Quentin Dye's Portfolio

Central New Mexico Community College
Advisor—Robert Garner
Fall 2021

Table of Contents

1.0 Class Construction

1.1 Java:

```java
package edu.cnm.deepdive.cards.model;

public class Card implements Comparable<Card> {

  private final Rank rank;
  private final Suit suit;


  public Card(Rank rank, Suit suit) {
    this.rank = rank;
    this.suit = suit;
  }

  public Rank getRank() {
    return rank;
  }

  public Suit getSuit() {
    return suit;
  }

  @Override
  public int compareTo(Card other) {
    int comparison = suit.compareTo(other.suit);
    if (comparison == 0) {
      comparison = rank.compareTo(other.rank);
    }
    return comparison;
  }

  @Override
  public String toString() {
    return rank.getSymbol() + suit.getSymbol();
  }
}
```

This example can be found on github. The class structure in java is fairly simple once you learn it. The only thing that is required is the class keyword and a valid class name. Almost anything you could imagine is a valid class name but you should stick to the uppercase noun convention. While those are the only things required in class declaration it has a lot more going on. First the implements keyword followed by an interface. This gives us the ability to override the compareTo() method. We also override the toString() method though that is not apart of the Comparable<T> interface. Finally we have a couple final variables called rank and suit which are set in our overloaded constructor.

This example uses the card class to create a deck of cards. Each have a suit and rank, which is set automatically by the card constructor. Each unique card is added to the cards Array-List.

```java
public class Deck {

  private final List<Card> cards;
  private final List<Card> dealt;

   public Deck() {
     dealt = new LinkedList<>();
     cards = new ArrayList<>();
     for (Suit s : Suit.values()) {
       for (Rank r : Rank.values()) {
         cards.add(new Card(r,s));
       }
     }
   }
```

1.2 C++:

```
10  class Game
11  {
12          private:
13                  int bet{ 0 };
14                  double money{ 1000.0 };
15                  int wins, losses, ties, numberOfBets;
16                  Deck deck;
17                  Hand playersHand;
18                  Hand dealersHand;
19                  Logger log;
20
21          public:
22                  Game();
23                  bool SetBet(int b);
24                  void InitialDeal();
25                  string ShowPlayerHand();
26                  string ShowDealersHand(bool hide);
27                  bool IsBlackJack();
28                  bool PlayerBusted();
29                  bool PlayerContinues();
30                  void PlayerHits();
31                  string PlayerWins();
32                  bool DealerContinues();
33                  string DealerWins();
34                  string Tie();
35                  string NoResults();
36                  string ShowResults();
37                  void ClearHands();
38                  bool  IsLogOpened() { return log.IsLogOpen(); }
39                  void EndGame();
40
41  };
```

In C++ the way classes are created are actually done in the header files. This is a game class for a card game called craps. As you can see the class structure here in c++ is far different than that of java. For example in java you need to set each member variable as public or private but it c++ you simply have a block of private variables/methods and a block of public variables/methods. In the actual c++ files Game.cpp we find that there is no class, but methods that define they are apart of the game class as shown below.

```
96   void Game::PlayerHits()
97   {
98       Card card;
99       deck.Deal(card);
100      playersHand.AddCard(card);
101  }
```

```
373          private: System::Void btnHit_Click(System::Object^ sender, System::EventArgs^ e) {
374                  if (myGame.PlayerContinues()) {
375                          myGame.PlayerHits();
376                          txtPlayerHand->Text = gcnew String(myGame.ShowPlayerHand().c_str());
377                          if (myGame.PlayerBusted()) {
378                                  txtDealerHand->Text = gcnew String(myGame.ShowDealersHand(false).c_str());
379                                  txtStatus->Text = gcnew String(myGame.ShowResults().c_str());
380                                  gbAction->Visible = false;
381                          }
382                  }
383          }
```

In the code above we actually see our game class in action as we have a button click event the triggers some methods like ‘PlayerContinues()’ and ‘PlayerHits()’ which continues the game, and alters the state of the actual game object.

1.3 C#:

```
9       class LogoOrderItem
10      {
11          private bool hasLogo;
12          private string itemType;
13          private int numColors;
14          private int numItems;
15          private string text;
16          private double cost;
17
18          public bool HasLogo { get { return hasLogo; } set { hasLogo = value; Calc(); } }
19
20          public string ItemType { get { return itemType; } set { itemType = value; Calc(); } }
21
22          public int NumColors { get { return numColors; } set { numColors = value; Calc(); } }
23
24          public int NumItems { get { return numItems; } set { numItems = value; Calc(); } }
25
26          public string Text { get { return text; } set { text = value; Calc(); } }
27
28          public double Cost { get { return cost;  } set { cost = value; } }
29
30
31          public LogoOrderItem(bool hasLogo, string itemType, int numColors, int numItems, string text)
32          {
33              HasLogo = hasLogo;
34              ItemType = itemType;
35              NumColors = numColors;
36              NumItems = numItems;
37              Text = text;
38          }
39
40          public LogoOrderItem(string text, bool hasLogo): this(hasLogo, "mug", 0, 0, text)
41          {
42
43          }
44          public LogoOrderItem() : this(false, "mug", 0, 0, "")
45          {
46
47          }
48
```

In C# the actual class declaration is similar to java in that the only thing required is the class keyword and the valid Class name. The big difference in C# is that you can have these things called properties that eliminate the need to create setter and getter methods for each private variable for proper encapsulation. C# also gives us the ability to do what is called constructor chaining. While you can do this in java I did not have a good example. Constructor chaining gives us the ability to reduce code redundancy and set default values fairly easily.

