CSC 325 Adv Data Structures

Lecture 2

Jump Tables

vtable

In many languages (Java included) you can make an array of potential functions to call. Consider the following code:

```
class Animal {
  public int foo() {
    return 1;
  }
}

class Dog extends Animal {
  @Override
  public int foo() {
    return 2;
  }
}

class BigDog extends Dog {
  @Override
  public int foo() {
    return 3;
  }
}

public class AnimalTest {
  public static void main(String[] args) {
    Animal a = new Animal();
    Animal b = new Dog();
    Animal c = new BigDog();
    System.out.println(a.foo());
    System.out.println(b.foo());
    System.out.println(c.foo());
  }
}
```

This code compiles and runs just fine (try it out!). For the last three lines of the main function, the compiler doesn't know which foo function to call. All three variables (a, b, and c) are of type 'Animal'. This is their "compile time type", i.e. the data type of the variable that the compiler knows about. However, it isn't until

we run the code that we see a's runtime type is Animal, b's runtime type is Dog, and c's runtime type is BigDog. We call this "runtime type" since it is only known at runtime. You may say that the compiler could have seen this too. However consider this slightly different code:

```
public class AnimalTest {
  public static void main(String[] args) {
    // this assumes we have imported the Scanner class
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an animal name: ");
    String animal = input.nextLine();

    Animal x = null;
    if (animal.equals("dog")) {
      x = new Dog();
    }
    else if (animal.equals("big dog")) {
      x = new BigDog();
    }
    else {
      x = new Animal();
    }

    System.out.println(x.foo());
  }
}
```

Does the compiler know the runtime type of the variable 'x' in this case? Of course not. It is only after running the code that we can then see what the user types and modify the datatype of x accordingly.

Having a discrepancy between the compile time type and the runtime type of variables, makes it hard for the compiler to know which function to call. After all, the compiler needs to convert the high level source code into Java byte code. When converted into byte code, the exact function to call must be known! So how does the Java compiler resolve this?

The Java compiler uses something called a virtual function table (or "vtable" for short)[1]. Basically, this is an array of possible functions that could be called. For the variable 'x' above, this may look like the following:

Index	Function
0	Animal's foo()
1	Dog's foo()
2	BigDog's foo()

[1] You can learn more about this concept from a Google search: https://www.google.com/search?q=vtable

This entire concept of calling a different function based on a runtime type should be familiar to you. Remember the concept of **polymorphism**? This is exactly that concept. In fact, being able to correctly describe polymorphism is a potential interview question when you apply for a career in Computer Science. So take this term seriously!

This is a good example of what is known as a "jump table", which is a table (aka an array) of functions. By knowing the index, you know which function to call. So, for example, in Java we could do the following:

```
vtable[0].invoke();
```

This would call Animal's foo since that is at the top of the table. But what is this "invoke" function? Well, here is the full code that would make this work. I will use different functions (and eliminate the need for multiple classes) just to make this concept a little easier to understand:

```
interface VTableEntry {
  public int invoke();
}

public class AnimalTestWithVTable {
  private static int AnimalFoo() { return 1; }
  private static int DogFoo()    { return 2; }
  private static int BigDogFoo() { return 3; }

  public static void main(String[] args) {
    VTableEntry[] vtable = new VTableEntry[3];
    vtable[0] = () -> { return AnimalFoo(); };
    vtable[1] = () -> { return DogFoo();    };
    vtable[2] = () -> { return BigDogFoo(); };

    for (int i = 0; i < vtable.length; i++) {
      System.out.println(vtable[i].invoke());
    }
  }
}
```

I've replaced the foo functions from each class in the code before with 3 different foo functions in the same class. This makes it easier to create the table. Actually, the compiler goes through something similar to this when constructing the vtable so this isn't completely far off!

Let's start our explanation off with the interface part. We need to have a way to specify the signature of each function in our table. All of our functions should return an int and not take any parameters. For this reason, we create an interface with only a single function in it (this is known as a "functional interface"). That function has the signature we want (i.e. returns an int and doesn't take any parameters). We can call this function whatever we want but I would suggest calling it "invoke". Next we have three functions (AnimalFoo, DogFoo, and BigDogFoo) that all match the signature of "invoke". To construct our vtable, we must put each of these functions into the table. Unfortunately, Java doesn't let us do this directly. However, we can create a new function that calls the appropriate Foo function. Basically, we can wrap the call to one of the Foo functions

(such as “AnimalFoo”) inside of another function and use that in the table. Calling this function will, in turn, call AnimalFoo. We use a “lambda function” to accomplish this.

