Design Patterns

Gerwin van Dijken (gerwin.vandijken@inholland.nl)

Program term 1.4

01 (wk-15)	abstract classes and interfaces
02 (wk-16)	Template Method pattern / Observer pattern
03 (wk-17)	MVC pattern
04 (wk-18)	*no classes*
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	repetition / practice exam

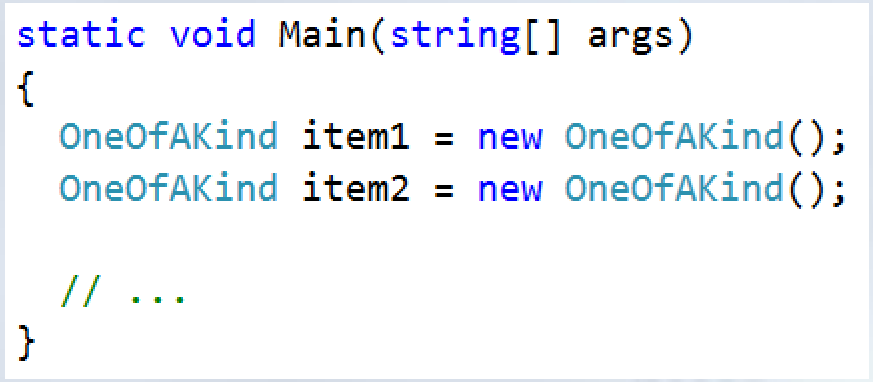
----

09 (wk-23)	exam *(computer assignments)*
10 (wk-24)	*retakes (courses term 1.3)*
11 (wk-25)	*retakes (courses term 1.4)*

Singleton pattern

The Singleton Pattern (GoF): 'ensures a class has only one instance, and provides a global point of access to it.'

- Make sure that there is only one (unique) instance of a class
- So, the next code must result in the same object



```
static void Main(string[] args)
{
  OneOfAKind item1 = new OneOfAKind();
  OneOfAKind item2 = new OneOfAKind();

  // ...
}
```

(one object on the heap, available by item1 and by item2)

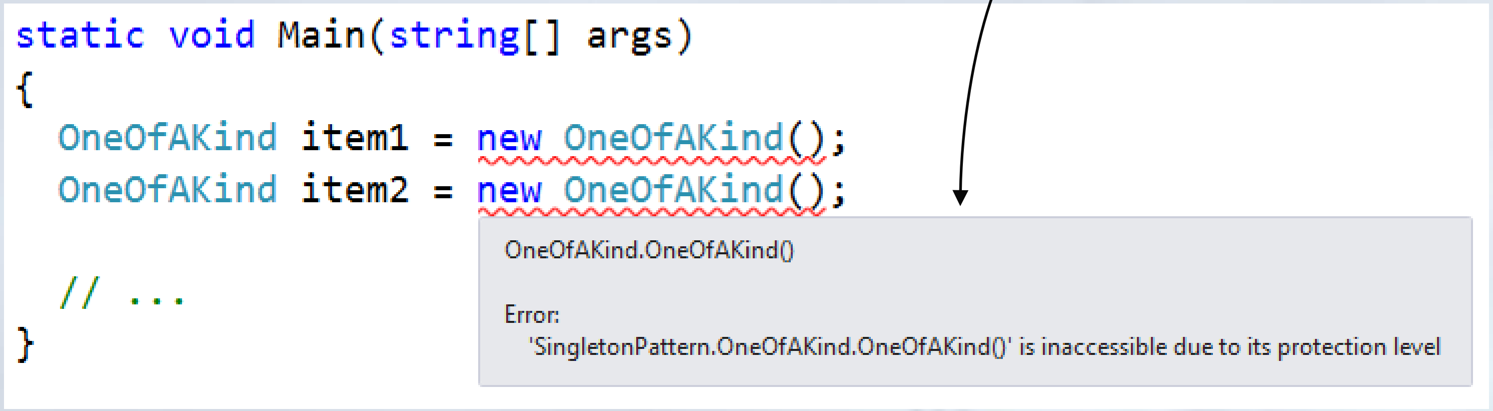
Singleton pattern

- How do we prevent multiple objects are created?!?
- With ‘new’ (constructor call) an object is created, so…
- …make the constructor <u>private</u>, but then…?
- Provide a <u>static method</u> that returns an instance

Singleton Pattern - example

```
public class OneOfAKind
{
  private OneOfAKind()
  {
    // ...
  }
}
```

The constructor is now private, so we can't create multiple objects with 'new'.



```
static void Main(string[] args)
{
  OneOfAKind item1 = new OneOfAKind();
  OneOfAKind item2 = new OneOfAKind();

  // ...
}
```

```
OneOfAKind.OneOfAKind()

Error:
  'SingletonPattern.OneOfAKind.OneOfAKind()' is inaccessible due to its protection level
```

Singleton Pattern - example

```
public class OneOfAKind
{
  private static OneOfAKind uniqueInstance;

  // constructor is not available
  private OneOfAKind() { }

  // static method that delivers a unique instance
  public static OneOfAKind GetInstance()
  {
    if (uniqueInstance == null)
      uniqueInstance = new OneOfAKind();

    return uniqueInstance;
  }
}
```

A static member contains the unique object.

