C#???

????

????

記入できるファイルをダウンロード

????????

ゲームを作っている間に、ゲームに関するいろんなデータを管理しなければならない。以下のデータについて考え、最も適する「データ型」を選択してください。

データ	型	例
レースコースの名称	答え 文字列 (string)	答え `string courseName = "レインボーコース"`
ステージの残り時間	答え 小数型 (float)	答え `float timeRemaining = 60.0f;`

C#???????

以下の表は、C#で書いた命令とその日本語の意味が書いてある。空枠を完成してください。

C#	日本語

`string firstName = "太郎";`
`string lastName = "大原";`
`string fullName = lastName + " " + firstName;`
	答え
	下の名前（firstName）と苗字（lastName）の文字列型変数を作成し、「太郎」と「大原」を代入する。
	最後、フルネームの変数を作成し、名前と苗字を連結し、代入する。
	つまり、「fullName」の中には「大原 太郎」が入っている

????????????????????var?

今まで、変数を宣言するときに、どのデータを入れるべきかを「型」で指定しなければならない。ただ、宣言の時に一緒に何かの「値」を代入する場合、その「値」から直接に「型」を判断し、自動的に変数の「型」を指定することもできる。

この場合は、変数を「var」で宣言してください。

例:

```
var counter = 0;         // 0 は整数なので、counterはintになる
var height = 1.70f;      // 1.70fは実数（小数）なので、heightはfloatになる
var name = "Player 1";   // "Player 1"は文字列なので、nameはstringになる
var isGameOver = false; // falseは「偽」なので、isGameOverはboolになる
```

「var」を使うことにより、代入する数値に合わせて、型を決める

Q. いつ「var」を使うか?
A. 自由に使えるが、普通に「一時的に使い捨てる変数」を使うケースが多い。

Q. 使えない時もある?
A. 以下の場合は「var」を使えない

- 宣言のみの時（代入なし）

```
var number = 10;  // OK!
var number;       // NG
```

- メソッドの「引数」と「戻り値」（ ※メソッドについてはまた後で）
- クラスの「メンバー変数」（ ※クラスとメンバー変数についてはまた後で）

??

????

記入できるファイルをダウンロード

C#???????

以下の表は、C#で書いた命令とその日本語の意味が書いてある。空枠を完成してください。

C#	日本語
`a > b`	変数「a」は変数「b」よりも大きい?
`i <= 10`	答え 変数「i」は「10」以下?
答え `playerHP > 0`	プレーヤーの体力は残っている?
`stageSelect == "Tutorial"`	答え 選択されたステージはチュートリアルであるか?
`coinCount >= itemCost`	答え 持っているコイン数はアイテム価格を超える?

C#	日本語
答え `keyCount == 3`	鍵がちょうど 3個ある?
答え `time < 60`	残りの時間は 60秒未満?
`player != "CPU Player"`	答え プレーヤーは CPUプレーヤーではないね?

?????

以下の比較を確認し、「true」（真・はい）または「false」（偽・いいえ）をチェックしてください。ただし、事前に以下の変数をすでに作成されている:

```
int a = 100;
int b = 20;
float c = 5.0f;
```

比較	true	false
`a == 100;`	✔	
`b < c`	答え falseでした! >> ・ b は 20 ・ c は 5 ・ b は c 未満 (<) ではない	答え ✔

比較	**true**	**false**
`c >= a / b;`	答え ✔	答え << trueでした! ・ c は 5 ・ a / b ⇒ 100 /20 ⇒ 5 ・ 5は 5以上 (>=)である
`c * 6 < b * 2;`	答え ✔	答え << trueでした! ・ c×6 ⇒ 5×6 ⇒ 30 ・ b×2 ⇒ 20×2 ⇒ 40 ・ 30は 40未満（ <）である
`a == b * c;`	答え ✔	答え << trueでした! ・ aは 100 ・ b×c⇒ 20×5 ⇒ 100 ・同じ（ ==）である
`b != a - 80;`	答え falseでした! >> ・ b は 20 ・ a - 80 ⇒ 100 - 80 ⇒ 20 ・異なる（ !=）のではない	答え ✔

??????????????????????

ゲームを開発しているときに、ifで同時に2つのことを確認したい場合もある。

例えば、レースゲームの勝利条件は：

- ゴールに着いた
- 時間がまだ残っている

両方とも同時に「true」だったら、ゲームの勝ちになる。2つの条件同時に満たさないといけない場合は「ANDを演算子」を使いましょう

????????AND??????`&&`

この演算子で両方の条件を満たさないと「true」にならない。2つの条件「A」と「B」があるとしたら:

A	B	A && B
false	false	false
true	false	false
false	true	false
true	true	true

つまり、いずれかの条件が「false」でしたら、すべて「false」になる。上記の例で、C#にすると:

```
// 走行距離
float runDistance = 1235.0f;

// ゴールまでの距離
float goalDistance = 1200.0f

// 残りの時間
float timeLeft = 35.7f;

// 走行距離はゴールよりも大きく「かつ」残りの時間が0以上だったら
if (runDistance >= goalDistance && timeLeft > 0)
{
  Debug.Log("勝ち!");
}
```

これで、必ず2つの条件を満たしたら、ゲームクリアになることを保証できる。

?????????OR?????`||`

もう一つのパターンは「いずれかの条件がtrue だったら、すべて true になる」がある。例えば、あるゲームの中で2つのドアの開き方がある:

- ドアの鍵を持っている
- プレーヤーの力が十分だったら、自力で壊す

この場合は、1つだけの条件を満たしたら、ドアを開くことができるので、「または」の演算子を使えば良いでしょう。

A	B	A || B
false	false	false
true	false	true
false	true	true
true	true	true

上記の例で、C#にすると:

```
// プレーヤーの力
float playerPower = 30;

// 今持っている鍵の数
int keyCount = 0;

// ドアの固さ
float doorHardness = 25;

// 鍵があれば「または」力は固さよりも大きければ…
if (keyCount > 0 || playerPower >= doorHardness)
{
  Debug.Log("ドアを開ける! ");
}
```

この例だと、鍵がなくても力で壊せるので、ドアを開くことができる!

??

以下、使った「かつ」と「または」の比較を確認し、「true」(真・はい)または「false」(偽・いいえ)を書いてください。ただし、事前に以下の変数をすでに作成されている:

```
int coin = 230;        // コインの数
int maxItem = 10;      // 最大のアイテムの数
int nowItem = 9;       // 現在持っているアイテムの数
int potionCost = 100; // ポーションのコスト
```

1. 比較:

```
coin > potionCost && nowItem + 1 <= maxItem
```

true	false

答え
✔
	答え
	<< trueでした!
	コインは230でポーションは100なので
	`coin > potionCost`
	は `true` (真) である
	で、持っているアイテムは9個で、最大10個なので
	`nowItem + 1 <= maxItem`
	も `true` (真) である
	両方とも「true」であり、「true かつ true」は「true」

2. 比較:

```
potionCost > coin || nowItem >= maxItem
```

true	false

答え

false でした! >>

ポーション（100）はコイン（230）を超えないので

```
potionCost > coin
```

は「false」である。

で、持っているアイテムの数（9）最大アイテム以上ではないため

```
nowItem >= maxItem
```

も「false」である

「false」または「false」（両方も「偽」）であるため、結果は false である

答え

✔

3. 比較:

```
potionCost * 2 <= coin && maxItem >= nowItem + 2
```

true	**false**

答え
false でした! >>
ポーション2個は200で、もっといるコインは230なので
`potionCost * 2 <= coin`
は `true` （真）である
だが、持っているアイテム（9個）にさらに2つを増やすと、最大数（10個）を超えてしまうので
`maxItem >= nowItem + 2`
も `false` （偽）である
「かつ」（&&）は両方とも「true」じゃないとだめなので、最終結果は「false」になってしまう。
	答え
	✔

