Chapter 11

Modeling and Simulation

What is in This Chapter ?

This chapter explains the concepts of modeling and simulation with some examples of how we can do this in JAVA. We begin with a simple example of modeling a traffic light that uses a state machine and then discuss how we can model some objects to simulate some interactions between objects that we often see in the real world.

11.1 Simulation With State Machines

Computers are often used to simulate real-world scenarios. Here is a wikipedia definition:

> *A **computer simulation** is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behaviour of the system.*

Computer simulation has become a useful part of modeling many natural systems in the areas of physics, chemistry, biology, economics, social science and engineering.

Video games are prime examples of simulation, where some real (or imaginary) life situations are simulated in a virtual world. As time progresses, video games are becoming more life-like as graphics and physical modeling become more precise and realistic.

Other places that require computer simulation are:

- network traffic analysis
- city/urban simulation
- flight/space/vehicle/medical simulators
- disaster preparedness simulations
- film and movie production
- theme park rides
- manufacturing systems

There is much to know about simulation, enough material to fill a few courses. In this course, however, we will just address two basic categories of simulations:

- Running a simulation to find an answer to a problem
- Virtual simulation (i.e., animation) of a real world scenario

When simulating, usually there is some kind of ***initialization*** phase, followed by a ***processing loop*** that continuously processes and (possibly) displays something on the screen.

Think of a simulation as a store. The ***initialization*** phase corresponds to the work involved in getting the store ready to open in the morning (e.g., tidying up, putting out signs, stocking the shelves, preparing the cash register, unlocking the door). The ***processing loop*** phase corresponds to the repeated events that occur during the day (i.e., dealing with customers one-by-one) until the store is to be closed.

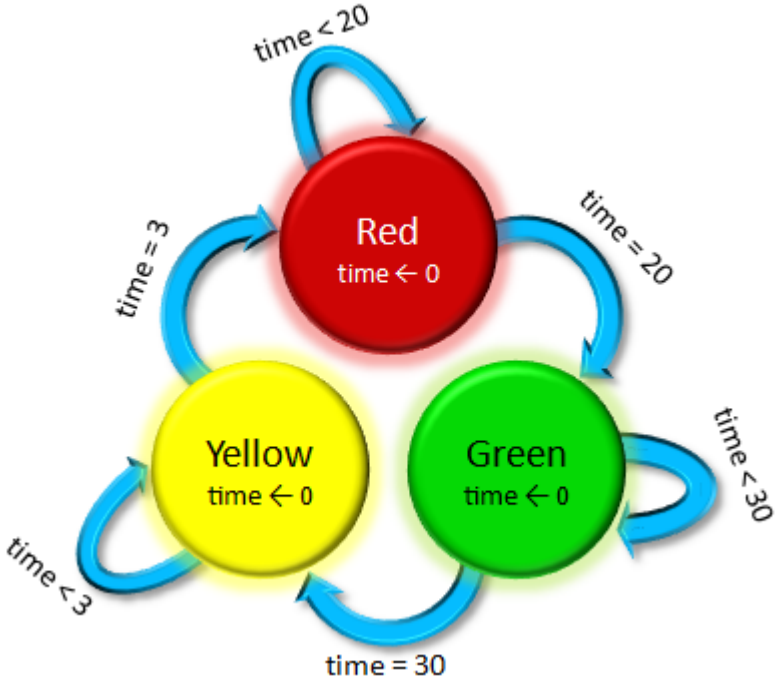
Example:

In the real world objects have what is known as ***state***. The state of an object is normally considered to be some condition of the object with respect to a previous state. For example, a light bulb is considered to be in a "*working*" state when we buy it, but if we smash it on the ground it would then be in a "*broken*" state.

How can we simulate a traffic light ? It should have 3 states ... RED, GREEN and YELLOW. Assume that the traffic light starts in a RED state and that we want it to cycle continuously between these states. We will assume that the light remains RED for 20 seconds, then GREEN for 30 seconds, then YELLOW for 3 seconds. To simulate the traffic light, it is good to think of it as a state machine.

> *A **state machine** is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change.*

We can then draw a ***state diagram*** to show how the traffic light changes from one state to another as time goes by:



Notice how the state changes from RED to GREEN only when the time has reached 20 seconds. Note as well that inside the state of GREEN, we reset the time counter to 0 so that we can count 30 seconds again in order to decide when to switch to the yellow state.

Let's represent the traffic light as an object. We will need to maintain the state/color of the light as well as perhaps a countdown timer that keeps track of how much longer the light will remain in that state. It will also be a good idea to create a constructor. We can create one that makes sure that the light starts in a state of RED. Lastly, we will create a **toString()**

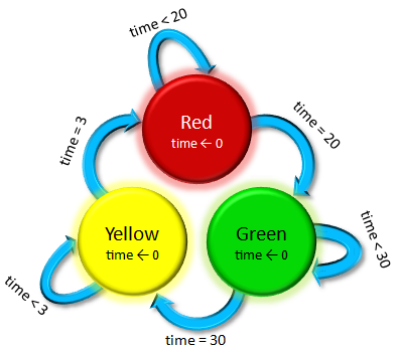
method that will allow our traffic light to display itself properly. Make sure that you understand the following code, as there should be nothing new here:

```
public class TrafficLight {
    public static final byte    RED = 1;
    public static final byte    YELLOW = 2;
    public static final byte    GREEN = 3;

    public int    state;            // The current state of the traffic light
    public int    timeRemaining;    // Seconds remaining in this state

    public TrafficLight(byte  iState) {
        state = iState;
        timeRemaining = 0;
    }

    public String toString() {
        switch(state) {
            case RED:    return "RED traffic light";
            case YELLOW: return "YELLOW traffic light";
            case GREEN:  return "GREEN traffic light";
            default:     return "OFF traffic light";
        }
    }
}
```

Now, for the simulation program that will simulate the light ... we will need to create the light and then simulate it by allowing it to go through the states repeatedly. Here is the structure:

```
public class TrafficLightSimulationProgram {
    public static void main(String[] args) {
        TrafficLight light = new TrafficLight(TrafficLight.RED);

        System.out.println(light);
        while(true) {
            // ... Add code here to simulate ...
        }
    }
}
```

