Лекція 4

Примітивні типи даних

Типи даних в Java поділяються на примітивні і посилання (рис. 2.1). ***Примітивні типи*** (які також називають скалярними, а інколи простими) є найбільш простими у використанні. Вони багато в чому схожі з типами даних мови C++. Змінна такого типу являє собою іменовану комірку пам'яті, в якій зберігається деяке значення – число або символ.

Типи-посилання – це типи *об'єктів*. Будь-який об'єкт в Java-програмі може бути доступний тільки через посилання на нього, звідси і назва відповідних типів. До типів-посилань відносяться: масиви, класи і інтерфейси.

Всі типи-посилання є спадкоємцями класу Object, тобто всі вони пов'язані спільною ієрархією класів. Примітивні типи стоять особняком і в цю ієрархію не входять. Необхідно завжди пам'ятати, що робота з примітивними типами і посиланнями принципово різниться.

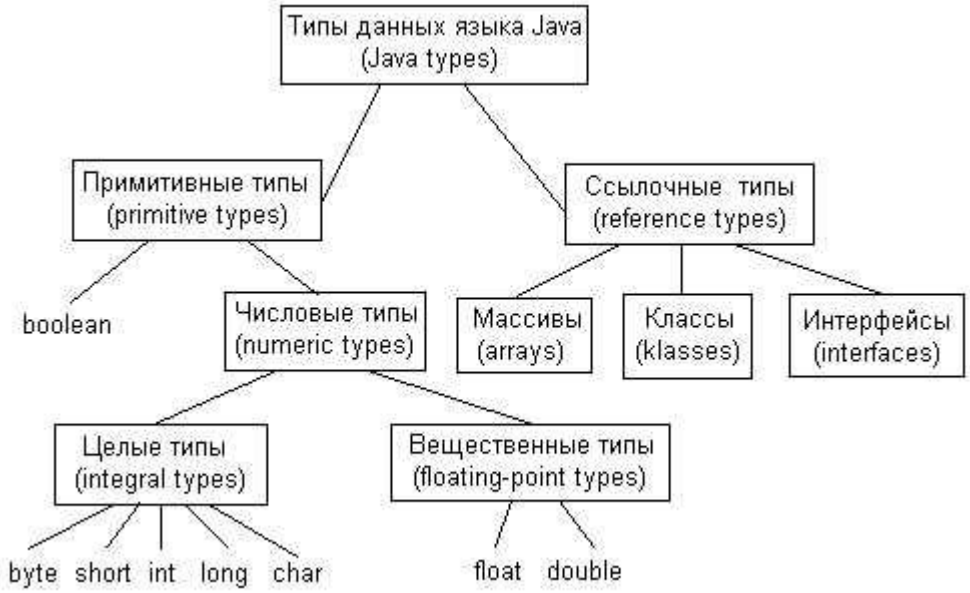

Рис. 2.1. Типи даних Java

Спочатку розглянемо примітивні типи Java (табл. 2.1). Їх всього 8: boolean, byte, char, short, int, long, float, double. Довжини і діапазони значень примітивних типів визначаються стандартом мови (а не конкретною її реалізацією: версією JDK, платформою) і однакові для всіх віртуальних машин. Вони наведені в таблиці 2.1. Тип char зробили двобайтовим, щоб він міг вміщати символи Unicode. Як вже говорилося раніше, це сильно полегшує локалізацію додатків: будь-який символ будь-якої мови має унікальний код і завжди коректно відображається. Коли складався стандарт мови Java, вже існував Unicode-16, але ще не було Unicode-32, тому символьний тип став саме двобайтовим. Оскільки в результаті не залишилося однобайтового типу, додали новий тип byte, причому в Java, на відміну від інших мов, він має знак. Зверніть увагу, в Java немає жодного беззнакового числового типу (тип char через жорсткі правила типізації з числовими типами несумісний).

Таблиця 2.1. Примітивні типи даних.

