BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

Important Problem Solving Elements

Computer science is all about using computers to solve problems. In this course you will be writing computer programs, or software, to solve many different kinds of problems. This document will serve as a brief tutorial on problem solving.

There are four very important aspects to successfully solving problems with computer programs:

1. **Make sure you have a thorough understanding of the exact problem to be solved**. This is critical. If you don't have a clear and thorough understanding of the problem, your resulting computer program may solve the wrong problem or may provide an incomplete or incorrect solution to the problem at hand.
2. **Make sure you have a carefully planned strategy for solving the problem**. Before starting to write your code it is very important to develop the approach you will take in solving the problem at hand. A highly recommended practice is to use *pseudocode* and *Top-Down Stepwise Refinement (TDSR*) to sketch out your approach. Pseudocode is also sometimes referred to as *program design language*. Pseudocode is essentially a language that allows a programmer to sketch out the steps that will be used to solve a problem without having to worry about following the strict syntax rules that a programming language like Java will enforce. For example, the following pseudocode outlines the major steps involved in preparing a bowl of cereal:

```
Get ingredients
Add cereal to bowl
Add milk to bowl
```

BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

TDSR is an approach to writing the pseudocode steps that starts with a very high-level step and iteratively refines that step into smaller, more specific steps. For example, the above pseudo code can be further refined by defining the specific ingredients that are needed in the first step:

Get a bowl
Get a spoon
Get milk
Get a box of cereal
Add cereal to bowl
Add milk to bowl

Students often rush right into writing code to solve problems…and that is not the best way to proceed. Try to develop the habit of designing your solution approach using pseudocode and TDSR. It can save you a considerable amount of time that would otherwise be spent debugging and reworking your code.

Pseudocode and TDSR are only two of many techniques that you will learn about in this course that will help you design better problem solutions. Another technique that many programmers use is flowcharting. This tutorial illustrates a type of flowchart called an *activity diagram* that can provide a visual view of a solution procedure.

Regardless of the tools used, what is important to keep in mind is that you should think through and design your problem solution approach before starting to write the code. As you progress through this course the problems you will be asked to solve will require that a substantial amount of design work be done prior to coding. And for most advanced computer science problems and real-world programming problems, coming up with an effective design solution prior to coding is a critical success factor.

Another thing to be aware of when planning your solution approach is that for many, if not most, computer science problems there can be several acceptable alternative solutions. This is where taking the time to "think out" your approach can be very helpful, and by sketching out alternative solution strategies using pseudocode it may be easier to pick out the most effective approach.

3. **Make sure you understand the kinds of building blocks that are available for solving the problem**. The capabilities provided by your programming language determine, to a large extent, your program building blocks. The textbook reading assignment for this course module introduced you to the concept of a variable and a few built-in Java functions, so those are the building blocks you have to work with for now. You will learn about many more building blocks as you progress through this course, but for now these will be the building blocks you have available to use.
4. **Make sure you use recommended program design and construction techniques**. In this course you will be exposed to many "best practices" that can be used to design and code programs, and it is important that you learn them and follow them in your own work. These practices are used by software professionals and help to improve the readability, reliability, reusability, and understandability of your designs and code.

Tips for Formulating Problem Solutions

Once you have a good understanding of the problem to be solved you can begin to sketch out a solution strategy. Here's a short list of questions that you might ask when formulating your approach to solving the problem at hand:

- What output information, if any, must my program produce?
- What input information, if any, does my program require?

BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

- How will my program get the input information it needs?
- What editing, if any, must my program perform?
- What information, if any, must be saved?
- What assumptions, if any, can I make?
- Are my assumptions valid for the problem I am trying to solve?
- What would be the impact on the problem solution if my assumptions are not valid?
- Can I simplify the overall problem to be solved by breaking it into sub-problems?
- What alternate solutions are possible?

These questions don't have to be asked in exactly the order presented. For example, I might start by listing any assumptions I can make. Nor are they intended to be an exhaustive list of questions. They are provided here to help you get started. In fact, it would be a good idea to be aware of additional questions that you find yourself asking when you design problem solutions and add them to this list. Such a list can be a valuable asset and can help you save time and rework effort. You will also find that as you answer these questions more questions will arise and you will revisit questions multiple times and most likely refine your solution strategy.