In the code we have a button click event that creates a LogoOrderItem grabs all of the relevant data and calculates the final price. It then calls its GetOrderSummary method to display the info to the user.

```
35          private void btnSubmit_Click(object sender, EventArgs e)
36          {
37              LogoOrderItem loi = new LogoOrderItem(txtPrintedText.Text, cboxLogo.Checked);
38              loi.NumItems = Int32.Parse(txtNumItems.Text);
39              if(rbtnMug.Checked)
40              {
41                  loi.ItemType = "Mug";
42              }
43              else if (rbtnPen.Checked)
44              {
45                  loi.ItemType = "Pen";
46              }
47              else
48              {
49                  loi.ItemType = "USB";
50              }
51              if(cboxLogo.Checked)
52              {
53                  loi.NumColors = Int32.Parse(txtNumColors.Text);
54              }
55
56              txtSummary.Text = loi.GetOrderSummary();
57
58          }
```

2.0 Inheritance and Polymorphism

In programming inheritance is the idea that a base class can pass on properties and behaviors to another derived class. In this example I have programmed Conway's game of life. LifeCell is the base cell type for all other cell types. We create this base class as there are a set number of methods and variables that we want all Cell types to share, such as initializeBoard() or setNextState(). These methods do not need to be unique as they will be the same for every derived class. Although GetLivingNeighbors and UpdateCells are unique to each cell type because they each differ in behavior. For example the DiagonalCell type looks for neighbors that are diagonal of that cell, hence the name. Although the default ConwayCell looks for neighbors right next to itself and above/below itself. Thus these are marked virtual and up the children to implement.

```
8   class LifeCell
9   {
10  protected:
11          static const int ROWS{ 45 };
12          static const int COLS{ 78 };
13          int count{ 0 }, rows{ 0 }, cols{ 0 };   //count is the number of neighbors alive counted
14          int pattern;            //the index of the initial pattern chosen.
15          bool bOpen{ false };            //states whether the file was opened successfully
16
17          char cell[ROWS][COLS];  //Grid of cells.  If '*' -> cell is alive, if '.' -> cell is dead
18          char nextCellState[ROWS][COLS];
19
20          void InitializeBoard();         //reads the pattern file and initializes the board
21          virtual void GetLivingNeighbors(int r, int c) = 0;
22          virtual void UpdateCells() = 0;         //apply the rules here
23          void SetNextState();            //set new states into the cells
24
25  public:
26          LifeCell();             //default constructor.  All cells are dead initially.
27          void SetPattern(int pat);       //Sets the initial pattern and calls InitializeBoard
28          void UpdateBoard();     //Calculates the next generation, sets the new values into the grid
29          string PrintBoard();    //returns a string of the board for display
30          void Clear();           //Reset the board to all dead cells
31  };
```

The ConwayCell class, as seen below, inherits from the LifeCell class. As you can see this only implements the virtual classes from LifeCell, all other methods are defined by the LifeCell class. The virtual methods are the only ones that need to be unique for each cell type.

```
6   #include "LifeCell.h"
7
8   class ConwayCell : public LifeCell
9   {
10  protected :
11          virtual void GetLivingNeighbors(int x, int y);
12          virtual void UpdateCells();
13
14  private:
15          int neighbors[9][2] = { {-1, -1}, {-1, 0}, {-1, +1}, {0, -1}, {0, 0}, {0, +1}, {+1, -1}, {+1, 0}, {+1, +1} };
16
17  public:
18          ConwayCell();
19  };
```

In the code to the right we see the creation of an array of LifeCell pointers. We then add 5 unique Cell Type objects to the array. This works because all of the Cell types seen below are children of the base class LifeCell.

```
29          //create polymorphic array
30          LifeCell* pLife[5];
31          ConwayCell      con;
32          FredkinCell fred;
33          ModifiedFredkin modfred;
34          SeedsCell seeds;
35          DiagonalCell diagonal;
36
37          pLife[0] = &con;
38          pLife[1] = &fred;
39          pLife[2] = &modfred;
40          pLife[3] = &seeds;
41          pLife[4] = &diagonal;
```

In the code below we see that our array of LifeCell pointers is being used to update our UI which we refer to as UpdateBoard(). How does this happen if each pointer has a unique type with a unique behavior?

