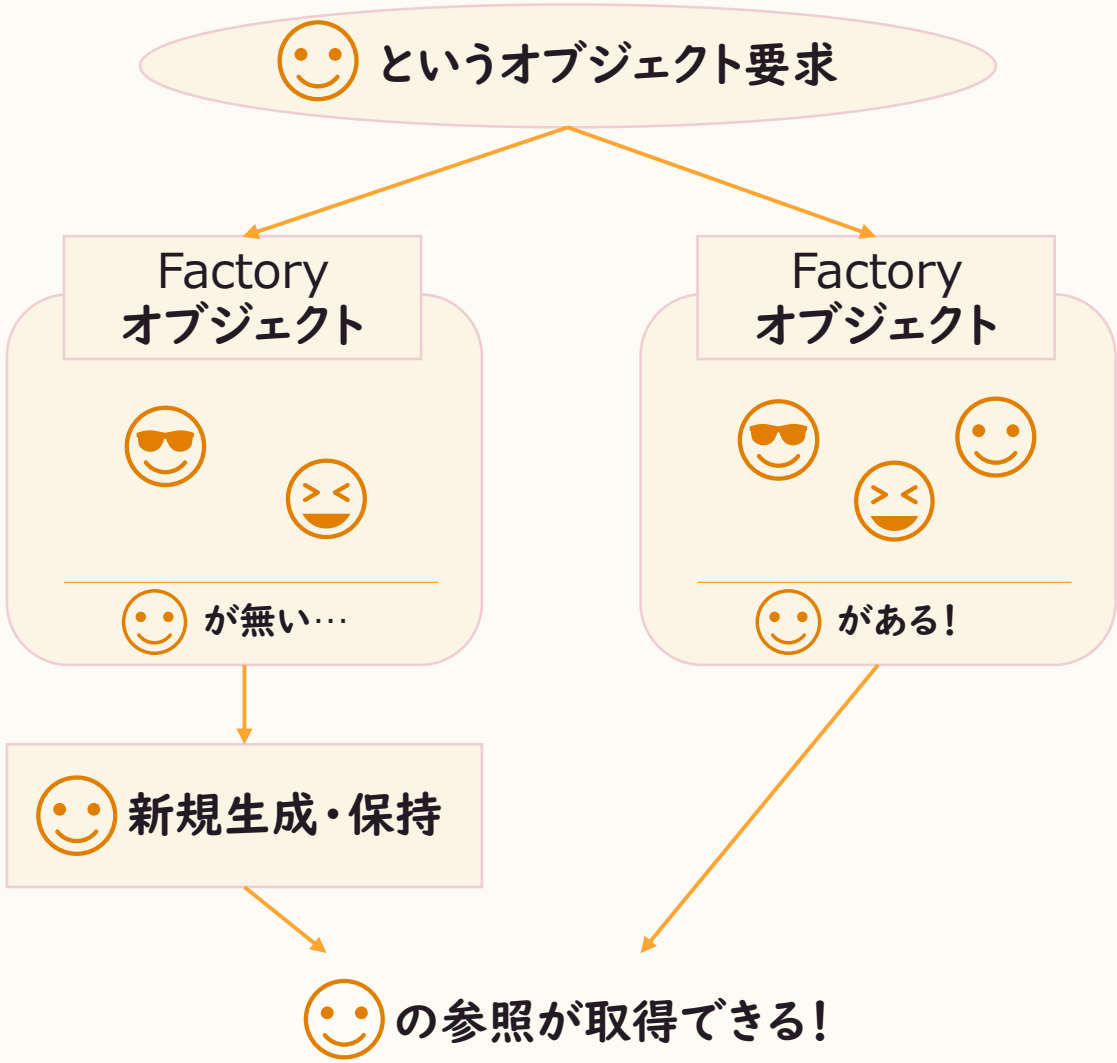
Flyweight

Seeeeee:D夏休み勉強会　Nanaka

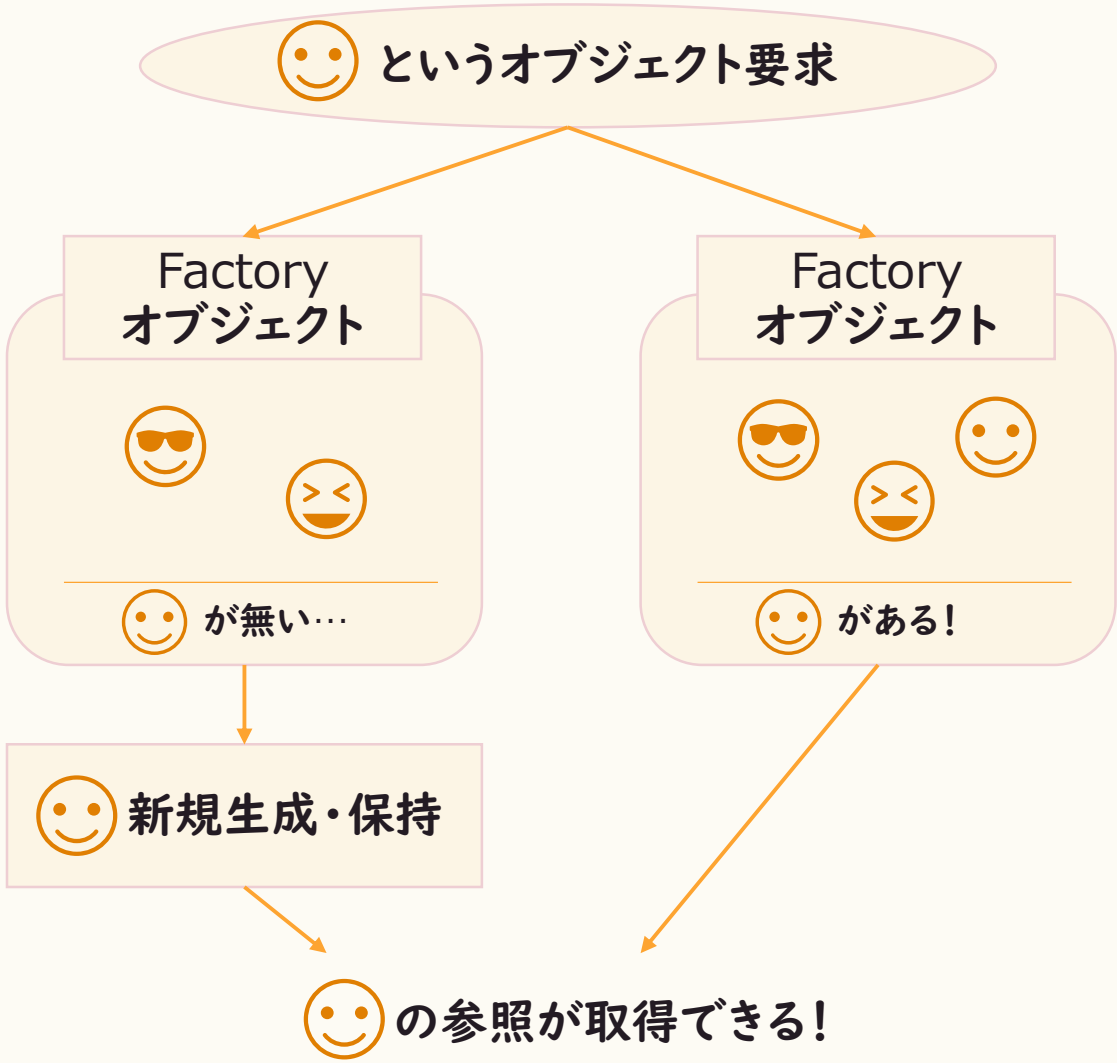
Flyweightパターンとは？

- 簡単に言うと、キャッシュを実現するための実装
- 一度生成したインスタンスを再利用することで、メモリの使用を抑えたり、リソースを軽く済ませる
- だから、「フライ級」を意味するFlyweightパターンと呼ばれる



