

# Object Orientation with Design Patterns



## **Lecture 4:** **Factory Method Pattern** **Singleton Pattern**

# Factory Method Pattern

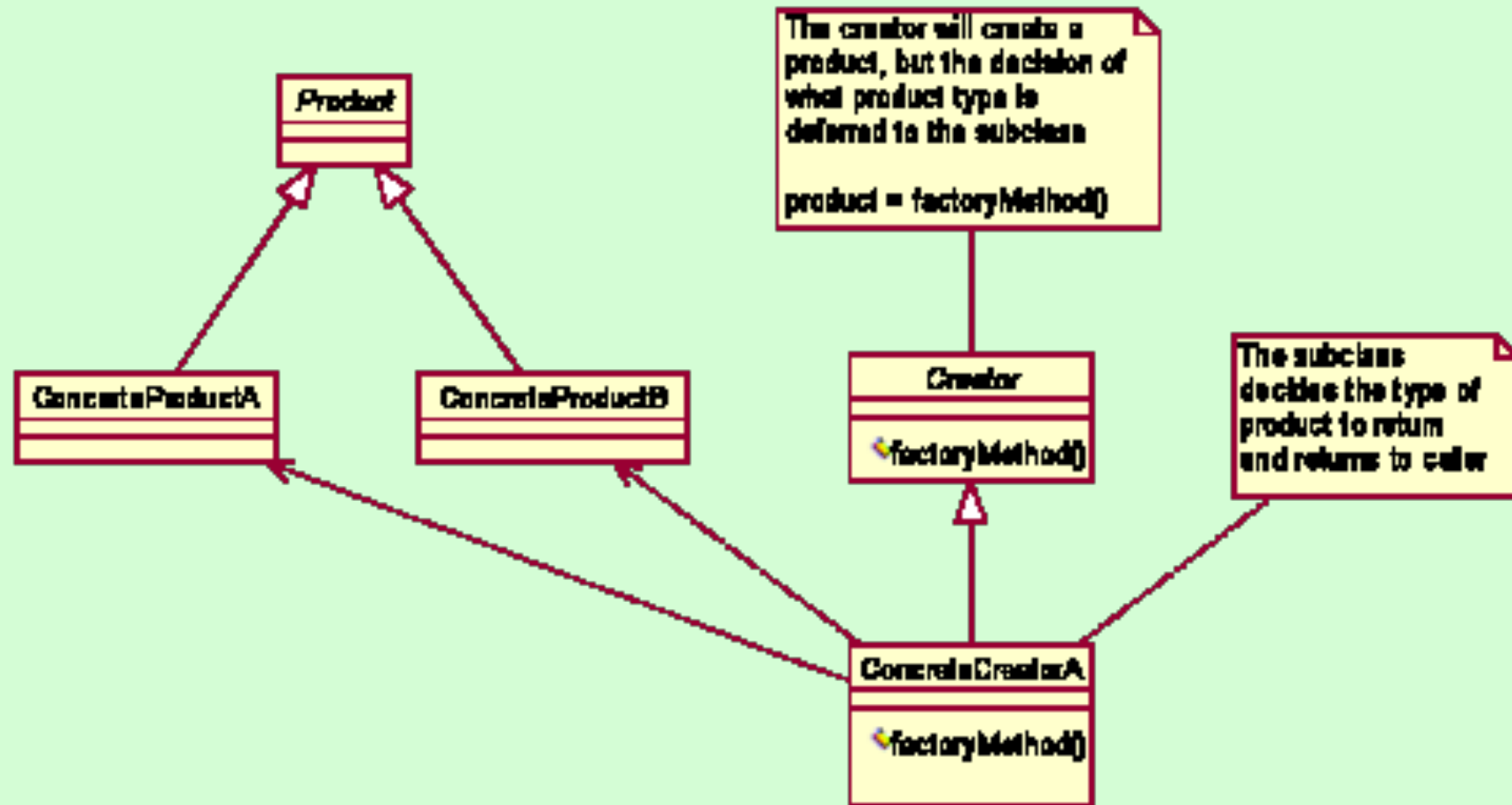
# Factory Method Pattern

- 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



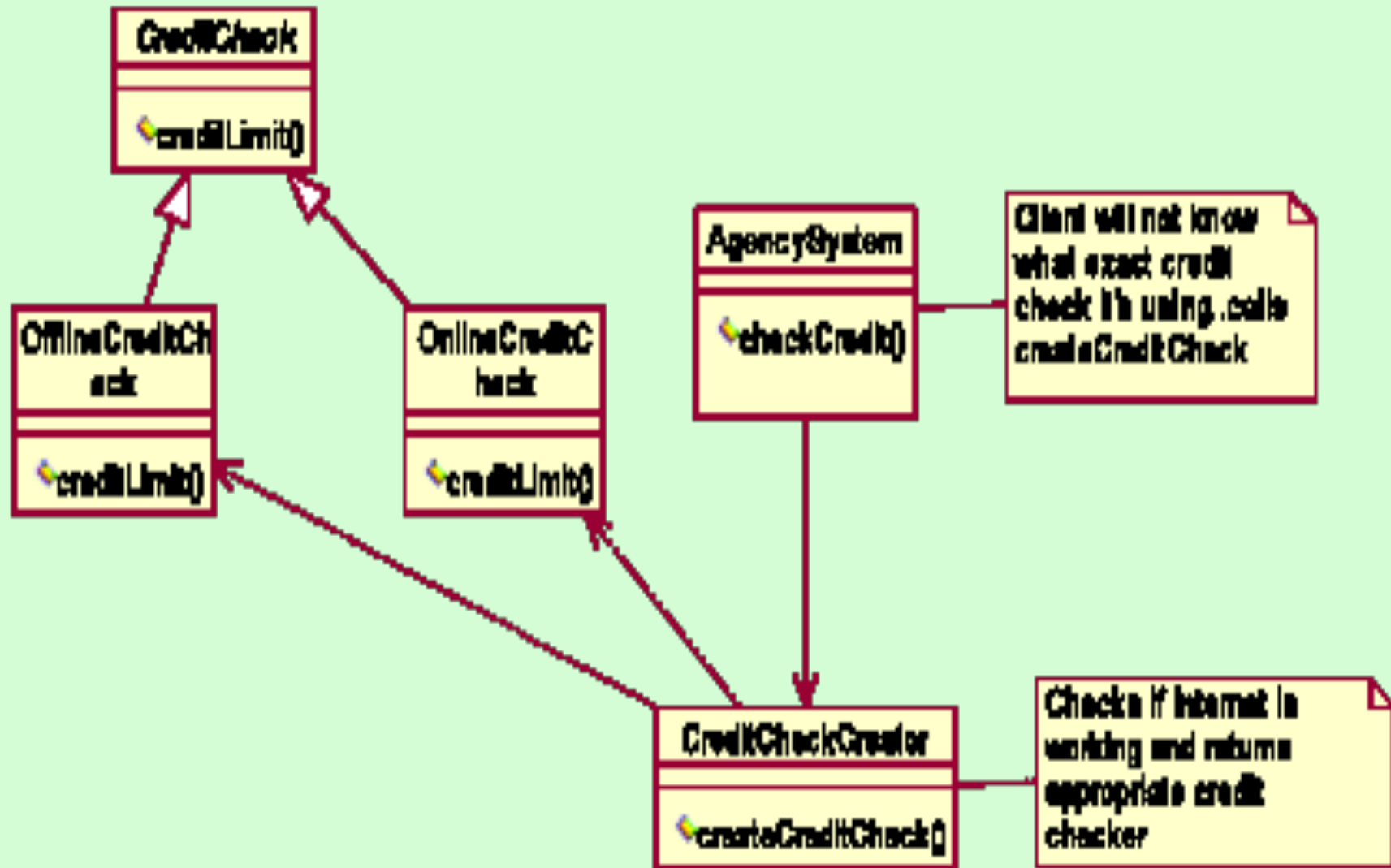
# Factory Method

- **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

# POP QUIZ



You are currently building a house and have decided to put arched windows into your house. You go to the windows supplier to purchase the windows.

However the supplier may not have the arched windows in stock and may need to import them from a German manufacturer.

You don't really care if the windows are in stock , you just want them delivered by a certain date.

Using the **Factory Method** pattern, draw a UML diagram that will illustrate the above example.

# The Singleton Pattern

# The Singleton Pattern

- The singleton pattern is grouped with other creational patterns, although it is a pattern that **actually limits the creation of objects** in most cases/contexts
- In its simplest form the Singleton pattern 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 if you were to create a model of our solar system there is only **one** Sun



# The Singleton Pattern

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

# Creating and using a static method

- In OOP the major “things” in your software are represented by classes (e.g. a **Sun** class)
- **Constructors** create objects using the keyword **new**
- In the case of creating a model of OUR solar system **we cannot allow the constructor to be called an unspecified number of times** as it would be possible to create more than one Sun

# Creating and using a static method

- We must restrict the use of the keyword **new** to **limit the number of instances of the Sun** available to maintain the consistency of the solar system's model ("there can be only one Sun in our solar system")
- In order to successfully restrict the construction of multiple Sun objects we must design the Sun class so that the **new** keyword can only be used in a restricted way
- This will involve **restricting access to constructors**