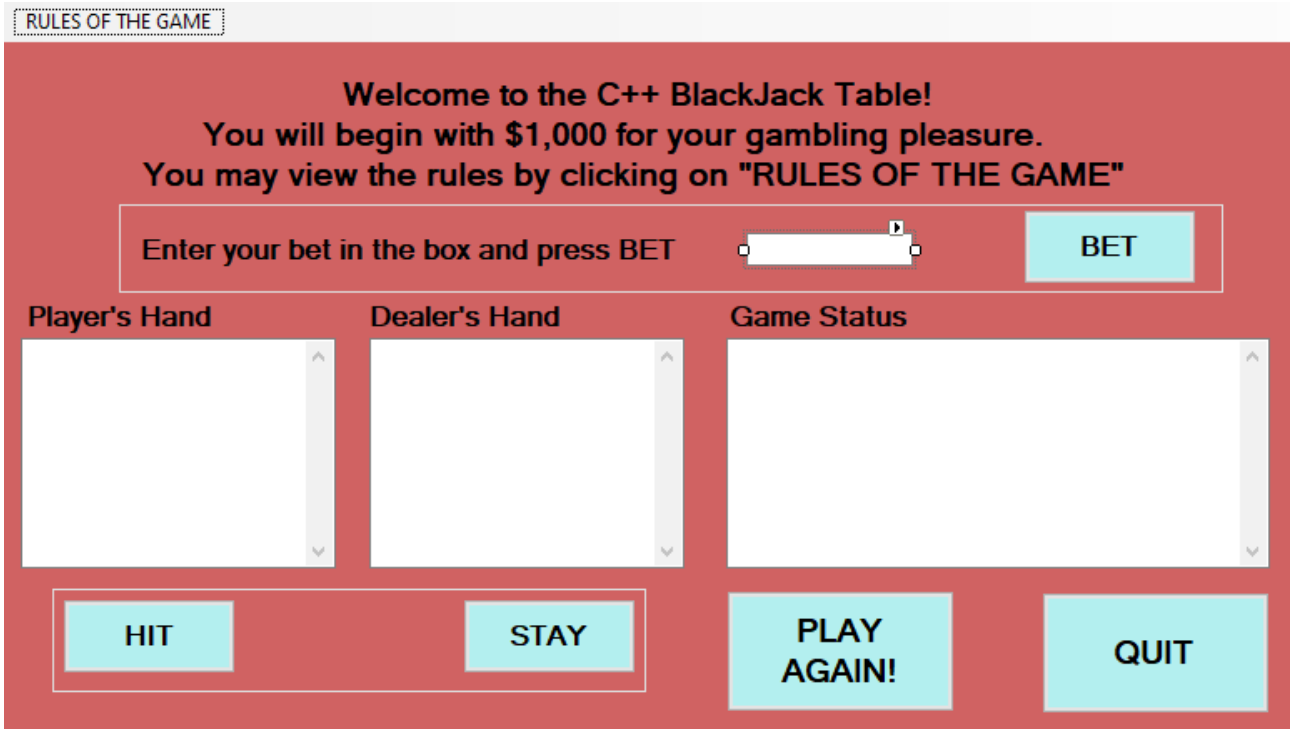
```
65                  do
66                  {
67                          pLife[config]->UpdateBoard();
68                          cout << pLife[config]->PrintBoard() << flush;
69                          Sleep(200);
70                          system("cls");
71                          lifeCount++;
72                  } while (lifeCount < 50);
73                  pLife[config]->Clear();
74                  ModifiedFredkin* mfc = dynamic_cast<ModifiedFredkin*>(pLife[config]);
75                  if (mfc != nullptr)
76                  {
77                          mfc->ClearAge();
78                  }
79                  lifeCount = 0;
80                  cout << "\n\nDo you want to run another simulation?(y or n) ";
81                  cin >> answer;
```

While UpdateBoard is a base class method only it calls UpdateCells() which is unique to each type. This gives us the ability to quickly switch between types and utilize unique behavior in each.

```
85  void LifeCell::UpdateBoard()
86  {
87      UpdateCells();
88      SetNextState();
89  }
```

3.0 Graphical User Interface Construction

While at CNM I created both windows form apps in both C# and C++. I also created some java GUI Applications using JavaFx. Although I believe in general C# and C++ are better when developing Desktop applications. The only down side is neither are very easy to port over to Mac or Linux. Below you see a group project I worked on in my first year that lets you play Blackjack.



Creating these forms is fairly simple with Visual Studios design mode, though there is also the option to write these forms using code. Although this difficult to visualize without actually looking at the form, so the designer mode is easiest. If you name all of you elements appropriately writing code to utilizing this GUI is fairly simple. The only difficult part is converting the type returned by the element to a useful type like Integer or String.

As you can see below this is the on click method written for the ‘bet button’ name ‘btnBet.’ To write code to use this UI we use the elements name and the element propertie we want. For example ‘txtBet→Text’ will return the text property associated with the txtBet element.

```
344     private: System::Void btnBet_Click(System::Object^ sender, System::EventArgs^ e) {
345         int bet{ 0 };
346         Boolean isValidBet = false;
347 
348         bet = Convert::ToInt32(txtBet->Text);
349         isValidBet = myGame.SetBet(bet);
350         if (isValidBet)
351         {
352 
353             myGame.InitialDeal();
354             txtPlayerHand->Text = gcnew String(myGame.ShowPlayerHand().c_str());
355             txtDealerHand->Text = gcnew String(myGame.ShowDealersHand(true).c_str());
356 
357             if (myGame.IsBlackJack()) {
358                 stringstream ss;
359                 ss << "BlackJack\r\n" << myGame.PlayerWins();
360                 txtStatus->Text = gcnew String(ss.str().c_str());
361             }
362             else {
363                 gbBet->Visible = false;
364                 gbAction->Visible = true;
365             }
366         }
367         else {
368             txtStatus->Text = gcnew String("Invalid Bet, try again.");
369         }
370     }
```

