KMITL

ASSIGNMENT 2

- DESIGN PATTERNS VS ANTI PATTERNS
- SINGLETON PATTERNS
- PROTOTYPE PATTERNS
- OBJECT POOL PATTERNS
- FUNCTIONAL PROGRAMMING

DESIGN PATTERNS VS ANTI PATTERNS

REF : https://www.savtec.org/articles/coding/web-development-the-10-coding-antipatterns-you-must-avoid.html

DESIGN PATTERNS

Design Pattern เป็น blueprint สำหรับการออกแบบซอฟต์แวร์ เพื่อให้การเขียนโค้ดได้มาตรฐานเดียวกัน โดย design patternนี้กว่าจะเกิดขึ้นมาแต่ละแบบก็มาจากการทดลองซ้ำๆจนหารูปแบบหรือวิธีที่ดีที่สุดในการแก้ปัญหาแต่ละอย่าง และด้วยความที่มันเป็นรูปแบบที่ใช้แก้ปัญหาต่างๆ เราจึงไม่สามารถที่จะก็อปปี้เอามาใช้งานทันที แต่ต้องทำความเข้าใจของ design pattern นั้นๆและนำไปปรับใช้กับโค้ดของเราเอง

Design Pattern นั้น แบ่งออกได้เป็น 3 กลุ่ม

Creational patterns

เป็นกลุ่มที่ไว้ใช้สร้าง object ในรูปแบบต่างๆ ให้มีความยืดหยุ่น(flexible) และนำโค้ดมาใช้ซ้ำ(reuse)ได้

Structural patterns

กลุ่มนี้จะเป็นวิธีการนำ object และ class มาใช้งานร่วมกัน สร้างเป็นโครงสร้างที่มีความซับซ้อนยิ่งขึ้น โดยที่ยังมีความยืดหยุ่นและทำงานได้อย่างมีประสิทธิภาพ

Behavioral patterns

กลุ่มสุดท้ายนี้เป็นวิธีการออกแบบการติดต่อกันระกว่าง object ให้มีความยืดหยุ่นและสามารถติดต่อกันกันได้อย่างไม่มีปัญหา

ANTI PATTERNS

นั้นมักถูกเรียกเช่นกัน รูปแบบของความล้มเหลว. ข่าวดีก็คือว่ามันเป็น เป็นไปได้ที่จะรับรู้และหลีกเลี่ยง

ตัวอย่าง Antipatterns

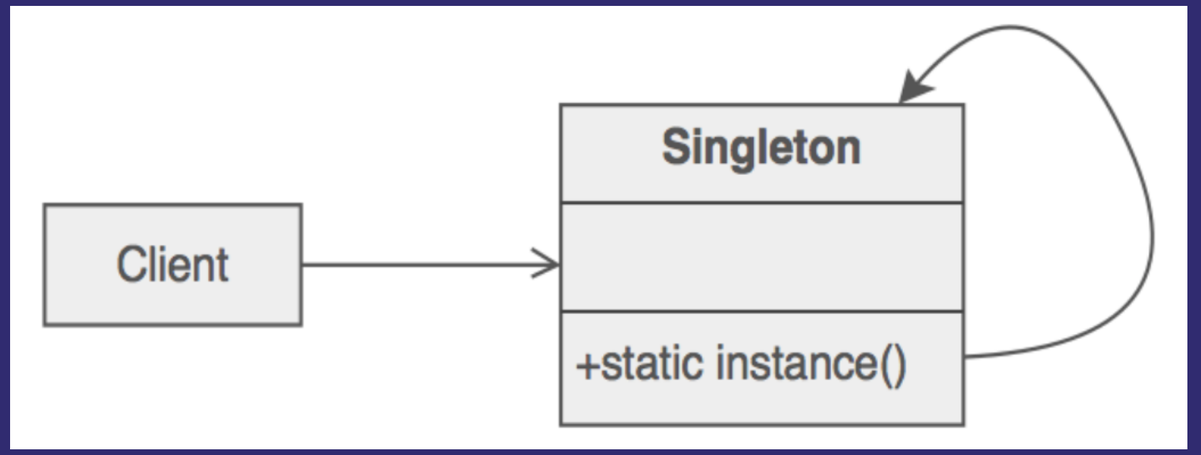
สปาเก็ตตี้ เป็นการเข้ารหัสที่มีชื่อเสียงที่สุด มันอธิบาย แอปพลิเคชันที่ยากต่อการตรวจแก้จุดบกพร่องหรือปรับเปลี่ยนเนื่องจากไม่มีสถาปัตยกรรมที่เหมาะสม

