

INTERNET ACADEMY

Institute of Web Design & Software Services

C#5

インターネット・アカデミー

C# 目次

1. (復習) 単純なRPGの作成(キャラ作成)
2. 継承
3. オーバーライド、オーバーロード
4. 抽象クラス
5. インターフェース
6. 抽象クラスとインターフェースの違い、使いどころ
7. (復習)Formと継承

(復習)単純なRPGの作成(キャラ作成)

▼ GameCharクラス

```
public class GameChar
{
    private int hp; // ヒットポイント
    public GameChar(int hp)
    {
        this.hp = hp;
    }
    public void Attack(int damage)
    {
        this.hp -= damage;
        if(this.hp <= 0)
        {
            hp = 0;
        }
    }
    public void ShowHP()
    {
        Console.WriteLine("HP:{0}", this.hp);
    }
    public int HP
    {
        get { return hp; }
    }
}
```

GameCharクラスに足りないもの

- ・ RPGのキャラクターの基本的な特徴を満たしてはいるが ...
- ・ 主人公（英雄）キャラクターのメンバとしては少々不足
- ・ ましてや、多種多様な敵キャラ（モンスター）に至っては...

GameCharクラスに足りないもの

- ・ RPGのキャラクターの基本的な特徴を満たしてはいるが …
- ・ 主人公（英雄）キャラクターのメンバとしては少々不足
- ・ ましてや、多種多様な敵キャラ（モンスター）に至っては…

