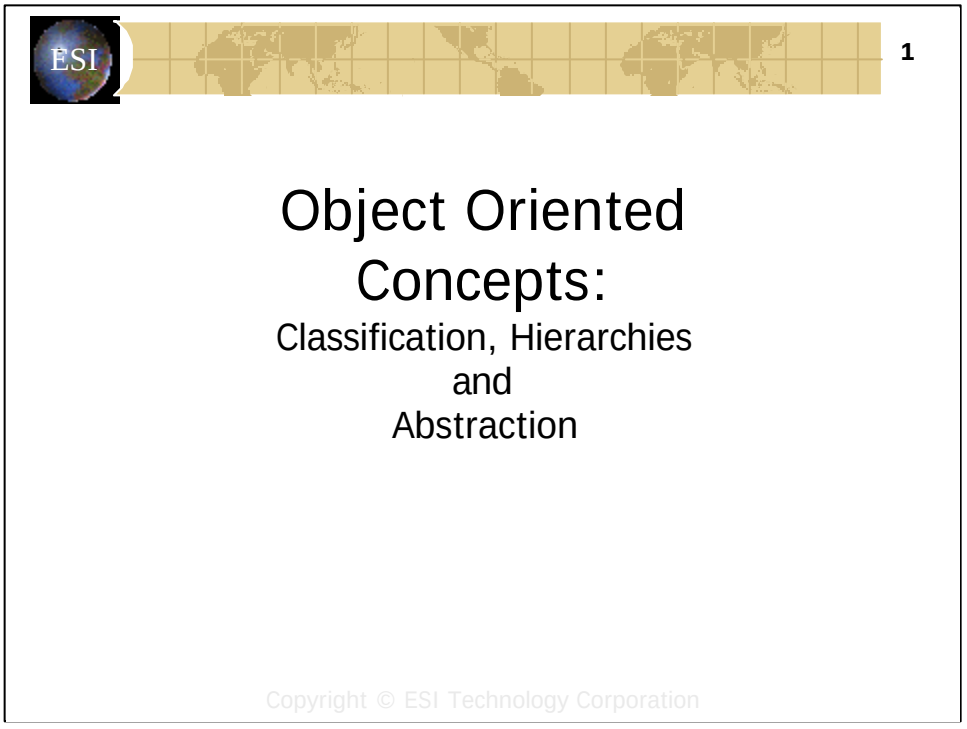
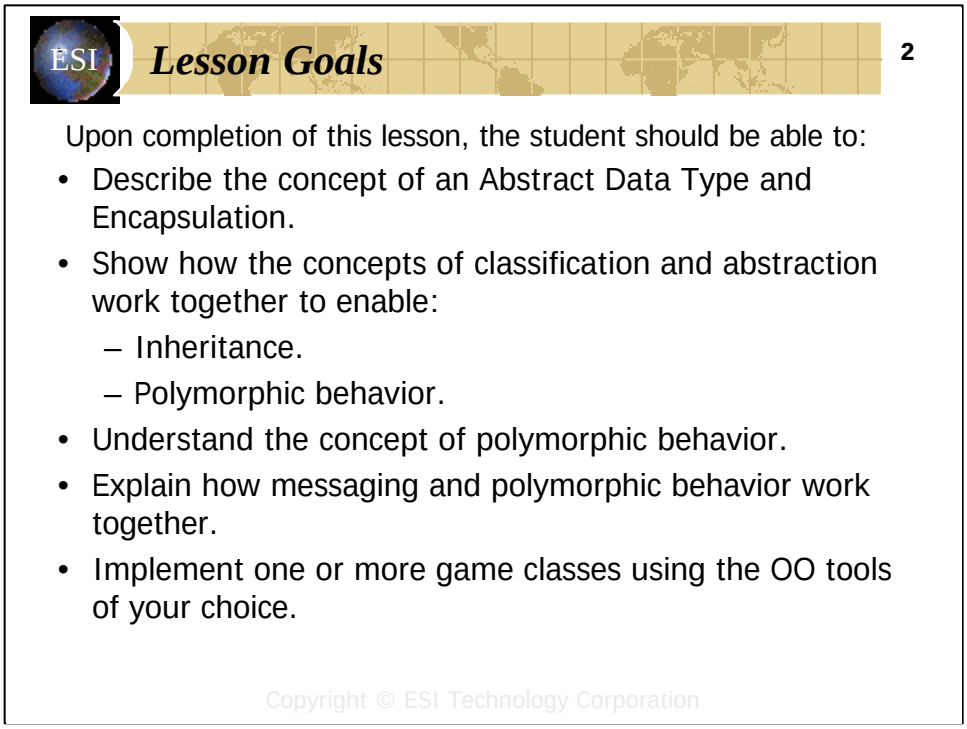


This is the first lesson of a series that describes basic Object Oriented Concepts within the context of a story. The story takes place in a Las Vegas casino where a software engineer has been sent to work with an Poker expert. The story walks you through the process of defining the problem domain (Poker Game), abstracting out objects as well as their behavior and state. It goes on to show how the game is modeled in a step by step fashion. At appropriate points throughout the story, enough object oriented concepts are covered to get you started building the Poker Game on your own.



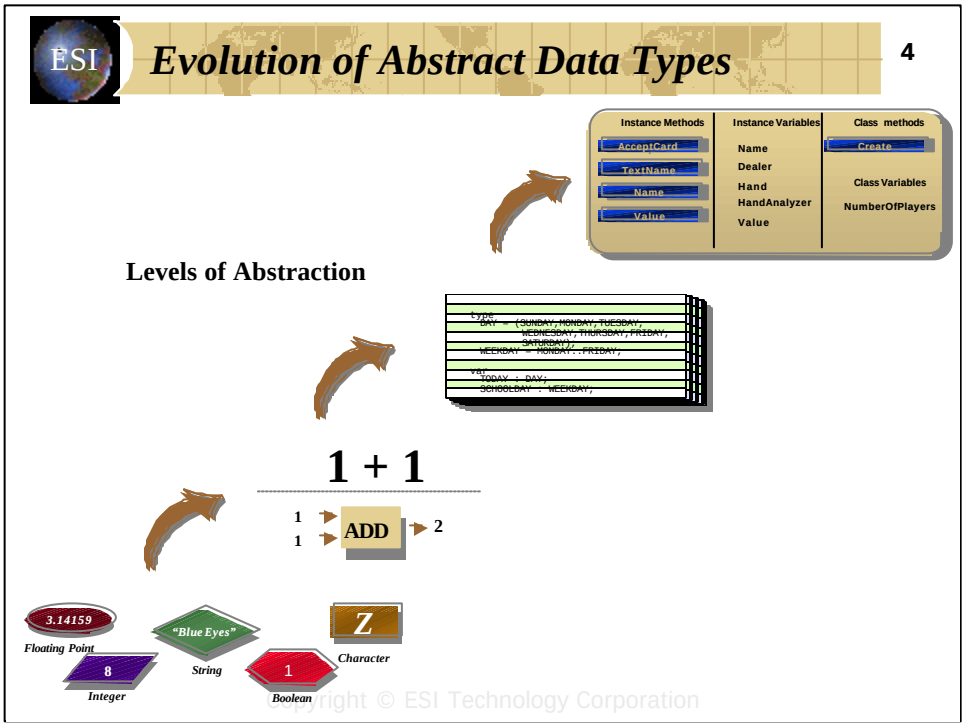
Read and understand the objectives of this lesson.

Assume you work for a company that sells computer games. After conducting a careful business analysis, your management assigns you the job of creating a computerized poker game.

You've played a little poker with your college roommates, however, to really understand the game, you've made arrangements to sit in on a game at a well known casino in Las Vegas. When you arrive at the casino, you are introduced to the Dealer. Upon learning that you are a top notch software engineer she asked if she could work more closely with you. She said she knew a couple programming languages but wants to learn more about object oriented programming. She heard you could get rich as an object oriented programmer. You agree to teach her object orientation as you build the system.

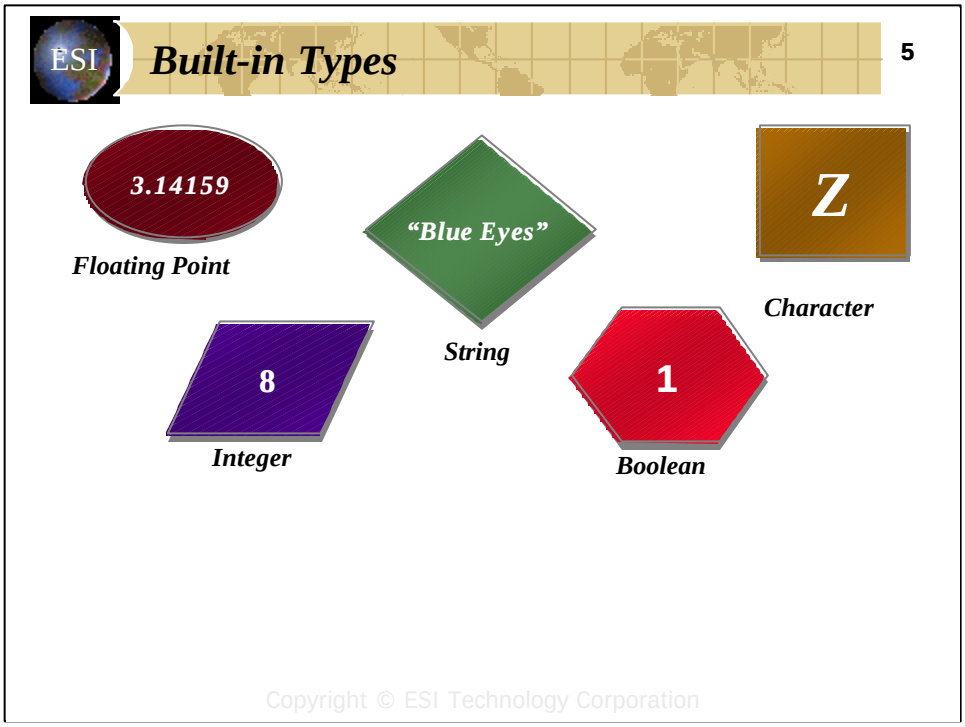
Of course, being a top notch object oriented engineer, you know that you should work within a well defined process that is iterative in nature. You also know from experience that building an object oriented system is labor intensive up front - during the Analysis and Design phases. Getting the initial design as generalized as possible will ultimately lead to a great deal of reusability and extensibility. You explain the object oriented approach to building systems to your new student and agree to meet daily for a lesson.