Now, to simulate the code, we just need to make the **timeRemaining** value of the **TrafficLight** count down from whatever it is at ... to zero. Then, we switch its state. We just have to switch at the right time and to the right color. So, we can write code that follows the state machine diagram.




```
if (timeRemaining is 0) then {
        if (state is Red) then {
                state ← Green
                timeRemaining ← 30
        }
        otherwise  if (state is Green) then {
                state ← Yellow
                timeRemaining ← 3
        }
        otherwise  if (state is Yellow) then {
                state ← Red
                timeRemaining ← 20
        }
}
```

Here is the final code. Notice how the 1 second delay is added to ensure that the changing of the states is correct:

```
public class TrafficLightSimulationProgram {
    public static void main(String[] args) {
        TrafficLight light = new TrafficLight(TrafficLight.RED);

        System.out.println(light);
        while(true) {
            try{ Thread.sleep(1000); } catch(Exception e){}; // Wait for 1 second
            if (light.timeRemaining == 0) {
                if (light.state == TrafficLight.RED) {
                    light.state = TrafficLight.GREEN;
                    light.timeRemaining = 30;
                }
                else if (light.state == TrafficLight.GREEN) {
                    light.state = TrafficLight.YELLOW;
                    light.timeRemaining = 3;
                }
                else if (light.state == TrafficLight.YELLOW) {
                    light.state = TrafficLight.RED;
                    light.timeRemaining = 20;
                }
                System.out.println(light);            // Display the changed light
            }
            else
                light.timeRemaining--;                // Count down each time
        }
    }
}
```

For some added fun, try writing a program to draw the traffic light as it changes state. You can start with the code from our Ball Simulation.

11.2 Modeling and Simulation Using Data Structures

Sometimes it is helpful to develop a model of some real world objects in order to simulate something. It helps to understand the simulation when we can think of all the objects involved in the same way that we would think of objects in the real world. So, for example, all objects have ***state*** (which is the attributes that make the object different from other objects of the same kind (e.g., name, size, color, age, etc...). Objects also have ***behavior*** (which is the things that you can do with the object ... its abilities).

Consider modeling a theatre in which patrons purchase tickets for movies from a box office. We can create object models to represent a **Patron**, a **Theatre** , and a **Ticket** for a movie ... and then make a simulation program that tests them altogether in a realistic kind of scenario.

To begin, we will define a **Ticket** object with a single instance variable indicating which movie theatre that the **Ticket** is valid for. Let's assume that the id is a char (e.g., 'A', 'B', 'C', etc). We will also make some decisions as to how much the tickets will cost and hardcode this as

constants in the class. Here is the basic class. Note that the theatreID represents the ID of the theatre for which the ticket is valid:

```
public class Ticket {
    public static final float REG_CHILD_PRICE = 7.99f;
    public static final float REG_ADULT_PRICE = 10.99f;
    public static final float REG_SENIOR_PRICE = 8.50f;

    public char    theatreID;   // ID of the theatre to which it is valid

    public Ticket(char id) {
        theatreID = id;
    }
}
```

Now, let us define a **Patron** object with instance variables called **age** (which is an **int**) and **ticket** (which is a **Ticket** object). We will write a simple **toString()** method as well that shows the **Patron** with the following format, depending on whether or not he/she has a ticket:

```
"19 yr old patron without a ticket"
"19 yr old patron with a ticket for theatre A"
```

Here is the code:

```
public class Patron {
    public int       age;
    public Ticket    ticket;

    public Patron(int anAge) {
        age = anAge;
        ticket = null;
    }

    public String toString() {
        String result = age + "yr old patron with";
        if (ticket == null)
            return result + "out a ticket";
        return result + " a ticket for theatre " + ticket.theatreID;
    }
}
```

We can test these two objects out with the following test program:

```
public class PatronTestProgram {
    public static void main(String args[]) {
        Patron jenny = new Patron(19);  // create 19 year old patron with no ticket
        Ticket aTicket = new Ticket('A'); // create a ticket for Theatre A

        jenny.ticket = aTicket;           // give the ticket to jenny
        System.out.println(jenny);        // make sure age is 19 and theatreID is A
        jenny.ticket = null;              // discard jenny's ticket
        System.out.println(jenny);        // make sure theatreID is gone now
    }
}
```

Here is the output:

```
19yr old patron with a ticket for theatre A
19yr old patron without a ticket
```

Now we can define a **Theatre** class that represents the place that will be showing movies. Here is the basic code for the class, showing the instance variables that we want:

```
public class Theatre {
    public char    id;              // ID of theatre
    public String  moviePlaying;    // title of movie currently playing
    public float   movieEarnings;   // amount of money movie has made so far
    public int     capacity;        // # people that can be seated in theatre
    public int     seatsSold;       // # seats sold for movie so far
    public int     admittedPatrons; // # patrons admitted into theatre

    public Theatre(char identification, int cap) {
        moviePlaying = "";
        movieEarnings = 0;
        capacity = cap;
        seatsSold = 0;
        id = identification;
        admittedPatrons = 0;
    }
}
```

Now, to simulate something interesting, we can pretend that some patrons will be buying some tickets for some movies. We need to think about what happens in real life. Here are things that we need to simulate:

- A new movie begins playing at a **Theatre**
- A **Patron** buys a ticket for a movie
- A **Patron** enters the **Theatre** and gives his ticket
- a **Patron** returns a ticket for a movie (before it starts) for a refund

Those are typical situations that can arise. So we need to simulate each of them. We can do this with 4 individual procedures/methods:

- `public static void openMovie(Theatre aTheatre, String newMovie) {...}`
- `public static void buyTicket(Patron aPatron, Theatre aTheatre) {...}`
- `public static void admit(Patron aPatron, Theatre aTheatre) {...}`
- `public static void returnTicket(Patron aPatron, Theatre aTheatre) {..}`

For each of these methods, we will need to do two things:

1. Look at what objects/data are available
2. Decide how the objects/data will change

Examining the first method for a movie opening, we notice that a theatre is passed in as a parameter as well as a movie name. We need to determine how the Theatre will *change* as a result of the movie's opening. To do that, we need to decide how the attributes/instance variables will change. Let's look at them ... which ones will change ?:

aTheatre: id, moviePlaying, movieEarnings, capacity, seatsSold, admittedPatrons

Well, the **moviePlaying** will change to the movie **newMovie** that is passed in to the method. Also, we should ensure that the earnings for the movie are 0 as well as the number of seats sold so far and the number of admitted patrons. Here is the code:

```
public static void openMovie(Theatre aTheatre, String newMovie) {
    aTheatre.moviePlaying = newMovie;
    aTheatre.movieEarnings = 0;
    aTheatre.seatsSold = 0;
    aTheatre.admittedPatrons = 0;
}
```

There. We have the code that will simulate the opening of a movie. It was not difficult. Now, for the second one. Let's simulate the purchasing of a ticket by a patron for a specific movie theatre. The **buyTicket()** method has two objects now ... a **Patron** and a **Theatre**, so we need to see how these will change. Let's look again at the attributes for these objects:

aPatron: age, ticket
aTheatre: id, moviePlaying, movieEarnings, capacity, seatsSold, admittedPatrons

When the patron buys a ticket for the movie playing in the theatre, the patron will need to store the ticket (i.e., to keep it). But there is no ticket passed as a parameter. That is because we need to create the ticket by calling the constructor. Here is how to start this:

```
public static void buyTicket(Patron aPatron, Theatre aTheatre) {
    Ticket  t = new Ticket(aTheatre.id);    // make the ticket
    aPatron.ticket = t;                     // store the ticket
}
```

As for the theatre, what changes ? The **movieEarnings** will increase depending on the amount of money paid for the ticket ... that is ... depending on the age of the **Patron** who purchased it and the fixed prices in the **Ticket** class. The **seatsSold** will also increase. But wait! We need to also ensure that the theatre is not full to capacity before selling the ticket. Here is the completed method:

```
public static void buyTicket(Patron aPatron, Theatre aTheatre) {
    if (aTheatre.seatsSold < aTheatre.capacity) {
        Ticket  t = new Ticket(aTheatre.id);    // make the ticket
        aPatron.ticket = t;                     // store the ticket

        // Now increase the earnings
        if (aPatron.age <= 12)
            aTheatre.movieEarnings += Ticket.REG_CHILD_PRICE;
        else if (aPatron.age >= 65)
            aTheatre.movieEarnings += Ticket.REG_SENIOR_PRICE;
        else
            aTheatre.movieEarnings += Ticket.REG_ADULT_PRICE;

        aTheatre.seatsSold++;       // Add 1 to the count of seats sold
    }
}
```

Now, what about allowing someone into the theatre to watch the movie ? Well, let's look again at the attributes for these objects:

aPatron: age, ticket
aTheatre: id, moviePlaying, movieEarnings, capacity, seatsSold, admittedPatrons

The patron's ticket will be given to the theatre attendant, so we need to remove the ticket from the patron. As for the theatre, the number of admitted patrons will increase by one:

```
public static void admit(Patron aPatron, Theatre aTheatre) {
    aPatron.ticket = null;
    aTheatre.admittedPatrons++;
}
```

However, we better make sure that the patron goes to the correct theatre. So, if the patron attempts to enter a theatre for which he/she does not have a ticket, we should not allow it:

```
public static void admit(Patron aPatron, Theatre aTheatre) {
    if ((aPatron.ticket != null) &&
        (aPatron.ticket.theatreID == aTheatre.id)) {

        aPatron.ticket = null;
        aTheatre.admittedPatrons++;
    }
}
```

Do you understand why we needed to check for a **null** ticket ? If we didn't, then the code would crash for anyone who did not have a ticket.

Our final method is for handling the returning of a ticket. To do this, we simply need to take the ticket away from the patron and take the price of the ticket away from the theatre's earnings. Of course, we'd better ensure that the patron actually has a ticket and that it matches the theatre passed in, otherwise we cannot properly return the ticket.

```
public static void returnTicket(Patron aPatron, Theatre aTheatre) {
    if ((aPatron.ticket != null) &&
        (aPatron.ticket.theatreID == aTheatre.id)) {

        aPatron.ticket = null;

        // Now decrease the earnings
        if (aPatron.age <= 12)
            aTheatre.movieEarnings -= Ticket.REG_CHILD_PRICE;
        else if (aPatron.age >= 65)
            aTheatre.movieEarnings -= Ticket.REG_SENIOR_PRICE;
        else
            aTheatre.movieEarnings -= Ticket.REG_ADULT_PRICE;

        aTheatre.seatsSold--;       // Deduct 1 from the count
    }
}
```

Looking at the code, there is a lot is similarity between this code as well as the **buyTicket()** method, in that we need to determine the cost of the ticket in a similar way. Let's extract the

code into a useful method so that we do not have duplication. Let's write a function called **priceFor(Patron p)** which returns the price that the patron should pay for his/her ticket:

```
public static float priceFor(Patron p) {
    if (p.age <= 12)
        return Ticket.REG_CHILD_PRICE;
    else if (p.age >= 65)
        return Ticket.REG_SENIOR_PRICE;

    return Ticket.REG_ADULT_PRICE;
}
```

Now that we have this useful method, we can update our **buyTicket()** and **returnTicket()** methods:

```
public static void buyTicket(Patron aPatron, Theatre aTheatre) {
    if (aTheatre.seatsSold < aTheatre.capacity) {
        Ticket  t = new Ticket(aTheatre.id);    // make the ticket
        aPatron.ticket = t;                     // store the ticket

        // Now increase the earnings and add 1 to the count of seats sold
        aTheatre.movieEarnings += priceFor(aPatron);
        aTheatre.seatsSold++;
    }
}
```

```
public static void returnTicket(Patron aPatron, Theatre aTheatre) {
    if ((aPatron.ticket != null) &&
        (aPatron.ticket.theatreID == aTheatre.id)) {

        aPatron.ticket = null;

        // Now decrease the earnings and deduct 1 from the count
        aTheatre.movieEarnings -= priceFor(aPatron);
        aTheatre.seatsSold--;
    }
}
```

There ... that is much cleaner now. How can we adjust our code so that it is not necessary to pass in **aTheatre** as a parameter to the **returnTicket()** method ?

