**Chapter XV**

**Program Design**

**Chapter XV Topics**

15.1 Introduction

15.2 The Design Compromise Triangle

15.3 Understand the Problem Description

15.4 Class Design

15.5 Method Design

15.6 Class Interaction

15.7 Testing and Debugging the Program

15.8 Information Hiding

15.9 The Tetris Game Program Design

15.10 Summary

**15.1 Introduction**

Many computer science textbooks include one or more chapters on program design. Such chapters can be found at various stages throughout the book. Some authors introduce program design immediately in an effort to create proper program development habits early. There are also authors who mention program design throughout the entire course to emphasize how design applies to the current topic.

The contents of Exposure Java and my feelings about *Program Design* are motivated by my early computer science courses that I took in the Seventies. My early beliefs about program design have been frequently confirmed and have also evolved with new trends in computer science.

In my very first computer science course, the first two lectures were very theoretical, and discussed proper program design. Basically, I was clueless. I did not yet know how to communicate with any type of computer. I knew nothing about any kind of programming language and I did not yet know the location of the university computer lab. But, ah. . . I knew about program design. After the conclusion of the two program design lectures, we received our first assignment. It was a very short program, which solved the *Quadratic Equation*. I looked at the assignment and tried to imagine how to apply proper design to the program and I was confused. The program I "designed" was barely ten lines in length. Next, I wondered how to log into the main frame computer.

I asked our instructor how to handle the login process. My instructor looked at me with perplexed eyes at the curious question. *"Your concern is with the implementation of correct program design. Details about logging in are trivial and of no consequence. Focus on the important topics in this course,"* I was told. This was excellent advice and I walked enthusiastically to the computer lab, after getting directions about its location. At the lab I asked the lab assistant how to log into the main frame computer and how to use the card punch machine. The very intelligent lab assistant spoke roughly twenty program languages, but was not very fluent in English. Two minutes later I was logged in. I still had no clue how to perform this task, but I sure learned that the lab assistant knew it.

Compile messages were sparse in those days. Essentially, there were two messages, which were *success* and *fatal error*. After a few fatal messages, the program assignment was done. It was actually much simpler than the login process and I was quickly finished. I did ponder how I had used correct program design and hoped that my printed assignment would not reflect my failure to apply the many fine principles that my instructor had labored so hard on to get me started correctly. What is the point that I am making here? My point is that I find it very difficult to use many program design principles for very short programs.

Over the years, and now over the decades, I have found that a program needs to have a certain minimum size before it starts to make sense to talk about program design. At this stage you are in unit 15 of this first computer science course or reviewing this material in a second course. You have already learned a considerable amount about programming with the Java language and you have also learned many Object Oriented Programming features. The time is now ripe to consider some tools that assist in the writing of large programs.

In the *pre-OOP* days, program design revolved around these five steps:

|  |  |
| --- | --- |
| **Step #** | **Step Mission** |
| 1 | **Understand the Problem** |
| 2 | **Develop an Algorithm** |
| 3 | **Code the Algorithm** |
| 4 | **Test and Debug the Program** |
| **5** | **Update and Enhancement** |

**Understand the Problem**

The first step in program design is always the same with or without *OOP*. It is also ignored more than anything else. Consider the following: Can a program expert with twenty years computer science experience write a program that will play chess, unless such a person understands the game of chess? Can somebody write a program which will multiply two matrices unless he or she knows how to multiply matrices? You know the answer to these questions. Therefore, before anything else happens, you need to understand the problem. This is particularly important, since the real situation is often such that problem originators are usually not programmers.

**Create an Algorithm**

An algorithm is a step-by-step solution to a given problem. Of the five steps, this will often be the most difficult and time-consuming step when program assignments start becoming complex. Entire textbooks are devoted to important algorithms. There will be a future chapter devoted entirely to popular computer algorithms.

**Code the Algorithm**

At this stage you need to determine the programming language that you will use. Java may not always be the best solution. As you become a more experienced programmer, you will be able to select from different languages for different applications. If the algorithm is well designed, it should comfortably translate into most programming languages. Furthermore, keep in mind that coding a program is not just completed because it works. Serious considerations need to be given to efficiency in execution time, in memory usage, and in program readability.

**Test and Debug the Program**

After a short introduction in Java you have learned to become dependent upon the compiler to point out errors. Keep in mind that the compiler is only equipped to detect *Compile/Syntax* errors. Both *Runtime* and *Logic* errors can only be detected when the program is executed and tested properly.

**Update and Enhance the Program**

Even programs that are very well planned and developed properly require updating and enhancing. Situations arise that were not considered in the early planning stages and the actual process of interacting with a working copy of the program often motivates improvement ideas.

Now that Schram has finished his nostalgic trip down memory lane we can return to the 21st century. Today, programs are more complicated. Object Oriented Programming was born to handle complicated programs in a reliable manner. The program design steps for OOP need to be adjusted. This is not to say that the previous five steps are meaningless. Even with *OOP* when you write an individual method, you will probably go through these exact five steps. When working with *OOP* there are issues that must be considered before you get down to the method level. A more up-to-date chart would look like this:

|  |  |
| --- | --- |
| **Step #** | **Step Mission** |
| 1 | **Understand the Problem Description** |
| 2 | **Class Design** |
| **3** | **Method Design** |
| **4** | **Class Interaction** |

Is this the first time that program design is mentioned? Hardly. Subtle statements about the appropriate use of various program constructs have appeared regularly along with frequent statements about the proper implementation of various classes and methods. However, it was mostly subtle. The first serious introduction to program design occurred back in Chapter VII. In that chapter you learned how to write your own methods and also how to create your own classes. The design aspects in that chapter could not be called *Object Oriented Design*, but it did present a very practical *Payroll* case study. With that program you observed many stages of the same program. The program evolved from a totally unreadable mess to a very manageable program. At the end of Chapter VII you should have realized the importance of the following fundamental rules:

|  |
| --- |
| **Fundamental Program Design Rules** |
| * **Write your program in a clear, consistent style.** * **Use meaningful, self-documenting identifiers.** * **Do not place all your code in the main method.** * **Create modules for recognizable tasks.** * **Place common purpose modules in a class.** |

These fundamental rules, provided in Chapter VII, were presented before you had learned much about Object Oriented Programming. I tend to have a rather simplistic view in life. It is my opinion that a discussion about the proper design of an automobile engine is difficult unless you know how an automobile engine actually works. The last chapter concluded the *Focus on OOP* topics and you have written a sufficient number of programs to start appreciating the importance of planning and designing a program. It is the aim of this chapter to give you some theoretical, yet practical rules, in designing a program.

The organization of this chapter is quite different from the pattern established by most of the previous chapters. You have observed that almost all chapters revolve around a sequence of program examples that are designed to illustrate one or more new programming concepts.

You also need to realize that this is AP Computer Science. What does that mean? Well for starters, you are expected to understand clearly how a program works. In other words, before you can design your own programs, you must be able to understand how somebody else's design works. You are also expected to design single classes. Design is great, but at some point comes implementation and students in a first course must be able to write the method body of the method signature described in a class design. Additionally, you should also be able to write a class that implements an interface.

Right now let us get serious and get a good introduction into *Objected Oriented Design* or *OOD*. I hope you like reading. This chapter is heavy on reading and basically extremely light on program examples.

|  |
| --- |
| **APCS Examination Program Design** |
| * **Comprehend the design of a provided program** * **Understand a problem description, purpose and goals** * **Apply data abstraction and encapsulation** * **Understand class specifications** * **Understand "is-a" class relationships** * **Understand "has-a" class relationships** * **Understand and implement a given class hierarchy** * **Identify and use existing class libraries** * **Identify reusable code from existing code** * **Design classes** * **Know implementation techniques** |

|  |
| --- |
| **AP Examination Alert** |
| **You can expect AP Computer Science Examination questions on the multiple choice segment and the free response segment about program design.** |

**15.2 The Design Compromise Triangle**

Something must be understood at the very beginning on any program design discussion. You know that life is full of compromises and priorities. Well it is no different in the world of computer science. Any large, important, program will have its share of compromises and priorities. The compromises and priorities can be illustrated with the *Computer Science Triangle of Compromise*, shown in figure 15.1. This triangle shows the interaction between four primary considerations in the design of a program.

