Object Orientation with Design Patterns

Lecture 3:
Factory Method Pattern
Singleton Pattern

Factory Method Pattern

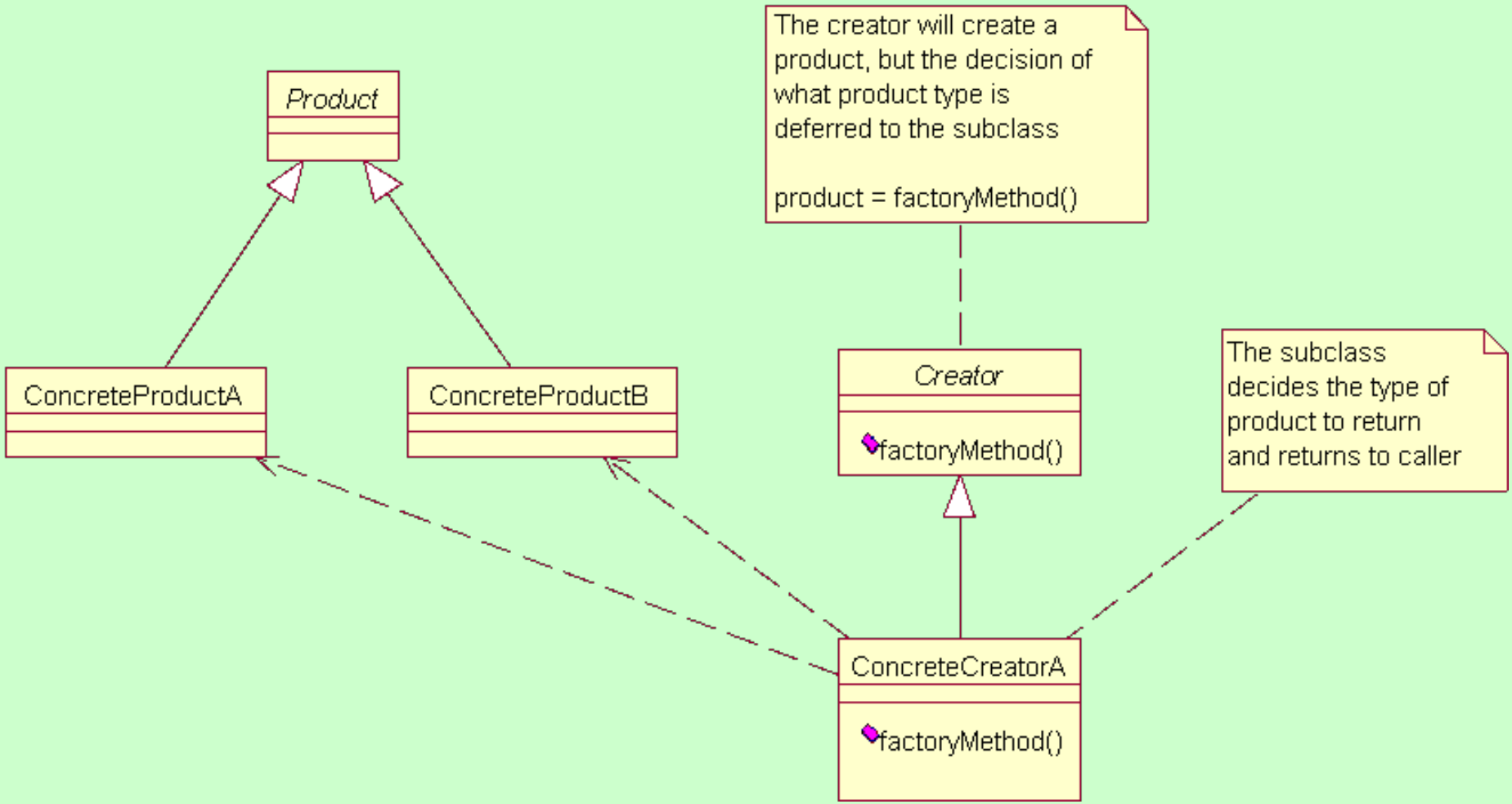
Factory Method Pattern

- This pattern is a **subtle extension** of the simple **Factory Pattern**
- No **single class** makes **the decision** as to **which class to instantiate**.
- The superclass **defers the decision** to the subclass

Structure of Factory Method

- Every factory method returns a **new object**
- A Factory Method returns a type that is an abstract class or an interface
- A **Factory Method** is usually implemented by several classes, **each subclass returning a particular type of object**

Structure of Factory Method



Factory Method

- **Intent:** Define **an interface** for creating an object, but let **the subclasses decide** which class to instantiate.
  Factory Method lets a **class defer instantiation to subclasses.**

- An object creator makes a choice about which of several possible objects to instantiate for the client.