4. 比較:

```
nowItem == 0 && coin != 0
```

true	**false**
答え false でした! >> 持っているアイテムは0ではなく、9個なので `nowItem == 0` は `false` （偽）である 「かつ」（&&）は両方とも「true」じゃないとだめなので、もう一つの条件は関係なく、最終結果は「false」になってしまう。	答え ✔

5. 比較:

```
maxItem - nowItem > 9 || coin > 0
```

true	false

答え
✔
	答え
	<< trueでした!
	maxItemは 10で、 nowItemは 9なので
	maxItem - nowItem ⇒ 10 - 9 ⇒ 1
	で、 1 は 9 以上ではないので、
	`maxItem - nowItem > 9`
	は `false` (偽) である
	だが、コインの数 (230) は0以上なので
	`coin > 0`
	も `true` (真) である
	「または」 (||) の場合は、 いずれか_____ 真だったら、すべて真になるため、この比較は「 true 」である。

????

????

C#?????????????

1. 以下の　C#のプログラムを完成してください

```
// ゲームでダメージを受け、残機を確認し、ゲームオーバーかどうか判定する

int hp = 50; // 体力
int lives = 1; // 残機
int damage = 100; // ダメージ

// ダメージを受け、体力を減らす

________________________________________

// もし、体力が0以下だったら...

________________________________________
{
    // 残機を一つ減らす

    ______________________________________
}

// もし、残機がゼロだったら

________________________________________
{
    Debug.Log("ゲームオーバー");
}
```

```
// そうじゃない場合は、残機を表示
else
{
    Debug.Log($"残機: {lives}");
}
```

答え

```
// ゲームでダメージを受け、残機を確認し、ゲームオーバーかどうか判定する

int hp = 50; // 体力
int lives = 1; // 残機
int damage = 100; // ダメージ

// ダメージを受け、体力を減らす
hp -= damage; // または hp = hp - damage;

// もし、体力が0以下だったら...
if (hp <= 0)
{
    // 残機を一つ減らす
    lives--;
}

// もし、残機がゼロだったら
if (lives == 0)
{
    Debug.Log("ゲームオーバー");
}

// そうじゃない場合は、残機を表示
else
{
    Debug.Log($"残機: {lives}");
}
```

このプログラムはコンソールで何を表示するか? その出力を書いてください:

答え

ゲームオーバー

---

2. 以下の C#のプログラムを完成してください

```
// レースゲームで、チェックポイントを通過することにより、時間切れを延期する

float timeLeft = 8.51; // 残りの時間（秒数）
float raceDistance = 3643.43; //走行距離

float checkPoint01 = 2300; // スタート → チェックポイント1の距離
float checkPoint02 = 1500; //  チェックポイント1 → チェックポイント2距離
float goalPoint = 2500; // チェックポイント2 → ゴールまでの距離

// もし、ゴールを超えていたら...

_____________________________________________________
{
    Debug.Log("レース終了！");
}

// 一方、もし走行距離がチェックポイント２を超えたら...

_____________________________________________________
{
    Debug.Log("チェックポイント２通過！");
    timeLeft += 80;
}

// 一方、もし走行距離がチェックポイント１を超えたら...

_____________________________________________________
{
    Debug.Log("チェックポイント1通過！");
    timeLeft += 60;
}
```

```
// 残りの時間を表示
Debug.Log($"残りの時間: {timeLeft}");
```

答え

```
// レースゲームで、チェックポイントを通過することにより、時間切れを延期する

float timeLeft = 8.51; // 残りの時間（秒数）
float raceDistance = 3643.43; //走行距離

float checkPoint01 = 2300; // スタート → チェックポイント1の距離
float checkPoint02 = 1500; //  チェックポイント1 → チェックポイント2距離
float goalPoint = 2500; // チェックポイント2 → ゴールまでの距離

// もし、ゴールを超えていたら...
if (raceDistance >= checkPoint01 + checkPoint02 + goalPoint)
{
    Debug.Log("レース終了！");
}

// 一方、もし走行距離がチェックポイント2を超えたら...
else if (raceDistance >= checkPoint01 + checkPoint02)
{
    Debug.Log("チェックポイント2通過！");
    timeLeft += 80;
}

// 一方、もし走行距離がチェックポイント1を超えたら...
else if (raceDistance >= checkPoint01)
{
    Debug.Log("チェックポイント1通過！");
    timeLeft += 60;
}

// 残りの時間を表示
Debug.Log($"残りの時間: {timeLeft}");
```

このプログラムはコンソールで何を表示するか？ その出力を書いてください：

答え

チェックポイント1通過!

残りの時間: 68.51

Unity???

体重と身長でBMIを計算し、画面で適切な出力を表示しよう（参考: BMIと適正体重)

日本肥満学会の判定基準

BMI値	判定
18.5未満	低体重(痩せ型)
18.5～25未満	普通体重
25～30未満	肥満(1度)
30～35未満	肥満(2度)
35～40未満	肥満(3度)
40以上	肥満(4度)

式: BMI ＝ 体重 kg ÷ (身長 m)^2

ヒント:

- 体重と慎重の変数を作成し、適当な数値を代入
- BMIを計算
- if分で BMIの判定し、コンソールで表示

答え

```
// 体重
float weight = 70;


// 身長
float height = 1.71;


// BMIを計算
float bmi = weight / (height * height);
```

```
if (bmi < 18.5)
{
    Debug.Log("低体重");
}
else if (bmi < 25)
{
    Debug.Log("普通体重");
}
else if (bmi < 30)
{
    Debug.Log("肥満(1度)");
}
else if (bmi < 35)
{
    Debug.Log("肥満(2度)");
}
else if (bmi < 40)
{
    Debug.Log("肥満(3度)");
}
else
{
    Debug.Log("肥満(4度)");
}
```

?????

??????????==???????switch-case?

次の問題について考えてみてください

サイコロの数値は1から6までの間にして、その値を漢字で表示したい。つまり「1」の場合は「一」、「2」の場合は「二」など。if-elseで作成するとしたら、以下のようなプログラムになる:

```
// サイコロの値
int dice = 5;
```

```
if (dice == 1)
{
    Debug.Log("一");
}
else if (dice == 2)
{
    Debug.Log("二");
}
else if (dice == 3)
{
    Debug.Log("三");
}
else if (dice == 4)
{
    Debug.Log("四");
}
else if (dice == 5)
{
    Debug.Log("五");
}
else if (dice == 6)
{
    Debug.Log("六");
}
else
{
    Debug.Log("無効な数値");
}
```

このプログラムは、カスケードのように「1じゃなければ、2を確認する、2じゃなければ、3を確認する」、次々に「==」で比較するパターンは時々に現れる。この場合は「switch-case」が便利。上のプログラムは以下のようになる:

```
// サイコロの値
int dice = 5;

switch(dice)
{
case 1: // dice == 1 と同じ
    Debug.Log("一");
```

```
    break;
case 2: // dice == 2 と同じ
    Debug.Log("二");
    break;
case 3: // dice == 3 と同じ
    Debug.Log("三");
    break;
case 4: // dice == 4 と同じ
    Debug.Log("四");
    break;
case 5: // dice == 5 と同じ
    Debug.Log("五");
    break;
case 6: // dice == 6 と同じ
    Debug.Log("六");
    break;
default: // elseと同じ
    Debug.Log("無効な数値");
    break;
}
```

??????????????enum?