ผลลัพธ์ของการออกแบบซอฟต์แวร์ที่ไม่ดีคือพวงของรหัสที่คล้ายกันในโครงสร้างกับชาม สปาเก็ตตี้ เช่น พันกันและซับซ้อน

การอ่านรหัสสปาเก็ตตี้นั้นต่ำมากและมักจะเป็นภารกิจที่แทบเป็นไปไม่ได้ที่จะเข้าใจว่ามันทำงานอย่างไร

SINGLETON PATTERNS

เป็นรูปแบบการออกแบบซอฟต์แวร์ที่จำกัดจำนวนของ Object ที่ถูกสร้างขึ้นในระบบ ซึ่งจะเป็นประโยชน์เมื่อระบบต้องการจะมี Object นั้นเพียงตัวเดียวเพื่อป้องกันไม่ให้เกิดการทำงานซ้ำซ้อนกันเช่น class สำหรับการเก็บข้อมูล หรือเป็น Model ที่มีการเรียกใช้งานทั้งระบบ

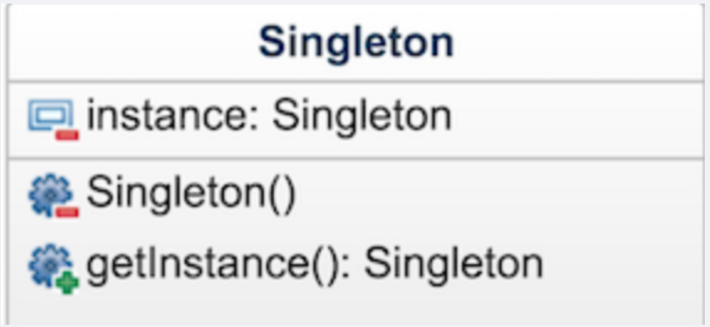


คุณสมบัติของ Singleton patternจำกัดการสร้าง instance หรือ instantiation ของคลาส เพื่อให้แน่ใจมีเพียง instace เดียวของคลาสที่ถูกสร้างใน

JVM ต้องสามารถเข้าถึงได้สาธารณะสำหรับการเอา instance ของคลาส

โครงสร้าง

ตัวอย่าง JAVAสร้าง Singleton Class เพื่อเก็บข้อมูลของผู้ใช้เอาไว้



JAVA

```java
public class UserData {

    private String fistName;
    private String lastName;

    private static UserData instance;

    private UserData() {

    }

    public static UserData getInstance() {
        if (instance == null)
            instance = new UserData();
        return instance;
    }

    public String getFistName() {
        return fistName;
    }

    public void setFistName(String fistName) {
        this.fistName = fistName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

UserData.java hosted with ❤ by GitHub view raw

ทำสร้างคลาสใหม่ขึ้นที่นี้คือ Example.java มาเพื่อเรียกใช้งานข้อมูลผู้ใช้ใน UserData.java

```java
class Example {
    public void printFistNameUser (){
        System.out.println(UserData.getInstance().getFistName());
    }

    public void printLastNameUser (){
        System.out.println(UserData.getInstance().getLastName());
    }
}
```

Example.java hosted with ❤ by GitHub view raw

ทำการทดสอบโดยทำการกำหนดค่าให้ตัวแปรใน UserData.java และทำการเรียกใช้งานข้อมูลผู้ใช้ผ่านคลาส Example.java

```java
public static void main(String[] args) {
        //init value
        UserData userData = UserData.getInstance();
        userData.setFistName("20Scoops");
        userData.setLastName("CNX");