Factory Method example

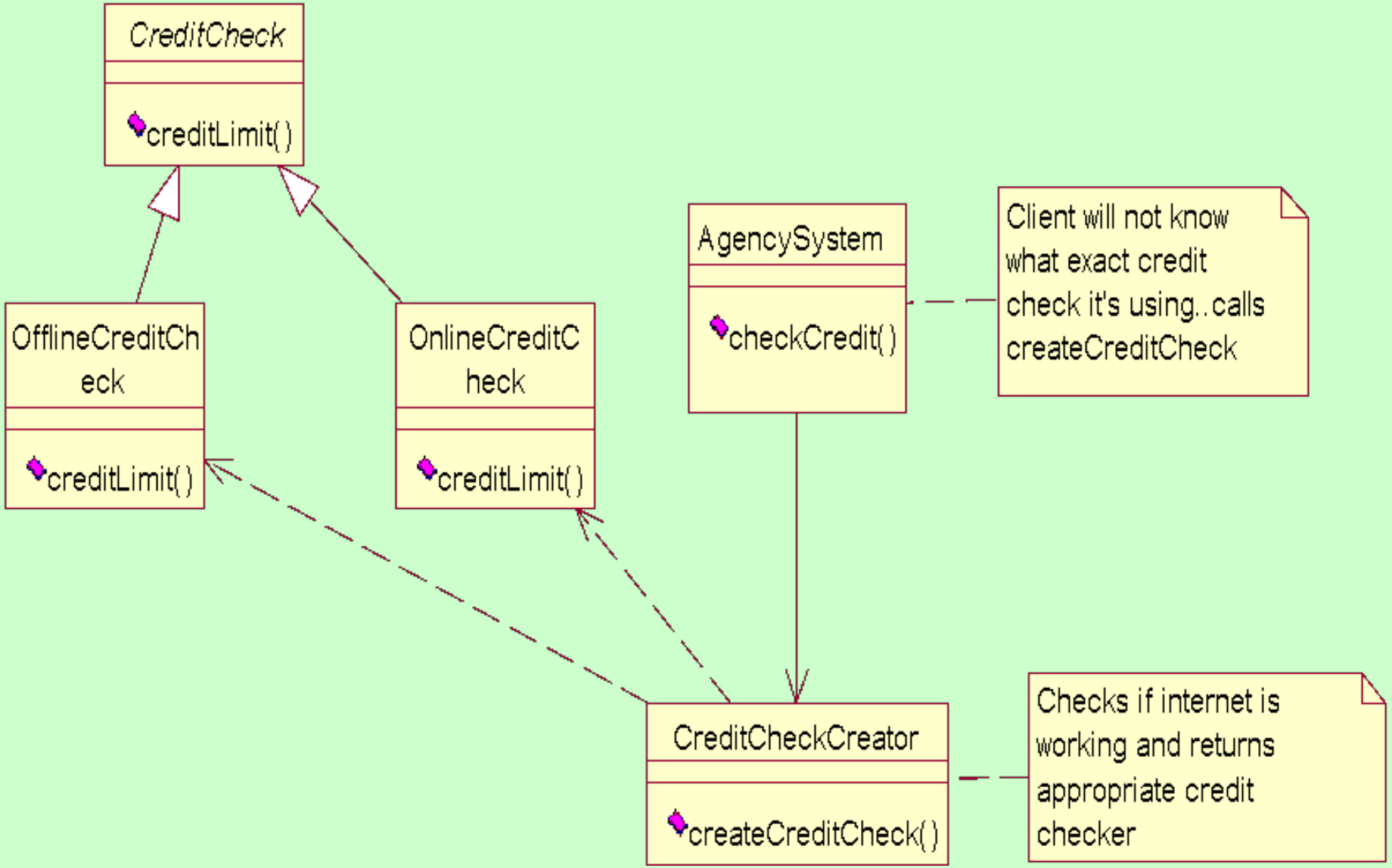
- Suppose a company carries out **credit checks** on it's customers using a **credit check agency.**

- The agency normally carries out the credit check using an **automated credit check system** (which is connected to the internet).

