後期プログラミング講座
第1回

～オブジェクト指向とは何だろう？～

by バード

自己紹介

HN：バード

- 3回生　電情の者
- メインはプログラマーで、時々ドット絵も描く
- 最近は Black SurvivalやLeague of Legendといったゲームをやり始めた　　みんなもやってね

突然ですが皆さん…

「オブジェクト指向」

という言葉をご存知ですか？

この講義を受けている人は、

- いや、知らんし聞いたこともないわ
- 聞いたことはあるけどオブジェクトって何？
- 興味があって調べてみたけどわけわかめ

といった方がほとんどかと思います。

実際のところ、オブジェクト指向を完璧に理解している人はこの世に本当にいるんか？って思うくらいに、オブジェクト指向は難しくて理解しにくい概念だと思います。

なので、この講座では

- オブジェクト指向とは何なのか
- オブジェクト指向を効果的に使うには

についてざっくり理解してもらうことを目標とします。

オブジェクト指向とは？

ひとことで言うと、

あらゆる要素を『オブジェクト』としてくくり、オブジェクトの持つデータや振る舞いを定義して、各オブジェクトの関係を見つけることでプログラムを組み立てていく考え方です。

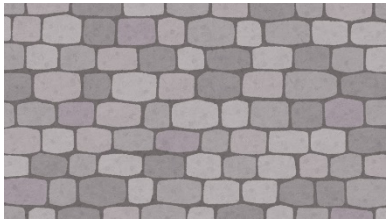
忍者アクションゲームで例えると

1.オブジェクトとしてくくる

「プレイヤー」
オブジェクト

「おにぎり」
オブジェクト

「手裏剣」
オブジェクト



「地面」
オブジェクト

「敵」
オブジェクト

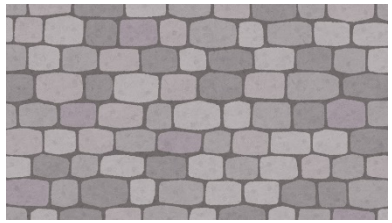
オブジェクト＝“モノ”