# 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 than 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.

# Methods of Singleton implementation

- There are several ways to implement a Singleton pattern
- The client program (creates the singleton object) can be aware of the Singleton, or it can be kept hidden
- Approaches to Singleton implementation include:
  - Attempting to create too many instances of the Singleton results in the **returning of the same instance of the Singleton**, i.e., we share same instance (e.g. you always get the same instance of a Sun object even if you request another one)
  - You can throw an exception to **inform** the client program that they are breaking the Singleton limit and refuse to return and instance

## ➤ Private constructor and static variable, return same instance

```
public class Sun extends CelestialBody {  
  
    private static Sun sun;  
    private static SolarSystem solarSystem;  
    private String name;  
    private Vector<CelestialBody> orbitingBodies= new Vector<CelestialBody>();  
  
    private Sun(String name, SolarSystem solarSystem) { //NOTE: Private constructor  
        this.name = name;  
        this.solarSystem = solarSystem;  
    }  
  
    public static Sun createSun(String name, SolarSystem system) {  
        if(sun==null) { //If no Sun has been created, allow Sun Singleton creation  
            sun=new Sun(name,system); //Add a Sun to the Solar System  
            solarSystem = system; //set Solar system  
            return sun; //Return instance of Sun  
        }  
        return sun; //always return the same instance  
    }  
}
```

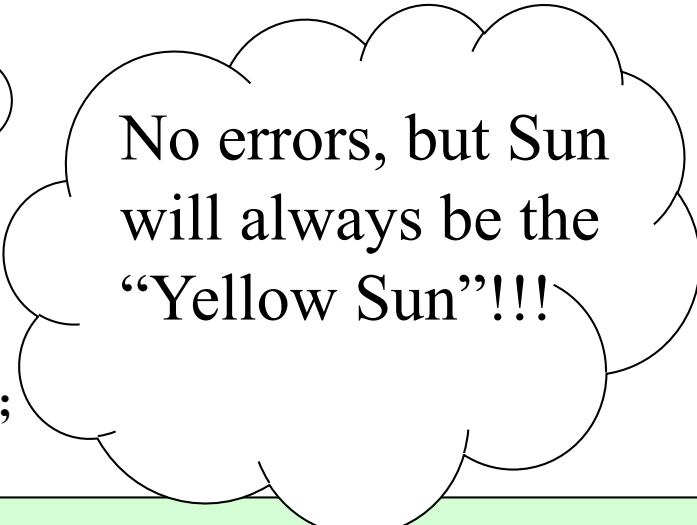
# Creating and using a static method

- This approach has the major advantage to the designer, i.e., **you don't have to worry about exception handling**, if the Singleton already **exists** you **always get the same instance of Sun (shared instance)**.
- And, should you **try to create instances of the Sun directly** you will receive **an error** from the compiler because the **constructor is private**.

```
earthSystem = new SolarSystem();  
Sun earthSun = new Sun("The Sun", earthSystem);  
(Error: Constructor is NOT visible!!!)
```

# ‘static’ method to create instance

```
public static void main(String[] args) {  
  
    SampleSolarSystem solarSystem = new SampleSolarSystem();  
  
    earthSystem = new SolarSystem();  
  
    //Call to create first singleton will pass  
    yellowSun = Sun.createSun("Yellow Sun",earthSystem);  
  
    //This second call will result in the Yellow Sun being returned, no NEW instance created!  
    blueSun = Sun.createSun("Blue Sun",earthSystem);  
  
    earthSystem.addSun(blueSun);  
    earthSystem.addSun(yellowSun);  
  
    blue_earth = new Planet("Blue Earth",blueSun);  
    earth = new Planet("Yellow Earth",yellowSun);  
  
    solarSystem.displayArea.append("" + earthSystem);  
}
```



No errors, but Sun  
will always be the  
“Yellow Sun”!!!



# Exceptions and Instances

- The issue with the last method of implementing the Singleton is that the client program is **not aware of the Singleton** and this might confuse the client programmer as the 'wrong' (unexpected) instance is returned
- This assumes that all programmers will remember to check the return value from the Singleton

# Exceptions and Instances

- It is more likely that the client programmer will be 'surprised' to find that instead of a second instance being returned **the previously created instance has been shared**
- Surprising client programmers is generally considered to be bad practice as the behaviour of the software should be **predictable and in line with normal expectations**

# Exceptions and Instances

- Another approach to the problem is to **use exceptions to make sure that only one instance** of the Sun class can be **created**.
- This approach **requires the programmer to take action** and is therefore a safer approach as the client programmer is not 'surprised'.
- To use this approach the first thing we need to do is **create** our own **Exception class**.