Since you discovered that the Dealer had some free time before her shift, you decided to give her some background that would help her understand the concepts as you proceed through the process of designing and developing the game.



You tell her that preliminary to understanding object orientation is the concept of Abstract Data Types (ADT) and that the concept of ADT's has evolved over the years as computing became more sophisticated. Today, we refer to them as 'objects'.

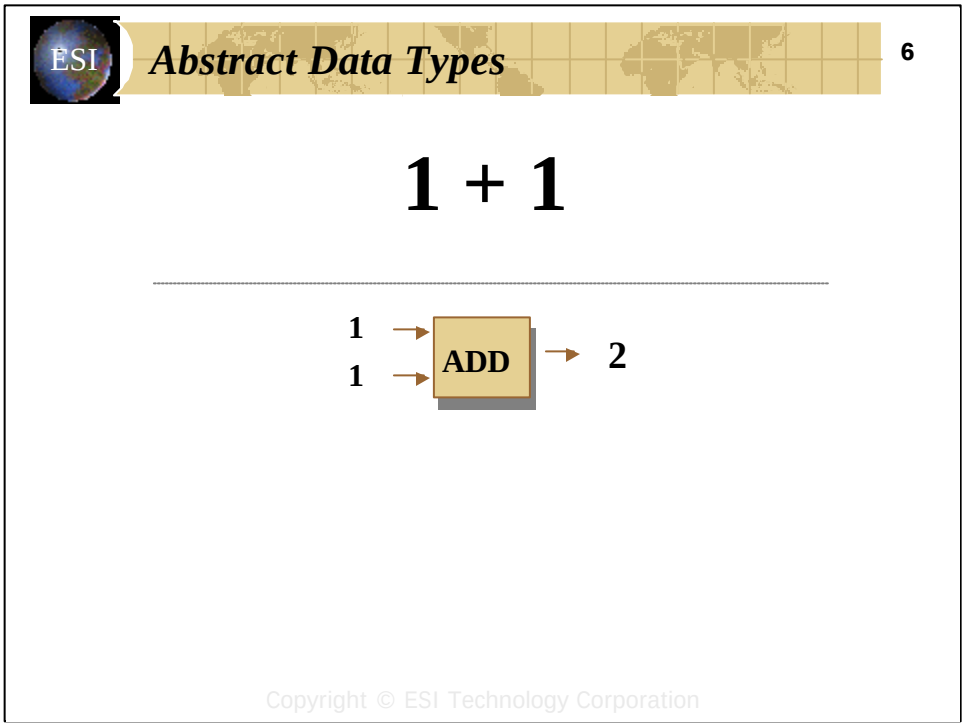
You draw the diagram above on a napkin and start to explain the concepts.



Let's start with the basic concept of a "data type". The need for data types arises whenever data must be categorized for a particular usage. Over the years, as computer languages evolved, the concept of data types evolved as well.

Today all computer programming languages support 'built-in' types. Built-in types are an integral part of the language specification. Some computer languages are referred to as 'weakly' typed. Weakly typed languages generally support one type. Other languages are referred to as 'strongly' typed. These languages support more than one type such as strings, integers, floating point numbers, etc. Generally, proper use of these types is enforced by an interpreter or compiler. Obviously, there is something to be said about both approaches.

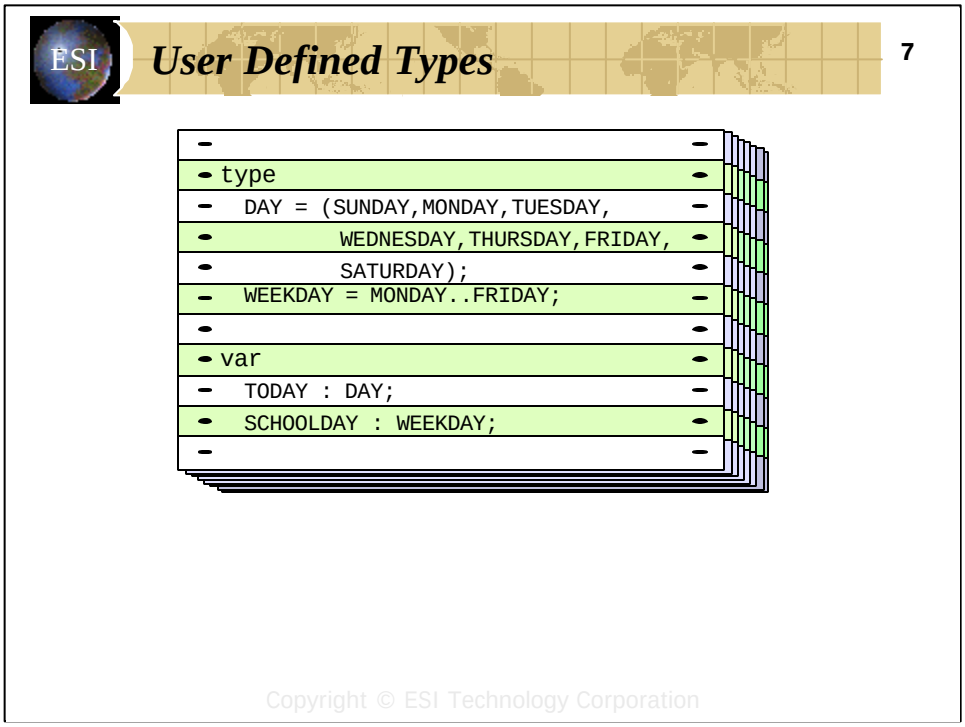
So far, so good! She said she understood this concept. Although most of her experience was with the MUMPS computer language, she had some experience with Basic. You told her you were a MUMPS programmer years ago but you moved on to an Object Oriented implementation of MUMPS. She said: "Cool! At least we speak the same base language."



You take one step up the abstraction ladder and explain just what an ADT is. In its most fundamental form, an ADT can be defined as the encapsulation of an operation (such as Add) and the data it operates upon.

You explain that we use the concept of an ADT more than we realize. When we mentally add two numbers, the numbers we learned in elementary school are operated upon the Add function that we also learned to produce a sum. When you mentally perform the addition the operation and underlying storage of numbers are hidden from the external world. They are encapsulated. For example, the dealer of the poker game has the responsibility of calculating the values of each player's hand and identifying the winner. In this case, you need to have knowledge of each card's value as well as how to combine them for their collective value.

You explain to her that the same thing happens in a computer. The numbers are supplied to an Add function. The compiler or interpreter transforms the external representation of the operation into an underlying representation that it understands. The Add operation is applied to the numeric representation to produce the sum. We only specify the correct syntax to have the result calculated. The subtle point in both cases is that the operation and data are encapsulated. The complexity of the operation is hidden from the user as it should be.



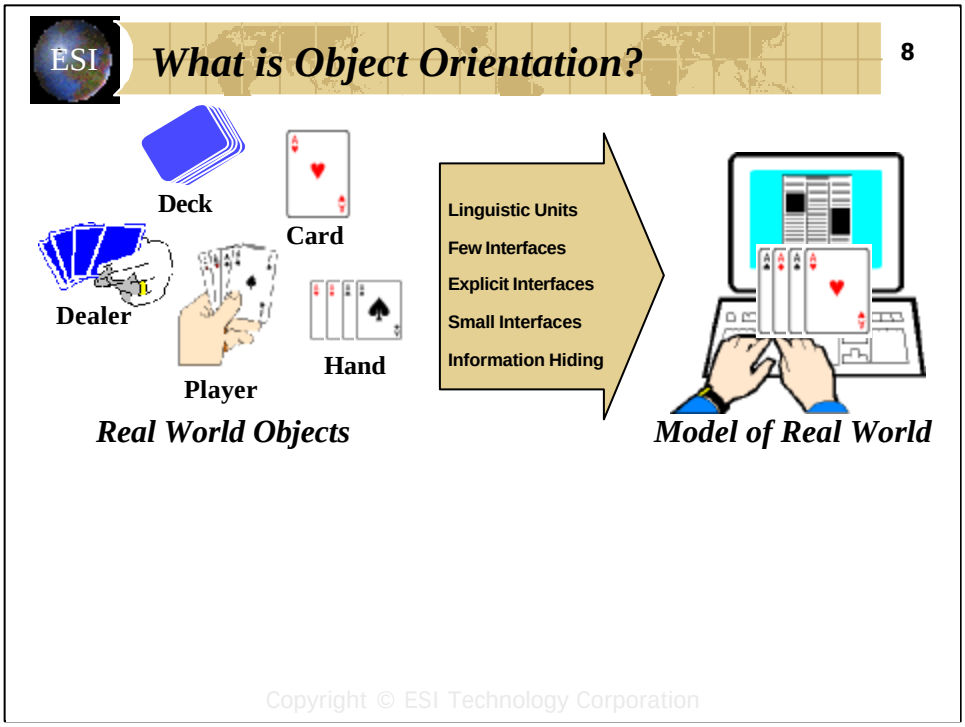
She indicated that she was still with you so you take one more step up the abstraction ladder and explain the concept of UDT's.

You explain that using data types built into a programming language proved very limiting. Eventually, languages supported User-Defined Types (UDTs) as illustrated by the Pascal example above. The user-defined type added more flexibility. However, one problem still remained - data could be manipulated by any code active within the scope of the data type, thereby destroying encapsulation.

You explain that encapsulation is often berated as something that is not necessary to enforcing true ADT behavior. **Although you may get along without it, not having it is in direct violation of the information hiding principle**. Not having it is one of the primary reasons for low software quality. She said that made sense since she had introduced some bugs into her first MUMPS program by not hiding a variable with the NEW command. It got changed later in the program and she wasted two hours debugging the problem. You explained that if those variables were properly scoped, she wouldn't have to worry about using a NEW command, the compiler would take care of generating the correct code. "But that's another story for later on."

You explain that the abstract data type (ADT) extended the UDT by specifying both the structure of the type and the operations of the data type. The ADT adds the concept of ***encapsulation.***

You ask her if she understands everything up until this point and she says she does. You think: Hmm! Maybe we've found a new employee here!



"So how does this relate to object orientation?" she asked. Of course the answer came easily to you.

"First," you said confidently "object orientation lets developers model real world objects as real world objects. It does not force them to create a data model and a separate functional module that operates upon that data like you had to in most procedural languages. It combines the code and the data so that when the code executes, it operates within the context of that object where the data is hidden. It picks up where the a concept of the ADT leaves off. Adding the concepts of classification and abstraction enables such features as inheritance and polymorphism. I know, I know! Big words! Have faith, we'll deal with them later on!"

