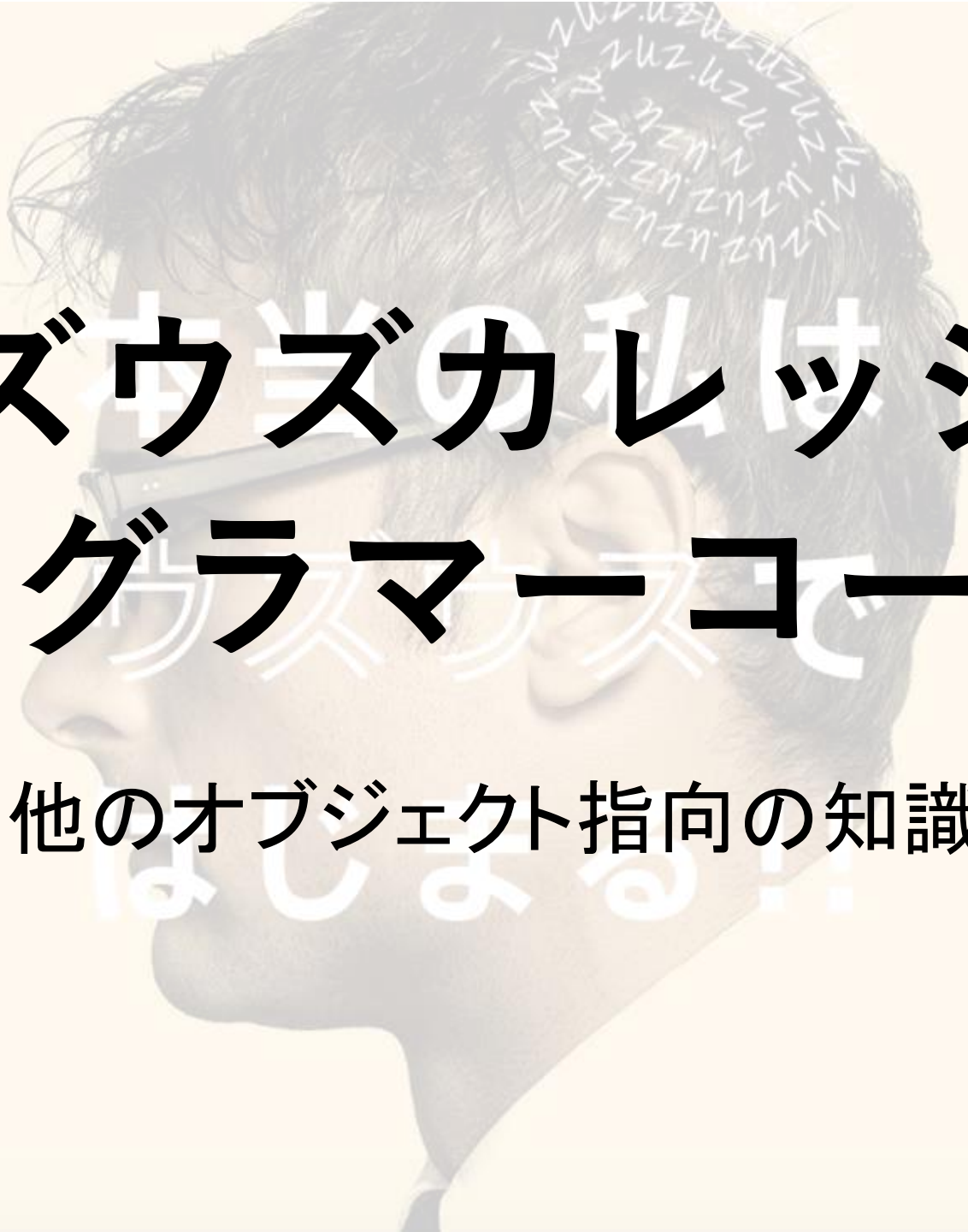
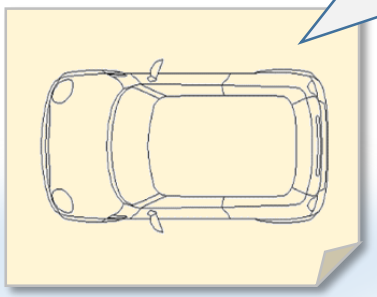