次の問題について考えてみてください。武器の種類により、攻撃力を変える処理を作りたい。処理しやすくするため、それぞれの武器に整数を与えた:

0: 非武装 (攻撃力 : 1)
1: 剣 (攻撃力 : 5)
2: ナイフ (攻撃力 : 2)
3: 鉄砲 (攻撃力 : 10)

「 switch-case」を使えば、以下のような処理になる:

```
int damage = 0;
int weapon = 2;
switch(weapon)
{
case 0:
  damage = 1;
  break;
case 1:
```

```
    damage = 5;
    break;
  case 2:
    damage = 2;
    break;
  case 3:
    damage = 10;
    break;
  }
  Debug.Log($"攻撃力: {damage}");
```

これでも実行ができるが、時間がたてば「0はなんだったっけ? 」「1は? 」のように、それぞれの整数の意味を忘れてしまう可能性がある。各数値に忘れないように、名前を付ける方法がある。「列挙」を作成することにより、各整数に読みやすい名前を付けることができる。

まず、列挙を定義しなければならない:

書式___

```
enum 列挙名
{
  整数「0」の名前,
  整数「1」の名前,
  整数「2」の名前,
  ...
  整数「n」の名前,
}
```

武器の例で、以下の列挙は適切でしょう:

```
enum Weapon
{
  Unarmed, // 0:非武装
  Sword,   // 1:剣
  Knife,   // 2:ナイフ
  Pistol,  // 3:鉄砲
}
```

そして、「 `switch-case`」はとても分かりやすくなる!

```
int damage;
Weapon wpn = Weapon.Knife;  // 新し「Weapon」というデータ型を使える!
switch(wpn)
{
case Weapon.Unarmed: // 「0」と同じ
  damage = 1;
  break;
case Weapon.Sword:   // 「1」と同じ
  damage = 5;
  break;
case Weapon.Knife:   //「2」と同じ
  damage = 2;
  break;
case Weapon.Pistol:  //「3」と同じ
  damage = 10;
  break;
}
Debug.Log($"攻撃力: {damage}");
```

????

ゲームでのアイテムショップを作ろうとして、以下のアイテムとその価格が決まった:

ポーション (小)	10ゼニ
ポーション (大)	25ゼニ
木の盾	50ゼニ
鉄の盾	100ゼニ
皮のブーツ	40ゼニ
魔法のマント	250ゼニ

ここで、次のプログラムを C#で作成し、 Unityで確認してください

1. 各種のアイテムを表すことができる「`ItemType`」の列挙を作成
2. プレーヤーを持っているお金の変数を準備 (`playerMoney` - 整数型)
3. プレーヤーが買いたいアイテムの変数を準備 (`buyItem` - `ItemType`の列挙型)
4. switch-caseで定めたアイテムの価格を求める (`itemCost` - 整数)

5. 変えるかどうかを　if分で判定し、コンソールで結果を表示

答え

```
// 各アイテムの列挙（名前を多少に異なる可能性あり！）
enum ItemType
{
  PotionSmall,   // ポーション（小）
  PotionLarge,   // ポーション（大）
  WoodShield,    // 木の盾
  IronShield,    // 鉄の盾
  LeatherBoots,  // 皮のブーツ
  MagicCape,     // 魔法のマント
}

// 持っているお金
int playerMoney = 150;

// 買いたいアイテム
ItemType buyItem = ItemType.WoodShield; // なんでもOK

// 買いたいアイテムの価格
int itemCost;
switch(buyItem)
{
    case ItemType.PotionSmall:
        itemCost = 10;
        break;
    case ItemType.PotionLarge:
        itemCost = 25;
        break;
    case ItemType.WoodShield:
        itemCost = 50;
        break;
    case ItemType.IronShield:
        itemCost = 100;
        break;
    case ItemType.LeatherBoots:
        itemCost = 40;
```

```
        break;
    case ItemType.MagicCape:
        itemCost = 250;
        break;
}

// 買えるかどうかを確認
if (itemCost <= playerMoney)
{
    Debug.Log("購入可能! ");
}
else
{
    Debug.Log("お金不足～");
}
```

????

???????????????for?

記入できるファイルをダウンロード

????

C#???????

以下の表は、C#で書いた命令とその日本語の意味が書いてある。空枠を完成してください。

C#	日本語
`for(int i = 0; i < 10; i++)`	「i」という変数は0から9までの値を変えながら、10回繰り返す
`for(int i = 1; i <= 10; i++)`	答え 「i」という変数は1から10までの値を変えながら、10回繰り返す
`for(int i = 10; i > 0; i--)`	答え 「i」という変数は10から1までの値を変えながら、10回繰り返す
`for(int i = 0; i < 15; i+=2)`	答え 「i」という変数は0, 2, 4, 6, 8, 10, 12, 14の値を変えながら、8回繰り返す

Unity???

1～100の整数を繰り返し、表示してください。ただし…

「3」で割れる整数の代わりに「ho」を表示
「5」で割れる整数の代わりに「ge」を表示
「3」と「5」両方で割れる整数の代わりに「hoge」を表示

例:

1, 2, ho, 4, ge, ho, 7, 8, ho, ge, 11, ho, 13, 14, hoge, 16, 17, ho …

ヒント

「余り」演算子（モジュロ、剰余演算子）を使ってください。余りの演算子は整数の割り算の余りを返す:

```
int a = 4 % 3;   // a は 1 になる
int b = 6 % 3;   // b は 0 になる
int c = 12 % 5;  // c は 2 になる
```

答え

```
// 1~100繰り返す
for(int i = 1; i <= 100; i++)
{
    // 3 と 5 のあまりを計算し、0 である場合は、
    // その数値で割れることである。
    bool div3 = i % 3 == 0;
    bool div5 = i % 5 == 0;

    // 両方で割れる?
    if (div3 && div5)
    {
        Debug.Log("hoge");
    }
    // 3で割れる?
    else if (div3)
    {
        Debug.Log("ho");
    }
    // 5で割れる?
    else if (div5)
    {
        Debug.Log("ge");
    }
    // その他
    else
    {
```

```
        Debug.Log($"{i}");
    }
}
```

?????????????????while?

for文は決まった回数（例: 0~ 10）を繰り返す命令であるが、場合により、回数が決まっていなく、何かの条件を満たしている間には繰り返す場合もある:

例: 体力残っていたら →遊び続ける

例: 時間切れになっていなければ →遊び続ける

例: 敵がプレーヤーを捕まっていなければ →追いかける

書式:

```
while(繰り返し条件)
{
    繰り返したい処理
}
```

例えば、サイコロは「6」になるまで繰り返すを書くとしたら:

```
int dice = 0;
while (dice != 6)
{
    // Random.RangeはUnityの機能であり、乱数を生成する
    dice = Random.Range(1, 7);  // 1~6の間の乱数を生成（7未満）
    Debug.Log($"サイコロの値: {dice}");
}
Debug.Log("終了");
```

上記の例で、回数が決まっていなくて、サイコロは6以外だったら繰り返す処理である。

????

以下のスクリプトでwhileを使い、完成してください（Unityでも OK）

```
// 体力
int playerHP = 100;

// プレーヤーが生きていたら、繰り返す
________________________
{
    Debug.Log($"現在の体力 {playerHP}");
    int damage = 30;
    playerHP -= damage;  // 体力を減らす
}

Debug.Log("ゲームオーバー");
```

答え

```
// 体力
int playerHP = 100;

// プレーヤーが生きていたら、繰り返す
while(playerHP > 0)
{
    Debug.Log($"現在の体力 {playerHP}");
    int damage = 30;
    playerHP -= damage;  // 体力を減らす
}

Debug.Log("ゲームオーバー");
```

このプログラムはコンソールで何を表示するか? その出力を書いてください:

答え

```
現在の体力    100
現在の体力    70
現在の体力    40
現在の体力    10
```

??