        Example example = new Example();
        example.printFistNameUser();
        example.printLastNameUser();
}
```

Main.java hosted with ❤ by GitHub view raw

PYTHON

```python
class SingletonGovt:
    __instance__ = None

    def __init__(self):
        """ Constructor.
        """
        if SingletonGovt.__instance__ is None:
            SingletonGovt.__instance__ = self
        else:
            raise Exception("You cannot create another SingletonGovt class")

    @staticmethod
    def get_instance():
        """ Static method to fetch the current instance.
        """
        if not SingletonGovt.__instance__:
            SingletonGovt()
        return SingletonGovt.__instance__
```

PROTOTYPE PATTERNS

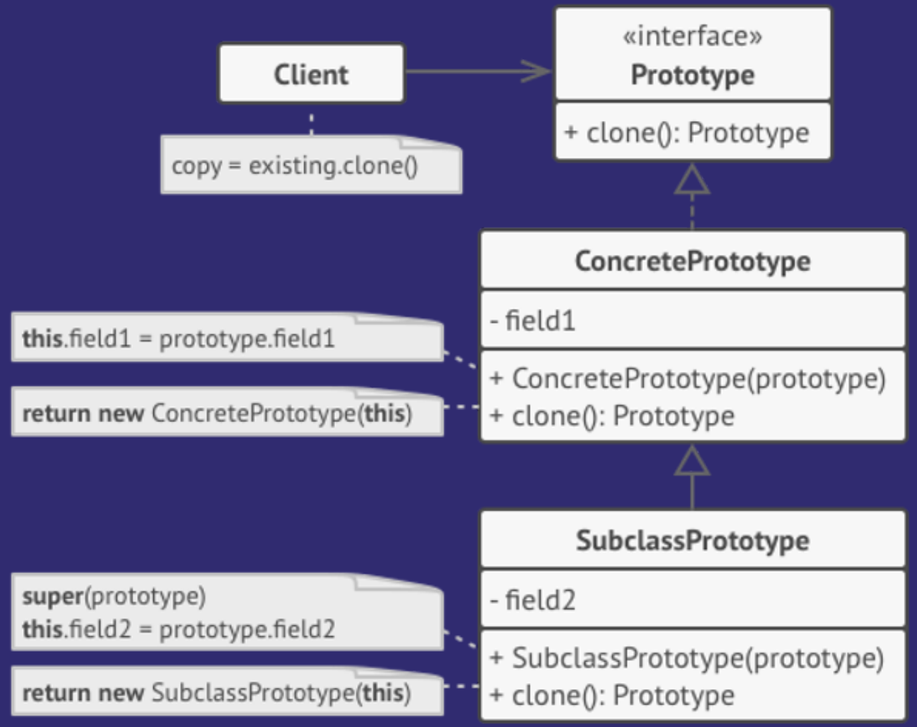
เป้าหมาย

ก๊อปปี้ object ตัวหนึ่งออกไปเป็นอีกตัวหนึ่ง โดยไม่ทำให้ code ของเรายุ่งกับ class ต่างๆ ที่เกี่ยวข้องกับ object นั้น

หลักการ

1. สร้าง interface ที่ใช้สำหรับก๊อปปี้/โคลน ออกมา 1 ตัว
2. Class ไหนที่ต้องการให้มีความสามารถในการก๊อปปี้ object ก็ไปทำการ implement interface ที่ว่านั้นซะ
3. การตั้งค่าเพิ่มเติมต่างๆสามารถไปใส่ไว้ใน subclass ได้

โครงสร้าง



Prototype - เป็น interface กลางเพื่อให้ class ที่เราอยากให้มันมีความสามารถในการก๊อปปี้/โคลน มา implement มันต่อ

Concrete Prototype - เป็น class ที่เราอยากให้มันมีความสามารถในการก๊อปปี้/โคลน (เราสามารถผลักภาระเรื่องการตั้งค่าไปให้กับ subclass มันได้)

Client - เมื่ออยากได้ก๊อปปี้ของ object ไหน เราก็แค่เรียก clone method

JAVA

