Sanjeev Penupala
SXP170022
Section: 501

1)

Static Scoping

```java
import static java.lang.System.*;

// Static Scope Example
class Main {
  static int x = 420;

  public static void main(String[] args) {
    out.println(second());
  }

  static int first() {
    return x;
  }

  static int second() {
    int x = 69;
    return first();
  }
}
```

Output: 420

Explanation:

When x = 420, that is a globally declared variable, so it can be accessed throughout the rest of the program. So, calling second() first, will not print out 69, as 69 is within the scope of the second() function, not in a global scope context.

Dynamic Scoping

```bash
#!/bin/bash
# Dynamic Scope Example

x=420

function first()
{
  echo $x
}

function second()
{
  x=69
  first
}

second
```

Output: 69

Explanation:

Unlike static scoping, dynamic scope works off the call-stack environment or what the current environment is. Even though x = 420 was defined globally, when the second() function is called, the most recent definition of x on the call stack is 69, so when first() is called, it will print 69

Three programing languages that have static scoping are C, C++, and Java. Three programing languages that support dynamic scoping is bash, LaTeX, and Perl. The main gain from getting static scoping is readability. With static scoping, a variable is always referred by its top-level environment. When reading the program, it is easy to tell how a program is run, to due it being unrelated to the call stack. Static scoping makes your code modular, so that any programmer can read the program and understand what the output is. The main thing that was lost in support of static scoping is the ability to utilize recent runtime environment variables on the call stack. Any recent declarations are irrelevant now.

Sanjeev Penupala
SXP170022
Section: 501

2)

```
from math import sqrt


# Global Variables
x, y = 2, 3

def func1(z1):
  def func2(z2):
    def func3(z3):
      def func4(z4):
        def func5(z5):
          distance = sqrt(sum(point ** 2 for point in [x, y, z5])) # Local Variable
          print(f"Distance at subprogram {z5}: {distance}")

        distance = sqrt(sum(point ** 2 for point in [x, y, z4])) # Local Variable
        print(f"Distance at subprogram {z4}: {distance}")
        func5(z4 + 1) # Calling Nested Programs Variables

      distance = sqrt(sum(point ** 2 for point in [x, y, z3])) # Local Variable
      print(f"Distance at subprogram {z3}: {distance}")
      func4(z3 + 1) # Calling Nested Programs Variables

    distance = sqrt(sum(point ** 2 for point in [x, y, z2])) # Local Variable
    print(f"Distance at subprogram {z2}: {distance}")
    func3(z2 + 1) # Calling Nested Programs Variables

  distance = sqrt(sum(point ** 2 for point in [x, y, z1])) # Local Variable
  print(f"Distance at subprogram {z1}: {distance}")
  func2(z1 + 1) # Calling Nested Programs Variables

print('Results of subprograms:\n')
func1(1)
```

In this nested subprogram program, it utilizes global variables, local variables, and nested variables to calculate the end program answer. The goal of this program is to calculate the distance formula of variables of x, y, z at each subprogram level. The only variable that is changing is z, as we are incrementing it by one, each time it goes one subprogram deep. The global variables that are constant are x and y, the local variable that is calculated at each subprogram level is distance, and calling the next subprogram essentially calls the nested variables, that will repeat this process until subprogram 5 is reached. The output of the program should output the distance of x, y, z from the center at different z values.

Sanjeev Penupala
SXP170022
Section: 501

3)

```c
#include <stdio.h>

// Narrowing Conversion
void narrowTest()
{
  double a = 398473847387438;
  float b = a;
  if(a != b) {
    printf("Narrowing conversion occured from double %f to float %f\n", a, b);
  }

  float x = 420.69;
  int y = x;
  if(x != y) {
    printf("Narrowing conversion occured from float %f to int %d\n", x, y);
  }

  int p = 56;
  char q = p;
  if(x != y) {
    printf("Narrowing conversion occured from int %d to char %c\n", p, q);
  }
}

// Widenining Conversion
void wideTest()
{
  short a = 20;
  int b = a;
  if(sizeof(b) > sizeof(a)){
    printf("Widening conversion occured from short %d to int %d\n", a, b);
  }

  float x = 39;
  double y = x;
  if(sizeof(y) > sizeof(x)){
    printf("Widening conversion occured from float %f to double %f\n", x, y);
  }

  char c = 'A';
  char s[2] = {c , '\0'};
  if(sizeof(s) > sizeof(c)){
```