Let's take an example to see how we can use these questions to help structure a problem solution. Let's suppose we were asked to write a computer program that will serve as a simple grade book for a teacher. The program should process a set of student test scores for an exam, calculate the lowest exam score, the highest exam score, the average exam score, the number of students that passed the exam, the number of students that failed the exam, and it must keep a record of each student's exam scores. The calculations should be displayed on the computer screen.

Introduction to Problem Solving

What output information, if any, must my program produce?

The basic output information is pretty clear. The program should calculate lowest exam score, highest exam score, average exam score, number of students that passed, and number of students that failed the exam. And these calculations must be displayed on the computer display device. It must also save exam information for each student. This is also an output, but it won't be displayed on the computer screen. This output information will need to be stored on the computer's file system. This then brings up additional questions and decisions regarding how exactly that information will be stored. Right out of the gate there are at least three possibilities for file storage strategies: the program could create a single file with all the information stored in it, it could create a file for each student, or it could create a file for each exam. I'll have to do more thinking about this a bit later on.

What input information, if any, does my program require?

The program obviously needs a set of exam scores. It also needs a way to associate an exam score with a specific student, so it will need a student name or student number, or maybe both, for each exam score. Any other input information that you can think of? Well, the program needs to determine whether a student passed the exam or failed the exam, so it needs to know the lowest passing grade.

How will my program get the input information it needs?

Let's take the student and exam information first. There are two general possibilities here…a user can be prompted to enter the information, or a user can type the information into an external file and the program can read the information from the file. So, how do we decide which one to select? In practice, I should go back to the teacher who is requesting the program and ask him or her which they would prefer. Another possibility is to make an assumption

about what I think the teacher might prefer…but this might not be the best course of action. If I do take this course of action, I should add this decision to my list of assumptions and revisit it when I review my assumptions. For discussion purposes, let's assume I asked the teacher and she said her vision was that the program would prompt her for inputting the data. So, now we know how the program will get that data. Oh yeah…and while I was at it I also asked her how she would like to associate students and test scores…and she replied that she would like to input a student number with each test score. That actually answered one of the issues that arose from my second question about what kind of input the program would need. Now, what about the lowest passing score information? I knew, and she confirmed it, that the lowest passing exam score is 70. I'm now going to make a few related assumptions about this piece of information. My first assumption is that the lowest passing score will be the same for every exam the teacher gives. My second assumption will be that there is no need to input this information every time the program is run. So…I will have the program store this information in the form of a constant. Okay, so now I know exactly how the program will get the information it needs.

What editing, if any, must my program perform?

Editing is often required for data that is input, especially when that data is input via user prompts. Editing is sometimes needed for calculations as well, but we'll ignore that for this example. Since the teacher will be prompted to input student number and exam score, these are candidates for editing. Let's start with exam score. I'll assume that the lowest possible exam score is 0 and the maximum possible exam score is 100. This seems like a safe assumption, but I still need to add it to my list of assumptions and verify it. For example, suppose the teacher gives an exam with bonus points? So, right now I'll plan to edit the exam scores to make sure they are between 0 and 100. But what about the student number? How would the program be able to edit a student number? Well, one way would be to check if it is in an allowable range of

student numbers, but that assumes there is such a range. And if there is such a range that's additional information the program must know. See how that takes me back to questions two and three? Hmm…suppose the teacher just happens to type a student number that was in a valid range but it wasn't the correct number for that particular student. My program may need to use a table of valid student numbers and verify each user input against the numbers in the table. I'll need to go back to the teacher and get more information before I can decide how best to deal with this.

What information, if any, must be saved?

The specific information that must be saved is clearly understood…student number and exam grade…but how to implement this isn't so clear. There are several possibilities as mentioned before: one file, one file per student, or one file per exam. I'll need to explore this a bit further and maybe get some input from the teacher as well.

What assumptions, if any, can I make?
Are my assumptions valid for the problem I am trying to solve?
What would be the impact on the problem solution if my assumptions are not valid?

I've actually been working these questions in all during this discussion, so you can go back and read some of them if you'd like to.