"Second," you explain "object orientation is based upon sound software engineering principles. Essentially, it integrates time-proven software engineering practices with the real world modeling approach. Classification and abstraction go a long way to implement these principles."

You show her the diagram (that you just happened to have in your pocket). It illustrates the five real world objects we would find in a poker game. A Deck, Card(s), Dealer, Hand and Player(s).

You explain to her that a good object oriented development environment should provide an easy way to model real world objects and enforce the time-proven principles. You give her Bertrand Meyer's book on *Object-oriented Software Construction* and tell her to read Part 1.

You realize that you've gone beyond her current level of understanding and it's time to get practical. Doing is the best way to learn this kind of technical babble!

At the end of the first lesson, she asked you if you would like to sit in on her first shift as an observer. You said: "Great - the sooner the better!"

Being an object oriented guru, you know that the first task is to identify the objects of the game. You thought you would take notes on the game and then write up a description later when you return to your room. After you think you've got enough notes, you tell your new student that they should meet for lunch tomorrow and go over the description. Since she is the expert, she can verify that it is correct. Once that is done, you can work together to identify the objects and then their properties and operations.

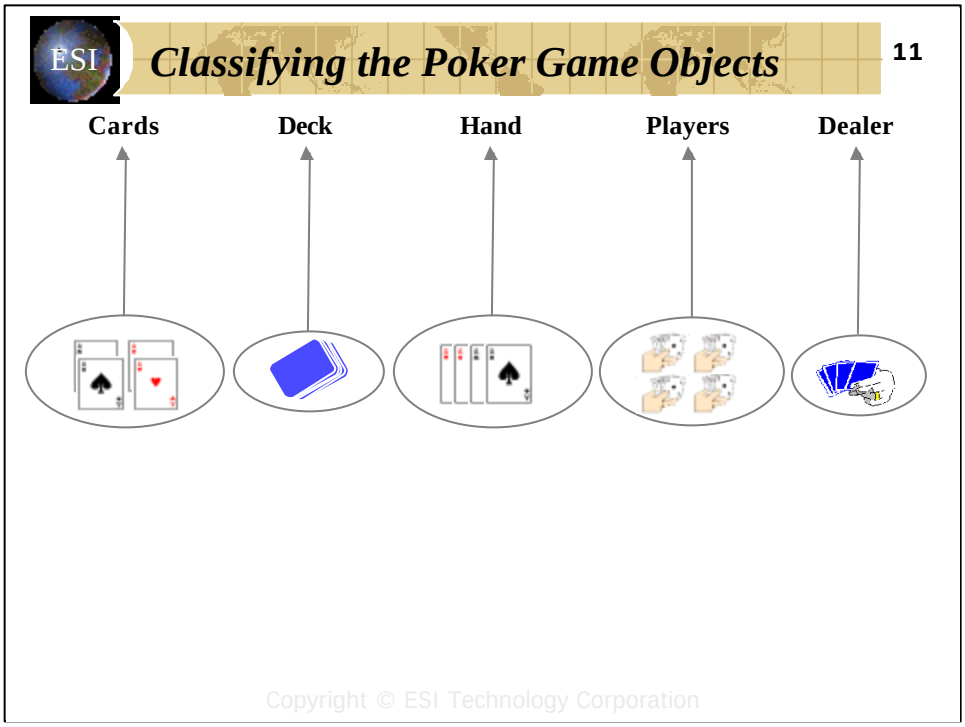
You go back to your room to write up the description.

The next day you meet for a working lunch and you read the description to your new student.

“ The game of poker has one dealer and up to four players. The dealer deals five cards from the deck to each player including herself. Each player, including the dealer, is given the opportunity to throw up to three cards away (discard) and ask the dealer for the number discarded. Each player is also given the opportunity to bet on his or her hand (value of combination of cards). The player with the best hand (highest poker value) wins the game.”

You explain to her that you are not a poker player and she said that was obvious! However, for the first iteration, this description will suffice. The goal is to identify the objects of the game as well as the properties and operations those objects can perform.

You ask her to identify all the objects by finding the nouns in the description. She reads through the description and identifies the following potential objects: Poker, Game, Dealer, Players Cards, Deck and Hand.



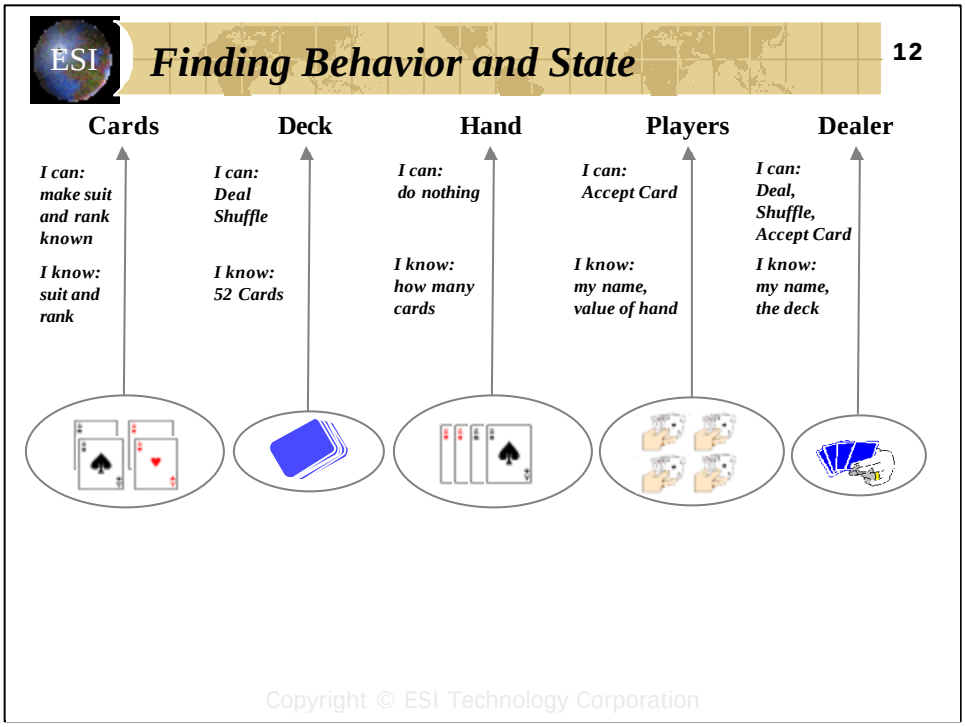
You explain that two important underlying concepts of object orientation are classification and abstraction. **Classification** is the process of organizing objects into groups that have like properties (attributes) and operations. **Abstraction** is the process of identifying and extracting out a quality of an object.

You explain that what we are doing is starting the classification process. By extracting the nouns, we have identified all the potential objects in the game. You said the next step is to look at each object and determine whether it actually contributes to the computerized model.

First, the words Poker and Game. She agreed that Poker is an object in the larger sense, however, for this project there is no need to implement it. If we were to implement different card games and reuse objects, it would then be a valid object. We decide that Game falls into the same category and is even more generic. She agreed with you that both names can be dropped from the list.

The base (or atomic) objects Cards, Dealer, Players and Hand are obvious. She feels they are absolutely necessary to the game. If Cards are relevant, then Deck is also relevant. Although the Deck is made up of a collection of card objects, it is an object in its own right.

You explain to your student that this is only the first pass and that we will make several passes before we actually get it “right”. This is the essence of iterative development.



Now that you've got the objects isolated, you tell her it is time to determine the object's behavior. An easy way to determine this on a first pass is to look at each object and say: "I'm an object - what do I know and what can I do?"

You start to apply this sentence to each object.

"I'm a Dealer and I know that I have a name, that I am in control of the deck and I can deal, shuffle and accept a card".

"I'm a Deck and I know that I have 52 cards and I can deal and shuffle"!

At this point she interrupts. "What do you mean when you say the Deck and the Dealer know how to deal and shuffle"? I thought the Dealer shuffled and the Deck just lets itself be manipulated complacently.

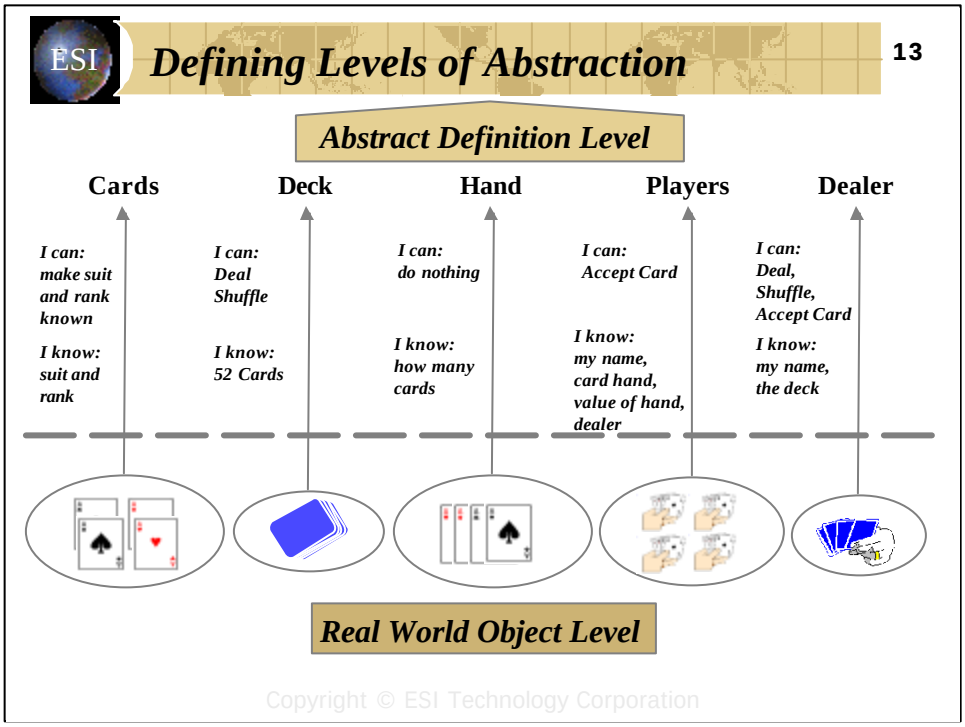
"Ah ha!," you say "Always remember the first heuristic of object oriented programming. One object NEVER does something to another, it ALWAYS requests the object to perform an action! That is, objects should know how to perform a function rather than having it imposed upon them. The Dealer merely initiates the shuffle and deal operations but the Deck actually knows how to perform those actions!"

She said, "I can understand that, let's go on".

"Ok, I'm a Hand and I know how many cards I have and I don't know how to do anything".