**Figure 15.1**

|  |
| --- |
| Triangle |

This triangle is very practical. It is a constant reminder of the reality in writing a computer program. Is it *Speed* that you want? Oh, that is possible. Simply, avoid the creation of any method calls, because multiple method calls have a negative impact of execution performance. That works fine, but your *Readability* is pretty much destroyed with thousands of program statements shoved inside the main method. Are you concerned about using too much memory? There is a simple solution to the memory usage. Use data types that require little memory, use identifiers that are single letters and do not bother with any comments. Terrific, you have just saved all kinds of memory. Once again, *Readability* gets a pretty severe hit, and you may also get inaccurate computations due to memory overflow. Actually, you are most interested in *Readability*. After all, it is with readability that you can easily find errors, and future program enhancements are much easier to implement with a readable program. Do you see that *Speed*, *Readability* and *Memory* fight with each other for priority? What about that word in the center of the triangle? *Reliability* has not been mentioned in any of the previous discussions. The reason is simple. You may wish to sacrifice some *Speed*. You may wish to sacrifice some *Memory.* You may also wish to sacrifice *Readability*. However, *Reliability* is placed dead center inside the triangle to indicate that there is no compromise here. The three performance issues on the outside of the triangle are "desirable" goals.

Make the program as fast as possible. Make the program use as little memory as possible. Make the program as readable as possible. These are all goals that any good program designer considers and there will be some serious fights between program team members who feel strongly in other direction or the other.

However, and this is a very serious however, there can be no compromise in the reliability department. It is not appropriate to state that the program needs to be as reliable as possible. Imagine the following statement given to a patient after brain surgery with a computer-assisted-laser-scalpel: *"Mr. Jones, your surgery had a few complications. A wrong section of your brain was removed. However, you will be pleased to know that this was done quicker than any operation in the history of brain surgery.* Imagine Mr. Jones responding with, *"Well as long as it was done very quickly, it does not really matter that I still have a tumor in my brain, but no longer the part of my brain responsible for vision."*

Ladies and gentlemen please consider this. The primary reason to use Object Oriented Programming in modern programs is to ensure ***reliability***. All program decisions assume reliability and no compromise can be made here. Consider an ATM (Automatic Teller Machine) program that 90% of the time gives people the correct amount of cash during a cash withdrawal operation. Ninety percent correct is a remarkable performance on an AP Examination, but programs need to give the correct output 100% of the time.

|  |
| --- |
| **The Ultimate Priority** |
| **Program design becomes a compromise between speed, memory usage and program readability.**  **Program reliability is never a compromise option. It is a requirement that must be satisfied with the first program release and all tests are designed to insure the reliability of a program. All program considerations revolve around this very fundamental requirement.** |

**15.3 Understand the Problem Description**

Computer programs suffer from an unfortunate problem that is difficult to avoid. The people who write the programs do not specify the requirements for a program and the people who specify the requirements for a program do not write the program. Educators need grade book programs and educators know what they want in a grade book program. Programmers know what programs can do, but they do not comprehend the needs of teachers. So what happens? Educators give specifications to programmers to create a grade book program. The grade book specifications make perfect sense to educators and they may be unaware that many assumptions, known to every-day educators, are not explicitly explained in the program specifications.

Sometime later a program comes out and the educators are not happy. Is it because the program is not reliable? Our very conscientious programming company developed a program that was tested to the nth degree and it performed perfectly in all specified areas. That is just the rub. Not everything was specified accurately and completely.

Do you think I am exaggerating? About fifteen years ago my school district ordered the creation of a comprehensive school administrative and attendance program. The teachers were not real excited about the program, but teachers as a group have trouble getting excited about administrative computer programs. After several months into the program I marched to one of our attendance clerks and asked for the attendance records of one of my students. I wanted to know if this student was absent in other classes as much as my class. In other words, is it just me, or is this student absent with everybody. I asked a simple question. *"How many days is student XYZ absent in all her classes?"* The clerk responded that she did not know and I had to specify a time period. She would then check each day of the time period and record the absences. I was perplexed. To my amazement this very expensive, custom-designed attendance program, did not record or display the total number of days that a student was absent.

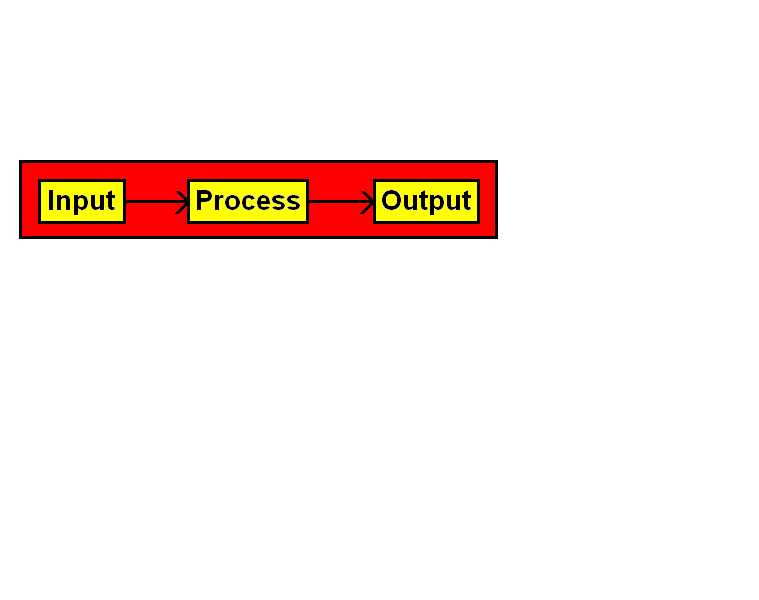
After some investigation I learned that the program worked correctly, but our district had never specified that this particular feature was a requirement. What was the problem? Most educators assume that it is established knowledge that an attendance program displays the total number of absent days for a specified student in a specified period. There is just one small snag, because your typical programmer is not your typical educator.

To this day the most practical educational program I have ever used was designed and created by Lynn Rosier. Mr. Rosier is one of those unusual combinations of a superb educator as well as a superb programmer. He did a great job communicating to himself.

This was a rather long-winded introduction to basically considering Step-1 of program design, which is **Understand the Problem**. This means that you, the programmer, need to both understand the problem and you must be able to handle the problem. You may completely understand that some client wants a program to play chess, but unless you personally understand the game of chess, you will not get too far in satisfying the client's requirements.

Now let us get practical. What actually is involved in understanding the problem? How do you prevent the frequent miscommunication headaches between clients, who are ignorant about programming and programmers, who are ignorant about the client's requirements. To put it simply, all program specifications can be simplified to the diagram in figure 15.2.

**Figure 15.2**



When somebody needs a computer program there is the option to purchase a program that satisfies the **input-process-output** needs of a person or - with enough money - a programmer can be hired to create a custom program. With either approach it is necessary to know the input and the output. The program user is unconcerned about the process.

Consider educational administration for a teacher, which was mentioned earlier. The input for the teacher includes: personal student information, test names and dates, test results by class, grade calculation considerations, student absences and anything else considered relevant. The output includes a wide variety of screens or printouts with grade statistics by student, by test and by class and report card summaries. It also requires attendance statistics per student, per class and per grade period.

Specifications run into two fundamental problems. First, a category is simply left out. Imagine that the specifications fail to mention the need for printing report cards. Teachers can happily enter tests and grades and then the school finds out that it has no means to generate report cards for a specified number of weeks. The report card feature was simply ignored. Report cards may be second nature to teachers, but not to people who think in Java, C++ or some other programming language. The second problem, and very common problem, is that the required category is included, but the simple **input-process-output** specification is incomplete or inaccurate.

Consider an *average grades* feature in a grade book program. The programmer is told that the program needs to be able to average grades for a cycle period. The program already has satisfied the ability to enter grades into a grade period of a specified number of weeks. So Mr. Programmer, make sure to average the grades in those grade cycles! Sounds easy?

Now the programmer may think that this is a no-brainer. Compute the grade average by adding up all the grades and divide by the number of grades. This type of average is called the *mean* of a set of numbers. This may sound fine, but it also means that a major test counts the same as a homework exercise. That cannot be right. There must be another way. Well there is not just another way; there are many other ways. Some teachers like to score all evaluations on a scale of 100, but different assignments carry a different weight. Other teachers count all grades the same within one category, but then assign percentages of the category during the grading cycle. Tests may count 60%, Quizzes 20% and Homework 20%. Many teachers do not use a scale of 100, but they use points. A quiz may be 20 points and a test may be 100 points, which makes the test worth five times as much as the quiz. Do you think I am finished? You'll be pleased to know that there is more, because there are teachers who like the *weight/percent* scale or the *point/percent* system and more.

The point here is not to bore you to oblivion about how teachers average grades. The point is that it is pretty easy to give an incomplete set of specifications to a programming company. If either the administrator ordering the program or the company writing the program, knows very little about the job of the other person, then there is bound to be trouble.

|  |
| --- |
| **Program Specifications Note** |
| **Program specifications follow a top-down approach.**  **This means that specifications are delivered from**  **general to specific.** |

You will see this "top down" statement mentioned at various locations in a chapter or article on program design. The two words *top down* simply means *from general to specific*. A business, which makes decisions *top-down* implies a business where there is strong leadership at the top, which then directs that policy down to the "bottom".