A lambda function is also known as an anonymous function and is a function without a name. We can use this concept by following this syntax:

```
(PARAMETERS) -> { BODY }
```

Since there are no parameters and this lambda function is just meant to return what AnimalFoo returns, we can write it like this:

```
() -> { return AnimalFoo(); }
```

This creates an unnamed function that doesn’t take any parameters and returns whatever AnimalFoo returns (which is an int). This matches the signature of our “invoke” function so Java doesn’t complain. If we then assign this lambda function to a spot in the vtable like so:

```
vtable[0] = () -> { return AnimalFoo(); };
```

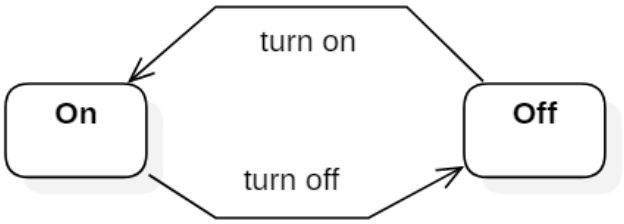
We can then call this function later using:

```
vtable[0].invoke();
```

Which is essentially what we do in the loop. Repeat this for DogFoo and BigDogFoo and you should be able to see how this jump table is built. If not, you may have to re-read this section again.

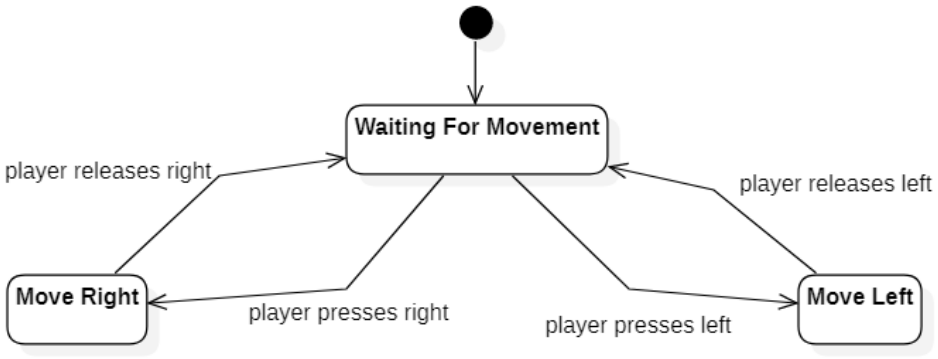
Finite State Machines (FSM)

Before we see the power of jump tables, let’s examine what’s called a “finite state machine”. If you think about a simple light, it can either be on or off. This represents a state machine in which there are two possible states:



If you turn the light on, it moves to the “On” state. If you turn the light off, it moves to the “Off” state. This is a pretty simple state machine but it explains the concept well. There are several states and the action an object takes depends upon its state.

Let’s look at another example. Let’s say we have a player to a game we are creating. We could design the player’s movement using the following state machine:



The black circle up top indicates the "start state" or the state that we start in. While there is no input from the user, the player doesn't do anything. When the user presses left, the player moves to the "Move Left" state. When the player releases the left key, the player moves to the "Waiting For Movement" state. Inside each state could be code that instructs the player to keep moving. So as long as you are in the "Move Left" state, the player will continue to move left. This is very useful and not just for game development. This concept can give entities a certain amount of autonomy. In other words, an object can always know what to do just by looking at its current state and by running the code for that particular state. But how do we implement this?

Using a Jump Table to Implement an FSM

Now that we understand the basics of jump tables and finite state machines, let's combine these concepts. The best way that I personally have found to implement a finite state machine, is to create a dictionary that maps the state in the finite state machine, with the function that contains the code for that state. In other words, let's map the following:

State	Function
WAITING_FOR_MOVEMENT	StateWaitingForMovement()
MOVE_RIGHT	StateMoveRight()
MOVE_LEFT	StateMoveLeft()

We can use an enum (remember them from the "Java Re-Introduction" lecture?) to represent our states in the left column and create functions for each of them (shown in the right column). So now we can have the following:

```
enum PlayerState {
  WAITING_FOR_MOVEMENT,
  MOVE_RIGHT,
  MOVE_LEFT;
}

class Player {
  private void StateWaitingForMovement() {  }
  private void StateMoveRight() {  }
  private void StateMoveLeft() {  }
}
```