ウズウズカレッジ プログラマーコース

その他のオブジェクト指向の知識

抽象メソッド
車ならこんな機能（メソッド）があるべきだ！
（具体的な処理内容はオーバーライドして定義してね！）

```
//acceleratorメソッド（現在の速度を上げる）
abstract void accelerator() ;

//brakeメソッド（現在の速度を下げる）
abstract void brake() ;

//paintメソッド（色を塗る）
abstract void paint( String cl ) ;
```



Sample2_07_1_AbstractCar
抽象クラス
仕様のみを定めた
インスタンス不可で継承前提のクラス

継承

Sample2_07_1_Car
インスタンス

抽象メソッドをオーバーライド していないとエラー！

≪抽象クラス（abstract）≫

□「このクラスを継承したサブクラスには〇〇というメソッドがあるべきだ」という**仕様のみを定義した継承前提のクラスを抽象クラス**と言います。
抽象クラスは**インスタンス化されて使用されることを想定していない**ため、インスタンス化しようとするとエラーになります。
classの前に**abstract修飾子**を付与することで抽象クラスとして扱うことが可能になります。

□抽象クラスでは**抽象メソッド**という機能の仕様を定義するためだけの**具体的な処理内容のないメソッド**を定義することが可能です。
以下のように記述し、大きな特徴として**{}が存在せず**、代わりに「;」が置かれます。

[アクセス修飾子] abstract 戻り値の型 メソッド名(仮引数の型 仮引数名) ;

□抽象メソッドは**オーバーライドして使用されることを前提**とします。
抽象メソッドをオーバーライドして具体的な処理内容を定義することを**実装**と言い、抽象クラスを継承して作成されたインスタンス内に**実装されていない抽象メソッドが１つでもあればインスタンス化の際にエラーとなります。**
抽象クラスを継承した全クラスに抽象メソッドの実装を強制することができるため、そのクラスで定義すべき機能が漏れたり、好き勝手な書き方で定義されないよう抑制できるといったメリットがあります。

～抽象クラスのメリット～

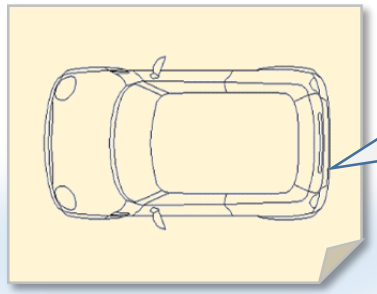
バラバラ・・・

accele
メソッド

speedUp
メソッド

kasoku
メソッド

～抽象クラスのメリット～



抽象クラス

抽象メソッド

```
//acceleratorメソッド（現在の速度を上げる）
abstract void accelerator() ;
```

継承

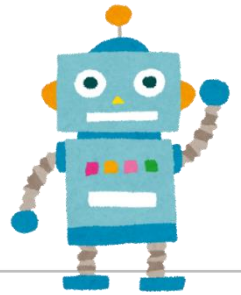
accelerator
メソッド

accelerator
メソッド

accelerator
メソッド

引数がStringだから
起動するのは・・・

println(“Hello World !!”)



