи, властивості та індексери можуть бути віртуальними (*virtual*); це означає, що їх реалізація може бути переписана у похідному класі. Клас може вказувати на те, що не всі його методи імплементовані, і він запроектований бути лише базовим класом для інших класів; у цьому випадку використовується модифікатор abstract. Такий клас називається абстрактним (*abstract class*). Абстрактний клас може також мати абстрактні методи (*abstract members*), тобто члени, які мають бути реалізованими у похідному класі (див. також табл. 5, п. 4).

[2b] У C# параметри функцій-методів класу можуть мати out, params (parameter array) та ref модифікатори.

[3] У Java змінні можуть бути позначені модифікатором transient, щоб вказати на те, що вони не будуть зберігатися на диск, методами, реалізованими системним класом object.

[4] Користувачеві не потрібно більше знати сховану реалізацію членів класу. Навіть більше — відтепер навіть не дозволяється безпосередньо використовувати сховані (hidden) дані.

[4a] У C# існує два додаткових типи доступу: internal та protected internal.

[4b] У C#, крім того, існує спеціальна конструкція, яка реалізує функції-властивості (property-functions).

[5] Оскільки об’єкти вводяться в програму шляхом опису їх імен (так само, як і імена змінних вбудованих у мову типів), то на імена об’єктів класу поширюється все те, що було сказано вище стосовно імен змінних — див. табл. 3.

Таблиця 5. **Базові концепції ООП в C++, Java 2 та C#**

	ООП конструкції мов програмування	C++	Java	C#
1	Data Abstraction [1]:			
	– class as a new *data type*:			
	– is a *user-defined type*	+	+	+
	– class members, i.e. a collection of logically related things (data aggregation) [2]:			
	– class data (*fields*), static versus class instance data; final (constant) fields	+	+ [3]	+ [2a]
	– class functions (*methods*)	+	+	+ [2b]
	– others: properties, indexers, events, etc.	–	–	+ [2a]
	– nested classes (as a nested data types); див. також *inheritance*	+	+	+
	– class as a new type of scope – *class scope*:			
	– the class scope is more powerful than block, function or file scope	+	+	+
	– there is no need to have the global scope any more	–	+	+
	– class as a *data hiding* and *access control* technique [4]:			
	– private, public, protected; internal, protected internal	+	+	+ [4a]
	– data access functions (so-called "properties", see also below)	+	+	+ [4b]
	– friends	+	–	–
	– class as an *encapsulation* mechanism (to gain the modularisation):			
	– is an ADT	+	+	+
	– is a compilation unit	–	+	+
	– package/namespace as a file and as a compilation unit	–	+	+
2	Objects creation and initialisation [5]:			
	– constructors and initialisation:			
	– instance constructors	+	+	+
	– static constructor	–	– [5a]	+
	– destructor	+	– [5b]	+
	– multiple instances of class data but single instance of class methods	+	+	+
	– constant objects and const member functions treatment	+	+	+
3	Inheritance, i.e. the extension of the basic class [6]:			
	– single inheritance	+	+ [7]	+ [7]
	– multiple inheritance	+	–	–
	– data access: public, private, protected, internal, protected internal	+	+ [8]	+ [8]
	– base class, derived classes, abstract class, member classes, nested classes [9]	+	+	+
	– *final / sealed* class	–	+	+
	– scope resolution operator “: :”	+	–	–
	– the class hierarchy common for all classes; the ultimate base class ”object”	–	+	+
4	Overridden class methods:			
	– statically and for the same class (overloading)	+	+	+
	– dynamically and for the classes of different level of class hierarchy:			
	– polymorphism (overriding); virtual functions; delegates	+	+ [10]	+
	– *abstract classes* [11]; final methods; (див. також нижче)	+	+	+
5	Methods of particular type or special purpose [12]:			
	– constructors (see also above)	+	+	+
	– destructor (see also above)	+	– [5b]	+
	– property, indexer, event constructs	–	–	+

Конструктор — це функція, яка не повертає ніякого значення; вона також не може бути викликана напряму з програми користувача. Якщо клас використовується лише для абстракції та організації агрегації даних, то немає великої необхідності створювати конструктор. Проте він потрібен, якщо клас включає в себе динамічні дані.

