Creating Java Programs with Greenfoot

Putting it All Together with Greenfoot

Overview

This lesson covers the following topics:

- Apply Greenfoot knowledge by creating a Java game

Putting It All Together

- In this lesson, you will review a case study and apply the skills you have learned in Greenfoot to program a Blackjack (or 21's) game.
- Open JF_scenarioB and save a copy to your computer.
- Save frequently as you progress through the lesson.

Components of BlackJack Scenario

- The BlackJack scenario includes the following classes:
  - One World class:
    - Table: BlackJack table to play the game on.
  - Three Actor classes:
    - Deck: Contains the behavior to deal the deck of cards to the dealer and players.
    - Card: Contains the behaviors associated with a playing card.
    - Button: After the player selects at least two cards, the player clicks this button to play the game and display the winner.

Create Variables in Card Class

- First, create the variables to associate to a Card.
- A Card instance will have the following properties:
  - A suit (hearts, clubs, spades, or diamonds).
  - A value in the range from 2 to Ace.
  - An associated numeric value in the range 2 to 11.
  - The ability to be flipped or not flipped.

Create the Card Variables

Open the code editor for the Card actor class. Create the variables as shown below to define the possible suits, numbers, and colors for the cards.

```
public class Card extends Actor
{
    /** The suits a card can belong to */
    public enum Suit {CLUBS, HEARTS, SPADES, DIAMONDS};

    /** The numbers a card can take */
    public enum Value {
      ACE(11), TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6), SEVEN(7), EIGHT(8), NINE(9), TEN(10), JACK(10), QUEEN(10), KING(10);

      private int numValue;

      Value(int numValue){
        this.numValue = numValue;
      }
    }
    /** The colours a card can be */
    public enum Colour {RED, BLUE}
```

Enter Remaining Card Variables

Enter the remaining variables as shown below. Save and compile the scenario.

```
/** The colours a card can be */
public enum Colour {RED, BLUE}

protected static final String CARD_IMAGE_LOCATION = "images/cards/";
private Colour colour;
private Suit suit;
private Value value;
private boolean flipped;
private boolean blank;
```

Enum Variables

- You may have noticed as you entered the variables that enum variables were used in this class.
- These variables are similar to an Array. They act like a list of values from which we can extract the value and position.

```
public class Card extends Actor
{
    /** The suits a card can belong to */
    public enum Suit {CLUBS, HEARTS, SPADES, DIAMONDS};

    /** The numbers a card can take */
    public enum Value {
      ACE(11), TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6), SEVEN(7), EIGHT(8), NINE(9), TEN(10), JACK(10), QUEEN(10), KING(10);

      private int numValue;

      Value(int numValue){
        this.numValue = numValue;
      }
    }
    /** The colours a card can be */
    public enum Colour {RED, BLUE}
```

Card Constructors

- This game will involve both blank and non-blank cards.
- The constructors for the Card class will include:
  - A blank card constructor which will show a face down card.
  - A constructor which show a face up card.

Create Card Constructors

Create the following two Card constructors below the variables you created previously to generate the blank and non-blank cards. Save and compile the scenario.

```
/**
 * Generate a blank card for placement in rows
 * @param blank whether it is an empty space(true) or a flipped card(false)
 */
public Card(boolean blank) {
    if (blank) {
        this.blank = blank;
        setImage(new GreenfootImage("images/cards/empty.png"));
    }else
    {
        setImage(new GreenfootImage("images/cards/blueflip.png"));
    }
}
```

```
/**
 * Generate a card with a colour, suit, and value
 * @param colour the colour of the card
 * @param value the value of the card
 * @param suit the suit of the card
 * @param flipped true if the card is face down, false otherwise
 */
public Card(Colour colour, Value value, Suit suit, boolean flipped) {
    this.colour = colour;
    this.value = value;
    this.suit = suit;
    this.flipped = flipped;
    draw();
}
```

Create the draw Method

Next, create the draw method. This will select and draw the image of the card based on its suit, value, and color. Save and compile the scenario.

```
/**
 * Select the image of the card based on its suit, value, and colour
 * and draw it.
 */
protected void draw() {
    String fileName = CARD_IMAGE_LOCATION;

    if(flipped) {
        fileName += colour;
        fileName += "flip";
    }
    else {
        fileName += value;
        fileName += suit;
    }

    fileName += ".png";
    fileName = fileName.toLowerCase();
    setImage(new GreenfootImage(fileName));
}
```

