例外

エラーの処理

オブジェクト指向プログラミングでは、実行時に動的に発生したエラーを補足するため
統一したエラー処理システムとして「例外」をサポートします
例外は C++ や Java などのプログラム言語経験者にとっては馴染みのあるものでしょう

例外は、私たちプログラマが予想しない処理に対して明示的に発生させるものと
正常に処理できない状態に発生する規定の例外があります
これらが発生した場合、プログラムは割り込みを発生させ本来のプログラムを中断します

規定の例外とは、例えば new 演算子を用いてヒープにインスタンスを生成しようとした時
ヒープの容量が足りなくて、インスタンスを生成できないといった実行時の例外です
たとえば、次のプログラムは代表的な例外の一つです

```
class Test {
 static void Main() {
  int zero = 0;
  System.Console.WriteLine(10 / zero);
 }
}
```

数学の問題上 0 の除算はできません
0 の除算が定数であればコンパイル時にエラーを出せますが
これが動的に、つまり変数によって実行時に発生した場合は例外となります

例外を捕捉することによって、実行時に発生した致命的なエラーを受け
特定の終了処理を施してスマートな、統一されたエラー処理を実現できます

さらに、例外はエラー処理をクラス利用者に強制する効果もあります
C 言語でシステムに強く関連したプログラムを経験している方はよくわかると思いますが
関数がエラーを返すという場合、プログラマはエラーをチェックする必要がありますが
これを怠った場合、その後の処理で致命的なクラッシュが発生する可能性があります
しかし例外は、利用者からエラーチェックを解放する役割もあるのです

例外が発生した場合、これを処理するためには例外を捕捉する必要があります
例外の捕捉には **try** と **catch** キーワードを用います

try try-block
catch (exception-declaration) catch-block
...

try-block は例外の発生を監視するコードブロックを指定します
このブロック内のコードが例外を発生（スロー）させた場合は捕捉します

exception-declaration は捕捉する**例外の型**を指定します
発生した例外がこの型であれば、catch-block を実行します
catch ブロックは、続けていくつでも指定することができます
catch-block は例外の処理用コードを記述します

例外の型とは **System.Exception** を基底とするクラスを指します
発生する例外には必ず型があり、catch は例外のインスタンスを受け取ります
try ブロック内で例外が発生すると、プログラムは catch ブロックを検索します
exception-declaration で発生した例外と一致する型が見つかれば、そのブロックに制御を移します

例外を処理する catch ブロックでは exceptioni-declaration で受け取った
発生した例外型の**例外変数**をローカル変数として扱い
例外変数から発生した例外に関連する情報を受け取ることができます
あらゆる例外は System.Exception を基底クラスとしなければなりません

public class Exception : ISerializable

Exception クラスで重要なのは、最低限例外の原因を
catch の例外変数が把握できるように設計されていることです
例外の説明は Message プロパティが文字列として返してくれます

public virtual string Message {get;}

独自の例外を作る場合も、このプロパティをオーバーライドすることによって
利用者が例外のメッセージを文字列として取得して表示することができます

```
class Test {
 static void Main() {
  try {
   int zero = 0;
   System.Console.WriteLine(10 / zero);
  }
  catch(System.Exception err) {
   System.Console.WriteLine(err.Message);
  }
 }
}
```

先ほどのプログラムと同様に、このプログラムでも 0 の除算が発生します
しかし try によって、例外が発生する可能性のあるコードを監視し
0 で除算しようとした場合は例外をキャッチするようにプログラムされています

System.Console.WriteLine(10 / zero); を実行しようとすると例外が発生するので
プログラムは即座に処理を中断し catch ブロックに制御を移します
正確には、0 の除算の例外は **System.DivideByZeroException** クラスです
ただし、この例外は基底クラス Exception に暗黙的に変換できるため
catch の検索時に Exception 型のブロックが実行されたという仕組みです
もちろん、次のようにプログラムしても結果は同じです

```
class Test {
 static void Main() {
  try {
   int zero = 0;
   System.Console.WriteLine(10 / zero);
  }
  catch(System.DivideByZeroException err) {
   System.Console.WriteLine(err.Message);
  }
 }
}
```