"I'm a Player and I know my name, the hand I'm holding, the value of the hand, who the dealer is and I can accept a card and place it in the hand I'm holding".

"I'm a Card and I know my rank, my suit and I can make them known to other objects".

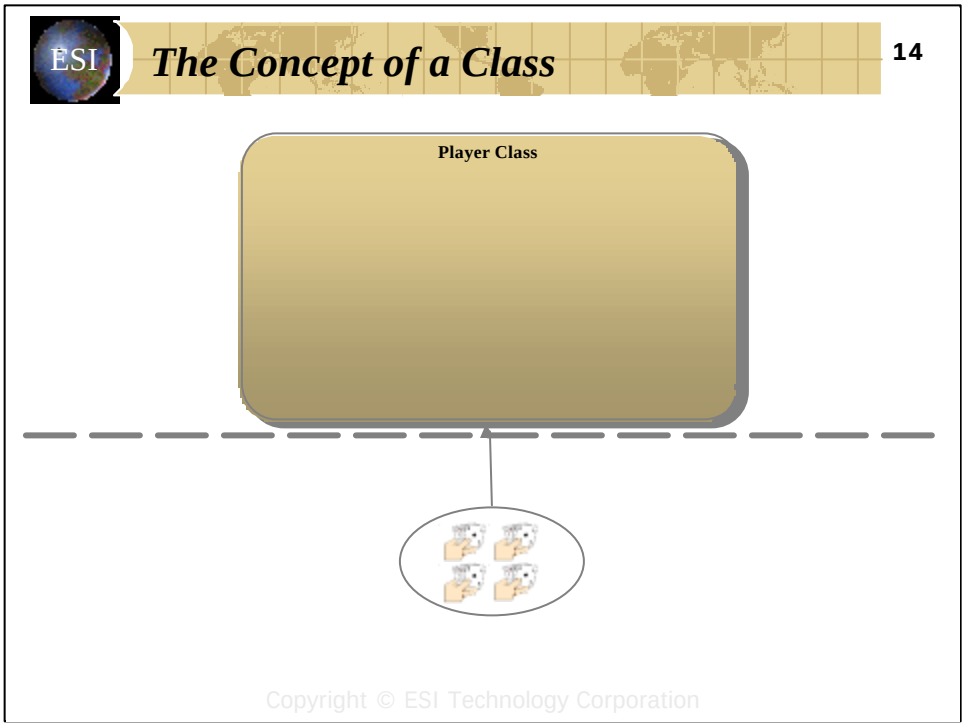


After some discussion you both agree that you have identified the fundamental behavior for each object - enough to start modeling the game.

Next you take the opportunity to explain what you have done using the diagram you developed. You draw a dotted line below the behavior lists and explain: “All objects below this line are real world objects that have properties and behavior.What we did was **abstract** out those properties and behavior, creating an abstract level. This level can be used to build what we call a definitional level. That is, we have enough information to start modeling the problem domain with an object oriented development tool.”

“However,” you say “it’s time to learn a few concepts.” Unfortunately she tells you she has to go to work and could we start the concepts tomorrow. Time flies when you’re having fun! You tell her to meet you for lunch again tomorrow for the next lesson. She asks you if you want to sit in on the game - this time as a participant. You say “Why not, what have I got to lose”?

You meet your new student for lunch again and apologize for being late. You explain that it took all morning to get another cash advance from the main office. Your boss didn’t understand how you could blow $200 on computer paper! You wonder why your new student wants to be a programmer - she should start her own casino!

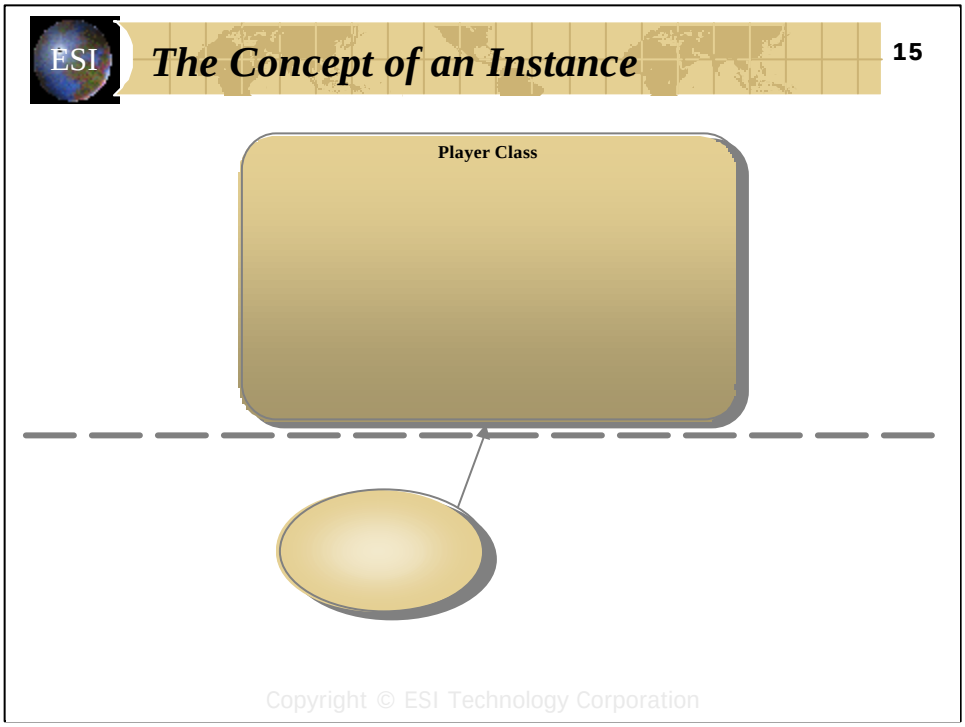


Picking up where you left off the previous day, you review the concepts of classification and abstraction. You explain that we've just started using these concepts. You ask her if she remembers what encapsulation is. She says "Yes, it is based on the concept of an Abstract Data Type and states that the data and the operations that work on that data are grouped together such that the user of that type does not need to know how operations are performed, just that they do what the say they will do."

You tell her she's correct, but now you would like to elaborate upon that concept. You use the diagram you drew the day before, but replace all information at the abstract level with a new diagram. You use Players to illustrate the concepts.

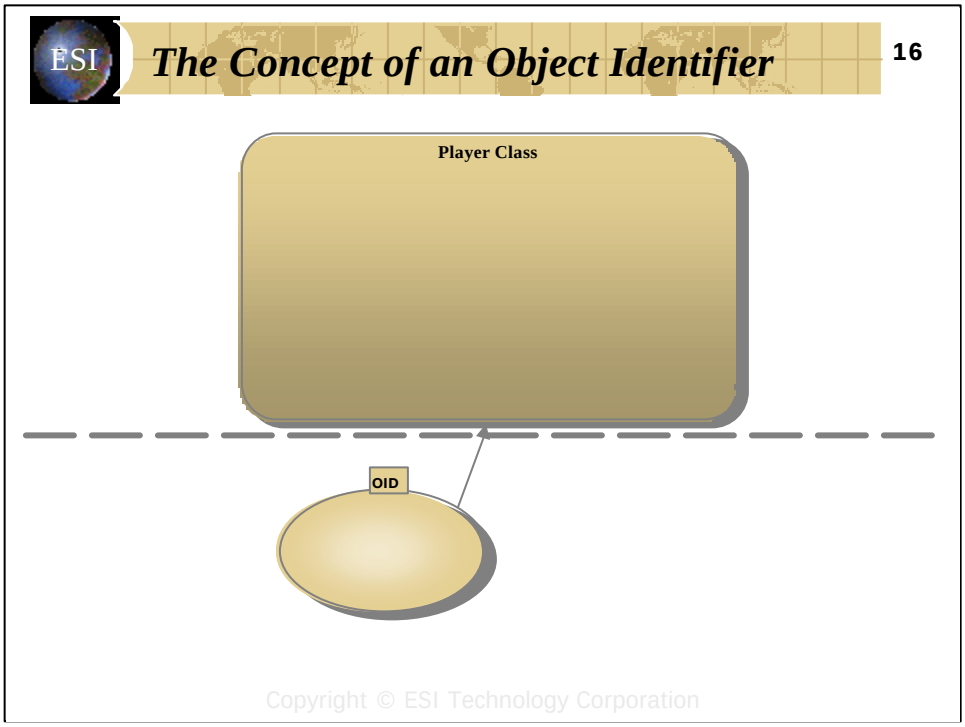
You tell her that most object oriented systems use the process of classification to organize information needed at the object definition level. Classification has been around for a long time. Aristotle and Plato wrote about it around 200 B.C. It is used extensively in biology to organize life forms here on earth. In fact, it becomes really powerful when the **Classes** are organized into **hierarchies**. This lets us extend the levels of abstraction to higher levels! You tell her not to worry about that, it will become 'intuitively obvious' later on. She looks at you quizzically! Obviously not a phrase you hear at the poker table.

You ask if she understands the concept so far. She said "Yes, it's intuitively obvious! Classes are used to organize the information we abstracted out of the real world objects. They are the organizing paradigm used in most object oriented development environments!"



Now you explain that if a class contains the definitional information, it must be there to create something. At this point you replace the real world depiction of Player objects with a new symbol and call it an **instance** of the class. That is, in all object oriented systems you must have the ability to create a real world object of that class and it is called an instance.

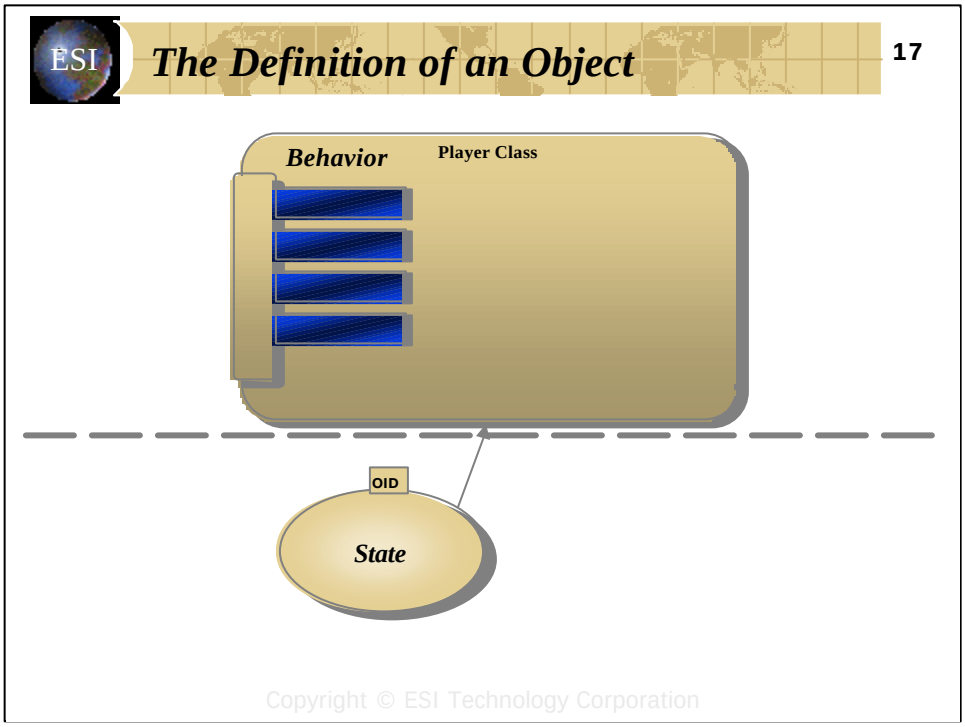
You then draw an arrow between the instance and the class and tell your card sharp friend that when an instance is created from the class, it must always know its maker! If it did not, it would not be able to call upon the class for services it may need. These instances are children and can do *only* what their maker knows. They possess no behavior of their own, only what is made available to them by their maker.