Via a (public) static method the (unique) instance can be retrieved.

An instance is only created when there isn't one yet.

```
class Program
{
  static void Main(string[] args)
  {
    OneOfAKind item1 = OneOfAKind.GetInstance();
    OneOfAKind item2 = OneOfAKind.GetInstance();

    if (item1.Equals(item2))
      Console.WriteLine("items are the same");
    else
      Console.WriteLine("items are not the same");

    Console.ReadKey();
  }
}
```

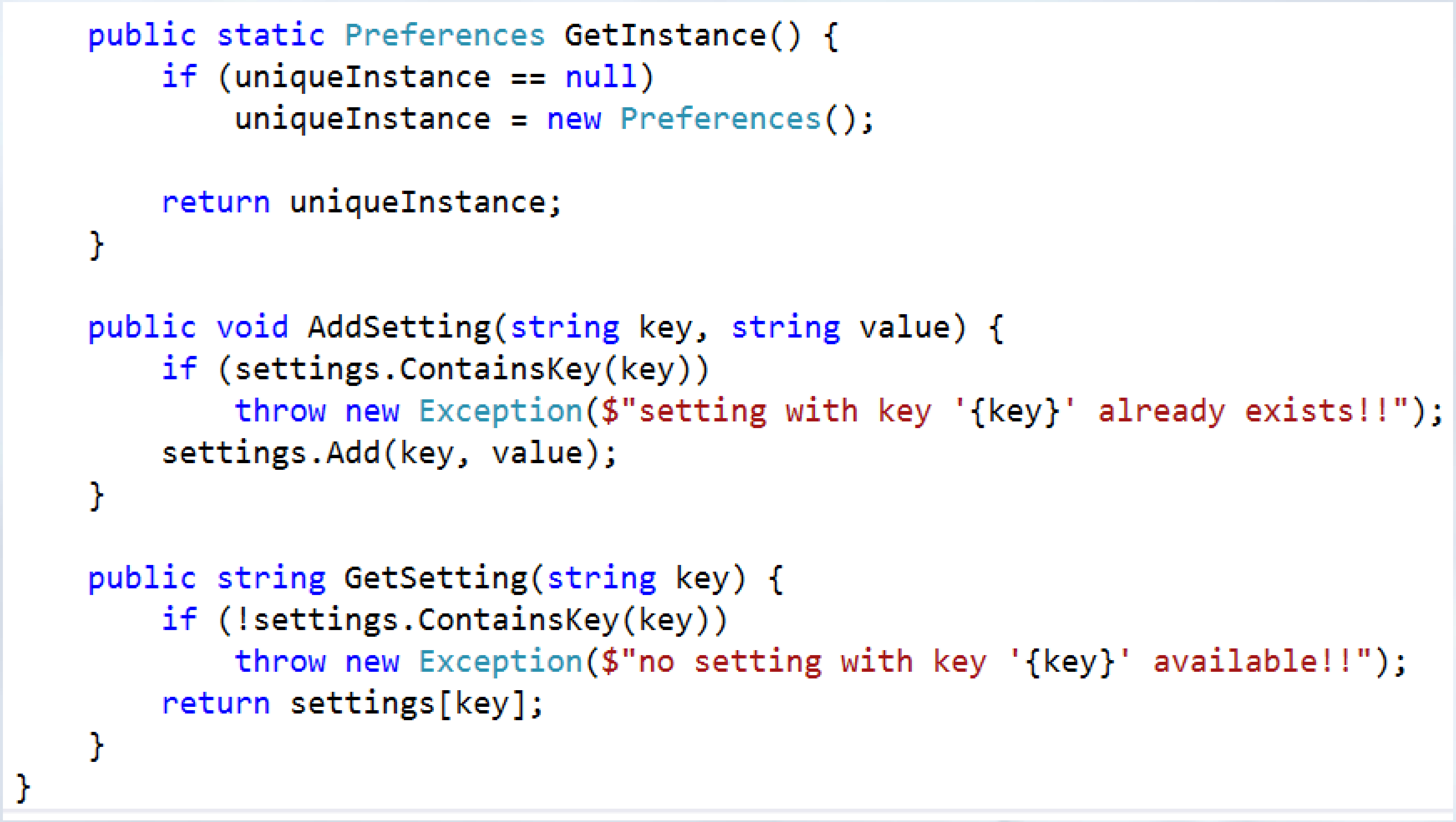
Singleton Pattern - example 2

```
public class Preferences
{
    private static Preferences uniqueInstance;
    private Dictionary<string, string> settings;

    // constructor is not available
    private Preferences()
    {
        settings = new Dictionary<string, string>();
    }

    // static method that delivers a unique instance
    public static Preferences GetInstance()
    {
        if (uniqueInstance == null)
            uniqueInstance = new Preferences();

        return uniqueInstance;
    }
```