- However when the **internet access is down** the agency's credit check system will carry out an **offline credit check.**

- The **agency user** need **not know** whether the system is carrying out an **online or offline credit check**, they just want to fail or pass a customer for credit.

Credit Check system



Credit Check System

- The createCreditCheck() method is a **factory method**
- The createCreditCheck() method will check to see whether the **internet connection is up**
- If the **internet connection is up** then an **online credit check object** is created and the credit check system tells the agency employee whether the credit check is good or bad
- If the **internet connection is down** then the system will automatically display the dialog for the credit check questions **(offline credit check)**

Factory method for Credit Check

```
public class CreditCheckCreator {


public static CreditCheck createCreditCheck() {
if(isNetWorking()) {
return new OnlineCreditChecker();
}
else {
return new OfflineCreditChecker();
}
}
}
```

Participants of Factory Method

Product: (CreditCheck)

- Defines the interface of objects the factory method creates

ConcreteProduct: (OnLineCreditCheck)

- Implements the product interface

Participants of Factory Method

Creator: (AgencySystem)

- Declares the factory method which **returns an object of type Product**. Could have a default implementation.
- May call the factory method to create a Product object

Concrete Creator (CreditCheckCreator)

- **Overrides the factory method** to return an instance of a ConcreteProduct

Consequences of Factory Method

- The Factory Method pattern **eliminates the need to build application specific classes** into your code

- Makes **code a lot more flexible**, easier to place new products into the system

- The need to subclass however means that clients need to involve themselves with a base class

The Singleton Pattern

The Singleton Pattern

- The singleton pattern is grouped with other creational patterns, although it is to some extent a pattern that **actually limits the creation of objects**.

- Specifically, it ensures that there is **one and only one instance of a class** and provides **a global point of access to that instance.**

- Any number of cases in programming in which you need to ensure that there can be only one instance of a class are possible.

- For example your system might have only **one window manager** or **print spooler** or **one point of access to a database system.**

Creating and using a static method

- The easiest way to create a class that can have only one instance is to embed a **static variable inside of the class that is set on the first instance** and then check for it each time that you enter the constructor.

- A ***static*** **variable** is a variable for **which there is only one instance**, no matter how many instances of the class exist.

- To **prevent instantiating the class more then once** we can **make the constructor private** so that an instance can be created only from **within a static method**. In other words an instance cannot be created in the normal way using the *new* operator.

- The following example illustrates this point !

```
class WindowManager
{
    // This will be the one and only WindowManager instance
    private static WindowManager wmanager;

    private WindowManager()
    {
        // Private constructor that does nothing
    }

    // Public synchronized method which will return a
    // WindowManager
    public static synchronized WindowManager getManager()
    {
        // If true then we need to create an instance of
        // WindowManager
        if (wmanager == null)
            wmanager = new WindowManager();

        return wmanager;
    }

    // Test method so we can ensure that our
    // object works
    public void print(String message)
    {
        System.out.println(message);
    }
}
```

Creating and using a static method

```
class WindowManagerTest
{
    public static void main(String[] args)
    {
        // This will create a new instance of WindowManager
        WindowManager wm1 = WindowManager.getManager();
        wm1.print("Hello im a new Singleton");

        // This will reuse the same instance of WindowManager
        WindowManager wm1 = WindowManager.getManager();
        wm1.print("Hello im the same Singleton");
    }
}
```

Creating and using a static method

- This approach has the major advantage that **you don't have to worry about exception handling**, if the Singleton already **exists** you **always get the same instance of WindowManager.**

- And, should you **try to create instances of the WindowManager directly** you will receive **an error** from the compiler because the **constructor is private**.

```
// Fails at compile time because constructor
// is private
WindowManager wm1 = new WindowManager();
```

Exceptions and Instances

- If you want to know whether **you have created a new instance**, you must **check the return value of the getManager** method to make sure its not null.

