**Chapter XIII**

**Boolean Logic & Control Structures**

**Chapter XIII Topics**

13.1 Introduction

13.2 What is a Boolean Statement?

13.3 Boolean Operators

13.4 Truth Tables

13.5 The boolean Data Type

13.6 Nested Selection

13.7 Nested Looping

13.8 Compound Decisions

13.9 Program Input Protection

13.10 Short-Circuiting

13.11 Examining GWCS Critter Class

13.12 The Critter Class act Method

13.13 Summary

**13.1 Introduction**

In this chapter we will investigate **Boolean Logic** concepts. The information in this chapter will be very beneficial in understanding control structures. In an earlier chapter you were introduced to control structures with single, simple conditions. In this chapter you will look at control structures again, but now there will be multiple conditions to consider. Control structures with compound conditions create many complex situations that can easily cause confusion. A good understanding of Boolean Logic will help tremendously in writing program code that is logically correct. On the other hand, a weak - or no understanding - about Boolean principles can cause program problems without a clue why the programs execute incorrectly.

A century ago there was a mathematician, **George Boole**, who took statements and wrote them in a precise format, such that a statement is always **true** or **false**. He founded a branch of mathematics called **Boolean Algebra**. Statements that are either **true** or **false** are called **Boolean** statements. The conditions you used with selection and repetition in the previous chapters were all **Boolean** statements.

This chapter will introduce program statements with control structures that must consider multiple conditions. Computer programs solve human problems and human problems have compound conditions. For instance, your parents may say: *you can go on the spring break skitrip if all your grades are at least B and you keep your room neat.* You might also hear a statement like: *I will only date a guy if he is honest and sensitive*. At a job interview the employer may say: *I only hire people who have a college-degree or five-years job experience*. In each case there were multiple conditions to consider determining the outcome.

|  |
| --- |
| **APCS Examination Alert** |
| The APCS Examination includes a variety of Boolean Logic  questions. Many questions require indirect knowledge of  Boolean Logic, and other questions are directly focused  on testing a student’s understanding of Boolean concepts.  Test results have shown that many students score quite  poorly on this part of the APCS Examination. |

**13.2 What is a Boolean Statement?**

A good starting point is to look at a variety of English sentences and determine if these sentences are Boolean statements or not. So, what are the criteria for a Boolean statement? The sentence, statement, condition, whatever, must be **true** or **false**. Questions, ambiguities, and arguments are not Boolean statements. You can see why this branch of mathematics has a major impact on computer science. The basis of processing data is the binary system of **on** and **off**, which certainly sounds a bunch like **true** or **false**. Each one of the following five statements is a Boolean statement.

*A mile is longer than a kilometer.*

*July and August both have the same number of days.*

*A pound of feathers is lighter than a pound of lead.*

*The Moon is larger than the Sun.*

*New York City has more people than Baltimore.*

The five sentences may not have seemed very Boolean to you. Let us look at the sentences again, translate them into brief logic statements, and indicate whether the statements are true of false. Some Java relational operators are used for the Boolean statements to help clarify the meaning.

|  |  |  |
| --- | --- | --- |
| **English Sentence** | **Boolean Statement** | **T/F** |
| *A mile is longer than a kilometer.* | Mile > Kilometer | True |
| *July and August have the same days.* | JulDays == AugDays | True |
| *A pound of feathers is lighter than a pound of lead.* | PoundF < PoundL | False |
| *The Moon is larger than the Sun.* | MoonSize > SunSize | False |
| *New York City has more people than Baltimore.* | NYPop > BaltPop | True |

Sentences are not always so short and straightforward. Frequently there are multiple conditions in one statement. Special rules need to be followed to determine if the entire statement is true or false. Consider the following sentences with compound conditions.