A concrete example is 'Preferences' that must be available throughout the (entire) application.

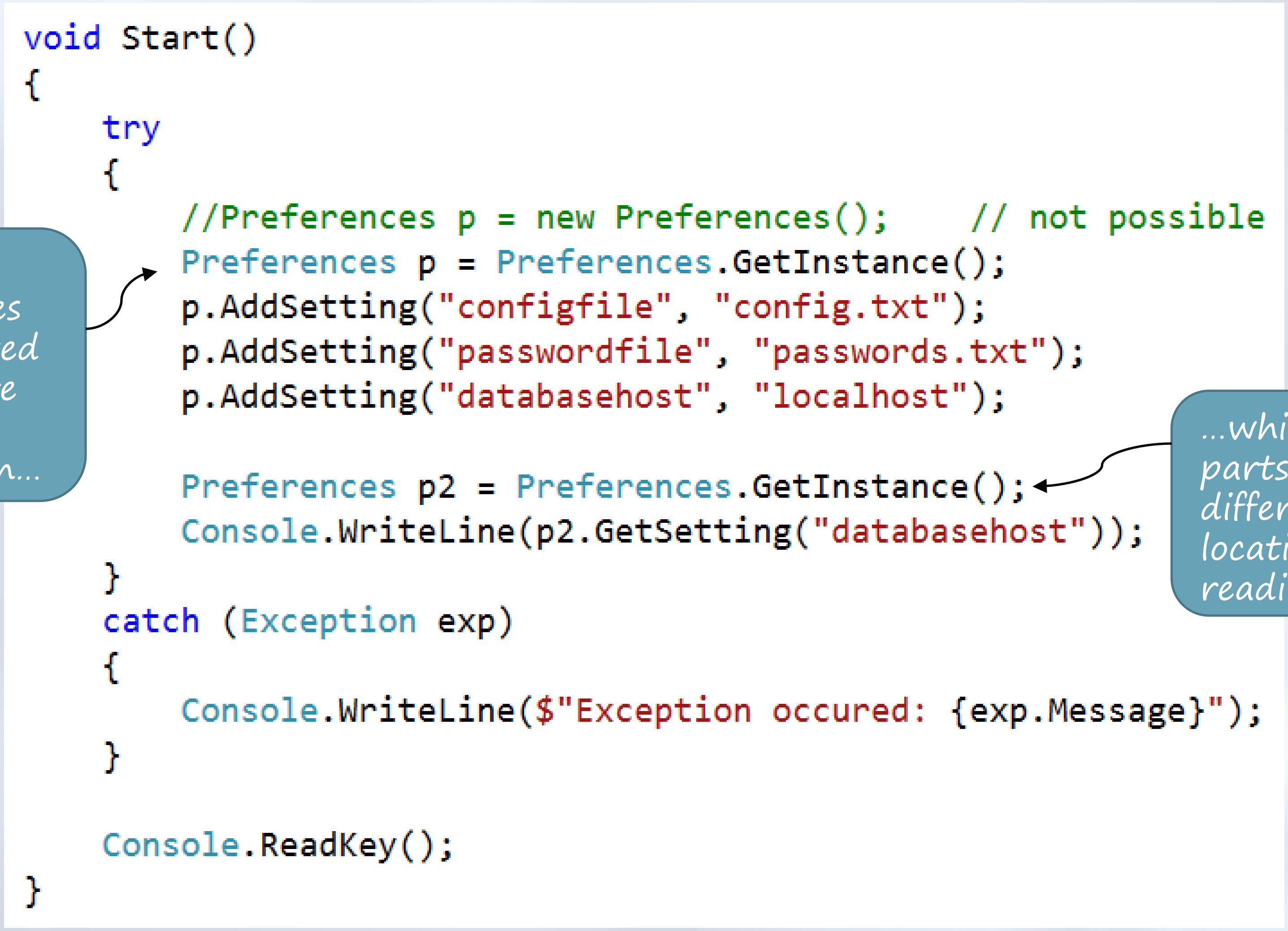
We don't want to pass such an object to all parts of the system.