2.データや振る舞いを定義する

「プレイヤー」
オブジェクト
データ：位置、体力
振る舞い：動く、投げる、
食べる、ダメージ

「おにぎり」
オブジェクト
データ：位置
振る舞い：スコアUP

「手裏剣」
オブジェクト
データ：位置、速度
振る舞い：動く、攻撃



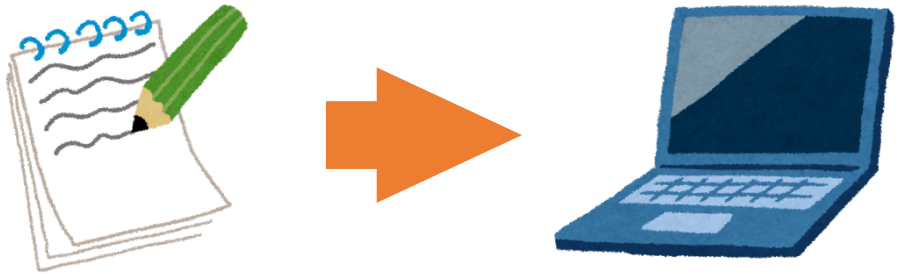
「地面」
オブジェクト
データ：位置
振る舞い：無し

「敵」
オブジェクト
データ：位置、体力
振る舞い：動く、攻撃

3.各オブジェクトの関係を見つける

このようにオブジェクトを定義したりオブジェクト同士を関連付ける作業のことを、オブジェクト指向プログラミングでは「設計」と呼びます。

設計が完了したらそれをプログラムに落とし込みます。
このような流れでプログラムを完成させていきます。



そして、オブジェクト指向においては

この設計の出来栄えがとても重要になるのです。

では、より良い設計をするためにはどうすればいいのかという話になってくるのですが…

その話をする前に、「クラス」という概念を理解する必要があるので、先にそちらを説明します。

クラスとは

クラスとはオブジェクトを生成するための仕様書のようなものであり、すべてのオブジェクトは必ず何らかのクラスをもとにして作られます。

一つのクラスから複数のオブジェクトを作ることも可能です。

オブジェクトがどのデータを持ち、どの振る舞いをするのかは、すべてクラスの中に記述していきます。

クラスの中身

プレイヤーのデータや振る舞いをクラスの中で定義します。

「プレイヤー」
オブジェクト
データ：位置、体力
振る舞い：動く、投げる、
食べる、ダメージ

```
public class Ninja {
  Vector2 pos;      //位置
  float hp;         //体力

  void Move(){      //動く
    pos.x += 1;
  }

  void Damage(){    //ダメージ
    hp -= 30;
  }
  ...
}
```

