C#????????event?

??

ここまで、C#で学んだ主な概念は

- メンバー変数：あるインスタンス（オブジェクト）のデータを保持するもの
  - 体力
  - アイテムの一覧
  - 速度
  - 色
  - など
- メソッド：あるインスタンス（オブジェクト）が「できる」こと。データを処理する「関数」
  - 走る
  - ジャンプ
  - 攻撃
  - フレーム毎の更新（Update）
  - 初期化（Start）
  - など
- クラス：インスタンスの「図形」であり、`new`（一般のC#）または`Instantiate`（Unityの`Monobehavior`/プレハブ）で具体化し、インスタンスになる。メンバー変数とメソッドのかたまりとも考えらる。
  - プレーヤー
  - 敵
  - 罠
  - 武器
  - 回復アイテム
  - など

ここで、新しいC#の機能「イベント」を学ぶ。イベントは「あるインスタンスが通知できること」と考えらる。次の例について考えてみてください。

格闘ゲームを開発中に、「体力」（`HitPoints`）を実装しようとする。ここで：

- 人間プレーヤーの体力がゼロになると、プレーヤーの負けになる（ゲームオーバー）
- CPUプレーヤーの体力がゼロになると、プレーヤーの勝ちになる（ゲームクリア）

体力は変わらないので、EnemyとPlayerのゲームオブジェクトは全く同じ体力を使うので、同じ「`HitPoints`」のスクリプトをアタッチする

```
using UnityEngine;

// 体力クラス
public class HitPoints : MonoBehaviour
{
    // 体力の値
    private int hp;

    private void Start()
    {
        hp = 100;
    }

    // 現在の体力を返す
    public int GetHP()
    {
        return hp;
    }

    // ダメージを受ける
    public void Hit(int damage)
    {
        hp = Mathf.Max(0, hp -damage);
    }
}
```

こうすると、ゲーム監督クラス「GameManager」では、どれの勝ちなのか確認したいので、Update で定期的に問い合わせるしかない。

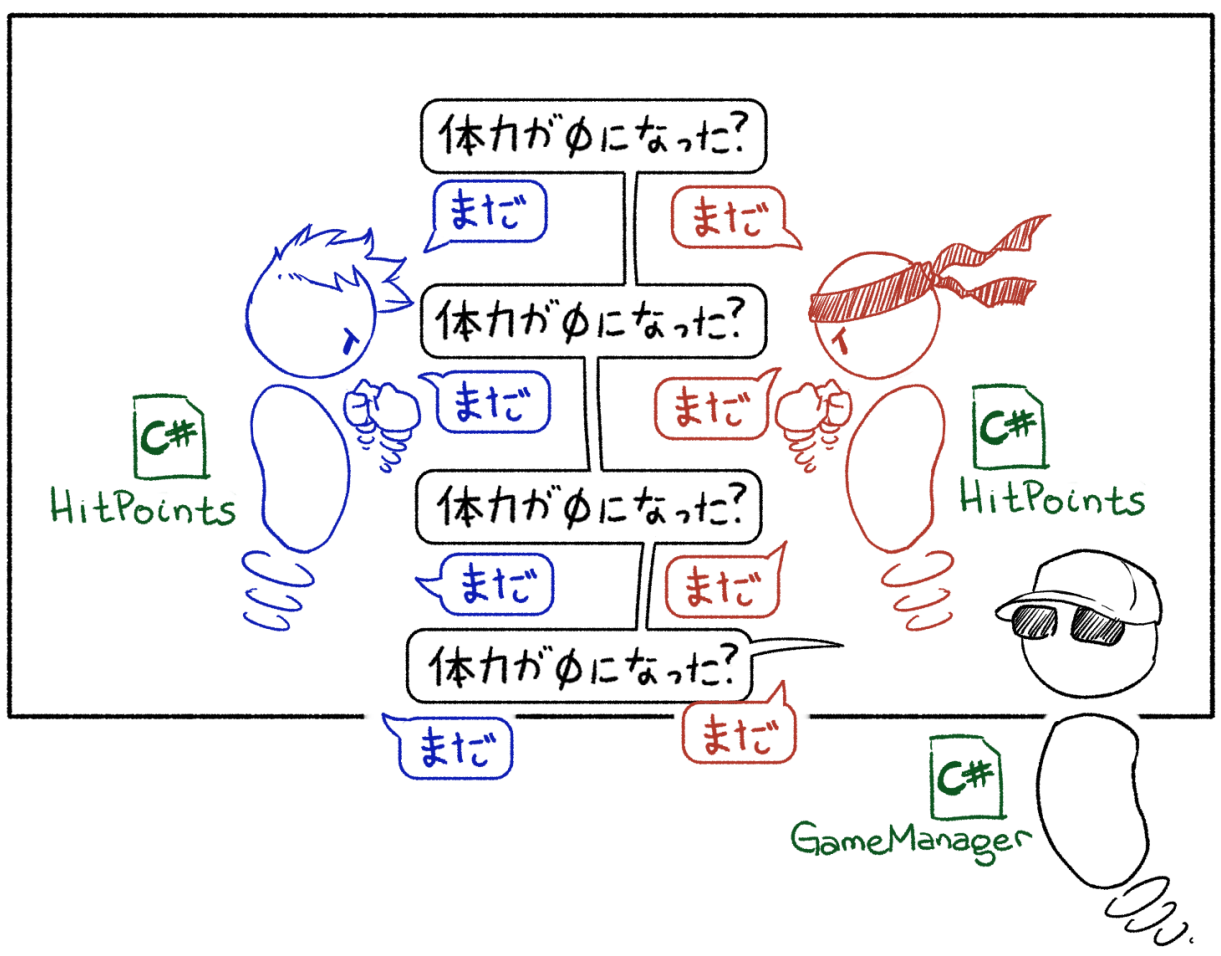
```
using UnityEngine;

// ゲーム監督
public class GameManager : MonoBehaviour
{
    // 人間プレーヤーの体力
    [SerializeField]
    private HitPoints playerHP;

    // CPUプレーヤーの体力
    [SerializeField]
```