PrintStreamクラス

printlnメソッド
（引数：int型）

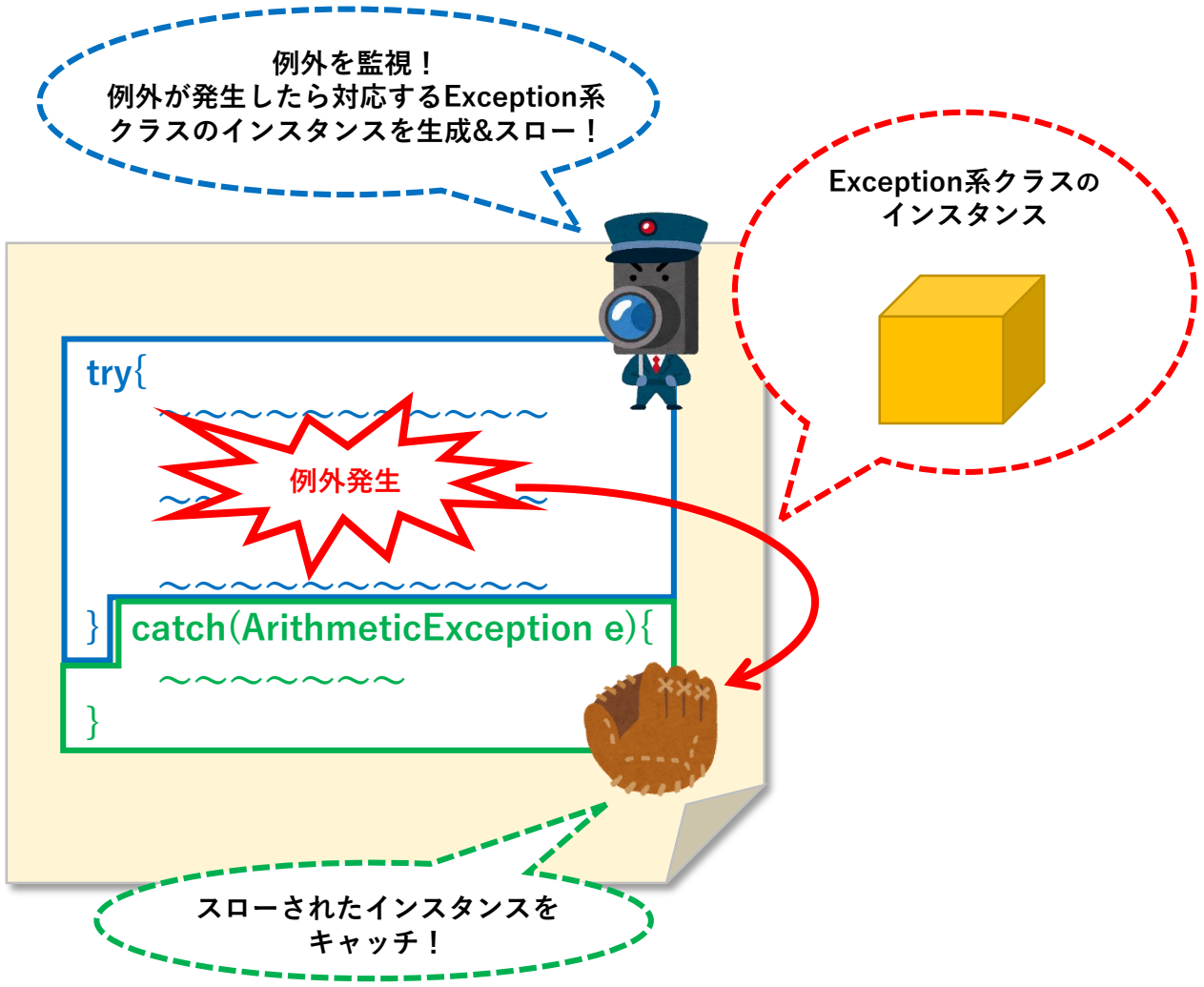
printlnメソッド
（引数：String型）

printlnメソッド
（引数：boolean型）

≪ポリモーフィズム≫

□**ポリモーフィズム**とは「**曖昧な命令に対する最適な処理をプログラム側が判断して行うこと**」です。
あまり実感はないかもですが、ポリモーフィズムの働きによってシンプルで直感的に理解のできるプログラムが書けるようになっています。