Well, we still need the **Theatre** object in order to deduct from the earnings. But we can make a change to store the **Theatre** object itself inside the **Ticket** instead of just the theatreID:

```
public class Ticket {
    public static final float REG_CHILD_PRICE = 7.99f;
    public static final float REG_ADULT_PRICE = 10.99f;
    public static final float REG_SENIOR_PRICE = 8.50f;

    public Theatre    theatre;   // The theatre to which it is valid

    public Ticket(Theatre t) {
        theatre = t;
    }
}
```

We would have to make some changes to the methods that we wrote, but this would be simple. Notice how the **returnTicket()** method would simplify with just the one parameter:

```
public static void returnTicket(Patron aPatron) {
    if (aPatron.ticket != null) {
        aPatron.ticket = null;

        // Now decrease the earnings
        if (aPatron.age <= 12)
            aTheatre.movieEarnings -= Ticket.REG_CHILD_PRICE;
        else if (aPatron.age >= 65)
            aTheatre.movieEarnings -= Ticket.REG_SENIOR_PRICE;
        else
            aTheatre.movieEarnings -= Ticket.REG_ADULT_PRICE;

        aPatron.ticket.theatre.seatsSold--;  // Deduct 1 from count
    }
}
```

So, now how do we test it all out to see if it works ? Well, we'd need to make a test program.

```
public class TheatreSimulatorProgram {

    // Simulate the opening of a movie to a theatre
    public static void openMovie(Theatre aTheatre, String newMovie) {
        aTheatre.moviePlaying = newMovie;
        aTheatre.movieEarnings = 0;
        aTheatre.seatsSold = 0;
        aTheatre.admittedPatrons = 0;
    }


    // Simulate a patron entering a theatre
    public static void admit(Patron aPatron, Theatre aTheatre) {
        if ((aPatron.ticket != null) &&
            (aPatron.ticket.theatreID == aTheatre.id)) {

            aPatron.ticket = null;
            aTheatre.admittedPatrons++;
        }
    }


    // Determine and return the price that the given Patron should pay for a ticket
    public static float priceFor(Patron p) {
        if (p.age <= 12)
            return Ticket.REG_CHILD_PRICE;
        else if (p.age >= 65)
            return Ticket.REG_SENIOR_PRICE;

        return Ticket.REG_ADULT_PRICE;
    }
```

```
    // Simulate a patron buying a ticket at a theatre
    public static void buyTicket(Patron aPatron, Theatre aTheatre) {
        if (aTheatre.seatsSold < aTheatre.capacity) {
            Ticket  t = new Ticket(aTheatre.id);    // make the ticket
            aPatron.ticket = t;                     // store the ticket

            // Now increase the earnings and add 1 to the count of seats sold
            aTheatre.movieEarnings += priceFor(aPatron);
            aTheatre.seatsSold++;
        }
    }

    // Simulate a patron returning a ticket for a theatre
    public static void returnTicket(Patron aPatron, Theatre aTheatre) {
        if ((aPatron.ticket != null) &&
            (aPatron.ticket.theatreID == aTheatre.id)) {

            aPatron.ticket = null;

          // Now decrease the earnings and deduct 1 from the count
          aTheatre.movieEarnings -= priceFor(aPatron);
          aTheatre.seatsSold--;
        }
    }

    // The program starts here
    public static void main(String args[]) {
        // Make a couple of theatres
        Theatre t1 = new Theatre('A', 3);       // capacity = 3
        Theatre t2 = new Theatre('B', 10);

        // Make a few of patrons
        Patron adam = new Patron(10);
        Patron biff = new Patron(17);
        Patron chad = new Patron(21);
        Patron dana = new Patron(73);

        System.out.println("Theatre 1 capacity:   " + t1.capacity);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);

        System.out.println("Theatre 1 opening Movie ... \"Jurassic World\"");
        openMovie(t1, "Jurassic World");
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);

        System.out.println("10 year old buying a ticket ... ");
        buyTicket(adam, t1);
        System.out.println(adam);

        System.out.println("17 year old and 21 year old buying tickets ... ");
        buyTicket(biff, t1);
        buyTicket(chad, t1);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);

        System.out.println("10 year old returning a ticket ... ");
        returnTicket(adam, t1);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);
```

```
        System.out.println(adam);

        System.out.println("73 year old buying a ticket ... ");
        buyTicket(dana, t1);
        System.out.println(dana);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);

        System.out.println("Admitting 17 year old into theatre ... ");
        admit(biff, t1);
        System.out.println(biff);

        System.out.println("Admitting 21 year old into wrong theatre ... ");
        admit(chad, t2);
        System.out.println(chad);
        System.out.println("Theatre 2 admitted patrons: " + t2.admittedPatrons);

        System.out.println("Admitting 21 year old into correct theatre ... ");
        admit(chad, t1);
        System.out.println("Theatre 1 admitted patrons: " + t1.admittedPatrons);
        System.out.println("Admitting 73 year old into theatre 1 ... ");
        admit(dana, t1);
        System.out.println("Theatre 1 admitted patrons: " + t1.admittedPatrons);
        System.out.println("Attempt to admit 10 year old into theatre 1 ... ");
        admit(adam, t1);
        System.out.println("Theatre 1 admitted patrons: " + t1.admittedPatrons);
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);
    }
}
```