There are also *bottom up* businesses, which plan from the bottom up or at least the lower level employees have a considerable input in the decisions of the management. Such a business model is also called *site-based management*.

The construction industry shows us good analogies for specifications. The general contractor, who is the person in charge of coordinating all the different construction projects, looks at dozens of blue prints. The blue prints are detailed floor plans and all types of drawings and explanations on every aspect of the construction project. There exist blue prints for every type of job. The carpenter gets floor plans and specifications on the design of the roof. The plumber gets blue prints with exact locations of sinks, showers, washer, toilets and other plumbing needs. The electrician has blue prints that show the location of every light, electric plug, appliance and air-condition needs.

What does this mean to you if you need to design a program and start by writing down the specifications for a program? Well, you start at the top and begin by explaining the general purpose of the program. The beginning may be something like the following statements:

*This program calculates building materials for a construction project.*

*This program judges gymnastic competition.*

*This program manages business finances.*

*This program records and averages student grades for a teacher.*

*This program plays the game of "Tic-Tac-Toe."*

This list can continue with thousands more examples, as evidenced by all the software that has been created to help and entertain people with many different needs in their daily life.

Consider the first program example, calculating the amount of building materials required for a construction project. This type of program is tremendously practical for any professional builder. Trips to a building supply company are time consuming and take time away from construction. You want the right amount of materials. If too few materials are purchased, additional materials must be purchased or ordered. If too many materials are ordered there is a problem with the extra materials. The materials need to be moved from the building site and then it needs to be stored somewhere. Storing materials takes time and it takes money.

Following *Top-Down* procedure, the next step is to divide the problem into large categories. You know that the program has to calculate building materials. Now you continue and look at materials needed for *foundation, framing, windows, doors, roofing, sheetrock, siding, insulation, plumbing, electrical, air-conditioning, appliances, trim, carpeting, tile, cabinets, and paint.*

You now have a pretty good list of major categories and the *Top-Down* process continues. It was mentioned earlier that program design will always involve instructions about *Input - Process - Output*. Time will not be taken to design an entire house, but let us at least continue with the *foundation*.

What input is required for the *foundation*? The program will need the *size* of the house, the *shape* of the foundation. Most importantly is the *type* of the foundation. Is the *foundation* a concrete slab, a pier and beam foundation, or a basement structure?

Assume now that it will be a *concrete* foundation. The square footage of the floor plan will determine the amount of concrete that is needed, but that is not all. The concrete is reinforced with *rebar*, which are the iron rods that give strength to the concrete slab. The foundation also needs *cushion sand* and the *rebar* needs little elevators that raise the *rebar* to the correct position when the concrete is poured. Then there is the *plastic* sheet that goes under the concrete and do not forget about the temporary wood needed to create a form for the foundation. This is quite a list and I assure you that the foundation of a house is far less complicated than many other parts of a house.

The specification details keep on rolling and we move yet another notch down the *Top-Down* list. The *Concrete* itself requires details. *Concrete* is a mixture of sand, cement, aggregate (small stones) and water and the make-up of this mixture changes for different projects. The reinforcing *rebar* rods also need information. *Rebar* comes in stiff iron bars of different lengths and different diameters. It is also possible to use a flexible type of reinforcement that can be tightened after the concrete is poured.

Have I mentioned everything about the foundation? You may feel that any more details in this area and you are going to check the title of the course that you are taking? Is this *Computer Science* or *Construction 101*? Amazingly, after all this detail there is yet more to consider in the specification department and that is priority or sequence. A construction project cannot simply be put together as you wish. It may sound really silly, but if the *roofing* crew shows up before anybody else has done any work, then they will not have much to do. Does this mean that the foundation is first? No, ironically, the plumber has a job to do before the foundation gets too far along. The so-called *rough-in* plumbing comes first. These are the pipes that are placed under the foundation in special ditches. You may observe that when concrete is poured there are all sorts of pipes sticking out.

Finally, the section on problem specification is ended. Why was it so long? I am really trying to impress upon you how important and how complex this first part of program specifications is. Millions of dollars can be spent on a program or a construction project that does not satisfy the need of a customer, because the precise needs of the customer were not clearly specified.

**15.4 Class Design**

A class encapsulates the data and the actions that process the data. The selection, design and implementation of a class are the most fundamental parts of *Object Oriented Design* and we are up to our nostril in OOD*.* Did you forget why we do Object Oriented Programming? That is right, we want reliability and OOP gives us more reliability than the earlier styles of programming.

An Object Oriented Program revolves around a single class, if it is a very small program, or the interaction of many classes, if the program is medium to very large. You, the professional programmer, have all the program specifications and now the design process starts. Keep in mind that the creation of the program specifications is not part of the design. It is necessary for the design and you may frequently return to the customer to get clarification, but the design assumes the existence of detailed specifications.

I just love construction analogies. If you do not like construction analogies then this will become a long chapter. What is a class? Essentially it is a toolkit. A well-designed toolkit places tools for a common purpose together. Tools in a toolkit for an electrician handle the cables, outlets, switches and other electrical materials that electricians use.

When you are thinking about starting the writing of a program, your thoughts need to go to what toolkits or classes to use. Consider a program that is used to judge a gymnastics competition. What would you use? You need to think about the classes and you need to think about the interaction of the classes.

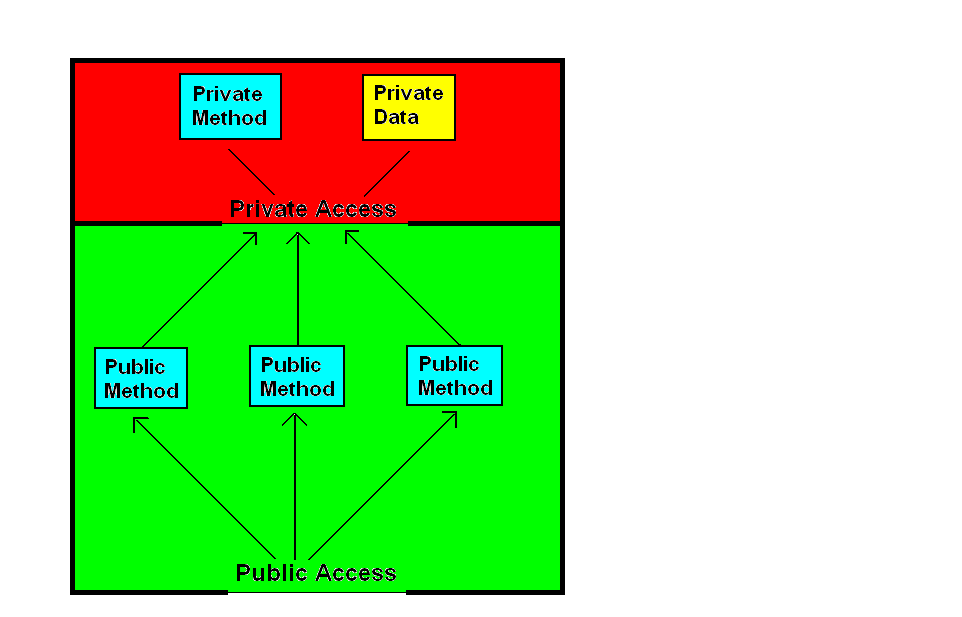
As you think about classes, remember that a class is a toolkit. Start by writing the class declaration, which includes the attributes and the method signatures. Your first concern is with the data structures that need to store data and the algorithms that will access and process the information.

Keep in mind that in a well-organized constructor's toolkit you will not find paint tools and plumber tools in the same container. Likewise in a well-designed class you should insure that data and the methods that access the data are all in the same class. Furthermore, do not place into one class the members that should be divided between two or more classes. In other words, keep the classes clean and manageable and not so large that the class is difficult to comprehend and also difficult to test properly.

When you place a set of members in a class, ask yourself if all these members satisfy the needs for a single goal. In some cases there are two or more groups of members that have separated functions. For instance a class that has methods to manage student homework and also to organize cookie recipes is an odd mix.

