日本電子専門学校ゲーム制作科 2 年次

Java Ⅰ 言語比較

Java と C++の言語仕様の違い

Norihisa Ishida

2015/08/13

このドキュメントは Java と C++の言語表現の違いを解説しています。

１．　基本データ型

Java

整数型
long　　８バイト
int　　４バイト
short　２バイト
byte　　１バイト

文字型
char　　２バイト(符号なし)

浮動小数点型
float　４バイト
double　８バイト

論理型
boolean １バイト

C++

整数型
long　　４バイト
int　　４バイト
short　２バイト

文字型
char　　１バイト

浮動小数点型
float　４バイト
double　８バイト

論理型
bool　　１バイト

２．　リテラル

Java

整数

1231（123L）	long 型
123，07，0xA	int，short，byte 型

文字

‘A’，‘あ’	char 型

文字列型

“abc”	String 型

浮動小数点型

12.3f	float 型
12.3d，12.3，1e2	double 型

論理型

true，false	boolean 型

C++

整数

123，07，0xA	long，int，short 型

文字

‘A’	char 型

文字列型

“abc”	char 配列型

浮動小数点型

12.3f	float 型
12.3，1e2	double 型

論理型

true，false	bool 型

３．　演算

Java

算術演算(すべて 2 項演算)
+ , − , * , / , %
オペランドは整数、浮動小数点に適用
演算結果は優先度による拡大変換が適用

比較・等価演算(すべて 2 項演算)
< , > , <= , >= , == , !=
オペランドは整数、浮動小数点、文字、
インスタンス値に適用
演算結果はいずれも true, false の 2 値

論理演算(OR と AND は 2 項演算、NOT は単項演算)
|| , && , !
オペランドは boolean
演算結果も boolean

ビット演算(~以外は 2 項演算)
& , | , ^ , ~ , << , >> , >>>
オペランドは整数、文字
演算結果は整数

3 項演算
オペランド 1?オペランド 2 :オペランド 3
オペランド 1 は boolean、それ以外はすべての型

C++

算術演算(すべて 2 項演算)
+ , − , * , / , %
オペランドは整数、浮動小数点に適用(%のみ整数と文字)
演算結果は優先度による拡大変換が適用

比較・等価演算(すべて 2 項演算)
< , > , <= , >= , == , !=
オペランドは整数、浮動小数点、文字、
に適用
演算結果はいずれも 1(true), 0(false)の 2 値

論理演算(OR と AND は 2 項演算、NOT は単項演算)
|| , && , !
オペランドは bool または整数・文字
演算結果は 1(true)または 0(false)

ビット演算(~以外は 2 項演算)
& , | , ^ , ~ , << , >>
オペランドは整数、文字
演算結果は整数

3 項演算
オペランド 1?オペランド 2:オペランド 3
オペランド 1 は bool、それ以外はすべての型

４．　制御構文

Java

if 文
if(boolean 値 1){
　　boolean値1がtrueのときの処理;
}else if(boolean 値 2){
　　boolean 値 1 が false でかつ
　　boolean値2がtrueのときの処理;
}else{
　　boolean 値 1、boolean 値 2 が
　　ともに false のときの処理;
}

switch 文
switch(値){
case ラベル値 1: 処理 1; break;
case ラベル値 2: 処理 2; break;
default:　　処理 N;
}

処理 1 は値がラベル値 1 に等しいとき処理、処理 2 は値がラベル値 2 に等しいとき処理、default は値が値 1～値 N-1 のどのラベル値とも等しくないときに処理。

値は long, int, short, byte, char, String 型のいずれか。

C++

if 文
if(bool 値 1){
　　bool 値 1 が true のときの処理;
}else if(bool 値 2){
　　bool 値 1 が false でかつ
　　bool 値 2 が true のときの処理;
}else{
　　bool 値 1、bool 値 2 が
　　ともに false のときの処理;
}
bool 値は int や char での 0 または 0 以外を false, true に代用可能(以下すべて)。

switch 文
switch(値){
case ラベル値 1: 処理 1; break;
case ラベル値 2: 処理 2; break;
default:　　処理 N;
}

処理 1 は値がラベル値 1 に等しいとき処理、処理 2 は値がラベル値 2 に等しいとき処理、default は値が値 1～値 N-1 のどのラベル値とも等しくないときに処理。

値は int, short, char 型のいずれか。

Java

while 文
while(boolean 値){
　　処理;
}
boolean 値が true のとき処理を繰り返す。

do-while 文
do{
　　処理;
}while(boolean 値);
boolean 値が false のとき処理を 1 度だけ行う。true のときは 2 回目以降の処理を繰り返す。

for 文
for(初期処理; boolean 値; 反復処理){
　　処理;
}
初期処理は最初に一度だけ処理される。反復処理は boolean 値が true のときに「処理」の後処理される。「処理」は boolean 値が true のとき最初に処理される。実行順序は以下の通り。

