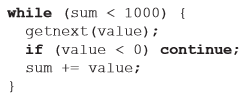
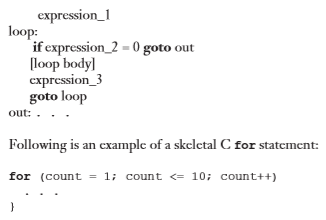
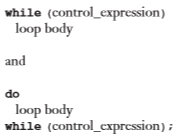
# 1. Loops

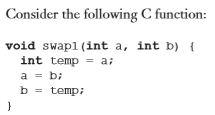
* **break/continue/goto (concepts):** In some situations, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body. As a result, some languages provide this capability. A syntactic mechanism for user-located loop control can be relatively simple, so its design is not difficult. Such loops have the structure of infinite loops but include one or more user-located loop exits. Perhaps the most interesting question is whether a single loop or several nested loops can be exited.
  + Break: unconditional unlabeled exits
  + Continue: unlabeled control statement that transfers control to the control mechanism of the smallest enclosing loop. This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop construct.
  + Goto: The motivation for user-located loop exits is simple: They fulfill a common need for goto statements using a highly restricted branch statement. The target of a goto can be many places in the program, both above and below the goto itself. However, the targets of user-located loop exit must be below the exit and can only follow immediately at the end of a compound statement.
* **break/continue/goto (code):**
  + Java break:  
    A screenshot of a cell phone

    Description automatically generated
  + Java continue:  
    
  + C++ goto:  
    
* **Counter-controlled loops:** in imperative languages use a counter variable, but such variables do not exist in pure functional languages. Rather than iteration to control repetition, functional languages use recursion. Rather than a statement, functional languages use a recursive function. Counting loops can be simulated in functional languages as follows: The counter can be a parameter for a function that repeatedly executes the loop body, which can be specified in a second function sent to the loop function as a parameter. So, such a loop function takes the body function and the number of repetitions as parameters.  
  A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
  + C   
    
  + Python  
    A screenshot of a social media post

    Description automatically generated
  + F#  
    A screenshot of a cell phone

    Description automatically generated
* **Logically controlled:** In many cases, collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression rather than a counter. For these situations, a logically controlled loop is convenient. Logically controlled loops are more general than counter-controlled loops. Every counting loop can be built with a logical loop, but the reverse is not true. Also, recall that only selection and logical loops are essential to express the control structure of any flowchart.  
  Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no **goto**
  + C#  
    
  + C++   
    

# 2. Subprograms

* **Parameter passing:** Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms.
  + In, out, in and out
    - Pass-by-value: in mode
    - Pass-by-result: out mode
    - Pass-by-value-result: in-out mode
    - Pass-by-reference: in-out mode
    - Pass-by-name: in-out mode
  + Conceptual model of parameter transfer:
    - By value: physically move a path
      * *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
    - By reference: move an access path
      * *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)
  + Pass-by-reference are the simplest to implement; only an address is placed in the stack
  + A subtle but fatal error can occur with pass-by-reference and pass-by-value-result: a formal parameter corresponding to a constant can mistakenly be changed
* Scoping:
  + Static scoping: A *static link* in an activation record for a subprogram points to its static parent’s activation record. The static chain from an activation record connects it to all its static ancestors
  + Dynamic scoping:
    - *Deep Access*: non-local references are found by searching the activation record on the dynamic chain. Length of the chain cannot be statically  
       determined
    - *Shallow Access*: put locals in a central place. One stack for each variable name. Central table with an entry for each variable name (details omitted)
* Generic concepts and coding (Java/C++):
  + Pass-by-reference: 
  + Pass-by-value: 
  + Parameter passing example:  
      
    // pass-by-value (c and d are unchanged because nothing transmitted back to the caller)  
    A screenshot of a cell phone

    Description automatically generated// the values of c and d are in fact interchanged  
    A screenshot of a cell phone

    Description automatically generated // C++ using reference parameters  
    In Java, a reference variable can point to only an object, not a scalar value.
* High order: A higher-order function, or functional form, is one that either takes one or more functions as parameters or yields a function as its result, or both.
* Lamda (concept): A lambda expression specifies the parameters and the mapping of a function. The lambda expression is the function itself, which is nameless. When a lambda expression is evaluated for a given parameter, the expression is said to be applied to that parameter.