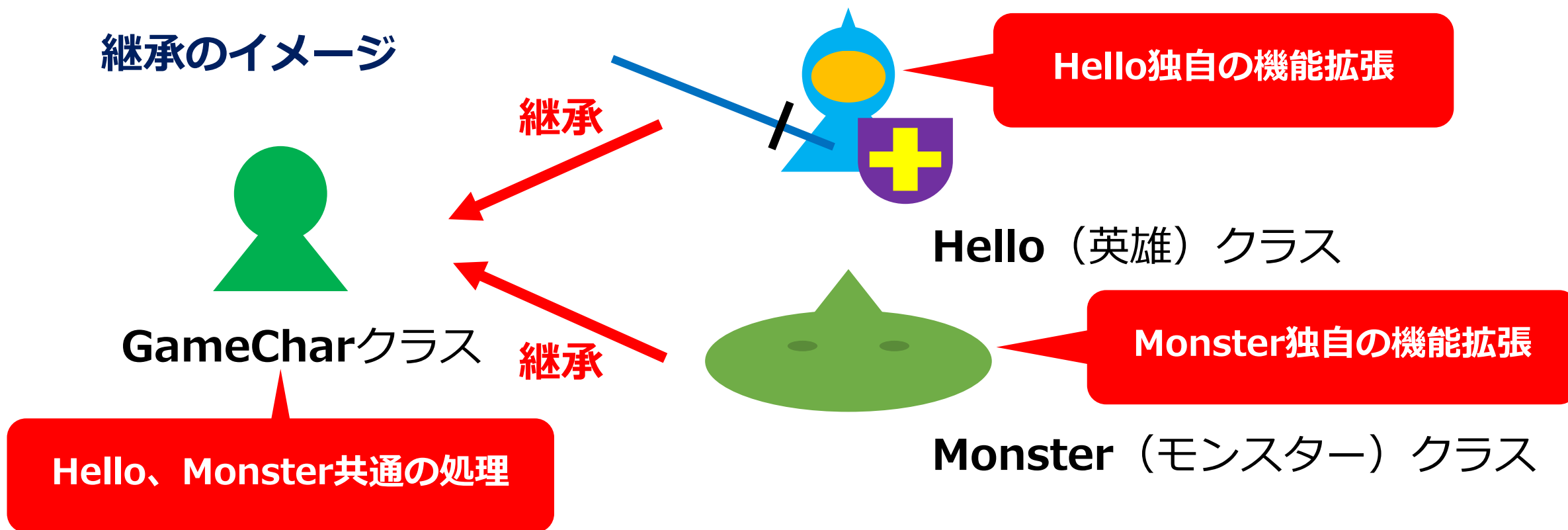
！ポイント

単純なクラスだけでRPGを作ろうとすると、似たようなクラスを大量に作る必要が出てくる

継承を用いて問題を解決

- ・ 解決策：基本機能を持つクラスを引き継いだ様々なクラスを作ればよい
- ・ このような方法を**継承（けいしょう）**と言う

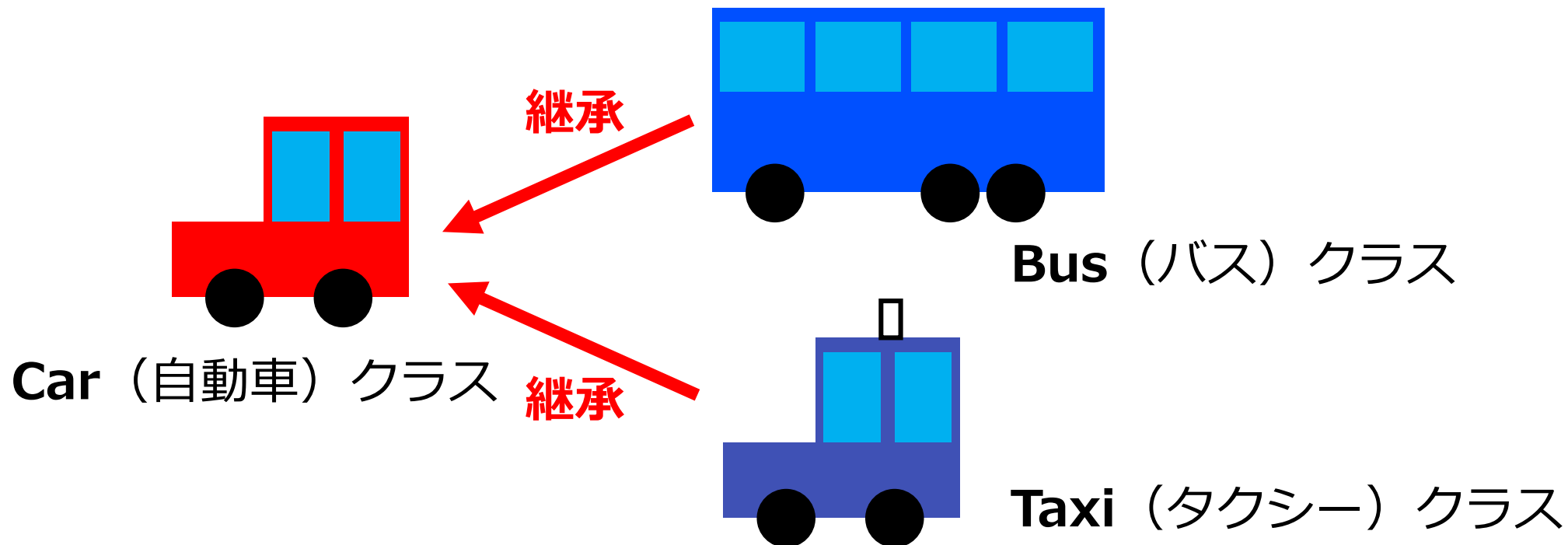
継承のイメージ



継承とは

継承（けいしょう）… あるクラスを拡張した新しいクラスを作る機能

継承のイメージ

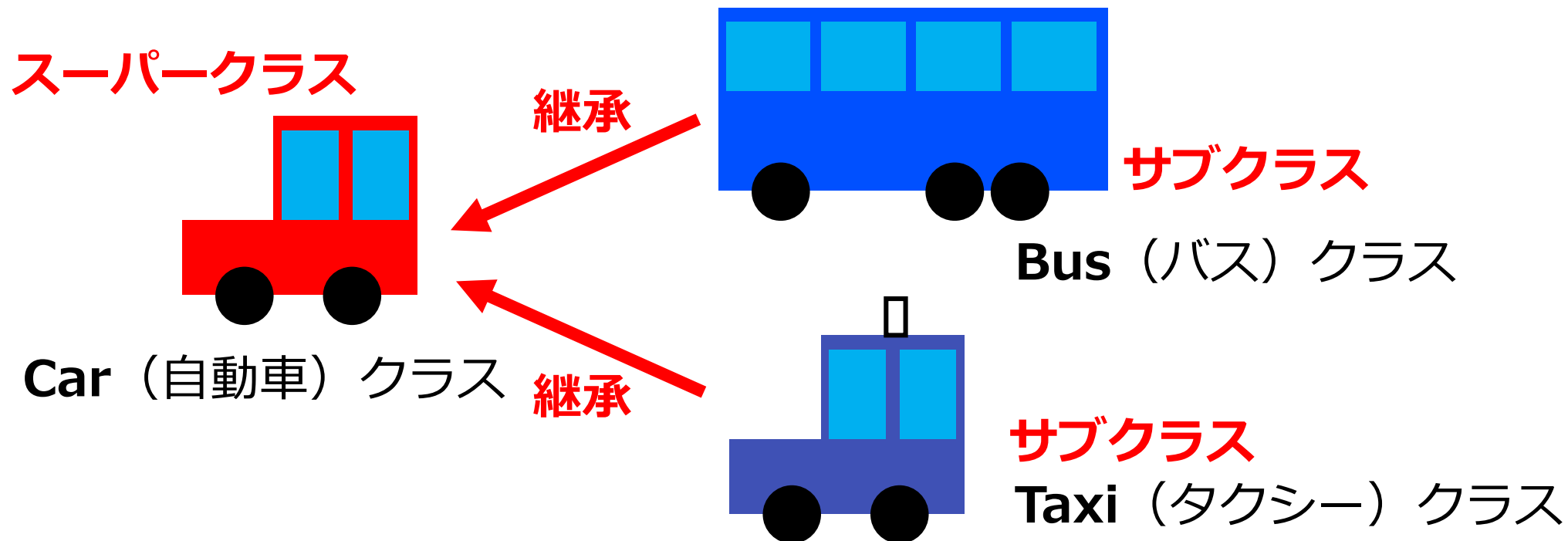


スーパークラスとサブクラス

スーパークラス (Super Class) … 継承の対象となるクラス

サブクラス (Sub Class) … スーパークラスを継承したクラス

継承のイメージ



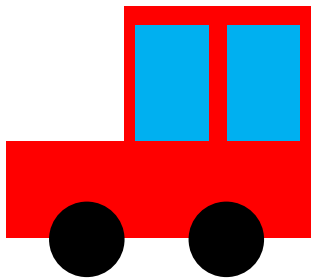
スーパークラスとサブクラス

スーパークラス (Super Class) … 継承の対象となるクラス

サブクラス (Sub Class) … スーパークラスを継承したクラス

継承のイメージ

スーパークラス

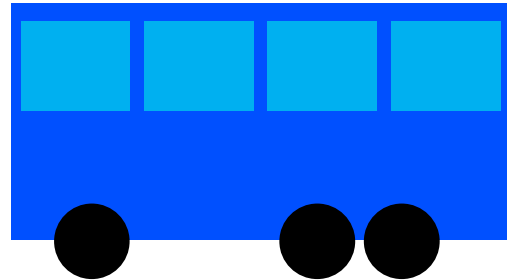


Car (自動車) クラス

親クラス・ベースクラス
とも言う

継承

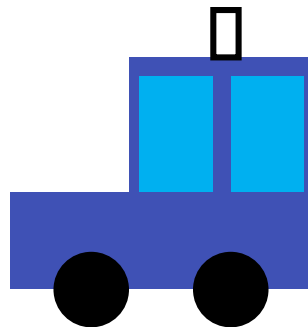
継承



Bus (バス) クラス

サブクラス

子クラス・派生クラス
とも言う



サブクラス

Taxi (タクシー) クラス

継承を用いるメリット

- ・ソフトウェアのアップデートが容易になる
- ・ソースコードの再利用が簡単になる
- ・クラスの機能拡張ができる

継承の記述

▼Carクラス（スーパークラス）

```
public class Car
{
    public int speed = 0;
    public void Run()
    {
        Console.WriteLine("時速{0}kmで走行する",speed);
    }
}
```

継承の記述

▼Carクラスの利用例

```
Car car = new Car();  
car.speed = 50;  
car.Run();
```

▼実行結果

時速50kmで走行する

継承の記述

▼Taxiクラス (サブクラス)

```
public class Taxi : Car
{
    public int price = 500;
    public void Show()
    {
        Console.WriteLine("初乗り{0}円",price);
    }
}
```

継承の記述

▼Taxiクラス (サブクラス)

```
public class Taxi : Car
{
    public int price = 500;
    public void Show()
    {
        Console.WriteLine("初乗り{0}円",price);
    }
}
```

スーパークラス

継承の記述

▼Taxiクラスの利用例

```
Taxi taxi = new Taxi();  
taxi.price = 600;  
taxi.Show();  
taxi.speed = 50;  
taxi.Run();
```

▼実行結果

時速50kmで走行する

継承の記述

▼Taxiクラスの利用例

```
Taxi taxi = new Taxi();  
taxi.price = 600;  
taxi.Show();  
taxi.speed = 50;  
taxi.Run();
```

サブクラスのメンバ

▼実行結果

時速50kmで走行する

継承の記述

▼Taxiクラスの利用例

```
Taxi taxi = new Taxi();  
taxi.price = 600;  
taxi.Show();  
taxi.speed = 50;  
taxi.Run();
```

スーパークラスのメンバ

▼実行結果

時速50kmで走行する

！ポイント

サブクラスはスーパークラスのメンバを利用できるとともに独自のメンバも利用できる

単一継承

- ・ C#で継承する際スーパークラスは1つしか持てない（単一継承）
- ・ 複数のスーパークラスを持てる継承を多重継承という
- ・ C#は多重継承を許さない

コンストラクタの呼び出し順

コンストラクタはスーパークラス→サブクラスの順で実行される

▼スーパークラス

```
public class SuperCls
{
    public SuperCls()
    {
        Console.WriteLine("super");
    }
}
```

▼サブクラス

```
public class SubCls : SuperCls
{
    public SubCls()
    {
        Console.WriteLine("sub");
    }
}
```

▼サブクラスのインスタンス生成

```
SubCls subCls = new SubCls();
```

▼実行結果

```
super
sub
```

コンストラクタの呼び出し順

コンストラクタはスーパークラス→サブクラスの順で実行される

▼スーパークラス

```
public class SuperCls
{
    public SuperCls()
    {
        Console.WriteLine("super");
    }
}
```

▼サブクラス

```
public class SubCls : SuperCls
{
    public SubCls()
    {
        Console.WriteLine("sub");
    }
}
```

▼サブのインスタンス生成

```
SubCls subCls = new SubCls();
```

▼実行結果

```
super
sub
```

protected修飾子

- **public** ... 無制限のアクセスが可能（クラスの先頭につけることも可能）
- **protected** ... クラス内、もしくは派生クラス内でアクセス可能
- **private** ... クラス内からのみアクセス可能
- メンバのアクセス修飾子を省略した場合には、privateと同じ扱いになる

protected修飾子

▼スーパークラス

```
public class SuperCls
{
    protected int p1 = 10;
    private int p2 = 20;
    public SuperCls()
    {
        Console.WriteLine("super");
    }
}
```

▼サブクラス

```
public class SubCls : SuperCls
{
    public SubCls()
    {
        Console.WriteLine("sub");
    }
    public void Show()
    {
        // protectedへはアクセスできる
        Console.WriteLine("p1={0}", p1);
        // privateにはアクセスできない
        //Console.WriteLine("p2={0}", p2);
    }
}
```


protected修飾子

スーパークラスのprivateなメンバはサブクラスでは利用できない

▼スーパークラス

```
public class SuperCls
{
    protected int p1 = 10;
    private int p2 = 20;
    public SuperCls()
    {
        Console.WriteLine("super");
    }
}
```

コメントをとるとエラーになる

▼サブクラス

```
public class SubCls : SuperCls
{
    public SubCls()
    {
        Console.WriteLine("sub");
    }
    public void Show()
    {
        // protectedへはアクセスできる
        Console.WriteLine("p1={0}", p1);
        // privateにはアクセスできない
        //Console.WriteLine("p2={0}", p2);
    }
}
```

protected修飾子

スーパークラスのprotectedなメンバはサブクラスでも利用できる

▼スーパークラス

```
public class SuperCls
{
    protected int p1 = 10;
    private int p2 = 20;
    public SuperCls()
    {
        Console.WriteLine("super");
    }
}
```