Sanjeev Penupala
SXP170022
Section: 501

```
    printf("Widening conversion occured from char %c to string %s\n", c, s);
  }
}

// Explicit Conversion
void explicitTest()
{
  // If any error is thrown from this function, then explicit conversion was not succ
essful
  int a = 11;
  float b = (float) a;
  printf("Explicit conversion occured from int %d to float %f\n", a, b);

  float x = 69.420;
  int y = (int) x;
   printf("Explicit conversion occured from float %f to int %d\n", x, y);

  int p = 65;
  char q = (char) p;
   printf("Explicit conversion occured from int %d to char %c\n", p, q);
}

// Implicit Conversion
void implicitTest()
{
  // If any error is thrown from this function, then implicit conversion was not succ
essful
  int a = 42;
  float b = 24;
  b = 24 * 42;
  printf("Implicit conversion occured for int %d times float %f to float\n", a, b);


  float x = 420.69;
  int y = x;
  printf("Implicit conversion occured from float %f to int %d\n", x, y);
}

int main(void) {
  narrowTest();
  wideTest();
  explicitTest();
  implicitTest();
}
```

Sanjeev Penupala
SXP170022
Section: 501

I wrote 4 functions that tested examples of narrowing conversion, widening conversion, explicit conversion, and implicit conversion. In each function, there are at least 2 examples to test whether the conversion works or not and mainly to test to see if it exists! In narrowing conversion, I had 3 different tests, testing 2 datatypes against each other and seeing if I converted one data type to another, if the same value is still held. If the value is not the same, then clearly a narrowing conversion has happened, because data was lost. For widening conversion, in order to tell if this conversion happened (as value will still be same), checking the size of the variable after conversion, will tell us if widening conversion has happened, which is exactly what I tested. For explicit and implicit conversion, it was straight forward which data types to use for these and how I casted variables. If there is anything wrong with the conversions, it will throw an error on runtime, which it did not, telling us these conversions were handling perfectly.

Sanjeev Penupala
SXP170022
Section: 501

4)

USING ENUMS

```
#include <iostream>
#include <string>
#include <algorithm>  // for std::find
#include <iterator>   // for std::begin, std::end
#include <cmath>      // for std::ceil

using namespace std;

/* ENUMS */
enum card {
  ACE=1,
  TWO=2,
  THREE=3,
  FOUR=4,
  FIVE=5,
  SIX=6,
  SEVEN=7,
  EIGHT=8,
  NINE=9,
  TEN=10,
  JACK=11,
  QUEEN=12,
  KING=13
};

enum suit {CLUBS=1, SPADES=2, HEARTS=3, DIAMONDS=4};

/* POKER FUNCTIONS */
bool heart_royal_flush(int hand[])
{
  sort(hand, hand + 5);
  int suit = hand[0] / 13;
  int flush[] = {ACE, TEN, JACK, QUEEN, KING};

  for(int i = 0; i < 5; i++)
  {
    int card = hand[i] % 13;
    card = card == 0 ? 13 : card;
    if(card != flush[i])
    {
```

Sanjeev Penupala
SXP170022
Section: 501

```
      return false;
    }
  }

  return true;
}

bool black_two_pair(int hand[])
{
  int cards[5];

  // Checks if all suits are black
  for(int i = 0; i < 5; i++)
  {
    int suit = ceil((double) hand[0] / 13);

    if(suit != CLUBS && suit != SPADES)
    {
      return false;
    }

    int card = hand[i] % 13;
    cards[i] = card == 0 ? 13 : card;
  }

  sort(cards, cards + 5);
  int pair = 0;
  // Checks for 2 pairs
  for(int i = 1; i < 5; i++)
  {
    if(cards[i] == cards[i - 1])
    {
      pair++;
    }
  }

  return pair == 2;
}


/* MAIN CODE */
int main()
{
  // Checks Your Hand, If You Got a Whacky Poker Combination
```

Sanjeev Penupala
SXP170022
Section: 501

```
  int hand[5];
  int i = 0;
  int temp;

  while(i < 5)
  {
    cout << "Enter a number between 1 - 52: ";
    cin >> temp;
    bool exists = find(begin(hand), end(hand), temp) != end(hand);
    while(exists || temp < 1 || temp > 52) {
      cout << "You have already used this number or its out of range. Please enter a
number between 1 - 52: ";
      cin >> temp;
      exists = find(begin(hand), end(hand), temp) != end(hand);
    }
    hand[i++] = temp;
  }

  if(heart_royal_flush(hand))
  {
    cout << "Congrats! You got a hearty royal flush!";
  }
  else if(black_two_pair(hand))
  {
    cout << "Congrats! You got a black two pair!";
  }
  else
  {
    cout << "Aww man...seems like you didn't get any combo! For sure next time!";
  }
}
```