Create Methods to Return Card Information

Next, create the methods shown here and on the next slide to return information about the cards: if the card is flipped, color, value, numeric value, and suit. Save and compile the scenario when finished.

```
/**
 * Set whether the card is flipped over or not
 * @param flipped true if card is face down, false otherwise
 */
public void setFlipped(boolean flipped) {
    this.flipped = flipped;
    draw();
}
```

```
/**
 * Determine whether the card is flipped over or not
 * @return true if the card is face down, false otherwise
 */
public boolean isFlipped(){
    return flipped;
}
```

```
/**
 * Get the colour of the card
 * @return the colour of the card
 */
public Colour getColour() {
    return colour;
}
```

Create Remaining Card Class Methods

```
/**
 * Get the value of the card
 * @return the value of the card
 */
public Value getValue() {
    return value;
}
```

```
/**
 * Get the numeric value of the card
 * @return the value of the card
 */
public int getNumValue() {
    return value.numValue;
}
```

```
/**
 * Get the suit of the card
 * @return the suit of the card
 */
public Suit getSuit() {
    return suit;
}
```

Create Variables for Deck of Cards

- Next, in the Deck class, you will create the variables that will be associated to the deck of cards.
- The deck of cards:
  - Will hold 52 instances of the Card object.
  - Will store them in an ArrayList (similar to an array, but more flexible when adding information).

Deck ArrayList and Variables

Open the Deck class. Create the ArrayList and variables as shown below. Save and compile the scenario.

```
public class Deck extends Actor
{
    private ArrayList<Card> cards;
    /** The number of sets of each of the suits. */
    private int spades, clubs, hearts, diamonds;
    private Card.Colour colour;
    /** The location of the picture of an empty deck (outline of a deck.) */
    private static final String EMPTY_DECK = "empty.png";
    private int showNum;
```

Deck Constructor

Next, below the variables, create the Deck constructor. This will create 52 cards of different values and suits and store them in the ArrayList. Save the scenario, but do not compile it.

```
/**
 * Create a customised deck of a certain colour
 * @param colour the colour of the deck and the suits required
 */
public Deck(Card.Colour colour, int spaces, int clubs, int hearts, int diamonds)
{
    this.colour = colour;
    this.diamonds = diamonds;
    this.clubs = clubs;
    this.spades = spades;
    this.hearts = hearts;
    setColour();
    fill();
    shuffle();
}
```

Deck Behaviors

- Next, code the following behaviors for the Deck. These will be inherited by all Deck instances (although you will only use one deck).
- Behaviors are as follows:
  - Fill the deck.
  - Shuffle the deck.
  - Set the color of the deck.

Create fill and shuffle Methods

In the Deck class, create the fill and shuffle methods.

```
/**
 * Fill the deck with a complete set of cards. Get rid of any cards
 * still in the deck.
 */

public void fill()
{
    cards = new ArrayList<Card>();
    for(Card.Suit suit : Card.Suit.values()) {
        for(Card.Value value : Card.Value.values()){
            cards.add(new Card(colour, value, suit, false));
        }
    }
    setColour();
}
```

```
/**
 * Shuffle the deck
 */
public void shuffle()
{
    Collections.shuffle(cards);
}
```

Create setColour Method

Below the shuffle method, create the setColour method.

```
/**
 * Set the deck to a certain colour
 */
private void setColour()
{
    if(colour==Card.Colour.BLUE) {
        setImage(new GreenfootImage("images/cards/blueflip.png"));
    }
    else {
        setImage(new GreenfootImage("images/cards/redflip.png"));
    }
}
```

Create drawCard and getSize Methods

Beneath the setColour method, create the drawCard and getSize methods.

```
/**
 * Draw a card from the deck.
 * @return the card that's been drawn, or null if no cards are left.
 */
public Card drawCard()
{
    if(getSize()==0) return null;
    Card card = cards.get(0);
    cards.remove(card);
    if(getSize()==0) {
        setImage(new GreenfootImage(Card.CARD_IMAGE_LOCATION+EMPTY_DECK));
    }
    return card;
}
```

```
/**
 * Get the size of the deck
 * @return the number of cards left in the deck
 */
public int getSize()
{
    return cards.size();
}
```