スーパークラスのメンバを利用できる

▼サブクラス

```
public class SubCls : SuperCls
{
    public SubCls()
    {
        Console.WriteLine("sub");
    }
    public void Show()
    {
        // protectedへはアクセスできる
        Console.WriteLine("p1={0}", p1);
        // privateにはアクセスできない
        // Console.WriteLine("p2={0}", p2);
    }
}
```

protected修飾子

▼SubClsクラスのサンプル

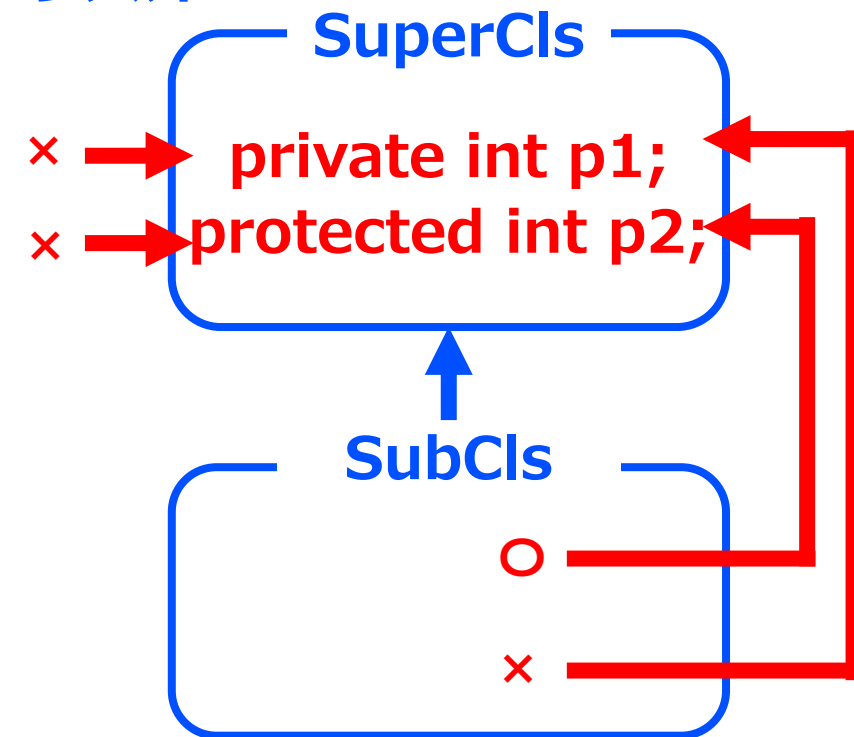
```
SubCls subCls = new SubCls();  
// private,protectedには外部からアクセスできない  
// subCls.p1 = 100;  
// subCls.p2 = 200;  
subCls.Show();
```

▼実行結果

```
super  
sub  
p1=10
```

○ : アクセス可能
× : アクセス不可能

クラス外



protectedはサブクラス以外ではprivateと同じ扱いになる

protected修飾子

▼SubClsクラスのサンプル

```
SubCls subCls = new SubCls();  
// private,protectedには外部からアクセスできない  
// subCls.p1 = 100;  
// subCls.p2 = 200;  
subCls.Show();
```

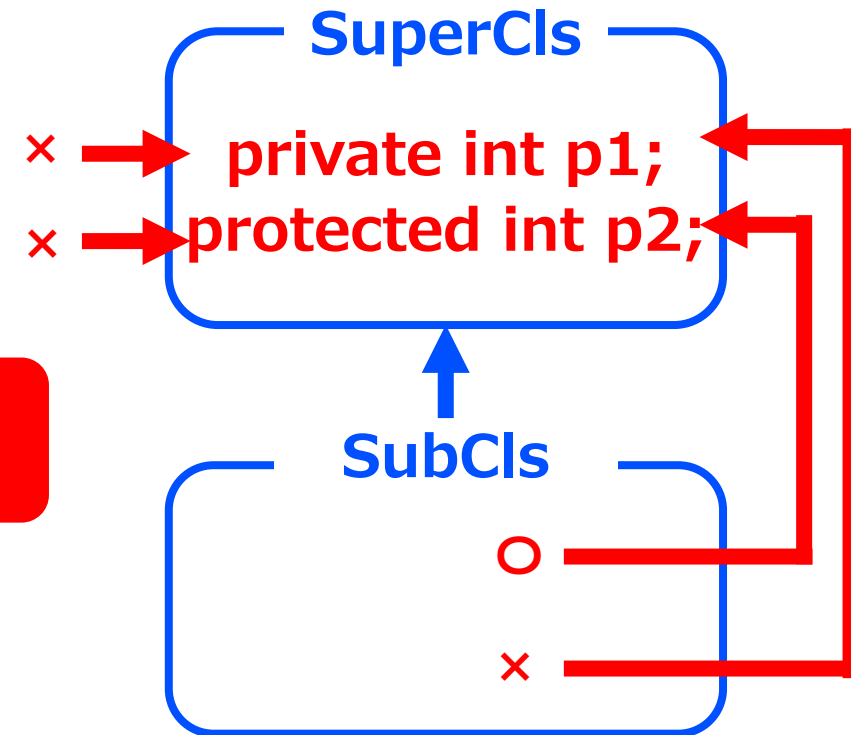
コメントを外すとエラーになる

▼実行結果

```
super  
sub  
p1=10
```

○ : アクセス可能
× : アクセス不可能

クラス外



protectedはサブクラス以外ではprivateと同じ扱いになる

オーバーライド

- ・ サブクラスでスーパークラスと同じメソッドを作ること
- ・ サブクラスでスーパークラスのメソッドが上書きされる
- ・ 引数名、メソッド名、戻り値の型は完全に一致させる必要がある

オーバーライド

▼スーパークラス

```
public class Person
{
    public string name;
    public virtual void Show()
    {
        Console.WriteLine("名前:{0}", name);
    }
}
```

▼実行のサンプル

```
Person person = new Person();
person.name = "山田太郎";
person.Show();
```

▼実行結果

名前:山田太郎

オーバーライド

▼スーパークラス

```
public class Person
{
    public string name;
    public virtual void Show()
    {
        Console.WriteLine("名前:{0}", name);
    }
}
```

virtual修飾子をつけたメソッドはオーバーロード可能

▼実行のサンプル

```
Person person = new Person();
person.name = "山田太郎";
person.Show();
```

▼実行結果

名前:山田太郎

オーバーライド

▼サブクラス

```
public class Student : Person
{
    public int grade = 1;
    // Showメソッドをオーバーライド
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

▼実行のサンプル

```
Student student = new Student();
student.name = "山田太郎";
student.grade = 3;
student.Show();
```

▼実行結果

学年:3年 名前:山田太郎

オーバーライド

▼サブクラス

```
public class Student : Person
{
    public int grade = 1;
    // Showメソッドをオーバーライド
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

▼実行のサンプル

```
Student student = new Student();
student.name = "山田太郎";
student.grade = 3;
student.Show();
```

▼実行結果

学年:3年 名前:山田太郎

override修飾子をつけると同名・同じ引数と戻り値の親クラスのメソッドをオーバーライドする

オーバーライド

▼サブクラス

```
public class Student : Person
{
    public int grade = 1;
    // Showメソッドをオーバーライド
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

「base.」をつけるとスーパークラスのメンバにアクセス可能

▼実行のサンプル

```
Student student = new Student();
student.name = "山田太郎";
student.grade = 3;
student.Show();
```

▼実行結果

学年:3年 名前:山田太郎

Personクラスの
Showメソッドの
実行結果

オーバーライド（コンストラクター）

▼スーパークラス

```
public class Person
{
    private string name;
    public Person(string name)
    {
        this.name = name;
    }
    public virtual void Show()
    {
        Console.WriteLine("名前:{0}", name);
    }
}
```

▼実行のサンプル

```
Person person
    = new Person("山田太郎");
person.Show();
```

▼実行結果

名前:山田太郎

オーバーライド（コンストラクター）

- コンストラクターは通常のメソッドのようにオーバーライドできない
- サブクラスの引数付きコンストラクターを実行してもスーパークラスのコンストラクタが実行されるわけではない
- サブクラスの引数付きコンストラクターを呼ぶには明示的に呼び出すしかない

オーバーライド（コンストラクター）

▼スーパークラス

```
public class Person
{
    private string name;
    public Person(string name)
    {
        this.name = name;
    }
    public virtual void Show()
    {
        Console.WriteLine("名前:{0}", name);
    }
}
```

引数付きの
コンストラクター

▼実行のサンプル

```
Person person
    = new Person("山田太郎");
person.Show();
```

▼実行結果

名前:山田太郎

オーバーライド（コンストラクター）

▼サブクラス

```
public class Student : Person
{
    private int grade = 1;
    public Student(int grade,string name) : base(name)
    {
        this.grade = grade;
    }
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

オーバーライド（コンストラクター）

▼サブクラス

```
public class Student : Person
{
    private int grade = 1;
    public Student(int grade, string name) : base(name)
    {
        this.grade = grade;
    }
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

①newによって呼び出されるコンストラクタ

オーバーライド（コンストラクター）

▼サブクラス

```
public class Student : Person
{
    private int grade = 1;
    public Student(int grade,string name) : base(name)
    {
        this.grade = grade;
    }
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

②スーパークラス（Person）のコンストラクタ呼び出し

オーバーライド（コンストラクター）

▼サブクラス

```
public class Student : Person
{
    private int grade = 1;
    public Student(int grade,string name) : base(name)
    {
        this.grade = grade;
    }
    public override void Show()
    {
        Console.WriteLine("学年:{0}年 ", grade);
        base.Show();
    }
}
```

③コンストラクタ内の処理を実行

オーバーロード (GameCharクラス)

▼GameCharクラス

```
public class GameChar
{
    protected int hp; // ヒットポイント()
    public GameChar(int hp)
    {
        this.hp = hp;
    }
    public virtual void Attack(int damage)
    {
        this.hp -= damage;
        if (this.hp <= 0)
        {
            hp = 0;
        }
    }
    public void ShowHP()
    {
        Console.WriteLine("HP:{0}", this.hp);
    }
    public int HP
    {
        get { return hp; }
    }
}
```