????

記入できるファイルをダウンロード

C#???????

以下の表は、C#で書いた命令とその日本語の意味が書いてある。空枠を完成してください。

C#	日本語
`int [] numbers = new int[10]`	10個の整数が入る「numbers」という配列を作成
`int [] numbers;`	答え 整数型の配列を宣言のみ（初期化なし）
`string [] names = new string[5];`	答え 5個の文字列が入る「names」という配列を作成
答え `names[0] = "たろう";`	「names」配列の一番最初（先頭）は「たろう」にする
答え `names[4] = "おおはら";`	「names」配列の一番最後（後尾）は「おおはら」にする

C#	日本語
`int a = numbers[3];`	答え 配列「numbers」の先頭から4つ目の要素を変数「a」の中に代入
答え	少数型の配列「distances」を宣言し、1.3f, 4.5f, 7.9fの3つの数値で初期化する

```
float [] distances = new float[3];
distances[0] = 1.3f;
distances[1] = 4.5f;
distances[2] = 7.9f;

// または
float [] distances =
   new float[3] { 1.3f, 4.5f, 7.9f };
```

C#?????????????

1. 配列とfor文を使い、以下のスクリプトを完成してください（Unityでも OK）

※：文字列の長さは「変数名.Length」で求めることができる

例：

```
string s = "Hello!";
int size = s.Length; // 文字列「s」の長さを求める
Debug.Log(size); // 6 を表示する
```

※：期待の結果は「最も長いフルーツ名は: grapefruit」

```
// フルーツの名前の配列
string [] fruits = {"lemon", "mikan", "orange", "grapefruit", "kiwi"};

// 最も長い文字列（結果）
string maxFruit = "";

// フルーツ1個ずつ確認
```

```
for (________; _________; ________)
{
    // i番目のフルーツを求める
    string f = _______________;

    // もし、今回のフルーツ名の方が長ければ
    if (f.Length > maxFruit.Length)
    {
        // 最も長いフルーツを更新
        ___________________;
    }
}

// 最大の長さのフルーツ名を表示
Debug.Log($"最も長いフルーツ名は: {______________}");
```

答え

```
// フルーツの名前の配列
string [] fruits = {"lemon", "mikan", "orange", "grapefruit", "kiwi"};

// 最も長い文字列（結果）
string maxFruit = "";

// フルーツ1個ずつ確認
for (int i = 0; i < 5; i++)
{
    // i番目のフルーツを求める
    string f = fruits[i];

    // もし、今回のフルーツ名の方が長ければ
    if (f.Length > maxFruit.Length)
    {
        // 最も長いフルーツを更新
        maxFruit = f;
    }
}

// 最大の長さのフルーツ名を表示
```

```
Debug.Log($"最も長いフルーツ名は: {maxFruit}");
```

2. 配列とfor文を使い、以下のスクリプトを完成してください（Unityでも OK）

```
// 4人分のスコア（整数型）の配列を準備してください
_____ scores = ___________;

//1人目のスコアは100点
__________= 100;

//2人目のスコアは80点
__________= 80;

//3人目のスコアは180点
__________= 180;

//4人目のスコアは20点
__________= 20;

// 同様に、4人分の名前（文字列型）の配列を準備してください
// それぞれの名前は「太郎」「さくら」「オオハラ」「Unity」

______ names = __________;
_______________;
_______________;
_______________;
_______________;

// for文を使って一人ずつのプレーヤーとスコアを表示する
for (________; _________; ________)
{
    Debug.Log($"名前: {________} → {_________}");
}
```

答え

```
// 4人分のスコア（整数型）の配列を準備してください
int[] scores = int[4];

//1人目のスコアは100点
scores[0] = 100;

//2人目のスコアは80点
scores[1] = 80;

//3人目のスコアは180点
scores[2] = 180;

//4人目のスコアは20点
scores[3] = 20;

// 同様に、4人分の名前（文字列型）の配列を準備してください
// それぞれの名前は「太郎」「さくら」「オオハラ」「Unity」

string[] names = string[4];
names[0] = "太郎";
names[1] = "さくら";
names[2] = "オオハラ";
names[3] = "Unity";

// for文を使って一人ずつのプレーヤーとスコアを表示する
for (int i = 0; i < 4; i++)
{
    Debug.Log($"名前: {names[i]} → {score[i]}");
}
```

???????? : List

配列を作るときに、事前にサイズを決めないといけないが、場合により、サイズが分からない可能性がある。例えば、同じゲームでも、プレーヤーによって拾うアイテムの数が変わるので、事前にアイテムの配列のサイズが分からない。

その場合は、サイズが自動的に変わる「リスト」を使うと便利。「List」は、要素を増やすほど、段々サイズが伸びる "配列 "（正しい名称は「コレクション」）である。

便利であるが、弱点もある：リストは配列と比べて処理が遅い。数十〜数百の要素以内だったら、あまり違いがないが、要素を増やすほど（数万程度）、 Listの方が明らかに遅い。

List???????

宣言	`List<型> 変数名;`
代入（作成）	`変数名 = new List<型>();`

例：

```
// 名前のリストを作成
List<string> names = new List<string>();

// リストに名前を追加する
names.Add("たろう");
names.Add("さくら");
names.Add("おおはら");

// 0番目の名前を表示（配列と同じ）
Debug.Log(names[0]); // 「たろう」を表示

// 要らなくなった場合、リストから削除できる
names.Remove("たろう");

// 0番目の名前を表示（配列と同じ）
Debug.Log(names[0]); // 「さくら」を表示
```

??

プレーヤーのスコアを表示するスクリプトを配列からリストに書き換える

答え

```
// スコア（整数型）のリストを準備してください
List<int> scores = new List<int>();
```

```
//1人目のスコアは100点
scores.Add(100);

//2人目のスコアは80点
scores.Add(80);

//3人目のスコアは180点
scores.Add(180);

//4人目のスコアは20点
scores.Add(20);

// 同様に、4人分の名前（文字列型）の配列を準備してください
// それぞれの名前は「太郎」「さくら」「オオハラ」「Unity」
List<string> names = new List<string>();
names.Add("太郎");
names.Add("さくら");
names.Add("オオハラ");
names.Add("Unity");

// for文を使って一人ずつのプレーヤーとスコアを表示する
for (int i = 0; i < 4; i++)
{
    Debug.Log($"名前: {names[i]} → {score[i]}");
}
```

???????????????? : foreach

「 for」を使って配列をループする方法を見てきた。変数を使って、インデックスを 0
から配列の最後まで一つずつ繰り返す。例えば:

```
string [] names = string[4] { "アリス", "ボブ", "チャーリー", "ジョン" };
for (int i= 0; i < 4; i++)
{
  string n = names[i]
  Debug.Log(n);
}
```

C#では、配列やリストをもっと簡単にループする方法がある。「`foreach`」（フォー・イーチ）を使うと、インデックス用の変数を用意しなくても、リストの各要素を順番に取り出すことができる。先ほどの例はこうなる：

```
string [] names = string[4] { "アリス", "ボブ", "チャーリー", "ジョン" };
foreach(string n in names)
{
  Debug.Log(n);
}
```

ただし、注意点が二つある。

1. `foreach`は常に要素を一つずつ順番に処理する。一部の要素をスキップしたい場合は、通常の`for`ループを使う必要がある。
2. `foreach`は読み取り専用である。リストや配列の要素を変更したい場合も、通常の`for`ループを使う必要がある。

??