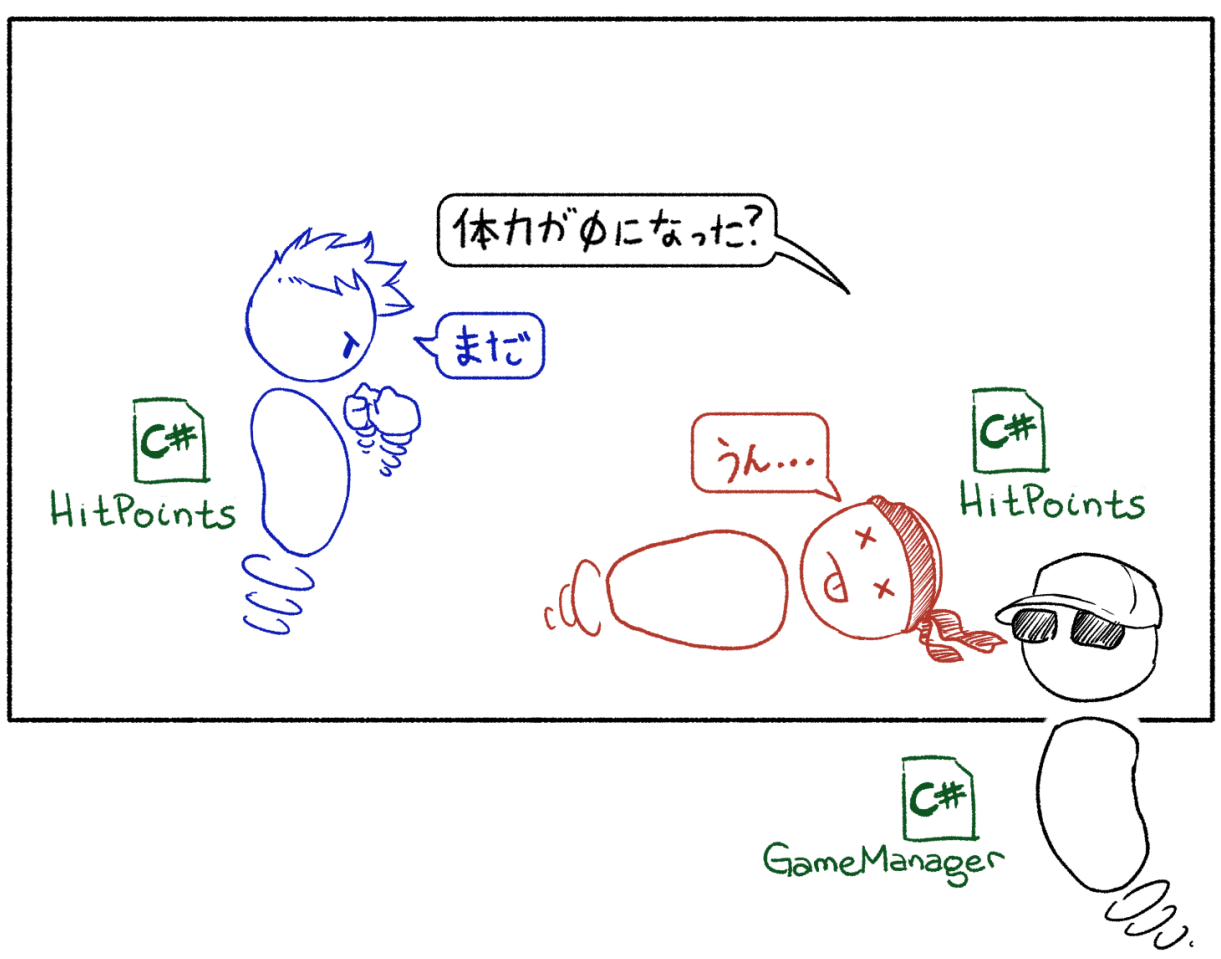
```
    private HitPoints cpuHP;

    private void Update()
    {
        // プレーヤーの負け
        if (playerHP.GetHP() == 0)
        {
            Debug.Log("YOU LOSE!");
        }

        // CPUの負け
        if (cpuHP.GetHP() == 0)
        {
            Debug.Log("YOU WIN!");
        }
    }
}
```