Тип	Довжина (в байтах)	Діапазон чи набір значень
boolean	1 – в масивах, 4 – в змінних	true, false
byte	1	−128..127

char	2	$0..2^{16}-1$, чи 0..65535
short	2	$-2^{15}..2^{15}-1$,чи −32768..32767
int	4	$-2^{31}..2^{31}-1$, чи −2147483648..2147483647
long	8	$-2^{63}..2^{63}-1$, чи приблизно $-9.2\cdot10^{18}..9.2\cdot10^{18}$
float	4	$-(2-2^{-23})\cdot2^{127}..(2-2^{-23})\cdot2^{127}$, чи приблизно $-3.4\cdot10^{38}..3.4\cdot10^{38}$, а також –∞, ∞, NaN
double	8	$-(2-2^{-52})\cdot2^{1023}..(2-2^{-52})\cdot2^{1023}$, чи приблизно $-1.8\cdot10^{308}..1.8\cdot10^{308}$, а також –∞, ∞, NaN

Принципи представлення значень примітивних типів в пам'яті (рис. 2.2) в цілому збігаються з тими, що використовувалися в мові C++, тому окремо зупинятися на них не будемо.

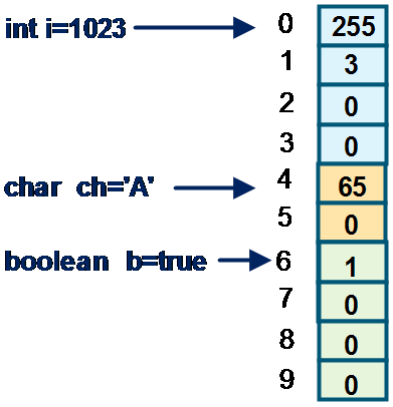

Рис. 2.2. Розташування примітивних типів в пам’яті

Важливо знати, що практично для всіх примітивних типів є також класи-обгортки, імена яких аналогічні, але починаються з великих літер (наприклад замість int – Integer, замість char – Character, замість double – Double). Ці класи підтримують ті ж операції, що і відповідні примітивні типи, а змінні цих класів у виразі автоматично приводяться до відповідних примітивних типів. В цілому можна сказати, що клас-обгортка може бути використаний скрізь, де можна використовувати примітивний тип.

Ці класи-обгортки в основному використовуються там, де за логікою програми необхідно число, але за правилами мови примітивний тип не допускається. Наприклад, в стандартний клас-список можна заносити тільки значення типів-посилань, тобто тільки об’єкти. Також класи-обгортки надають ряд нових можливостей: зокрема – містять наступні константи і методи:

1) Спеціальні значення речових типів float і double (–∞, ∞, і «не число» NaN) доступні у вигляді констант в класах Double і Float:

```
Double.POSITIVE_INFINITY,   // Плюс нескінченність
Double.NEGATIVE_INFINITY,   // Мінус нескінченність
Double.NaN;                 // “Не число”
```

Клас Float також має схожий набір констант.

Значення Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY можуть з'являтися в програмі в результаті переповнення типу double, а відповідні константи потрібні для перевірки таких ситуацій. Обидва значення «нескінченності» інтерпретуються як звичайні числа і можуть брати участь в арифметичних операціях та операціях порівняння: при цьому Double.POSITIVE_INFINITY вважається більше будь-якого іншого значення типу double, а Double.NEGATIVE_INFINITY – менше будь-якого іншого значення типу double.

Значення Double.NaN може з'являтися в програмі в результаті обчислення виразів, результат яких не визначений, наприклад ділення (0.0 / 0.0), або корінь з від'ємного числа Math.sqrt(–1). Це значення має наступні спеціальні властивості:

- NaN не дорівнює жодному іншому значенню (навіть самому собі); відповідно, найпростіший метод перевірки результату на NaN – це порівняння отриманої величини із самою собою.
- Будь-яка нетривіальна операція, яка бере NaN як операнд, завжди повертає NaN незалежно від значення інших операндів.

2) Мінімальні нормалізовані додатні значення, які приймають типи дійсних чисел float і double, наприклад:

```
Double.MIN_NORMAL;  // = 2^-1022 ~= 2.2E-308
```

3) Границі (мінімальні і максимальні значення), визначені для кожного цілочислового типу – також у вигляді констант, наприклад:

```
Byte.MIN_VALUE   = -128;
Byte.MAX_VALUE   = 127;
```

4) Методи для визначення, чи є величина скінченною. Саме цими методами необхідно користуватися для перевірки значення на NaN або на нескінченність (нагадаємо, що порівняння виду `variable = Double.NaN` не має сенсу і завжди має результат false):

```
static boolean Double.isNaN(double v);
static boolean Double.isInfinite(double v);
```

