# **Java Building Blocks**

## **Data Types**

Các kiểu dữ liệu nguyên thuỷ - Primitive Data type

Có 8 kiểu dữ liệu nguyên thủy (Primitive) đó là: boolean, byte, char, short, int, long, float, double.

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Type** | **Example** |
| boolean | true or false | true |
| byte | 8-bit integral value | 123 |
| short | 16-bit integral value | 123 |
| int | 32-bit integral value | 123 |
| long | 64-bit integral value | 123L |
| float | 32-bit floating-point value | 123.45f |
| double | 64-bit floating-point value | 123.456 |
| char | 16-bit Unicode value | 'a' |

* Các kiểu float và double được sử dụng cho các giá trị dấu phẩy động (thập phân).
* Dữ liệu float yêu cầu chữ f theo sau số để Java biết đó là float.   
  float a = 1; // COMPILE  
  float a = 1.1 //DOES NOT COMPILE phải khai báo float a = 1.1f;
* Các kiểu byte, short, int và long được sử dụng cho các số không có dấu thập phân.
* Tất cả các loại số đều được xác định bằng Java. Điều này có nghĩa là họ dự trữ một trong các bit của mình để xác định phạm vi âm. Ví dụ: byte có phạm vi từ -128 đến 127. Bạn có thể ngạc nhiên khi phạm vi này không phải là -128 đến 128. Đừng quên, 0 cũng cần được tính trong phạm vi đó.

### **Writing Literals**

* Cân chỉ định kiểu dữ liệu như trường hợp ví dụ về float bên trên:

long max = 3123456789; // DOES NOT COMPILE - Java hiểu nó là kiểu **int**

long max = 3123456789L; // Java biết nó là kiểu **long**

* Một cách khác để chỉ định số là thay đổi “cơ số”.  
  Java cho phép bạn chỉ định các chữ số theo một số định dạng khác:
  + Bát phân (chữ số 0–7), sử dụng số 0 làm tiền tố—ví dụ: 017
  + Hệ thập lục phân (các chữ số 0–9 và các chữ cái A–F/a–f), sử dụng 0x hoặc 0X làm tiền tố—ví dụ: 0xFF, 0xff, 0XFf. Hệ thập lục phân không phân biệt chữ hoa chữ thường nên tất cả các ví dụ này đều có cùng giá trị.
  + Nhị phân (chữ số 0–1), sử dụng số 0 theo sau là b hoặc B làm tiền tố — ví dụ: 0b10, 0B10

### **Literals and the Underscore Character (Chữ và ký tự gạch dưới)**

Điều cuối cùng cần biết về chữ số là bạn có thể có dấu gạch dưới trong số để dễ đọc hơn:

int million1 = 1000000;  
int million2 = 1\_000\_000;

Bạn có thể thêm dấu gạch dưới ở bất cứ đâu ngoại trừ ở đầu chữ, ở cuối chữ, ngay trước dấu thập phân hoặc ngay sau dấu thập phân. Bạn thậm chí có thể đặt nhiều ký tự gạch dưới cạnh nhau, mặc dù không nên làm như vậy:

|  |  |
| --- | --- |
| double notAtStart = \_1000.00;  double notAtEnd = 1000.00\_;  double notByDecimal = 1000\_.00;  double annoyingButLegal = 1\_00\_0.0\_0;  double reallyUgly = 1\_\_\_\_\_\_\_\_\_\_2; | // DOES NOT COMPILE // DOES NOT COMPILE // DOES NOT COMPILE // Ugly, but compiles // Also compiles |

### **Identifying identifiers**

Chỉ có bốn quy tắc cần nhớ đối với identifier(mã định danh, mã định danh là tên của variable, method, class,interface, hay package) hợp pháp:

* Identifier phải bắt đầu bằng một chữ cái, ký hiệu $ hoặc ký hiệu \_.
* Identifier có thể bao gồm các số nhưng không bắt đầu bằng chúng.
* Kể từ Java 9, một dấu gạch dưới \_ không được phép làm mã định danh.
* Bạn không thể sử dụng cùng tên với một từ dành riêng cho Java. Từ dành riêng là từ đặc biệt mà Java đã giữ lại để bạn không được phép sử dụng nó. Hãy nhớ rằng Java phân biệt chữ hoa chữ thường, vì vậy bạn có thể sử dụng các phiên bản của từ khóa chỉ khác nhau về chữ hoa chữ thường.  
  Lưu ý: String String = “Mars”; // Hợp lệ