Create drawFlippedCard Method

Beneath the getSize method, create the drawFlippedCard method. Save and compile the scenario.

```
/**
 * Draw a card from the deck but flip it
 * @return the card that's been drawn, or null if no cards are left.
 */
public Card drawFlippedCard()
{
    if(getSize()==0) return null;
    Card card = cards.get(0);
    cards.remove(card);
    if(getSize()==0) {
        setImage(new GreenfootImage(Card.CARD_IMAGE_LOCATION+EMPTY_DECK));
    }
    return card;
}
```

Creating the Gameplay

- The objects have been created and populated with variables, methods and behaviors.
- The next step is to create the following gameplay behaviors in the Table class:
  - The dealer takes a hand, which is hidden from the player.
  - The player can then select up to 5 cards aiming to score as close to 21 as possible.
  - When the player is ready to check their cards against the dealer, they click the Play Cards button.
  - The dealers hand is shown and the winner of that hand declared.

Table Variables

- In the Table class, you will create the variables that will be associated to the Table and used in gameplay.
- You will use Arrays to hold the player and dealer hands, as well as a number of counters to control the gameplay.

Create Table Variables

Create the following variables in the Table class. Save and compile the scenario.

```
public class Table extends World
{

    // Deck Object
    public Deck mainDeck;

    //Button used when player ready to check hand against dealer
    public Button playButton;

    //Cards used within Gamplay
    public Card card,blandCard,dealer1,dealer2,placeCard,tempCard;

    //Players Cards Array
    public Card[] cardGrid = new Card[5];

    //Card Array used to hide dealers Cards
    public Card[] blankGrid = new Card [5];

    //Dealers Cards Array
    public Card[] dealerCard = new Card[5];

    //Various counts used within game
    public int dealerCount = 2;
    public int playerCount = 0;
    public int dealerTotal = 0;
    public int playerTotal = 0;
```

Setting Up the BlackJack Table

- The BlackJack table (the Table class) requires the following set up each time the scenario is initialized:
  - The deck of cards to draw from.
  - Five spaces to hold the dealer cards.
  - Five spaces to hold the players cards.
  - A button to end the round and begin the card checking.
- This set up should take place automatically each time the scenario is initialized.

Edit Table Class

- Open the code editor for the Table class and edit the Table constructor to increase the playing surface area:

```
Super(600, 400, 1);

within the public Table() { method to

Super(800, 600, 1);
```

- Then, enter the code shown on the following slide in the Table constructor to construct the objects on the table.
- When finished, save the scenario, but do not compile.

Table Constructor

```
/**
 * Constructor for objects of class Table.
 *
 */
public Table()
{
    //Create a new world with 600x400 cells with a cell size of 1x1 pixels.
    super(800, 600, 1);

    //Create the deck of cards to draw from
    mainDeck = new Deck(Card.Colour.BLUE, 1, 1, 1, 1);
    addObject(mainDeck, 500, 500);

    //Add the play cards button which when pressed will check player cards against dealer cards
    playButton = new Button();
    playButton.setImage("images/playcard.png");
    addObject(playButton, 570, 150);

    //Draw the dealers hand of cards
    dealerHand();

    //Set up the spaces for the players hand of cards
    for (int row = 0; row < 5; row++){
        //card = new Card(Blank=true);
        cardGrid[row] = new Card(true);
        addObject(cardGrid[row], 100 + 90*row+1, 150);
    }

    //Start the scenario and await player responses
    Greenfoot.start();
}
```

Coding Dealer Gameplay

- Specifications for the BlackJack dealer:
  - Fully automated card dealing.
  - Results in a hand of cards for the dealer which may be more than 21 but never less than 15.
  - Two cards will be drawn and if they are less than 15, another will be drawn until the total is greater than 15.
  - The number of cards drawn will be shown to the player but they will be face down.