*She is a computer science teacher and she is a math teacher.*

*The number is odd or the number is even.*

*Enter again if gender is not male or gender is not female.*

*Employment requires a CPA and five years experience.*

The same sentences converted into Boolean statements are:

(She == CSTeacher) and (She == MathTeacher)

(Number % 2 == 1) or (Number % 2 != 1)

(Gender != Male) or (Gender != Female)

(CPA == "Y") and (YrExp >= 5)

These statements are certainly more complex than the earlier examples. Keep in mind that Boolean statements can be made considerably more complicated. It is the intention of this chapter to get a firm grip on Boolean logic.

**13.3 Boolean Operators**

In this section we will look at four different Boolean operators. You know how to handle arithmetic operators ( + - \* / % ). Addition, subtraction, multiplication, division and remainder operations are performed according to the rules for each operator. There are also a set of Boolean operators with their own set of rules. The rules of Boolean operators can be conveniently displayed in a **truth table**. This is a table, which shows the possible combinations of Boolean statements and indicates the value (true or false) of each statement.

In the truth tables that follow, a single letter indicates a single, simple Boolean condition. Such a condition is either **true** or **false**. Boolean statement **A** is true or false. Likewise Boolean statement **B** is true or false. The truth tables will show the results of Boolean statements that use both **A** and **B** with a variety of Boolean operators. Employment requirements will be used to explain the logic of each truth table. In each case imagine that an accountant needs to be hired. Condition **A** determines if the applicant has a **Degree** and condition **B** determines if the applicant has at least five years' experience.

**Boolean Or**

|  |  |  |
| --- | --- | --- |
| **The or Operator** | | |
| A | B | A **or** B |
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

Notice that two conditions have four possible combinations. It is important that you know the results for each type of combination. In this case the employment analogy requires a **Degree OR** **Experience**. This requirement is quite relaxed. You have a Degree, fine. You have Experience, that’s also fine. You have both, definitely fine. You have neither, that’s not fine and causes a problem.

**Boolean And**

|  |  |  |
| --- | --- | --- |
| The **and** Operator | | |
| A | B | A **and** B |
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

Now employment requires a **Degree AND** **Experience**. This requirement is much more demanding than the **or** operator. You have a Degree, fine, provided you also have Experience. If you have only one qualification, or the other qualification, that is not good enough.

**Boolean Xor**

|  |  |  |
| --- | --- | --- |
| The **xor** Operator | | |
| A | B | A **xor** B |
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

Everyday human language, like English, uses both the **or** operator, as well as the **and** operator in communication. How about the “exclusive or” **xor**? The use of **xor** is another story. A peek at the truth table shows something pretty weird. If conditions **A** and **B** are both true, the compound result is false. I will try to explain this by using the “cheap boss” analogy. A manager wants to hire somebody and not pay much money. The advertisement states that a degree or experience is required.

Candidate X walks in and the boss says: “I’ll hire you, but your pay will be low. You see, you have a degree but you have no experience at all.”

Candidate Y walks in and the boss says: “I’ll hire you, but your pay will be low. You see, you have experience but you have no degree.”

Candidate Z walks in and the boss says: “I’m sorry I cannot hire you. You are over qualified since you have a degree and also experience.”

**Boolean Not**

|  |  |  |
| --- | --- | --- |
| The **not** Operator | | |
| A | **not** A |
| T | F |
| F | T |

This section will finish with the simplest Boolean operator, **not**. This operator takes the condition that follows and changes true to false or false to true. There are special rules that need to be followed when a complex Boolean statement is used with not. Such rules will be explained later in the chapter. Right now we want to understand the simple truth table shown above. In English we need to use “double negative” sentences to create an appropriate analogy. I can say “It is **not** true that Tom Smith is valedictorian.” This statement results in Tom not being Valedictorian. On the other hand, if I say “It is **not** true that Tom Smith is **not** the Valedictorian.” Now Tom is the Valedictorian.