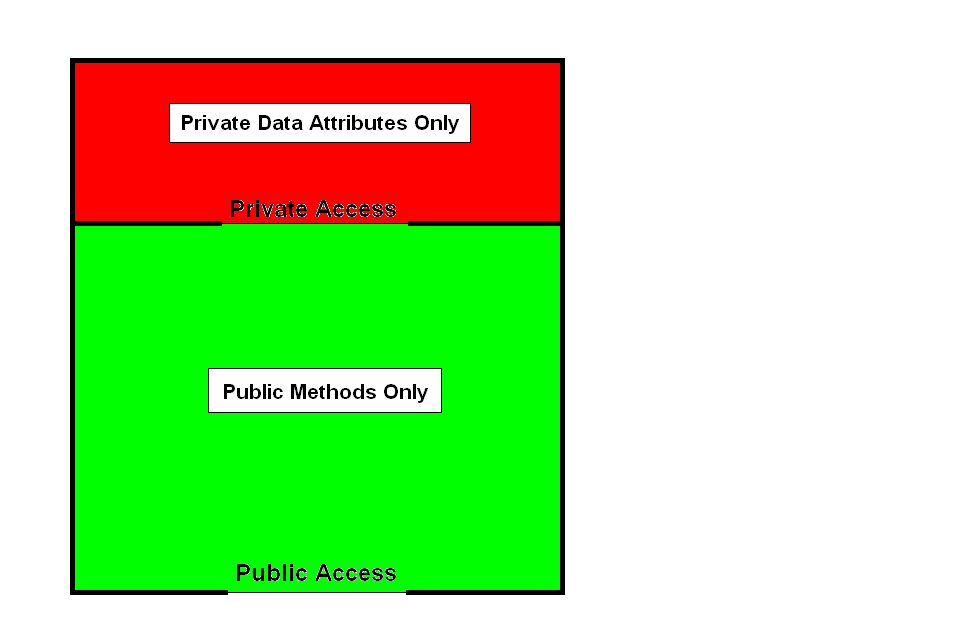
It is also possible that you wish additional classes, not because the members are working towards unrelated goals, but a large singular goal should be sub-dived into smaller, more manageable, goals. If you create a class for a school administration program with a single class called **Administration**, you may rapidly see the need for additional classes. How many methods would an **Administration** class have if it handled attendance, student admissions, school finances and student grades? It can be argued that all methods work together to process student data, but the class becomes too large. In such a case divide the class and create logical sub divisions. It makes more sense to create classes for **Attendance**, **Admissions**, **Finance** and **Grades**. After the class members are selected, you need to decide which members are **private** and which members are **public**. Figure 15.3 shows a diagram for a class with the two sections of *private access* and *public access*.

**Figure 15.3**



What determines if a class member has **private** access or **public** access? In a general sense the question is easily answered. Figure 15.4 shows the normal, recommended case. Data is **private** and **methods** are **public**.

**Figure 15.4**



|  |
| --- |
| **AP Computer Science Examination Alert** |
| **For the specific purpose of writing solutions to free response *class design* questions, declare data attributes private and action methods public.**  **Keep in mind that there are situations where it is appropriate for attributes to be public and methods to be private.** |

Figure 15.5 shows what is possible. Note that it is possible to declare attributes as **private** or **public**. It is also possible to declare most methods as **private** or **public**. It is possible to declare a *constructor method* as **private**, but it does not make any sense to do for any functional program.

**Figure 15.5**

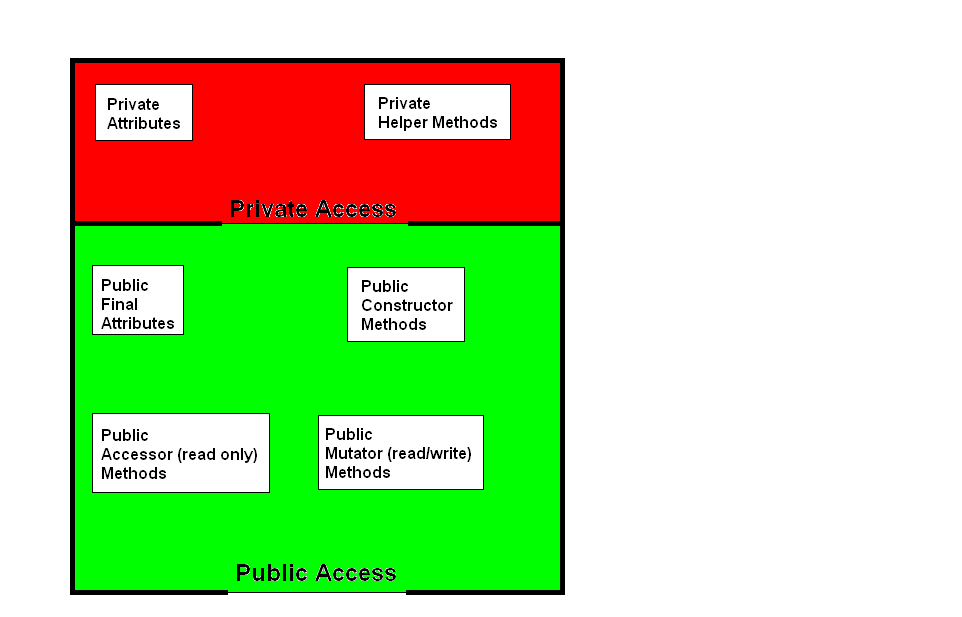


Figure 15.5 also shows the three different types of methods, which are *constructors*, *accessors* and *mutators*. All three methods have the opportunity to access data information, but the intent of each group is different. *Accessor* methods are *read-only* methods, which return data information, but do not change any data values. *Mutator* methods access data and store new data values. *Constructor* methods prepare new objects for reliable data processing.

Use *constructors* to provide initial values to the data of your class. Sometimes that may be sufficient, but think about reliability. The constructor sets the stage for the newly created object. Assigning initial values for data is important, but it is not always sufficient. One example from a previous chapter may remind you. In a **CardDeck**class the constructor needs to handle parameters for the number of card decks, the number of cards dealt with each hand and the number of players in the game. Additionally, the constructor of the **CardDeck** class also shuffled the cards of the newly created object.

|  |
| --- |
| **Class Attributes** |
| **Also Called: state, data, instance variables, fields** |
| **Class attributes store information in primitive data types or data structures, depending upon the information requirements.**  **Normally class attributes are declared private and should only be accessed by methods of the same class.**  **Sometimes class attributes may be declared public in the event the information is constant, like PI in the Math class. When class attributes are declared public, access should be strictly *read-only* and attributes must be declared as constants by using final.** |

|  |
| --- |
| **Constructor Methods** |
| **All classes need to be designed with one or more constructors.**  **Constructors are automatically called during the instantiation of an object.**  **At a minimum, constructors need to provide initial values for the data attributes of a class.**  **All constructors are declared public and they are neither void methods nor return methods.**  **Many classes require multiple "overloaded" constructors to allow objects flexibility during the instantiation of a new object.** |

|  |
| --- |
| **Accessor Methods** |
| **Methods are also called: actions, operations, procedures, functions, subroutines**  **Accessor methods are also called: get methods** |
| **Accessor methods are *read-only* methods. It is not required, but most accessor methods are *return* methods, which provide information about the private data information of an object.** |

|  |
| --- |
| **Mutator Methods** |
| **Methods are also called: actions, operations, procedures, functions, subroutines**  **Mutator methods are also called: set methods, modifier methods** |
| **Mutator methods are *read/write* methods. Program design requires careful scrutiny of the *mutator* methods*.* Mutator methods not only implement some algorithm to process data, but they also alter the data. Carelessly designed mutator methods can do serious, unwanted, damage to object data.** |

A common mistake with class design is to confuse good classes with good methods. It is the mistake where the designer realizes that some important process is required for a program and then decides to make the process a class. A class is not an operation. A method is an operation. Look at the **Math** class. Mathematics is not an operation; it includes many operations. Data requires organization with sorting and searching operations, but you should not design a *Sorting* class. Sorting data is an operation; it is a verb and that should be done in a method.

It is not easy to design a program properly and efficiently. The selection of classes is a key component of any object. Before we can learn much more about selecting classes we need to look at methods first. Methods are the bread and butter of programming. At the start of program design there are many requirements that must be implemented and some of these requirements will be implemented with methods. Common methods are then combined into a class to process the appropriate data. So we will return to this issue after spending some time with methods.

**15.5 Method Design**

At the method level the concept of *input-process-output* takes on a very direct meaning. Methods have *preconditions* and they have *postconditions*. It is the job of the method to implement an algorithm that takes the *precondition* to the required *postcondition*. A *precondition* is what you can assume is true before a method begins. A *postcondition* specifies what must be true at the end of the method -- provided the *precondition* was **true**. Two examples of method signatures with *preconditions* and *postconditions are* shown in figure 15.6 and figure 15.7.

**Figure 15.6**

|  |
| --- |
| // **precondition:**  list is a non-empty ArrayList object of Integer elements  // **postcondition:** The Integer values in list are sorted in ascending order  **public static void sortList (ArrayList list)** |

**Figure 15.7**

|  |
| --- |
| // **precondition:**  list is a non-empty ArrayList object of Integer elements  // The Integer elements in list are randomly ordered  // **postcondition:** getMedian returns the median value of list  **public static double getMedian (ArrayList list)** |

A method processes a singular operation. Do not confuse yourself and future programmers by creating exotic, multi-task methods. Remember the golden rule of *one task, one module*. You may even find that the implementation of one task may include several sub-tasks. These sub-tasks themselves may become methods.