▼Helloクラス

```
public class Hello : GameChar
{
    public Hello() : base(10)
    {
    }
    public override void Attack(int damage)
    {
        base.Attack(damage);
        if (hp == 0)
        {
            Console.WriteLine("== 勇者よ、死んでしまうとは情けない ==");
        }
    }
}
```

ゲームのキャラクターの親クラスをGameCharクラスとし、それを継承して各種キャラクタを作るようにしてみる

オーバーロード (GameCharクラス)

▼GameCharクラス

```
public class GameChar
{
    protected int hp; // ヒットポイント()
    public GameChar(int hp)
    {
        this.hp = hp;
    }
    public virtual void Attack(int damage)
    {
        this.hp -= damage;
        if (this.hp <= 0)
        {
            hp = 0;
        }
    }
    public void ShowHP()
    {
        Console.WriteLine("HP:{0}", this.hp);
    }
    public int HP
    {
        get { return hp; }
    }
}
```

▼Helloクラス

```
public class Hello : GameChar
{
    public Hello() : base(10)
    {
    }
    public override void Attack(int damage)
    {
        base.Attack(damage);
        if (hp == 0)
        {
            Console.WriteLine("== 勇者よ、死んでしまうとは情けない ==");
        }
    }
}
```

オーバーライドされた
Attackメソッド

ゲームのキャラクターの親クラスをGameCharクラスとし、それを継承して各種キャラクタを作るようにしてみる

オーバーロード (GameCharクラス)

▼実行のサンプル

```
Console.WriteLine("RPGゲーム");  
Hello c = new Hello();  
c.ShowHP();  
Random r = new Random();  
while (c.HP != 0)  
{  
    int attack = r.Next(1, 10);  
    c.Attack(attack); // 敵の攻撃  
    Console.WriteLine("勇者が{0}の攻撃を受ける", attack);  
    c.ShowHP();  
}  
Console.WriteLine("**ゲームオーバー**");
```

▼実行結果

```
RPGゲーム  
HP:10  
勇者が1の攻撃を受ける  
HP:9  
勇者が2の攻撃を受ける  
HP:7  
勇者が6の攻撃を受ける  
HP:1  
== 勇者よ、死んでしまうとは情けない ==  
勇者が9の攻撃を受ける  
HP:0  
**ゲームオーバー**
```

オーバーロード (GameCharクラス)

▼実行のサンプル

```
Console.WriteLine("RPGゲーム");  
Hello c = new Hello();  
c.ShowHP();  
Random r = new Random();  
while (c.HP != 0)  
{  
    int attack = r.Next(1, 10);  
    c.Attack(attack); // 敵の攻撃  
    Console.WriteLine("勇者が{0}の攻撃を受ける", attack);  
    c.ShowHP();  
}  
Console.WriteLine("**ゲームオーバー**");
```

▼実行結果

```
RPGゲーム  
HP:10  
勇者が1の攻撃を受ける  
HP:9  
勇者が2の攻撃を受ける  
HP:7  
勇者が6の攻撃を受ける  
HP:1  
== 勇者よ、死んでしまうとは情けない ==  
勇者が9の攻撃を受ける  
HP:0  
**ゲームオーバー**
```

HPが0に

HelloクラスのAttackメソッドを実行するとオーバーライドされた処理が実行される

オーバーライド（コンストラクター）

▼実行のサンプル

```
Student student = new Student(3, "山田太郎");  
student.Show();
```

▼実行結果

学年:3年 名前:山田太郎

オーバーライド（コンストラクター）

▼実行のサンプル

```
Student student = new Student(3, "山田太郎");  
student.Show();
```

▼実行結果

学年:3年 名前:山田太郎

①newによる
コンストラクタの呼び出し

②スーパークラス（Person）の
コンストラクタ呼び出し

③コンストラクタの実行

オーバーロード

- ・ 1つのクラスに同名のメソッドを複数定義すること
- ・ 戻り値の型、引数の組み合わせなどが異なるようにする
- ・ 呼び出す際には引数の違いで区別をする

オーバーロード

同名のメソッド (Run) が複数定義されている

▼オーバーロードのあるクラス

```
public class Car
{
    public void Run()
    {
        Console.WriteLine("走行する");
    }
    public void Run(int speed)
    {
        Console.WriteLine("時速{0}kmで走行する", speed);
    }
}
```

オーバーロード

同名のメソッド (Run) が複数定義されている

▼オーバーロードのあるクラス

```
public class Car
{
    public void Run()
    {
        Console.WriteLine("走行する");
    }
    public void Run(int speed)
    {
        Console.WriteLine("時速{0}kmで走行する", speed);
    }
}
```

引数なし

int型の引数あり

オーバーロード

▼実行のサンプル

```
Car car = new Car();  
car.Run();      // Runメソッド（引数なし）  
car.Run(50);    // Runメソッド（引数あり）
```

▼実行結果

走行する
時速50kmで走行する

オーバーロード

▼実行のサンプル

```
Car car = new Car();  
car.Run();      // Runメソッド（引数なし）  
car.Run(50);    // Runメソッド（引数あり）
```

引数なし

▼実行結果

走行する
時速50kmで走行する

オーバーロード

▼実行のサンプル

```
Car car = new Car();  
car.Run();      // Runメソッド（引数なし）  
car.Run(50);    // Runメソッド（引数あり）
```

▼実行結果

走行する
時速50kmで走行する

int型の引数あり

オーバーロード（コンストラクター）

コンストラクタが複数定義されている

▼オーバーロードのあるクラス

```
public class Person
{
    string name;
    public Person()
    {
        name = "NoName";
    }
    public Person(string name)
    {
        this.name = name;
    }
    public void ShowName()
    {
        Console.WriteLine(name);
    }
}
```

▼実行のサンプル

```
Person p1 = new Person();
p1.ShowName();
Person p2 = new Person("Taro");
p2.ShowName();
```

▼実行結果

```
NoName
Taro
```

オーバーロード（コンストラクター）

コンストラクタが複数定義されている

▼オーバーロードのあるクラス

```
public class Person
{
    string name;
    public Person()
    {
        name = "NoName";
    }
    public Person(string name)
    {
        this.name = name;
    }
    public void ShowName()
    {
        Console.WriteLine(name);
    }
}
```

▼実行のサンプル

引数なしのコンストラクタ

```
Person p1 = new Person();
p1.ShowName();
Person p2 = new Person("Taro");
p2.ShowName();
```

▼実行結果

```
NoName
Taro
```


オーバーロード（コンストラクター）

コンストラクタが複数定義されている

▼オーバーロードのあるクラス

```
public class Person
{
    string name;
    public Person()
    {
        name = "NoName";
    }
    public Person(string name)
    {
        this.name = name;
    }
    public void ShowName()
    {
        Console.WriteLine(name);
    }
}
```

▼実行のサンプル

```
Person p1 = new Person();
p1.ShowName();
Person p2 = new Person("Taro");
p2.ShowName();
```

▼実行結果

```
NoName
Taro
```

引数ありのコンストラクタ

ポリモーフィズム

- ・ **ポリモーフィズム (Polymorphism) … オブジェクト指向の概念の一つ**
- ・ **同名のメソッドを状況に応じて使い分けるようにできること**
- ・ **オーバーライド、オーバーロードもこれに含まれる**

抽象クラスとは

- ・ 抽象メソッドを持つクラス
- ・ 抽象メソッド ... 戻り値の型と引数のみ定義され実態がないメソッド
- ・ 抽象メソッドはサブクラスで実装されることが期待される
- ・ そのため抽象クラス自体はインスタンスを作ることができない

抽象クラス

クラス名と抽象メソッドにabstract修飾子がつく

▼Birdクラス（抽象クラス）

```
public abstract class Bird
{
    protected string name;
    // 通常のメソッド
    public void Show()
    {
        Console.WriteLine(name);
    }
    // 抽象メソッド
    public abstract void Sing();
}
```

▼インスタンスの生成はできない

```
// コメントをとるとエラーになる
//Bird b = new Bird();
```

抽象クラスは、「class」の前に抽象クラスであることを意味する「abstract」修飾子がつき、一つ以上の抽象メソッド（メソッドの前に「abstract」修飾子がつく）を持つ

抽象クラス

クラス名と抽象メソッドにabstract修飾子がつく

▼Birdクラス（抽象クラス）

```
public abstract class Bird
{
    protected string name;
    // 通常のメソッド
    public void Show()
    {
        Console.WriteLine(name);
    }
    // 抽象メソッド
    public abstract void Sing();
}
```

classの前に
「abstract」

抽象メソッド（実装がない）

▼インスタンスの生成はできない

```
// コメントをとるとエラーになる
//Bird b = new Bird();
```

抽象クラスは、「class」の前に抽象クラスであることを意味する「abstract」修飾子がつき、一つ以上の抽象メソッド（メソッドの前に「abstract」修飾子がつく）を持つ