Singleton Pattern - example 2



```
    public static Preferences GetInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new Preferences();

        return uniqueInstance;
    }

    public void AddSetting(string key, string value) {
        if (settings.ContainsKey(key))
            throw new Exception($"setting with key '{key}' already exists!!");
        settings.Add(key, value);
    }

    public string GetSetting(string key) {
        if (!settings.ContainsKey(key))
            throw new Exception($"no setting with key '{key}' available!!");
        return settings[key];
    }
}
```

Singleton Pattern - example 2



```
void Start()
{
    try
    {
        //Preferences p = new Preferences();    // not possible
        Preferences p = Preferences.GetInstance();
        p.AddSetting("configfile", "config.txt");
        p.AddSetting("passwordfile", "passwords.txt");
        p.AddSetting("databasehost", "localhost");

        Preferences p2 = Preferences.GetInstance();
        Console.WriteLine(p2.GetSetting("databasehost"));
    }
    catch (Exception exp)
    {
        Console.WriteLine($"Exception occured: {exp.Message}");
    }

    Console.ReadKey();
}
```

These preferences are adjusted somewhere in the application…

…while several parts (at different locations) are reading them.

State pattern

The State Pattern (GoF): *'allows an object to alter its behavior when its internal state changes. The object will appear to change its class.'*

- Behavior of an object depends on the current state of the object

- With the Strategy pattern the behavior is changed by an <u>external</u> party, with the State pattern it is changed <u>internally</u>

State pattern – an example

- Gumball machine
- Actions?
  → InsertCoin, EjectCoin, TurnHandle
- States?
  → NoCoinPresent, CoinPresent, SoldGumball, SoldOut

State pattern – an example

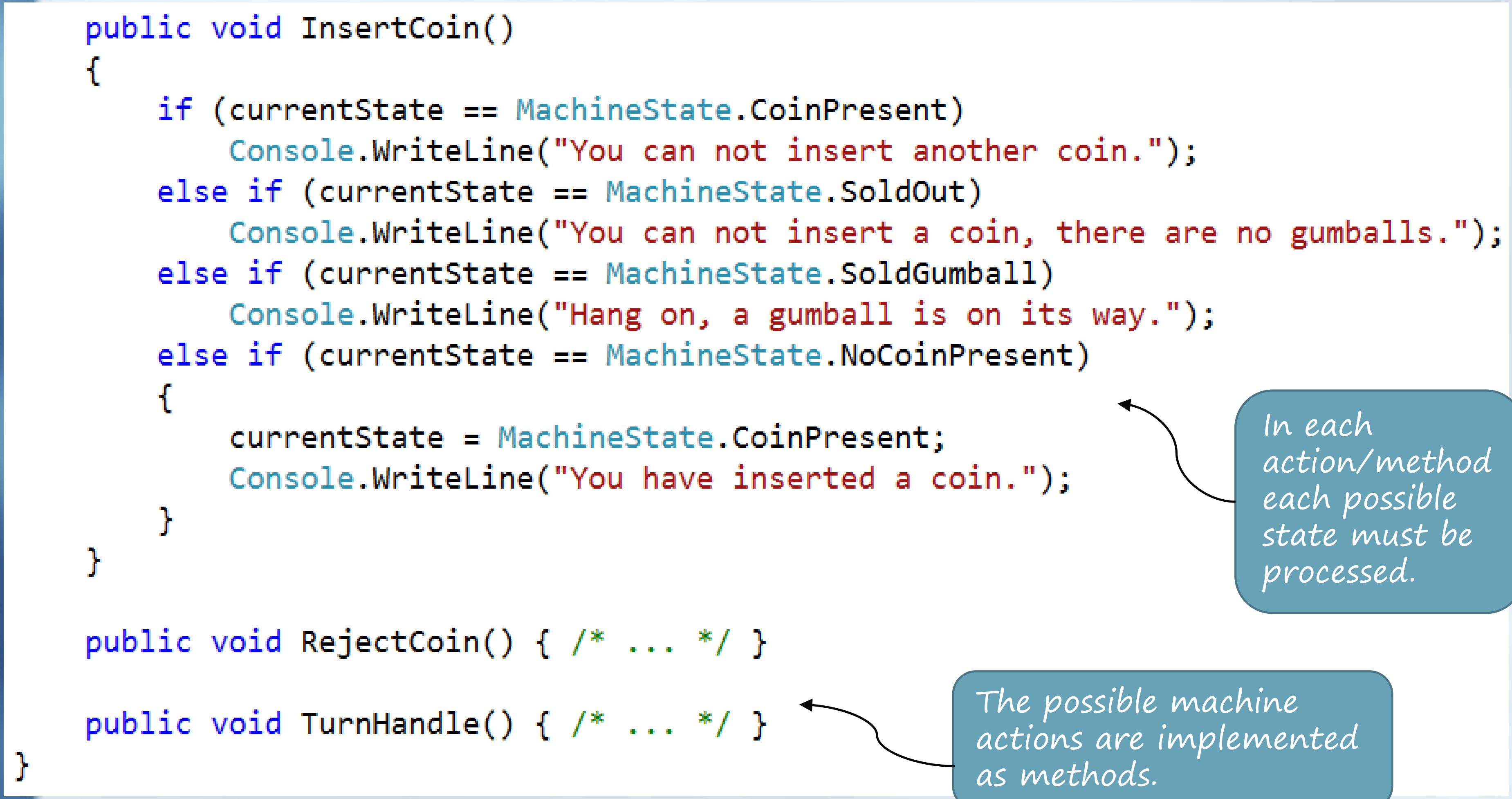
- How to implement the Gumball machine?