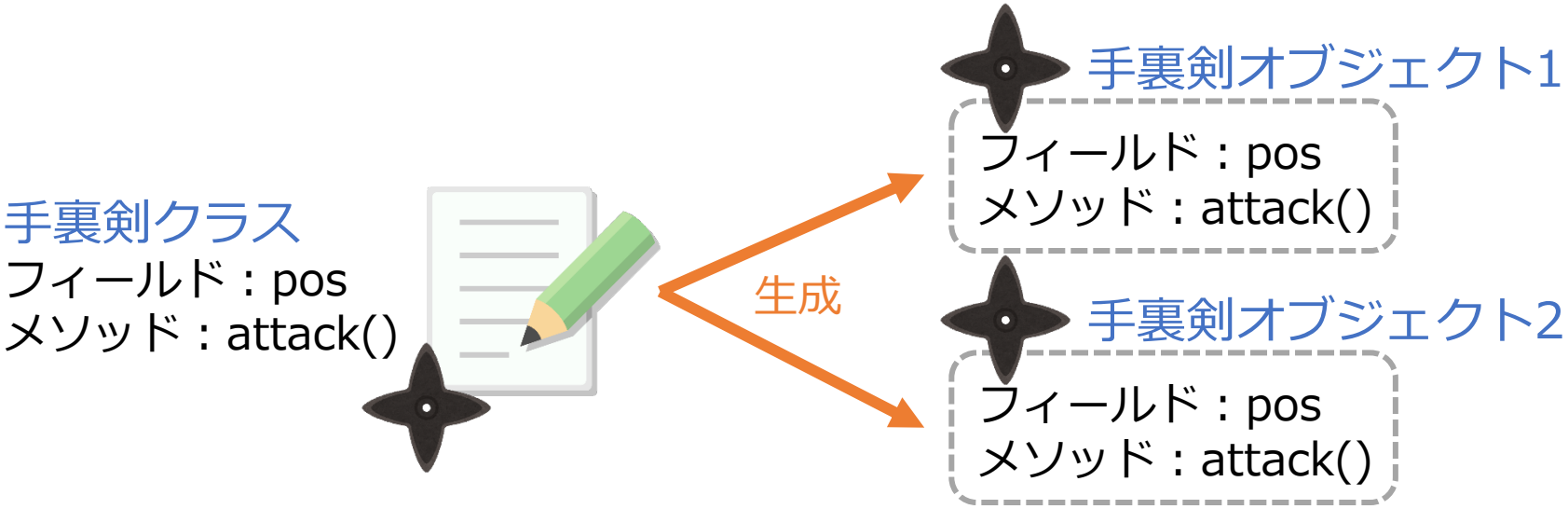
データ

振る舞い

フィールドとメソッド

クラス内のデータのことをフィールドと呼び、
振る舞いのことをメソッドと呼びます。

クラスから生成された各オブジェクトは、内部に
フィールドとメソッドを所持することになります。



オブジェクト同士の関係

オブジェクトは変数の中に格納することができ、変数を通してそのオブジェクト内のデータやメソッドを参照することができます。

その処理はもちろんクラス内で記述します。

（例）「敵が攻撃すると忍者にダメージ」という関係を表したい時

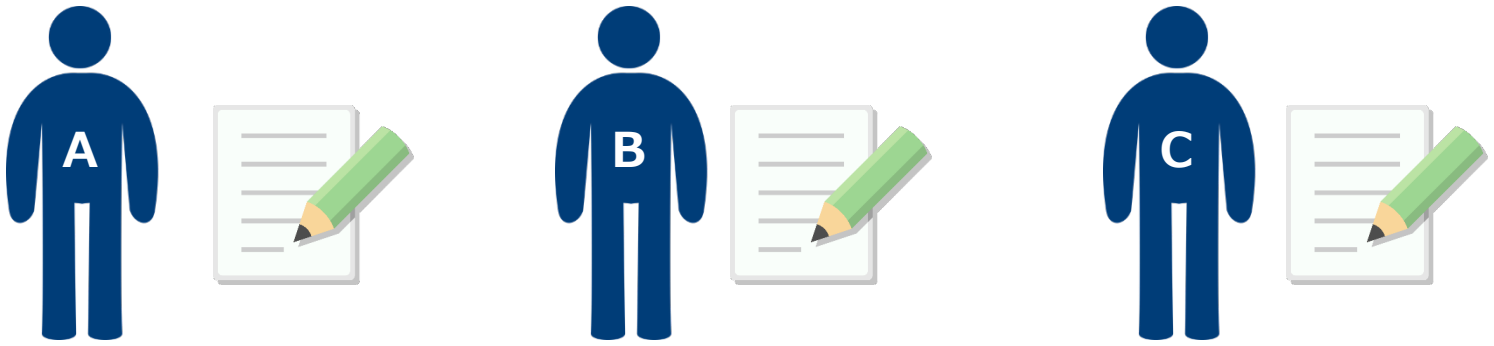
```
public class Enemy {
    Ninja ninja;
    void Attack(){
      ninja.Damage();
    }
}
```

```
public class Ninja {
    float hp;
    void Damage(){
      hp -= 30;
    }
}
```

クラスに分けるメリット

処理をクラスに分ける書き方は、特に複数人での作業するときに分担してプログラムできるため便利です。

また、機能の修正や拡張もしやすく、デバッグも容易に行えるといったメリットもあります。



ここまでのまとめ

- オブジェクト指向は、データや振る舞いを持つオブジェクト同士のメッセージのやり取りを記述していく、一種のプログラムの書き方である。
- オブジェクトのデータや振る舞い、関係性はすべてクラスという仕様書に記述する。
- オブジェクト指向は複数人での開発をしやすい

オブジェクト指向三大要素

オブジェクト指向には3つの考え方があります。

- カプセル化
- 継承
- ポリモーフィズム

どれもオブジェクト指向を効果的に使うのに、
大切な概念となっています。

カプセル化

例えば、よく「テレビのリモコン」を日常生活の中で使いますが、我々はリモコンの中の構造がどうなっているかを気にしませんよね？

「電源ボタンを押せばテレビがつく」といった、必要最低限の知識さえあればリモコンは使えます。

このように、最低限の機能を公開して、複雑な処理を内部に隠すことを「カプセル化」といいます。

ボタンを押すと、
テレビがつくよ！

電子回路を作動して…
赤外線を飛ばして…

カプセル化の例

例えば、敵クラスとプレイヤークラスが以下の関係にあるとします。

```
public class Enemy {
    Ninja ninja;
    void Attack(){
      ninja.hp -= 30;
    }
      …
}
```

```
public class Ninja {
    float hp;
      …
}
```

「敵が攻撃すると忍者にダメージ」という関係

カプセル化の例

一見、関係をちゃんと記述できているように見えますが、このコードには一つ問題があります。

例えば、後の仕様変更でプレイヤーHPがゲージ制ではなくライフ制に変更になった場合、プレイヤークラスだけでなく、敵クラスも変更する必要があります。ダメージを与えるクラスが複数いれば、修正の手間も倍以上になります。すごく手間ですね。