Let's create a dictionary to map the enum values to the functions. This will be our jump table. In Java we can do this by creating a HashMap:

```
import java.util.HashMap;

interface StateFunction {
  public void invoke();
}

enum PlayerState {
  WAITING_FOR_MOVEMENT,
  MOVE_RIGHT,
  MOVE_LEFT;
}

class Player {
  private HashMap<PlayerState, StateFunction> fsmJumpTable;

  /// CTOR
  public Player() {
    fsmJumpTable = new HashMap<>();

    fsmJumpTable.put(PlayerState.WAITING_FOR_MOVEMENT,
                                    () -> { StateWaitingForMovement(); });
    fsmJumpTable.put(PlayerState.MOVE_RIGHT, () -> { StateMoveRight(); });
    fsmJumpTable.put(PlayerState.MOVE_LEFT, () -> { StateMoveLeft(); });

    state = PlayerState.WAITING_FOR_MOVEMENT;
  }

  private void StateWaitingForMovement() {  }
  private void StateMoveRight() {  }
  private void StateMoveLeft() {  }
}
```

We initially start our player off in the Waiting For Movement state since that is what our state machine above had shown (with the black dot pointing to the state). This is very similar to our Animal Foo functions. The only real differences is that these functions do not return a value and we have an enum value as the key instead of an int.

To make things a little easier to change to a new state, we can introduce a helper function for changing states. We also need to have the Player object remember what state they are currently in, so we need a field for that. We could also introduce some pseudo code into our state functions to show what would happen in these functions:

```
import java.util.HashMap;

interface StateFunction {
  public void invoke();
}
```

```
enum PlayerState {
  WAITING_FOR_MOVEMENT,
  MOVE_RIGHT,
  MOVE_LEFT;
}

class Player {
  private PlayerState state;
  private HashMap<PlayerState, StateFunction> fsmJumpTable;

  /// CTOR
  public Player() {
    fsmJumpTable = new HashMap<>();
    fsmJumpTable.put(PlayerState.WAITING_FOR_MOVEMENT,
                                        () -> { StateWaitingForMovement(); });
    fsmJumpTable.put(PlayerState.MOVE_RIGHT, () -> { StateMoveRight(); });
    fsmJumpTable.put(PlayerState.MOVE_LEFT, () -> { StateMoveLeft(); });

    state = PlayerState.WAITING_FOR_MOVEMENT;
  }

  /// State Meths
  public void changeState(PlayerState newState) {
    if (state != newState) {
      state = newState;
    }
  }

  private void StateWaitingForMovement() {
    /*
    if (user presses left arrow key) {
      changeState(PlayerState.MOVE_LEFT);
    }
    else if (user presses right arrow key) {
      changeState(PlayerState.MOVE_RIGHT);
    }
    */
  }

  private void StateMoveRight() {
    /*
    move player to the right
    if (user releases right arrow key) {
      changeState(PlayerState.WAITING_FOR_MOVEMENT);
    }
    */
  }
```

```
  private void StateMoveLeft() {
    /*
    move player to the left
    if (user releases left arrow key) {
      changeState(PlayerState.WAITING_FOR_MOVEMENT);
    }
    */
  }
}
```

Now let's add the ability to actually call one of the state functions, depending on which state the player is in. We can also add an entry point and some test code so we can compile and run this thing. Here is the complete code:

```
import java.util.HashMap;

interface StateFunction {
  public void invoke();
}

enum PlayerState {
  WAITING_FOR_MOVEMENT,
  MOVE_RIGHT,
  MOVE_LEFT;
}

class Player {
  private PlayerState state;
  private HashMap<PlayerState, StateFunction> fsmJumpTable;

  /// CTOR
  public Player() {
    fsmJumpTable = new HashMap<>();
    fsmJumpTable.put(PlayerState.WAITING_FOR_MOVEMENT,
                                    () -> { StateWaitingForMovement(); });
    fsmJumpTable.put(PlayerState.MOVE_RIGHT, () -> { StateMoveRight(); });
    fsmJumpTable.put(PlayerState.MOVE_LEFT, () -> { StateMoveLeft(); });

    state = PlayerState.WAITING_FOR_MOVEMENT;
  }

  /// State Meths
  public void changeState(PlayerState newState) {
    if (state != newState) {
      state = newState;
    }
  }
```

```
  public void doState() {
    if (fsmJumpTable.containsKey(state)) {
      fsmJumpTable.get(state).invoke();
    }
  }

  private void StateWaitingForMovement() {
    // if (user presses left arrow key) {
    //   changeState(PlayerState.MOVE_LEFT);
    // }
    // else if (user presses right arrow key) {
    //   changeState(PlayerState.MOVE_RIGHT);
    // }
    System.out.println("Checking for movement");
  }

  private void StateMoveRight() {
    // move player to the right
    // if (user releases right arrow key) {
    //   changeState(PlayerState.WAITING_FOR_MOVEMENT);
    // }
    System.out.println("Moving player right");
  }

  private void StateMoveLeft() {
    // move player to the left
    // if (user releases left arrow key) {
    //   changeState(PlayerState.WAITING_FOR_MOVEMENT);
    // }
    System.out.println("Moving player left");
  }
}

public class PlayerWithFSM {
  public static void main(String[] args) {
    Player p = new Player();

    /* each frame of game call p.doState() for that frame */
    p.doState();

    // just to test
    p.changeState(PlayerState.MOVE_LEFT);
    p.doState();

    p.changeState(PlayerState.MOVE_RIGHT);
    p.doState();
  }
}
```