This program also keeps track of all of the wins and losses via a file it writes to every time the player wins, losses or ties with the dealer. It also performs a close action to summarize the games played while the logger was open.

```
103  string Game::PlayerWins()
104  {
105      money += (bet * 1.5);
106      stringstream ss;
107      ss << playersHand.Show(false, false) << "\r\n" << dealersHand.Show(true, false) << "\r\n"
108          << "Player wins: " << bet * 1.5 << "\r\n" << " Player remaining balance: $" << money << "\r\n \r\n";
109      log.WriteLog(ss.str());
110      wins++;
111      return ss.str();
112  }
113
114
115  string Game::DealerWins()
116  {
117      stringstream ss;
118      money = money - bet;
119      if (money < 0) money = 0;
120      ss << playersHand.Show(false, false) << "\r\n" << dealersHand.Show(true, false) << "\r\n"
121          << "Player losses: $" << bet << "\r\n" << " Player remaining balance: $"<< money << "\r\n \r\n";
122      log.WriteLog(ss.str());
123      losses++;
124      return ss.str();
125  }
126
127  string Game::Tie()
128  {
129      stringstream ss;
130      ss << playersHand.Show(false, false) << "\r\n" << dealersHand.Show(true, false) << "\r\n"
131          << "Tie, bet: " << bet << "\r\n" << " Player remaining balance: $" << money << "\r\n \r\n";
132      log.WriteLog(ss.str());
133      ties++;
134      return ss.str();
135  }
```

These forms also give you the ability to open dialogs to get more information. For example in sprocket order form I made for C# 1 I have the main window open a separate window to enter detailed info about the sprocket that needs to be ordered as seen below.

```
56          private void btnAdd_Click(object sender, RoutedEventArgs e)
57          {
58              SprocketForm sf = new SprocketForm();
59              Sprocket sp;
60
61              sf.ShowDialog();
62              if (sf.DialogResult == true)
63              {
64                  sp = sf.GetSprocket();
65                  MessageBox.Show("Student " + sp.GetType() + " Added!");
66                  if (sp is SteelSprocket)
67                  {
68                      sp = (SteelSprocket)sp;
69                  }
70                  else if (sp is AluminumSprocket)
71                  {
72                      sp = (AluminumSprocket)sp;
73                  }
74                  else if (sp is PlasticSprocket)
75                  {
76                      sp = (PlasticSprocket)sp;
77                  }
78                  Address address = new Address();
79                  if (!(bool)cboxLocalPickup.IsChecked)
80                  {
81                      address.City = txtCity.Text;
82                      address.State = txtState.Text;
83                      address.Street = txtStreet.Text;
84                      address.Zipcode = txtZipCode.Text;
85                  }
86                  else
87                  {
88                      address = null;
89                  }
90                  lboxItems.Items.Add(sp);
91                  spo.address = address;
92                  spo.CustomerName = txtCustomerName.Text;
93                  spo.items.Add(sp);
94
95              }
```

All you have to do to work with data between two forms is to create an instance of the dialog in the main form and use it to gather the data needed. Here you can see we create the Sprocket Form and show it to the user. Once the user confirms his submission in the form its pushed back to the main form and we can add the data we just got from the Sprocket form to the Sprocket order.

4.0 Database Manipulation:

Connecting to a database was one of the more challenging things but I was able to accomplish this with both java spring and c#. This example is written in C#. To create a database in a Blazor application you have to add a new class library by doing the command ‘dotnet new classlib -o P5DeprecationCalc.Data’ you also need to add this to your sln with the command ‘dotnet sln add P5DeprecationCalc.Data.’ Finally you need to add the EntityFrameworkCore.Tools package and the EntityFrameworkCore.Sqlite package to your project. After you complete those steps you simply start creating your data classes as seen below. This class holds a couple different values including asset name, Initial value and Date added and Date removed variables. This is kind of like a blue print as this will be generated into a SQL table.

```
 9  namespace P5DepricationCalc.Data.Models
10  {
11      public class MyAsset : IMyAsset
12      {
13          public int Id { get; set; }
14          [Required]
15          public string Name { get; set; }
16          [Required]
17          public double InitValue { get; set; }
18          [Required]
19          public double SalvageValue { get; set; }
20          [Required]
21          public int UsefullLife { get; set; }
22          [Required]
23          public DateTime DateIn { get; set; }
24          [Required]
25          public DateTime DateOut { get; set; }
26          public bool GraphVis { get; set; }
27          public double YearlyDeprecation { get; set; }
```