Клас може бути спроектованим таким чином, щоб заборонити створення екземплярів класу в будь-якій частині тексту програми, що не входить в опис самого класу. У цьому випадку необхідно об'явити хоча б один конструктор, щоб запобігти створення неявного конструктора, а також об'явити всю решту конструкторів з модифікатором private.

Ініціалізація виконується для об'єктів класу, об'єктів-членів класу та (під-)об'єктів базового класу.

[5a] У Java безпосередньо немає конструкції "static constructor", але є механізм ініціалізації даних класу (саме класу, а не об'єкта класу), до множини яких входять члени класу, об'явлені з модифікатором *static*. Статичні ініціалізатори (static initializers) — це блоки коду, що використовуються, щоб допомогти проініціалізувати клас, коли він вперше завантажується в пам'ять.

[5b] У Java безпосередньо немає конструкції "destructor", але є механізм *finalize*.

[6] Зробимо деякі зауваження стосовно термінології. Для позначення базового класу (base class) в Java вживається термін "super-class", на відміну від C++ та C#. Для позначення похідного класу (derived class) в Java вживається термін "subclass". Термін "subclass" в Java, на думку автора, доцільніше було б замінити на термін "supra-class", тобто "надклас". У той же час при утворенні похідного класу в Java використовується ключове слово "extended", що правильно підкреслює відношення "включеності-належності" елементів базового класу до множини елементів похідного класу.

Зауважимо також, що C++ слідує об'єктно-орієнтованому підходу, введеному в мові Simula, а не Smalltalk. В Smalltalk весь набір методів об'єкта називають message protocol. Це вводить в термінологічне протиріччя із використанням слова message в інших контекстах, наприклад при обміні повідомленнями між модулями операційної системи і, як наслідок, при програмуванні у MS Visual C++ з використанням бібліотеки класів MFC. В той же час в програмах, написаних на ANSI C для Windows, використовується також термін *клас*, який так само ні якого відношення не має до того значення цього терміну, яке використовується в C++ при ООП. (Для більш детального знайомства з деякими особливостями ООП з використанням C++, а також про можливі пастки, що виникають при цьому, можна ознайомитися в роботі [25].)

[7] Усі базові класи в Java та C# походять від системного класу object.

[8] У Java ключове слово *protected* створює члени класу, які можуть бути доступні тільки для підкласів (sub-classes), що розташовані зовні даного пакету (package), а також з будь-якого місця, що знаходиться в середені даного пакету. В цьому відмінність *protected* членів в Java та C++.

[9] На відміну від C++, де класовий тип є типом, об'явленим з global або file scope, в Java та C# клас може

бути об'явленим як *global class*, *local class*, *static* (sic!) *class*. Клас-член (*member class*) — це такий клас, декларація якого безпосередньо розташована у даному класі. Вкладений клас (*nested class*) це такий клас, декларація якого розташована у будь-якому місті, включаючи підкласи даного класу. Нестатичні вкладені класи називають також внутрішніми (*inner*). Тобто опис *member* класів розміщується в області декларації класу (в header-file у випадку C++), а опис *nested* класів може розміщуватися в тілі функцій-членів класу, чи у деякому місті базових класів.

[10] Якщо два методи одного класу (неважливо при цьому, чи вони об'явлені в тому самому класі, чи в різних наслідуваних класах однієї класової ієрархії) мають однакове ім'я, але різну сигнатуру, такі методи називаються перевантаженими (*overloaded*). У випадку перевантаження (overloading) функції базового класу (base class) функцією похідного класу (derived class), його називають домінуючим перевантаженням (*overriding*). У Java, якщо клас декларує, наприклад, два методи з однаковим ім'ям і похідний клас (subclass) перевантажує (overrides) одну з них, похідний клас (subclass) також наслідує ще другу функцію. У цьому відношенні Java відрізняється від C++. Тобто похідний клас в Java буде мати дві функції, а в C++ — одну! (Хоча друга функція завжди може бути викликана через вказівник на об'єкт базового класу.) З поняттям overriding тісно пов'язане поняття поліморфізму (polymorphism), тобто динамічного зв'язування (dynamical binding) функцій, які є перевантаженими (overridden), та передачі цих функції як параметрів в іншу функцію за рахунок використання вказівника на функцію.