Can I simplify the overall problem to be solved by breaking it into sub-problems?

Yes I can. There are several major things the program must do, so I can break up my solution strategy into solving those major pieces of the problem. The program must input data…that's one sub-problem. It must then edit that data…that's a second sub-problem. It must perform some calculations and display those calculations…that's two more sub-problems. And finally, it must save the student and exam data…that's another sub-problem. So there are five sub-

problems right off the bat. These sub-problems could serve as the top level activities in the first pass through a TDSR solution, which might look like this:

Prompt user for input information
Edit user input
Perform calculations
Display calculations
Save information

Each of these steps can be further refined as I make my concrete design and implementation decisions.

What alternate solutions are possible?

For this problem, the key thing that will drive solution alternatives is how to store the data. And that is pretty much independent of the other sub-problems. Depending upon the solution strategy, additional input information may be required…such as how to identify the data files.

Wow…we went through that short list of questions for a very straightforward computer problem and we did a lot of work. We made some design decisions and came up with some issues that need to be resolve before we go much further along. And…we haven't written a single line of code yet.

Can you see how important it is to really "think through" problems and solutions before writing any code?

Algorithms

An *algorithm* is a procedure that can be used to solve a computing problem. For now, we will focus on designing an algorithm that solves a single specific computing problem. Later on in the course you will develop solutions to problems that require several algorithms.

BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

An algorithm specifies two basic things:

1. The set of actions that must be taken.
2. The order in which these actions must occur.

Algorithms can be very simple or can be quite complex, depending upon the specific problem that must be solved. Here's an example of a simple algorithm, sketched out using pseudocode, that you've already seen. It describes how to prepare a bowl of cereal.

1. Get a bowl
2. Get a spoon
3. Get milk
4. Get a box of cereal
5. Add cereal to bowl
6. Add milk to bowl

In this example the algorithm rather trivial…it's just a set of obvious actions organized into a sequence of steps that can be followed to prepare a bowl of cereal…but, it serves its purpose.

Now, let's revisit something that was mentioned earlier in this document…*for many problems there can be more than one acceptable solution*. And this is true for our "bowl of cereal" problem. Note that the first four steps could be interchanged and rearranged in any order and still yield an acceptable result, so many alternative algorithms possible…more than you might realize at first.

So, speaking of problem solving, here's a problem for you to solve: How many possible algorithms could there be that solve our bowl of cereal problem by changing the order of the first four steps?

The last two steps could also be interchanged. We could add milk to the bowl first and then add the cereal. In fact, these steps could also be interchanged with some, but not all, of the other steps…resulting in even more acceptable alternative algorithms.

Program Control Structures

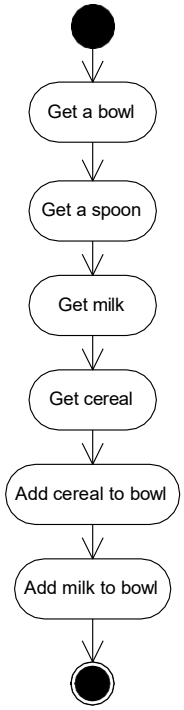
When you design an algorithm you must determine the specific actions to be performed (e.g., add cereal to bowl) as well as the order in which each action is performed. As was illustrated in the "bowl of cereal" example, certain constraints may exist that require some steps precede or succeed other steps. When we implement an algorithm in code we use *program control structures* to specify the order in which certain actions must be performed.

In the 1960s, two computer scientists demonstrated that all computer programs could be written in terms of three program control structures: *sequence* structures, *selection* structures (also called choice and decision structures), and *repetition* structures (also called iteration structures). Their work led to the birth of *structured programming*, a set of practices for writing programs that is still used by professional programmers today. Simply put, structured programming limits the allowable control structures to sequence, selection, and repetition, requires that each control structure have a single point of entry, a single point of exit, and prohibits non-structured transfer of control in a program by statements, such as *goto* statements, which can directly transfer the flow of execution in a program out of one control structure and into an arbitrary part of another control structure. The Java language defines a *goto* keyword but does not implement it. Other languages, like C++, C, and C# implement a *goto* statement, but professional programmers try to avoid using them. In this document we will illustrate how these structured programming control structures can be utilized in the design of simple algorithms.

BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

In this course you will learn quite a bit about the *Unified Modeling Language* (UML). The UML consists of many different types of diagrams that are used in software development projects. One type of UML diagram that is useful to help visually describe an algorithm is an *activity diagram*. An activity diagram can show the steps involved in an algorithm and the order in which those steps must take place. Our "bowl of cereal" algorithm could be expressed using an activity diagram as illustrated below:



Representing Sequential Control Structure Using a UML Activity Diagram

In the above example, the solid circle at the top of the diagram is technically called the *initial state*…and it represents the starting point of our algorithm. The slightly different circle, sometimes referred to as a "bullseye", is technically called the *final state*…and it represents the ending point of our algorithm. The oval-like symbols, technically called *action states*, represent

the actions or steps in our algorithm. And…since each step in our algorithm is performed in sequence…this algorithm is a *sequential control structure*.

In many algorithms it is necessary to use a *selection structure* (also called a *decision structure*). For example, if we examine our "bowl of cereal" algorithm, what happens if we are out of milk…or if we don't have any cereal? If we think these are reasonable things that could occur, we might alter our algorithm to incorporate these possibilities. A decision structure can be represented using pseudocode or an activity diagram. For purposes of discussion, let's assume we want to incorporate the possibility that we could be out of milk into our algorithm. A revised algorithm, using pseudocode, might look like this:

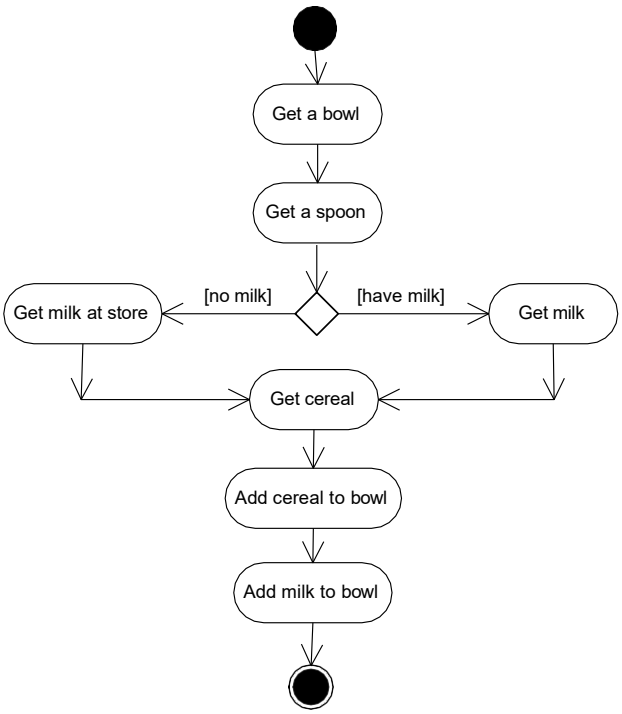
```
Get a bowl
Get a spoon
If we have milk
        Get milk
Otherwise
        Get milk at store
Get a box of cereal
Add cereal to bowl
Add milk to bowl
```

Representing Selection Control Structure Using Pseudocode

It's easy to see that in the revised algorithm a decision must be made as to whether we have milk. If we do, we get it, and if we don't we go to the store and buy some, and then continue with the next step of our algorithm…which is to get a box of cereal. Note how the decision was documented using the word "if". The "if" is creating a test…the test being "do we have milk?" You've already seen Java's if statement in the reading assignment for this course module, and that's one of the ways we can implement a decision structure when we ultimately write our code. Note also the use of the word "otherwise". This is used to document what our algorithm needs to do if the test for "do we have milk?" fails. So in this case, we're saying that we'd need

to buy some more milk before continuing with the algorithm. Sometimes, the work "else" is used instead of "otherwise". And finally, note how the actions "get milk" and "get milk at store" are indented slightly. Indentation is used to make the selection structure more visually obvious and to help improve the readability of the algorithm.