We can use an enumeration for the machine state.

```
class GumballMachine
{
    private enum MachineState { SoldOut, CoinPresent, NoCoinPresent, SoldGumball }
    private MachineState currentState;
    private uint numberOfGumballs;

    public GumballMachine(uint numberOfGumballs)
    {
        this.numberOfGumballs = numberOfGumballs;
        if (numberOfGumballs > 0)
            currentState = MachineState.NoCoinPresent;
        else
            currentState = MachineState.SoldOut;
    }
```

The machine gets an initial (current) state.

State pattern – an example



```
    public void InsertCoin()
    {
        if (currentState == MachineState.CoinPresent)
            Console.WriteLine("You can not insert another coin.");
        else if (currentState == MachineState.SoldOut)
            Console.WriteLine("You can not insert a coin, there are no gumballs.");
        else if (currentState == MachineState.SoldGumball)
            Console.WriteLine("Hang on, a gumball is on its way.");
        else if (currentState == MachineState.NoCoinPresent)
        {
            currentState = MachineState.CoinPresent;
            Console.WriteLine("You have inserted a coin.");
        }
    }

    public void RejectCoin() { /* ... */ }

    public void TurnHandle() { /* ... */ }
}
```

In each action/method each possible state must be processed.

The possible machine actions are implemented as methods.

State pattern – an example

- What if we want to add a new state to the gumball machine? Then we need to modify <u>all</u> actions (methods)…

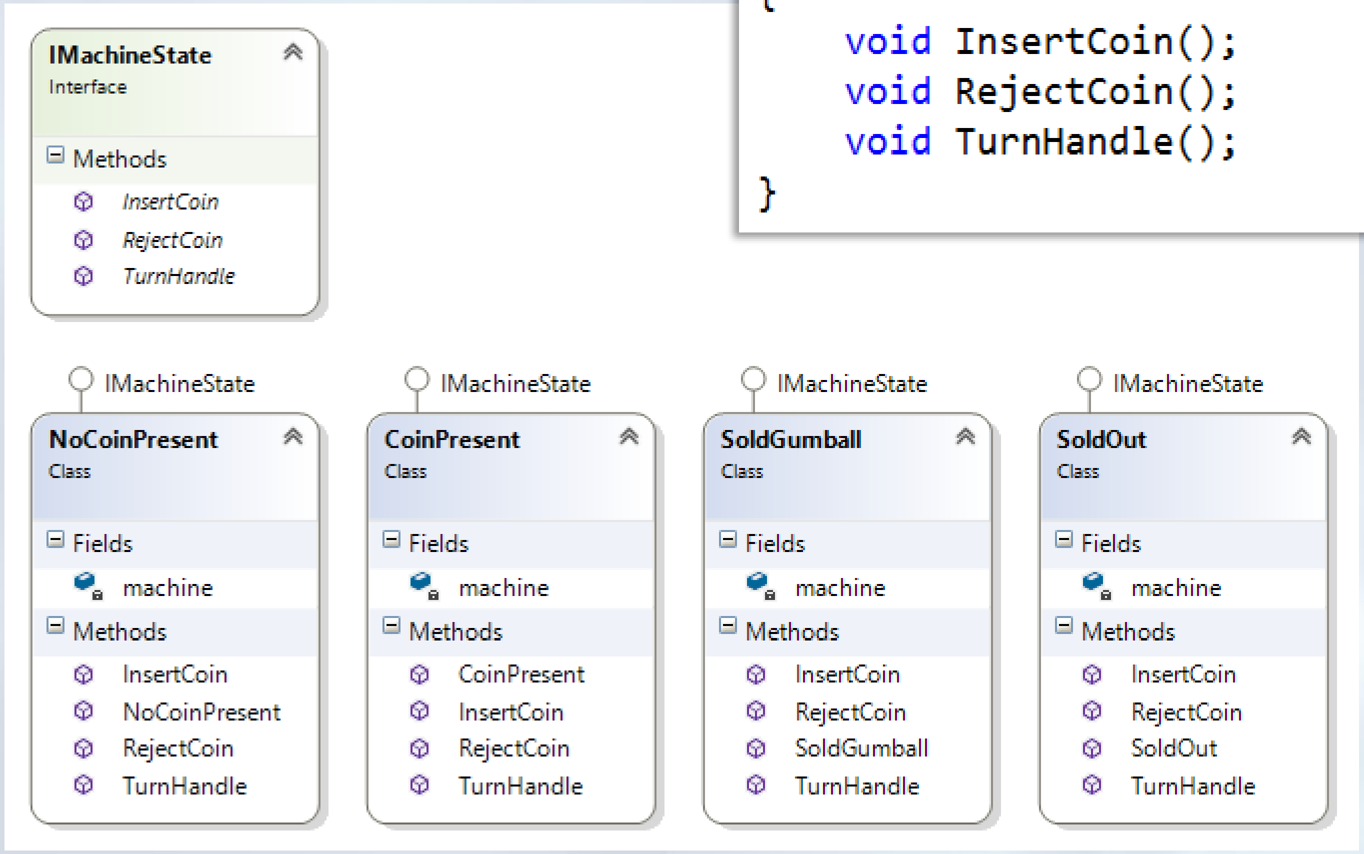
- Another approach: create a 'State' interface and create for each state a class; these 'state classes' implement the behavior for that state