抽象クラスとは

- ・ 抽象メソッドの先頭には「abstract」がつく
- ・ 抽象メソッドは暗黙的に「virtual」となっている
- ・ 抽象メソッドはサブクラスで同名のメソッドをオーバーライドして使う

抽象クラス

抽象クラスBirdクラスのサブクラスを実装する

▼抽象クラス

```
public class Crow : Bird
{
    public Crow()
    {
        base.name = "カラス";
    }
    public override void Sing()
    {
        Console.WriteLine("カーカー");
    }
}
```

▼インスタンスの生成とメソッド呼び出し

```
Crow cr = new Crow();
cr.Show();
cr.Sing();
```

▼実行結果

```
カラス
カーカー
```


抽象クラス

抽象クラスBirdクラスのサブクラスを実装する

▼抽象クラス

```
public class Crow : Bird
{
    public Crow()
    {
        base.name = "カラス";
    }
    public override void Sing()
    {
        Console.WriteLine("カーカー");
    }
}
```

▼インスタンスの生成とメソッド呼び出し

```
Crow cr = new Crow();
cr.Show();
cr.Sing();
```

実装された
抽象メソッドの
呼び出し

▼実行結果

```
カラス
カーカー
```

抽象メソッドの実装 (override)

抽象クラス

抽象クラスを作ると容易に多様な機能の別のクラスを作れる

▼抽象クラス

```
public class Chicken : Bird
{
    public Chicken()
    {
        base.name = "にわとり";
    }
    public override void Sing()
    {
        Console.WriteLine("コケコッコー");
    }
}
```

▼インスタンスの生成とメソッド呼び出し

```
Chicken ch = new Chicken();
ch.Show();
ch.Sing();
```

▼実行結果

```
にわとり
コケコッコー
```

抽象クラス

抽象クラスを作ると容易に多様な機能の別のクラスを作れる

▼抽象クラス

```
public class Chicken : Bird
{
    public Chicken()
    {
        base.name = "にわとり";
    }
    public override void Sing()
    {
        Console.WriteLine("コケコッコー");
    }
}
```

▼インスタンスの生成とメソッド呼び出し

```
Chicken ch = new Chicken();
ch.Show();
ch.Sing();
```

実装された
抽象メソッドの
呼び出し

▼実行結果

```
にわとり
コケコッコー
```

抽象メソッドの実装 (override)

抽象クラスを用いる利点

- ・ 似たような機能を持つ複数のクラスが容易に作れる
- ・ 共通する処理は通常のメソッドとして記述する
- ・ クラスごとにことなる機能は抽象メソッドにしサブクラスで実装する

抽象クラスを用いる利点

Crow、ChickenともにBirdクラスとして扱うことができる

▼Birdクラスのインスタンスの集合として扱う

```
Bird[] birds = new Bird[2];  
// カラス・ニワトリともに「鳥」として扱える  
birds[0] = new Crow();  
birds[1] = new Chicken();  
for(int i = 0; i < birds.Length; i++)  
{  
    birds[i].Sing();  
}
```

▼実行結果

にわとり
コケコッコー

インターフェースとは

- ・ 抽象クラスをさらに推し進めたもの
- ・ メソッドは抽象メソッドしか持つことができない
- ・ 1つのクラスは複数のメソッドを実装することができる

インターフェースとは

▼インターフェースをの書式

```
interface インターフェース名
{
    戻り値の型 抽象メソッド名 (引数1の型 引数1, 引数2の型 引数2, ... );
    戻り値の型 抽象メソッド名 (引数1の型 引数1, 引数2の型 引数2, ... );
    ...
}
```

- インターフェースの定義は「interface」から始まる
- 複数の抽象メソッドを持つがabstract修飾子をつける必要はない
- 定義されるメソッドは暗黙のうちにpublicであるので記述を省略できる
- 単体で用いることはなくクラスで実装をする

インターフェース

▼インターフェースを実装したクラス

```
public class CellPhone : IPhone,IMailer
{
    public void Phone(string telno)
    {
        Console.WriteLine("{0}に電話をかける",telno);
    }
    public void Mail(string addr)
    {
        Console.WriteLine("{0}にメールを送る", addr);
    }
}
```

▼インターフェース

```
public interface IPhone
{
    void Phone(string telno);
}
```

▼インターフェース

```
public interface IMailer
{
    void Mail(string addr);
}
```

インターフェース

インターフェース
(複数記述可能)

▼インターフェースを実装したクラス

```
public class CellPhone : IPhone,IMailer
{
    public void Phone(string telno)
    {
        Console.WriteLine("{0}に電話をかける",telno);
    }
    public void Mail(string addr)
    {
        Console.WriteLine("{0}にメールを送る", addr);
    }
}
```

実装

実装

▼インターフェース

```
public interface IPhone
{
    void Phone(string telno);
}
```

▼インターフェース

```
public interface IMailer
{
    void Mail(string addr);
}
```

インターフェース

▼インターフェースの利用(IPhone)

```
CellPhone cp = new CellPhone();  
IPhone ph = (IPhone)cp;  
ph.Phone("03-xxx-xxxx"); // 電話の機能  
// コメントをとるとエラー (Mailメソッドは使えない)  
// ph.Mail("hoge@fuga.com");
```

▼実行結果

03-xxx-xxxxに電話をかける

▼インターフェースの利用(IMailer)

```
IMailer ml = (IMailer)sp;  
ml.Mail("hoge@fuga.com"); // メール機能の利用  
// コメントをとるとエラーになる  
// ml.Phone("03-xxx-xxxx");
```

▼実行結果

hoge@fuga.comにメールを送る

インターフェース

▼インターフェースの利用(IPhone)

```
CellPhone cp = new CellPhone();  
iPhone ph = (iPhone)cp;  
ph.Phone("03-xxx-xxxx"); // 電話の機能  
// コメントをとるとエラー (Mailメソッドは使えない)  
// ph.Mail("hoge@fuga.com");
```

CellPhoneクラス型のcpをiPhone
クラス型に型変換 (キャスト)

▼実行結果

03-xxx-xxxxに電話をかける

▼インターフェースの利用(IMailer)

```
IMailer ml = (IMailer)sp;  
ml.Mail("hoge@fuga.com"); // メール機能の利用  
// コメントをとるとエラーになる  
// ml.Phone("03-xxx-xxxx");
```

CellPhoneクラス型のspをIMailer
クラス型に型変換 (キャスト)

▼実行結果

hoge@fuga.comにメールを送る

インターフェース

▼インターフェースの利用(IPhone)

```
CellPhone cp = new CellPhone();  
IPhone ph = (IPhone)cp;  
ph.Phone("03-xxx-xxxx"); // 電話の機能  
// コメントをとるとエラー (Mailメソッドは使えない)  
// ph.Mail("hoge@fuga.com");
```

▼実行結果

03-xxx-xxxxに電話をかける

CellPhone
クラスのメ
ソッド実行

▼インターフェースの利用(IMailer)

```
IMailer ml = (IMailer)sp;  
ml.Mail("hoge@fuga.com"); // メール機能の利用  
// コメントをとるとエラーになる  
// ml.Phone("03-xxx-xxxx");
```

▼実行結果

hoge@fuga.comにメールを送る

CellPhone
クラスのメ
ソッド実行

インターフェース

▼インターフェースの利用(IPhone)

```
CellPhone cp = new CellPhone();  
IPhone ph = (IPhone)cp;  
ph.Phone("03-xxx-xxxx"); // 電話の機能  
// コメントをとるとエラー (Mailメソッドは使えない)  
// ph.Mail("hoge@fuga.com");
```

CellPhoneクラスで定義してあるメソッドでも実行できない

▼実行結果

03-xxx-xxxxに電話をかける

▼インターフェースの利用(IMailer)