Your student scratches her head and says “Ok, I understand that the instance object must know its maker, the class, but how do other objects know it?”

“Very perceptive!” you say. You then explain that when the real world object is created by the class, it assigns it an unique **Object Identifier** commonly called an **OID**. The newly born object is known to all other objects by this OID.

You caution her on one point however. The OID is an internal (to the computer) device for identifying the object instance and it is almost always implementation specific.



Now you decide it is time to introduce your student to a more formal description of an object.

“At this point,” you say “we know the definitions of a class and an instance of the class. Let’s talk about the three commonly accepted aspects of an object.”

You then proceed to explain that an object 1) has **Identity**, 2) contains **State**, and 3) displays **Behavior**.

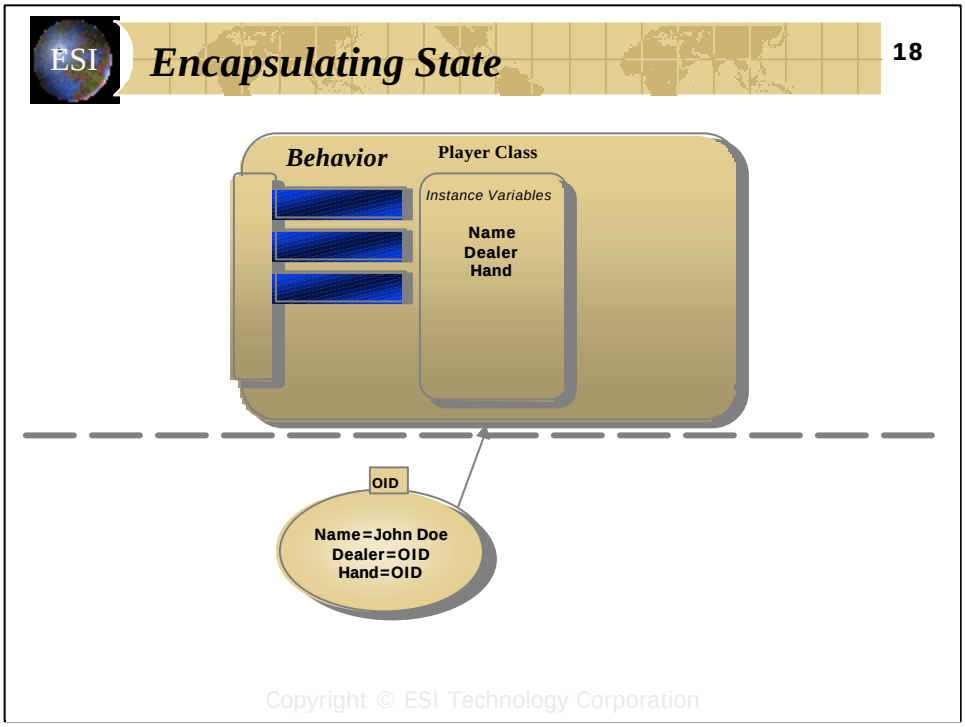
“Ah,” she says “I know what identity is, we just covered that! That is represented by the OID. But what does State and Behavior mean?”

Being the superb teacher you are, you take her back to the exercise where you answered the question: ‘I am an object, what do I know and what can I do?’. You ask her if she sees a connection.

“Let me guess” she says “the ‘what do I know’ is represented as the state of an object and the ‘what can I do’ is represented through the behavior of an object. Correct?”. “Right on!” you exclaim.

Now you go on to tell her that the state of an object is actually stored in the instance object, but, the behavior is stored in the class.

“Wow, now I see!” she says “Because ‘what I know’,the state, is data and is unique to each individual object, it must be stored in that object. Also, because ‘what I can do’ or behavior, is code, is in common to all instances of the class, and it must be stored in the class. Storing it in each object would be, well, not smart! Now I see why each instance must know its maker - without that link, it would not have access to the behavior. Cool!”



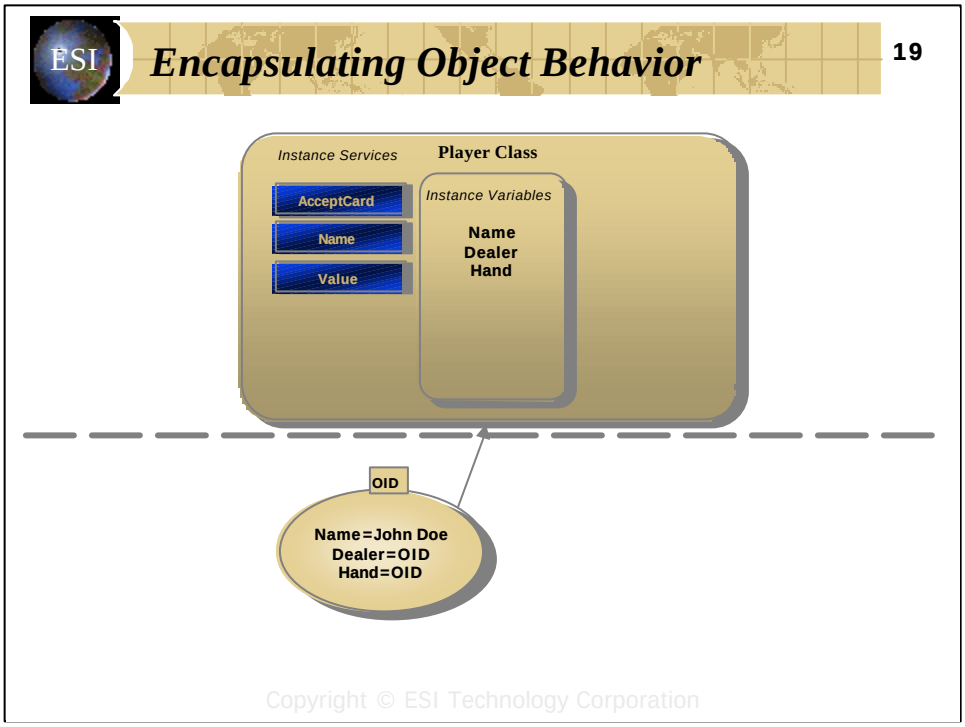
You ask her if she understands everything so far. She says you're the best teacher she's ever had. Ah, positive feedback, what a motivator. That may even make up for the 200 bucks she took from you!

She tells you she's only got an hour more before her shift starts and could you move on."Now," you say "let's talk about State. An object, if it knows anything, must have that information stored in it. In the case of the Player class, that information must be the name of the player, a pointer to the dealer and a pointer to the hand the player is holding." You then add this to the diagram.

You say to her "Remember your MUMPS programming experience? When you wrote the code to implement a function, it typically accessed local variables to get that data. However, that data was accessible to all other functions that would execute within that process. Notice that in our object oriented model, that data is confined to the object, that is, it is encapsulated! However, the concept of a symbol table that held name-value pairs is good. So, think of an object as a symbol table because that is what it is. More specifically, it is an 'instance' symbol table because it only holds name-value pairs that have a lifetime of the object. If the object dies, they die!"

"Got it!," she said "So my code can simply set the value to a name just like I do in MUMPS?". "Yes,"you replied "as long as the language you're using permits dynamic creation of the name-value pair. Some languages require you to 'declare' that name. This is even better since it lets the compiler know more about the name and value, consequently, it can do more for you if it has that knowledge. In our object model above, that knowledge would be stored in a table in the class. From now on we will refer to them as Instance Variables."

"Let me guess" she said "knowledge is power!".



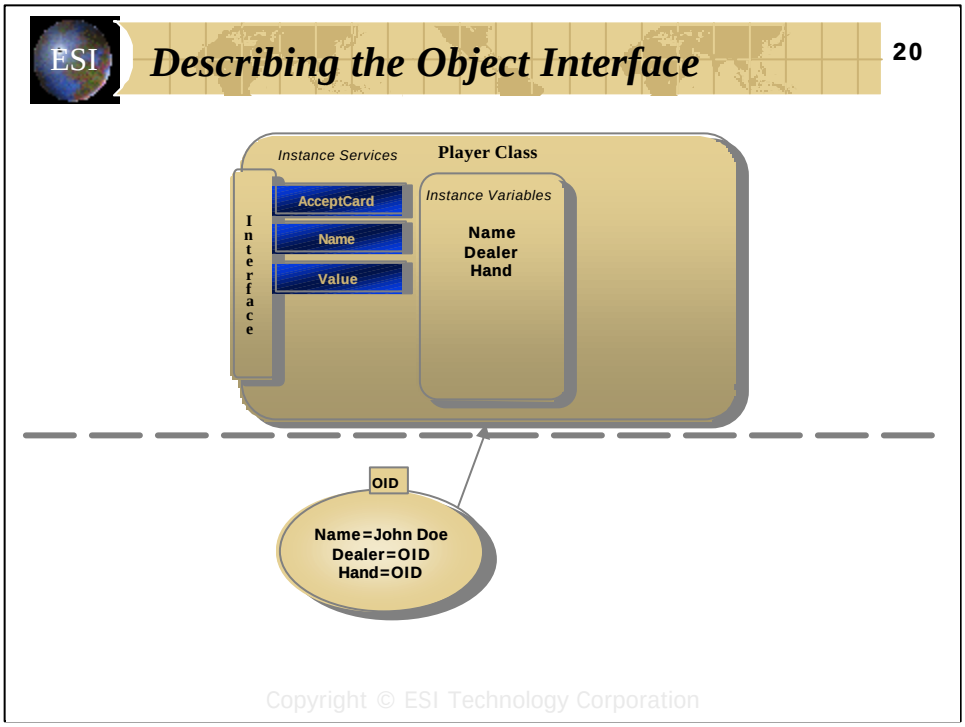
“We’re on a roll!” she said “I bet you will explain Behavior next.” “Well” you said, “the thought entered my mind!”

