isとas

実装の確認

インターフェイスやボクシング変換を学習し、それを実用に移すとある問題が見つかります
それは、アップキャストは常に保証されますが、ダウンキャストがまったく保証されないということです

オブジェクト指向プログラミングでは、実行時まで型がわからないケースが多く
参照型を目的の型にダウンキャストする場合、確実性を持ちません
例えば、次のプログラムを見てください

```
class A {}
class B {}

class Test {
 static void Main() {
  System.Object objA = new A();
  B objB = (B)objA;
 }
}
```

これは、A クラスを Object 型に変換した objA という参照型があります
これを B クラスに変換しようとしていますが、Object 参照が B 型の実装を持つかどうかは
コンパイル時点では判断できません（このように、単純な場合は見ればわかりますが…）

そのため、コンパイルすることはできますが、実行すると処理が停止します
A クラスは B クラスと互換性がないため、A クラスの実装を持つ参照型 objA は
B クラスにキャストすることはユーザー定義変換を除いてありえません
この場合「例外」と呼ばれる割り込みが発生し、次のような文字を残して強制終了されます

```
未処理の例外 : System.InvalidCastException: 種類 System.InvalidCastException の
例外がスローされました。
   at Test.Main()
```

例外の捕捉（処理方法）については、後ほど詳しく説明しますが
このように、保証されない変換というのは頻繁に行われるケースもあり
その度に、例外を発生させ処理を停止させるようでは、正常なプログラムとは言えません

そこで、キャストする前に**実装を確認**する方法が存在します
もちろん、実行時に動的に参照型の実装を確認できます
実装の問い合わせには **is** 演算子を用います

expression is type

expression には参照型の式、type には型を指定します
expression が type である場合は true そうでなければ false を返します

これをキャストする前に行い、実装を確認すれば
実装を持つ場合と、持たない場合で異なる処理を施すことができます

```
class A {}
class B {}

class Test {
 static void Main() {
  System.Object obj = new A();
  Check(obj);
  obj = new B();
  Check(obj);
 }
 static void Check(System.Object obj) {
  if (obj is B) System.Console.WriteLine("This is B");
  else System.Console.WriteLine("This is not B");
 }
}
```

このプログラムの Test.Check() メソッドは
渡された参照型が B クラスの実装を持つかどうかを確認するメソッドです
渡された参照型を is 演算子で確認してその結果を文字列で出力します

もちろん、インターフェイスの実装を持つかどうかも調べることができます
おそらく、動的に参照型を変換する最も多いケースの一つが
何らかのクラスの参照をインターフェイス型にキャストするプログラムでしょう
このようなケースに is 演算子は大きく貢献してくれます

```
interface KittyStandard {
 string Name { get; }
}

enum KittyName { RENA , YUKI , MIMI }

class Kitty : KittyStandard {
 private string name;
 public Kitty(KittyName name) {
  this.name = name.ToString();
 }
 string KittyStandard.Name { get { return name; } }
}
```

```
class Test {
 static void Main() {
  System.Object kitty = new Kitty(KittyName.MIMI);
  CheckKittyStandard(kitty);

  kitty = new Test();
  CheckKittyStandard(kitty);
 }
 static void CheckKittyStandard(System.Object obj) {
  if (obj is KittyStandard)
   System.Console.WriteLine(((KittyStandard)obj).Name);
  else System.Console.WriteLine("This is not KittyStandard");
 }
}
```

Test.CheckKittyStandard() メソッドは渡された参照型を is 演算子で調べます
参照型が KittyStandard インターフェイスを実装している場合と
そうでない場合に応じて、コンソールに結果を示す文字列を出力します

このように、is は参照型を目的の型に変換できるかどうかを動的に調べられます
ただし、is 演算子が調べるのは参照変換、ボクシング変換、アンボクシング変換のみです
is 演算子はユーザー定義変換など、その他の変換は**考慮しません**

安全な変換

is 演算子は実装を調べ、その結果をブールで返すというものでしたが
もともと、調べる目的がキャストならば **as** 演算子を用いるべきです

as 演算子は明示的なキャストを安全に変換する機能を提供します
この演算子を用いてキャストした場合、例外は発生しません
もし、目的の型の実装を持たない場合は **null を返す**のです

expression as type

expression には変換する参照型を、type には変換する型を指定します
この結果、変換できる場合はその型を、できない場合は null を返します

```
class A {}
class B {}
class C : B {}

class Test {
 static void Main() {
  System.Object objA = new A();
  System.Object objC = new C();

  B objB = objA as B;
  Check(objB);

  objB = objC as B;
  Check(objB);
 }
 static void Check(System.Object obj) {
  if (obj == null) System.Console.WriteLine("this can't cast B type");
  else System.Console.WriteLine("Success");
 }
}
```

B objB = objA as B は、objA を B に as 演算子を用いて明示的に変換しようとしています
しかし objA は B を実装しないので、この結果は null となるでしょう
通常のキャスト変換では、例外となりますが、as を用いた場合は発生しないのです

ただし、null を参照した場合は例外を発生されるので
変換した参照にアクセスするならば、やはり if で調べる必要があります

is と as 演算子で重要な違いは MSIL でも見られます
キャストが目的の場合 is 演算子はブールを調べた後にキャストすることになりますが
この時、実装を調べる時に isinst コードを用いて実装を確認し
さらに、キャストする場合は castclass コードでキャストするという2重のキャストになります
しかし as 演算子ならば isinst コードを用いて1度キャストするだけなので
is に比べてキャストだけが目的の場合は as の方が効率的と考えられます

また、as 演算子は参照変換とボクシング変換しか行いません
ユーザー定義変換など、その他の変換には通常のキャストを用いるしかありません

---

is

expression is type

参照型の実装を動的に問い合わせ、その結果をブーリアンで返します
expression が type 型の実装を持つならば true、そうでなければ false を返します

expression - 調べる参照型の式を指定します
type - 型を指定します

as

expression as type

参照変換、またはボクシング変換を明示的に行います
expression が type 型の実装を持たなければ null が返ります

expression - 変換する式を指定します
type - 型を指定します

前のページへ　戻る　次のページへ