カプセル化の例

これを解決するには、まずダメージを受けるという一連の処理を一つの関数にまとめて処理の詳細を内部に隠してしまいます。

```
public class Enemy {
    Ninja ninja;
    void Attack(){
      ninja.Damage();
    }
      …
}
```

```
public class Ninja {
    float hp;
    void Damage(){
        hp -= 30;
        …
    }
}
```

公開と隠蔽

そして、外部から隠したい要素には先頭に private、公開したい要素にはpublic を付け加えます。

```
public class Enemy {
    Ninja ninja;
    void Attack(){
      ninja.Damage();
    }
      …
}
```

```
public class Ninja {
    private float hp;
    public void Damage(){
        hp -= 30;
        …
    }
}
```

カプセル化の利点

こうすることで、プレイヤーHPの仕様がどう変わろうと他のクラスには影響なく、ただ「ダメージを与える」という処理を実行すれば望み通りの結果を得ることができます。

このように、外部から見たクラスの機能を単純にすることがカプセル化で大切なことです。

拙者を殴ることができるでござる！

殴られたらHPを減らして…
アニメーションして…
HPが0になったら…

継承

あるクラスのデータや振る舞いを引き継いだ新しいクラスを作ることを継承と呼びます。

引き継ぎ元のクラスを親クラス、引き継ぎ先のクラスを子クラスといいます。

継承により、親クラスが持っている情報は子クラスも持っているという関係を作れます。

親クラスと子クラス

クラス名の後ろに継承したいクラス名を書くことで、継承元のクラスのフィールドやメソッドを引き継ぐことができます。

親クラス

```
public class A {
    int a;
    void aa(){
      …
    }
}
```

子クラス

```
public class B : A{
    float b;
}
```

継承によって、直接記述せずともBクラスは変数aとメソッドaaも所持することができる。