- This assumes that all programmers will remember to check the return value, which we know will cause problems.

Exceptions and Instances

- Another approach to the problem is to **use exceptions to make sure that only one instance** of the WindowManager class can be **created.**

- This approach **requires the programmer to take action** and is therefore a safer approach.

- To use this approach the first thing we need to do is **create** our own **Exception class**.

Exceptions and Instances

```
class SingletonException extends RuntimeException
{

    public SingletonException()
    {
        super();
    }

    public SingletonException(String s)
    {
        super(s);
    }
}
```

Exceptions and Instances

- Note that other than calling its parent class constructor through the *super* method this class doesn't do very much.

- However it is convenient for us to have our **own Exception type** so that the **compiler warns us** of the type of exception to be caught when we try and create an instance of the WindowManager class.

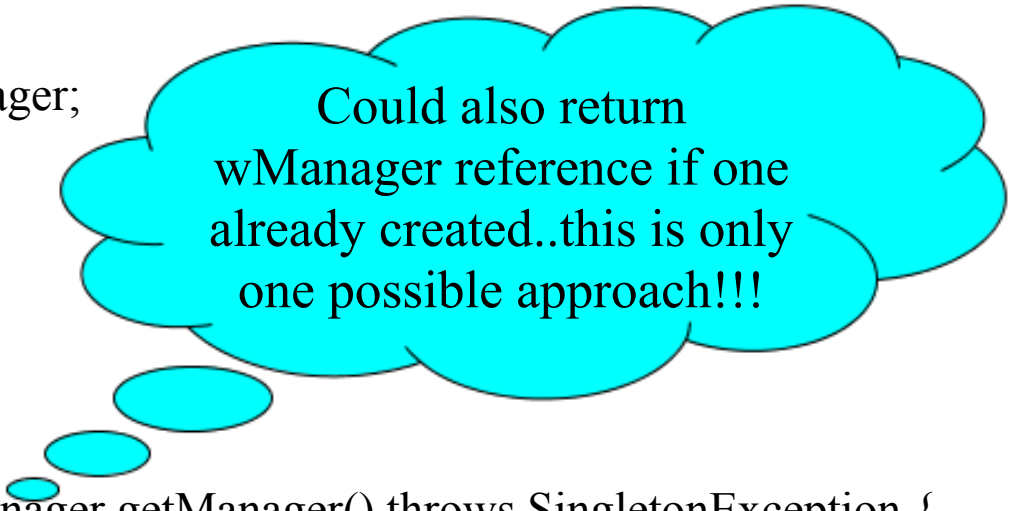
- The next thing we need to do is **re-write our WindowManager class** so that it will **throw an exception** if we try to create **more than one instance**.

- The following slide shows the second version of WindowManager.

```
public class WindowManager {
private static WindowManager wManager;


private WindowManager() {
}


public static synchronized WindowManager getManager() throws SingletonException {
if(wManager==null) {
wManager = new WindowManager();
return wManager;
}
else {
throw new SingletonException(“Only one window manager allowed");
}
}
}
```



```
public class SingletonTest {
public static void main(String args[]) {
try {
WindowManager wManager = WindowManager.getManager();
}
catch(SingletonException e) {
e.printStackTrace();
}

try {
WindowManager wManager = WindowManager.getManager();
}
catch(SingletonException e) {
e.printStackTrace();
}
}
}
```

Exceptions and Instances

- As you can see from the program output below, when we try to create the second WindowManager a SingletonException is thrown and the exception message is output to the screen.

```
SingletonException: Singleton class already instatntiated

at WindowManager.getManager(WindowManager.java:15)

at SingletonTest.main(SingletonTest.java:18)
```

Providing a Global Point of Access

- Since a singleton pattern is used to **provide a single point of access to a class,** your program design must provide for **a way to reference the Singleton** throughout the program, even though Java has no global variables.
- One solution is to create such Singletons at the beginning of your program and pass them as arguments (parameters) to the major classes that might need to use them.

```
try{
     wm1 = new WindowManager();

     AppWindow appwin = new AppWindow(wm1);
}
catch(SingletonException e){
     System.out.println(e.getMessage());
}
```

Providing a Global Point of Access

- The disadvantage is that **you might not need all of the Singletons that you create for a given program** execution. This could have performance issues.