State pattern – an example

- State interface contains a method for each action

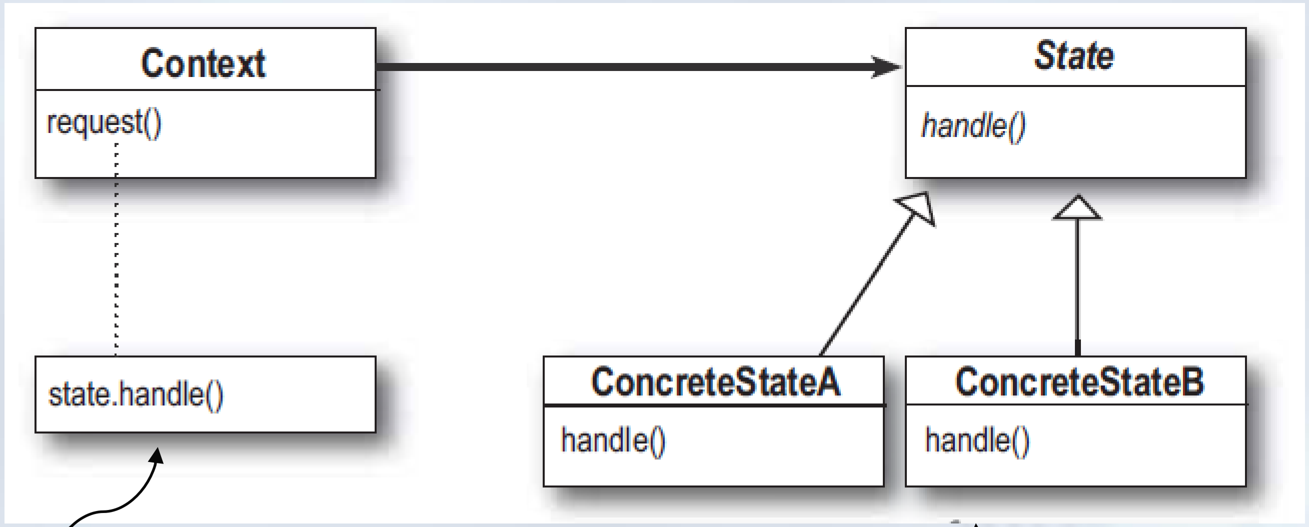
```
public interface IMachineState
{
    void InsertCoin();
    void RejectCoin();
    void TurnHandle();
}
```

Each state object can handle each possible action.