We also need to create a database context class which is how we will actually interact with the database. This file also contains a second class that is used to configure our database when running migrations. Each time we change something about the data classes we must run a migration to update the tables in our database. To actually run our migrations we simply run the command ‘dotnet-ef migrations add migrationName.’ This example also shows us the location of the database. Its located in the root folder of the project and it is called Asset.db.

```
7       public class AssetDbContext : DbContext
8       {
9
10          public AssetDbContext(DbContextOptions<AssetDbContext> context) : base(context)
11          {
12
13          }
14
15          public DbSet<MyAsset> MyAssets { get; set; }
16      }
17
18      public class AssetDbContextFactory : IDesignTimeDbContextFactory<AssetDbContext>
19      {
20          public AssetDbContext CreateDbContext(string[] args)
21          {
22              var optionsBuilder = new DbContextOptionsBuilder<AssetDbContext>();
23              optionsBuilder.UseSqlite("Data Source = ../Asset.db");
24
25              return new AssetDbContext(optionsBuilder.Options);
26          }
27      }
```

While we have created a database we have no way of accessing it until we create a database API. The best way to do this in Blazor is to create an interface which will outline our Get, Save, and Delete operations. Then write the actual methods in a API class called MyAssetApiServerSide which uses our interface MyAssetApi.

There are a couple of different operations that are setup in my Api. One of them being GetAssetAsync by Id, which will match the id passed to a unique id in the database. The other two operations use a MyAsset object to get the id and match them to an item in the database. All of this is done using our IDbContextFactory to create a database context and using that context to make changes to the database.

```
13      public class MyAssetApiServerSide : MyAssetApi
14      {
15          IDbContextFactory<AssetDbContext> factory;
16
17          public MyAssetApiServerSide (IDbContextFactory<AssetDbContext> factory)
18          {
19              this.factory = factory;
20          }
21
22          public async Task<MyAsset> GetAssetAsync(int id)
23          {
24              using var context = factory.CreateDbContext();
25              return await context.MyAssets.FirstOrDefaultAsync(p => p.Id == id);
26          }
27
28          public async Task DeleteAssetAsync(MyAsset item)
29          {
30              using var context = factory.CreateDbContext();
31              context.Remove(item);
32              await context.SaveChangesAsync();
33          }
34
35          public async Task<MyAsset> SaveAssetAsync(MyAsset item)
36          {
37              using var context = factory.CreateDbContext();
38              if(item.Id == 0)
39              {
40                  context.Add(item);
41              }
42              else
43              {
44                  var asset = item as MyAsset;
45                  var currentAsset = await context.MyAssets.FirstOrDefaultAsync(p => p.Id == asset.Id);
46                  currentAsset.InitValue = asset.InitValue;
47                  currentAsset.SalvageValue = asset.SalvageValue;
48                  currentAsset.UsefullLife = asset.UsefullLife;
49                  currentAsset.GraphVis = asset.GraphVis;
50                  currentAsset.DateOut = asset.DateOut;
51                  currentAsset.DateIn = asset.DateIn;
52                  await context.SaveChangesAsync();
53              }
54              await context.SaveChangesAsync();
55              return item;
56          }
```

Finally for the database to work you need to include the following packages.

```
13  using Microsoft.EntityFrameworkCore;
14  using P5DepricationCalc.Data.Interfaces;
15  using P5DepricationCalc.Data.Models;
```

You also need to add the following lines to the ConfigureServices method in Startup.cs to add the database context factory as a service, as well as your Api.

```
30          public void ConfigureServices(IServiceCollection services)
31          {
32              services.AddDbContextFactory<AssetDbContext>(opt => opt.UseSqlite($"Data Source=../Asset.db"));
33
34              services.AddScoped<MyAssetApi, MyAssetApiServerSide>();
```