```
// Clone interface
public interface ICloneable
{
    Shape Clone();
}

// Cloneable classes
public abstract class Shape : ICloneable
{
    public int X { get; set; }
    public int Y { get; set; }

    public Shape() { }

    public Shape(Shape shape)
    {
        X = shape.X;
        Y = shape.Y;
    }

    public abstract Shape Clone();
}
public class Rectangle : Shape
{
    public int Width { get; set; }
    public int Height { get; set; }

    public Rectangle()
    {
    }

    public Rectangle(Rectangle shape) : base(shape)
    {
        Width = shape.Width;
        Height = shape.Height;
    }

    public override Shape Clone()
        => new Rectangle(this);
}
```

```
public class Circle : Shape
{
    public int Radius { get; set; }

    public Circle()
    {
    }

    public Circle(Circle shape) : base(shape)
    {
        Radius = shape.Radius;
    }

    public override Shape Clone()
        => new Circle(this);
}

// Client
class Program
{
    static void Main(string[] args)
    {
        var rec1 = new Rectangle
        {
            X = 1,
            Y = 2,
            Height = 10,
            Width = 20,
        };
        var rec2 = rec1.Clone() as Rectangle;
        Console.WriteLine($"Are they equal: {rec1 == rec2}"); // false
        Console.WriteLine($"Origin - X:{rec1.X}, Y:{rec1.Y}, W:{rec1.Width}, H:{rec
        Console.WriteLine($"Cloned - X:{rec2.X}, Y:{rec2.Y}, W:{rec2.Width}, H:{rec
    }
}
```

PYTHON

```python
import copy


class Prototype:
    """
    Example class to be copied.
    """

    pass


def main():
    prototype = Prototype()
    prototype_copy = copy.deepcopy(prototype)


if __name__ == "__main__":
    main()
```

OBJECT POOL PATTERNS

จุดมุ่งหมาย

Object pooling สามารถช่วยเพิ่มประสิทธิภาพได้อย่างมากให้กับระบบที่ใช้ทรัพยากรจำนวนมากในการเตรียม object, มีการสร้าง object เป็นจำนวนมาก แต่มีจำนวนการใช้งานในช่วงเวลาหนึ่งค่อนข้างน้อย

ปัญหา

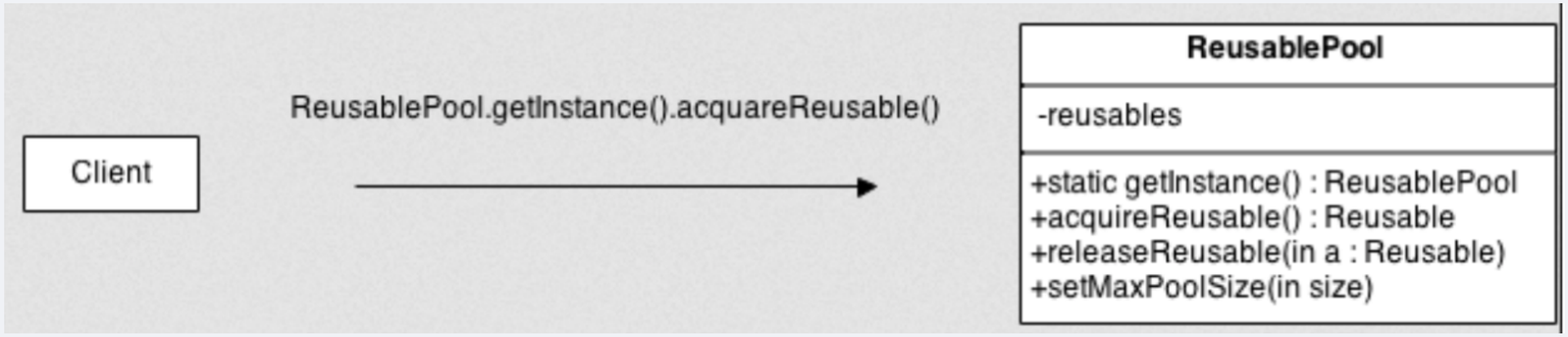
Object pool ถูกใช้เพื่อจัดการ การ cach object เมื่อผู้ใช้ต้องการ object ก็สามารถไปค้นหาใน pool ก่อน ถ้ามีอยู่แล้วก็ไม่ต้องสร้าง object ใหม่ โดยปกติแล้ว pool จะมีขนาดใหญ่ขึ้นเรื่อยๆ เพื่อความสะดวกในการจัดการ Reusable Object ที่ยังไม่ถูกใช้ควรถูกเก็บไว้ใน pool เดียวกัน ดังนั้น คลาส Reusable Pool จึงเป็นแบบ singleton class

REF : https://sourcemaking.com/design_patterns/object_pool/python/1

โครงสร้าง

แนวคิดของ Pool pattern คือ ถ้า instances ของ class ตัวไหน สามารถ reuse ได้ ก็ให้ reuse พยายามไม่สร้างขึ้นมาใหม่โดยไม่จำเป็น

Reusable - Instances ของ classes ที่ถูกใช้งานร่วมกับ object อื่นในช่วงเวลาหนึ่ง แล้วเวลาต่อมาไม่ได้ถูกใช้งาน
Client - Instances ของ classes ที่ใช้ Reusable objectReusablePool - Instances ของ classes ที่เป็นตัวจัดการให้ Client objects สามารถใช้ Reusable objects ได้



EXAMPLE

OBJECT POOL PATTERN

Object pool pattern เปรียบเสมือนเป็นโกดังของออฟฟิศ เมื่อมีพนักงานใหม่ หัวหน้างานก็จะไปเบิกอุปกรณ์สำนักงานมาจากโกดังมาให้ ถ้าของไม่มีหรือมีไม่พอ ก็ทำเรื่องจัดซื้อเพิ่ม และถ้าหากมีพนักงานลาออก อุปกรณ์สำนักงานก็จะถูกเก็บเข้าโกดัง รอวันให้พนักงานใหม่นำไปใช้ต่อ

JAVA