最も長いフルーツの名前を求めるスクリプトを更新し、`for`から`foreach`に書き換える

答え

```
// フルーツの名前の配列
string [] fruits = ["lemon", "mikan", "orange", "grapefruit", "kiwi"];

// 最も長い文字列（結果）
string maxFruit = "";

// フルーツ1個ずつ確認
foreach(string f in fruits)
{
    // もし、今回のフルーツ名の方が長ければ
    if (f.Length > maxFruit.Length)
    {
        // 最も長いフルーツを更新
        maxFruit = f;
    }
}
```

```
// 最大の長さのフルーツ名を表示
Debug.Log($"最も長いフルーツ名は: {maxFruit}");
```

????

????

C#???????

```
int Sum(int a, int b)
{
    return a + b;
}
```

説明:

2つの整数（　　aと b）を足して、その結果を整数として返す

```
int Max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

答え

2つの整数（　　aと b）の最も大きい数値（最大値）を返す

```
bool CheckHasItem(string[] itemList, string item)
{
    for (int i = 0; i < itemList.Length; i++)
    {
        if (itemList[i] == item)
            return true;
    }
    return false;
```

```
}
```

答え

「itemList」配列の中に、「item」という文字列が入っているかどうかを確認し、あったら「true」を返し、なければ、「false」を返す。つまり、アイテム存在しているかどうかを確認。

Unity???

1. 配列で作った「最も長いフルーツ」のスクリプトを更新し、「`FindLongString`」というメソッドにしよう。配列を引数として渡し、最も長い文字列を返すようにてください。

使い方の例:

```
void Start()
{
    string [] fruits = {"lemon", "mikan", "orange", "grapefruit", "kiwi"};

    // このメソッドを作ってください
    string maxFruit = FindLongString(fruits);□

    // grapefruitを表示されるはず
    Debug.Log($"最も長いフルーツ名は: {maxFruit}");
}

// ここからメソッドを作る...
```

答え

```
// 最も長い文字列を探索
string FindLongString (string [] fruits)
{
  □// 最も長い文字列
    string maxFruit = "";

    // フルーツ1個ずつ確認
```

```
    foreach(string f in fruits)
    {
        // もし、今回のフルーツ名の方が長ければ
        if (f.Length > maxFruit.Length)
        {
            // 最も長いフルーツを更新
            maxFruit = f;
        }
    }

  □// 結果を返す
    return maxFruit;
}
```

---

2. 「じゃんけん」ゲーム結果の判定メソッド「`CheckJanken`」を作成してみてください。

前提条件:

- グーは「0」、チョキは「1」、パーは「2」として表現する
- グー、チョキ、パー（0, 1, 2）以外の数値入れたら、エラーを返す
- プレーヤーは2人（AとB）
- 判定の結果は: Aの勝ち: 1、Bの勝ち: 2、ひきわけ: 0、エラー: -1
- メソッド名: `CheckJanken`
    - 引数: 各プレーヤーの選択肢
    - 戻り値: 判定の結果

使い方の例1（Aの勝ち）

```
void Start()
{
  int playerA = 1; // チョキ
  int playerB = 2; // パー
  int result = CheckJanken(playerA, playerB);□// 1を返す（Aの勝ち）
  Debug.Log (result)
}
```

使い方の例2（Bの勝ち）

```
void Start()
{
```

```
  int playerA = 0; // グー
  int playerB = 2; // パー
  int result = CheckJanken(playerA, playerB);□// 2を返す（Bの勝ち）
  Debug.Log(result);
}
```

使い方の例3（あいこ）

```
void Start()
{
  int playerA = 2; // パー
  int playerB = 2; // パー
  int result = CheckJanken(playerA, playerB);□// 0を返す（あいこ）
  Debug.Log(result);
}
```

使い方の例4（エラー）

```
void Start()
{
  int playerA = 4; // ???
  int playerB = 2; // パー
  int result = CheckJanken(playerA, playerB);□// -1を返す（エラー）
  Debug.Log(result);
}
```

答え

```
// じゃんけんの結果の判定を求める
// playerA: プレーヤーAの選択肢（0:グ, 1:チョキ, 2:パー)
// playerB: プレーヤーBの選択肢（0:グ, 1:チョキ, 2:パー)
// 戻り値:
//   -1: エラー（無効な選択肢)
//    0: あいこ
//    1: プレーヤーAの勝ち
//    2: プレーヤーAの勝ち
int CheckJanken (int playerA, int playerB)
{
    // エラーを確認（プレーヤーA）
```

```
    if (playerA < 0 || playerA > 2)
    {
        return -1;
    }
  // エラーを確認（プレーヤーB）
    if (playerB < 0 || playerB > 2)
    {
        return -1;
    }

    // あいこ
    if (playerA == playerB)
    {
        return 0;
    }

    // プレーヤーAの勝ち
    if ((playerA == 0 && playerB == 1) || (playerA == 1 && playerB == 2) || (playerA == 2
&& playerB == 0))
    {
        return 1;
    }

    // プレーヤーBの勝ちしかのこらない!
    return 2;
}
```

???

C#?????? vs MonoBehaviour

お気づきかもしれないが、通常の C#クラスと Unityの「`MonoBehaviour`」クラスにはいくつかの違いがある。基本的に、どちらのクラスもプロパティ（クラス変数）とメソッドを持てる。しかし、大きな違いもある。

1. 通常の C#クラスは Unityエディターでは使用できない。GameObjectにアタッチできるのは `MonoBehaviour` のみである。
2. 通常の C#クラスは `new` キーワードを使ってインスタンス化するが、`MonoBehaviour` は直接インスタンス化できない。

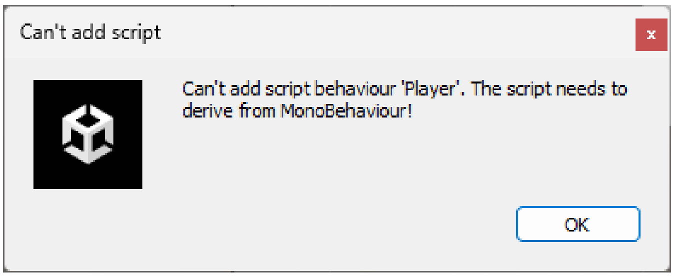
<table>
<tr><th>通常の C#クラス</th><th>MonoBehaviour</th></tr>
<tr><td>一般 C# のものであり、Unity 以外でも使える</td><td>Unity専用のキーワードである（Unity以外に使えない）</td></tr>
<tr><td>「`: MonoBehaviour`」がつかない

```
class Player
{
  // メンバー変数とメソッド
}
```
</td><td>「`: MonoBehaviour`」がつく

```
class Enemy : MonoBehaviour
{
  // メンバー変数とメソッド
}
```
</td></tr>
<tr><td>`new` をつかい、クラスからインスタンスを作成