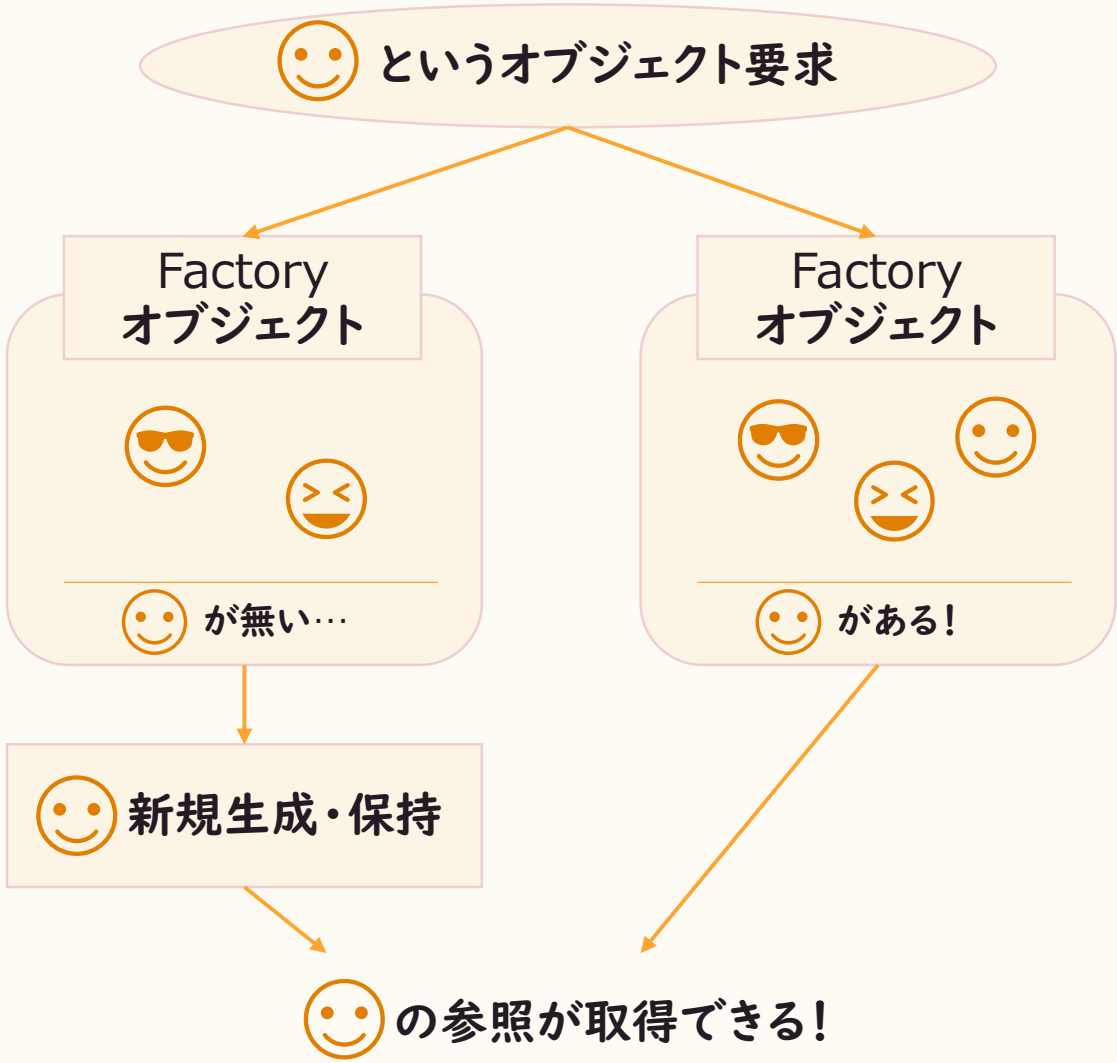
Flyweightパターンとは？

- 対象のインスタンスが生成されていない場合
  - 対象のインスタンスを新たに生成する
  - 生成したインスタンスをプールする
  - 生成されたインスタンスを返す
- 対象のインスタンスが既に生成されていた場合
  - 対象のインスタンスをプールから呼び出す
  - 対象のインスタンスを返す