**13.4 Truth Tables**

Truth tables provide a convenient way to decide when a Boolean expression is **true**, and if an expression is equivalent to another Boolean expression. The last section introduced simple truth tables to explain the Boolean operators. Now we are going to look at more complex Boolean statements with the help of more complex truth tables. The Boolean statements will not only be more complex, we will also consider a larger number of different conditions. Let us start with the statement (A **and** B) **or** B, shown in Truth Table #1

|  |  |  |  |
| --- | --- | --- | --- |
| **Truth Table #1** | | | |
| A | B | A **and** B | (A **and** B) **or** B |
| T | T | T | T |
| T | F | F | F |
| F | T | F | T |
| F | F | F | F |

Does this truth table tell us anything? It may look just like a bunch of **T**s and **F**s to you, but perhaps you notice something else. The compound Boolean statement of (A **and** B) **or** B has the same truth table as **B**. Forget **A**. The value of A is totally irrelevant. If **B is true** the whole expression is **true**. Likewise, if **B is false** the entire expression is **false**.

That was a pretty good warm up. How about the statement (A **and** B) **or** C? The statement is similar to the previous problem, but now a third Boolean operand is introduced. This third operand suddenly changes the rules considerably. With two operands (A and B) there are four possible combinations to consider. Now that we have three operands the Truth Table #2 will need to consider eight different possibilities.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Truth Table #2** | | | | |
| A | B | C | A **and** B | (A **and** B) **or** C |
| T | T | T | T | T |
| T | T | F | T | T |
| T | F | T | F | T |
| T | F | F | F | F |
| F | T | T | F | T |
| F | T | F | F | F |
| F | F | T | F | T |
| F | F | F | F | F |

Do you feel that any profound observations can be made about the truth table above? Probably not. Consider a similar Boolean statement with altered operators in Truth Table #3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Truth Table #3** | | | | |
| A | B | C | A **or** B | (A **or** B) **and** C |
| T | T | T | T | T |
| T | T | F | T | F |
| T | F | T | T | T |
| T | F | F | T | F |
| F | T | T | T | T |
| F | T | F | T | F |
| F | F | T | F | F |
| F | F | F | F | F |

You might observe that **C** must be true, otherwise the entire statement cannot be true. So what is the point here? The point is somewhat obscure. There are Boolean rules that most people would not expect. In fact, a lot of people find some of the rules pretty weird and do not believe them. With the use of truth tables these rules can be proven. How? Consider the following truth table fact.

|  |
| --- |
| **Truth Table Fact** |
| The truth tables of **equivalent** Boolean expressions are  identical. |

Boolean expressions that use the **not** operator often create the most confusion and mistakes in computer science. In Boolean Algebra the **tilde** ( **~** ) is used for the **not** operator. Consider the following expression:

**~** (A **or** B)

For reasons unknown I have the desire to remove the parentheses and still maintain a Boolean expression with the same value. Armed with high school Algebra, I cleverly use the *distributive property* and create the following:

**~**A **or ~**B

It is just terrific that I used the *distributive property* to make the decision that the Boolean Expression **~** ( A **or** B ) is equivalent to **~**A **or ~**B. This type of logic works fine in **Algebra**, but we are doing **Boolean Algebra**. This is where our trusty truth tables come to our rescue. I will just create a truth table for each one of the expressions and compare the values.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Truth Table #4** | | | | | | |
| A | B | A **or** B | **~**(A **or** B) | **~**A | **~**B | **~**A **or** **~**B |
| T | T | T | >F< | F | F | >F< |
| T | F | T | >F< | F | T | >T< |
| F | T | T | >F< | T | F | >T< |
| F | F | F | >T< | T | T | >T< |

Truth Table #4 shows that the *distributive property* logic of regular Algebra does not apply. The truth tables of the two expressions are not equivalent. The final truth tables have been highlighted with arrows. The table {F F F T} is quite different from the table {F T T T}.