We can also represent our algorithm using an activity diagram, as illustrated below:



Representing Selection Control Structure Using a UML Activity Diagram

In the above diagram, the diamond symbol represents a *decision point*, and the terms in square brackets are called *guard conditions*. A decision point is just that…it's a point in our algorithm where a decision must be made…in this case do we or don't we have milk. Guard conditions can be any expression that evaluates to *true* or *false*. If a guard condition is true…for example, we have milk, then the execution of our algorithm branches off on the specified path…in this case

perform the “get milk” step…and our algorithm continues at the “get cereal” step. If a guard condition is false, then our algorithm branches off to an alternate set of steps.

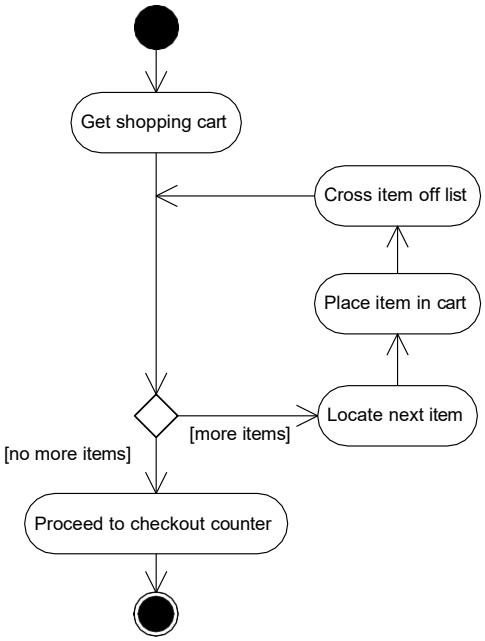
Some algorithms require that steps must be repeated. For example, consider part of an algorithm that determines whether we are finished shopping for groceries. Pseudocode for such an algorithm might look like this:

```
Create a shopping cart
While there are more items on my list
        Locate the next item
        Place the item in the cart
        Cross the item off the list
Proceed to checkout counter
```

Representing Repetition Control Structure Using Pseudocode

In the above example, the word “while” is used to specify that three actions must be repeated while there are still items remaining on my shopping list…locate the item, place it in the cart, and cross it off the shopping list. Once all the items on the shopping list are crossed off, the algorithm could proceed to another step, such as “proceed to checkout counter”. The word “while” is a common pseudocode choice for repetition control structures, but you can choose any word or phrase that implies repetition. Other examples could be “until”, “repeat until”, “do until” and so forth.

This algorithm could also be rendered using an activity diagram, which would look like this:



Representing Repetition Control Structure Using Pseudocode

The first step in the activity diagram is to “get shopping cart”. The next step is the decision point, where the condition “are there more items on my list?” is tested. If there are more items on the list, the item is located, placed in the cart, and crossed off the list. Control flow then merges back to the decision point, where the test is repeated, and the process continues until the “no more items” guard condition is true. At that point, the “proceed to checkout” step is executed

BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

Variables, Input, and Output

Depending on how detailed and complex the algorithm you are developing is, you may want to include the use of variables, input, and output. The use of variables, in particular, may be helpful for many algorithms. A variable is something that we use to store information used by an algorithm. Java has lots of different types of variables, and you'll begin to learn about them in the next course module…but for purposes of this tutorial it's not important to know about them. When we convert an algorithm into a computer program, the variables we use will store information only while the program is running. Here's an example of using input, output, and variables in the design of a program that will calculate the average exam score for a class of students.

```
Set total_points = 0          // total points earned on 10 tests
Set grade_counter = 0         // current number of grades entered by user

    // Read in test grades
    While grade_counter is less than or equal to 10
    Prompt user to enter next grade
    Input grade
    Add grade to total_points
    Add 1 to grade_counter

class_average = total_points divided by 10  // calculate class average

Display class_average
```

Note that I used several comments in the above pseudocode to explain the purpose of particular variables. Note also that I used italics to indicate the variables in the program. This is not mandatory, but some find it useful to distinguish variables from other text.

BU.350.700
Introduction to Java for Business

Introduction to Problem Solving

Summary

This document provides some tips, guidelines and techniques for problem solving that you can apply as you progress through the course.

Please remember to use only the structured program control flow constructs in your problem solutions.

Also, please note that we didn't introduce a single Java programming statement in this document because we wanted to get you thinking about the importance of solving the problem before jumping in and writing the code.