5) Константи для визначення довжини представлення типа в бітах, наприклад:

```
static int Integer.SIZE  = 32;
static int Double.SIZE  = 64;
```

6) Методи, що створюють екземпляр класу-оболонки із значення примітивного типу (ці методи в виразах викликаються автоматично), наприклад:

```
static Double Double.valueOf(double d);
static Integer Integer.valueOf(int i);
```

7) Перетворення числа в рядок (також у виразах виконується автоматично):

```
static String Double.toString(double d);
static String Integer.toString(int i);
```

8) Перетвоорення рядка в число, наприклад:

```
static Double Double.valueOf(String s);
static Double Double.parseDouble(String s);
static Integer Integer.valueOf(String s)
static Integer Integer.parseInt(String s)
```

Приклад. Використання констант и методів класів Double та Integer.

```
System.out.println("inf = " + Double.POSITIVE_INFINITY );
System.out.println("MIN_DOUBLE = " + Double.MIN_NORMAL);
System.out.println("SIZE of int = " + Integer.SIZE);
double d  = Double.valueOf("123") + Double.parseDouble("12.5");
System.out.println("123 + 12.5 = " + d );   // Неявно викликається Double.toString()
```

Результат виконання цього коду буде наступним:

```
inf = Infinity
MIN_DOUBLE = 2.2250738585072014E-308
SIZE of int = 32
123 + 12.5 = 135.5
```

Перетворення типів

В Java, як і в мові С, перетворення типів при обчисленні виразів можуть виконуватися або автоматично, або за допомогою оператора приведення типу, який має вигляд:

```
(новий_тип) значення
```

Однак правила приведення типів в Java більш суворі, ніж в мові С/С++.

Автоматичне перетворення відбувається при виконанні присвоювання і арифметичних операцій між *сумісними типами*. Дозволені лише *розширюючі* автоматичні перетворення (widening conversion) в напрямку:

byte→short→int→long→float→double.

Автоматичні перетворення в зворотному напрямку (тобто *звужуючі*) заборонені, щоб не виникало неявної втрати даних.

Також необхідно пам'ятати, що типи char і boolean не сумісні з числовими типами.

Результат будь-якої операції над значеннями byte, short завжди приводиться до типу int. У цьому легко переконатися на простому на прикладі:

```
byte a = 1, b = -4;
short c = a + b;        // Помилка – спроба перетворення int->short
```

Тут у другому рядку результат складання a і b має тип int, який, в свою чергу, не може бути автоматично приведений до більш «вузького» типу short.

Операція приведення типу дозволяє виконувати звужуючі перетворення (narrowing conversion) в напрямку

byte←short←int←long←float←double,

а також перетворення між типом char і числовими типами.

Перетворення між типом boolean і числовими типами виконати неможливо.

Наприклад:

```
int i,b = 123;
long l = 38388;
i = l;              // Помилка – звужуюче перетворення повинне бути явним
i = (int) l;        // Вірно, явне звужуюче перетворення
char a = b;         // Помилка – немає автоматичного перетворення int->char
System.out.println( (char)b );    // Вірно, виведе символ "}"
System.out.println( (char)1234 ); // Недрукований символ (виводиться '?')
boolean k = (boolean)i;          // Помилка – такого перетворення немає
```

Операції над примітивними типами

Операції над примітивними типами в Java також багато в чому повторюють набір операцій мов С/С++, але є й відмінності. Розглянемо ці операції по групах:

1) Логічні операції для типу `boolean`. На відміну від С, є дві форми логічних операцій:
   1. `!, &, |, ^` – повна форма, в якій обов'язково обчислюються всі операнди;
   2. `&&, ||` – скорочена форма, в якій другий операнд обчислюється тільки якщо від нього залежить результат операції. Наприклад, у виразі `(2>10)&&(a>0)`

обраховувати другий операнд немає сенсу, оскільки вже перший операнд `(2>10)` визначає результат логічного «і», незалежно від значення змінної a.

2) Арифметичні операції для числових типів: `+,-,*,/,%,++,--`. Операції `++`, `--` мають префіксну і постфіксну форми, які діють аналогічно до мови C/C++.

3) Операції порівняння для числових типів: `==, !=, >, >=, <, <=`.