But let us take care of first things first. I have seen hundreds, if not thousands of students stare at a method heading with a look that indicates a complete lack of confidence in the ability to write the method at hand. You see it is nice to make a statement like you need to write method **getMove**. Method **getMove** is meant to provide the next move in a chess game. I have intentionally given an example where you can see that talking about the requirements of a method is much simpler than implementing the operation. If you know the game of chess then you understand what is required. Look at the chess board. Consider the possibilities and select the next move. Now all you need to do is write the method.

The problem here is that students try to write the code before they have a clue about the necessary algorithm. An algorithm is *a step-by-step sequence to complete a given operation*. In simple English this means that you have no chance writing the program code to implement an operation, unless you personally know how to do the required operation. Let us go ahead and apply this principle to the method heading shown in figure 15.8. You are presented with the heading of a method to compute the Greatest Common Factor (**GCF**) of the two integer parameters. You remember that the GCF is the largest number that divides into both numbers. For instance, the GCF of 120 and 108 is 12. Now how do we implement method **getGCF**? Look at the hint below.

**Figure 15.8**

|  |
| --- |
| // precondition: n1 and n2 are positive, non-zero, integers  // postcondition: getGCF returns the Greatest Common Factor of n1 and n2  **public int getGCF (int n1, int n2)** |

|  |
| --- |
| **Method Writing Hint: Devise an Algorithm** |
| **If you cannot implement the code in the method body, take out a piece of paper and write down the actual steps necessary to solve the problem. You cannot write the steps in a program language, when you cannot write the steps in English.** |

A good approach with creating an algorithm is to first look at one specific example. You will see the truth that computer programming is not simply a *top-down* approach. It jumps all over the place between *top-down*, *sideways* and now in this case *bottom-up*.

Two thousand years ago, mathematician *Euclid* devised the steps to compute the *Greatest Common Factor*. I will use his solution to find the answer of the GCF of 120 and 108. The steps are shown in figure 15.9. I do the computation in the **Computation** column and then I write down in the **Process** column what I did.

**Figure 15.9**

|  |  |  |
| --- | --- | --- |
| **Step** | **Process** | **Computation** |
| **1** | Divide num 1 by num 2 | 120 / 108 = 1 |
| **2** | Determine remainder | 120 - (1 \* 108) = 12 |
| **3** | Check if remainder  equals zero | 12 is not 0 |
| **4** | Numerator (num1)  becomes Denominator | Num = 108 |
| **5** | Denominator (num2)  becomes remainder | Deb = 12 |
| **6** | Divide num1 by num2 | 108 / 12 = 9 |
| **7** | Determine remainder | 108 - (9 \* 12) = 0 |
| **8** | Check if remainder  equals zero | 0 = 0 |
| **9** | If remainder equals 0  then the GCF is  the denominator | GCF is 12 |

The process column tells the story of the algorithm, *Euclid's Algorithm* in this case. Figure 15.10 shows the steps of the algorithm. They are almost identical to the steps shown in figure 15.9. The main difference is that the algorithm provides a mechanism to repeat certain steps as frequently as is required. In essence, figure 15.10 makes the specific problem of figure 15.9 a general solution for all GCF computations of two numbers.

**Figure 15.10**

|  |  |
| --- | --- |
| **Step** | **Process** |
| **1** | **Divide num1 by num2** |
| **2** | **Determine the remainder of step 1** |
| **3** | **Check if the remainder equals zero** |
| **4** | **If the remainder equals zero, go to step 6** |
| **5** | **If the remainder does not equals zero, then**  **num1 becomes num2**  **num2 becomes the remainder**  **Go to step 1** |
| **6** | **The GCF is num2** |

Now we go back to implementing the **getGCF** method with Java code. We are now equipped with a functional algorithm and our job is much simpler. The mission is to translate the English algorithm steps into Java program statement. This is precisely what is done in figure 15.11.

**Figure 15.11**

|  |
| --- |
| **public int getGCF (int n1, int n2)**  **{**  **int temp = 0;**  **int rem = 0;**  **do**  **{**  **rem = n1 % n2;**  **if (rem == 0)**  **temp = n2;**  **else**  **{**  **n1 = n2;**  **n2 = rem;**  **}**  **}**  **while (rem != 0);**  **return temp;**  **}** |

At the method level, design was already discussed back in Chapter VII. This means you need to use self-documenting identifiers. You need to use comments that clarify the steps in your code. You need to use control structures with a consistent indentation style that is easy to follow.

**15.6 Class Interaction**

Program design is difficult to nail down in a very precise, yet realistic, set of steps. In the days before Object Oriented Programming many computer science texts suggested that programs are designed in a set of sequential steps that are very organized. That sounds good, but it is not that simple. It is easy to state that we follow step-1, step 2, etc., but is that what really happens? We start out with good intentions to create a program for a purpose, like organizing our product inventory. This means that *ProductInventoryProgram* is at the top of our design, and top-down design is the way to develop a program. Now we start to slowly work down the pyramid as we go from general to specific. In theory it may seem that there is a steady process of continuously becoming-more-specific-and-detailed in the design of our program. This is called "top-down design with step-wise refinement". In reality, it is will frequently happen that we encounter problems along the way or we realize that important considerations were overlooked that must now be addressed. The real-life program may bounce up and down considerably between top and bottom.

Object Oriented Programming does help tremendously. As we steadily develop a program there may be many classes that will be used. It is entirely possible that as individual classes are implemented with attributes and methods that it makes sense to add additional classes or possibly combine two classes into one class. Frankly, the way that I am speaking right now may seem pretty fuzzy. I would simplify my life and probably yours, if I simply listed a neat set of steps to follow and state that this is how you always design and write a program. Well theory is terrific, but I have been involved in the creation of some sizable programs, and I have learned that the reality of writing programs is sometimes less clean than a program design chapter may imply. The programs I wrote were sizable for a single individual, which in my case means programs greater than 10,000 lines. A program with 10,000 lines seems small compared to multi-million line programs, but I assure you that such large programs are designed and written by a considerable team of programmers.

|  |
| --- |
| **Program Design Reality** |
| **It is easy to explain program design as a clean sequence**  **of steps that steadily develop the finished program.**  **In reality all the aspects of the program requirements are rarely known at the start of program development and additions, deletions, alterations are common.** |

Object Oriented Programming allows calm sanity in the development of your program. In particular, you can completely concentrate on one area of the program and not be concerned about any other parts. The use of classes, which encapsulate all the data and those methods that access the data in one module allow this type of focus.

So here you are and you have created some practical classes for your program. The implementation of these classes may sometimes be possible in a vacuum. This means that such classes can be written and tested without consideration of any other classes. That works for some classes, like the **Math** class, but many other classes interact. Testing the reliability of many classes will depend on the interaction with other classes. There are three ways that classes can interact.

|  |
| --- |
| **Class Interaction** |
| * **Utility** * **Inheritance** * **Composition** |

**Class Interaction: Utility**

Some frowns may appear by established computer science professionals. What is this utility business? In the technical sense it is not much of a class interaction, but it certainly describes how a class may be used.

There are classes that provide useful tools that may be used at any part of a program. Such classes are utility classes and usually the methods are *class* methods or *static* methods. The **Math** class is prime example of a utility class. There is class interaction in the sense that you will create some client class, which includes methods that use **Math** class methods. The utility class is also an example of a class that can be easily tested independently.

In the area of design and implementation a *utility* class has the least complications. A set of method tools are created and then they are placed in a class toolkit. Each individual method is tested by some client program of the utility class and after thorough testing the utility class is ready to be used anywhere in the program.

**Class Interaction: Inheritance**

If you want reliability then inheritance is your friend. Start with a class, any class that is needed for your program and test this class thoroughly. Now you move on and you require several classes that are similar in nature to your established, tested class. It is not necessary to start from scratch. Do not reinvent the wheel, but use what is already available. Extend the existing class and then develop additional methods or redefine existing methods. Inheritance is called an *"is-a"* relationship. A classic example of using inheritance is available in the GridWorld Case Study. In this case study there is an **Actor** class. Objects of the **Act** class have a defined behavior and a set of methods. Now other actors come on the scene. There are **Rocks**, **Bugs**, **Flowers**, **BoxBugs** and **Critters**. We can now create new classes, where each one of the new classes extends the **Actor** class or some subclass of the **Actor** class. With inheritance we can completely rely on the features already established in the **Actor** superclass. Given that the superclass methods have already been tested properly, we can use these methods without additional testing or possibly make modifications. The job of the new subclasses is to *re-define* any existing method or newly-define one or more methods.

