# 例外のスロー

## 例外を発生させる

様々な処理をパッケージ化した独自のメソッドなどを制定した時
利用者が予想しない値や不正な値をパラメータで使用するようなこともあります
例えば、インタプリンタを開発して構文エラーが見つかった場合なども同じことが言えます
この時、エラーが致命的ならばあなたのメソッドは例外を投げて処理を停止させるでしょう

例外を明示的に発生させる場合は **throw** キーワードを用います
これを用いて明示的に例外を発生させることを「例外をスローする」と表現します

**throw [expression];**

expression は Exception 型に互換性のある例外の式を指定します
System.Exception については、.NET クラスライブラリのドキュメントを参照してください
この場では、とりあえず Exception はコンストラクタで文字列を受け取り
その文字列は Message プロパティで活かされるということだけを知ってください

**public Exception();**
**public Exception(string message);**

実際には Exception は四つのコンストラクタを持っていますが、省略します

```
class Test {
 static void Main(string[] args) {
  try { Kitty(); }
  catch (System.Exception err) {
   System.Console.WriteLine(err.Message);
  }
 }
 public static void Kitty() {
  throw new System.Exception("Kitty on your lap");
 }
}
```

このプログラムの Kitty() メソッドは明示的に例外を発生させます
例外がスローされるので、呼び出しもとでは catch が実行されます

このプログラムでは Exception 例外をそのままスローしていますが
Exception の派生クラスであれば、独自のクラスを例外としてスローすることもできます

構文上の規則はありませんが、習慣として例外クラスは
クラス名の終端に Exception という名前をつけ
System.Exception の四つのコンストラクタを再現することが推奨されます
実用する独自の例外クラスであれば、このような規則を尊重する必要があるでしょう

```
class KittyException : System.Exception {
 public override string Message {
  get { return "Kitty on your lap"; }
 }
}

class Test {
 static void Main(string[] args) {
  try { Kitty(); }
  catch (KittyException err) {
   System.Console.WriteLine(err.Message);
  }
 }
 public static void Kitty() {
  throw new KittyException();
 }
}
```

KittyException クラスは Exception クラスを継承した例外クラスです
この例外をスローすることで、独自の例外処理を行うことができます
必要であれば、オーバーライド以外に独自のメソッドを追加して
特殊化した処理に対応することも可能となるでしょう

例外は、catch から、さらにそれを含む上位層へスローすることも可能です
これを、**例外の再スロー**と呼びます
例外の処理中に、さらに例外を再スローするには式を省略した throw を指定します
ただし、式を省略した throw は catch 句の内部でしか指定できません

一般に、例外が最初に発生したポイントを**スローポイント**と呼びます
そして、例外がスローされてから適切な例外ハンドラに到達するまでの手続き
すなわち catch の検索過程を**例外の伝播**と呼びます

```
class Test {
 static void Main(string[] args) {
  try { A(); }
  catch (System.Exception err) {
   System.Console.WriteLine(err.Message);
  }
 }
 static void A() {
```

```
  try { B(); }
  catch {
   System.Console.WriteLine("Silver Gene");
   throw;
  }
 }
 static void B() {
  throw new System.Exception("Kitty on your lap");
 }
}
```

B() メソッドは例外をスローします
すると、A() メソッドの包括的 catch がこれを捕捉します
例外が捕捉されれば、これ以上例外が捕捉されるようなことはありません
つまり、Main() メソッドまで制御が戻るようなことはないはずなのです

しかし、A() メソッドの catch は throw ステートメントを指定することによって
さらに、例外を再スローして Main() メソッドの try 句に例外をスローします
このプログラムを実行した結果を見れば、動作を理解できるでしょう

**Silver Gene**
**Kitty on your lap**

A() メソッドの再スローをコメント化してコンパイルして実行すれば
Main メソッドの catch に制御が移らないことを確認できるでしょう

例外の捕捉構造は、場合によって複雑化することもあります
このような場合に備え、技術者は例外の伝播がどのような手続きをとるかを知るべきです

例外の伝播は、まずスローポイントを包含する try を確認し
もっとも内側の try から外側に向かって catch を検索していきます
catch の検索は、書かれた順番に評価して適切なハンドラを探します

catch の検索ができなかった場合は、次の try 句に移行しますが
各 try 句に finally が記述されている場合は、必ずそれを実行します
これを繰り返しながら、例外は上位のメソッドに向かって帰還していきます
現在のスレッド、プロセスに適切な例外処理がなければ、規定の方法で強制終了します

```
class Test {
 static void Main(string[] args) {
  try { A(); }
  catch (System.Exception err) {
   System.Console.WriteLine(err.Message);
  }
  finally {
   System.Console.WriteLine("Silver Gene");
  }
 }
 static void A() {
  try { B(); }
  catch (System.DivideByZeroException err) {
   System.Console.WriteLine(err.Message);
  }
  finally {
   System.Console.WriteLine("Di Gi Charat");
  }
 }
 static void B() {
  try {
   int x = 0;
   x = 10 / x; //DivideByZeroException 発生
  }
  catch (System.IndexOutOfRangeException err) {
   System.Console.WriteLine(err.Message);
  }
  finally {
   System.Console.WriteLine("Kitty on your lap");
  }
 }
}
```

このプログラムは、B() メソッドの try 句がスローポイントとなります
B() メソッドは DivideByZeroException を処理しないので、例外は捕捉されません
そのため、B() メソッドからスタックをさかのぼって検索が続行されます

すると A() メソッドの try 句で捕捉できるハンドラが発見され例外が捕捉されます
もし、この場でも見つからなければ、さらにスタックを辿って上位層を検索します
例外は捕捉されたので、再スローされなければ上位が検索されることはありませんが
finally には必ず制御が移されるので、これを追って検索の様子を確認できます

**Kitty on your lap**
**0 で除算しようとしました。**
**Di Gi Charat**
**Silver Gene**

プログラムを実行すると、このような結果を得ることができます
スローポイントからメソッドを呼び出した順にスタックを登っていることがわかります

---

# throw

**throw [expression];**

例外をスローします

**expression** - スローする例外クラスの式を指定します

---

前のページへ　戻る　次のページへ