Create dealerHand Method

Beneath the Table constructor, create the dealerHand method shown on this slide and the next slide to code the dealer gameplay. Save and compile the scenario.

```
public void dealerHand()
{
    //Draw first dealer card
    dealerCard[0] = mainDeck.drawCard();

    //Draw second dealer card
    dealerCard[1] = mainDeck.drawCard();

    //Calculate initial dealer total to decide whether to take more cards
    dealerTotal = dealerCard[0].getNumValue() + dealerCard[1].getNumValue();

    //Play dealer hand to draw more cards if required
    //- threshold is set at 15 but this could be lowered or raised
    if (dealerTotal!=21)
    {
        while((dealerTotal < 15) && (dealerCount < 5)){
            dealerCard[dealerCount] = mainDeck.drawCard();
            dealerCount++;
            //Start at row 2 since first two cards are already totaled.
            for (int row = 2; row <dealerCount; row++){
                dealerTotal = dealerTotal + dealerCard[row].getNumValue();
            }
        }
    }
```

Create dealerHand Method

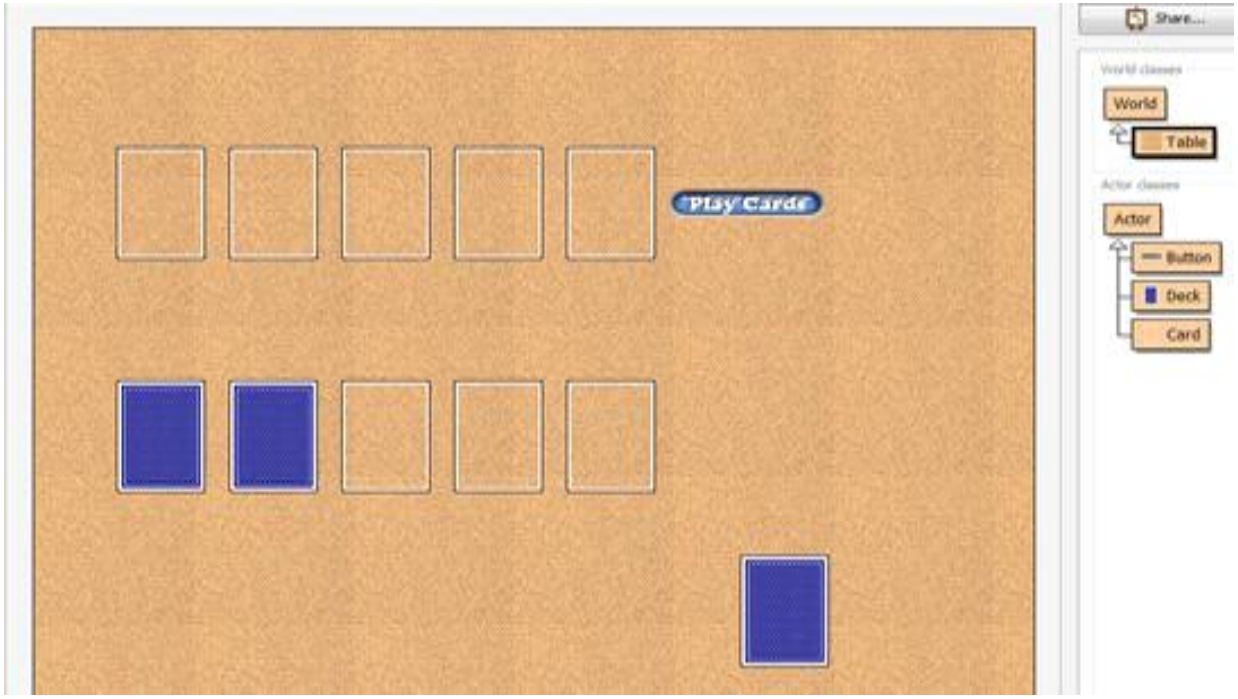
```
    //Set blank cards for remaining dealer cards
    for (int row = dealerCount; row < 5; row++){
        dealerCard[row] = new Card(true);
    }

    //Set and place turned cards for number of dealer cards
    for (int row = 0; row < dealerCount; row++){
        //card = new Card(Blank=true);
        blankGrid[row] = new Card(false);
        addObject(blankGrid[row], 100 + 90*row+1, 350);
    }

    //Set and place blank cards for number of dealer cards
    for (int row = dealerCount; row < 5; row++){
        //card = new Card(Blank=true);
        blankGrid[row] = new Card(true);
        addObject(blankGrid[row], 100 + 90*row+1, 350);
    }
}
```

Scenario Example

Your scenario should look similar to the following. Reset the scenario a few times to test that the dealer draws a different number of cards.