**Class Interaction: Composition**

Utility is great and inheritance is very nice, but the truth is that your program will probably rely more on composition than any other type of class interaction. Composition is the ***"has-a"*** relationship. With composition, programs become much more manageable. Consider a complex program used for hospital administration. You can start with a **Patient** class. Objects of the **Patient** class enter, store, display and alter personal information on any one patient. Another class may be **Patients**, which is a class that includes an array attribute of many **Patient** objects.

Now look at accounting. A **Billing** class may also contain an array attribute of **Patient** objects. Another basic class, like **Employee** will finds its way in a **Schedule** class responsible for the work schedule for all the doctors, nurses and other hospital employees. This same **Employee** classis also used by the **Payroll** class to take care of employee salaries. With all this interactive complexity, it will seem like a nightmare of organization. This is precisely where Object Oriented Programming assists. Remember that computer programs are designed to perform real-life task that formerly were performed by people. In real-life we are surrounded by objects and such objects need to be incorporated in computer programs.

A good starting place is with *unit* classes. Some years ago we (John Schram - son and Leon Schram - father) designed a program to handle the data processing needs for *Academic Decathlon* competition. This is a challenging competition in ten academic fields. Each team has nine members. The program handles all the score keeping, winner computations and medal standings award processing. Where did we start? The first - and foundation class - was the **Team** class. It is a classic example of a class that handles a single key unit. Each object of the **Team** class stores all the data information for one team. This includes name, competition level, ID numbers, coach's names, scores for each tests and award rankings. Objects of a single unit **Team** class become attributes in many other Academic Decathlon classes.

|  |
| --- |
| **Unit Classes** |
| **A unit class is a class that contains the attributes and process methods of a single practical unit.**  **Classic examples of unit classes are:**  **Student - Patient - Employee - Product - Passenger** |

The biggest mistake in program design is not to use unit classes. The result is that the classes become very large, which makes them difficult to test properly and manage efficiently. An exaggerated example would be the hospital program. Imagine that you create a single class called **Administration**. This class would be totally unmanageable and impossible to test correctly.

|  |
| --- |
| **Managing Problems and Programs** |
| **The secret of managing complex problems, or programs, is to divide the problem into smaller manageable chunks.**  **In the case of Object Oriented Programming this starts by**  **selecting appropriate classes. A good starting point is to select the "unit" classes for your program.** |

**15.7 Testing and Debugging the Program**

When an inexperienced student programmer finishes a program, and *it compiles*, there is often little concern given to proper test data. A particular set of data is used that is often the data specified for the program execution hard copy. Since the program compiles and executes, a premature conclusion is drawn that it must be correct. As a matter of fact, it is not unusual for students to turn in programs with totally illogical outputs. Students recognize compile errors, because the computer stops and indicates such errors. But there are in fact three kinds of errors.

|  |
| --- |
| **Three Types of Errors** |
| **• Compile or Syntax Errors**  **• Runtime or Execution Errors**  **• Logic Errors** |

Recognizing which one of the three errors occurs is the most important step in debugging a troubled program. Remember, no matter how stumped you are by your program’s uncooperative nature, you always should be able to state what type of error your program has. I can assure you that your teacher will be very pleased if you can say something like:

*Excuse me Mrs. Hromcik, my program has a compile error that says I have an unknown identifier,* or something like *Excuse me, Mr. Rosier, my program compiles and executes, but the output does not make any sense at all.*

Please do not impress your teacher with the following dialogue:

Student: *Can you help me?*

Teacher: *What do you need?*

Student: *I have a problem.*

Teacher: *What type of problem do you have?*

Student: *My computer has a problem.*

Teacher: *What is wrong with your computer?*

Student: *It does not work.*

Teacher: *What is* ***it*** *that does not work?*

Student: *My program does not work.*

Teacher: *What is working or not working with your program?*

Student: *I don’t know. It does not work.*

Teacher: *You must know what is wrong, even if you cannot fix the problem.*

Student: *If I knew what was wrong, I would not ask for help.*

Students often think that they have no responsibility in identifying problems. It is sufficient to tell a teacher that a problem exists. Having announced the existence of the problem, transfers ownership of said problem to the teacher. Student can now relax and teacher gets to work. This theory has some difficulties. First, there are a variety of teachers who give pitifully little help to students who have put forward minimal or no effort themselves. Second, you will soon find yourself in college or on the job where easy ownership transfer of problems is not an available option.

**Compile or Syntax Errors**

Remember that *Syntax* means *Language Sentence Structure*. All lan­guages, especially computer languages, have rules for proper ­sentence structure, or *syntax*. A program compiler goes through two steps. First, the compiler checks if the program has been writ­ten with acceptable syntax. Second, it translates the program. It would be inefficient to start the second step unless the *first step can be completed.* Since the compiler specifically checks syntax errors, and it gives a syntax error message, the error is also called a **compile** error. Your Java compiler rapidly finds any syntax errors, but you must accept the­ following three facts about syntax error messages.

|  |
| --- |
| **Compile Error Facts** |
| • The compiler will catch all syntax errors.  • The error message is not always an accurate  description of the actual syntax error.  • The compiler does not necessarily stop at the correct  *error location*. The *error indicator* stops either on or  **before** the actual error location. |

Students usually ignore cryptic error messages. Even if a message is totally logical, it is still overlooked in most cases. The real problem is that students expect the *error indicator* to be at the *error location*. I have time, and time, and time again seen students in total puzzlement. The error message was very accommodating stating: *Semi-colon expected*.

My hardworking, motivated students would look and look at the highlighted line with the error message. Everything looked just fine and the required semi-colons is in place on the indicated line. Amazingly, the missing semi-colon, at the end of the immediately preceding program statement was not noticed. Please realize that the compiler does not have intelligence. Certain syntax rules must be obeyed and failure to obey the syntax rules will cause problems. However, the compiler does not necessarily recognize what you did wrong or where. What happens is that your error brings about something that is not digestible to the compiler. At the moment that it is clear that a mistake is made, the compiler will let you know. This can be many lines below the actual mistake.

**Runtime or Execution Errors**

Runtime errors are much tougher to correct than syntax errors. Your program may compile just fine, but when you attempt to run it, the program "crashes." You may or may not get any useful message, but it will be very clear that something is wrong. There are many different causes for runtime­ errors, but in general let us define this error as follows:

|  |
| --- |
| **Runtime Error Definition** |
| A **runtime** error or **execution** error is an error that interrupts  program execution. This is more popularly known as the  program"crashes". |

In the greater majority of cases you will receive some kind of error message. Your first step with execution errors is to see if the message makes sense. Even more uncomfortable are the errors that cause a computer crash and there is no message at all. Sometimes you may even have to re-boot the system. At other times the computer is so distraught by the nature of your error that a *courtesy reboot* is automatically performed. Whether you get a message or not, your first task is to isolate the program segment that causes the runtime error.

Java does not use the term runtime errors or execution errors. Java has *exceptions* and all runtime error messages include the word exception and the type of exception that has been detected. There are five common runtime exceptions, which require closer inspection.

|  |
| --- |
| **NullPointerException** |
| **A NullPointerException occurs when an attempt is made to reference an object using an object variable that is null.**  **In a correct situation an object is a memory reference to a location where the object's attribute values can be accessed.**  **If the object's reference is a null value then any attempt to access some attribute field will fail.** |

|  |
| --- |
| **IllegalArgumentException** |
| **An IllegalArgumentException occurs when an illegal argument is used for a method call.** |

|  |
| --- |
| **ArrayIndexOutOfBoundsException** |
| **An ArrayIndexOutOfBoundsException occurs when an attempt is made to access an array with an index outside the range of the array's indexes.**  **This error is quite common, because students frequently forget that the number of elements in an array is not the same as the largest index in an array. An array with ten elements starts at index 0 and concludes with index 9.** |

|  |
| --- |
| **ArithmeticException** |
| **An ArithmeticException occurs an attempt is made to perform an illegal arithmetic operation. One example occurs when the program tries to divide by zero.**  **This error does not occur with all illegal arithmetic operations. For instance the statement System.out.print(Math.sqrt(-10)); does not display an exception message, but displays NaN, which means Not a Number.** |

|  |
| --- |
| **ClassCastException** |
| **A ClassCastException occurs when an attempt is made to cast a variable to a non-matching class.** |

Is this exception business practical? Yes, it is. You are at a stage where your program compiles, which only means that the compiler is happy and finds no syntax errors in your Java code. You move on to executing your programs and problems start. Something is not right and the type of **exception** message can help to pinpoint the problem. There are situations where Java gets concerned that you are not aware of potential problems. In such a situation your program code may not even compile because you have not told the compiler how to deal with potential situations. You see Java is different from many other program languages. The majority of program languages display runtime error messages when the program executes in a manner that causes problems. Java goes beyond the simple display of error messages. Programmers can decide at the time when a program is written how to deal with exceptions. A statement is added to the method heading, such as **public void storeData() throws IOException**. Method **storeData** is designed to store important data in an external file. File handling has various opportunities to cause problems during program execution. For instance, what happens if an attempt is made to store data in an external file that does not exist? At the time that the method is written, you need to decide how to handle such a problem. You can use the **throws IOException** statement, which means to Java that you wish to ignore any potential Input/Output (IO) errors. Basically, you are telling Java to *throw away problems*. Another alternative is to *catch* the errors and then provide some process to deal with the errors.