4) Побітові операції для цілих типів: `~, &, |, ^`.

5) Побітові зсуви <<, >>, а також беззнаковий зсув >>>. Обидві операції >> і >>> виконують побітовий зсув вправо, але відрізняються своєю поведінкою з від'ємними числами. Операція >> при зсуві копіює знаковий біт числа в молодші розряди (при цьому число залишається від'ємним), а операція >>> заповнює старші розряди нулями (у результаті число стає додатним). Таким чином, операцію >> можна використовувати для ділення на 2, а операцію >>> – ні.

6) Операції звичайного присвоєння (=) і комбінованого присвоєння: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= ; >>>=`.

Результатом присвоювання у всіх випадках є нове значення лівої частини, тобто можливі ланцюжки виду `a=b=c`.

При комбінованому присвоєнні приведення типу виконується автоматично, наприклад:

```
byte b = 1;
b = b + 10; // Помилка! Спроба неявного перетворення із  int в byte
b += 10; // Правильно!
```

7) Умовна операція: `умова ? вираз1 : вираз 2`, наприклад

```
c = (b != 0 ? a/b : 0 );
```

У всіх операціях, крім присвоювання, операнди обчислюються зліва направо.

Пріоритет операцій визначається наступною послідовністю (від вищого до нижчого):

1)	Спеціальні операції	( ) [ ] .
2)	Унарні	~ !
3)	Арифметичні	* / %
4)	Арифметичні	+ –
5)	Зсуви	>> >>> <<
6)	Порівняння	> >= < <=
7)	Порівняння	== !=
8)	Побітові (по спаданню пріоритету)	& ^ |
9)	Логічні (по спаданню пріоритету)	&& ||
10)	Умова	?:
11)	Присвоєння	=, op=

Змінні

Як вже згадувалось, в Java всі типи змінних можна розділити на кілька великих груп (табл. 2.2).

Таблиця 2.2. Порівняння різних типів змінних Java.

Тип змінної	**Можливі значення**	**Значення за замовчуваням**
Примітивний	Визначаються границями типу	false, '\u0000', 0
Клас	null (нульове посилання), або посилання на об'єкт	null

	даного класу чи його спадкоємця	
Інтерфейс	null, або посилання на об'єкт будь-якого класу, що реалізує даний інтерфейс	null
Масив	null, або посилання на масив об'єктів відповідного типу	null
Object	null, або посилання на *будь-який* об'єкт	null

При цьому класи, інтерфейси, масиви відносяться до типів-посилань і є спадкоємцями класу Object.

Змінні примітивних типів

У багатьох мовах програмування, в залежності від області видимості, змінні поділяються на локальні і глобальні. Java має тільки один вид змінних – *локальні змінні*. ***Час життя*** будь-якої змінної в Java визначається правилом:

> *Змінна створюється в точці її опису і існує до моменту закінчення того блоку, в якому знаходиться її опис.*

Областю видимості змінної (scope) є фрагмент коду від точки її опису до кінця поточного блоку. Тобто час життя змінних в Java збігається з їх областю видимості, з урахуванням відмінності самих цих понять.

У кожному блоці може бути свій власний набір локальних змінних. При цьому у внутрішньому блоці неможна оголошувати змінні з таким же ім'ям, як і у зовнішньому по відношенню до нього блоці. Наприклад:

```
class myClass {
   int myVar=3;           // Оголошуємо змінну myVar
   public static void main (String args []){
      int myVar=5;        // Помилка! Така змінна вже існує
} }
```

Змінні-посилання

В Java для маніпулювання об'єктами в програмному коді використовуються ***посилання*** на об'єкти. Посилання зберігає в собі адресу деякого об'єкту в оперативній пам'яті. Об'єкти в програмі доступні тільки через посилання, але не може існувати посилання на примітивний тип. Клас об'єкта також є і типом посилання. Однак поняття «посилання» і «об'єкт» не рівнозначні. Так, може існувати кілька посилань на один об'єкт. На якийсь об'єкт може взагалі не бути посилань (наприклад, ми вийшли за межі області видимості всіх посилань на даний об'єкт), і тоді він для нас безповоротно втрачений. Також існує константа null – це порожнє посилання, яке не пов'язане з жодним об'єктом.

