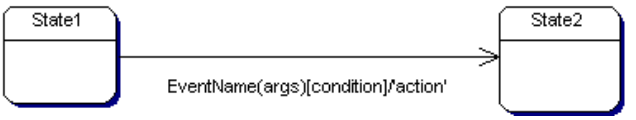
Implementing state diagrams

- State diagrams are a tool to allow us to describe the non-trivial internal changes that an *individual object* goes through in response to external "events"
  - The events may be GUI events or, frequently, messages received from other objects
  - So: A state diagram is associated with a *specific class*
- Bruegge & Dutoit define: "A state is a condition satisfied by the attributes of an object"
  - A object may have a conceptual "lifecycle" with specific situations (states) and steps between them (transitions)
  - At any moment the object will be in precisely one state
  - We use the *values* of one or more of the object's *attributes* (instance variables) to record/determine the state
  - We can tell *which* state an object is in by checking the values of its attributes

- The receipt of a message (a call on one of the object's public operations):
  - May cause a change in the attribute values that record the state,
  - Thus causing a *transition* from one state to another
- Some public operations may *only* be applicable in *some* states of the object
  - Either the caller must check the state before calling the operation ("design by contract")
  - Or each such operation must internally check whether it should act before doing anything
  - "Preconditions"
- A transition has (all optional!):
  - An event name
  - Event arguments
  - Guard condition
  - An action to be taken

- Implementing a transition:
  - The pattern for a typical transition in Class1 with all optional components present:



  - Typical implementation: a method in Class1:

```
public void eventName(args) {
  // First check precondition
  boolean OK = currently in State 1 && condition;
  if ( !OK )
    return; // Ignore and remain in State 1
  // Precondition OK, so:
  action;   // Do something (so move to State 2)
}
```

  - In "design by contract" we might only have:

```
if ( ! condition ) ...
```
 or no `if` at all

- In general, the same event name may appear on transitions from several states, with different conditions and/or actions:
  - The "precondition" test must be more complex, e.g:

```
public void eventName(args) {
  boolean OK =
        (currently in State 1 && condition 2)
     ||(currently in State 4 && condition 3)
     || ... );
  if ( !OK )    // Check precondition
    return; // Ignore, remain in current state
  if (in State 1 && condition 2)
    action 2;  // Do something (so change state)
  else
  if (in State 4 && condition 3)
    action 3;  // Do something (so change state)
  else ...
}
```

  - Can be simplified, as on next slide

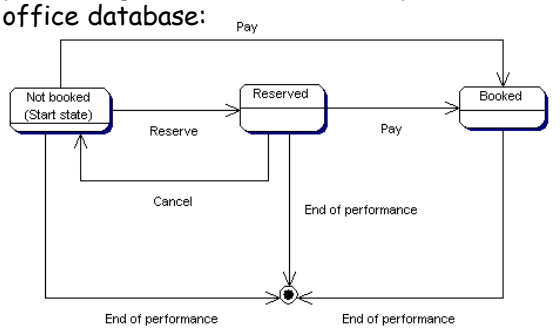
- A simplified structure for the general operation on previous slide – avoids re-computing the precondition components:

```
public void eventName(args) {
  if (in State 1 && condition 2)
    action 2;  // Do something (so change state)
  else
  if (in State 4 && condition 3)
    action 3;  // Do something (so change state)
  else
  ...
  // No actions taken if precond is false
}
```

- Could even be more general/flexible

Example

- A class representing a theatre seat for a particular event, in a booking office database:



- Implementation requirements:
  - Sufficient attributes to represent the states (others are allowed too)
  - Operations/methods for reserve, cancel and pay
  - The methods check the state before acting

- Possible implementation: an `int` state code variable:

```
public class Seat {
  public final int NOTBOOKED = 0;  // Constants
  public final int RESERVED = 1;   // for state
  public final int BOOKED = 2;     // codes

  private int state = NOTBOOKED;   // Initially

  public void reserve() {
    if (state != NOTBOOKED)
      return;       // Ignore the feedback issue!
    state = RESERVED;
  }

  public void cancel() {
    if (state != RESERVED) return;
    state = NOTBOOKED;
  }

  public void pay() {
    if (state == BOOKED) return;
    state = BOOKED;
  }
```

continued…


```
  public int getState() {
    return state;
  }
}
```

- Notes:
  - The constants are public so that client objects can obtain and check the state for themselves (via **`getState`**)
  - Using a Java **`enum`** would be better (but equivalent)
  - There may be other attributes (eg price, seat location, customer name, etc)
  - Most of the methods need parameters to bring information (eg the customer's name)
  - Any special "failure" feedback is ignored here
  - The state variable is *private* to prevent malicious "tweaking"
  - In "design by contract" **`getState`** is vital to enable clients to check before sending messages

- Another possible implementation – different state encoding:
  - Two boolean variables: `reserved`, `paidFor`
  - So four possible states, but only three used:

State	`reserved`	`paidFor`
Not booked	`false`	`false`
Reserved	`true`	`false`
Booked	`true`	`true`

```
public class Seat {

  private boolean reserved = false;  // Initially
  private boolean paidFor = false;   // not booked
  public void reserve() {
    if (reserved) return;
    reserved = true;
  }
```

continued…


```
  public void cancel() {
    if (!reserved || paidFor) return;
    reserved = false;
  }
  public void pay() {
    if (paidFor) return;
    paidFor = true;
    reserved = true;
  }
  public boolean isReserved() {
    return reserved;
  }
  public boolean isPaidFor() {
    return paidFor;
  }
}
```

Note about creation and destruction

- If a state diagram has an action on the transition from the *start* symbol to the *initial state*:
  - That action is placed in the class's *constructor*
- Java objects have no "destroy yourself" method to be called on the transition to the *stop state*
- The "death" of an object will usually be when the *last reference to that object* is "dropped" by the owning object(s) in the system
- If there *is* some specific action to be taken on the transition to the stop state (e.g releasing/tidying up resources):
  - Could have a `public void destroy()` method to be called by the object dropping the *last* reference – but this can be hard to determine!
  - A `public void finalize()` method will be called *automatically by the JVM garbage collector* – but we cannot control if/when!

End of lecture