```
// クラスから具体的に使える「p1」のプレーヤーを作成
Player p1 = new Player();
```
</td><td>`new` を使えない

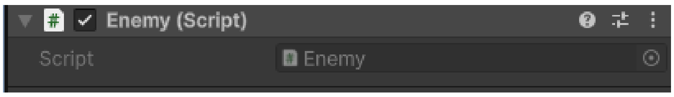
```
Enemy e = new Enemy(); // エラーが発生する
```
</td></tr>
<tr><td>Unityで GameObjectにアタッチ不可



Can't add script

Can't add script behaviour 'Player'. The script needs to derive from MonoBehaviour!

</td><td>Unityで GameObjectにアタッチ可能



</td></tr>
</table>

??????

では、どちらをいつ使うべきか? 基本的な目安として、GameObjectに機能を追加したい場合は`MonoBehaviour`を使う。例えば、プレイヤーの入力に基づいてGameObjectを制御したい場合や、画面上のスコアを更新したい場合は、その処理に専念する`MonoBehaviour`を作ればよい。

ゲーム内で通常表示されない内部処理にはC#クラスを使うパターンはよくある。例えば、プレイヤーのインベントリを管理するC#クラスや、ファイルのセーブ・ロードを制御するC#クラスを作ることができる。

???????

基本的な目安として、データを保護しクラスの外部から誤って値を変更しないよう、クラスのメンバー変数は常に「`private`」に設定する。

クラスの外部から値にアクセスする必要がある場合は、`public`メソッドまたは`public`ゲッター（下記の例を参照）を使うことができる。

`MonoBehaviour`の場合、Unityがエディターで変数を変更できるようにする必要がある。これは変数を`private`のままにし、変数に`[SerializeField]`を追加することで実現できる。

- やりたいこと________: Unityにある数値を公開し、エディター（Inspector）で編集できるようにしたい
  例: 車の速度

<table>
<tr><th>NG</th><th>正しい方法</th></tr>
<tr>
<td>上記の例と同様に、Unity
に限らず、どこからも編集できる変数なので、大変危ない!
<pre>
class Car : MonoBehaviour
{
  □// 車の速度をUnityから編集したいため、publicにした!
    public int Speed;
}
</pre>
</td>
<td><code>private</code>にし、<code>[SerializeField]</code>をつければ良い。
<pre>
class Car : MonoBehaviour
{
  □// UnityのInspectorから変えることができるが、
    // 他のクラスからアクセス不可
    [SerializeField]
    private int speed;
}
</pre>
</td>
</tr>
</table>

- やりたいこと________: ある数値を別のクラスで読みたい。
  例: プレーヤーを持っているお金

NG	正しい方法

public（公開）にすると、Player以外からアクセスでき、誤って数値を消したり可能性があるので､「public」を避けよう。

```
class Player
{
  // これを避けましょう!
    public int Money;
}
```

1. まず、メンバー変数を非公開にし（ `private` ）、メソッドで取得。

```
class Player
{
  // privateで変数を保護
    private int money;

  // メソッドで数値を返し、外から代入できない
    public int GetMoney()
    {
        return money;
    }
}
```

2. もう少し賢い方法は、「ゲッター」を使うこと:

```
class Player
{
  // Moneyというメンバー変数は:
    //「get」（取得）は公開である
    //「set」（代入）は非公開である
    public int Money { get; private set; }
}
```

- やりたいこと_________: ある数値を別のクラスで更新した。
  例: プレーヤーの体力。

NG	正しい方法

この場合は、攻撃や回復アイテムで現在の体力の数値を別のクラスから変えたい場合もあると思い、「public」にした。

```
class PlayerHP : MonoBehaviour
{
    // 最大体力はUnityで調整できるようにする
    [SerializeField]
    private int maxHP;

  □// 当然、攻撃などで、現在の体力を外からでも変えたい
    public int nowHP;
}
```

ただ、ここは大きなお落とし穴がある。次の例を考えてください
:

・最大耐力 `maxHP` は 120 に設定にした
・現在の体力 `nowHP` は 50 である
・回復アイテムを使用し、 +100を回復

とすると、 `nowHP + 100 = 150`
であり、最大体力を超えてしまう!

同様に 80 の攻撃を受け取るとしたら: `nowHP - 80 = -30`
マイナス体力になってしまった!

1. まず、メンバー変数を非公開にし（ `private` ）、代入の取得のメソッドを作成。

```
class PlayerHP : MonoBehaviour
{
    // 最大体力はUnityで調整できるようにする
    [SerializeField]
    private int maxHP;

  □// 体力を保護する
    private int nowHP;

    // 読み込む専用
    public int GetHP()
    {
        return nowHP;
    }

    // 書き込む専用
    public int SetHP(int value)
    {
        // 数値の制限を確認
        // Mathf.Min は 2つの引数の最小値を返す
        nowHP = Mathf.Min(value, maxHP);

        // Mathf.Max は 2つの引数の最大値を返す
        nowHP = Mathf.Max(value, 0);
    }
}
```

これで、 SetHPでは、体力が必ず `0 ~ maxHP` の間になることを保証できる。

2. もう少し賢い方法は、「ゲッター」と「セッター」を使うこと:

```
class PlayerHP : MonoBehaviour
{
    // 最大体力はUnityで調整できるようにする
    [SerializeField]
    private int maxHP;

  □// 非公開で中身を守る
    private int nowHP;
```

????

単純なインベントリを作成してみましょう:

- アイテムを表すクラスを作る（ファイル名: `Item`）
    - アイテム名を代入と取得できるようにする
    - 数も代入と取得できるようにする
    - 使用例

```
Item coin = new Item();
coin.Name = "ｺｲﾝ";
coin.Count = 20;
```

答え

```
// 何かのアイテムを表すクラス
class Item
{
    // アイテム名（非公開）
    private string name;

    // アイテムの数（非公開）
    private int count;

    // アイテム名（公開）
    public string Name {
        get { return name }
        set { name = value; }
    }

    // アイテムの数（公開）
    public string Count {
        get { return count }

        // マイナスアイテムは無意味なので、必ず0以上にする
        set { name = Mathf.Max(0, count); }
    }
}
```

- アイテムを管理する「`Inventory`」クラスを作成（ファイル名: `Inventory`）
    - アイテムを何個も追加できるようにする（ヒント: 自由に伸びる配列！）
    - 中身のすべてを確認できるようにする
    - 使用例

```
Inventory inv = new Inventory();
inv.Add(coin); // 先作ったコインを追加
inv.Print();   // Debug.Logで中身を表示する:「コイン: 20」
```

答え

```
// アイテムを管理する
class Inventory
{
    // アイテムのリスト
    private List<Item> items = new List<Item>();

    // 追加メソッド
    public void Add(Item item)
    {
        items.Add(item);
    }

    // 表示メソッド
    public void Print()
    {
        foreach(Item i in items)
        {
            Debug.Log($"{i.Name} : {i.Count}");
        }
    }
}
```

- テスト用のスクリプトを作成（ファイル名: `RunTest`）
    - コイン 20個を作成し、インベントリに追加
    - ポーション2個を作成し、インベントリに追加
    - ナイフ1個を作成し、インベントリに追加
    - インベントリの中身を表示

答え

```
// 確認用のスクリプト
class RunTest : MonoBehaviour
{
    void Start()
    {
        Item coin = new Item();
        coin.Name = "ｺｲﾝ";
        coin.Count = 20;

        Item potion = new Item();
        potion.Name = "ﾎﾟｰｼｮﾝ";
        potion.Count = 2;

        Item knife = new Item();
        knife.Name = "ﾅｲﾌ";
        knife.Count = 1;

        Inventory inv = new Inventory();
        inv.Add(coin);
        inv.Add(potion);
        inv.Add(knife);

        inv.Print();
    }
}
```

????

変数、条件分岐、配列（またはリスト）、列挙、メソッドとクラスを使い、以下のスクリプト作りましょう

?????????

参考: ポーカー・ハンドの一覧

??

手札にある5枚のカードを確認し、評価し、Debug.Logで判定を表示。

例: A♥ A♦ 10♣ 7♦ 3♠ - ワンペア

を表示するのは目標である

??

かなり大きな課題なので、わかりやすい「ミニ課題」に分割しましょう!

???

スートは「ハート（Heart）、ダイヤ（Diamond）、クラブ（Club）、スペード（Spade）」のことである。固定である値なので、列挙として表現できるので、「`CardSuit`」の列挙を作る:

CardSuit列挙（答え）