Sanjeev Penupala
SXP170022
Section: 501

WITHOUT USING ENUMS

```cpp
#include <iostream>
#include <string>
#include <algorithm>  // for std::find
#include <iterator>   // for std::begin, std::end
#include <cmath>      // for std::ceil

using namespace std;

/* POKER FUNCTIONS */
bool heart_royal_flush(int hand[])
{
  sort(hand, hand + 5);
  int suit = hand[0] / 13;
  int flush[] = {1, 10, 11, 12, 13};

  for(int i = 0; i < 5; i++)
  {
    int card = hand[i] % 13;
    card = card == 0 ? 13 : card;
    if(card != flush[i])
    {
      return false;
    }
  }

  return true;
}

bool black_two_pair(int hand[])
{
  int cards[5];

  // Checks if all suits are black
  for(int i = 0; i < 5; i++)
  {
    int suit = ceil((double) hand[0] / 13);

    if(suit != 1 && suit != 2)
    {
      return false;
    }
```

Sanjeev Penupala
SXP170022
Section: 501

```
    int card = hand[i] % 13;
    cards[i] = card == 0 ? 13 : card;
  }

  sort(cards, cards + 5);
  int pair = 0;
  // Checks for 2 pairs
  for(int i = 1; i < 5; i++)
  {
    if(cards[i] == cards[i - 1])
    {
      pair++;
    }
  }

  return pair == 2;
}


/* MAIN CODE */
int main()
{
  // Checks Your Hand, If You Got a Whacky Poker Combination
  int hand[5];
  int i = 0;
  int temp;

  while(i < 5)
  {
    cout << "Enter a number between 1 - 52: ";
    cin >> temp;
    bool exists = find(begin(hand), end(hand), temp) != end(hand);
    while(exists || temp < 1 || temp > 52) {
      cout << "You have already used this number or its out of range. Please enter a
number between 1 - 52: ";
      cin >> temp;
      exists = find(begin(hand), end(hand), temp) != end(hand);
    }
    hand[i++] = temp;
  }

  if(heart_royal_flush(hand))
  {
    cout << "Congrats! You got a hearty royal flush!";
```

Sanjeev Penupala
SXP170022
Section: 501

```
  }
  else if(black_two_pair(hand))
  {
    cout << "Congrats! You got a black two pair!";
  }
  else
  {
    cout << "Aww man...seems like you didn't get any combo! For sure next time!";
  }
}
```

As this is relatively a small program, I will speak in that context. In terms of readability between both programs (enums and no-enums), it is clear to me that the readability is at best when enums are used. They are essentially filler words for integers, which explains variable assignments to a developer, that an integer doesn't because it doesn't hold any contextual significance. In terms of reliability, when developing this program further, any developer would be able to produce better code and consistent code with desired output, with enums, and we know what enums are being exactly used. I used cards and suits as my enums, and when playing any card game, it is best to define what card and suit you have in your hand, rather than assigning a card a number. It ensures that you know what each variable is doing and going.

Sanjeev Penupala
SXP170022
Section: 501

5)

The main argument for the exclusive use of Boolean expression in control statements in languages like Java is reliability. The reason it is reliable in Java is because Java always results in a True or False and disallowed any other types to be returned from control statements.

Arithmetic expressions in languages like C/C++, allow any type to be in there control statements, which result in typing errors, meaning that incorrect types in the control statements are not detected by the compiler, which will cause confusion and errors later down the line of programming.

The main disadvantages of Boolean expressions exclusively are demotion in readability and writability.

6)

There are a couple of differences between C++ pointers and Java references. First and foremost, C++ allows arithmetic between pointers, while Java disallows that for its references. Secondly, the concept of pointer is implicit in Java, not explicitly stated by using an *, like in C++. In terms of safety, pointers are very dangerous to handle and use in C++, as memory access and memory allocation are difficult concepts to understand and could cause major problems, while coding in C++. Java doesn't use pointers, not only to increase safeness, but enforces a more secure way to accessing memory in the heap. Lastly, in terms of convenience, Java uses a garbage collector to automatically deallocate memory that is not being used in the program, while in C++, you must deallocate memory that you allocate after you are finished using it, which can add a lot of lines of code, depending on the complexity of your project.