継承の例

例えば敵の種類を増やしたいとします。一番単純な実現方法としては、増やしたい分だけバラバラにクラスを作ることでしょう。

Enemyクラス

GirlEnemyクラス

DarkEnemyクラス

SmallEnemyクラス

継承の例

しかしよく考えてみると、すべての敵に共通しているものが多く、同じ要素を各クラスに記述するのは大変無駄が多い作業です。

Enemyクラス
データ:位置、体力
振る舞い：動く、攻撃

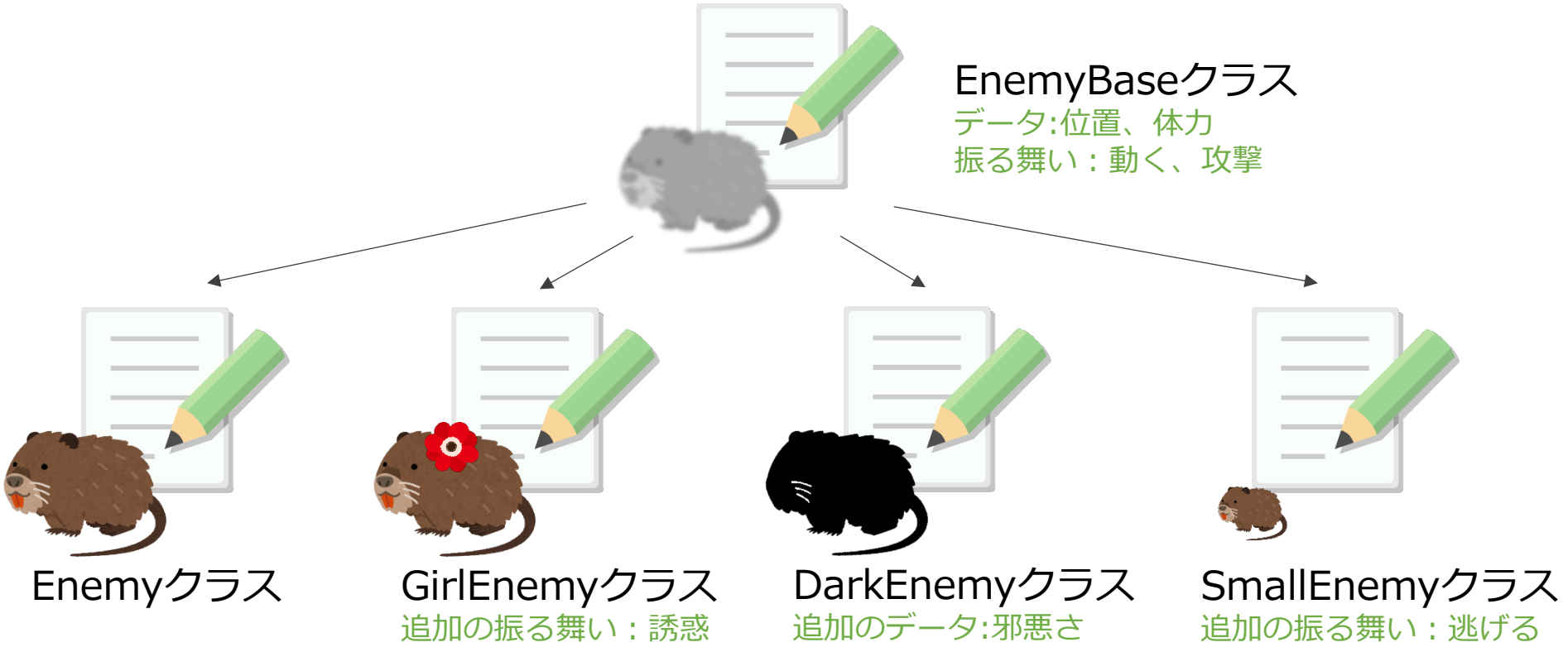
GirlEnemyクラス
データ:位置、体力
振る舞い：動く、攻撃、誘惑

DarkEnemyクラス
データ:位置、体力、邪悪さ
振る舞い：動く、攻撃

SmallEnemyクラス
データ:位置、体力
振る舞い：動く、攻撃、逃げる

継承の例

そこで、共通しているものを抽出して親クラスとして分離することで記述する手間を省けます。



抽象と具体

少し視点を変えてみると、親クラスは抽象的、子クラスは具体的な概念とも考えられます。

クラスを抽象と具体に分けることも継承の目的の一つだと言えます。

オーバーライド

親クラスを継承した子クラスは、親クラスのメソッドの内容を別の内容に書き換えることができます。これをメソッドのオーバーライドと呼びます。

```
public class EnemyBase {

    void Move(){
      pos.x += 1;
    }
      …
}
```

書き換え

```
public class GirlEnemy :
EnemyBase {
    void Move(){
      pos.y += 1;
    }
      …
}
```

ポリモーフィズム

3つのうちで一番理解しづらいのがこのポリモーフィズムだと思います。

日本語では「多態性」といいますが、オブジェクト指向での意味はざっくり言うとこんな感じです。

「オブジェクトの種類によって振る舞いを変化させる仕組み」

ポリモーフィズムの例

例えば、「1体の敵にダメージを与える手裏剣クラス」を記述するとき、おおまかに以下のように書くとします。

```
public class Shuriken {
    Enemy enemy;
    void Attack(){
      enemy = GetEnemy();
      enemy.Damage();
    }
}
```