# Singleton Exception class

```
public class SolarSystemSunLimitException extends Exception {  
  
    public SolarSystemSunLimitException(String message) {  
  
        super(message);  
  
    }  
  
}
```

# 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 will warn us** of the type of exception to be caught when we try and create an instance of the Sun class.
- The next thing we need to do is **re-write our Sun 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 the **Sun** class.

## ➤ Private constructor and static variable, uses Exception

```
public class Sun extends CelestialBody {

    private static Sun sun;
    private static SolarSystem solarSystem;
    private String name;
    private Vector<CelestialBody> orbitingBodies= new Vector<CelestialBody>();

    private Sun(String name, SolarSystem solarSystem) {
        this.name = name;
        this.solarSystem = solarSystem;
    }

    public static Sun createSun(String name, SolarSystem system) throws
    SolarSystemSunLimitException {
        if(sun==null) { //If no Sun has been created, allow Sun Singleton creation
            sun=new Sun(name,system); //Add a Sun to the Solar System
            solarSystem = system; //set Solar system
            return sun; //Add sun to the list of Suns in Solar system
        }
        else { //If the Singleton pattern is exceeded, i.e., more than allowed objects created
            throw new SolarSystemSunLimitException("This Solar System can only have 1 Sun");
        }
    }
}
```

# Create instances using Exception

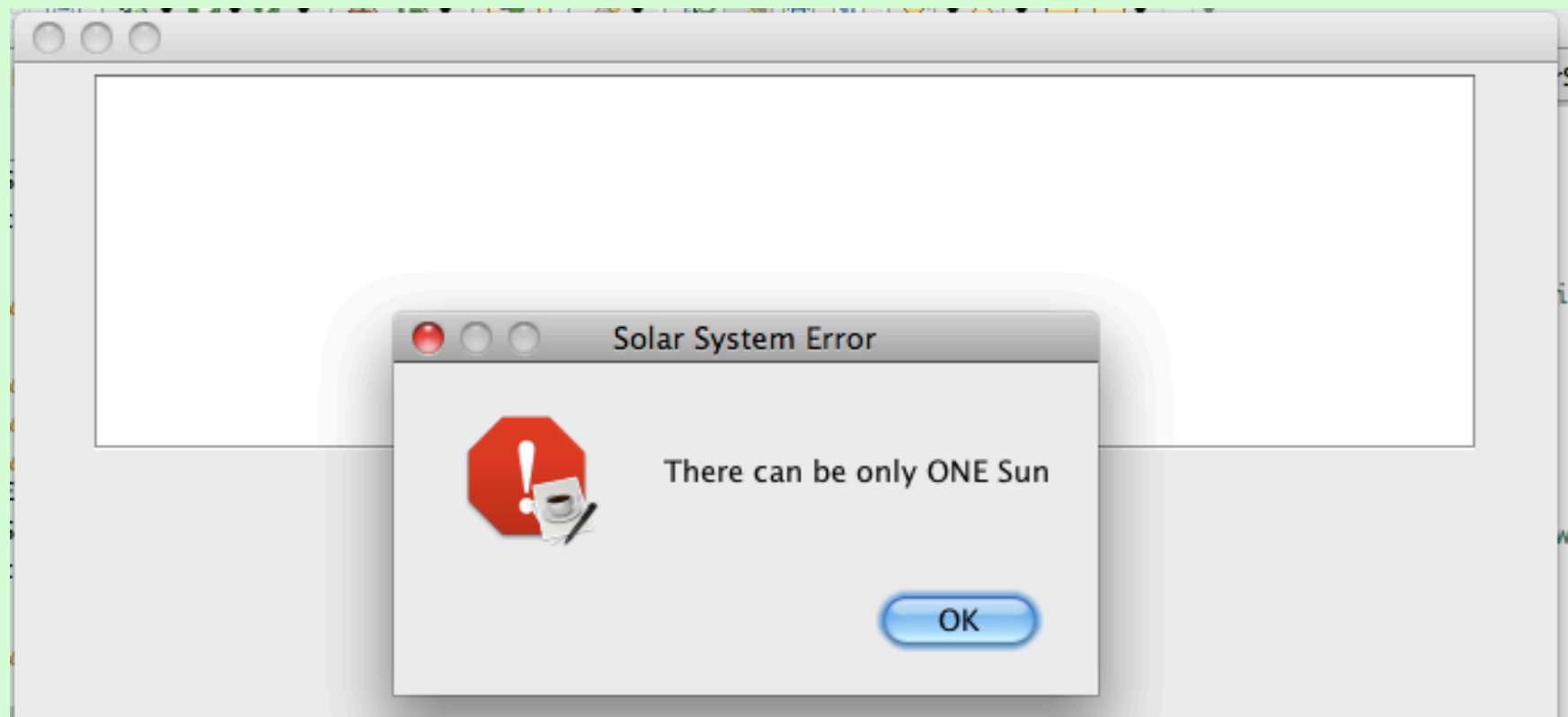
```
try {  
    //Create solar system, restricted singleton max set in parameter  
    earthSystem = new SolarSystem();  
  
    //Call to create first singleton will pass  
    yellowSun = Sun.createSun("Yellow Sun",earthSystem);  
  
    //This second call result in an SolarSystemSunLimitException being thrown  
    blueSun = Sun.createSun("Blue Sun",earthSystem);  
  
    earthSystem.addSun(blueSun);  
    earthSystem.addSun(yellowSun);  
  
    blue_earth = new Planet("Blue Earth",blueSun);  
    earth = new Planet("Yellow Earth",yellowSun);  
  
    solarSystem.displayArea.append("'" + earthSystem);  
  
} catch (SolarSystemSunLimitException e) {  
    JOptionPane.showMessageDialog(solarSystem, "There can be only ONE Sun","Solar  
System Error",JOptionPane.ERROR_MESSAGE);  
}
```