### **Default initialization values by type**

|  |  |
| --- | --- |
| **Variable type** | **Default initialization value** |
| boolean | false |
| byte, short, int, long | 0 |
| float, double | 0.0 |
| char | \u0000' (NUL) |
| All object references (everything else) | null |

### **Giới thiệu var**

Bắt đầu từ Java 10, bạn có tùy chọn sử dụng từ khóa var thay vì loại cho các **biến cục bộ** trong một số điều kiện nhất định. Bạn chỉ cần sử dụng var thay vì primitive hay reference type.

public void whatTypeAmI() {  
 var name = "Hello";  
 var size = 7;  
}

var là local variable type inference. Đầu tiên là biến cục bộ. Điều này có nghĩa đúng như tên gọi của nó. Bạn chỉ có thể sử dụng tính năng này cho các biến cục bộ. OCP có thể cố gắng đánh lừa bạn bằng mã như thế này:

public class VarKeyword {  
 var tricky = "Hello"; // DOES NOT COMPILE  
}

**Type Inference of var**

Khi bạn sử dụng var, bạn đang hướng dẫn trình biên dịch xác định kiểu dữ liệu. Trình biên dịch xem code trên dòng khai báo và sử dụng nó để suy ra kiểu dữ liệu. Hãy xem ví dụ này:

7: public void reassignment() {  
8: var number = 7;  
9: number = 4;  
10: number = "five"; // DOES NOT COMPILE  
11: }

Dòng 8, xác định kiểu dữ liệu là **int.**

Dòng 10, có vấn đề, nó đã yêu cầu nó gán String cho biến int. Điều này không được phép. Nó tương đương với việc gõ cái này:

int number = "five";

Vì vậy, kiểu dữ liệu của var không thể thay đổi trong thời gian, nhưng còn giá trị thì sao? Hãy xem đoạn mã sau:

var apples = (short)10;  
apples = (byte)5;  
apples = 1\_000\_000; // DOES NOT COMPILE

Dòng đầu tiên tạo ra một biến var có tên là apple với kiểu short.

Dòng cuối cùng không được biên dịch, vì một triệu vượt xa giới hạn **short**. Trình biên dịch xử lý giá trị dưới dạng **int** và báo lỗi cho biết nó không thể được gán cho apple.

**Examples with var**

4: public void twoTypes() {  
5: int a, var b = 3; // DOES NOT COMPILE  
6: var n = null; // DOES NOT COMPILE  
7: }

Dòng 5 sẽ không hoạt động ngay cả khi bạn thay thế var bằng loại thực. Tất cả các kiểu được khai báo trên một dòng phải cùng kiểu và có chung một khai báo. Chúng ta không thể viết int a, int v = 3; hoặc. Tương tự như vậy, điều này không được phép:

5: var a = 2, b = 3; // DOES NOT COMPILE

Java không cho phép var khai báo nhiều biến.

Dòng 6 là một dòng duy nhất. Trình biên dịch đang được yêu cầu suy ra kiểu dữ liệu của null. Đây có thể là bất kỳ loại tham chiếu nào. Lựa chọn duy nhất mà trình biên dịch có thể thực hiện là Object. Tuy nhiên, đó gần như chắc chắn không phải là điều mà đoạn mã dự định. Các nhà thiết kế của Java đã quyết định rằng tốt hơn là không cho phép var null hơn là phải đoán ý định.

* Không được khai báo **null** cho var ngoại trừ bạn xác định kiểu dữ liệu cho null.  
  Ví dụ: var a = (String) null;

### **Quy tắc khai báo var**

1. **var** được sử dụng làm biến cục bộ trong constructor, method, hay initializer block..

2. Không thể sử dụng **var** trong constructor parameters, method parameters, instance variables, hay class variables.

3. **var** luôn được khởi tạo trên cùng một dòng (hoặc câu lệnh) nơi nó được khai báo.

var a; // Does not complite

4. Giá trị của biến var có thể thay đổi nhưng kiểu thì không.

5. var không thể được khởi tạo với giá trị null nếu không có loại.