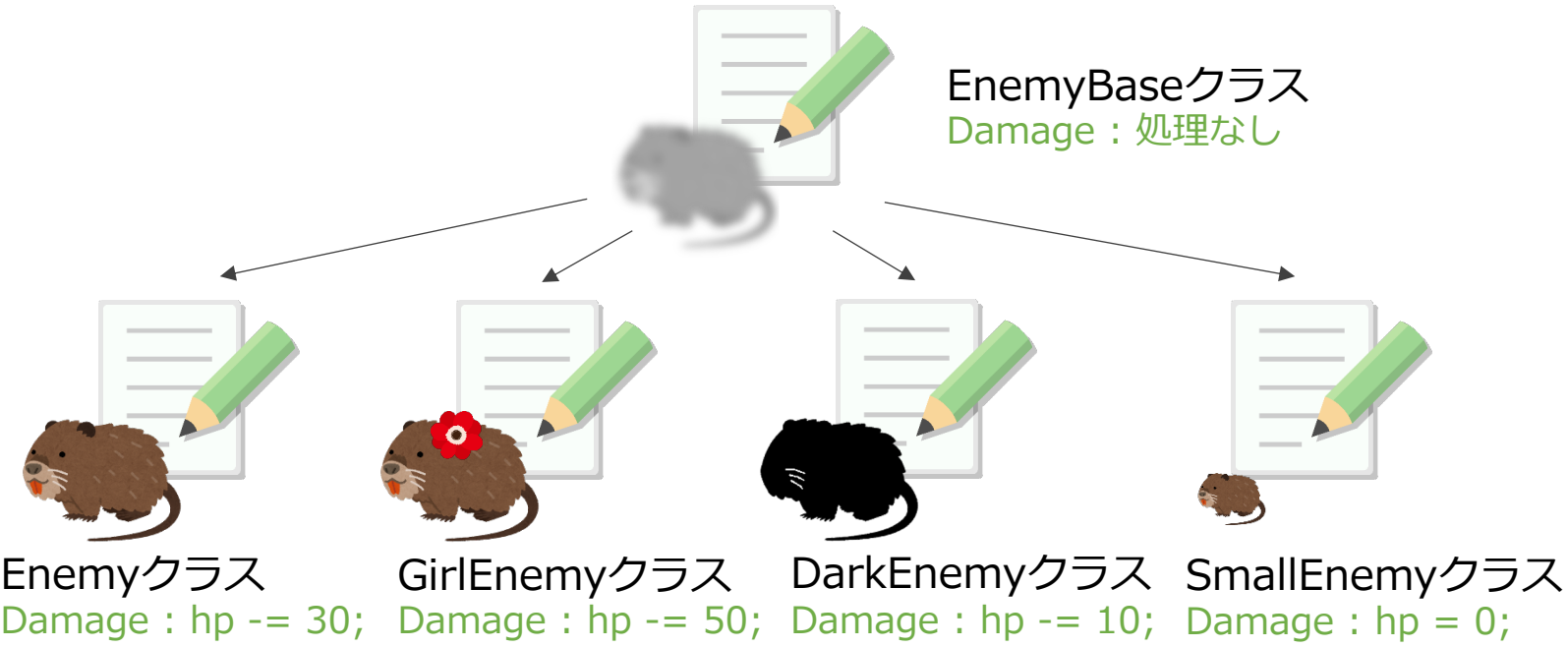
```
public class Enemy {
    private float hp;
    public void Damage(){
        hp -= 30;
        …
    }
}
```

GetEnemyはEnemyクラスのオブジェクトを取得するメソッドとする。

ポリモーフィズムの例

しかし大幅な仕様変更で「3体の異なる種類の敵にダメージを与える」ようになったとします。

このとき、Damageメソッドは各々の子クラスで以下のようにオーバーライドされています。



ポリモーフィズムの例

この場合、敵の種類分だけ変数を用意して一つ一つに処理を書いていくのは面倒です。

```
public class Shuriken {
    GirlEnemy girlenemy;
    DarkEnemy darkenemy;
    SmallEnemy smallenemy;
    void Attack(){
      girlenemy = GetGirlEnemy();
      girlenemy.Damage();
      darkenemy = GetDarkEnemy();
      …
    }
}
```