State pattern

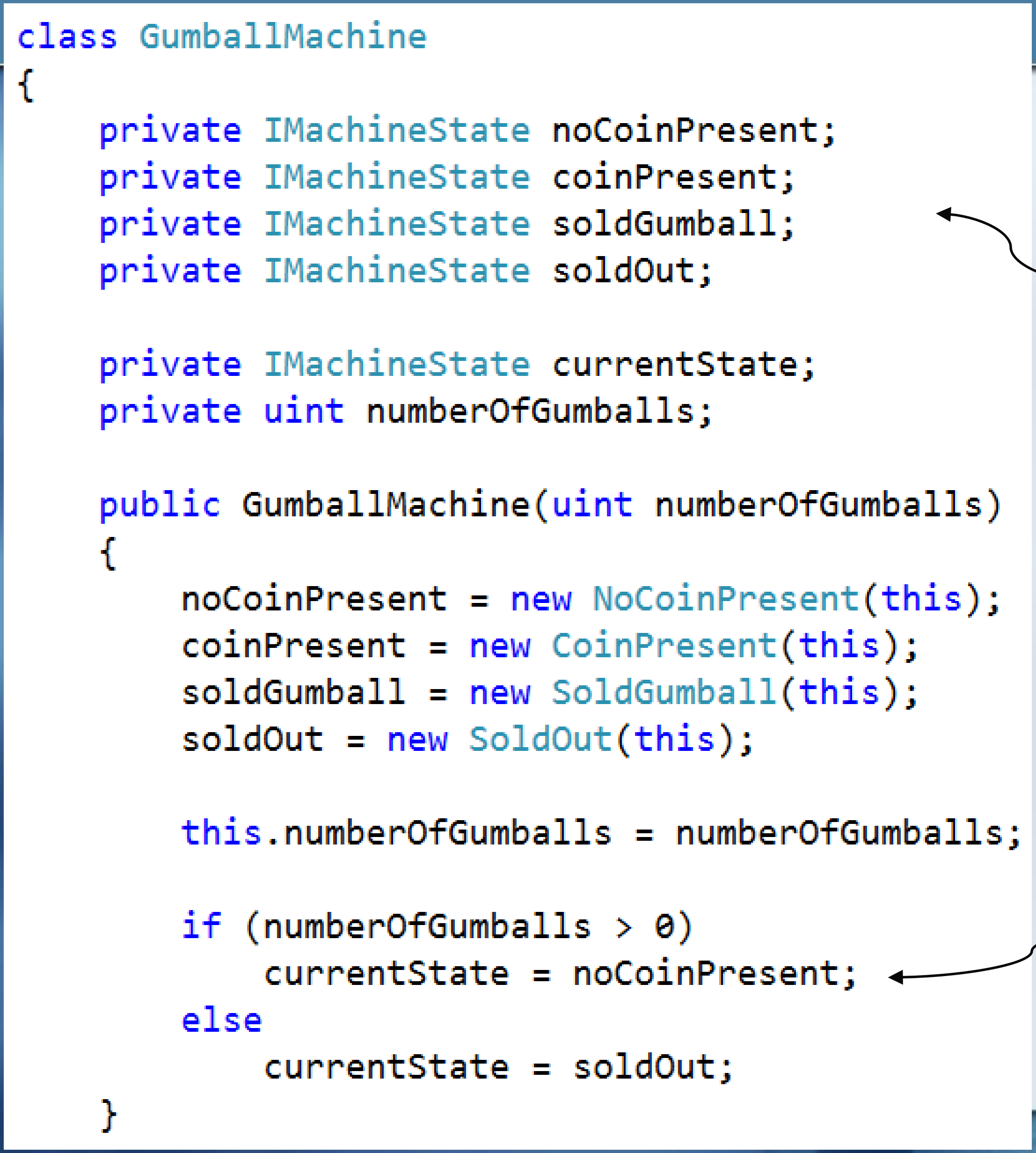
- State pattern is strongly related to Strategy pattern



Each action is delegated to a state object.

Each concrete class implement (the behavior of a) a state.

State pattern – an example



```
class GumballMachine
{
    private IMachineState noCoinPresent;
    private IMachineState coinPresent;
    private IMachineState soldGumball;
    private IMachineState soldOut;

    private IMachineState currentState;
    private uint numberOfGumballs;

    public GumballMachine(uint numberOfGumballs)
    {
        noCoinPresent = new NoCoinPresent(this);
        coinPresent = new CoinPresent(this);
        soldGumball = new SoldGumball(this);
        soldOut = new SoldOut(this);

        this.numberOfGumballs = numberOfGumballs;

        if (numberOfGumballs > 0)
            currentState = noCoinPresent;
        else
            currentState = soldOut;
    }
```

No enumerations anymore, but separate state objects.

One of the state objects is the current one.