```java
// ObjectPool Class

public abstract class ObjectPool<T> {
  private long expirationTime;

  private Hashtable<T, Long> locked, unlocked;

  public ObjectPool() {
    expirationTime = 30000; // 30 seconds
    locked = new Hashtable<T, Long>();
    unlocked = new Hashtable<T, Long>();
  }

  protected abstract T create();

  public abstract boolean validate(T o);

  public abstract void expire(T o);

  public synchronized T checkOut() {
    long now = System.currentTimeMillis();
    T t;
    if (unlocked.size() > 0) {
      Enumeration<T> e = unlocked.keys();
      while (e.hasMoreElements()) {
        t = e.nextElement();
        if ((now - unlocked.get(t)) > expirationTime) {
          // object has expired
          unlocked.remove(t);
          expire(t);
          t = null;
        } else {
          if (validate(t)) {
            unlocked.remove(t);
            locked.put(t, now);
            return (t);
          } else {
            // object failed validation
            unlocked.remove(t);
            expire(t);
            t = null;
          }
        }
      }
    }
    // no objects available, create a new one
    t = create();
    locked.put(t, now);
    return (t);
  }

  public synchronized void checkIn(T t) {
    locked.remove(t);
    unlocked.put(t, System.currentTimeMillis());
  }
}

//The three remaining methods are abstract
//and therefore must be implemented by the subclass

public class JDBCConnectionPool extends ObjectPool<Connection> {

  private String dsn, usr, pwd;

  public JDBCConnectionPool(String driver, String dsn, String usr, String pwd) {
    super();
    try {
      Class.forName(driver).newInstance();
    } catch (Exception e) {
      e.printStackTrace();
    }
    this.dsn = dsn;
    this.usr = usr;
    this.pwd = pwd;
  }