Посилання в Java в корені відрізняються від *покажчиків* в C++. Не можливо створити посилання на якусь довільну область пам'яті – за виконанням цього обмеження жорстко стежать як компілятор Java, так і JVM. Посилання може або бути пов'язане з деяким об'єктом, або дорівнювати null. При цьому компілятор і JVM також жорстко стежать за відповідністю типу об'єкта і типу посилання – в результаті можливості перетворення типів-посилань дуже обмежені. Немає ніякого способу перетворити посилання класу MyClass на посилання класу OtherClass, якщо OtherClass не є предком MyClass в ієрархії успадкування. З іншого боку, оскільки клас Object знаходиться на самій вершині ієрархії класів, то будь-яке посилання може бути перетворене на типу Object.

Всі об'єкти в Java створюються тільки явно, для чого використовується операція **new**.

```
MyType ref = new MyType();  // В дужках – параметри конструктора
```

Оператор new створює сам об'єкт і посилання на нього, причому його результатом є саме посилання.

Оскільки об'єкти доступні в програмі тільки через посилання на них, то ***область видимості об'єкта*** визначається загальною областю видимості всіх посилань на цей об'єкт. ***Час життя об'єкта*** тоді визначається наступним правилом:

Об'єкт існує, доки існує хоча б одне посилання на цей об'єкт.

Це правило, однак, не стверджує, що об'єкт буде знищений, як тільки зникне останнє посилання на нього. Просто такий об'єкт стає недоступним і може бути знищений без всяких небажаних наслідків для програми. Явно звільняти пам'ять в Java не потрібно. Об'єкти знищуються (або утилізуються) ***збирачем сміття*** (garbage collector) – окремим процесом, який завжди працює у фоновому режимі паралельно з Java-програмою.

Для пояснення сказаного розглянемо приклад:

```
SomeType globalReference = null;
{
  SomeType localReference = new SomeType();
  globalReference = localReference;
}
// ...
```

Тут всередині блоку створюється об'єкт класу SomeType, і посилання на цей об'єкт заноситься в локальну змінну localReference. Після цього те ж посилання з localReference копіюється в globalReference. При виході з блоку {} змінна-посилання localReference знищується, але змінна globalReference продовжує існувати. Відповідно, продовжує існувати і створений об'єкт.

Фінальні змінні

Змінні, оголошені з модифікатором **final**, називаються «фінальними» і відрізняються від звичайних тим, що можуть отримати своє значення тільки один раз при ініціалізації. При цьому ініціалізація може відбуватися як в момент оголошення змінної, так і пізніше, окремим оператором, наприклад:

```
final int i = 5;        // Ініціалізація при оголошенні
final String s;
s = "ddddd";            // Ініціалізація окремо від оголошення
i = 10;                 // Помилка! Надати інше значення не можна
```

Примітка. Ініціалізація фінальних членів класу повинна проводитися або при оголошенні, або в конструкторі. Ініціалізація фінальних статичних змінних класу повинна проводитися тільки при оголошенні (насправді також можливо надати їх значення в так званому статичному блоці ініціалізації, але використання цієї можливості виходить за рамки даного курсу).

Все це зовсім не означає, що фінальну змінну взагалі не можна змінити. Для змінних-посилань обмеження `final` полягає лише в тому, що в змінній не можна зберегти посилання на інший об'єкт. Але в той же час через фінальну змінну-посилання можна викликати будь-які методи пов'язаного з нею об'єкта, навіть якщо вони змінюють стан або внутрішні дані об'єкту, наприклад:

```
final SomeClass obj = new SomeClass();      // Ініціалізація при оголошенні
obj.change();                               // Виклик методу,
                                            // що змінює об’єкт
```

Статичні змінні

Статичними (**static**) в Java оголошуються тільки члени класу. Значення статичної змінної повинно бути спільним для всіх об'єктів класу, в якому вона оголошена. Звичайні змінні оголошені в класі (без модифікатора `static`) прийнято називати ***змінними реалізації (instance variables)***, тому що в кожному об'єкті класу (тобто в кожній його реалізації) знаходиться своє

значення такої змінної. Аналогічно, оскільки значення статичної змінної відноситься не до конкретного об'єкта (реалізації), а до класу в цілому, такі змінні називають ***змінними класу (class variables).***

Змінні, оголошені всередині методів, не можуть бути статичними, тобто модифікатор `static` не застосовується до локальних змінних. Більш докладно про змінні класу йтиметься в наступній лекції.