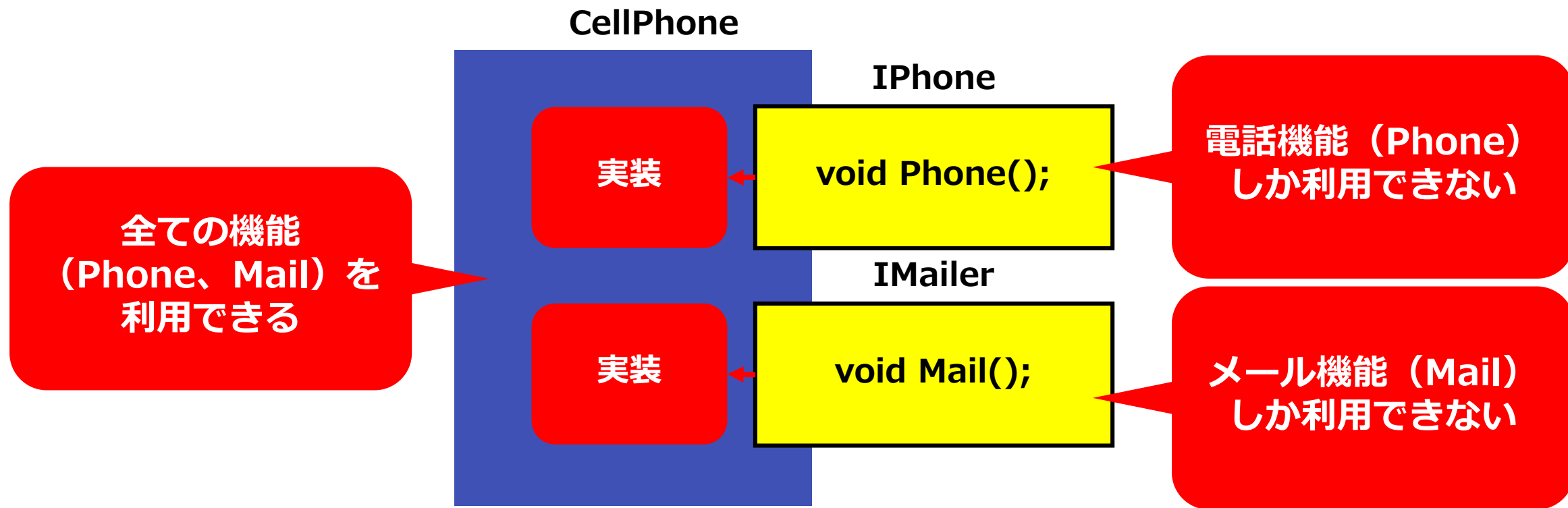
```
IMailer ml = (IMailer)sp;  
ml.Mail("hoge@fuga.com"); // メール機能の利用  
// コメントをとるとエラーになる  
// ml.Phone("03-xxx-xxxx");
```

CellPhoneクラスで定義してあるメソッドでも実行できない

▼実行結果

hoge@fuga.comにメールを送る

インターフェースを利用する意味



インターフェースを用いると1つのクラスがあたかも複数のクラスに分割したように扱うことができる。



6.抽象クラスとインターフェイスの違い、使いどころ

抽象メソッドとインターフェースの違い

- ・ 抽象クラスとインターフェースは似ているが役割が違う
- ・ 抽象クラスは、似たようなクラスを作るための「型」にできる
- ・ インターフェースでは異なるクラスに同一の操作を与える事ができる

インターフェース

▼SmartPhoneクラス

```
public class SmartPhone : IMusicPlayer,ICamera
{
    public void Play()
    {
        Console.WriteLine("スマホで音楽を聴く");
    }
    public void Photo()
    {
        Console.WriteLine("スマホで写真を撮る");
    }
}
```

▼IMusicPlayerインターフェース

```
public interface IMusicPlayer
{
    void Play();
}
```

▼ICameraインターフェース

```
public interface ICamera
{
    void Photo();
}
```

インターフェース

▼SmartPhoneクラス

```
public class SmartPhone : IMusicPlayer,ICamera
{
    public void Play()
    {
        Console.WriteLine("スマホで音楽を聴く");
    }
    public void Photo()
    {
        Console.WriteLine("スマホで写真を撮る");
    }
}
```

▼IMusicPlayerインターフェース

```
public interface IMusicPlayer
{
    void Play();
}
```

▼ICameraインターフェース

```
public interface ICamera
{
    void Photo();
}
```

インターフェースIMusicPlayer、ICamera実装

インターフェース

▼SmartPhoneクラス

```
public class SmartPhone : IMusicPlayer,ICamera
{
    public void Play()
    {
        Console.WriteLine("スマホで音楽を聴く");
    }
    public void Photo()
    {
        Console.WriteLine("スマホで写真を撮る");
    }
}
```

実装

▼IMusicPlayerインターフェース

```
public interface IMusicPlayer
{
    void Play();
}
```

▼ICameraインターフェース

```
public interface ICamera
{
    void Photo();
}
```

実装

インターフェース

▼Audioクラス

```
public class Audio : IMusicPlayer
{
    public void Play()
    {
        Console.WriteLine("オーディオで音楽を聴く");
    }
}
```

▼DigitalCameraクラス

```
public class DigitalCamera : ICamera
{
    public void Photo()
    {
        Console.WriteLine("デジカメで写真を撮る");
    }
}
```

▼IMusicPlayerインターフェース

```
public interface IMusicPlayer
{
    void Play();
}
```

▼ICameraインターフェース

```
public interface ICamera
{
    void Photo();
}
```

インターフェースの役割

▼Audioクラス

```
public class Audio : IMusicPlayer
{
    public void Play()
    {
        Console.WriteLine("オーディオで音楽を聴く");
    }
}
```

▼DigitalCameraクラス

```
public class DigitalCamera : ICamera
{
    public void Photo()
    {
        Console.WriteLine("デジカメで写真を撮る");
    }
}
```

▼IMusicPlayerインターフェース

```
public interface IMusicPlayer
{
    void Play();
}
```

▼ICameraインターフェース

```
public interface ICamera
{
    void Photo();
}
```

インターフェースの役割

▼Audioクラス

```
public class Audio : IMusicPlayer
{
    public void Play()
    {
        Console.WriteLine("オーディオで音楽を聴く");
    }
}
```

▼IMusicPlayerインターフェース

```
public interface IMusicPlayer
{
    void Play();
}
```

実装

▼DigitalCameraクラス

```
public class DigitalCamera : ICamera
{
    public void Photo()
    {
        Console.WriteLine("デジカメで写真を撮る");
    }
}
```

▼ICameraインターフェース

```
public interface ICamera
{
    void Photo();
}
```

実装

インターフェースの役割

▼IMusicPlayerインターフェースの利用

```
Audio audio = new Audio();  
SmartPhone phone = new SmartPhone();  
// 音楽プレイヤーもスマートフォンも音楽プレイヤーの使い方は共通  
IMusicPlayer mp;  
mp = (IMusicPlayer)audio;  
mp.Play();  
mp = (IMusicPlayer)phone;  
mp.Play();
```

▼実行結果

オーディオで音楽を聴く
スマホで音楽を聴く

インターフェースの役割

▼IMusicPlayerインターフェースの利用

```
Audio audio = new Audio();  
SmartPhone phone = new SmartPhone();  
// 音楽プレイヤーもスマートフォンも音楽プレイヤーの使い方は共通  
IMusicPlayer mp;  
mp = (IMusicPlayer)audio;  
mp.Play();  
mp = (IMusicPlayer)phone;  
mp.Play();
```

▼実行結果

オーディオで音楽を聴く
スマホで音楽を聴く

インターフェースの役割

▼ICameraインターフェースの利用

```
SmartPhone phone = new SmartPhone();  
DigitalCamera camera = new DigitalCamera();  
// デジタルカメラでもスマートフォンもカメラの使い方は共通  
ICamera cm;  
cm = (ICamera)phone;  
cm.Photo();  
cm = (ICamera)camera;  
cm.Photo();
```

▼実行結果

スマホで写真を撮る
デジカメで写真を撮る

インターフェースの役割

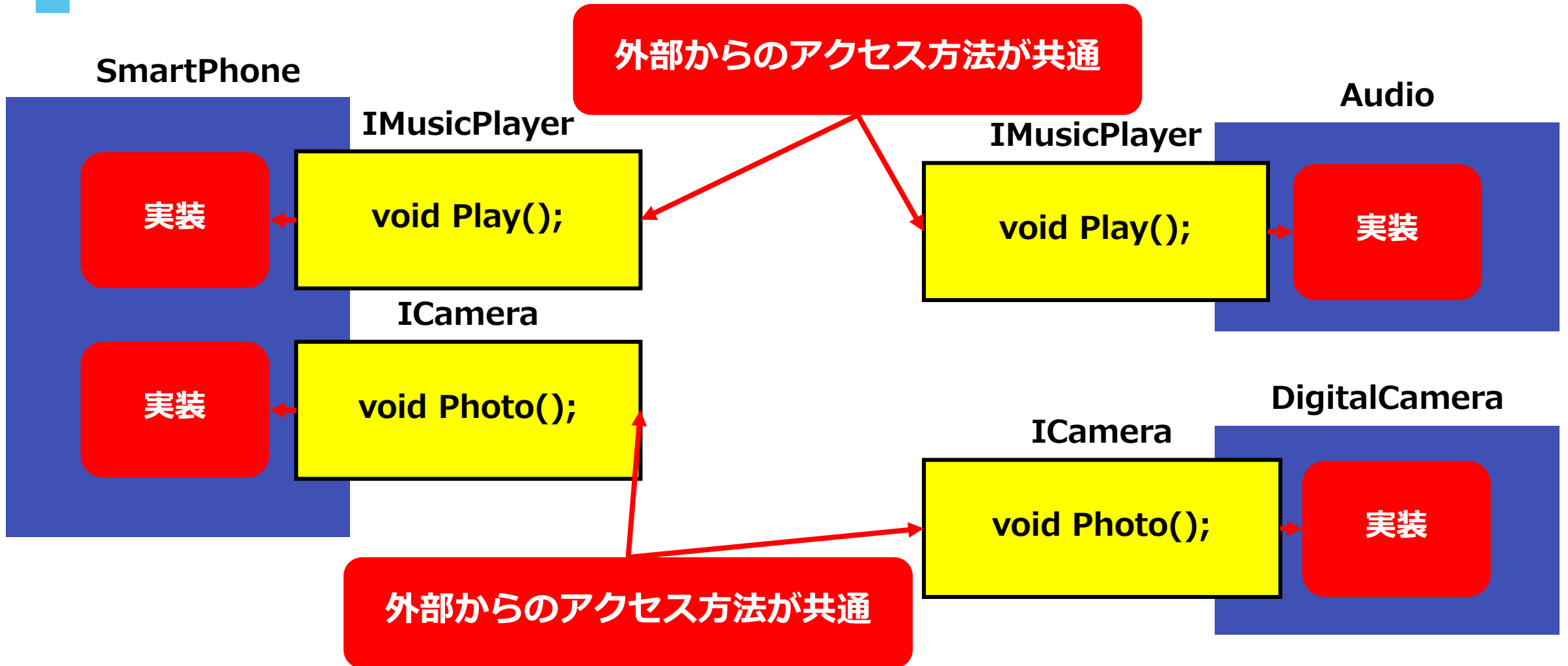
▼ICameraインターフェースの利用

```
SmartPhone phone = new SmartPhone();  
DigitalCamera camera = new DigitalCamera();  
// デジタルカメラでもスマートフォンもカメラの使い方は共通  
ICamera cm;  
cm = (ICamera)phone;  
cm.Photo();  
cm = (ICamera)camera;  
cm.Photo();
```