[11] Абстрактні методи — це функції, які об'явлені, але не реалізовані в даному класі.

[12] Крім функції спеціального призначення в класах вводиться також спеціальна конструкція this. Ключове слово this є вказівником на область пам'яті, де розташовані дані об'єкта і на яку функції-члени класу мають область доступу. Цей вказівник завжди (невидимо) передається як параметр до функції. Він відіграє таку саму роль, як і вказівник на структуру struct, що передається параметром до функції.

[13] Декларація інтерфейсу вводить новий посилальний тип; його членами є класи, інші інтерфейси, константи та абстрактні методи. Інтерфейси можуть також містити властивості, індексери та події. Інтерфейс може бути безпосереднім розширеням (*extension*) одного чи декількох інших інтерфейсів і таким чином наслідувати зміст інших інтерфейсів. Всі члени інтерфейсу неявно мають доступ public. Реалізація інтерфейсу проводиться в класі, в декларації якого використовується ключове слово implements, що указує далі на інтерфейс, який має бути реалізованим цим класом.

Деякі деталі, не пов'язані з ООП-концепціями мов програмування, такі, наприклад, як конструкція *native*, *synchronized* тощо в мові Java, в даній статті не розглядаються, оскільки скоріше відносяться до проблем, пов'язаних з організацією взаємодії прикладної програми з операційною системою та вирішенням інших проблем такого роду.

4. Критичний аналіз еволюції базових конструкцій мов

Як зазначалося вже вище (в п. 2), у сучасній літературі та

в Internet-джерелах починають наводитися порівняльні таблиці конструкцій для ряду нових мов, проте такі порівняння проводяться в основному з позицій огляду граматичних конструкцій мов і не упорядковане подання та аналіз системи концепцій та конструкцій мов. Систематизоване подання дозволяє провести аналіз еволюції базових конструкцій мов.

Для проведення такого аналізу достатньо прослідкувати кожний рядок табл. 3, 3.1, 3.2 та 5, та побачити, чи еволюціонувала відповідна конструкція/концепція мови (представлена у рядку таблиці) від мови C до мови C++ і потім далі до Java та C#, а чи вона "відмерла" — її доцільність була заперечена практикою.

Так, наприклад, можна побачити, що конструкція header files була відхилена як непотрібна в мовах Java та C#. Проте ми пам'ятаємо, що ця конструкція вважалась у свій час досить прогресивною і однією з відмінних рис C та C++ у порівнянні їх, наприклад, з Pascal — свого роду конкурентом C на той час.

Інший приклад — тип даних enum. Конструкція, яка була свого часу введена в C, а потім і в C++ та вважалась досить елегантним засобом мови, раптом була відкинута як непотрібна в мові Java. В той же час в мові C#, що є, як ми бачимо, дуже подібною до Java, тип даних enum знову отримав право на життя!

Те саме можна сказати ще про пару конструкцій цих мов програмування — label та "оператор з поганою репутацією" goto.

Іншою особливістю еволюцій мов, природно, є розширення набору їх концепцій та конструкцій, а також "остаточне" позбавлення від (наслідування) конструкцій, що були чи виявилися невдалими в мові C чи C++. Так, наприклад, типи даних union та bit field не підтримуються більше ні в Java, ні в C#. Конструкція template була введена в C++ як ефективний засіб параметризації класів та функцій, проте потім — в Java та C# — від неї відмовилися. Те саме стосується конструкції typedef — з введенням та розвитком концепції класів в Java та C# потреба в typedef відпала повністю.