DivideByZeroException 例外は、0 の除算が発生した時に生成される例外です
例外は必ず処理しなければならないというものではないので
try 区の 0 の除算だけを処理する場合はこれを指定します
当然、何らかのエラー専用の例外は、そのエラーに応じた特殊化した機能を持つこともあります

catch 句は必要な限り何個でも記述することができます
例外の型に応じて、それに適したエラー処理を施すことができるのです

```
class Test {
 static void Main(string[] args) {
  try {
   if (args[0] == "A") {
    int x = 0;
    x = 10 / x;
   }
   else if (args[0] == "B") {
    System.Object obj = null;
    string str = obj.ToString();
   }
  }
  catch (System.IndexOutOfRangeException err) {
   System.Console.WriteLine(err.Message);
  }
  catch (System.DivideByZeroException err) {
   System.Console.WriteLine(err.Message);
  }
  catch (System.NullReferenceException err) {
   System.Console.WriteLine(err.Message);
  }
 }
}
```

このプログラムの try は、複数の catch を持ちます
try で発生した例外は、その型に応じて適切な処理を施されるのです

IndexOutOfRangeException 例外は、配列で不正なインデックスにアクセスした場合の例外です
このプログラムでは、コマンドライン引数を与えないで実行するとこれが発生します
NullReferenceException は null 参照をアクセスした場合に発生します
MS-DOS プロンプトで次のように実行すると、結果はこうなりました

```
C:¥program¥C#>test
種類 System.IndexOutOfRangeException の例外がスローされました。

C:¥program¥C#>test A
0 で除算しようとしました。

C:¥program¥C#>test B
オブジェクトのインスタンスが必要とされる場所で、値 'null' が見つかりました。
```

それぞれ、適切な例外処理が実行されていることを確認できますね

もし、例外が発生する型や情報に興味がない場合は
このような処理はむしろ面倒に感じるかもしれません
本来、プロジェクトレベルの正式はソフトウェアは適切で親切なエラー処理をするべきですが
その必要がない場合は**包括的 catch 句**という構文を使います

try try-block catch catch-block

try-block には例外を監視するコードを
catch-block には例外が発生した時の処理用コードを指定します

これまでの、例外変数を受ける catch 句を**具体的 catch 句**と呼び
包括的 catch 句はこれに対して、例外変数を受け取りません
どのような例外にも対応し、文字どおり包括的にエラー処理を行います
具体的 catch 句との併用も可能ですが、その場合は必ず最後に包括的 catch 区を指定します

```
class Test {
 static void Main(string[] args) {
  try {
   if (args[0] == "A") {
    int x = 0;
    x = 10 / x;
   }
   else if (args[0] == "B") {
    System.Object obj = null;
    string str = obj.ToString();
   }
  }
  catch {
   System.Console.WriteLine("例外がスローされました");
  }
 }
}
```

このプログラムは、どのような例外が発生しても catch が実行されます
例外変数を指定しない包括的 catch は、try 句で発生した例外を全て受け取ります

後片付け

try 句が無事に終了するか、例外が発生することによって
場合によっては、何らかの後処理が必要になることがあります

例えば、try 句でディスクファイルを操作している場合
最終的には、バッファをフラッシュしてストリームを閉じる動作が必要になるでしょう
このような場合、例外に応じて同じ処理を書くというのは合理的ではありません
そこで、try 句の後処理として **finally** ブロックが存在します

try try-block finally finally-block

try-block には例外を監視するブロックを
finally-block は、制御が try から離れた時に実行するステートメントを記述します

このことから、try ステートメントは3つの構文に分けられます
try-catch と try-finally、そして try-catch-finally という構文です

try ブロック catch 句群
try ブロック finally 句
try ブロック catch 句群 finally 句

catch を指定する場合、finally は必ず最後に指定します
finally は catch と異なり必ず実行されるというところに注目してください

```
class Test {
 static void Main(string[] args) {
  try {
   System.Console.WriteLine(args[0]);
  }
  catch (System.Exception err) {
   System.Console.WriteLine(err.Message);
  }
  finally {
   System.Console.WriteLine("Kitty on your lap");
  }
 }
}
```

このプログラムは、コマンドラインから引数を渡さなければ例外が発生します
例外が発生してもしなくても、try 句が終了するか
または、catch を終了すると制御が finally 句に移行します
どのようなケースでも、必ず finally が実行されることが確認できます

---

try

try try-block
catch (exception-declaration) catch-block
...

try try-block catch catch-block

try try-block finally finally-block

例外の発生を監視し、例外を捕捉してエラー処理を行います
exception-declaration を指定しない catch の場合は
全ての例外を捕捉する包括的 catch となります

try-block - 例外を監視するコードブロックを指定します
exception-declaration - 捕捉する例外と、例外変数を指定します
catch-block - 例外発生時に実行するブロックを指定します
finally-block - try 終了後に実行するブロックを指定します