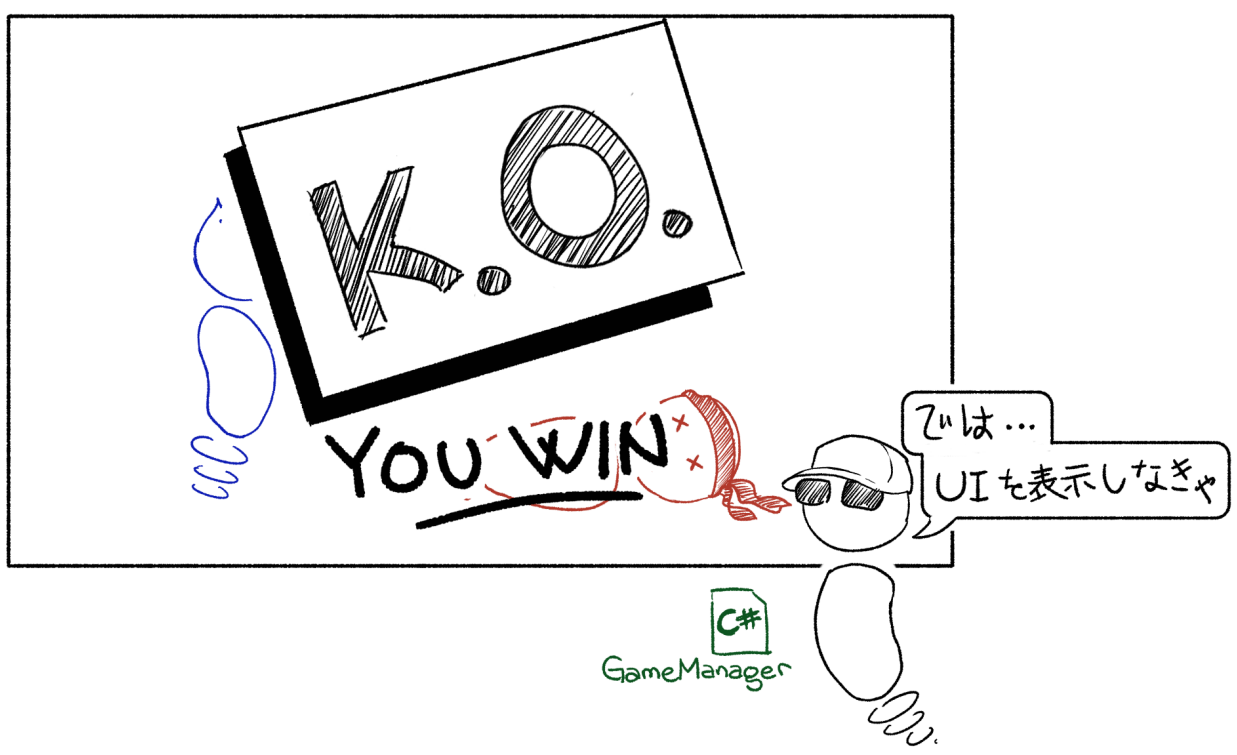
この作り方で問題なく、ゲームが動くが、体力が