|  |
| --- |
| **APCS Examination Alert** |
| Students are expected to understand the common runtime errors generated by Java, which include:  **NullPointerException**  **IllegalArgumentException**  **ArrayIndexOutOfBoundsException**  **ArithmeticException**  **ClassCastException**  It is not required that students know how to create exception handling code with **throw**, **try** and **catch.** |

**Logic Errors**

We arrive now at the toughest error of them all, the **logic error**. Why are logic errors toughest? The reason is that your computer gives zero help in the logic error department. This baby is strictly your affair and if you are not careful you will get bitten severely. But first things first, what is a logic error?

|  |
| --- |
| **Logic Error Definition** |
| A **logic** error occurs when a program's output is not logically correct, even though the program compiles, and the program executes without crashing. The careful use of well-chosen test data catches logic errors***.*** |

One of the biggest problems with **logic errors** is that many programmers, especially beginning students, do not realize that a logic error exists. How do you know that you have this problem? The answer is proper **TEST DATA**. Consider a simple example first. Your program assignment involves computing the payroll for a number of employees. Every employee works 20 or more hours and earns at least $7.50 per hour. You run your program and look at the results. Some of the employees earn **negative** amounts of money. It does not require a programming genius to figure out that something is wrong. Ironically, there are students who do not catch such errors. Avoid surprises by using the following steps:

|  |
| --- |
| **Program Testing Steps** |
| 1. Test the program first with **easy to tell** test data. For  example, a program that averages student's grades should  start with all grades of 100. Your average has to be 100.  2. Test the program with a set of test data for which the correct  output is known. If you average a set of peculiar numbers,  check the answer first on a calculator.  3. Test the program with a wide variety of data, for every known  path that is possible in the program.  4. Test the program carefully with test data at borderline cases. |

|  |
| --- |
| **Types of Test Data** |
| **Minimal Test Data** is a set of data that checks every possible path in a program, at least once.  **Thorough Test Data** is a set of data that checks every possible path in a program, and checks the border cases as well, at least once. |

**Carburetor Logic**

Every year I meet students who do some really bizarre things to their programs. These students are motivated, they are hardworking and they have good intentions about completing their program assignments. They also recognize that some things are wrong with their programs. It is with logic errors that this unique group of students starts using very interesting logic. All of a sudden previously, correct program segments are gone, only to be replaced with new strange looking code. I call this **Carburetor Logic**. This can be explained by the following conversation between a customer and a mechanic.

Mechanic: *What is wrong with your car?*

Customer: *My car does not start.*

Mechanic: *Let me check. Wow, your carburetor is gone. No wonder*

*your car does not start. What happened to your carburetor?*

Customer: *I removed my carburetor.*

Mechanic: *Why?*

Customer: *My car would not start.*

Mechanic: *Your car cannot start without a carburetor.*

Customer: *My car did not start with the carburetor, so I removed it.*

Please do not ever expose your teacher to *carburetor logic*. Strange things may happen to your teacher. Eye ticks - long under control - return and Vietnam flashbacks - assumed cured - suddenly return.

|  |
| --- |
| **Watch out for the "hidden" logic error** |
| Students often think that a computer with a totally black screen  has “crashed.” Nothing is showing and nothing is happening.  Frequently, the program is stuck in an infinite loop without a  means to exit. This is not a runtime error. The computer is  busily doing what it is told. This is a **logic** error. |

You might wonder why "testing and debugging" was not listed in the steps shown for Object Oriented Program Design. The reason is quite natural. You are expected to test the stages along the way. That is the OOP way. Develop a class and insure that each method in the class works correctly. Develop a second class and now test the proper interaction between these classes. In other words, you do not wait until the program is finished to start testing. At the final program stage, the complexity of the program can be so great that thorough testing has become impossible. On the other hand, a short method can easily be tested to see if the postconditions are satisfied after the method executes. When each method works correctly, you move on and test the entire class. This means that testing occurs in a "bottom-up" approach.

**15.8 Information Hiding**

The *information hiding* topic is not new, but it has not been officially explained until this design chapter. Program design and information hiding go hand in hand. Let us spend a little time and study this very important concept.

|  |
| --- |
| **Information Hiding Definition** |
| **Information Hiding** is the concept of thinking about certain  programming features and using these programming features  without concern, or even knowledge about the implementation  of such features. |

Every single human being functions, every day, in some capacity of information hiding. Consider the following silly story that explains the mechanical details about starting a car.

|  |
| --- |
| *OK, I put the key in the ignition and turn the key. This will complete an electronic circuit and activate the starter motor. As the starter motor turns the engine over, the following process brings the car in motion. Fuel is pumped from the tank to the fuel injectors. The fuel injectors convert the liquid gas to a fine spray and inject this spray inside the chambers of each cylinder. The turning of the engine moves the pistons upward inside the chambers and it compresses the gas into a tight space. With perfect timing the rotor, inside the distributor, completes an electric circuit that sends an electric pulse to the appropriate cylinder. The electric pulse is passed through a spark plug that protrudes inside the cylinder and a small spark is emitted in the chamber. The compressed gas is ignited by the spark and explodes. The explosion causes instant expansion and the piston is driven downward in the cylinder. The piston head is connected to a piston rod which moves up and down. The up and down motion of the piston rod from the repeated explosions is converted to a circular motion. This circular motion is then transferred to the crankshaft. The crankshaft continues the energy path to the transmission. The transmission then selects the appropriate gear for the car movement. From the transmission the turning force goes by drive shaft and various universal joints to the differential. The differential distributes the turning force to the wheels and the car starts to move.* |

Is it necessary for you to know all those car details before you drive a car? No it is not necessary. Furthermore, as much detail as I wrote about car functionality, you can count on many details that are left out. There are details that I never knew, because I am not a mechanical engineer and it simply does not matter. A car is a classic example of an object. It contains attributes and actions, and you can activate the car actions without knowing how such actions are implemented.

|  |
| --- |
| **Reasons for Information Hiding** |
| • There can be valid economic reasons.  • Implementation details may be too complex and  too difficult to understand  • It allows focus on new and different topics without  concern about prior details. |

**There can be valid economic reasons**.

Suppose that you have a large program team working on a new software application. When your product is finished, you eagerly market and sell the product to anybody interested in buying it. What do the customers get? They get an executable file of your software application and all the necessary files to run the program. Your customers do not get any of the source code. The majority of customers would not be interested in the source code and the few people who are interested in the source code could duplicate your product and compete in an unfair manner without much effort. You may argue that a competitor can simply copy your distribution files and sell them as well. That is true, but it can be easily proven that such sales are copies of a copyrighted product. On the other hand, you may have created a really revolutionary approach to solving a certain problem. This new tool can be used in different applications as well. A competitor can use the source code and use it in a different package. In such a scenario it becomes more difficult to prove that the competitor stole your research.

**Implementation details may be too complex and too**

**difficult to understand**.

The **Graphics** class includes a **circle** method. This method is implemented with trigonometric concepts. Do you understand trigonometry? Some students do, and some students do not. There are also rotating three-dimensional cubes that involve mathematical formulas that will challenge all but the most mathematically talented students. The point is that you can use methods very nicely without the knowledge of such complexity. You happen to be writing a program that will display a snowman. You need circles, and trigonometry is not your concern.

**It allows focus on new and different topics without concern about prior details**.

Most students can really understand the first two reasons for information hiding. It is with this third reason that there are problems. You can call it a *letting go* sort of problem. Consider the following situation: You want to create a video game, and this video game requires a large number of small characters that bounce on the screen. After various approaches to creating your video characters, you give up and decide that you need a special graphics editor, or icon editor to assist you in this part of the program.

At this point the focus is completely on the icon editor. You design it, write it, test it and then use it for your graphics game. It is precisely at this point that the fully tested icon editor is used to create some graphics sprites. You are no longer concerned about the implementation of the icon editor. That part is done now and the focus switches.

I am personally completely convinced that many students have difficulty developing large programs because of information hiding. They do not let go and keep too many details floating around at the same time. It is true that some people can handle a much larger quantity of details than other people can. It really does not matter because everybody has a mental focus limit. Information hiding is your friend. It helps you to maintain sanity.