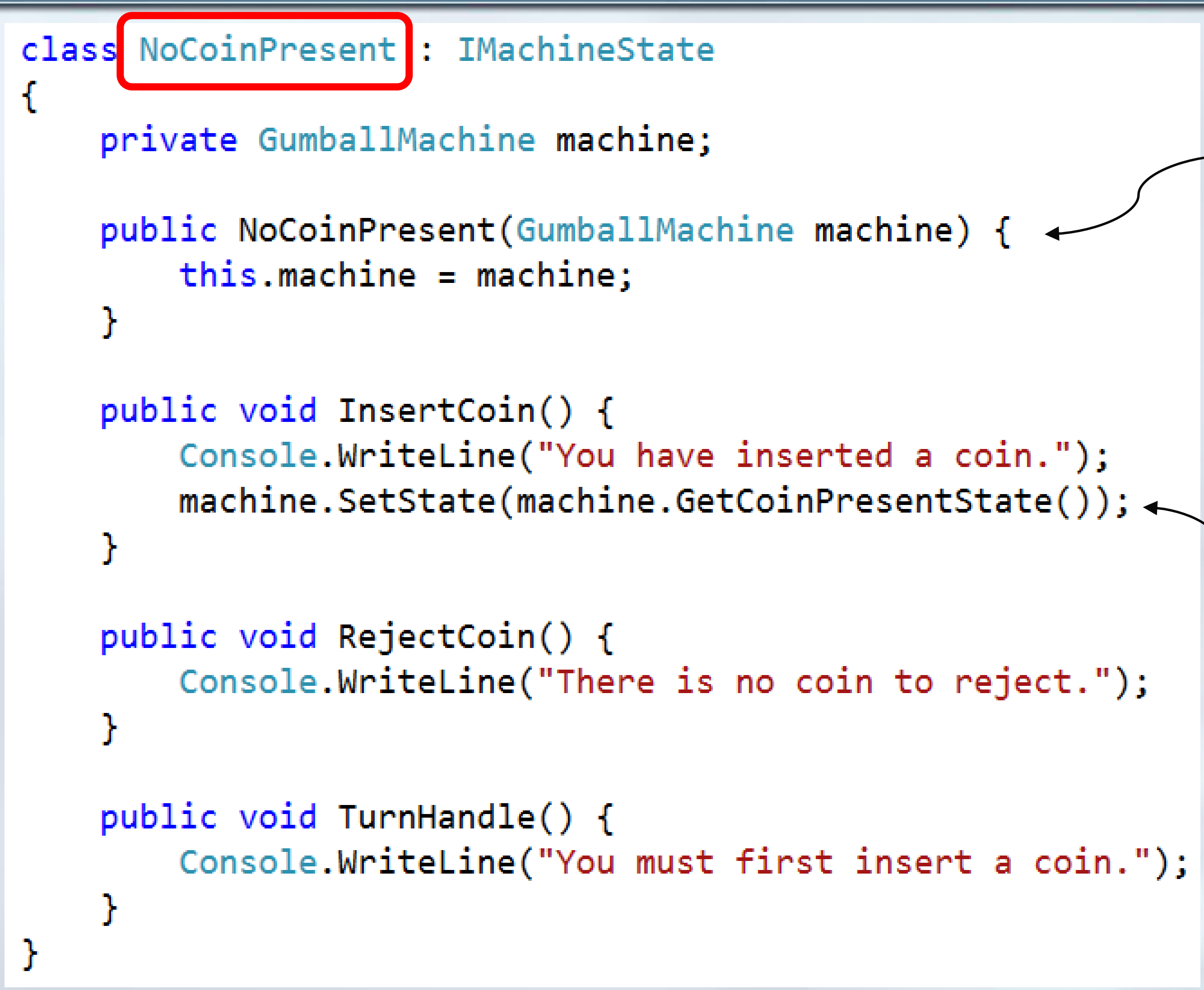
State pattern – an example

```
    public void InsertCoin() {
        currentState.InsertCoin();
    }

    public void RejectCoin() {
        currentState.RejectCoin();
    }

    public void TurnHandle() {
        currentState.TurnHandle();
    }

    public void SetState(IMachineState newState) {
        currentState = newState;
    }

    public IMachineState GetCoinPresentState() {
        return coinPresent;
    }

    // ...
}
```

The actions are delegated to the current state object.

These methods are needed to change the state (called by one of the state objects).

State pattern – an example



```
class NoCoinPresent : IMachineState
{
    private GumballMachine machine;

    public NoCoinPresent(GumballMachine machine) {
        this.machine = machine;
    }

    public void InsertCoin() {
        Console.WriteLine("You have inserted a coin.");
        machine.SetState(machine.GetCoinPresentState());
    }

    public void RejectCoin() {
        Console.WriteLine("There is no coin to reject.");
    }

    public void TurnHandle() {
        Console.WriteLine("You must first insert a coin.");
    }
}
```

The machine is passed as a constructor parameter.

The state of the machine is changed by the state objects.

Summary

- Singleton Pattern: ensures that there is only one unique instance of a class
  (and provides a global point of access to it)

- State Pattern: implement behavior in several classes; behavior depends on the (current) state

Assignments

- Moodle: ‘Week 5 assignments’