Go ahead and compile and run the code above. Notice the output.

This is a very nice way to structure code when there is a good bit of complex actions for an object to take. Each state has its own function where you add the code, so it is very modular. Adding a new state is as simple as adding a new value to the enum, adding a new entry in our HashMap that maps the new state to the new state function, and then creating that new state function. This means it is very scalable.

The alternative to using state machines is to use booleans to try and mark what actions should be taken by the object. You could have a boolean for movingRight and one for movingLeft. If movingRight is true, move right. If movingLeft is true, move left. If neither are true, then you are waiting for movement. But what happens if both are true? What do you do then? Notice that in our state machine above, this was NOT possible as we can't be in two states at once. So this problem doesn't even come up when using state machines. Also, what if you want to add a jumping action to this? You could have a boolean isJumping. Now the code looks something like this:

```
if (!movingRight && !movingLeft) {
  // waiting for movement
}
else if (movingRight && !movingLeft) {
  if (isJumping) {
    // jump while moving right
  }
  else {
    // moving right
  }
}
else if (!movingRight && movingLeft) {
  if (isJumping) {
    // jump while moving left
  }
  else {
    // moving left
  }
}
else {
  // attempting to move left and right at same time
}
```

This is definitely an example of what we call "spaghetti code". If-else statements are not bad. In fact we must use them quite often. However if you find yourself creating complex logic using multiple nested if-else statements like the code above, things are going to get out of control fast! This is NOT code that is easy to maintain, change, or add features to. This is not scalable. This is not elegant. If you were using a finite state machine, you could add a transition to a JumpRight state if the jump button was pressed while in the MoveRight state. In fact, it would look like this:

```
import java.util.HashMap;

interface StateFunction {
  public void invoke();
}

enum PlayerState {
  WAITING_FOR_MOVEMENT,
  MOVE_RIGHT,
  MOVE_LEFT,
  JUMP_RIGHT,
  JUMP_LEFT;
}

class Player {
  private PlayerState state;
  private HashMap<PlayerState, StateFunction> fsmJumpTable;

  /// CTOR
  public Player() {
    fsmJumpTable = new HashMap<>();

    fsmJumpTable.put(PlayerState.WAITING_FOR_MOVEMENT,
                                    () -> { StateWaitingForMovement();  });
    fsmJumpTable.put(PlayerState.MOVE_RIGHT, () -> { StateMoveRight(); });
    fsmJumpTable.put(PlayerState.MOVE_LEFT,  () -> { StateMoveLeft();  });
    fsmJumpTable.put(PlayerState.JUMP_RIGHT, () -> { StateJumpRight(); });
    fsmJumpTable.put(PlayerState.JUMP_LEFT,  () -> { StateJumpLeft();  });

    state = PlayerState.WAITING_FOR_MOVEMENT;
  }

  /// State Meths
  public void changeState(PlayerState newState) {
    if (state != newState) {
      state = newState;
    }
  }

  public void doState() {
    if (fsmJumpTable.containsKey(state)) {
      fsmJumpTable.get(state).invoke();
    }
  }
```

```
  private void StateWaitingForMovement() {
    // if (user presses left arrow key) {
    //   changeState(PlayerState.MOVE_LEFT);
    // }
    // else if (user presses right arrow key) {
    //   changeState(PlayerState.MOVE_RIGHT);
    // }
    System.out.println("Checking for movement");
  }

  private void StateMoveRight() {
    // move player to the right
    // if (user releases right arrow key) {
    //   changeState(PlayerState.WAITING_FOR_MOVEMENT);
    // }
    // else if (user presses jump key) {
    //   changeState(PlayerState.JUMP_RIGHT);
    // }
    System.out.println("Moving player right");
  }

  private void StateMoveLeft() {
    // move player to the left
    // if (user releases left arrow key) {
    //   changeState(PlayerState.WAITING_FOR_MOVEMENT);
    // }
    // else if (user presses jump key) {
    //   changeState(PlayerState.JUMP_LEFT);
    // }
    System.out.println("Moving player left");
  }

  private void StateJumpRight() {
    // if (jump is over) {
    //   changeState(PlayerState.MOVE_RIGHT);
    // }
    System.out.println("Jumping player right");
  }

  private void StateJumpLeft() {
    // if (jump is over) {
    //   changeState(PlayerState.MOVE_LEFT);
    // }
    System.out.println("Jumping player left");
  }
}
```