**15.9 The Tetris Game Program Design**

Let us now try and put some of the concepts presented in this chapter together and talk about the design for a program to play Tetris. I am under the belief that this is the most widely known computer game to many age levels. It was first created in Russia in 1986 and it is still popular 20 years later. I am assuming you understand the purpose of the game and have played it at some time. If this is not the case, ask somebody or Google **Tetris** and you will get plenty of practice with different versions of the same game.

**Step 1, Understand The Problem**

This first step appears easy, because I belief that many people understand Tetris and know how to play the game. Keep in mind that this stage includes the problem description and all the requirements of the game. A brief list of requirement concerns is shown in figure 15.12. This chapter section is more concerned with the program design steps beyond the problem description.

**Figure 15.12**

|  |
| --- |
| The game window is 300 pixels wide and 500 pixels high.  There will be seven different Tetris pieces.  Each new piece will be displayed in a separate 100 X 100 pixel window.  Pieces move automatically downward.  Players can move pieces left, right, and rotate.  Players can also "drop" a piece immediately to its final position.  Left/right arrow keys will control pieces left and right.  Up arrow key will rotate pieces.  Down arrow drops piece immediately.  *And many more program specifications.* |

**Step 2, Select Classes**

Frequently, a new program starts with brain storming that is not concerned with sequence, priorities or implementations. One, two or more people can sit around and suggest needed operations and features, such as shown in figure 15.13.

**Figure 15.13**

|  |
| --- |
| Creation of the seven Tetris pieces  Display the game layout where the pieces move  Operations to move the pieces downward and rotate the pieces  Display of the next piece coming down  Mechanism to check if piece fits correctly  Mechanism to see if a solid row needs to be removed  Compute and display player score  Detect keyboard interaction for piece movement and rotation  Check if game is over  Control piece movement speed |

Do you think that all the operations and features were mentioned in figure 15.13? It would be nice if designers could think that clearly. The reality is that program design has a fascinating back-and-forth pattern. It is usually in the reality to implement or to test the program that design flaws show up. For now let us assume that we have a pretty decent list. Remember this is Object Oriented Design and that means that we need classes and we need to establish a relationship between the classes. So what classes shall we use?

How about starting with a **GameWindow** class? We will use the **GameWindow** class to display everything that is happening in our game. In this window the new pieces drop down, rotate, and join the pieces that are puzzled together at the bottom. That sounds pretty good. Let us list some of the **GameWindow** responsibilities in figure 15.14.

**Figure 15.14**

|  |
| --- |
| **GameWindow class Responsibilities** |
| Store piece location for dropped pieces.  Store location of moving piece.  Display dropped pieces.  Display moving piece. |

The most important part of the game revolves around the Tetris pieces. There are usually seven different pieces. One class to handle the operations of the pieces makes good sense. And right now we need to stop and realize something. There has never been a project of any degree of complexity that was planned in a room around a conference table and planned into sequential perfection before the project started. This means that there is no concern with designing a program and then realizing later that you must go back to the first class and make some significant changes. Figure 15.15 lists the **Pieces** class responsibilities.

**Figure 15.15**

|  |
| --- |
| **Pieces Class Responsibilities** |
| Store seven different pieces.  Randomly select the next piece.  Move the piece downward.  Rotate the piece |

One of the problems with Tetris is that pieces must fit neatly together. This is no trivial operation. As the piece moves down it cannot go too far left or too far right. It also needs to rotate and during rotation, it is possible to hit a wall. The trip down is actually the easy part. At the bottom the piece needs to fit perfectly in the growing puzzle of earlier pieces. I have also decided that this class needs to check if any of the pieces form an entire horizontal row that must be removed. We need a class to handle this complexity, and I decide to call the class **ProperFit**, shownin figure 15.16.

**Figure 15.16**

|  |
| --- |
| **ProperFit Class Responsibilities** |
| Check movement during downward fall.  Check if piece can rotate.  Check fit in final position.  Check for complete rows. |

The first three classes are lovely, but nothing seems to glue these classes together into a functional game. There needs to be a group of operations that control everything that happens in the class. This class checks the input from the game player to move the pieces. This class also needs to get the next piece ready for action, make sure that completed rows are removed, the score is updated, and tell the player that the game is over. In short, we need a managing class, which takes charge of all the operations, called **Control**, in figure 15.17.

**Figure 15.17**

|  |
| --- |
| **Control Class Responsibilities** |
| Get game player input.  Get next piece ready.  Remove completed rows.  Update score.  Determine finished game.  Provide instructions. |

It appears that the **Control** class is quite busy, and I make the decision that it makes sense to use a separate class to handle all the game input operations. This is a class that passes all the game player's wishes on to the **Control** class. This will include starting the game, display instructions, move the piece left and right, rotate the piece, and drop the piece down quickly. I will call this the **Player** class and take some complexity away from the **Control** class, in figure 15.18.

**Figure 15.18**

|  |
| --- |
| **Player Class Responsibilities** |
| Get player keyboard input for:  Start of game.  Left/right movements.  Rotating piece.  Rapid vertical drop.  Game instructions. |

**Step 3, Class Interaction**

Now comes a tricky, and very significant, stage in program development. At this point our Tetris game has five classes and these five classes may appear to be separate containers with methods to process data. However, classes are not meant to sit around by themselves; they interact with other classes. After all, you are creating a program for the specific purpose of playing Tetris. Please keep in mind that you will not walk away from this chapter with a fully functional Tetris game. The Tetris game is used to put some specific examples in a pretty dry and theoretical section in the program design chapter.

Class relationships fall into two categories. There is *inheritance* and there is *composition*. Inheritance involves the "is-a" relationship where a sub class **extends** the features of a superclass. We have classes **GameWindow**, **Pieces**, **ProperFit**, **Control** and **Player**. Do you see any inheritance relationship with these five classes? I doubt that you do, but you could decide to use inheritance if it seems advantageous.

For instance, what if you first created a **Piece** class, like the unit class discussed earlier? The **Piece** class stores data to know its (the piece) location in the game window and provides methods for different types of movement. Once this single piece class is created, you can now design other classes that are specialty pieces. There might be an **LPiece**, **SquarePiece**, and other classes for each one of the pieces in the game. I am not saying this is what you must do. This is a chapter on program design and not a chapter explaining the actual design of a Tetris game. This stuff is mostly hypothetical. For now I have decided to keep all the pieces in my **Pieces** class and not use inheritance.

The other relationship, *composition*, is where class interaction becomes complex and requires careful design. Consider the **GameWindow** class, which has the responsibility to store the game information and display the current state. Is this needed in other classes? Any method that moves a piece needs information about the environment. Initially on the way down there is concern about the left and right side of the window. As the piece moves further down and is rotated there must be clearance with the existing pieces.

|  |
| --- |
| **Design Group Discussion** |
| **This chapter may have made you numb. You have been reading and reading and the usual style of learning that revolved around the explanation of program examples is nowhere in sight.**  **Well, you will get a chance to get involved. The different classes and their methods will need to seriously interact with each other. This is an opportunity for you and some other students to work in small teams and examine the suggested classes and methods.**  **Make a list of classes and methods and decide what interaction is necessary and how it should be implemented.** |

**15.10 Summary**

The first and major segment of this chapter discusses program design. Program design is not simply a listing of available methods or program constructs. Program design is very significant in a world where programs reach complexities that were difficult to imagine even a few years ago. Above all else, programs must be reliable for every possible situation.

After reliability, it must be understood that design priorities will involve a tug-of- war between speed, memory and readability. All three of these objectives cannot be maximized simultaneously. A compromise must be reached.

Program design is not possible without very precise and complete specifications that clearly describe the objectives and details of the program. Program difficulties can arise when assumption are made in the descriptions.

In an Object Oriented World, programs revolve around the selection of classes. Individual classes represent programming toolkits, designed for a specific purpose. The class is the toolkit and the methods are the tools.

In most cases classes are designed, such that attributes are private and methods are public. There are some exceptions. Constructors are always public. Attributes are almost always private, but certain data values will never change and may justify public access. The **PI** field in the **Math** class is one such example. Whenever attributes are accessible publicly, they must be declared as a *read-only*, **final** member of a class. Methods are usually public, but there are methods whose only purpose is to assist methods that are used publicly. Such methods are called *helper methods* and should have private access.

Methods need to have clear pre-conditions and post-conditions. The accuracy test of the method is that the implemented method satisfies the post-conditions, provided the pre-conditions are true.

Another major program design issues is the interaction of classes. Relationships between classes can be by inheritance and composition. The benefit of many interacting classes is that individual classes can be tested for reliability. If we assume that all classes in a program work correctly at the class level, then the chances of overall program reliability increase, as long as the class interaction is properly implemented.

Information hiding is intentional in computer programming. It is possible to think about the design of a program, including the classes to use, the purpose of the classes, the functionality of any methods, without discussing the implementation details of any class or method.