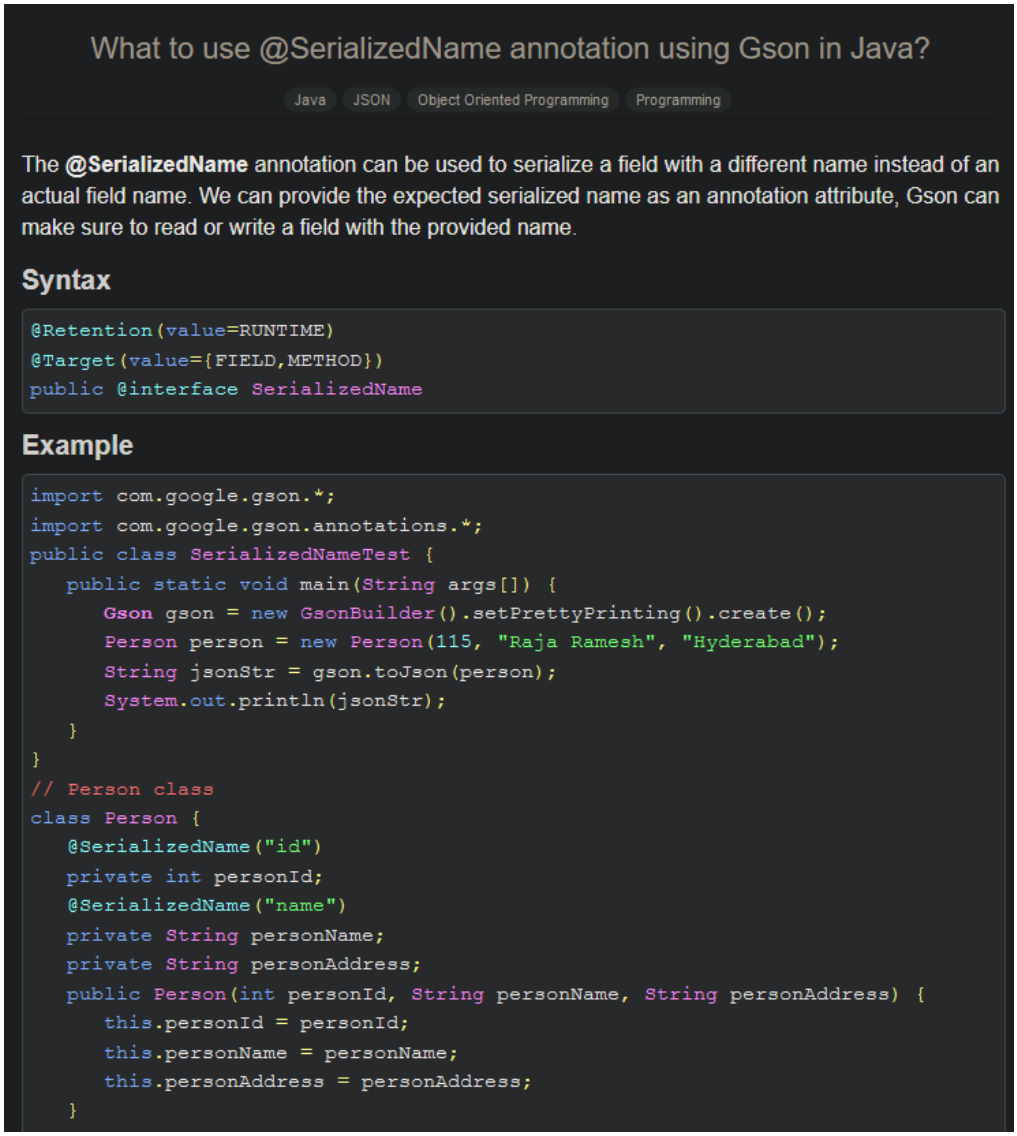
With that setup you can now interact and save items to the database as shown in the code below. This code uses the existing MyAsset data class to build an object to be saved to the database. Now all you have to do is use the api variable, injected as a service with the line '@inject MyAssetApi api', and call SaveAssetAsync.

```
145     public void Save(MyAsset asset)
146     {
147         asset.Name = name;
148         asset.InitValue = initialValue;
149         asset.SalvageValue = salvageValue;
150         asset.UsefullLife = usefulLife;
151         asset.DateIn = dateIn;
152         asset.DateOut = dateOut;
153         asset.GraphVis = false;
154
155         api.SaveAssetAsync(asset);
156         asset.YearlyDeprecation = depCalc.CalcYearlyDepreciation(asset);
157
158         NavigationManager.NavigateTo("/deprication");
159     }
```

5.0 Web Research:

5.1 Ski America App:

One of the many skills acquired through my journey at CNM includes the ability to research existing code and implementing those solutions in my own code. One of the first times I found how useful the internet can be was when I began building an Android app that utilizes Weather data from ski resorts to give skiers a forecast on the mountain. This app uses googles gson which can automatically serialize java objects to json and serialize json to java objects.



What to use @SerializedName annotation using Gson in Java?

Java JSON Object Oriented Programming Programming

The **@SerializedName** annotation can be used to serialize a field with a different name instead of an actual field name. We can provide the expected serialized name as an annotation attribute, Gson can make sure to read or write a field with the provided name.

Syntax

```
@Retention(value=RUNTIME)
@Target(value={FIELD,METHOD})
public @interface SerializedName
```

Example

```
import com.google.gson.*;
import com.google.gson.annotations.*;
public class SerializedNameTest {
   public static void main(String args[]) {
      Gson gson = new GsonBuilder().setPrettyPrinting().create();
      Person person = new Person(115, "Raja Ramesh", "Hyderabad");
      String jsonStr = gson.toJson(person);
      System.out.println(jsonStr);
   }
}
// Person class
class Person {
   @SerializedName("id")
   private int personId;
   @SerializedName("name")
   private String personName;
   private String personAddress;
   public Person(int personId, String personName, String personAddress) {
      this.personId = personId;
      this.personName = personName;
      this.personAddress = personAddress;
   }
```