初期処理→boolean 値の評価→true のとき処理→反復処理→boolean 値の評価→true の…(以降繰り返し)

いずれも評価値が false のときは for 文を終了。

C++

while 文
while(bool 値){
　　処理;
}
bool 値が true のとき処理を繰り返し行う。

do-while 文
do{
　　処理;
}while(bool 値);
bool 値が false のとき処理を 1 度だけ行う。true のときは 2 回目以降の処理を繰り返す。

for 文
for(初期処理; bool 値; 反復処理){
　　処理;
}
初期処理は最初に一度だけ処理される。反復処理は bool 値が true のときに「処理」の後処理される。「処理」は bool 値が true のとき最初に処理される。実行順序は以下の通り。

初期処理→bool 値の評価→true のとき処理→反復処理→bool 値の評価→true の…(以降繰り返し)

いずれも評価値が 0 以外(false)のときは for 文を終了。

Java

for-each 文
for(データ型 変数:集合データ){
　　処理;
}
処理は集合データ先頭から末尾まで逐次処理。集合データは配列、enumeration、コレクションが使用可能。

C++

for-each 文
for(データ型 変数:集合データ){
　　処理;
}
処理は集合データ先頭から末尾まで逐次処理。集合データは配列、enum、コレクションが使用可能。

５．　データ構造

Java

配列型

宣言：データ型[] 変数名;
用例：int[] x;

実体宣言および初期化：
変数名=new データ型[要素数];
変数名=new データ型[]{データリスト};
データ型[] 変数名={データリスト};
用例：
x = new int[3];
x = new int[]{1,2};
int[] x = {1,2};

宣言時の[]は変数の後につけることもできる。次元数を増やす場合は[][]…のように続けて記述する。初期化時には{}の内側にさらに{}を記述し、要素位置を明示する。
例：
int[][] x = {{1,2},{3,4},{5,6}};

C++

配列型

宣言：データ型 変数名[要素数];
用例：int x[3];

初期化：
データ型 変数名[]={データリスト};
用例：
int x[] = {1,2};

宣言時の[]は変数の後につけることもできる。次元数を増やす場合は[][]…のように続けて記述するが、左端以外の要素数を省略することはできない。
例：
int x[][2]={{1,2},{3,4},{5,6}};

Java

構造体、共用体

存在しない。クラスまたはインターフェースを使う。

ビットフィールド

存在しない。

列挙型

存在しない。インターフェースの Enumeration またはクラスの Enum を使う。

ポインタ型、アドレス型

存在しない。アドレスの概念もないが toString メソッドで文字列による識別は可能。

C++

構造体、共用体

定義：

```
struct タグ名{
        データ型 メンバ変数1;
        データ型 メンバ変数2;
};
union タグ名{
        データ型 メンバ変数1;
        データ型 メンバ変数2;
};
```

宣言：

```
struct タグ名 変数名;
タグ名 変数名;
```

ビットフィールド：
struct TagName{
	データ型 メンバ変数1:バイト数1;
	データ型 メンバ変数2:バイト数2;
};
現在のコンパイラの大半がこのコードを無視している。

列挙体
enum タグ名{列挙1,列挙2};

ポインタ型・アドレス型
型名* 変数名;

クラス型
定義：
class クラス名{
	データ型 メンバフィールド名;
	コンストラクタ名(引数リスト){
		処理;
	}
	データ型 メソッド名(引数リスト){
		処理;
	}
}
メンバーフィールドの初期化は可能
スタティックメンバはフィールド、メソッドとスタティックメソッド内のメンバ変数にのみ可能
final はフィールドについては宣言時またはコンストラクタで値を決めなければならない。
メソッドについてはオーバーライド禁止、クラスについては継承禁止の制約になる。
abstract はクラスについている場合のみメソッドにもつけられる。

public はパッケージ外でのアクセスを可能とする。
private はクラス内でのみアクセス可能。
protected は同一パッケージ内とパッケージ外で継承関係にあるサブクラスでアクセス可能。
修飾子なしは同一パッケージ内でのアクセスが可能。

interface 型

ジェネリクス
Iterable インターフェースを実装したすべてのクラスで、そのデータの型を明示するジェネリクスを使用できる。

宣言：クラス名<格納するデータの型> 変数名;
用例：ArrayList<String> al = new ArrayList<String>();

ただしクラス名の部分はコレクター等の Iterable インターフェースを実装したクラスに限る。

６．　クラス
７．　あ
８．　あ