Масиви

Кожен масив в Java є об'єктом, а змінні типу масив є *посиланнями*. Масив створюється в три етапи:

1) ***Оголошення*** змінної – посилання на масив, наприклад:

```
double[] a, b;// Дві посилання на масиви
double c[], d;// Посилання  на масив a, і число d
```

2) ***Виділення пам'яті*** для масиву, в результаті чого в змінну заноситься адреса реального масиву. Цей крок можна робити одночасно з оголошенням масиву або окремо.

```
double a[];
a = new double[5];
double c[] = new double[5];
```

Після цього кроку до елементів масиву можна звертатися за допомогою операції **[ ]**. Індексація завжди починається з 0. Якщо індекс виходить за межі масиву, генерується виключення `IndexOutOfBoundsException`, але такій ситуації необхідно запобігати вже на етапі написання програми.

Розмір масиву зберігається у властивості `length`. Використовувати цю властивість завжди краще, ніж підставляти в код число або константу.

3) ***Ініціалізація*** елементів масиву. Існує два способи ініціалізації масиву:

а) ініціалізація при створенні масиву:

```
double b[] = new double[] {0.1, 0.2, -0.3, 0.45, -0.02};
double[] d = {0.01, -3.4, 2.89, 4.5, -6.7};
```

б) звертання до елементів масиву, наприклад:

```
double a[] = new int[5];
for (int i = 0; i < a.length; i++) a[i] = i*i;
```

Також можна створити *безіменний масив*, використовуючи результат операції `new` в виразі, наприклад:

```
System.out.println (new char [] {'H', 'e', '1', '1', 'o'});
```

Такий безіменний масив завжди повинен буди ініціалізований при створенні, оскільки після його створення не залишається посилання на масив.

Масиви об'єктів

Масив об'єктів – це масив посилань на об'єкти. Відповідно, потрібно створити як масив, так і самі об'єкти. І те й інше створюється за допомогою оператора **new**. Нехай у нас є певний клас SomeClass і потрібно побудувати масив з 4-х об'єктів цього класу.

```
// Варіант 1. (явне занесення об'єктів в масив)
SomeClass arr[] = new SomeClass[4];         // Створення масива
for (int j = 0; j < 4; j++ )                // Ініціалізація масива
     arr[j] = new SomeClass();              // Ініціалізація елементів

// Варіант 2. (використання списка ініціалізації)
```

```
SomeClass arr[] = new SomeClass[] {              // Створення масива
          new SomeClass(), new SomeClass(),      // Ініціалізація елементів
          new SomeClass(), new SomeClass()
     };
```

Наприклад, створимо масив із двох рядків і зразу ж його ініціалізуємо:

```
String s[] = {new String("aaa"), "bbb"};
```

Багатовимірні масиви

В Java багатовимірні масиви як окремий тип даних відсутні. Але логічно багатовимірний масив можна організувати, якщо в звичайному масиві кожен елемент є посиланням на масив меншої розмірності. Наприклад, створимо двовимірний масив цілих чисел розміру 3 * 3.

```
// Вариант 1. Просте створення масиву
int ar[][] = new int[3][3];           // Далі необхідна ініціалізація!
for(int i=0; i<ar.length; i++ ) {     // Ініціалізація
     for(int j=0; j<ar[i].length; j++) ar[i][j] = i+j;
}

// Варіант 2. Використання списка ініціалізації
int ar1[][] = new int[][] {  {1, 2, 3}, {4, 5, 6}, {7, 8, 9}  };
```

Для ініціалізації багатовимірного (як і одновимірного) масиву об'єктів кожен об'єкт повинен бути створений окремо (за допомогою `new`) і записаний в масив. Це можна зробити в списку ініціалізації або окремим циклом.

Оскільки вкладені масиви є об'єктами, вони також можуть створюватися окремо, і навіть мати різні розміри, наприклад:

```
int ma[][] = new int[4][];       // Створюємо масив посилань на масиви,
                                 // але не створюємо самі вкладені масиви
for(int i=0; i<ma.length; i++ ) {  // Для кожного елемента верхнього рівня
     ma[i] = new int[i];         // створюємо посилання на  масив-рядок
     for(int j=0; j<i; j++) ma[i] = i+j; // Ініціалізуємо масив-рядок
}
```

Операції над посиланнями і масивами

Над масивами, як і над іншими змінними-посиланнями (екземплярами класу `Object`) можна виконувати наступні загальні операції:\

1) ***Присвоєння*** a = b, яке є копіюванням посилання. В результаті обидва посилання a й b вказують на один і той самий об'єкт (або масив) в пам'яті.