So you proceed to add some behavior to the diagram. You tell her the behavior is in common to all instances created from a class, therefore it is stored in the class. She tells you you’re repeating yourself and that she only has twenty minutes to her shift.

“Remember your experience with MUMPS again.You had two ways to store and execute code. In routines or in data. We’re not going to talk about the Xecute command and code in data. Although it may seem like a neat idea, within the object paradigm it succeeds admirably in ‘breaking encapsulation’. So let’s move on to the concept of a routine. Within the object paradigm, the routine equivalent is often referred to as a **method**. To stay with mainstream terminology, we will use the word method.”

“So,” she says “looking at your diagram, I bet that AcceptMethod, Name and Value are methods.”

“Well, close,” you say “AcceptMethod is a method in the strictest sense of the word. However, Name and Value are what we call **properties**. Notice that their names are the same as the instance variables Name and Value. Properties are a special kind of method in that they contain code to access the state of an object. Property methods have the function of exposing the state of a variable. Property methods are necessary because there is no other way to set or get the value of an instance variable because it is encapsulated. Properties are different from methods in that, often, the object oriented implementation will permit language operations on it like $GET for instance.”



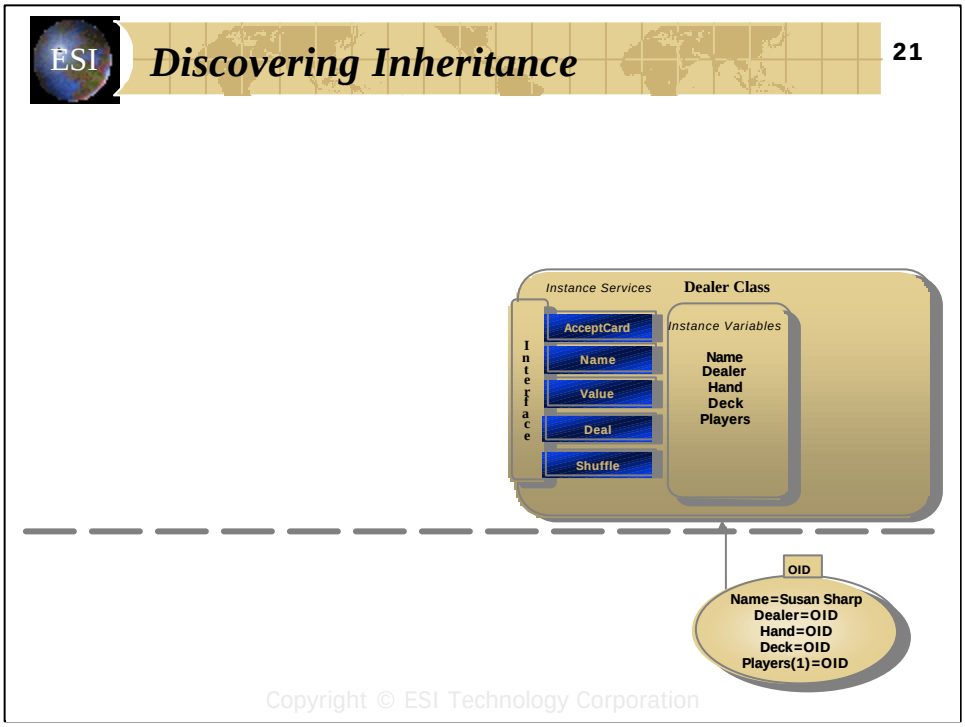
“One last thing for today.” you say “Some implementations of object orientation permit you as the programmer to organize your methods and properties through the concept of an **Interface**. An interface is simply a way of organizing behavior into logical groupings. Generally, you as the programmer can create these interfaces. However, there is always a primary interface which is the case if the implementation does not support the concept.”

“Ok, my shift is starting in 5 minutes,” she said “This has been a really great session. I want to buy a good book that explains these concepts - can you recommend one?”

“Of course there are a number of them. Check out the bibliography, especially the Khoshafian and Abnous book.”

“By the way” she interjected as she was walking out of the restaurant “would you like to sit on the game again tonight?”

“Forget it lady,” you replied, “I’m going to buy a six pack of beer, some potato chips and watch an old John Wayne movie back in my room. After that I’m going to start modeling the Player class on my computer. Meet you here tomorrow at the same time and we will continue talking about classification and abstraction. I’m sure we will also have time to talk about how objects talk to each other - messaging. What I want you to do tonight is draw a class structure for the Dealer using the ‘what I know’ and ‘what I can do’ information. Compare the Dealer to the Player and be ready to answer some questions. You’re going to earn that $200.”



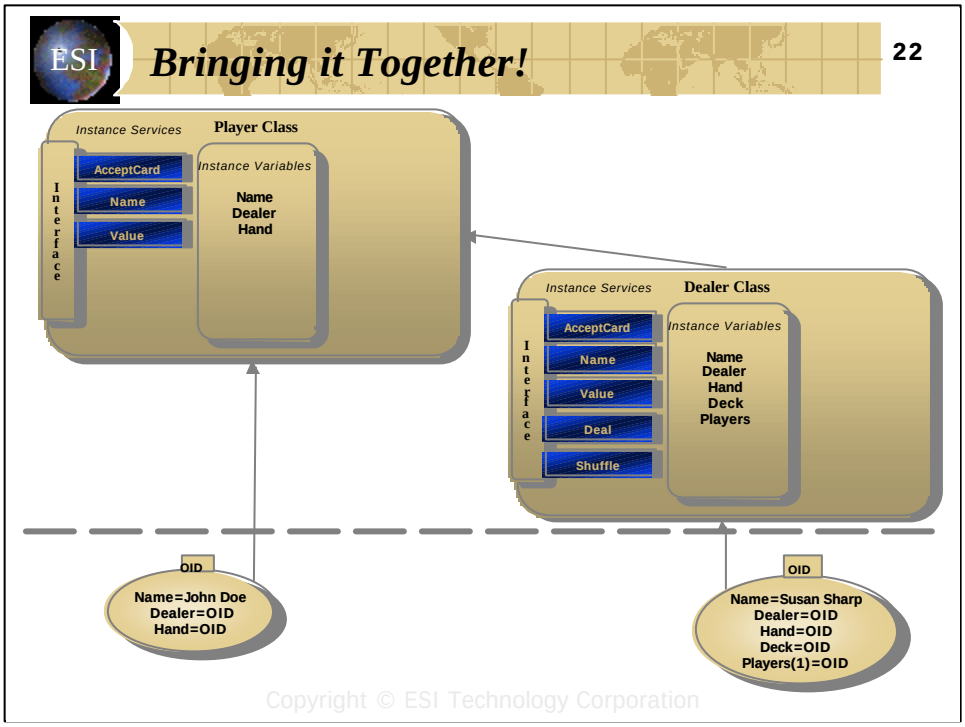
You meet again for lunch. You ask how her shift went and she said it was a slow shift - not as profitable as the day before. Right you thought - $200 less profitable!

You start by saying “I see from the diagram that you completed your homework. Tell me, after you finished the Dealer class did you compare it to the Player class? What did you notice? ”

“Interesting,” she said “I noticed that although the Dealer class had all of the state and behavior of the Player, it had additional state and behavior unique to a Dealer. The Dealer performs the same functions as a Player and a couple more, namely, the Dealer can Deal and Shuffle. Also the Dealer knows the same as the Player but knows more, like all the Players in the game and the Deck. I guess you can say that the Dealer is a much more specialized Player. Am I right?”

“ Absolutely right on!” you said. Wow, am I a good teacher or what!

“Hold on,” she said “I just had a premonition! What if you created another pointer that pointed from the Dealer class to the Player class. Then, what if the compiler could take all of the methods, properties and variables of the Player class and make them available to the Dealer Class so the Dealer class does not have to implement them. We could call this, let’s see, I know - **Inheritance**!”



She lays her diagram out on the table and quickly sketches the Player to the diagram and draws a pointer from the Dealer to the Player.

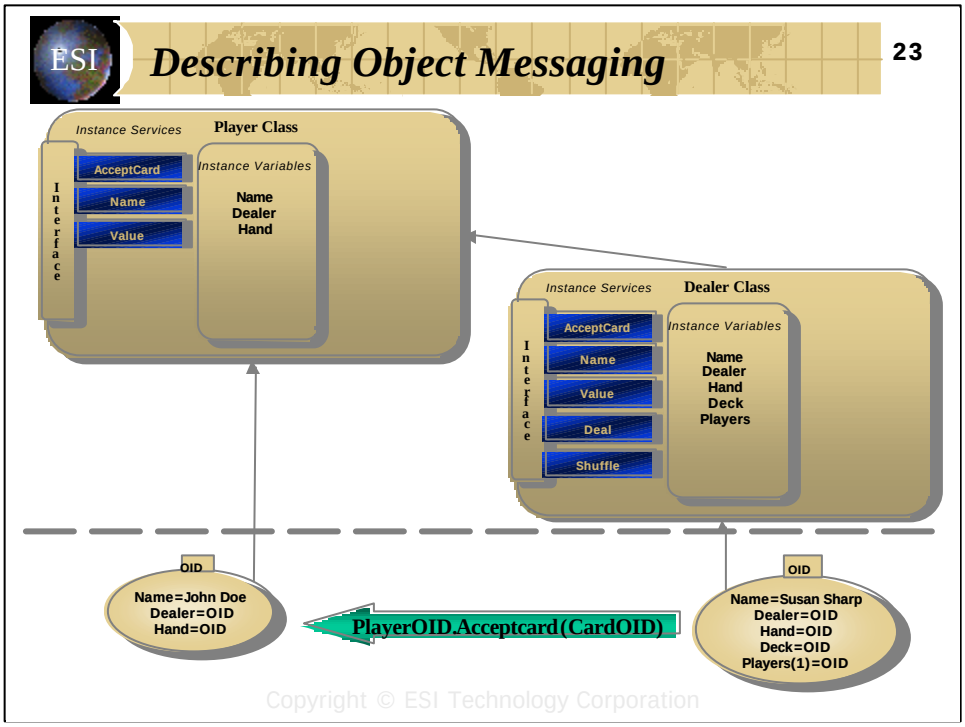
You look at her in amazement and say “Wow! How did you figure all that out?”