The functionality of truth tables perhaps has been demonstrated to you. But now you are very curious. Just exactly what happens when you remove the parenthesis? Regular Algebra is no help, the truth tables confirmed that. How about considering if the Expression **~** ( A **or** B ) is equivalent to **~**A **and ~**B with Truth Table #5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Truth Table #5** | | | | | | |
| A | B | A **or** B | **~**(A **or** B) | **~**A | **~**B | **~**A **and** **~**B |
| T | T | T | >F< | F | F | >F< |
| T | F | T | >F< | F | T | >F< |
| F | T | T | >F< | T | F | >F< |
| F | F | F | >T< | T | T | >T< |

Can you believe it? The two expressions are equivalent. This is entirely too weird, but the facts are staring you straight in the face. Perhaps you can appreciate now why this chapter is needed. Armed with only the rudimentary truth tables of the previous section, you would not simply conclude what was just proven. Lack of this knowledge has negative consequences on your programs and it does not help your AP Examination score much either.

In Truth Table #6 we take a look at a similar problem that at first may look identical. It is a matter of altering the Boolean operands and checking to see if the expression **~** ( A **and** B ) is equivalent to **~**A **or ~**B.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Truth Table #6** | | | | | | |
| A | B | A **and** B | **~**(A **and** B) | **~**A | **~**B | **~**A **or** **~**B |
| T | T | T | >F< | F | F | >F< |
| T | F | F | >T< | F | T | >T< |
| F | T | F | >T< | T | F | >T< |
| F | F | F | >T< | T | T | >T< |

Once again, the unexpected --- or perhaps by now expected --- expressions are equivalent to each other. You have actually been observing one of the more common and more important laws of Boolean Algebra, called *DeMorgan's Law*.

|  |
| --- |
| **DeMorgan’s Law** |
| **not(A or B) = not A and not B same as ~(A + B) = ~A \* ~B**  **not(A and B) = not A or not B same as ~(A \* B) = ~A + ~B** |

**13.5 The boolean Data Type**

Back in chapter 5 you saw how selection control structures assisted in controlling program flow. The reserved word **if**, combined with a variety of conditions, impacts the outcome of a program. You were told that these conditions were **true** or **false**. Everything that you have learned about computers drums the recurring theme that data processing involves two states. There is **on** and **off**, **1** and **0**, and there is **true** and **false**. Consider program **Java1301.java**, in figure 13.1.

# Figure 10.1

|  |
| --- |
| // Java1301.java  // This program demonstrates that conditional statements have  // true or false Boolean values and can display such values.  public class Java1301  {  public static void main(String args[])  {  System.out.println("\nJAVA1301.JAVA\n");  int x =10;  System.out.println(x == 10);  System.out.println(x == 5);  System.out.println();  }  } |

|  |
| --- |
| Java1001.java Output JAVA1001.JAVA  true  false |

The program output is small, but significant. The condition **(x == 10)** is used in an outputstatement. **x** is initialized to **10** and the first condition is true. The program output displays **true**, and the next line displays **false** for the false statement.

It is helpful in programming logic to test if something is true. You test for correct password entries, objects that are found, values that are equivalent and so on. Whenever a true condition exists, a value of 1 can be assigned in the program to some integer variable. However, programmers prefer greater readability and want to assign **true** and **false**. Modern program languages handle this requirement with a special ***Boolean*** data type that only has two possible values: **true** or **false**.

If you are like most students, you will find a data type with two possible values less than handy. That is OK, a Boolean data type will grow on you and there are many applications where both program logic and program readability are served well with such a simple data type.

Program **Java1302.java**, in figure 13.2, presents a math problem and checks if the answer is correct. The **boolean** variable **correct** is used to assign **true** or **false** based on the entry of the answer. Yes the same result can be achieved without the **boolean** variable, but the program has gained readability and the intention of the source is clearer.