2) ***Порівняння*** на рівність і нерівність за допомогою операцій = =,! =, а також методу

```
boolean Object.equals (Object obj);
```

Операції == і != зіставляють адреси, що містяться в посиланнях, а не вміст об'єктів (або масивів), тобто ми можемо дізнатися, чи не посилаються обидва посилання на один і той самий об'єкт в пам'яті. Поведінка методу `equals()` різниться для різних класів, наприклад для `String` метод `equals()` порівнює вміст рядків, а для масивів – цей же метод порівнює лише посилання, тобто працює аналогічно до порівняння ==.

Проілюструємо роботу присвоювання і обох способів порівняння на прикладі:

```
String s = new String("abc");   // Рядок
String s1 = s;                  // Копія посилання
String s2 = new String("abc");  // Іще один такий самий рядок
System.out.println( s == s1 );  // Виведе true, оскільки посилання рівні
System.out.println( s == s2 );  // Виведе false, оскільки посилання різні
```

```
System.out.println( s.equals(s2) );  // Виведе true, т. я. рядки однакові
```

Примітка. Рядок можна ініціалізувати рядковим літералом, наприклад:

```
String s3 = "abc";
```

При цьому також створюється об'єкт-рядок, і посилання на нього заноситься в змінну s3. Але якщо створити ще однин такий самий рядок:

```
String s4 = "abc";
```

компілятор оптимізує наш код, і не буде створювати ще один зайвий об'єкт, а замість цього збереже в s4 те саме посилання, що і в s3. Тому при порівнянні посилань (s3 == s4) вони виявляться рівними.

3) ***Створення копії*** об’єкта чи масиву за допомогою методу **Object.clone()** :

```
int a[] = {1,2,3}, b[];
b = a;    a.equals(b);                 // true
c = a.clone();  a.equals(c);           // false
```

Важливо пам’ятати, що при цьому якщо в масиві зберігаються посилання, то об'єкти-елементи *не копіюються*, а в новий масив вносяться лише копії посилань.

Цих загальних методів класу `Object` абсолютно недостатньо для повноцінної роботи з масивами. Тому в стандартному класі **java.util.Arrays** реалізовані додаткові методи обробки масивів. Розглянемо їх на прикладі:

```
import java.util.*;  // Підключити пакет java.util
// ...
     int a[] = new int[1000];
     // Заповнити випадковими числами в діапазоні [10..99]
     for(int i = 0; i<a.length; i++) a[i] = (int)(Math.random()*90) + 10;

     // Заповнити елементи від 2 до 5 (не включно) значенням -1
     Arrays.fill(a, 2, 5, -1);

     // Створити копію діапазону елементів від 0-го до 10-го (не включно)
     // і перетворити отриманий масив на рядок
     int b[] = Arrays.copyOfRange(a, 0, 10);
     System.out.println( Arrays.toString(b) );

     // Перевірити рівність всіх елементів для двох масивів
     Arrays.equals(Arrays.copyOfRange(a, 2, 5), new int [] {-1,-1,-1});

     // Відсортувати масив методом QuickSort
     Arrays.sort(a);  System.out.println( Arrays.toString(a) );

     // Знайти елемент у відсортованому масиві методом бінарного пошуку
     // Результат - індекс знайденого елемента або число <0
     int pos = Arrays.binarySearch(a, 20);
     System.out.println(  pos>=0? pos : "none" );
```

Всі наведені методи працюють з **одновимірними** масивами. Масив більшої розмірності вони розглядають як одновимірний масив посилань. Наприклад, код:

```
System.out.println(Arrays.toString( new int[][] {{1},{2,3}} ));
```

виведе три посилання на масиви-рядки, а не елементи 1, 2, 3:

```
[[I@8965fb, [I@867e89, [I@1dd7056]
```

Тому для роботи з багатовимірними масивами клас **java.util.Arrays** має спеціальні методи **Arrays.deepEquals()** і **Arrays.deepToString(),** які діють аналогічно Arrays.equals() і Arrays.toString(), але рекурсивно обробляють всі вкладені масиви.