Here is the expected output:

```
Theatre 1 capacity:   3
Theatre 1 seats sold: 0
Theatre 1 opening Movie ... "Jurassic World"
Theatre 1 earnings:   0.0
10 year old buying a ticket ...
10yr old patron with a ticket for theatre A
17 year old and 21 year old buying tickets ...
Theatre 1 seats sold: 3
Theatre 1 earnings:   29.97
10 year old returning a ticket ...
Theatre 1 seats sold: 2
Theatre 1 earnings:   21.98
10yr old patron without a ticket
73 year old buying a ticket ...
73yr old patron with a ticket for theatre A
Theatre 1 seats sold: 3
Admitting 17 year old into theatre ...
17yr old patron without a ticket
Admitting 21 year old into wrong theatre ...
21yr old patron with a ticket for theatre A
Theatre 2 admitted patrons: 0
Admitting 21 year old into correct theatre ...
Theatre 1 admitted patrons: 2
Admitting 73 year old into theatre 1 ...
Theatre 1 admitted patrons: 3
Attempt to admit 10 year old into theatre 1 ...
Theatre 1 admitted patrons: 3
Theatre 1 earnings:   30.48
```

11.3 Converting to Object-Oriented Programming Style

In our previous example, we created 3 classes as data structures: **Ticket**, **Patron** and **Theatre** as well as a couple of test programs, one being the simulator itself. But we actually wrote our code in a non-object-oriented style.

In our main simulation program, we wrote 5 procedures that perform useful portions of the simulation. However, at the highest level, the actual simulation took place in the main method, where we made use of these methods. Here is a reduced form of our code (with some code omitted):

```
public class TheatreSimulatorProgram {

    public static void  openMovie(Theatre aTheatre, String newMovie) { ... }
    public static void  admit(Patron aPatron, Theatre aTheatre) { ... }
    public static void  buyTicket(Patron aPatron, Theatre aTheatre) { ... }
    public static void  returnTicket(Patron aPatron, Theatre aTheatre) { ... }
    public static float priceFor(Patron p) { ... }

    public static void main(String args[]) {
        // Make a couple of theatres
        Theatre t1 = new Theatre('A', 3);       // capacity = 3
        Theatre t2 = new Theatre('B', 10);

        // Make a few of patrons
        Patron adam = new Patron(10);
        Patron biff = new Patron(17);
        Patron chad = new Patron(21);
        Patron dana = new Patron(73);

        openMovie(t1, "Jurassic World");
        buyTicket(adam, t1);
        buyTicket(biff, t1);
        buyTicket(chad, t1);
        returnTicket(adam, t1);
        buyTicket(dana, t1);
        admit(biff, t1);
        admit(chad, t2);
        admit(chad, t1);
        admit(dana, t1);
        admit(adam, t1);
    }
}
```

In order to write code that uses proper object-oriented programming style, the 5 static methods written here need to be moved elsewhere so that all that is left here is the main method. But to where do we move these methods ? The simple answer is ... to the data structure classes.

Consider, for example, the **priceFor()** method. Recall the code, as shown on the next page. You will notice that it makes use of both the **Patron** data structure as well as the **Ticket** data structure. So, we can move this code into either of those classes.

```
    public static float priceFor(Patron p) {
        if (p.age <= 12)
            return Ticket.REG_CHILD_PRICE;
        else if (p.age >= 65)
            return Ticket.REG_SENIOR_PRICE;

        return Ticket.REG_ADULT_PRICE;
    }
```

Consider first, moving it to the **Ticket** class. Here is how the **Ticket** class would now look:

```
public class Ticket {
    public static final float REG_CHILD_PRICE = 7.99f;
    public static final float REG_ADULT_PRICE = 10.99f;
    public static final float REG_SENIOR_PRICE = 8.50f;

    public char    theatreID;   // ID of the theatre to which it is valid

    public Ticket(char id) {
        theatreID = id;
    }
    public static float priceFor(Patron p) {
        if (p.age <= 12)
            return REG_CHILD_PRICE;
        else if (p.age >= 65)
            return REG_SENIOR_PRICE;
        return REG_ADULT_PRICE;
    }
}
```

Notice that the method is simply moved here. However, you may also notice that `Ticket.REG_CHILD_PRICE` has become simply `REG_CHILD_PRICE`. That is because the method is now inside the **Ticket** class, so we don't need to tell JAVA to "go inside" the **Ticket** class by saying `Ticket.`.

What would the code look like now that calls this method ? Well, consider the **buyTicket()** method from our **TheatreSimulatorProgram**:

```
    public static void buyTicket(Patron aPatron, Theatre aTheatre) {
        if (aTheatre.seatsSold < aTheatre.capacity) {
            Ticket  t = new Ticket(aTheatre.id);
            aPatron.ticket = t;
            aTheatre.movieEarnings += priceFor(aPatron);
            aTheatre.seatsSold++;
        }
    }
```

This code won't compile now because the priceFor() method has been moved to a different class/file. So, we need to tell JAVA where to go look for it. Hence the line:

```
    aTheatre.movieEarnings += priceFor(aPatron);
```

needs to be replaced by:

```
    aTheatre.movieEarnings += Ticket.priceFor(aPatron);
```

Now JAVA knows where we moved it to. It will "go inside" the **Ticket** class to find the method.

Now, instead of putting it into the **Ticket** class, we could have also put it into the **Patron** class. Here is how the **Patron** class would now look:

```
public class Patron {
    public int       age;
    public Ticket    ticket;

    public Patron(int anAge) {
        age = anAge;
        ticket = null;
    }

    public String toString() {
        String result = age + "yr old patron with";
        if (ticket == null)
            return result + "out a ticket";
        return result + " a ticket for theatre " + ticket.theatreID;
    }

    public float priceFor() {
        if (age <= 12)
            return Ticket.REG_CHILD_PRICE;
        else if (age >= 65)
            return Ticket.REG_SENIOR_PRICE;
        return Ticket.REG_ADULT_PRICE;
    }
}
```

Notice a couple of things. First, we removed the word **static** from the method's first line (i.e., the method signature). That is because the method is no longer considered a class method, but is now an instance method. What does that mean ?

Well, notice also that the **Patron** parameter is missing! Notice as well that we are now using `age` instead of `p.age`. Since the method is inside the **Patron** class now, we don't need to tell JAVA where to find the age. It finds it right here at the top of the class definition.