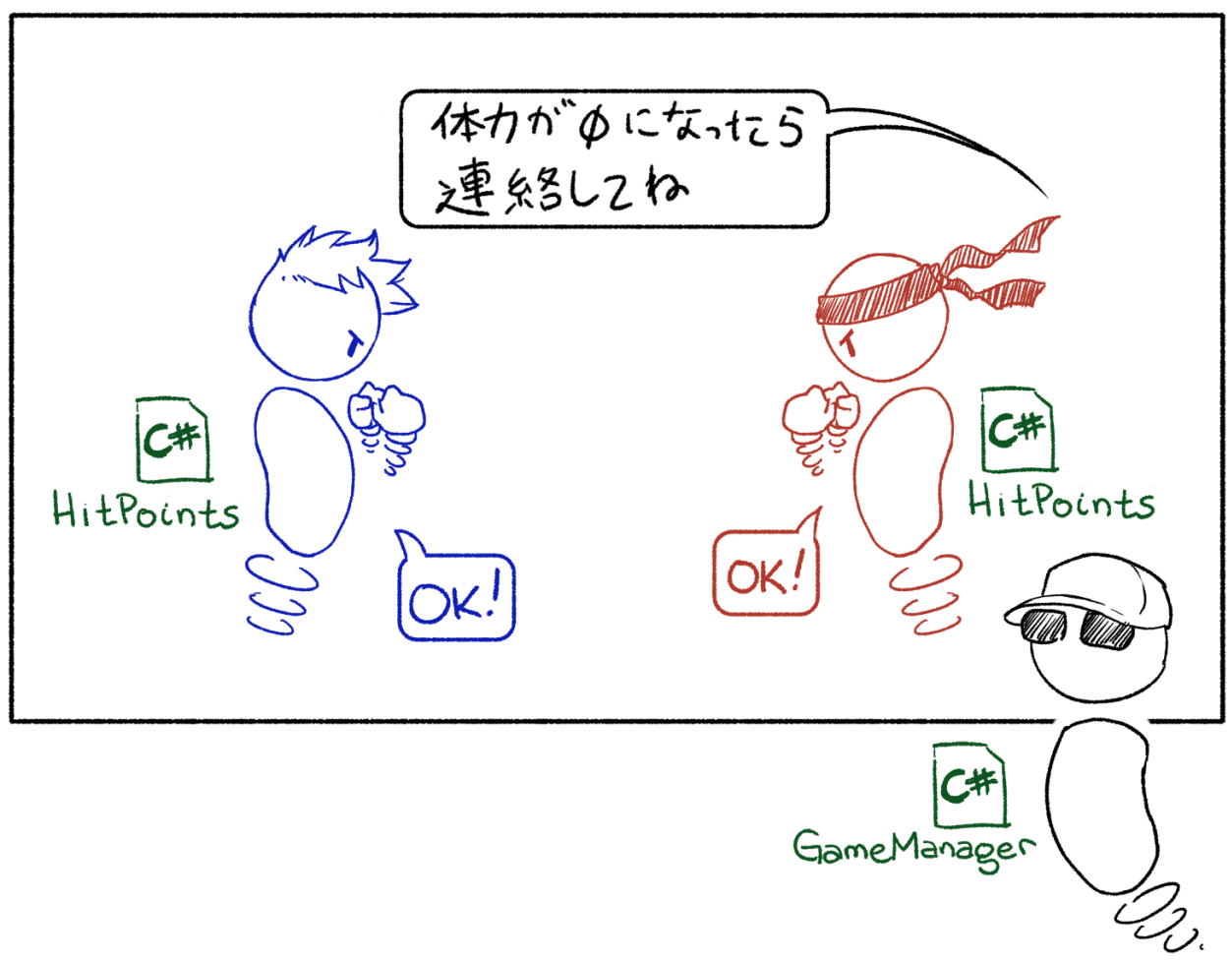
変わっていないのに、何度も問い合わせるのは無駄な処理　になる。逆に、体力が　変わった瞬間　に