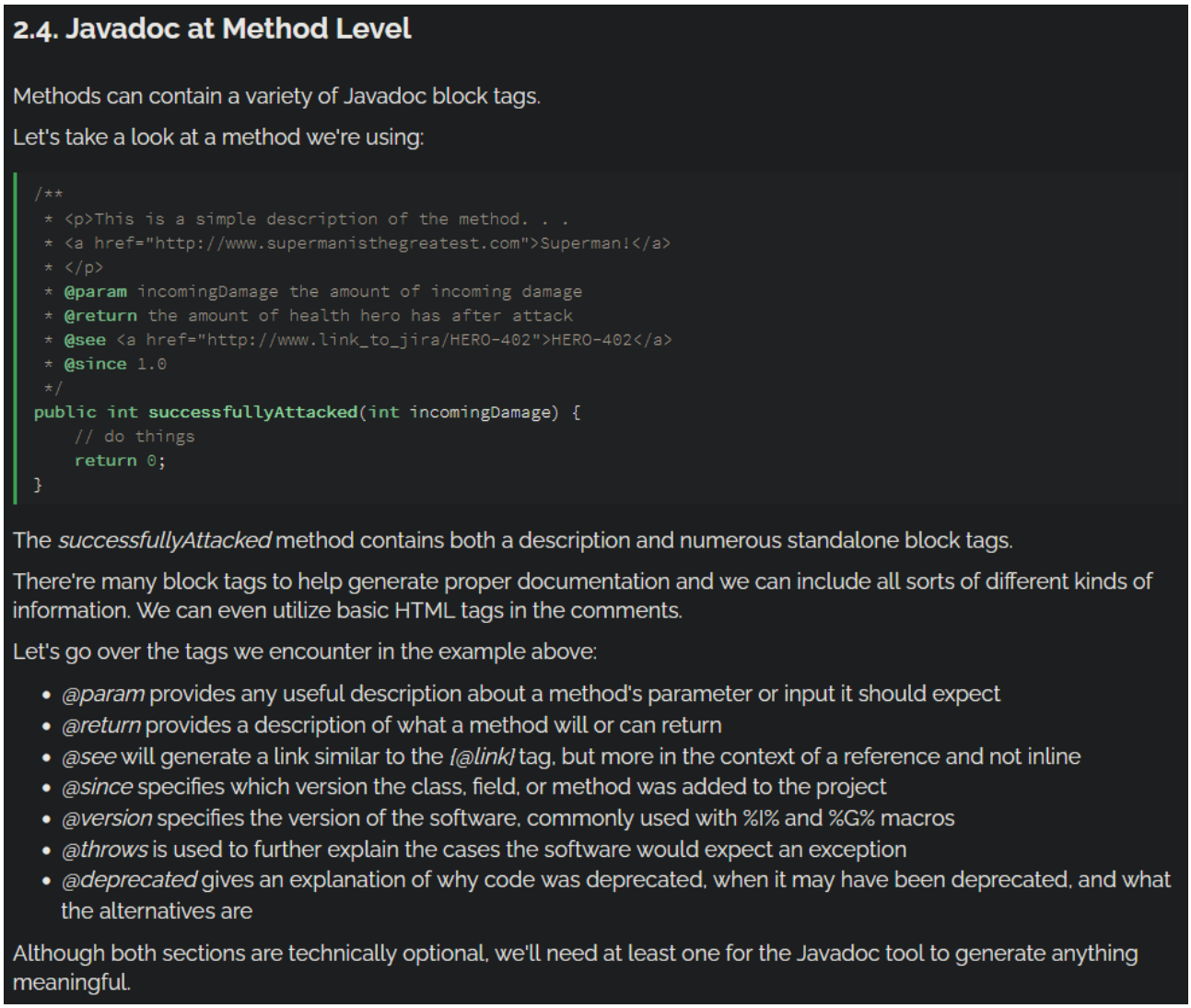
Link

I used my research to automatically serialize json data coming from my weather forecast service to java objects to be displayed to the user. This is useful to ensure your object is being serialized correctly, its also helpful when a variable name does not match your own naming conventions like “totalSnowfall_cm” being encoded to totalSnowFall.

```
18    public static class Data {
19
20      @SerializedName("weather")
21      private List<Weather> weather;
22
23      public List<Weather> getWeather() {
24        return weather;
25      }
26
27      public void setWeather(
28          List<Weather> weather) {
29        this.weather = weather;
30      }
31
32      public static class Weather {
33
34        @SerializedName("bottom")
35        private List<Bottom> bottom;
36
37        @SerializedName("totalSnowfall_cm")
38        private String totalSnowFall;
39
40        @SerializedName("chanceofsnow")
41        private String chanceOfSnow;
42
43        public String getChanceOfSnow() {
44          return chanceOfSnow;
45        }
46
```

5.2 Stock Rollers Android Client:

Documentation is important in any project and can be key to remembering what your own code does. It will also help other programmers work on your code as they know exactly what is supposed to be happening. Javadoc is a widely used application that will turn all of your code into clear documentation by simply including what are know as javadoc comments. These comments include specific syntax that helps auto generate documentation.



2.4. Javadoc at Method Level

Methods can contain a variety of Javadoc block tags.

Let's take a look at a method we're using:

```
/**
 * <p>This is a simple description of the method. . .
 * <a href="http://www.supermanisthegreatest.com">Superman!</a>
 * </p>
 * @param incomingDamage the amount of incoming damage
 * @return the amount of health hero has after attack
 * @see <a href="http://www.link_to_jira/HERO-402">HERO-402</a>
 * @since 1.0
 */
public int successfullyAttacked(int incomingDamage) {
    // do things
    return 0;
}
```

The *successfullyAttacked* method contains both a description and numerous standalone block tags.

There're many block tags to help generate proper documentation and we can include all sorts of different kinds of information. We can even utilize basic HTML tags in the comments.