“I read Khoshafian and Abnous last night!” she replied smugly. “Also,” she said “I remembered what you taught me about being able to create levels of abstraction by linking classes together in a hierarchy. If you do this, it is ‘intuitively obvious’ that you could build your compiler to support inheritance.”

You thought, “Oh no, I’m creating a monster!” Next thing you know she will be matriculating at MIT.

“Also,” she said “now that you can inherit the method, properties and variable definitions, it occurred to me that you may want to **override** some of the inherited items at the Dealer level and specialize the operation. This would be a great feature to have. And, by the way, if you can override a service, why shouldn’t you be able to **promote** a service as well. Now I can see how classification, hierarchies and abstraction come together to form the basis for object orientation. Is there more? This is as far as I got in Khoshafian and Abnous!”

“Yes,” you replied “let’s move on to the concept of object messaging. Through this concept we can illustrate another very powerful feature of objects - polymorphism.”



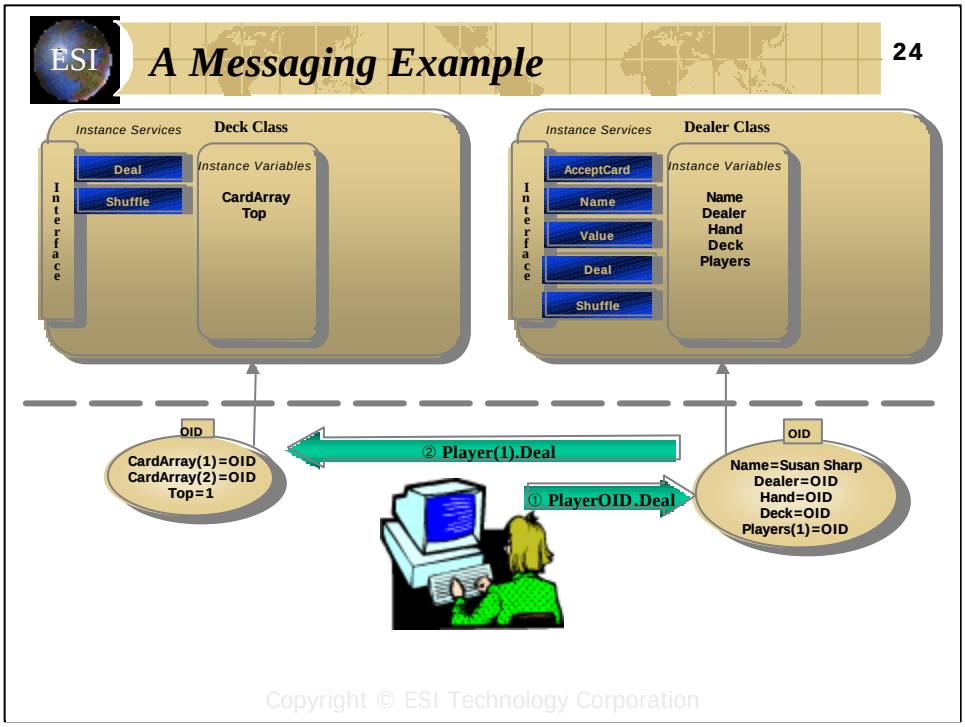
You realized that there may have been some things that were implied in your discussions so you decided to review some facts:

1) An object can only be manipulated via the services or operations defined within the interface of the class.

2) Instance methods and properties are the only mechanisms for manipulating an object's state. Methods (or operations): 1) Control an object's behavior 2) Can only be executed by sending a message to the object.

You then draw an arrow between the Dealer object and the Player object and put *PlayerOID.AcceptCard(CardOID)* on it. You explain that this is typical object messaging syntax and that Player is a variable that contains the Player objects OID, *AcceptCard* is the method and *CardOID* is the parameter that passes the OID of the Card object to the player object. You explain that messages identify the method or property to be executed and specifies the parameters to be used by the method. It's like a MUMPS Do command but much more flexible because it permits polymorphic behavior.You go on to state that the set of messages to which an object can respond is called its **protocol**, that is, the protocol consists of the methods and properties that will respond to a message in the interface.

"That's the second time you used the term 'polymorphic behavior'. What does that mean?" she asked.



You take out your pad of paper and draw a new diagram. On it you draw the Dealer class and an instance of the Dealer as well as the Deck class and instance of that class. You ask her to take notice of the fact that the two classes are totally different and that there is no inheritance involved.

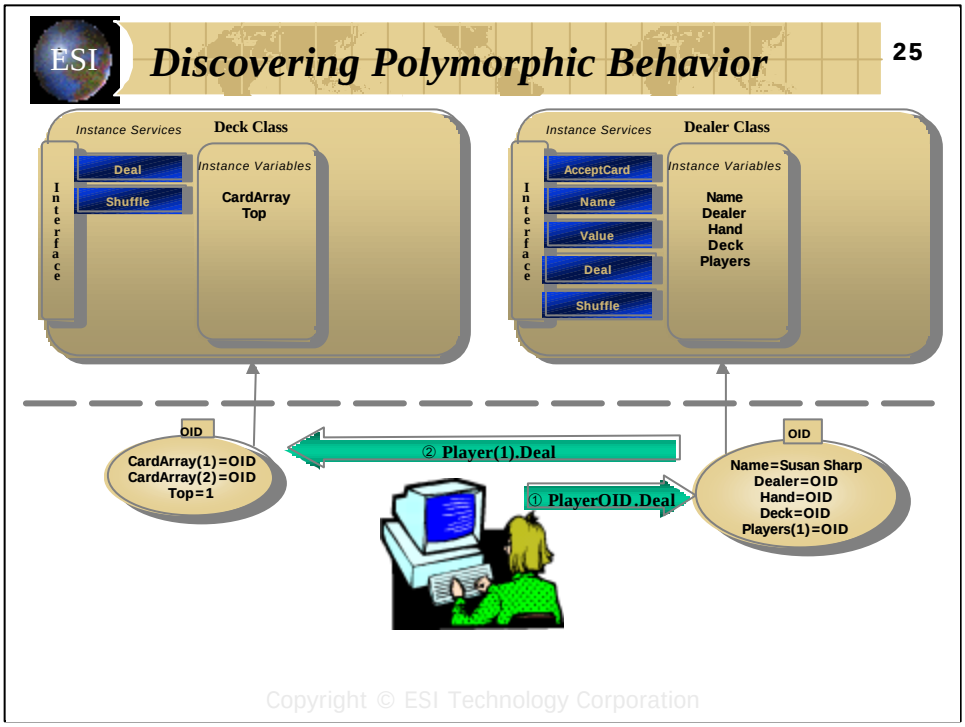
You also review what we called the first heuristic of object oriented programming: An object NEVER does anything to another object, it ALWAYS asks the object to perform that task.

“Assume” you said “that you are sitting at your computer and you have created an instance of a Deck and an instance of a Player. Notice that the Deck and the Player have the method Deal in their interfaces. Are they the same method or are they different? Think of the responsibilities of each object and then answer.”

“Hmm” she replied “I’ve been a dealer for a couple of years now and I’ve always thought of myself as the person in control of the Deck, that is, that it was always me doing the dealing. But one thing I’ve learned about object oriented design is that it is important to separate responsibilities. I will have to say that if the Dealer gets a message to Deal, its responsibility is to the Player (or players if there are more than one). In order to deal a card to a player it has to ask the Deck instance for a card. It is the responsibility of the Deck instance to pick the top card off the deck and return it to the Dealer object so it can give it to the player. Ah ha! I bet it does that through the AcceptCard message.”

She quickly draws the messages on the diagram and labels them 1 and 2.

You think - she’s really getting scary! Time to make a job offer!



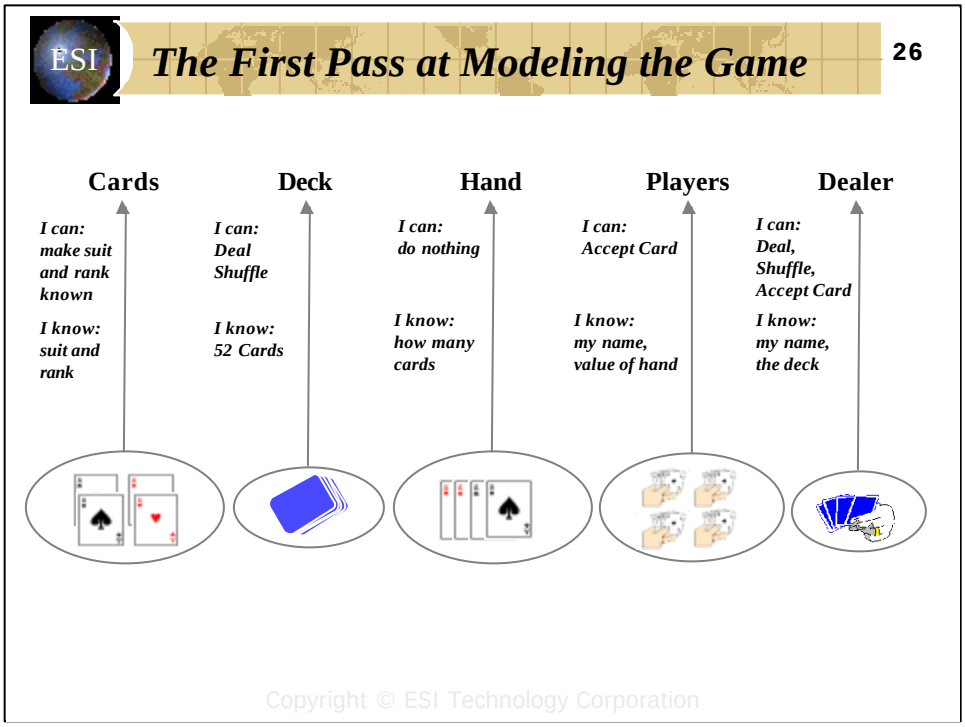
"Ok," you say "think! What just happened?"

"Well," she said "two messages with the same method name got sent to two different objects, however, the behavior of each message was different. The Player object simply received the message and delegated it to the Deck object which knew how to deal. Ok! Ok! I get it! **Polymorphic Behavior** - poly means many and morphic means forms. This must mean that in object oriented programming, you can send the same message to two different objects and as long as it's in each object's protocol, it will respond to it. The response may be the same or different. They may do something totally different or - they may work together to accomplish a common goal! Awesome!"