No nested if statements, no messy logic, no sea of booleans used as flags to try to determine action to take. Just clean logic with a nice function to put the actions for each state. I'm speaking from experience when I say that finite state machines are worth the extra code to get everything set up. Normally my finite state machines

are about 3 to 6 states each. A big one would be around 10 states. The biggest I've ever created has 60 states! Yes, I'm not kidding. The class was around 6,000 lines of code. If I used flags (aka booleans) to try to control the logic, it would have been an absolute nightmare. Instead, with the finite state machine, I took more time scrolling to find the function then I did maintaining this thing. In fact, I had students working with me creating a similar version of the same software but for mobile devices. They didn't use a finite state machine. I was constantly hearing them complain about how hard it was to maintain, add features and fix bugs in their code. It honestly was a bit of a mess to look at. Take the time to learn jump tables and finite state machines. When you have thousands of lines of code in a class and you are still able to change an object's behavior with ease, you'll thank me.

One more small thing. This is a trick I thought of one day (years ago) and it has served me well. Sometimes you want to have actions occur when you first enter a state. I used to create new states for this, but that didn't scale well and was troublesome to keep up with. So instead, I came up with this approach. Simply make another HashMap that maps states to new "enter" functions. These functions will handle what to do when you first enter a state. If there are no actions to take, leave the function empty. Let's alter our Player class to show this concept (and also get rid of pseudo and test code so the concept is more clear):

```
import java.util.HashMap;

interface StateFunction {
  public void invoke();
}

enum PlayerState {
  NONE,
  WAITING_FOR_MOVEMENT,
  MOVE_RIGHT,
  MOVE_LEFT;
}

class Player {
  private PlayerState state;
  private HashMap<PlayerState, StateFunction> stateEnterMeths;
  private HashMap<PlayerState, StateFunction> stateStayMeths;

  /// CTOR
  public Player() {
    stateEnterMeths = new HashMap<>();

    stateEnterMeths.put(PlayerState.WAITING_FOR_MOVEMENT,
                        () -> { StateEnterWaitingForMovement(); });
    stateEnterMeths.put(PlayerState.MOVE_RIGHT, () -> { StateEnterMoveRight(); });
    stateEnterMeths.put(PlayerState.MOVE_LEFT,  () -> { StateEnterMoveLeft();  });
```

```
    stateStayMeths = new HashMap<>();
    stateStayMeths.put(PlayerState.WAITING_FOR_MOVEMENT,
                       () -> { StateStayWaitingForMovement(); });
    stateStayMeths.put(PlayerState.MOVE_RIGHT,  () -> { StateStayMoveRight(); });
    stateStayMeths.put(PlayerState.MOVE_LEFT, () -> { StateStayMoveLeft(); });

    state = PlayerState.NONE;
    changeState(PlayerState.WAITING_FOR_MOVEMENT);
  }

  /// State Meths
  public void changeState(PlayerState newState) {
    if (state != newState) {
      state = newState;
      if (stateEnterMeths.containsKey(newState)) {
        stateEnterMeths.get(newState).invoke();
      }
    }
  }

  public void doState() {
    if (stateStayMeths.containsKey(state)) {
      stateStayMeths.get(state).invoke();
    }
  }

  private void StateEnterWaitingForMovement() { }
  private void StateEnterMoveRight() { }
  private void StateEnterMoveLeft() { }

  private void StateStayWaitingForMovement() { }
  private void StateStayMoveRight() { }
  private void StateStayMoveLeft() { }
}
```