6. Không được phép sử dụng var trong khai báo nhiều biến.

7. Var là tên loại dành riêng nhưng không phải là từ dành riêng, nghĩa là nó có thể được sử dụng làm identifier danh ngoại trừ tên class, interface, hoặc tên enum.

var var = 100; // complite

## **Garbage Collection trong java**

Trong java, rác (garbage) có nghĩa là các đối tượng không còn được tham chiếu nữa.

Bộ thu gom rác (Garbage Collection) trong java được sử dụng để thực hiện quá trình tự động khôi phục lại bộ nhớ không được sử dụng tại runtime một cách tự động. Nói cách khác, đó là một cách để phá hủy các đối tượng không sử dụng nữa.

### **Ưu điểm của Garbage Collection**

* Nó làm cho việc sử dụng **bộ nhớ java hiệu quả**bởi vì bộ thu gom rác (Garbage Collection) loại bỏ các đối tượng không được tham chiếu từ bộ nhớ heap.
* Nó được **được thực hiện tự động**bởi trình thu gom rác (một phần của JVM) vì vậy chúng ta không cần phải nỗ lực nhiều để giải phóng bộ nhớ.

### **Làm thế nào có thể một đối tượng không được tham chiếu?**

Có vài cách như sau:

* Bởi gán giá trị null.
* Bởi việc gán đối tượng đến một tham chiếu khác;
* Bởi một đối tượng annonymous.

#### **Bởi gán giá trị null**

|  |  |
| --- | --- |
| 1  2 | Employee e=**new** Employee();  e=**null**; |

### **Bởi việc gán đối tượng đến một tham chiếu khác**

|  |  |
| --- | --- |
| 1  2  3 | Employee e1=**new** Employee();  Employee e2=**new** Employee();  e1=e2; |

#### **Bởi một đối tượng annonymous**

|  |  |
| --- | --- |
| 1 | **new** Employee(); |

### **Phương thức finalize()**

Phương thức finalize() được gọi mỗi lần trước khi đối tượng được thu gom rác. Phương thức này có thể được sử dụng để thực hiện xử lý dọn dẹp. Phương thức này được định nghĩa trong lớp Object như sau:

**protected** **void** finalize(){}

Ghi chú: Garbage Collection của JVM chỉ thu thập các đối tượng được tạo ra bởi từ khoá new. Vì vậy, nếu bạn đã tạo ra bất kỳ đối tượng nào mà không có new, bạn có thể sử dụng phương thức finalize() để thực hiện quá trình xử lý dọn dẹp (phá hủy các đối tượng còn lại).

### **Phương thức gc()**

Phương thức gc () được sử dụng để gọi bộ thu gom rác để thực hiện quá trình dọn dẹp. Phương thức gc() được cài đặt trong các lớp System và Runtime.

Ghi chú: Garbage Collection được thực hiện bởi một luồng hiểm gọi là Garbage Collector (GC). Luồng này gọi phương thức finalize() trước khi đối tượng thu gom rác.

### **Ví dụ về garbage collection trong java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **public** **class** TestGarbage1 {  **public** **void** finalize() {          System.out.println("object is garbage collected");      }    **public** **static** **void** main(String args[]) {          TestGarbage1 s1 = **new** TestGarbage1();          TestGarbage1 s2 = **new** TestGarbage1();          s1 = **null**;          s2 = **null**;          System.gc();      }  } |

Output:

object is garbage collected

object is garbage collected

# **Operators**

Java cung cấp rất nhiều toán tử đa dạng để thao tác với các biến. Chúng ta có thể chia tất cả các toán tử trong Java thành các nhóm sau:

## **Toán tử số học**

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ: B = 20, A = 10** |
| + | Phép cộng | A + B sẽ cho kết quả 30 |
| – | Phép trừ: trừ toán hạng trái cho toán hạng phải | A – B sẽ cho kết quả -10 |
| \* | Phép nhân | A \* B sẽ cho kết quả 200 |
| / | Phép chia: chia toán hạng trái cho toán hạng phải | B / A sẽ cho kết quả 2 |
| % | Phép chia lấy phần dư: Lấy phần dư của phép chia toán hạng trái cho toán hạng phải | B % A sẽ cho kết quả 0 |
| ++ | Phép lượng gia: lượng gia giá trị toán hạng thêm 1 | B++ sẽ cho kết quả 21 |
| — | Phép lượng giảm: lượng giảm giá trị toán hạng đi 1 | B– sẽ cho kết quả 19 |

## **Toán tử quan hệ**

|  |  |  |
| --- | --- | --- |
| TOÁN TỬ | MIÊU TẢ | VÍ DỤ**: B = 20, A = 10** |
| == | Kiểm tra nếu giá trị của hai toán hạng có cân bằng hay không, nếu có thì điều kiện là true. | (A == B) là không true. |
| != | Kiểm tra nếu giá trị hai toán hạng là cân bằng hay không, nếu không cân bằng, thì điều kiện là true | (A != B) là true. |
| > | Kiểm tra nếu toán hạng trái có lớn hơn toán hạng phải hay không, nếu có thì điều kiện là true | (A > B) là không true. |
| < | Kiểm tra nếu toán hạng phải có lớn hơn toán hạng trái hay không, nếu có thì điều kiện là true | (A < B) là true. |
| >= | Kiểm tra nếu toán hạng trái có lớn hơn hoặc bằng toán hạng phải hay không, nếu có thì điều kiện là true | (A >= B) là không true. |
| <= | Kiểm tra nếu toán hạng phải có lớn hơn hoặc bằng toán hạng trái hay không, nếu có thì điều kiện là true | (A <= B) là true. |

Lưu ý: Toán tử == và != được sử dụng trong các trường hợp sau:

* So sánh hai kiểu nguyên thủy số hoặc ký tự. Nếu các giá trị số thuộc các kiểu dữ liệu khác nhau, các giá trị sẽ tự động được thăng hạng. Ví dụ: 5 == 5,00 trả về true vì phía bên trái được thăng cấp thành double.
* So sánh hai giá trị boolean.
* So sánh hai đối tượng, bao gồm giá trị null và chuỗi.

## **Toán tử thao tác bit**

|  |  |  |
| --- | --- | --- |
| TOÁN TỬ | MIÊU TẢ | VÍ DỤ**: B = 20, A = 10** |
| & | Toán tử Và nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong cả hai toán hạng | (A & B) sẽ cho kết quả 12, hay là 0000 1100 |
| | | Toán tử Hoặc nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong một hoặc hai toán hạng | (A | B) sẽ cho kết quả 61, hay là 0011 1101 |
| ^ | Toán tử Hoặc loại trừ nhị phân sao chép bit nếu nó được thiết lập trong một toán hạng nhưng không phải trong cả hai | (A ^ B) sẽ cho kết quả 49, hay là 0011 0001 |
| ~ | Toán tử đảo bit là toán tử một ngôi. Đảo bít 1 thành 0 và ngược lại | (~A ) sẽ cho kết quả -61, hay là 1100 0011 |
| << | Toán tử dịch trái. Giá trị toán hạng trái được dịch chuyển sang trái bởi số các bit được xác định bởi toán hạng bên phải. | A << 2 sẽ cho kết quả 240, hay là 1111 0000 |
| >> | Toán tử dịch phải. Giá trị toán hạng trái được dịch chuyển sang phải bởi số các bit được xác định bởi toán hạng bên phải | A >> 2 sẽ cho kết quả 15, hay là 1111 |
| >>> | Toán tử dịch phải và điền 0 vào chỗ trống | A >>>2 sẽ cho kết quả 15, hay là 0000 1111 |

## **Toán tử logic**

|  |  |  |
| --- | --- | --- |
| TOÁN TỬ | MIÊU TẢ | VÍ DỤ**: B = TRUE, A = FALSE** |
| && | Toán tử **Và** logic. Nếu **cả hai** toán hạng là **true**, thì khi đó điều kiện là **true** | (A && B) là false. |
| || | Toán tử **Hoặc** logic. Nếu **một trong hai** toán tử là **true**, thì điều kiện là **true** | (A || B) là true. |
| ! | Toán tử **Phủ định** logic. Sử dụng để đảo ngược lại trạng thái logic của toán hạng đó. Nếu điều kiện toán hạng là true thì phủ định nó sẽ là false. | !(A && B) là true. |