Let's go over the tags we encounter in the example above:

- *@param* provides any useful description about a method's parameter or input it should expect
- *@return* provides a description of what a method will or can return
- *@see* will generate a link similar to the *{@link}* tag, but more in the context of a reference and not inline
- *@since* specifies which version the class, field, or method was added to the project
- *@version* specifies the version of the software, commonly used with %I% and %G% macros
- *@throws* is used to further explain the cases the software would expect an exception
- *@deprecated* gives an explanation of why code was deprecated, when it may have been deprecated, and what the alternatives are

Although both sections are technically optional, we'll need at least one for the Javadoc tool to generate anything meaningful.

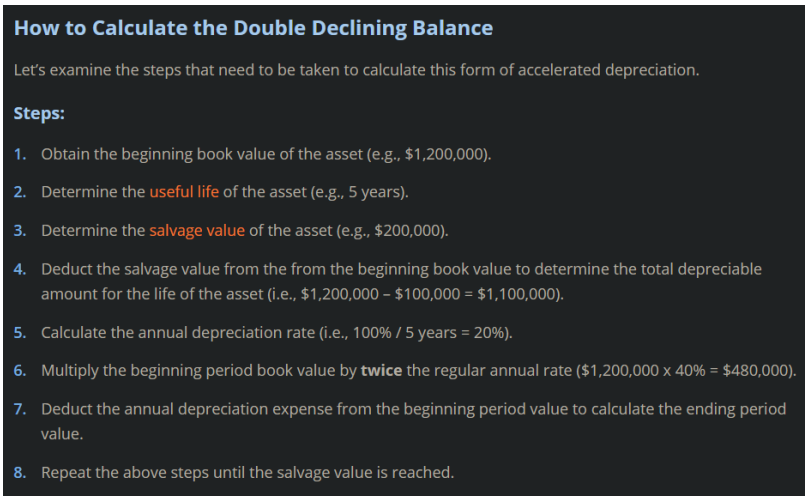
Link

I use these javadoc comments in my own code as shown below. You can include @param and describe what the variable is supposed to be. In this case it shows the type Stock and the name ticker, indicating that you need to pass a fully qualified Stock object to this method. These comments also give you the ability to describe what they are supposed to be doing. Useful for not only yourself but other programmers as well.

```
23  /**
24   * Class to query our Server Application with HTTP
25   */
26  public interface StockrollersService {
27
28    /**
29     * Uses a Single Instance
30     * @return single instance of StockrollersService
31     */
32    static StockrollersService getInstance() {
33      return InstanceHolder.INSTANCE;
34    }
35
36    /**
37     * Querys the server for a stock object
38     * @param token Authorization header
39     * @param symbol Stock ticker
40     * @return Single Stock object
41     */
42    @GET("stocks/{symbol}")
43    Single<Stock> getStock(@Header("Authorization") String token, @Path("symbol") String symbol);
44
45    /**
46     * Gets A list of History objects for the specified ticker
47     * @param token Authorization header
48     * @param symbol Stock ticker
49     * @return Flowable List of History Objects
50     */
51    @GET("history/{symbol}")
52    Flowable<ArrayList<HistoryResponse>> getHistoryForStock(@Header("Authorization") String token, @Path("symbol") String symbol);
53
54    /**
55     * Querys the server for a random Stock
56     * @param token Authorization Header
57     * @return Single Stock
58     */
59    @GET("stocks/random")
60    Single<Stock> getRandom(@Header("Authorization") String token);
```

5.3 Depreciation Calculator

While a lot of the time my research is about programming, occasionally I am asked to perform a calculation that I am not familiar with. Internet research is a perfect approach to this problem. For example in Program 4 of my C# 1 class I was asked to make a program that calculates both straight line deprecation and double declining deprecation. So I did some research on how to calculate both of those things.



How to Calculate the Double Declining Balance

Let's examine the steps that need to be taken to calculate this form of accelerated depreciation.

Steps:

1. Obtain the beginning book value of the asset (e.g., $1,200,000).
2. Determine the useful life of the asset (e.g., 5 years).
3. Determine the salvage value of the asset (e.g., $200,000).
4. Deduct the salvage value from the from the beginning book value to determine the total depreciable amount for the life of the asset (i.e., $1,200,000 – $100,000 = $1,100,000).
5. Calculate the annual depreciation rate (i.e., 100% / 5 years = 20%).
6. Multiply the beginning period book value by **twice** the regular annual rate ($1,200,000 x 40% = $480,000).
7. Deduct the annual depreciation expense from the beginning period value to calculate the ending period value.
8. Repeat the above steps until the salvage value is reached.

Link

I used this explanation to write the correct code to calculate the double declining depreciation.

```
17          protected override void Calc()
18          {
19              double annualDepExpense;
20              double depRate;
21
22              annualDepExpense = (StartValue - SalvageValue) / LifeTime;
23
24              depRate = annualDepExpense / (StartValue - SalvageValue);
25
26              EndValue = StartValue;
27
28              for(int i = 0; i < (DateRemovedFromInventory.Year - DataAddedToInventory.Year); i++ )
29              {
30                  EndValue -= depRate * EndValue;
31              }
32          }
```