# Figure 13.2

|  |
| --- |
| // Java1302.java  // This program demonstrates that boolean variables add readability to programs.  import java.util.Scanner;  public class Java1302  {  public static void main (String args[])  {  System.out.println("\nJAVA1302.JAVA\n");  Scanner input = new Scanner(System.in);  int gcf;  boolean correct = false;  int attempt = 0;  **while (!correct)**  {  attempt++;  System.out.print("\nWhat is the GCF of 120 and 108? --> ");  gcf = input.nextInt();  **if (gcf == 12)**  **correct = true;**  **else**  **correct = false;**  }  System.out.println("\nAnswered correctly after " + attempt + " Attempt(s).\n");  }  } |

|  |
| --- |
| Java1302.java Output JAVA1302.JAVA  What is the GCF of 120 and 108? --> 1  What is the GCF of 120 and 108? --> 2  What is the GCF of 120 and 108? --> 3  What is the GCF of 120 and 108? --> 12  Answered correctly after 4 Attempt(s).  Process completed. |

Program **Java1303.java**, in figure 13.3 accomplishes the exact same task as the previous program with some simplified code. Note that the **if ... else** statement is gone. In its place is the rather “bizarre” looking **correct = (gcf == 12);**

|  |  |
| --- | --- |
| **if (gcf == 12)**  **correct = true;**  **else**  **correct = false;** | **correct = (gcf == 12);** |

This statement is both proper Java syntax and the preferred way to handle two possible outcomes. Yes, the **if...else** works just fine, but the shorter approach can make excellent sense. For starters, **correct** is a **boolean** type. This means it can only take on the value of **true** or **false**. You found out earlier that conditional statements have a value, which is **true** or **false**. This value can be assigned to a boolean variable. The program output is exactly identical to the output of the previous program and will not be repeated.

**Figure 13.3**

|  |
| --- |
| // Java1303.java  // This program executes in the same manner as Java1002.java.  // The abbreviated Boolean assignment statement is used in place of the  // longer if ... else syntax.  import java.util.Scanner;  public class Java1303  {  public static void main (String args[])  {  System.out.println("\nJAVA1303.JAVA\n");  Scanner input = new Scanner(System.in);  int gcf;  boolean correct = false;  int attempt = 0;    while (!correct)  {  attempt++;  System.out.print("\nWhat is the GCF of 120 and 108? --> ");  gcf = input.nextInt();    **correct = (gcf == 12);**  }  System.out.println("\nAnswered correctly after " + attempt + " Attempt(s).\n");  }  } |

**Figure 13.3 Continued**

|  |
| --- |
| Java1303.java Output JAVA1303.JAVA  What is the GCF of 120 and 108? --> 24  What is the GCF of 120 and 108? --> 1080  What is the GCF of 120 and 108? --> 6  What is the GCF of 120 and 108? --> 12  Answered correctly after 4 Attempt(s).  Process completed. |

**13.6 Nested Selection**

The early *Control Structures*  chapter was so nice and clean. Selection structures and loop structures were uncomplicated. There were no compound conditions, and there certainly was nothing called “nesting.” Nesting a structure actually is not that bad. Essentially, it means that a one control structure is placed inside another control structure.

Program **Java1304.java**, in figure 13.4,is a selection example. In this program example, a student is admitted based on SAT performance. There is also the question of financial aid. For this program, financial aid is determined by family income. However, it is not necessary to consider financial aid unless the student is admitted. The solution is to nest the financial aid condition inside the college admission condition.

When you look closely at the nested control structure examples, you will note that nothing really new is introduced. The syntax and the logic of the conditional statements are the same. Using braces with multiple statements is also the same. The only difference is an appearance of complexity because you see one conditional statement inside another control structure. Anytime that you are bothered by complexity, focus on a smaller, less complex parts. Digest only what you can handle and computer science becomes much easier.