## **Các toán tử gán**

|  |  |  |
| --- | --- | --- |
| TOÁN TỬ | MIÊU TẢ | VÍ DỤ**: B = 20, A = 10** |
| = | Toán tử gán đơn giản. Gán giá trị toán hạng bên phải cho toán hạng trái. | C = A + B sẽ gán giá trị của A + B vào cho C |
| += | Thêm giá trị toán hạng phải tới toán hạng trái và gán giá trị đó cho toán hạng trái. | C += A là tương đương với C = C + A |
| -= | Trừ đi giá trị toán hạng phải từ toán hạng trái và gán giá trị này cho toán hạng trái. | C -= A là tương đương với C = C – A |
| \*= | Nhân giá trị toán hạng phải với toán hạng trái và gán giá trị này cho toán hạng trái. | C \*= A là tương đương với C = C \* A |
| /= | Chia toán hạng trái cho toán hạng phải và gán giá trị này cho toán hạng trái. | C /= A là tương đương với C = C / A |
| %= | Lấy phần dư của phép chia toán hạng trái cho toán hạng phải và gán cho toán hạng trái. | C %= A là tương đương với C = C % A |
| <<= | Dịch trái toán hạng trái sang số vị trí là giá trị toán hạng phải. | C <<= 2 là giống như C = C << 2 |
| >>= | Dịch phải toán hạng trái sang số vị trí là giá trị toán hạng phải. | C >>= 2 là giống như C = C >> 2 |
| &= | Phép AND bit | C &= 2 là giống như C = C & 2 |
| ^= | Phép OR loại trừ bit | C ^= 2 là giống như C = C ^ 2 |
| |= | Phép OR bit. | C |= 2 là giống như C = C | 2 |

## **Toán tử hỗn hợp**

**Toán tử điều kiện ( ? : )**

Toán tử điều kiện là một loại toán tử đặc biệt vì nó bao gồm ba thành phần cấu thành biểu thức điều kiện. Cú pháp:

<biểu thức 1>   ?   <biểu thức 2>   :   <biểu thức 3>;

* **biểu thức 1:** Biểu thức logic. Trả trả về giá trị True hoặc False
* **biểu thức 2:** Là giá trị trả về nếu <biểu thức=”” 1=””>xác định là True</biểu>
* **biểu thức 3:** Là giá trị trả về nếu <biểu thức=”” 1=””>xác định là False</biểu>

Ví dụ:

|  |  |
| --- | --- |
|  | **public** **class** Test {  **public** **static** **void** main(String[] args) {  **int** a = 20;  **int** b = 3;            String s = (a % b == 0) ? "a chia het cho b" : "a khong chia het cho b";          System.out.println(s);      }  } |

## **Toán tử instanceof**

Toán tử này chỉ được sử dụng cho các biến tham chiếu đối tượng. Toán tử kiểm tra có hay không đối tượng là một kiểu cụ thể (kiểu class hoặc kiểu interface). Toán tử instanceof được viết như sau:

( Object reference variable ) instanceof (class/interface type)

Ví dụ:

|  |
| --- |
| **public** **class** Test {    **public** **static** **void** main(String args[]){        String name = "gpcoder";        // Kiểm tra name có phải là kiểu chuỗi hay không  **boolean** result = name **instanceof** String;        System.out.println( result ); // kết quả: true     }  } |

## **Thứ tự ưu tiên của các toán tử**

|  |  |  |
| --- | --- | --- |
| LOẠI | TOÁN TỬ | THỨ TỰ ƯU TIÊN |
| Postfix | () [] . (toán tử dot) | Trái sang phải |
| Unary | ++ – – ! ~ | Phải sang trái |
| Tính nhân | \* / % | Trái sang phải |
| Tính cộng | + – | Trái sang phải |
| Dịch chuyển | >> >>> << | Trái sang phải |
| Quan hệ | > >= < <= | Trái sang phải |
| Cân bằng | == != | Trái sang phải |
| Phép AND bit | & | Trái sang phải |
| Phép XOR bit | ^ | Trái sang phải |
| Phép OR bit | | | Trái sang phải |
| Phép AND logic | && | Trái sang phải |
| Phép OR logic | || | Trái sang phải |
| Điều kiện | ?: | Phải sang trái |
| Gán | = += -= \*= /= %= >>= <<= &= ^= |= | Phải sang trái |
| Dấu phảy | , | Trái sang phải |