```
// カードのスート（列挙）を宣言
public enum CardSuit
{
□Heart,
□Spade,
□Diamond,
□Club,
}
```

???

カードはスートと数値を持っているし、最後に表示しないといけないので、カードから文字列に変換できる

メソッドがほしい。

※ ポイント：数値は1～13まで確保しましょう。

Cardクラス　　**-** メンバー変数（答え）

```
// MonoBehaviour（Unityのクラス）ではないので
public class Card
{
□//カードの数字
□private int number;

□//カードのスート
□private CardSuit suit;

□//カードの数字
□public int Number
□{
□□get { return number; }
□□set
□□{
□□□// Mathf.Clamp□□□値と最大値の制限をかける
□□□number = Mathf.Clamp(value, 1, 13);
□□}
□}

□//カードのスート
□public CardSuit Suit
□{
□□get { return suit; }
□□set { suit = value; }
□}
}
```

??????

文字列にするとき「1，11、12、13」は「A、J、Q、K」にする。

```
// 文字列にするメソッド
public string ConvertToString()
```

```
{
}
```

Cardクラス - ConvertToString メソッド

```
public class Card //（続き）
{
    //文字列にするメソッド
    public string ConvertToString()
    {
        string result = "";

        /□字を文字列にする
        if (number >= 2 && number <= 10) // 2~10までの場合
        {
            result = $"{number}"; □□を文字列にする
        }
        else if (number == 1) /□A」の場合
        {
            result = "A";
        }
        else if (number == 11) /□J」の場合
        {
            result = "J";
        }
        else if (number == 12) /□Q」の場合
        {
            result = "Q";
        }
        else /□K」の場合
        {
            result = "K";
        }

        /□様も文字列にして、連結する
        switch (suit) // "suit"で切り替える
        {
            case CardSuit.Heart:
                result += "♥";ハ□□
```

```
□□□□break;
□□□case CardSuit.Diamond:
□□□□result += "♦";ダ□□
□□□□break;
□□□case CardSuit.Club:
□□□□result += "♣";ク□□
□□□□break;
□□□case CardSuit.Spade:
□□□□result += "♠";ス□□ド
□□□□break;
□□}

□□return result;
□}
}
```

???

ハンドはカード5枚のことである。まず、メンバー変数を考えましょう。必ず 5
枚になるので、伸びるリスト必要なく、ただの配列で十分。

Handクラス **-** メンバー変数（答え）

```
// 手札を表すクラス（C#クラス）
public class Hand
{
    // カードの配列
    Card[] cards = new Card[5]; // 5枚
}
```

また、それぞれのカード（1枚目、2枚目、3枚目、4枚目、5枚目）の設定できるメソッドもほしい。

```
// カードを設定
// i: 位置（0~4）
// suit: スート
// number: 数値
public void SetCard(int i, CardSuit suit, int number)
{
}
```

Handクラス - SetCardメソッド

```
public class Hand // (続き)
{
□//カードを設定
  □// i: 位置 (0~4)
    // suit: スート
    // number: 数値
□public void SetCard(int i, CardSuit suit, int number)
□{
□□//□囲を確認
□□if (i < 0 || i > 5)
□□{
□□□Debug.Log(□"カードの位置!");
□□□return; //□もしない
□}□□

□□//□しいカードを作る
□Card card = new Card();
□□card.Suit = suit;
□□card.Number = number;
□□
□□//□定
□cards[i] = card;
□}
}
```

?????

つぎ、それぞれのハンドの役を評価しましょう。個別に対応すれば、楽になるでしょう。

??????

まず、「同じ数値のカード何枚あるのか」はよく現れるパターンであるので、メソッドにしましょう:

```
// 同じカードの枚数を数える
// number: 数えたい数値 (1~13)
private int CountSameNumber(int number)
{
}
```

Handクラス - **CountSameNumber** メソッド

```
// 同じカードの枚数を数える
// number: 数えたい数値（1~13）
private int CountSameNumber(int number)
{
    // 範囲を確認
    if (number < 1 || number > 13)
    {
        Debug.Log("無効なカードの数値!");
        return 0; // なにもしない
    }

    int count = 0;
    foreach (Card c in cards)
    {
        if (c.Number == number)
            count++;
    }

    return count;
}
```

????

同じ数値のカードが2枚ある? - `CountSameNumber` を活用!

```
// ワンペアであるか?
private bool IsOnePair()
{
}
```

Handクラス - **IsOnePair** メソッド

```
// ワンペアであるか?
private bool IsOnePair()
{
    // いずれかの2枚のカードは同じ数値だったら、ワンペアになる
```

```
    for (int i = 1; i <= 13; i++)
    {
        if (CountSameCard(i) == 2)
            return true;
    }
    return false;
}
```

????

ワンペアと似ているが、1つ目のペアの数値は2つ目のペアと異なることが前提条件であるので、ご注意ください!

```
// ワンペアであるか?
private bool IsTwoPair()
{
}
```

Handクラス - IsTwoPair メソッド

```
// ツーペアであるか?
private bool IsTwoPair()
{
    // ワンペアと似ているが、2つのペアは異なるカードじゃないとだめ
    int match = 0;  // 1つ目のペアの数値
    for (int i = 1; i <= 13; i++)
    {
        if (CountSameCard(i) == 2)
            match = i;
    }

    // なにも一致しなかった
    if (match == 0)
        return false;

    // もう一つのペアがないか確認する
    for (int i = 1; i <= 13; i++)
    {
```

```
        // 2枚がある「かつ」前のペアと異なる
        if (CountSameCard(i) == 2□&& i != match)
            return true;
    }

    return false;
}□
```

?????????????

ワンペアと似ているが、2枚ではなく、3枚を確認すべき。

```
// スリー・オブ・ア・カインドであるか?
private bool IsThreeOfAKind()
{
}
```

Handクラス **- IsThreeOfAKind** メソッド

```
// スリー・オブ・ア・カインドであるか?
private bool IsThreeOfAKind()
{
    // いずれかの3枚のカードは同じ数値だったら、スリー・オブ・ア・カインドになる
    for (int i = 1; i <= 13; i++)
    {
        if (CountSameCard(i) == 3)
            return 3;

    return false;
}□
```

?????

これは新しいやつ! すべての数値が連続であることが確認すべき。

ヒント:

- まず、最も数値が小さいカードを求める
- ストレートであれば、以下は事実である:
    - 最も小さいカード +0 → 1枚

  - 最も小さいカード +1 → 1枚
  - 最も小さいカード +2 → 1枚
  - 最も小さいカード +3 → 1枚
  - 最も小さいカード +4 → 1枚

```
// ストレートであるか?
private bool IsStraight()
{
}
```

Handクラス - IsStraight メソッド

```
// ストレートであるか?
private bool IsStraight()
{
    // 連続になっているのかを確認が必要
    // 例えば: 8-9-10-J(11)-Q(12)

    // まず、最低のカードを求めましょう
    int smallCard = 14; // Kよりも高いから始まる
    foreach (Card c in cards)
    {
        if (c.Number < smallCard)
            smallCard = c.Number; // 最低カードを更新
    }

    // では、5枚も連続であれば、各カード +1 は
    // 必ず1枚ずつがあるはず
    for (int i = smallCard; i < smallCard + 5; i++)
    {
        // 1つだけ!
        if (CountSameNumber(i) != 1)
            return false;
    }

    // 順番に1枚ずつがあった!
    return true;

    // メモ: A-K-Q-J-10 に対応していない
```

```
        // 興味がある学生、ご自分でお試しください。□□
    }
```

?????

すべてのカードが同じスートであるかを確認しましょう!