▼実行結果

スマホで写真を撮る
デジカメで写真を撮る

インターフェースの役割



インターフェースを用いると異なるクラスに同一の操作方法を持たせることができる

抽象クラス

▼AbsCarクラス

```
public abstract class AbsCar
{
    public abstract void Charge();
    public void Run()
    {
        Console.WriteLine("走行する");
    }
}
```

ガソリン車、電気自動車ともに走行
(Run) などの基本操作は共通だが、
燃料の補充方法(Charge) が異なる

▼GasCar (ガソリン車) クラス

```
public class GasCar : AbsCar
{
    public override void Charge()
    {
        Console.WriteLine("ガソリンを給油する");
    }
}
```

▼IElecCar (電気自動車) クラス

```
public class ElecCar : AbsCar
{
    public override void Charge()
    {
        Console.WriteLine("充電する");
    }
}
```

抽象クラス

▼AbsCarクラス

```
public abstract class AbsCar
{
    public abstract void Charge();
    public void Run()
    {
        Console.WriteLine("走行する");
    }
}
```

ガソリン車、電気自動車ともに走行
(Run) などの基本操作は共通だが、
燃料の補充方法(Charge) が異なる

実装

▼GasCar (ガソリン車) クラス

```
public class GasCar : AbsCar
{
    public override void Charge()
    {
        Console.WriteLine("ガソリンを給油する");
    }
}
```

▼IElecCar (電気自動車) クラス

```
public class ElecCar : AbsCar
{
    public override void Charge()
    {
        Console.WriteLine("充電する");
    }
}
```

抽象クラスの役割

▼抽象クラスの利用

```
// 内部の変更をユーザーに意識させることなくバージョンアップさせることができる
AbsCar gc = new GasCar(); // ガソリン車のインスタンスの生成
gc.Charge();
gc.Run();
AbsCar ec = new ElecCar(); // 電気自動車のインスタンスの生成
ec.Charge();
ec.Run();
```

▼実行結果

ガソリンを給油する
走行する
充電する
走行する

抽象クラス役割

▼抽象クラスの利用

```
// 内部の変更をユーザーに意識させることなくバージョンアップさせることができる  
AbsCar gc = new GasCar(); // ガソリン車のインスタンスの生成  
gc.Charge();  
gc.Run();  
AbsCar ec = new ElecCar(); // 電気自動車のインスタンスの生成  
ec.Charge();  
ec.Run();
```

▼実行結果

ガソリンを給油する

走行する

充電する

走行する

走行（Runメソッド）処理は共通

抽象クラス役割

▼抽象クラスの利用

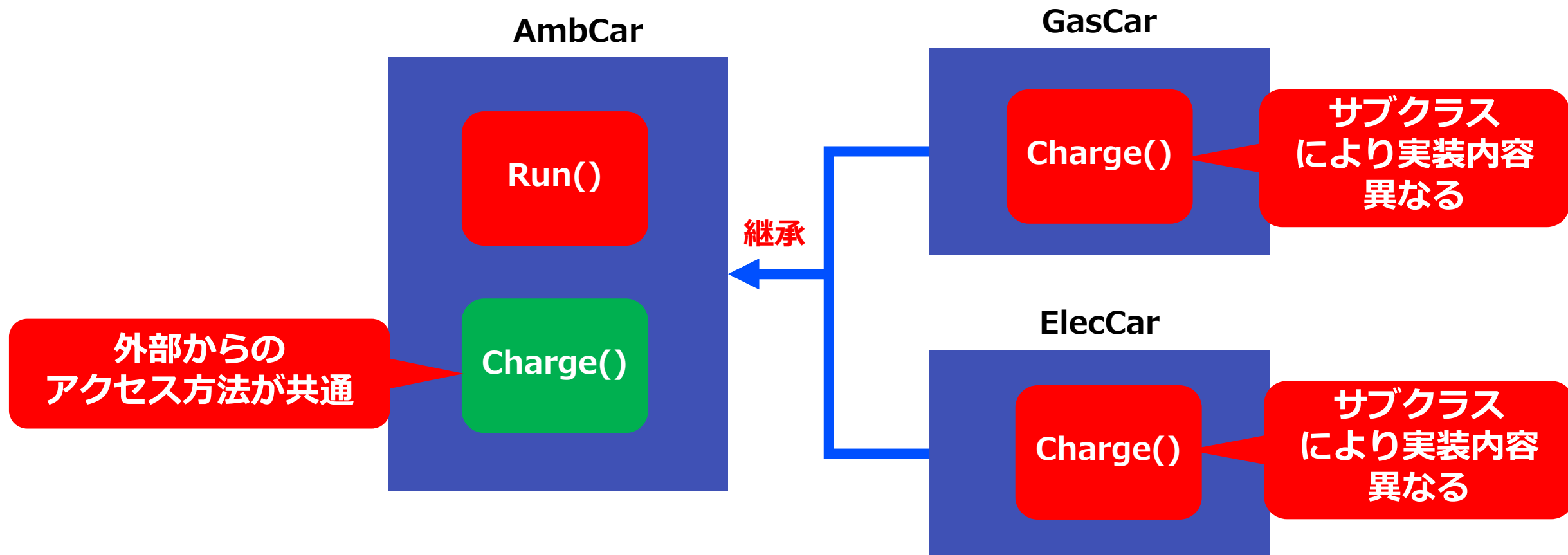
```
// 内部の変更をユーザーに意識させることなくバージョンアップさせることができる  
AbsCar gc = new GasCar(); // ガソリン車のインスタンスの生成  
gc.Charge();  
gc.Run();  
AbsCar ec = new ElecCar(); // 電気自動車のインスタンスの生成  
ec.Charge();  
ec.Run();
```

▼実行結果

ガソリンを給油する
走行する
充電する
走行する

燃料の補充（Charge）方法は異なる

抽象クラスの役割



抽象クラスを用いれば、基本操作が同じ異なるクラスを容易に作ることができる

インターフェースと抽象クラスの使い分けのポイント

- ・ インターフェースはクラスの「見え方」を規定する
- ・ クラスの一部をインターフェースにすれば機能制限に使える
- ・ 違うクラスに同一インターフェースを使えば同じ使い方ができるとわかる
- ・ 抽象クラスはクラスの「基本の型」を規定する
- ・ 共通機能を抽象クラスにすれば似たようなクラスを複数作れる
- ・ 抽象クラス経由でのアクセスにすればソースのバージョンアップが容易に