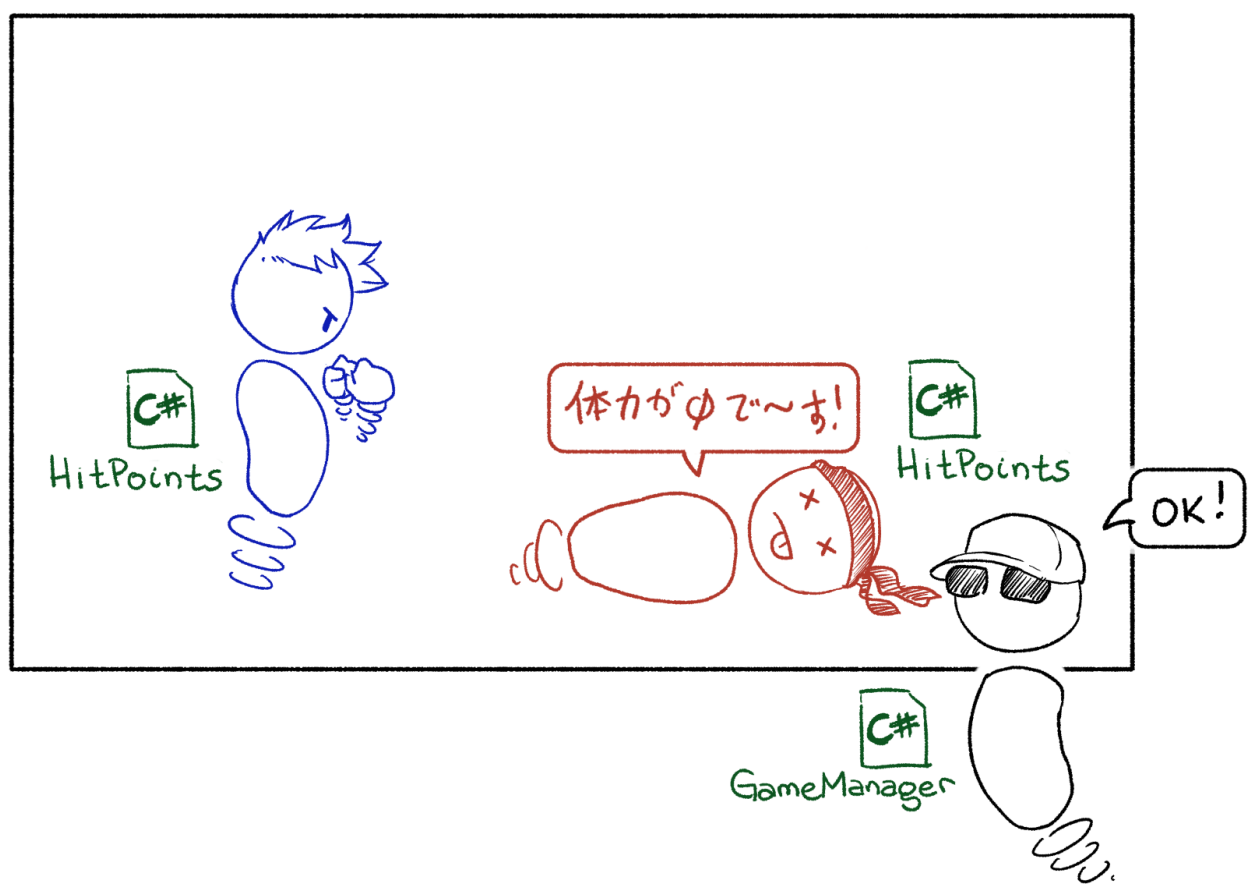
GameManagerに知らせ、　処理すれば効率が良い　！

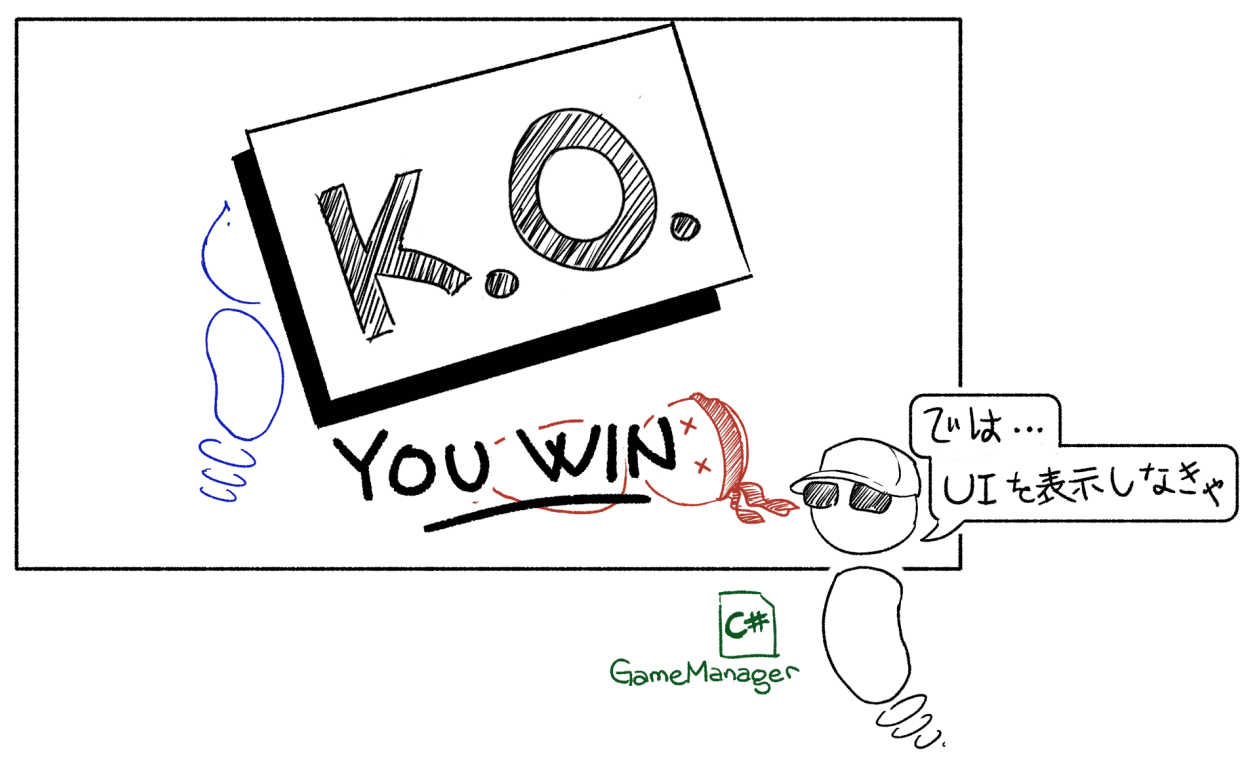












Updateで問い合わせる

event で連絡待ち

???

イベントを実装するには、2つのクラスが関連する: 「イベントが発生する側」と「イベントを聞き取る側」

??????????

イベントを発生するクラスは「通知ができる」ことを　　宣言
しないといけない。または、イベントの条件が満たしたら、イベントを　　発生 しなければならない。

??