□引数の型や数によって最適なメソッドが選ばれる**オーバロード**や、スーパークラスとサブクラスに同名のメソッドが存在してもサブ側を優先して起動する**オーバーライド**もポリモーフィズムの一種です。



≪例外処理（try～catch文）①≫

□エラーには以下のような種類があります
- **文法エラー**：Javaの文法に反しているために起こるエラー。
  コンパイルが通らない。
- **論理エラー**：正常終了するも予想した結果と異なるというエラー。
- **例外**　　：コンパイルが通るも実行時に発生するエラー。
  文法の使い方に間違いがあったり予期せぬユーザーの操作やデータにプログラムとして対処できていない場合に発生する。処理の途中で強制終了する。

□**例外処理**とは例外を監視し、例外が起こった場合の対処を記述することです。主に**try～catch文**が使用されます。
tryブロックで例外が発生しそうな処理を監視し、catchブロックに例外対処用の処理を記述します。try～catchを使用することで本来の処理と例外対処用の処理を分離することが可能になり、プログラムの複雑化を防ぐことができます。

try～catch文で例外を検知した場合は強制終了とならずに**tryブロックを抜けた先から処理が再開**します。

□tryブロック内で例外が発生した場合、処理を中断し、**そのエラーに対応するException系クラスのインスタンスオブジェクトを生成&エラー情報を格納してcatchブロックに投げ渡します。**
これを「**例外のスロー**」「**例外のキャッチ**」と言います。
catchブロックは対処したい例外の分だけオーバーロードします。

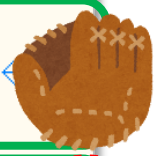
～try-catchのしくみ～

＜コマンドプロンプト＞

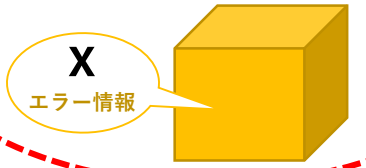
```
C:¥WorkSpace>java Sample2_07_2_2 1 x
```

```
try{
    int x = Integer.parseInt(args[0]) ;
    int y = Integer.parseInt(args[1]) ;

    System.out.println( x / y ) ;

}catch(ArithmeticException e){
    System.out.println("[ゼロ割]" + e + "が発生しました") ;

}catch(ArrayIndexOutOfBoundsException e){
    System.out.println("[インデックス範囲外指定]" + e + "が発生しました") ;

}catch(NumberFormatException e){
    System.out.println("[入力値不正（数字でない）]" + e + "が発生しました") ;

}
```

例外発生
（数値の型に関する例外）



NumberFormatExceptionクラス
のインスタンス（例外オブジェクト）



～try-catchのしくみ～

＜コマンドプロンプト＞

C:¥WorkSpace>java Sample_07_02

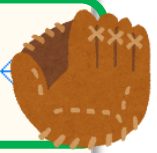
キャッチブロックはメソッドブロックのようなもの！
tryブロック内で発生する可能性のある例外オブジェクトの分だけ
オーバーロードして用意しておく！

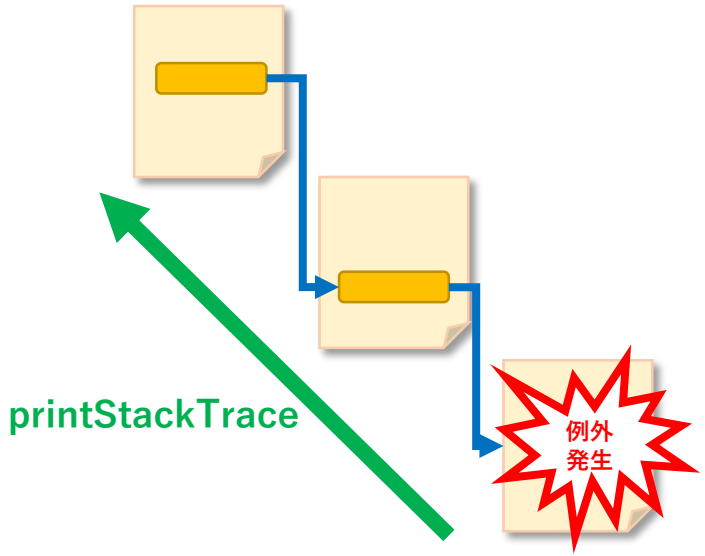
NumberFormatExceptionクラス
のインスタンス（例外オブジェクト）

