# Overview:

* The State design pattern allows an object to alter its behavior when its internal state changes
  + Object will appear to change its class.
  + An objects behavior is the result of the function of its state:
    - Behavior gets changed at runtime depending on the state (polymorphism)
* Normally, if we have to change behavior of an object based on its state, we have a state variable in the object and use if-else condition to perform different actions based on the state.
  + With the state pattern, we can remove dependencies on if/else or switch/case conditional logic.
* We create objects, which represent various states, and a context object whose behavior varies as its state object changes.
* Used to provide a systematic and loosely coupled way to achieve state change through Context and State implementations.
* Is very similar to Strategy Pattern, which we will discuss, in a future lecture.

# Examples:

* A traffic signal:
  + Stop, go , and slow down states
* Consider a TCP network connection:
  + An object that is responsible for communication can be in various states
    - Established, listening, and closed.
  + When a TCP Connection object receives requests from other objects, it responds differently depending on its current state.
    - E.g., an Open request depends on whether the connection is in its closed state or its Established state.
* How about a job processing application where we can process only one job at a time.
  + If a new job appears, either the application will process that job (accepted).
  + Alternatively, it will signal that the new job cannot be processed at this moment because the system is already processing the maximum number of jobs in it (queued).

# When to use the state pattern?

* When an object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.
* When operations have large, multipart conditional statements that depend on the object’s state.
  + State is usually represented by one or more enumerated constants
  + Often, several operations will contain the same conditional structure
    - State pattern puts each branch of the conditional in a separate class.
    - Let’s you treat the object’s state as an object in its own right that can vary independently from other objects.

# Advantages:

* Puts all behavior associated with a state into one object.
  + Improves cohesion.
* Allows state transition logic to be incorporated into state object rather than in monolithic if or switch statement.
* Helps avoid inconsistent states since state changes occur by rebinding one variable rather than several.
* Very easy to add more states for additional behavior.
  + Makes code more robust, easily maintainable and flexible.
* One drawback is that the pattern does increase the number of objects (one of each state).

# Implementation:

# Participants:

* **Context:**
  + Defines the interface of interest to clients
  + Maintains an instance of a **ConcreteState** subclass that defines the current state.
* **State:**
  + Defines an interface for encapsulating the behavior associated with a particular state of the **Context**.
* **ConcreteState** subclasses:
  + Each subclass implements a behavior associated with a state of the Context.

# Workflow:

* The **Context** delegates state-specific requests to the current **ConcreteState** object
  + Has a reference to the current state object.
* A context may pass itself as an argument to the State object handling the request
  + Let’s the State object access the context if necessary.
* Clients can configure a context with State objects:
  + Once a context is configured, its clients do not have to deal with the State objects directly.
* Either **Context** or the **ConcreteState** subclasses can decide which state takes priority over another under what circumstances.

# Advantages of implementation:

* As mentioned previously, this pattern puts all behavior associate with a particular state into one object:
  + New states and transitions can be added easily by defining new subclasses
* Large conditional statements are undesirable
  + Monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend.
  + State pattern offers a better way to structure state-specific code
  + Logic that determines the state transitions does not reside in monolithic if or switch statements but instead is partitioned between the State subclasses.
* Makes state transitions explicit by introducing separate objects for different states
* State objects can protect the Context from inconsistent internal states
  + State transitions are atomic from the Context’s perspective
    - Happen by rebinding one variable (the Context’s State object variable).
* State objects can be shared
  + If state objects have no instance variables.

# Summary

* Allows an object to have many different behaviors that are based on its internal state.
* Represents state as an object.
* The context gets its behavior by delegating to the current state object it is composed with
* By encapsulating each state into a class, we localize any changes that will need to be made
* State transitions can be controlled by the State classes or by the Context classes.