クラスの先頭に「　　event」キーワードを使い、イベントを宣言する

書式（引数なし）＿＿＿＿＿＿

```
public event System.Action イベント名;
```

書式（引数あり）＿＿＿＿＿＿

```
public event System.Action<引数型1, 引数型2, ...> イベント名;
```

今回の格闘ゲームだと:

```
// 体力クラス
public class HitPoints : MonoBehaviour
{
    // 体力がゼロになった時に発生するイベント。
    // 引数はゼロになった体力コンポーネントの参照
    public event System.Action<HitPoints> ZeroHP;

    // 体力の値
    private int hp;

    // （後は同じ）
}
```

ここの例だと:

- 「`ZeroHP`」というイベントを宣言した。このイベントは体力がゼロになった瞬間に発生するべき
- 引数の型は「　　`HitPoints`
  」の参照である。これでどの体力コンポーネントゼロになったかを引数として渡すできる。

（※）引数は任意であり、必要なときに使ってください。不要なら「引数なし」の宣言方法を利用してください。

??

条件を満たしたら、イベントを発生しなければならない。これは、「 `Invoke` 」で行う。

書式（引数なし）_____

```
イベント名?.Invoke();
```

書式（引数あり）_____

```
イベント名?.Invoke(引数1, 引数2, ...);
```

今回の格闘ゲームだと:

```
// ダメージを受ける
public void Hit(int damage)
{
    hp = Mathf.Max(0, hp -damage);

    // 体力がゼロになったら、イベントを発生
    // 引数は「this」（自分の参照）
    if (hp == 0)
        ZeroHP?.Invoke(this);
}
```

ここの例だと:

- ダメージを喰ったら、体力がゼロになったのかを確認する
- もしゼロだったら、 `ZeroHp` のイベントを発生する。引数は自分（ `this` ）の参照

??????????

イベントを発生するのはただ半分であり、聞き取る側もある。ここで、イベントに 登録・登録解除
とそのイベントを 処理 するメソッドを作成しなければならない。

???????

まず、定めたイベントを処理するため、先に登録しなければならない。また、イベントが不要になった場合、必ず登録解除してください。

書式: 登録_____

```
オブジェクト名.イベント名 += イベントを処理するメソッド;
```

書式: 登録解除_____

```
オブジェクト名.イベント名 -= イベントを処理するメソッド;
```

??

処理は上記の「イベントを処理するメソッド」で行う。

引数はイベント宣言の引数を一致しなければならない 。

今回の格闘ゲームだと:

```
using UnityEngine;

// ゲーム監督
public class GameManager : MonoBehaviour
{
    // 人間プレーヤーの体力
    [SerializeField]
    private HitPoints playerHP;

    // CPUプレーヤーの体力
    [SerializeField]
    private HitPoints cpuHP;

    private void Start()
    {
        // 人間、またはCPUプレーヤーの体力がゼロになるイベントに登録
        playerHP.ZeroHP += OnZeroHP;
        cpuHP.ZeroHP += OnZeroHP;
    }

    private void OnDestroy()
    {
        // ゲーム監督クラス廃止のとき、イベントに登録解除する
        playerHP.ZeroHP -= OnZeroHP;
        cpuHP.ZeroHP -= OnZeroHP;
    }

    // イベントを処理
    // sender: ゼロになった体力コンポーネント
    private void OnZeroHP(HitPoints sender)
    {
        // プレーヤーの負け
```

```
        if (sender == playerHP)
        {
            Debug.Log("YOU LOSE!");
        }

        // CPUの負け
        if (sender == cpuHP)
        {
            Debug.Log("YOU WIN!");
        }
    }

    // Updateは要らなくなった
    private void Update()
    {
    }
}
```

もちろん、別々の処理メソッドを作れるので、以下の書き方も可能:

```
using UnityEngine;

// ゲーム監督
public class GameManager : MonoBehaviour
{
    // 人間プレーヤーの体力
    [SerializeField] private HitPoints playerHP;

    // CPUプレーヤーの体力
    [SerializeField] private HitPoints cpuHP;

    private void Start()
    {
        // 人間、またはCPUプレーヤーの体力がゼロになるイベントを登録
        playerHP.ZeroHP += OnPlayerZeroHP;
        cpuHP.ZeroHP += OnCPUZeroHP;
    }

    private void OnDestroy()
```

```
    {
        // ゲーム監督クラス廃止のとき、イベントの登録解除を行う
        playerHP.ZeroHP -= OnPlayerZeroHP;
        cpuHP.ZeroHP -= OnCPUZeroHP;
    }

    // イベントを処理（プレーヤーの負け）
    // sender: ゼロになった体力コンポーネント
    private void OnPlayerZeroHP(HitPoints sender)
    {
        Debug.Log("YOU LOSE!");
    }

    // イベントを処理（CPUの負け）
    // sender: ゼロになった体力コンポーネント
    private void OnCPUZeroHP(HitPoints sender)
    {
        Debug.Log("YOU WIN!");
    }

    // Updateは要らなくなった
    private void Update()
    {
    }
}
```