But how does JAVA know which **Patron** to get the age from ? Well, this is where the removal of the **static** word comes into play. Consider how the method will be called now.

```
    public static void buyTicket(Patron aPatron, Theatre aTheatre) {
        if (aTheatre.seatsSold < aTheatre.capacity) {
            Ticket  t = new Ticket(aTheatre.id);
            aPatron.ticket = t;
            aTheatre.movieEarnings += aPatron.priceFor();
            aTheatre.seatsSold++;
        }
    }
```

Here you will notice that we put `aPatron.` in front of the method call. This is a little different. For the first time, we are putting a variable/parameter name in front of the dot instead of the **Patron** class name.

This means that JAVA will considers the actual **Patron** object that is in front of the dot. It is this patron (i.e., **aPatron** in this case) that is used inside the method when JAVA accesses the **age** attribute. If we used a different Patron like this:

```
anotherPatron.priceFor();
yetAnotherPatron.priceFor();
```

then JAVA will used these other **Patron** objects when inside the method in order to access their **age** attribute. So the **age** depends on the object instance being used in front of the dot. That is, it varies according to the instance. Therefore, the method is now called an ***instance method***. And since it varies, it is no longer fixed or **static**. That is why the **static** keyword has been removed.

The method name should likely be changed now, though, because there is no parameter:

```
aTheatre.movieEarnings += aPatron.ticketPrice();
```

```
    public float ticketPrice() {
        if (age <= 12)
            return Ticket.REG_CHILD_PRICE;
        else if (age >= 65)
            return Ticket.REG_SENIOR_PRICE;
        return Ticket.REG_ADULT_PRICE;
    }
```

Let us now consider this method that we wrote in our **TheatreSimulatorProgram**:

```
    public static void openMovie(Theatre aTheatre, String newMovie) {
        aTheatre.moviePlaying = newMovie;
        aTheatre.movieEarnings = 0;
        aTheatre.seatsSold = 0;
        aTheatre.admittedPatrons = 0;
    }
```

You will notice that it makes use of the **Theatre** class in the parameter. So we can move this method into the **Theatre** class as follows:

```
public class Theatre {
    public char    id;              // ID of theatre
    public String  moviePlaying;    // # people that can be seated in theatre
    public float   movieEarnings;   // title of movie currently playing
    public int     capacity;        // amount of money movie has made so far
    public int     seatsSold;       // # seats sold for movie so far
    public int     admittedPatrons; // # patrons admitted into theatre

    public Theatre(char identification, int cap) {
        moviePlaying = "";
        movieEarnings = 0;
        capacity = cap;
        seatsSold = 0;
        id = identification;
        admittedPatrons = 0;
    }
```

```
    // Simulate the opening of a movie to this theatre
    public void openMovie(String newMovie) {
        moviePlaying = newMovie;
        movieEarnings = 0;
        seatsSold = 0;
        admittedPatrons = 0;
    }
}
```

Notice again that the **static** is now removed and the **Theatre** parameter is no longer needed. We access the theatre's attributes directly now. In the **main()** method of our **TheatreSimulatorProgram**, we change the way that we call the method from this:

```
    openMovie(t1, "Jurassic World");
```

to this:

```
    t1.openMovie("Jurassic World");
```

Do you understand why ?

Finally, we need to move the **admit()**, **buyTicket()** and **returnTicket()** methods to where they belong. Looking at the parameters, we have two choices ... the **Patron** class or the **Theatre** class. For the **admit()** it makes sense to put that one in the **Theatre** class, passing a **Patron** in as the only parameter:

```
    // Simulate a patron entering this theatre
    public void admit(Patron aPatron) {
        if ((aPatron.ticket != null) &&
            (aPatron.ticket.theatreID == id)) {

            aPatron.ticket = null;
            admittedPatrons++;
        }
    }
```

Again, the **static** is now removed and the **Theatre** parameter is no longer needed. In the **main()** method of our **TheatreSimulatorProgram**, we change the way that we call the method from this:

```
    admit(chad, t1);
```

to this:

```
    t1.admit(chad);
```

For the other two methods, we will put them in the **Patron** class, leaving the **Theatre** object as the sole parameter:

```
    // Simulate this patron buying a ticket at a theatre
    public void buyTicket(Theatre aTheatre) {
        if (aTheatre.seatsSold < aTheatre.capacity) {
            Ticket  t = new Ticket(aTheatre.id);    // make the ticket
            ticket = t;                     // store the ticket

            // Now increase the earnings and add 1 to the count of seats sold
            aTheatre.movieEarnings += ticketPrice();
            aTheatre.seatsSold++;
        }
    }

    // Simulate this patron returning a ticket for a theatre
    public void returnTicket(Theatre aTheatre) {
        if ((ticket != null) &&
            (ticket.theatreID == aTheatre.id)) {

            ticket = null;

            // Now decrease the earnings and deduct 1 from the count
            aTheatre.movieEarnings -= ticketPrice();
            aTheatre.seatsSold--;
        }
    }
```

Here is the final **TheatreSimulatorProgram**. Notice how simple it is now:

```
public class TheatreSimulatorProgram {
    public static void main(String args[]) {
        // Make a couple of theatres
        Theatre t1 = new Theatre('A', 3);       // capacity = 3
        Theatre t2 = new Theatre('B', 10);

        // Make a few of patrons
        Patron adam = new Patron(10);
        Patron biff = new Patron(17);
        Patron chad = new Patron(21);
        Patron dana = new Patron(73);

        System.out.println("Theatre 1 capacity:   " + t1.capacity);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);

        System.out.println("Theatre 1 opening Movie ... \"Jurassic World\"");
        t1.openMovie("Jurassic World");
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);

        System.out.println("10 year old buying a ticket ... ");
        adam.buyTicket(t1);
        System.out.println(adam);

        System.out.println("17 year old and 21 year old buying tickets ... ");
        biff.buyTicket(t1);
        chad.buyTicket(t1);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);

        System.out.println("10 year old returning a ticket ... ");
```

```
        adam.returnTicket(t1);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);
        System.out.println(adam);

        System.out.println("73 year old buying a ticket ... ");
        dana.buyTicket(t1);
        System.out.println(dana);
        System.out.println("Theatre 1 seats sold: " + t1.seatsSold);

        System.out.println("Admitting 17 year old into theatre ... ");
        t1.admit(biff);
        System.out.println(biff);

        System.out.println("Admitting 21 year old into wrong theatre ... ");
        t2.admit(chad);
        System.out.println(chad);
        System.out.println("Theatre 2 admitted patrons: " + t2.admittedPatrons);

        System.out.println("Admitting 21 year old into correct theatre ... ");
        t1.admit(chad);
        System.out.println("Theatre 1 admitted patrons: " + t1.admittedPatrons);
        System.out.println("Admitting 73 year old into theatre 1 ... ");
        t1.admit(dana);
        System.out.println("Theatre 1 admitted patrons: " + t1.admittedPatrons);
        System.out.println("Attempt to admit 10 year old into theatre 1 ... ");
        t1.admit(adam);
        System.out.println("Theatre 1 admitted patrons: " + t1.admittedPatrons);
        System.out.println("Theatre 1 earnings:   " + t1.movieEarnings);
    }
}
```