Coding End of Gameplay

- Next, enter the endGame method in the Table class to code what happens when the player clicks the Play Cards button. This method:
  - Starts the endGame method
  - Counts the players' total
  - Shows the dealer's cards
  - Checks to see who wins
  - Updates the Play Game button image to display winner
  - Stops the game
- Enter the endGame method as shown on the following slide. Save and compile the code.

Create endGame Method

```
public void endGame()
{
    //Calculate the playerTotal
    for (int row = 0; row <playerCount; row++){
        playerTotal = playerTotal + cardGrid[row].getNumValue();
    }

    //Show dealer cards
    for (int row = 0; row < 5; row++){
        addObject(dealerCard[row], 100 + 90*row+1, 350);
    }

    //Check who wins
    if (dealerTotal >= playerTotal && dealerTotal <=21){
        //Dealer wins
        playButton.setImage("images/dealerwins.png");
    }
    else {
    if (playerTotal > dealerTotal && playerTotal <=21){
        //Player wins
        playButton.setImage("images/playerwins.png");
    }
    else{
    if (playerTotal <=21 && dealerTotal > 21){
        //Player wins
        playButton.setImage("images/playerwins.png");
    }
    else {
        //Dealer wins
        playButton.setImage("images/dealerwins.png");
    }
    }
    }
    Greenfoot.stop();
}
```

Coding the Game's Action

- Finally, code the following specifications in the Table class's act method for how the BlackJack table will act when the player clicks on objects:
  - React to clicks on the player's hand to deal a card, but only up to 5 cards.
  - A click anywhere in the hand will place the card in the next open spot.
  - React to clicks on the Play Cards button, but only if the player has played at least two cards.

Code Table Actions in the Act Method

In the act method for the Table class, enter the following code on this slide and the next slide. Save and compile the scenario.

```
/**
 * If the deck is clicked on, then draw a card from it (unless it's empty.)
 */
public void act()
{
    //React to play button being clicked.  No action will
    //take place until the player has placed two cards
    if(Greenfoot.mouseClicked(playButton) && playerCount >=2) {
        endGame();
    }

    //Reac to mouse click on any of players cards
    //Next available place will be filled
    if(Greenfoot.mouseClicked(cardGrid[0]) && mainDeck.getSize()>0 && playerCount <5){
        cardGrid[playerCount] = mainDeck.drawCard();
        playerCount++;
    }

    if(Greenfoot.mouseClicked(cardGrid[1]) && mainDeck.getSize()>0 && playerCount <5){
        cardGrid[playerCount] = mainDeck.drawCard();
        playerCount++;
    }
```

Remaining Table Actions in the Act Method

```
    if(Greenfoot.mouseClicked(cardGrid[2]) && mainDeck.getSize()>0 && playerCount <5){
        cardGrid[playerCount] = mainDeck.drawCard();
        playerCount++;
    }

    if(Greenfoot.mouseClicked(cardGrid[3]) && mainDeck.getSize()>0 && playerCount <5){
        cardGrid[playerCount] = mainDeck.drawCard();
        playerCount++;
    }

    if(Greenfoot.mouseClicked(cardGrid[4]) && mainDeck.getSize()>0 && playerCount <5){
        cardGrid[playerCount] = mainDeck.drawCard();
        playerCount++;
    }

    //Update cards drawn and placed by player
    for (int row = 0; row < playerCount; row++){
        addObject(cardGrid[row], 100 + 90*row+1, 150);
    }
}
```

Instructions to Play the Game

- You are finished coding the game. Instructions to play the game:
  - Run the scenario to start the game.
  - Click inside the first empty card space in the player area to play a card.
  - When you are satisfied with the cards you have drawn, click Play Cards to see if you beat the dealer. The Play Cards button will display the winner.
  - Play a few games to try and beat the dealer.
- Review the code in the completed scenario example, JF_scenarioBworking, if you have problems with your scenario.

Additional Features

- There are many features which could be added to this scenario. Can you think of any?
- Write them in your journal and try to code them for yourself. As you practice coding, you will become more skilled.

Summary

In this lesson, you should have learned how to:

- Apply Greenfoot knowledge by creating a Java game