違う型の変数に代入

ここで、一つ工夫して「子クラスのオブジェクトを親クラス型の変数に入れる」ことをしてみます。

```
public class Shuriken {
    EnemyBase[] enemies;
    void Attack(){
      enemies[0] = GetGirlEnemy();
      enemies[1] = GetDarkEnemy();
      enemies[2] = GetSmallEnemy();
    }
}.
```

これは、子クラスのオブジェクトを疑似的に親クラスのオブジェクトとして扱うことを意味しています。

同じメソッドでも違う振る舞い

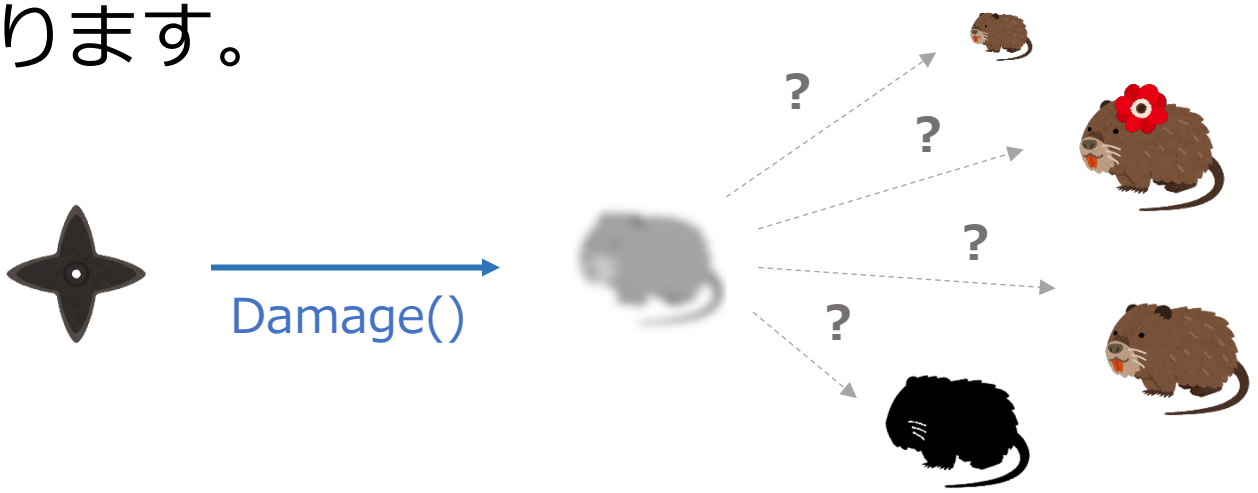
そして、enemiesを通して親クラスのメソッドとしてDamageメソッドを呼び出します。

3つの変数内のオブジェクトはどれもクラスが違うので、それぞれ異なった振る舞いをします。

```
void Attack(){
  enemies[0] = GetGirlEnemy();
  enemies[1] = GetDarkEnemy();
  enemies[2] = GetSmallEnemy();
  for(int i = 0; i < 3; i++){
      enemies[i].Damage();
  }
}
```

ポリモーフィズムとは多様性

このように、異なる子クラスのオブジェクトを、共通する親クラスのオブジェクトとして扱うことで、統一した記述をしながらも実際には各オブジェクトによって違った振る舞いをさせることが可能になります。



まとめ

- カプセル化…複雑な処理は隠して必要なものだけを公開する
- 継承…クラスのデータや振る舞いを引き継ぐ
- ポリモーフィズム…統一した記述で異なる振る舞いをさせる

資料に書いたことをすべて理解していなくても、ざっくり上記のまとめだけでも分かってくれれば十分です。正直一発で理解出来る人は天才だと思います…

最後に

今回話した内容は決して簡単なものではなく、一度聞いただけではいまいち理解できないかもしれません。しかし自分でクラスを書いて何度も設計をしていけば、いつかはここで書いた内容が理解できるようになります。プログラミングで大事なのは学びと実践の繰り返しだと僕は思っています。ぜひ皆さんもいろんなコードを書いて良いプログラマーライフを送ってください。応援しています。