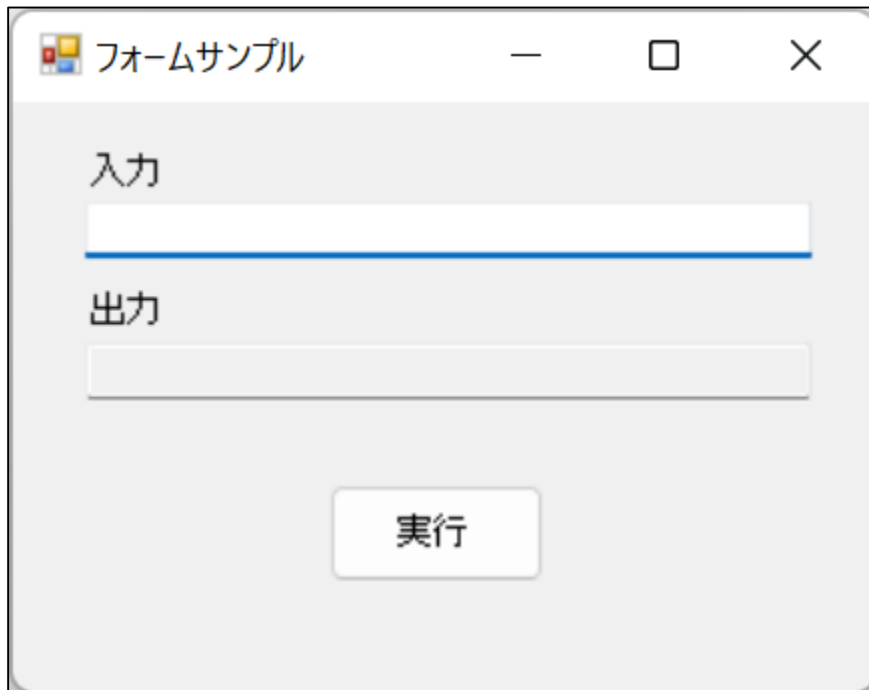


7. (復習)Formと継承

フォームのクラス

フォームとクラス

ツールはクラスであり配置されたものはインスタンスであるという関係性がある



フォームサンプル

入力

出力

実行

フォームのクラス

フォームとクラス

ツールはクラスであり配置されたものはインスタンスであるという関係性がある

インスタンス (変数名)

クラス

inputLabel

Label

inputText

TextBox

outputLabel

Label

outText

TextBox

runButton

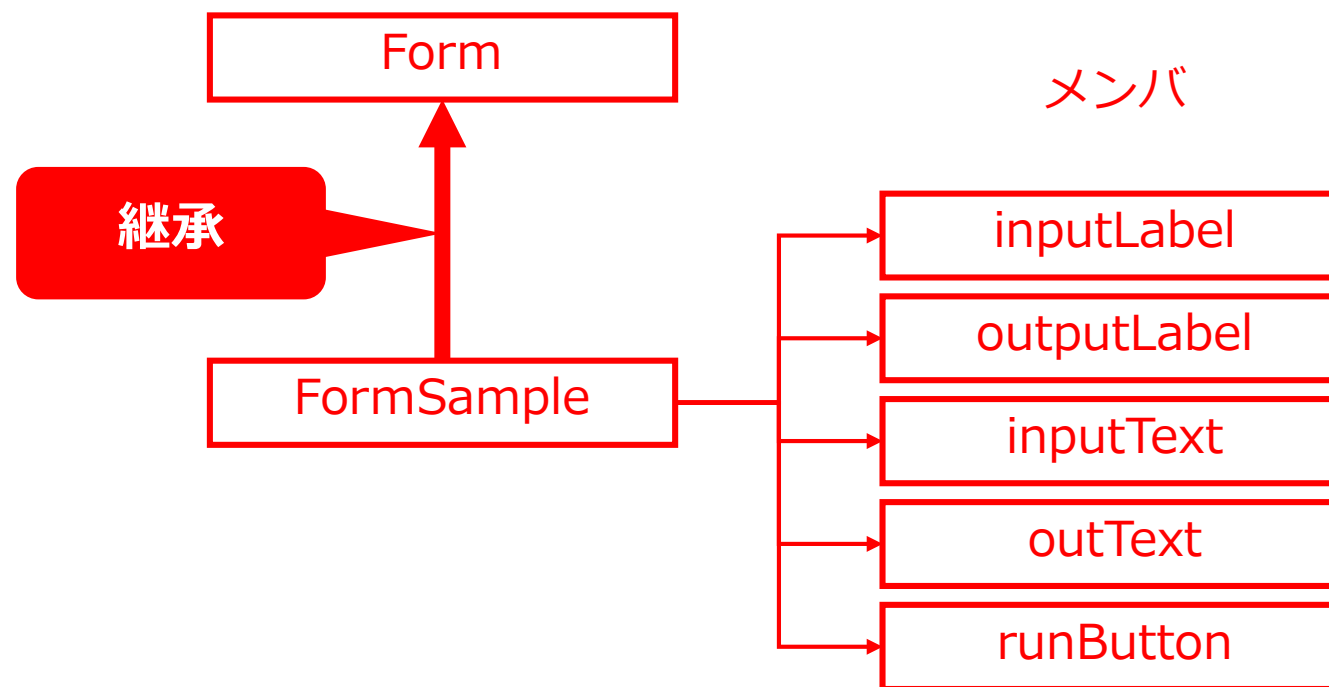
Button

FormSample

フォームのクラス

フォームのクラスの継承関係

FormSampleクラスはFormクラスを継承しメンバとして各種ツールを持つ



演習問題 1 : 単純なRPGの作成

つぎの条件をみたす怪物（Monster）クラスを作りなさい

- ・ スライド43のGameCharクラスを継承すること
- ・ コンストラクタでHPを整数型の引数として渡せるようにする
- ・ Attackメソッドをオーバーライドすること
- ・ Attackの基本処理はGameCharクラスのものを利用すること
- ・ AttackでHPが0になれば「 ==ゴールドをゲット! == 」と表示すること

演習問題：単純なRPGの作成 1

▼GameCharクラス

```
public class GameChar
{
    protected int hp; // ヒットポイント()
    public GameChar(int hp)
    {
        this.hp = hp;
    }
    public virtual void Attack(int damage)
    {
        this.hp -= damage;
        if (this.hp <= 0)
        {
            hp = 0;
        }
    }
    public void ShowHP()
    {
        Console.WriteLine("HP:{0}", this.hp);
    }
    public int HP
    {
        get { return hp; }
    }
}
```

▼解答（Monsterクラス）

```
public class Monster : GameChar
{
    public Monster(int hp) : base(hp)
    {
    }
    public override void Attack(int damage)
    {
        base.Attack(damage);
        if (hp == 0)
        {
            Console.WriteLine("==ゴールドをゲット！==");
        }
    }
}
```

Helloクラスの場合と同様にGameCharクラスを継承し
メソッドをオーバーライドする

演習問題：単純なRPGの作成 1

▼GameCharクラス

```
public class GameChar
{
    protected int hp; // ヒットポイント()
    public GameChar(int hp)
    {
        this.hp = hp;
    }
    public virtual void Attack(int damage)
    {
        this.hp -= damage;
        if (this.hp <= 0)
        {
            hp = 0;
        }
    }
    public void ShowHP()
    {
        Console.WriteLine("HP:{0}", this.hp);
    }
    public int HP
    {
        get { return hp; }
    }
}
```

▼解答（Monsterクラス）

```
public class Monster : GameChar
{
    public Monster(int hp) : base(hp)
    {
    }
    public override void Attack(int damage)
    {
        base.Attack(damage);
        if (hp == 0)
        {
            Console.WriteLine("==ゴールドをゲット！==");
        }
    }
}
```

引数でHPを渡す

HPが0の時の処理

Helloクラスの場合と同様にGameCharクラスを継承し
メソッドをオーバーライドする

演習問題2：単純なRPGの作成

スライド45のRPGゲームの処理を以下のように変更しなさい

- ・ Helloクラスの代わりに演習問題 1 で作成したMonsterクラスを利用する
- ・ Attackメソッドの次に「モンスターが○のダメージを受ける」と表示する
- ・ 前述の○にはAttackメソッドの引数を表示する
- ・ 最後の処理は「**敵を撃破!!**」と表示する処理にすること

オーバーロード (GameCharクラス)

▼実行のサンプル

```
Console.WriteLine("RPGゲーム");
Monster c = new Monster(10);
c.ShowHP();
Random r = new Random();
while (c.HP != 0)
{
    int attack = r.Next(1, 10);
    c.Attack(attack); // 敵の攻撃
    Console.WriteLine("モンスターが{0}のダメージを受ける", attack);
    c.ShowHP();
}
Console.WriteLine("**敵を撃破!!**");
```

▼実行結果

```
RPGゲーム
HP:10
モンスターが7のダメージを受ける
HP:3
==ゴールドをゲット!==
モンスターが9のダメージを受ける
HP:0
**敵を撃破!!**
```

オーバーロード (GameCharクラス)

▼実行のサンプル

```
Console.WriteLine("RPGゲーム");  
Monster c = new Monster(10);  
c.ShowHP();  
Random r = new Random();  
while (c.HP != 0)  
{  
    int attack = r.Next(1, 10);  
    c.Attack(attack); // 敵の攻撃  
    Console.WriteLine("モンスターが{0}のダメージを受ける", attack);  
    c.ShowHP();  
}  
Console.WriteLine("**敵を撃破!!**");
```

▼実行結果

```
RPGゲーム  
HP:10  
モンスターが7のダメージを受ける  
HP:3  
==ゴールドをゲット!==  
モンスターが9のダメージを受ける  
HP:0  
**敵を撃破!!**
```