??

まず、 このプロジェクト をダウンロードし、 Unity に追加してください。このミニゲームでは、車が走行し、「ゴール」を通過すれば、 `Debug.Log` でメッセージが表示される。

ここで、 `event` を使用し、画面上でレースの結果を表示しましょう。

1. 発生する側: 車の通過を検知する「 CarDetector 」にイベントを宣言し、車を通過したらイベントを発生
2. 聞き取る側: UIManager で CarDetector のイベントに登録し、車を通過したら、表示する

CarDetector の答え

```
// □□□□□□□
public class CarDetector : MonoBehaviour
{
    // □□□□□□□□□□□□□□□□
    // □□□□□□□□□
    public event System.Action<RaceCar> CarDetected;

    private void OnTriggerEnter(Collider other)
    {
        // RaceCar□□□□□□□□□□□□□□□□□
        RaceCar raceCar = other.GetComponent<RaceCar>();
        if (raceCar□== null)
            return;

        // □□□□□□□□
        CarDetected?.Invoke(raceCar);
    }
}
```

UIManager □□□

```
// UI□□□□□
public class UIManager : MonoBehaviour
{
    // □□□□□□□□
    [SerializeField] private TextMeshProUGUI[] carPosition;

    // □□□□□□□□□□□□
    [SerializeField] private CarDetector detector;

    // □□□□
    private int position = 0;

    private void Start()
    {
        // □□□□□□□□□
        foreach (var item in carPosition)
            item.gameObject.SetActive(false);
```

```
        // イベントに登録
        detector.CarDetected += OnCarDetected;
    }

    private void OnDestroy()
    {
        // イベントに解除
        detector.CarDetected -= OnCarDetected;
    }

    // ゲームを再開
    public void RestartGame() =>
        SceneManager.LoadScene("Racer");

    // 車が通過したイベントを受け取る
    private void OnCarDetected(RaceCar raceCar)
    {
        ShowCar(raceCar, position);
        position++;
    }

    // 車を定めた位置に表示
    private void ShowCar(RaceCar raceCar, int index)
    {
        var item = carPosition[index];
        item.gameObject.SetActive(true);
        item.color = raceCar.Color;
        item.text = raceCar.name;
    }
}
```

もちろん、聞き取る側がクラス1つだけではなく、どのクラスでもイベントを受け取ることができる。紙吹雪のエフェクトを実装されているので、 Confetti クラスでも CarDetector のイベントに登録し、車を通過したら、車の色の紙吹雪を飛ばすようにしましょう。

Confetti の答え

```
// □□□□□□□□□□□□□□□
public class Confetti : MonoBehaviour
{
    // □□□□□□□□□□□□□□□
    [SerializeField] private ParticleSystem[] effects;

    // □□□□□
    [SerializeField] private CarDetector detector;

    private void Start()
    {
        // □□□□□□□□□
        detector.CarDetected += OnCarDetected;
    }

    private void OnDestroy()
    {
        // □□□□□
        detector.CarDetected -= OnCarDetected;
    }

    // □□□□□□□□□□□□
    private void OnCarDetected(RaceCar raceCar)
    {
        Play(raceCar.Color);
    }

    // □□□□□□□□□
    private void Play(Color color)
    {
        foreach (var effect in effects)
        {
            var main = effect.main;
            main.startColor = color;
            effect.Play();
        }
    }
}
```

Revision #5
Created 2026-06-08 02:29:23 UTC by Menendez Francisco
Updated 2026-06-15 08:56:00 UTC by Menendez Francisco