You congratulate her on the intuitive leap and go on to explain that the same thing happens with the Shuffle message. The responsibilities are the same as the Deal method.

"So," you say "what are the benefits of polymorphic behavior and what features of object orientation enable it?"

" The benefits," she replies "are obvious, polymorphic behavior permits homogeneous messaging between objects eliminating the need to write different methods and employing case statements to select the proper method. It also eliminates the need for multiple entry points into an object. The concepts of classification, inheritance and abstraction enable polymorphic behavior."

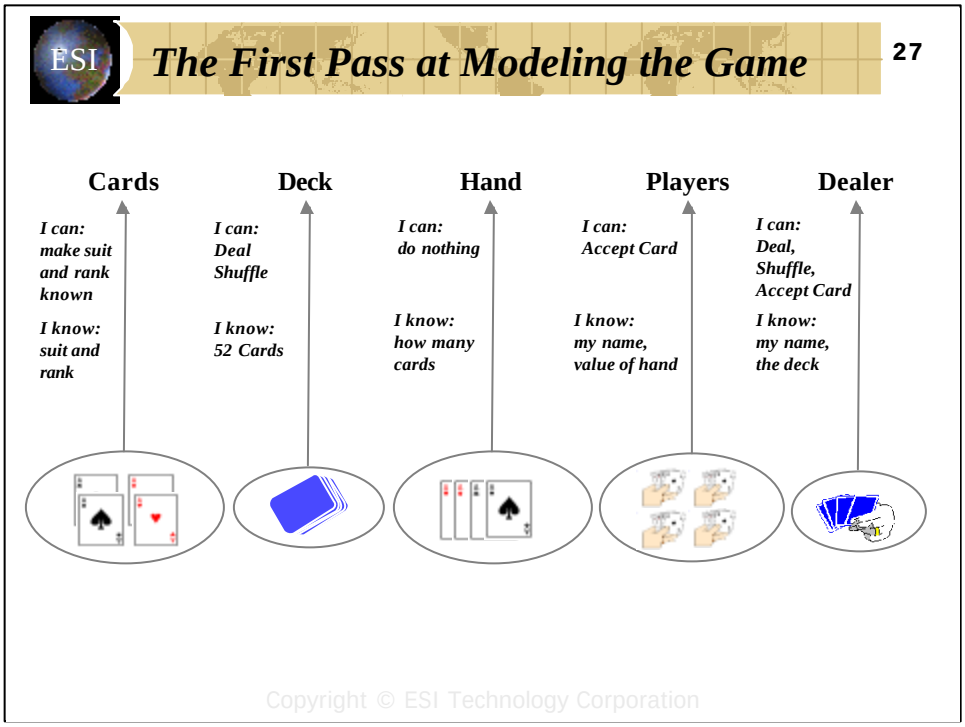


You tell her we are ready to actually create a model of the game. You ask her if she has a computer at home. She says she does. You give her instructions on how to download and install one of the two object oriented MUMPS systems.

Next you pull out the drawing you made of the poker game objects and explain that what you want her to do is use the documentation that comes with the system to:

1) Create a library to hold the classes she will create if the implementation supports libraries

2) Create the Card class and add the instance variables Suit and Rank. We determine that these must be initialized by user when an object is created. So the card's state will be set at creation by the caller and not modified after that. If your implementation permits, a create method can be implementated that will accept 2 input parameters. What are the valid values to be accepted for these parameters? It could be very fancy and allow the user to specify "H" or "Hearts" for the heart suit, etc. The rank could simply be a number 1-13 (ace to king) or 2-14 (2 to ace) or 2-10 and "Jack", "Queen", or "King". You can allow any types of inputs like this. Internally however, you can store it any way you want since the state is encapsulated. Internally you may end up storing 1-13 for Rank and "H","C","S", or "D" for Suit. How much flexibility you allow for the inputs and how you validate the input is an exercise that could be skipped if time is short.

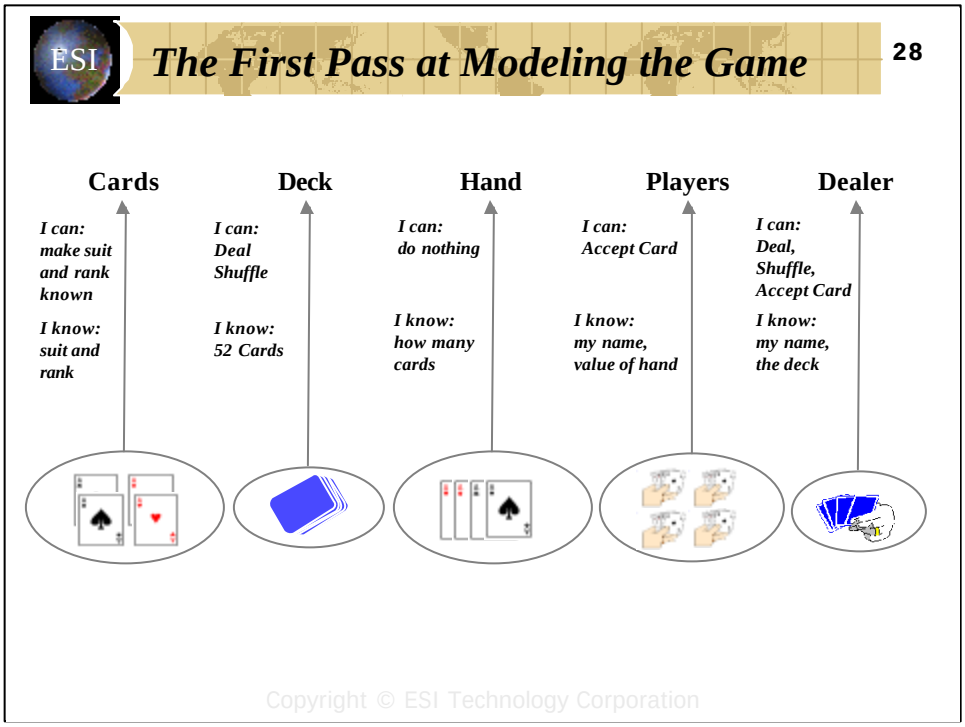


3) What happens if invalid inputs are found? Should the create command not fail, an object will be created. But we can set the state of the object to some state that would indicate an invalid object. For example, we could set Rank to –1 and Suit to "X" or something.

4) Using the implementation approach for interacting with the interpreter, create a card object with inputs. If you have an Object Browser that lets you look at the internal state of an object, go ahead and inspect the object.

5) Since the state is encapsulated and we haven't exposed the state yet, we then add the Rank and Suit properties (Read only). We could make these properties return the internal values as is, or transform them into text output. For example, instead of Suit and Rank being "H" and 1, we could return "Hearts" and "Ace". However, since the output of these properties will be used to compare against other cards to determine a player's hand, and the winning hand, the output should be presented in a way that will make those operations easier. Thus Rank should be exposed as a number 1-13 (or 2-14). Suit can be exposed as "H" or "Heart" without adding any more complexity to the comparison operations.

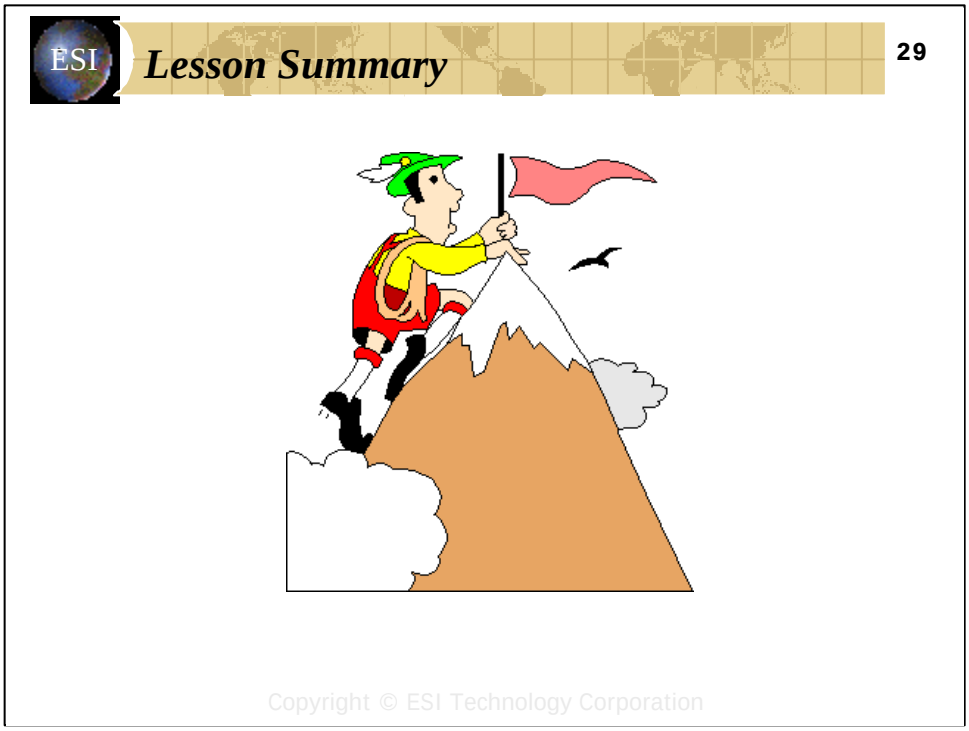
6) If you feel courageous, you may add a property called Color which is not directly tied to an instance variable, but uses the value of Suit variable to determine "Red" or "Black". This highlights how properties of an object can be from the internal state directly, or computed, etc. Since the game does not need a Color property (and the user can determine Color for themselves from the Suit property, it is not needed. But it's a good little side topic).



7) No using your implementations method to create several card objects, and test their properties. Use an Object Browser if you have one.

“Good grief,” she yelled, “my shift starts in 2 minutes! Got to go! I’ll work on that tonight. ”

You to need to implement the Card object so you can compare it to her implementation tomorrow.



Object orientation is well established within the field of Information Technology. When new applications are written, most organizations simply assume they will be written using this paradigm.

Object orientation is founded on the concept of Abstract Data Types. The ADT concept introduced you to the concept of Encapsulation. Object orientation is based on the concepts of Classifications, Hierarchies and Abstraction. These concepts come together to form an infrastructure of object orientation. Other powerful concepts that are a consequence of this coming together are Inheritance of operations and definitional information as well as Polymorphic Behavior.

Essentially, object orientation consists of the merging of solid, time-proven software engineering quality principles with the concepts of classification, hierarchies and abstraction.

The next lesson (Lesson 4) will proceed with the next iteration of the Poker Game. We will cover some more Object Oriented Concepts as needed.