## **Thay đổi thứ tự ưu tiên của các toán tử**

Để thay đổi thứ tự ưu tiên trên một biểu thức, bạn có thể sử dụng dấu ngoặc đơn ():

* Phần được giới hạn trong ngoặc đơn được thực hiện trước.
* Nếu dùng nhiều ngoặc đơn lồng nhau thì toán tử nằm trong ngoặc đơn phía trong sẽ thực thi trước, sau đó đến các vòng phía ngoài.
* Trong phạm vi một cặp ngoặc đơn thì quy tắc thứ tự ưu tiên vẫn giữ nguyên tác dụng.

Ví dụ:

|  |
| --- |
| **public** **class** App {  **public** **static** **void** main(String[] args) {  **int** a = 20;  **int** b = 5;  **int** c = 10;            System.out.println("a + b \* c   = " + (a + b \* c));          System.out.println("(a + b) \* c = " + ((a + b) \* c));          System.out.println("a / b - c   = " + (a / b - c));          System.out.println("a / (b - c) = " + (a / (b - c)));      }  } |

Output:

a + b \* c = 70  
(a + b) \* c = 250  
a / b – c = -6  
a / (b – c) = -4

## **So sánh toán tử logic và toán tử thao tác bit**

### **So sánh || và |**

Như bạn thấy, với toán tử **||**, chương trình sẽ không thực thi kiểm tra vế sau nếu vế trước cho kết quả là true. Còn với toán tử **|**, chương trình sẽ luôn luôn thực thi cả 2 vế.

### **So sánh && và &**

Tương tự, với toán tử **&&**, chương trình sẽ không thực thi kiểm tra vế sau nếu vế trước cho kết quả là false. Còn với toán tử **&**, chương trình sẽ luôn luôn thực thi cả 2 vế.

## **Thăng hạn dữ liệu (Numeric Promotion)**

Như chúng ta đã biết, mỗi kiểu số nguyên thủy (primitive number) có một độ dài bit xác định. Chúng ta không cần biết kích thước chính xác của các loại này, nhưng nên biết loại nào lớn hơn các loại khác. Ví dụ, giá trị **long** chiếm nhiều dung lượng hơn giá trị **int**, int chiếm nhiều dung lượng hơn giá trị **short**, …

Các quy tắc mà Java sẽ tuân theo khi áp dụng các toán tử cho các kiểu dữ liệu:

* Nếu hai giá trị có kiểu dữ liệu khác nhau, Java sẽ tự động thăng hạng một trong các giá trị lên giá trị lớn hơn trong hai kiểu dữ liệu. Ví dụ: int + long -> long.
* Nếu một trong các giá trị là nguyên và giá trị kia là dấu phẩy động, Java sẽ tự động chuyển giá trị nguyên sang kiểu dữ liệu của giá trị dấu phẩy động. Ví dụ: int + double -> double.
* Các kiểu dữ liệu nhỏ hơn, cụ thể là byte, short và char, lần đầu tiên được thăng cấp thành int bất kỳ khi nào chúng được sử dụng với toán tử số học nhị phân Java, ngay cả khi không có toán hạng nào là int.
* Sau khi tất cả việc thăng hạng đã xảy ra và các toán hạng có cùng kiểu dữ liệu, giá trị kết quả sẽ có cùng kiểu dữ liệu với các toán hạng được thăng hạng của nó.

Lưu ý:

* Rule cuối cùng không áp dụng đối với các toán tử Unary. Chẳng hạn, áp dụng ++ trên short, sẽ cho kết quả là short.
* Đối với các toán tử kết hợp (compound operator): trình biên dịch sẽ tự động chuyển giá trị kết quả sang kiểu dữ liệu của giá trị ở phía bên trái của toán tử ghép. Ví dụ:

# **Making Decisions**