  @Override
  protected Connection create() {
    try {
      return (DriverManager.getConnection(dsn, usr, pwd));
    } catch (SQLException e) {
      e.printStackTrace();
      return (null);
    }
  }
```

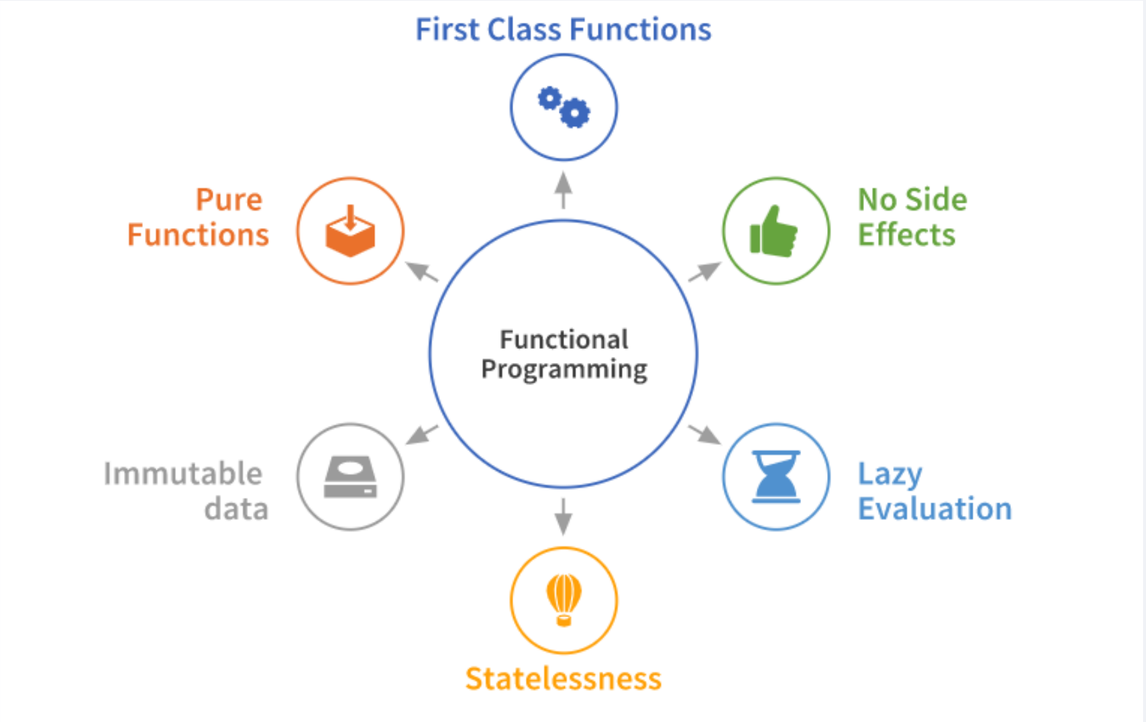
```
  @Override
  public void expire(Connection o) {
    try {
      ((Connection) o).close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  @Override
  public boolean validate(Connection o) {
    try {
      return (!((Connection) o).isClosed());
    } catch (SQLException e) {
      e.printStackTrace();
      return (false);
    }
  }
}
```

```
public class Main {
  public static void main(String args[]) {
    // Do something...
    ...

    // Create the ConnectionPool:
    JDBCConnectionPool pool = new JDBCConnectionPool(
      "org.hsqldb.jdbcDriver", "jdbc:hsqldb://localhost/mydb",
      "sa", "secret");

    // Get a connection:
    Connection con = pool.checkOut();

    // Use the connection
    ...