Окремої уваги також заслуговує поняття вказівника (pointer) в мові C та його подальша еволюція. В C pointers були чи не найвідмітнішою рисою мови. Вони надавали їй гнучкості та ефективності, що вплинуло на поширеність використання мови як потужного інструмента розробки не тільки системних, але і прикладних задач. У той же час конструкція pointer мала певні недоліки. За словами відомого спеціаліста в області теорії мов програмування Ч. Хоара, "їх інтродукція в мови програмування високого рівня була кроком назад, від якого ми можемо ніколи не отямитися" (1973). Невдалим в мові C також слід вважати синтаксис для опису pointers: використання символу "*" приводило до неоднозначності, оскільки він використовувався також для позначення операції множення, тобто його вживання було контекстно залежним. Всі переваги та недоліки pointers були повністю збережені в мові C++. В той же час в нових мовах Java та C# конструкція pointers відсутня. (В C# дозволяється працювати з pointers у так званому "не небезпечному режимі" — unsafe mode.) Ось такий яскравий приклад "природного відбору" в еволюції мов програмування.

Такі приклади можна продовжувати. Читач запрошується проробити досліди самостійно.

Ще одним джерелом чи, так би мовити, рушійною силою прогресу у розвитку мов програмування були проблеми програмної інженерії (software engineering) та, зокрема, технологій побудови програмних компонентів, які можна було б повторно використовувати (reusable components) і які надавали б можливість їх взаємодії (component interoperability). Потреба у вирішенні такого типу проблем врешті-решт відобразилася в конструкціях нових мов програмування. Проте на шляху цієї еволюції була спочатку розробка так званої моделі компонентних об'єктів — COM. Згідно COM, зокрема, для того щоб об'єкти могли взаємодіяти один з одним, були компонентами один одного (компонентними об'єктами), всі вони повинні мати деяку "спільну базу", "спільний набір правил взаємодії". Така спільна основа полягає в наборі положень:

- програмний об'єкт повинен бути *об'єктом* у розумінні об'єктно-орієнтованого програмування, тобто мати структуру даних та набір методів, що визначають перелік того, що саме може робити об'єкт;
- доступ до даних об'єкта повинен відбуватися тільки через його методи (наприклад, Say/Get); логічно пов'язані один з одним методи повинні утворювати так званий *інтерфейс* (іншими словами, частина методів об'єкта об'єднується в групу і така група називається інтерфейсом об'єкта);
- можна розглядати інтерфейс як окремий абстрактний клас, у якого не має власних даних і є лише набір чисто віртуальних функцій, що декларують набір функцій методів-членів компонентного об'єкта, реалізація цих функцій знаходиться в тілі самого об'єкта;
- з метою забезпечення можливості повторного використання компонентів інтерфейси повинні мати механізм, подібний спадковості класів (так звані containment/delegation та aggregation);
- після того як інтерфейс був опублікований і почав десь працювати, його не можна змінювати, добавлення нової чи зміна існуючої функціональності потребує визначення повністю нового інтерфейсу (так розв'язується проблема керування новими версіями програмного продукту — versioning problem).

Як бачимо, деякі з вимог COM знайшли своє відображення у нових конструкціях мов, таких, як інтерфейс (Java та C#), properties (C#) тощо.

Висновок

Проведений порівняльний аналіз базових концепцій сучасних мов імперативного та об'єктно-орієнтованого програмування та їх еволюції може надати читачеві додаткову інформацію для кращого розуміння мов та певну теоретичну підготовку до сприймання нових ідей з теорії мов програмування, що можуть з'явитися у майбутньому. Запропоновані таблиці, крім, так би мовити, "теоретичної цінності", дають також практичну користь: вони можуть бути використані на практиці при вивченні мов програмування C, C++, Java 2 та C#. Для цього автор рекомендує, за його прикладом, створити певний репозитарій зразків (sample) програм на основі табл. 3–5. Такі програми мають бути невеличкими за розміром і реалізовувати

лише одну або ряд споріднених концепцій із вказаного в таблицях набору. Вивчення мови програмування за такою схемою придасть певної систематизації знань конкретної мови програмування, неминуче забезпечить їх повноту, що все разом покращить розуміння матеріалу і підвищить якість його знань. Табл. 3–5 можуть бути також використані при формулюванні запитань для контролю цих знань.