The code is now considered to be "object-oriented" because the methods are organized into the various classes as kinds of ***behaviors*** that the object is able to perform. For example, **Patrons** are now able to buy tickets and return tickets and **Theatres** are able to open movies and admit patrons.

Hopefully, you understand why we put the methods in those particular classes ... because the simulation code is much more readable now. And also, if we made a different simulation, we don't have to re-write the methods again, since they are stored in the appropriate classes.

You will learn much more about object-oriented programming in the next course.

11.4 Convergence in Simulation

It is often the case that we need to compute an answer to some problem in which the parameters are complex and/or uncertain. In some situations, it may be unfeasible or impossible to compute an exact result to a problem using a well-defined and predictable algorithm.

There is a specific type of simulation method that is well-suited for situations in which you need to make an estimate, forecast or decision where there is significant uncertainty:

*The **Monte Carlo Method** uses randomly generated or sampled data and computer simulations to obtain approximate solutions to complex mathematical and statistical problems.*

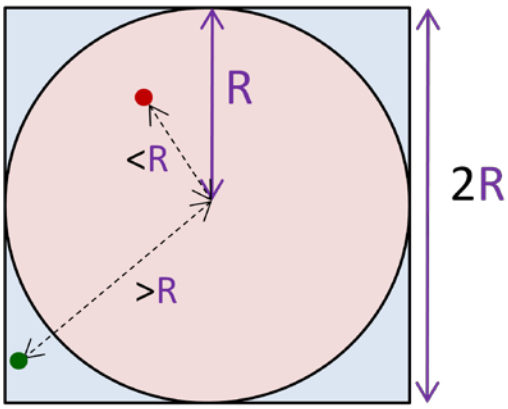
There is always some error involved with this scheme, but the larger the number of random samples taken, the more accurate the result.

The simplest example that is used to describe the Monte Carlo method is that of computing an approximation of **π** (i.e., **pi**). **π** is a mathematical constant whose value is the ratio of any circle's circumference to its diameter. It is approximately equal to **3.141593** in the usual decimal notation. **π** is a very important number in math and computer science as it relates to many trigonometric functions and geometric algorithms and is used in graphics and animation.

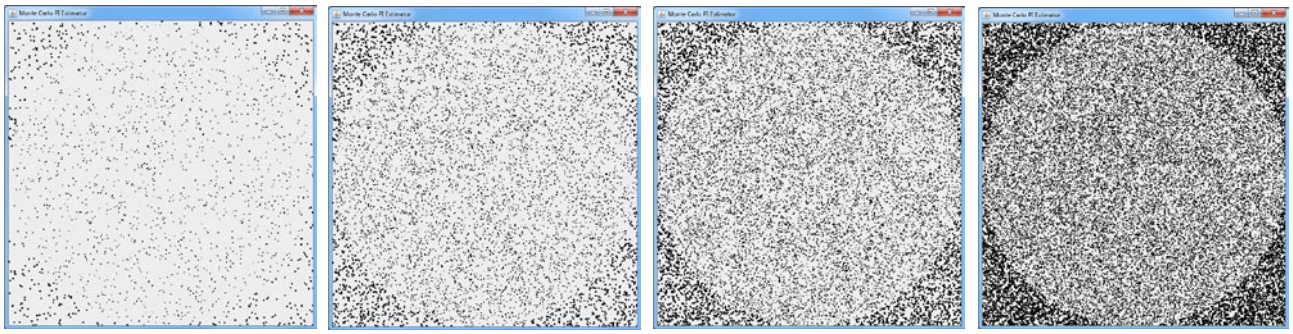
The value of **π** can be approximated using a Monte Carlo method based on this principle:

Given a circle inscribed in a square (i.e., the largest circle that fits in the square), the ratio of the area of the circle to that of the square is π / 4.

Knowing this, if we can get an estimate for the area of the circle as well as the area of the square, then we can find an approximation for **π / 4**, of course then multiplying by **4** to get an approximation for **π**.



We can estimate the area of the square and circle by uniformly scattering some points throughout the square. Some will lie within the circle, some will lie outside the circle. The more points that we add, the better the approximation of the area, as the whole square and circle will eventually be covered as time goes on.