Повний перелік методів класу `Arrays` і їх опис можна знайти в документації.

Порівняння різних способів копіювання і порівняння масивів.

При копіюванні і порівнянні об'єктів і масивів завжди необхідно чітко розуміти, що саме буде робити кожен конкретний метод. Якщо цих тонкощів не враховувати, то з'являються дуже важко вловимі помилки. Щоб закріпити тонкощі копіювання і порівняння в Java, спробуйте самостійно заповнити наступні дві таблиці (табл. 2.3, 2.4). Зверніть особливу увагу на випадки, де *результат може залежати від типу і розмірності масиву*.

Таблиця 2.3. Різні способи порівняння для масиву рядків: `String a[] = {"1","2"};`
і для масиву чисел: `int a[] = {1, 2};`.

B	b==a	b.equals(a)	Arrays. equals(a,b)	Arrays. deepEquals(a,b)	b[0]==a[0]
b= a	true	true			
b= a.clone()					
b= Arrays. copyOf(a,a.length)					
b={"1","2"}					

Таблиця 2.4. Різні способи порівняння для двовимірного масиву рядків:
`String a[] = { {"1","2","3"}, {"1","2"} };`.

b	b==a	b.equals(a)	Arrays. equals(a,b)	Arrays. deepEquals(a,b)	b[0]==a[0]
b= a	true	true			
b= a.clone()					
b= Arrays. copyOf(a,a.length)					
b={{"1","2",…},…}					

Передача параметрів до методів

В Java існує тільки один тип передачі параметрів – ***за значенням***. Це означає, що під час виклику методу йому передається *копія* поточного значення параметра, яка знищується при виході з методу.

При передачі параметра типу посилання створюється *копія посилання*, але не самого об'єкта. Подивимося більш детально, що це означає. Нехай в метод method() передаються посилання на об'єкт ref:

```
public static void main(String[] args) {
     SomeClass ref = new SomeClass();
     method(ref);
}
```

Всередині методу method() ми можемо змінити посилання ref (тобто присвоїти йому посилання на інший об'єкт), але це ніяк не вплине на посилання ref у методі main(). Проте якщо всередині method() змінити сам об'єкт, викликавши який-небудь його метод ref.change(), то ця зміна вплине на об'єкт ref у методі main().

Примітка. Константні посилання і підтримка так званих *const correctness* в Java відсутні. Це означає, що при оголошенні методу неможливо «заборонити» йому змінювати об'єкти через свої параметри-посилання.

Приклад. Маємо методи, що змінюють значення своїх параметрів:

```
static void change(String s) {  s = "bbb"; }
```

```
static void change(int a[]) {
     if(a!=null) for(int i = 0; i<a.length; i++ ) a[i] = 0;
}
static void change(int a[][]) {
     if(a!=null)
          for(int i = 0; i<a.length; i++ )
               for(int j =0; j<a[i].length; j++ ) a[i][j] = 0;
}
```

Також маємо локальні змінні:

```
String s = "aaa";
int x[] = {1,2,3}, y[][] = {{1,2,3},{4,5,6},{7,8,9}};
```

Тоді в результаті викликів:

```
change(s); change(a); change(y);
```

Рядок s залишиться незмінним, оскільки в метод change() передається копія посилання s, і ця копія отримує нове значення (тобто посилання на інший рядок).

Вміст масивів x і y зміниться, оскільки при виклику методу change() вміст цих масивів не буде копіюватися і доступний зсередини методу change() за посиланням.

Під час викликів

```
change(x.clone()); change(y.clone());
```

явно створюються копії масивів, разом із збереженими в них значеннями. Зміна елементів масиву-копії x.clone() ніяк не впливає на вміст вихідного масиву **x**. З масивом **y** все складніше, оскільки копія y.clone() – це одновимірний масив посилань на менші масиви-рядки. При цьому з масиву **y** копіюються всі посилання на рядки, але не самі масиви-рядки. Тому всередині change() копія масиву **y** містить посилання на ті ж самі об'єкти-рядки, що і сам масив **y**. Через ці посилання вміст масиву **y** змінюється зсередини методу change().