```
// フラッシュであるか?
private bool IsFlush()
{
}
```

Handクラス - IsFlush メソッド

```
// フラッシュであるか?
private bool IsFlush()
{
    // すべて同じスートでであればOK
    CardSuit suit = cards[0].Suit;

    // 残りのを確認
    foreach(Card c in cards)
    {
        if (c.Suit != suit)
            return false;
    }

    return true;
}
```

??????

これはツーペアと似ているが、2枚と2枚ではなく、2枚と3枚のパターンである

```
// フルハウスであるか?
private bool IsFullHouse()
{
}
```

Handクラス　　**- IsFullHouse** メソッド

```
// フルハウスであるか?
private bool IsFullHouse()
{
    // ツーペアと似ているが、枚数は２枚と３枚になる
    int match = 0;  // １つ目のペアの数値
    for (int i = 1; i <= 13; i++)
    {
        if (CountSameNumber(i) == 2)
            match = i;
    }

    // なにも一致しなかった
    if (match == 0)
        return false;

    // 残りの3枚を数える
    for (int i = 1; i <= 13; i++)
    {
        // ３枚がある「かつ」前のペアと異なる
        if (CountSameNumber(i) == 3　&& i != match)
            return true;
    }

    return false;
}
```

?????????????

もうわかるでしょう～

```
// フルハウスであるか?
private bool IsFullHouse()
{
}
```

Handクラス　　**- IsFourOfAKind** メソッド

```
// フォー・オブ・ア・カインドであるか?
private bool IsFourOfAKind()
{
    // いずれかの4枚のカードは同じ数値だったら、フォー・オブ・ア・カインドになる
    for (int i = 1; i <= 13; i++)
    {
        if (CountSameNumber(i) == 4)
            return true;

    return false;
}□
```

??????????

ストレート「かつ」フラッシュである

```
// ストレートフラッシュであるか?
private bool IsStraightFlush()
{
}
```

Handクラス - **IsStraightFlush** メソッド

```
// ストレートフラッシュであるか?
private bool IsStraightFlush()
{
    // ストレート「かつ」フラッシュなので…
    return IsStraight() && IsFlush();
}
```

?????

最後、強い →弱い順番で各判定を確認し、手札の中身とその結果をコンソールで表示しましょう:

```
// ハンドの判定し、表示する
public void CheckResult()
{
}
```

Handクラス - **CheckResult** メソッド

```
// ハンドの判定し、表示する
public void CheckResult()
{
    // まず、ハンドの文字列を構築
    string result = "";
    foreach (Card c in cards)
    {
        result += c.ConvertToString();
    }

    result += " - ";

    // 強い→弱い順番で評価
    if (IsStraightFlush())
    {
        result += "ストレートフラッシュ";
    }
    else if (IsFourOfAKind())
    {
        result += "フォー・オブ・ア・カインド";
    }
    else if (IsFullHouse())
    {
        result += "フルハウス";
    }
    else if (IsFlush())
    {
        result += "フラッシュ";
    }
    else if (IsStraight())
    {
        result += "ストレート";
    }
    else if (IsThreeOfAKind())
    {
        result += "スリー・オブ・ア・カインド";
```

```
    }□□
    else if (IsTwoPair())
    {
        result += "ツーペア";
    }
    else if (IsOnePair())
    {
        result += "ワンペア";
    }
    else
    {
        result += "ハイカード";
    }

    // 最後に表示
    Debug.Log(result);
}
```

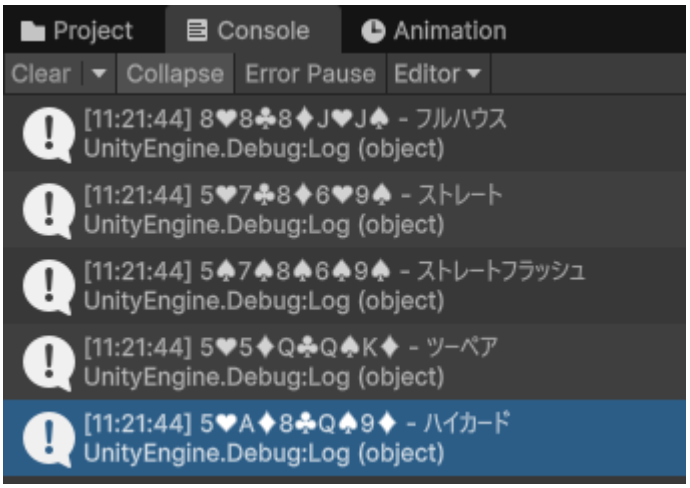
?????????

いくつかのハンドを確認してみてください (ダウンロード)

```
// TestPoker.cs

using UnityEngine;

public class TestPoker : MonoBehaviour
{
    void Start()
    {
        // いろんな組み合わせを試す
        Hand hand = new Hand();

        // フールハウス
        hand.SetCard(0, CardSuit.Heart, 8);
        hand.SetCard(1, CardSuit.Club, 8);
        hand.SetCard(2, CardSuit.Diamond, 8);
        hand.SetCard(3, CardSuit.Heart, 11);
```

```
        hand.SetCard(4, CardSuit.Spade, 11);
        hand.CheckResult();

        // ストレート
        hand.SetCard(0, CardSuit.Heart, 5);
        hand.SetCard(1, CardSuit.Club, 7);
        hand.SetCard(2, CardSuit.Diamond, 8);
        hand.SetCard(3, CardSuit.Heart, 6);
        hand.SetCard(4, CardSuit.Spade, 9);
        hand.CheckResult();

        // ストレートフラッシュ
        hand.SetCard(0, CardSuit.Spade, 5);
        hand.SetCard(1, CardSuit.Spade, 7);
        hand.SetCard(2, CardSuit.Spade, 8);
        hand.SetCard(3, CardSuit.Spade, 6);
        hand.SetCard(4, CardSuit.Spade, 9);
        hand.CheckResult();

        // ツーペア
        hand.SetCard(0, CardSuit.Heart, 5);
        hand.SetCard(1, CardSuit.Diamond, 5);
        hand.SetCard(2, CardSuit.Club, 12);
        hand.SetCard(3, CardSuit.Spade, 12);
        hand.SetCard(4, CardSuit.Diamond, 13);
        hand.CheckResult();

        // ハイカード
        hand.SetCard(0, CardSuit.Heart, 5);
        hand.SetCard(1, CardSuit.Diamond, 1);
        hand.SetCard(2, CardSuit.Club, 8);
        hand.SetCard(3, CardSuit.Spade, 12);
        hand.SetCard(4, CardSuit.Diamond, 9);
        hand.CheckResult();
    }
}
```

?????


Project
Console
Animation
Clear
Collapse
Error Pause
Editor
[11:21:44] 8♥8♣8♦J♥J♠ - フルハウス
UnityEngine.Debug:Log (object)
[11:21:44] 5♥7♣8♦6♥9♠ - ストレート
UnityEngine.Debug:Log (object)
[11:21:44] 5♠7♠8♠6♠9♠ - ストレートフラッシュ
UnityEngine.Debug:Log (object)
[11:21:44] 5♥5♦Q♣Q♠K♦ - ツーペア
UnityEngine.Debug:Log (object)
[11:21:44] 5♥A♦8♣Q♠9♦ - ハイカード
UnityEngine.Debug:Log (object)