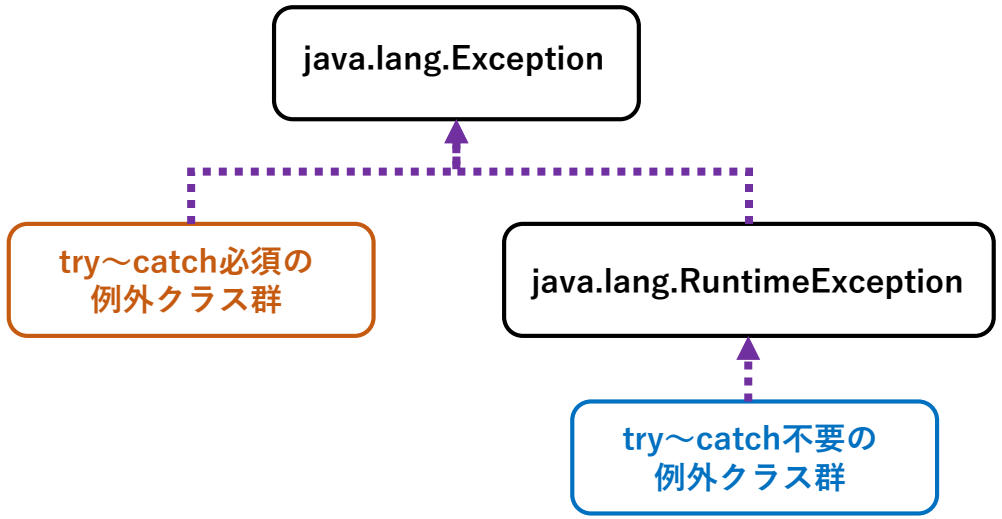
X
エラー情報

例外発生
（数値の型に関する例外）

```
try{
    int x = Integer.parseInt(arg[0]);
    int y = Integer.parseInt(arg[1]);

    System.out.println( x / y ) ;

}catch(ArithmeticException e){
    System.out.println("[ゼロ割]" + e + "が発生しました") ;

}catch(ArrayIndexOutOfBoundsException e){
    System.out.println("[インデックス範囲外指定]" + e + "が発生しました") ;

}catch(NumberFormatException e){
    System.out.println("[入力値不正（数字でない）]" + e + "が発生しました");

}
```







≪例外処理（try～catch文）②≫

□**printStackTrace**は例外クラスで用意されているメソッドで、実行すると例外が発生するまでの経路を画面に表示してくれます。
この情報はバグの原因を特定する際に重宝されます。

□try～catch文には**finallyブロック**を記述することが可能で、ブロック内の処理を**例外が発生するしないに関わらず必ず実行**します。
多くの場合、処理終了後に行うシステム処理（ネットワークやデータベースとの切断 など）やメモリの解放が記述されます
なお、finallyブロック内ではreturnの使用が禁止されています。

□メソッドの定義において発生する可能性のある例外があれば**throws節**を用いて明示することが可能です。throws節の付いたメソッドを呼び出す場合、呼び出す側でその例外処理を記述する必要があります。（例外処理をしなければコンパイルエラー）

```
public FileReader( String filename ) throws FileNotFoundException {  }
```
ファイルを開く処理→ファイルがない場合例外が発生する→throws節で定義

□すべての例外クラスはExceptionクラスを継承していますが、このExceptionクラスを直に継承している例外クラスのメソッドは全てtry～catchによる例外処理が必要になります。
逆にRuntimeExceptionクラスを継承している例外クラスはすべて例外処理不要です。