    // Return the connection:
    pool.checkIn(con);

  }
}
```

PYTHON

```
"""
Offer a significant performance boost; it is most effective in
situations where the cost of initializing a class instance is high, the
rate of instantiation of a class is high, and the number of
instantiations in use at any one time is low.
"""


class ReusablePool:
    """
    Manage Reusable objects for use by Client objects.
    """

    def __init__(self, size):
        self._reusables = [Reusable() for _ in range(size)]

    def acquire(self):
        return self._reusables.pop()

    def release(self, reusable):
        self._reusables.append(reusable)


class Reusable:
    """
    Collaborate with other objects for a limited amount of time, then
    they are no longer needed for that collaboration.
    """

    pass


def main():
    reusable_pool = ReusablePool(10)
    reusable = reusable_pool.acquire()
    reusable_pool.release(reusable)


if __name__ == "__main__":
    main()
```

FUNCTIONAL PROGRAMMING



FUNCTIONAL PROGRAMMING เป็น programming paradigm
หรือรูปแบบวิธีคิดในการเขียนโปรแกรมแบบหนึ่ง

REF : https://www.facebook.com/programmerthai/posts/2121809908108849/

แนวคิดก็ต้องเน้นสร้างฟังก์ชั่น แต่หลักสำคัญต้องออกแบบให้หลีกเลี่ยง **side-effect** (ผลข้างเคียง) ที่จะเกิดต่อ **function** ตัวเอง และตัวอื่น โดยมีหลักยึด **2** อย่างใหญ่คือ

1) function ที่สร้างขึ้นมา เมื่อมีอินพุตค่าเดิมส่งไปหา (เป็นค่าอากิวเมนต์) ไม่ว่าจะกี่ครั้งก็ตาม function จะรีเทิร์นค่าออกมาเหมือนเดิมทุกครั้ง เช่น
func(1); // เรียกครั้งที่ 1 ก็จะได้ค่ารีเทิร์นออกมาเป็น 30
func(1); // เรียกครั้งที่ 2 ก็จะได้ค่ารีเทิร์นออกมาเป็น 30 เหมือนเดิม
func(1); // เรียกครั้งที่ 3 ก็จะได้ค่ารีเทิร์นออกมาเป็น 30 เหมือนเดิม

2) function ต้องไม่ไปเปลี่ยนแปลงค่าของตัวแปรจำพวก global variable หรือ static variable หรือตัวแปรที่อยู่ข้างนอก function เพื่อไม่ให้ function อื่นได้รับผลกระทบถ้า function ที่เราประกาศไว้
คุณสมบัติ 2 อย่างที่ว่านี้ ก็จะเรียกว่า pure function (ฟังก์ชั่นบริสุทธ์แท้ๆ)

อีกทั้งคำว่า first-class function คุณสมบัตินี้ function จะถูกมองเป็นข้อมูลประเภทหนึ่ง ไม่ต่างจากข้อมูลตัวเลข สตริง บูลีน ด้วยเหตุนี้จึงสามารถนำ function ไปกำหนดค่าให้กับตัวแปรได้เลย เช่น x = function(){ .....}
ส่วนคุณสมบัติ Higher-order function:คุณสมบัตินี้หมายถึง เราสามารถใช้ function ส่งไปเป็นค่าอากิวเมนต์แก่ function ตัวอื่น หรือ function จะรีเทิร์นออกมาจาก function ตัวอื่นออกมาก็ได้ด้วย

ประโยชน์

FUNCTIONAL PROGRAMMING

Testability

test ง่าย เนื่องจากพฤติกรรมของ function นั้นคาดเดาได้ง่าย เพราะการหลีกเลี่ยง side-effect ทำให้เกิด deterministic function ถ้าใส่ input a ได้ output b แล้วผลลัพธ์จะเป็นเช่นนั้นเสมอ

Parallel Computing

ผลจากการหลีกเลี่ยง side-effect ทำให้ function ใดๆ ไม่สามารถไปเปลี่ยนแปลง state ไม่สร้างผลกระทบใดๆ ต่อ function ที่ทำงานคู่ขนานกันได้ ทำให้เราจัดการกับการทำงานแบบคู่ขนานได้ง่ายกว่าการปล่อยให้มีการ mutate state

Readability

functional programming ทำให้ code อ่านง่ายกว่ามาก แต่ก็มีมีบางกรณีที่ทำให้ code อ่านเข้าใจยากมากๆเช่นกัน

MEMBER

Krittanai Pahonkan

61070003

Sirawit Bosri

61070220

Arnon Unthon

61070268

Nirawit Naktham

61070343

Angwara Paolaklaem

61070351