Catch Singleton  
Exception

# Exceptions and Instances

- As you can see from the program output below, when we try to create the second Sun a SingletonException is thrown and the exception message is output using an option pane.





# Restricted number of instance

- As we have seen so far the Singleton Pattern restricts the creation of objects of a class
- In many software system it may be necessary to restrict **the number of instances of a class to a small number of instances of the class**, rather than just ONE instance
- The Singleton pattern can be modified to allow this, instead of the constructor allowing only one instance it could be modified to allow a **limited number of instances**

# Restricted number of instance

- Our Solar System only has ONE Sun
- If you are planning to model the entire Universe there may be instances where there is **more than one Sun** being orbited by planets
- In this case the Solar system would have more than one instance **but the number of Suns must still be limited**

# 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.

# Restricted number of instance

- The following class restricts the creation of the instances of the Sun class, however, the limit is set by the Solar System that it is in
- If the solar system is created to allow four Suns then THREE instances of Sun are permitted but no more than that:
- *//three Suns allowed in this Solar System*  
*alienSystem = **new SolarSystem(3);***

# Multiple instance static method

```
public static Sun createSun(String name, SolarSystem system) throws
SolarSystemSunLimitException {

    if(sunList.size()==0) { //If no Sun has been created, allow Sun Singleton creation
        numSun++; //set number of Suns created
        sunList.add(new Sun(name,system)); //Add a Sun to the Solar System
        solarSystem = system; //set Solar system
        return sunList.elementAt(0); //Add sun to the list of Suns in Solar system
    }
    else if(numSun<solarSystem.getNumSuns()) { //If multiple Suns in the Solar System
        numSun++;
        solarSystem = system;
        sunList.add(new Sun(name,system));
        return sunList.lastElement(); //return most recent sun created
    }
    else { //If the Singleton pattern is exceeded, i.e., more than allowed objects created
        throw new SolarSystemSunLimitException("This Solar System can only have " +
solarSystem.getNumSuns() + " sun");
    }
}
```

# Create multiple instances

```
try {  
    //Create solar system, restricted singleton max set in parameter  
    alienSystem = new SolarSystem(3);  
  
    //Call singleton Sun public access method  
    yellowSun = Sun.createSun("Yellow Sun",alienSystem);  
    blueSun = Sun.createSun("Blue Sun",alienSystem);  
    greenSun = Sun.createSun("Green Sun",alienSystem);  
  
    alienSystem.addSun(blueSun);  
    alienSystem.addSun(yellowSun);  
    alienSystem.addSun(greenSun);  
  
    earth = new Planet("Yellow Earth",yellowSun);  
    blue_earth = new Planet("Blue Earth",blueSun);  
    green_earth = new Planet("Green Earth",greenSun);  
  
    solarSystem.displayArea.append("" + alienSystem);  
} catch (SolarSystemSunLimitException e) {  
    JOptionPane.showMessageDialog(solarSystem, "Attempt to exceed upper limit of Suns  
which has been set to " + alienSystem.getNumSuns() + "\nIncrease Number of Suns or  
Decrease Number of Suns added","Solar System  
Error",JOptionPane.ERROR_MESSAGE);  
}
```