- A more elaborate solution is to **create a registry of all of the programs Singleton classes and make the registry generally available.** Each time a Singleton is instantiated, it notes that in the registry.

- Then any part of the program **can ask for the instance of any Singleton** using an **identifying string** and get back that instance variable.

Providing a Global Point of Access

- One disadvantage of the registry approach is that **type checking might be reduced**, since the table of singletons in the registry probably keeps all of the Singletons as objects, for example in a Hashtable or Vector object.

- And of course the registry itself is probably a Singleton and it will have to be passed to other objects which need to use some of the Singleton classes which it manages.

- Probably the most **common way** of providing **global access** is through **static methods**.

Providing a Global Point of Access

- The **class name** is always **available**, and the **static methods** can be called **only from the class** and not from its instances, so there is never more than **one such instance** no matter how many places in your program call that method.

- Java uses this approach in its serial port package, javax.comm.

- The javax.comm is provided separately from the JDK and is downloadable from the Sun Web site. This package is designed to **provide control over the serial and parallel ports of a computer**.

- **Serial ports** are good example examples of resources which could be controlled by a **singleton**. For example the Singleton could manage a collection of ports and let out one instance of each port.

Providing a Global Point of Access

- Staying with the communication theme the next example shows how a Singleton class can be used for socket communication.

- Using this Singleton any object can send a message without having to deal with network/socket code.

Providing a Global Point of Access

- Also, because all the comm's are encapsulated within a Singleton an object can send a message by getting a handle (reference) to the Singleton and call a send method.

- So this means that at any point in a program we can send a message without having to worry about whether we have a reference to a send object because we can get one at any time !

MessageManager Singleton

```
class MessageManager
{
    // This will be the one and only MessageManager instance
    private static MessageManager manager;
    private static DatagramSocket dsock;

    private MessageManager()
    {
        try
        {
            dsock = new DatagramSocket();
        }
        catch(IOException e)
        {
            System.out.println(e.toString());
        }
    }
```

MessageManager Singleton

```
// Public synchronized method which will return a
// MessageManager
public static synchronized MessageManager getManager()
{
     // If true then we need to create an instance of
     // WindowManager
     if (manager == null)
          manager = new MessageManager();

     return manager;
}
```

MessageManager Singleton

```
public void send(String hostname, int port, String message)
{
    try
    {
        InetAddress address = InetAddress.getByName(hostname);

        byte[] sendBuf = message.getBytes();

        DatagramPacket packet =
        new DatagramPacket(sendBuf, sendBuf.length, address, port);

        dsock.send(packet);
    }
    catch (Exception e)
    {
        System.out.println(e.toString());
    }
}
```

MessageManager Singleton

- The MessageManager singleton class follows pretty much the same outline as the WindowManager example seen previously.
- Its **constructor method** creates a **DatagramSocket which will be used to send a simple text message**.
- It has a **static getManager method** which operates in much the same way as the WindowManager getManager method.
- Instead of a simple print method it has a **send method** which **will look after create a packet** and **sending the text message using the DatagramSocket.**

MessageManager Singleton

- A simple test program shows how we can utilize the MessageManager class.

```
class MessageManagerTest
{
    public static void main(String[] args)
    {
        MessageManager m = MessageManager.getManager();

        for(int i = 0; i < 10; i++)
        {
            m.send("localhost", 5001, "Hello");
        }
        m.send("localhost", 5001, "STOP");
    }

}
```

Consequences of the Singleton

- The singleton pattern has the following additional consequences:

1. **Subclassing (extending) a Singleton** can be **difficult**, since this can work only if the **base Singleton class has not yet been instantiated.**

2. You can easily change a Singleton to **allow a small number of instances** where this is allowed and meaningful.

Exercise 1 – Singleton Pattern

- Create **3 applications** to demonstrate the various ways that a global point of access to a PrintSpooler can be implemented.
  Within one of the programs, show how 2 different variables object names are the same instance of the PrintSpooler class.

- In your own words, explain how each example works and the differences between each example.

Exercise 2

- In your own words explain the differences between the Factory, the Abstract Factory and the Factory method patterns.
- Illustrate your answer with real world examples