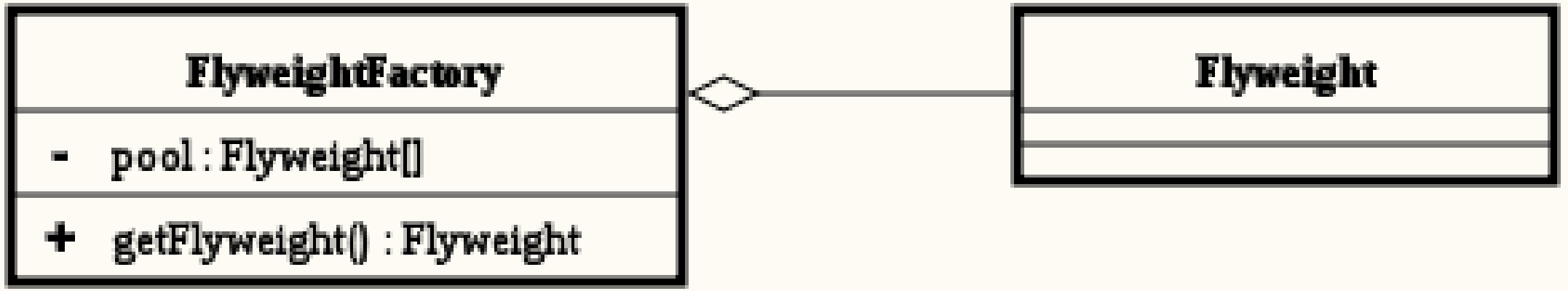


Flyweightパターンとは？

- Factoryオブジェクトでは、状況によって異なる処理が行われる
- でも、利用者が得る結果は同じ
- なので、利用者側は中身を意識しないで使うことができるのがこのパターンの特徴



Flyweightのクラス図



- 生成されたインスタンスを管理するFactoryオブジェクトはsingletonとして実装する
- Factoryオブジェクトの中に、生成済みインスタンスの一覧を持っておく

Javaの実装例

```
class StampFactory {
    Map<Character, Stamp> pool;
    StampFactory(){
        this.pool = new HashMap<Character, Stamp>();
    }
    Stamp get(char type){
        Stamp stamp = this.pool.get(type);
        if(stamp == null) {
            stamp = new Stamp(type);
            this.pool.put(type, stamp);
        }
        return stamp;
    }
}
```

Javaの実装例

```
class Stamp {
    char type;
    Stamp(char type){
        this.type = type;
    }
    void print(){
        System.out.print(this.type);
    }
}
```

Javaの実装例

```
public class FlyweightTest {
    public static void main(String[] args) {
        StampFactory factory = new StampFactory();
        List<Stamp> stamps = new ArrayList<Stamp>();
        stamps.add(factory.get('た'));
        stamps.add(factory.get('か'));
        stamps.add(factory.get('い'));
        stamps.add(factory.get('た'));
        stamps.add(factory.get('け'));
        stamps.add(factory.get('た'));
        stamps.add(factory.get('て'));
        stamps.add(factory.get('か'));
        stamps.add(factory.get('け'));
        stamps.add(factory.get('た'));
        for(Stamp s : stamps){
            s.print();
```

- １０回getメソッドを呼んでる
- だけど、インスタンスの生成は５回だけ
- やったぁ、軽いね！

C#で実装してみよう！

実装方針

- 実装例では、Factoryクラスを作ってたけど、要は生成済みインスタンスを管理するなにかしらがひとつあれば良い
- 静的メンバとしてStampクラスの中にDictionary(生成済みインスタンスの一覧)を作っちゃう
- 静的メンバは最初からシングルトンになっているので、あとはここを参照してインスタンスの生成を行えばOK

```
0 個の参照
public class Stamp
{
    char type;

    1 個の参照
    private Stamp(char type)
    {
        this.type = type;
    }

    0 個の参照
    public void Print()
    {
        Console.WriteLine(this.type);
    }

    static Dictionary<char, Stamp> cache = new Dictionary<char, Stamp>();

    3 個の参照
    public static Stamp Get(char type)
    {
        if (!cache.ContainsKey(type))
            cache.Add(type, new Stamp(type));
        return cache[type];
    }
}
```

静的メンバが上手く伝わらなかった時のためのURL
https://docs.microsoft.com/ja-jp/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

実装方針

- 静的ななんやかんやがある言語だとFactoryクラスを作らなくても実装できる！

```
0 個の参照
public class Stamp
{
    char type;

    1 個の参照
    private Stamp(char type)
    {
        this.type = type;
    }

    0 個の参照
    public void Print()
    {
        Console.WriteLine(this.type);
    }

    static Dictionary<char, Stamp> cache = new Dictionary<char, Stamp>();

    3 個の参照
    public static Stamp Get(char type)
    {
        if (!cache.ContainsKey(type))
            cache.Add(type, new Stamp(type));
        return cache[type];
    }
}
```

静的メンバが上手く伝わらなかった時のためのURL
https://docs.microsoft.com/ja-jp/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

実行結果を
確認してみよう！

まとめ

- Flyweightパターンはコスパの鬼らしいです
- 一番だいじなのは作ったインスタンスを再利用しようねってことだけ
- だから、汎用性がありそう

- 静的なんちゃらをもっと使いこなせるようになりたい