1. *Blaauw G., Brooks F.P.* Computer architecture: concepts and evolution. — Addison Wesley Longman, Inc., 1997. — 1213 p.
2. *Knuth D.E., Pardo L.T*. The early development of programming languages // A history of computing in the twentieth Century: A collection of essays. — Academic Press. 1980. — P. 197–273.
3. *Sammet J.E.* Some approaches to, and illustrations of, programming language history // IEEE Annals of the History of Computing J. 1991. — Vol. 13. — N 1. — P. 33–50.
4. *History* of programming languages: Proc. of the History of programming languages Conference, Los Angeles, CA, June 1–3, 1978 / Ed. By Wexelblat R.L. — Academic Press, 1981. — 758 p.
5. *ADA:* past, present, future; Jean Ichbiah // Commun. ACM. — 1984, 27, 10 Oct. — P. 990–997.
6. *Wirth Niklaus*. From programming language design to computer construction // Ibid. — 1985, 28, 2 Feb. — P. 160–164.
7. *Sammet J.E.* Why Ada is not just another programming language // Ibid. — 1986, 29, 9 Sep. — P. 722–732.
8. *Cohen J.* A view of the origins and development of Prolog // Ibid. — 1988, 31, 1 Jan. — P. 26–36.
9. *Kowalski R.A.* The early years of logic programming // Ibid. — 1988, 31, 1 Jan. — P. 38–43.
10. *Robinson J.A.* Logic and logic programming // Ibid. — 1992, 35, 3 Mar. — P. 40–65.
11. *Tichy W.F.* Programming-in-the-large: past, present, and future // Proc. of the 14th Intern. conference on Software engineering, 1992. — P. 362–367.
12. *Kay A.C.* The early history of Smalltalk // The 2 ACM SIGPLAN conf. on History of programming languages, 1993. — P. 69–95.
13. *Guy L.S., Gabriel R.P.* The evolution of Lisp // Ibid. — 1993. — P. 231–270.
14. *Liskov B.* A history of CLU // Ibid. — 1993. — P. 133–147.
15. *Whitaker W.A.* Ada — the project: the DoD high order language working group // Ibid. — 1993. — P. 299–331.
16. *Lindsey C.H.* A history of ALGOL 68 // Ibid. — 1993. — P. 97–132.
17. *Ritchie D.M.* The development of the C language // Ibid. — 1993. — P. 201–208.
18. *Stroustrup B.* A history of C++: 1979-1991 // Ibid. — 1993. — P. 271–297.
19. *Колодницький М.М.* Технічне та програмне забезпечення комп'ютерних інформаційних технологій. — Житомир: ЖІТІ, 1995. — 231 с.
20. *Nicholls J.E.* The structure and design of programming languages. Reading, Mass. — Addison-Wesley, 1975. — 572 p.
21. *Horowitz E.* Fundamentals of programming languages. Rockville, Md.: 2nd ed. / Computer Science Press: 1984. — 446 p.
22. *Pratt T.* Programming languages: Design and implementation. — NJ: Prentice Hall, 1996. — 654 p.
23. *Sethi R.* Programming languages: Concepts and constructs. — Addison-Wesley, 1996. —640 p.
24. *Sebesta R.W.* Concepts of Programming Languages. — Addison Wesley, 2002. — 698 p.
25. *Колодницький М.М*. Аналіз парадигми ООП мовою С++. ("Що таке класи?") // Вісник ЖІТІ: Технічні науки. 2002. — №21. — С. 120–124.
26. *American* national standard for information systems: Programming language-C. ANSI 3.159-1989. Approved December 14, 1989 by the American National Standards Institute / Ed. by the Computer and Business Equipment Manufacturers Association, the Secretariat, and the American National Standards Institute. — New York: American National Standards Institute, 1990. — 119 p.
27. *Kernighan B.W., Ritchie D.M*. The C programming language: 2nd. ed. — N.J.: Prentice Hall, 1988. — 272 p.

28. *Stroustrup B*. The C++ programming language. Special ed. Reading, Mass. — Addison-Wesley, 2000. — 1019 p.
29. *The Java*™ Language Specification. Second Edition / James Gosling, Bill Joy, Guy Steele and Gilad Bracha. — Addison-Wesley. 1996. — 532 p.
30. *C# Language* Specification. Version 0.28, 5/7/2001.

Отримано ***27.05.04***

Про автора

Колодницький Микола Михайлович
професор

Місце роботи автора:
Natural Interactive Systems Laboratory
University of Southern Denmark
Campusvej 55, DK-5230 Odense M
Denmark