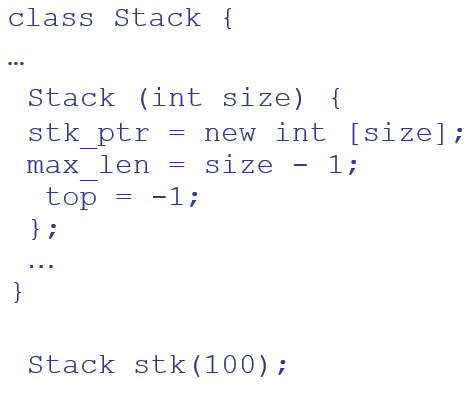
# 3. ADT and OOP

* Abstraction: a view or representation of an entity that includes only the most significant attributes. In a general sense, abstraction allows one to collect instances of entities into groups in which their common attributes need not be considered.
* OO features: OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding
  + Exclusivity of objects: object allocation, de-allocation, initialization
  + Abstract data type: classes in C++, Java
  + Inheritance: allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts (reuse ADTs after minor changes and define classes in a hierarchy)
  + Polymorphism: virtual functions
  + Notes:
    - ADTs are usually called *classes*
    - Class instances are called *objects*
    - A class that inherits is a *derived class* or a *subclass*
    - The class from which another class inherits is a parent class or *superclass*
    - Subprograms that define operations on objects are called *methods*
* ADT and parameterized ADT (Java vs C++): An *abstract data type* (ADT) is a user-defined data type that satisfies the following two conditions:
  + The representation of, and operations on, objects of the type are defined in a single syntactic unit
  + The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition
  + C++  
    A screenshot of a cell phone

    Description automatically generated  
    A screenshot of a cell phone

    Description automatically generated  
    A screenshot of a social media post

    Description automatically generated
  + Java  
    A screenshot of a cell phone

    Description automatically generated
  + Parameterized ADT C++  
      
    A screenshot of text

    Description automatically generated
  + Parameterized ADT Java  
    A screenshot of a cell phone

    Description automatically generated

# 4. ICE7 Exam2 Mock Test

1. A Python for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). For example:

for x in "banana":  
   print(x)

Compare Java/C++ for loop with that of Python, what is the key difference in loop design?

**Python: for loop counter controlled; Java/C++: for loop could be counter controlled or logically controlled.**

1. Statements such as break and continue in Java/C++ are controversial. Name an advantage of using such statements. Name a problem that may be caused by using such statement.

**Advantage: writability; Problem: reliability or readability (or: easy to make mistakes in programs; hard to understand code’s meaning; etc.) (1 point)**

1. Consider the following C++ like function:

void multiply (int n, int m)

              {n = n + 1; m = m \* n;  return;}

//code in main()

           int a = 3;

int B[5] = {2, 4, 6, 8, 1};     #index of B starts from 0.

           multiply(a, B[a]);

What will be the values for ***a*** and the array ***B*** after the subprogram call?

(a)  Assume that the parameters are passed by value. **a, B no change, i.e. a=3, B={2,4,6,8,1}**

(b)  Assume that the parameters are passed by reference. **a=4, B={2,4,6,32,1}**

(c) Assume that the parameters are passed by name. **a=4, B={2,4,6,8,4}**

1. Subprograms
2. The following Python code illustrates a higher order function map. Use this example to explain what is a higher order function.

names = [“john”, “dave”, “jessica”, “allen”, “pat”]

bigNames = map(upper, names) #upper() is a function converting to uppercase

**Higher order function: takes function as parameter.**

1. Does Java support higher order function? If yes, would you be able to convert the above code to Java? If no, say so. If yes but you’re not sure the syntax, please design one to add this feature in Java, i.e. convert the above code to the one you designed for Java.

**Yes (since Java 8); coding: answer varies.**

1. Inheritance
2. Convert the following C++ code skeleton to corresponding Java code. Do not change features used in the code (i.e. if inheritance used in C++ code, it must be used in the same way in Java code.)

class Shape {

public :

void draw () = 0;

void area() { … return …; }

private:

…

}

class Cirle : public Shape {

public:

void draw() {cout << “Draw a circle!” << endl; }

private:

…

}

**abstract class Shape {**

**public abstract void draw() ;**

**public area() { … }**

**…**

**}**

**class Circle extends Shape {**

**…**

**}**

1. In terms of features used in the codes above, which one, C++ or Java, has better design?

Use language evaluation criteria to briefly justify your answer.

**Java better readability**

1. Java supports an interface in addition to abstract class. What is the similarity and difference between Java’s interface and abstract class.

**interface: all methods implicitly abstract, cannot have implementations**

**abstract class: may contain both abstract and non-abstract methods, i.e. some methods may have implementation.**

1. Someone suggests that Java doesn’t need to support both interface and abstract class, supporting one of them sufficient. Comment on this idea (based on your own judgement.)

**Answer varies.**

1. Dynamic Binding

Given a code skeleton in pseudo code:

public class Old {

…

public void f () { … print out “Hello” … }

public void g() { … print out “Bye” … }

}

public class Young extends Old {

…

public void g() { … print out “Yeh” … }

}

//somewhere in main method

Young john = new Young ( …);

john.f(); //question #1 and question #2

Old kate = new Young ( …);

kate.g(); //question #3

Question #1: Assume the language uses static binding only, will this call result an error? If the call does not result in error, what will be printed out? **Hello**

Question #2: repeat Question #1 but this time assume the language uses dynamic binding only. **Hello**

Question #3: Assume the language uses dynamic binding, will this call (call to g, see above) result an error? If not, what will be printed out? **Yeh**

Question #4: How does Java support dynamic binding? **By default**

Question #5: How does C++ support dynamic binding? **Using virtual function or by virtual keyword**