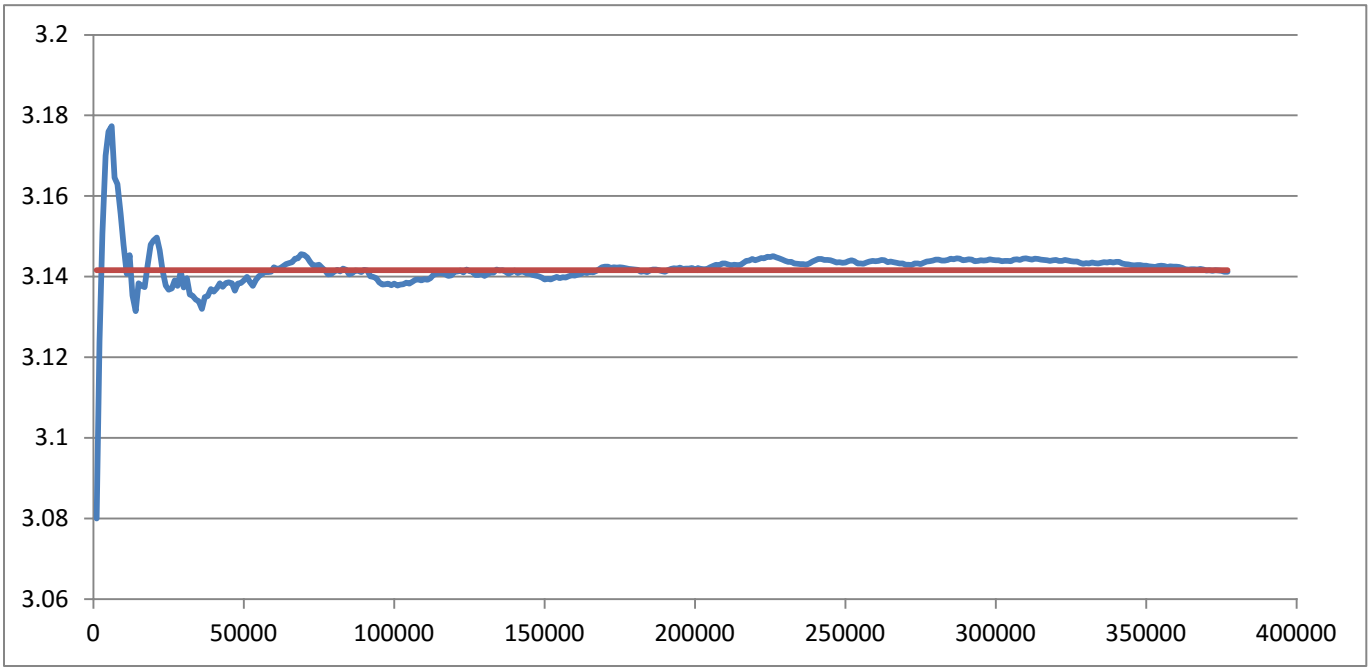
2,000	8,000	16,000	32,000
3.15242	3.15461	3.14075	3.144625

Here is the algorithm, assuming that the circle and square are centered at point **(R,R)**:

```
Algorithm: ComputePi
        R:              the radius of a circle and ½ width & ½ height of a square

1.      pointsInCircle = 0
2.      pointsInSquare = 0
3.      repeat for a user-chosen amount of iterations {
4.          x = random value from 0 to 2R-1
5.          y = random value from 0 to 2R-1
6.          pointsInSquare = pointsInSquare + 1
7.          if ((the distance from (x,y) to (R,R) < R) then
8.                  pointsInCircle = pointsInCircle + 1
        }
9.      print (pointsInCircle / pointsInSquare * 4)
```

We can stop the loop at any time. As the loop goes on, however, the algorithm will slowly converge to a better approximation as more data points are sampled. If the points are purposefully chosen only around the center of the circle, they will not be uniformly distributed, and so the approximation will be poor. An approximation will also be poor if only a few points are randomly chosen throughout the whole square. Thus, the approximation of π will become more accurate both with more points and with their uniform distribution.

Here is an example showing how the approximated value (blue) will converge towards the optimal value (solid red line).



In JAVA, we can write a program that shows us visually what is happening by drawing each point that is randomly chosen. For those inside the circle we can draw them as small circles and those outside as small squares. The code below creates a simple **JPanel** and then enters an infinite loop to compute the random points and draw them each time. It also

computes the estimate for PI and continually shows it. DrJava has a difficult time handling a lot of output to the console, so your program may slowly crawl to a halt at around 30,000 points.

```
import java.awt.*;
import javax.swing.*;

public class MonteCarloPiEstimationProgram {
    public static void main(String[] args) {
        int     R = 300;
        int     x, y;
        int     pointsInCircle = 0;
        int     pointsInSquare = 0;

        JFrame frame = new JFrame("Monte Carlo PI Estimator");
        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(2*R, 2*R));
        frame.add(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack(); // Makes size according to panel's preference
        frame.setVisible(true);

        // Start Simulating
        while(true) {
            x = (int)(Math.random()*2*R);
            y = (int)(Math.random()*2*R);
            if (Math.sqrt((x-R)*(x-R) + (y-R)*(y-R)) < R) {
                panel.getGraphics().fillOval(x, y, 3, 3);
                pointsInCircle++;
            }
            else
                panel.getGraphics().fillRect(x, y, 3, 3);

            pointsInSquare++;
            System.out.println("Points: " + pointsInSquare + "   PI estimate: " +
                               (double)pointsInCircle/pointsInSquare*4);
        }
    }
}
```

It will take a long time for this code to produce a reasonable approximation for PI. We could add a **for** loop in the **while()** loop to add many points each time (such as 1000) in order to speed up the approximation process. The code is on the following page.

It may take quite a few added points to get a good value for PI. In one iteration, at about 1,000,000 points, I was able to get 3.140212. It will be very hard for us to get an accurate value of 3.14159 because we are depending on the random number generator. Hence, we will always have an approximation, which will get closer to the optimal value in time.

In some areas of computer science (such as Learning Automata), approximate solutions to complex problems are often easier to obtain, although they are not always accurate. Regardless, most approximations are "close enough" in real world applications.

Here is the updated code:

```
while(true) {
    for (int i=0; i<1000; i++) {
        x = (int)(Math.random()*2*R);
        y = (int)(Math.random()*2*R);
        if (Math.sqrt((x-R)*(x-R) + (y-R)*(y-R)) < R) {
            panel.getGraphics().fillOval(x, y, 3, 3);
            pointsInCircle++;
        }
        else
            panel.getGraphics().fillRect(x, y, 3, 3);

        pointsInSquare++;
    }
    System.out.println("Points: " + pointsInSquare + "   PI estimate: " +
                       (double)pointsInCircle/pointsInSquare*4);
}
```