Sanjeev Penupala
SXP170022
Section: 501

7)

```
import static java.lang.System.*;

class Main
{
  static int multiply(int x, int y)
  {
    return x * y;
  }

  public static void main(String[] args)
  {
    int a = 2, b = 7, c = 8, d = 9;

    int result = a + b * multiply(c, d);

    out.println(result);
  }
}
```

The main operation that is being evaluated here is `int result = a + b * multiply(c, d);`.

The first thing that is evaluated here is the function call multiply (). This takes highest precedence in any operation. When we multiply, 8 and 9, we get 72 as our answer. The equation then becomes `result = 2 + 7 * 72;` Because multiplication is the next highest order in this equation, 7 * 72 is multiplied to get a result of 504. Then, we finally have `result = 2 + 504` where we have addition left. When that is finished, our result is 506. In terms of order precedence, Java evaluated the result in this order: *method -> multiplication - > addition*.

Sanjeev Penupala
SXP170022
Section: 501

8)

- Python is an interpreted language because it goes through an interpreter, which takes the code you write and writes it in a language that your computer's processor recognizes. Then, that intermediate language is then converted to machine code that can be run. It allows Python to have a high-level of language.
- Arrays and Lists are very similar in Python, in terms of mutability. The only difference between them is that you must define your data type for an array beforehand, unlike Lists, where its not restricted to one datatype. A tuple on the other hand is like a list, but it is not mutable. And lastly, a record is somewhat different from these other data types, as it is closely related to the built-in dictionary in Python. PyRecords mimic structs from C++, in that you can define variables as keys, and its assignment is the value. Records are immutable objects, in terms of adding more variables to a record.

```
# Python Arrays, Lists, Tuples, and Records
arr = arr.array("i", [1, 2, 3, 4])

list = [1, 2, 3, 4]

tuple = (1, 2, 3, 4)

Customer = Record.create_type("Customer", "name", "email_address", "dob", "credit_card")
bob = Customer("Bob Ross", "bob.ross@hotmail.com", "10/29/1942", "1518458768551234")
print(bob.name)
```

- [::-1] is syntax for python slicing. It allows you define a slice of a from a datatype, like list or string. The syntax is: [start:end:step]. By leaving the start and end variables empty, we assume that we would like to encapsulate the entire variable for whatever we are slicing for. The step determines whether how many positive steps we are taking and how many negative steps we are taking. In this case, we are doing -1 steps, meaning we are traversing the list or string backwards, or essentially **reversing** the variable

```
l = [1, 2, 3, 4]
s = "1234"

print(l[::-1])
print(s[::-1])
```

- Use the shuffle() method from the random library in Python, to randomize a list of items in-place

```
# Randomize List In-Place
l = [1, 2, 3, 4, 5, 6, 7]
random.shuffle(l)
print(l)
```

Sanjeev Penupala
SXP170022
Section: 501

- range() and xrange() are both used to generate objects that's most common use case is to loop through. Keys differences are that range() object returns a list object and xrange() returns a generator object that displays numbers only by looping through them one by one. Because of how the objects are returned, xrange() tends to take less memory space than the range() object. [NOTE: These differences apply only for Python 2.7, as xrange() is taken out in Python 3+]

```
# range vs. xrange
a = range(1000)
b = xrange(1000)
print(type(a), "Size:", sys.getsizeof(a))
print(type(b), "Size:", sys.getsizeof(b))
```

- NumPy lists are faster than the traditional lists in python. This is primarily because numpy is written in C which is optimized for memory. When representing the same data looping over numpy lists will be much faster because of the way it points to the list in memory. The disadvantage is that numpy lists is that the data is restricted to only one data type.
- There are 2 ways to insert items into a list in Python. One is through the append() method and the other one is through insert() method. In the code below, I am appending a number to end of the list and inserting a number at the beginning of the list, by specifying which index I want to insert the item at.

```
# Add items to list
l = [0]
l.append(1)
l.insert(0, -1)
print(l)
```

- split() is used to take in any string, and split the string into various substrings by a defined delimiter, and return them in a list. The default delimiter is any whitespace, and its most common use case is to split a string into a list of words.

```
# split()
s = "I want to break up this string into words as a a list"
l = s.split()
print(l)
```