**Figure 13.4**

|  |
| --- |
| // Java1304.java  // This program displays an admission message based on an entered SAT score. It also determines financial  // need with a nested if...else structure.  import java.util.Scanner;  public class Java1304  {  public static void main (String args[])  {  System.out.println("Java1304\n");  Scanner input = new Scanner(System.in);  int sat;  double income;  System.out.print("Enter your SAT score ===>> ");  sat = input.nextInt();  if (sat >= 1100)  {  System.out.println("You are admitted");  System.out.print("Enter your family income ===>> ");  income = input.nextDouble();  if (income <= 20000)  System.out.println("You will receive financial aid");  else  System.out.println("You will not receive financial aid");  }  else  {  System.out.println("You are not admitted");  }  }  } |

|  |
| --- |
| **Java1304.java #1**  Java1304  Enter your SAT score ===>> 1500  You are admitted  Enter your family income ===>> 10000  You will receive financial aid |
| **Java1304.java #2**  Java1304  Enter your SAT score ===>> 1200  You are admitted  Enter your family income ===>> 75000  You will not receive financial aid |
| **Java1304.java #3**  Java1304  Enter your SAT score ===>> 800  You are not admitted |

The next program example uses multiple nested **if...else** statements to control multiple selections. Now **switch** is specifically designed to handle multiple selections. Keep in mind that **switch** has limitations. Consider the situation that is illustrated with the next program example. You need to convert numerical scores to a letter grade. Using **switch** requires a precise value. Now realize that every value in the **90.000 ... 100.000** range becomes an **A**. It is not possible to handle that by using **switch**. If you are willing to use an incredibly large number of cases, you might argue that **switch** works. Even if you are willing to go that route, Java will not accept the use of a real number for a **switch** variable.

The logical course of action is to use the multiple nested **if...else** statements that are shown by program **Java1305.java**, in figure 13.5. Please note the special indentation style that is traditionally used with multiple **if...else** statements. Lining up all the **if** statements helps readability and it is easier to fit the statements on the screen and on paper.

**Figure 13.5**

|  |
| --- |
| // Java1305.java  // This program assigns grades 'A'..'F' based on numerical scores  // using multiple nested if..else statements.  import java.util.Scanner;  public class Java1305  {  public static void main (String args[])  {  System.out.println("Java1305\n");  Scanner input = new Scanner(System.in);  double score;  char grade;  System.out.print("Enter your numerical score ===>> ");  score = input.nextDouble();  if (score >= 90.0)  grade = 'A';  else if (score >= 80.0)  grade = 'B';  else if (score >= 70.0)  grade = 'C';  else if (score >= 60.0)  grade = 'D';  else  grade = 'F';    System.out.println("Your grade will be: " + grade);  }  } |

**Figure 13.5 Continued**

|  |
| --- |
| **Java1305.java Output #1**  Java1305  Enter your numerical score ===>> 100  Your grade will be: A |
| **Java1305.java Output #2**  Java1305  Enter your numerical score ===>> 90.5  Your grade will be: A |
| **Java1305.java Output #3**  Java1305  Enter your numerical score ===>> 83.756  Your grade will be: B |
| **Java1305.java Output #4**  Java1305  Enter your numerical score ===>> 79.999  Your grade will be: C |
| **Java1305.java Output #5**  Java1305  Enter your numerical score ===>> 60  Your grade will be: D |
| **Java1305.java Output #6**  Java1305  Enter your numerical score ===>> 59.999  Your grade will be: F |

**13.7 Nested Looping**

Selection structures are not the only control structures that can be nested. Repetition structures can be nested as well. Two loop structures will be shown with the same type of program. The mission of each program is to display a variety of multiplication tables. It requires one loop to display a multiplication table and it takes another loop to display additional multiplication tables.

Each of the two programs needs an outer loop to control the table size 11, 12 and 13. They also need a nested, inner loop structure to repeat each of the multiplication tables five times. Compare the two program examples that follow in figures 13.6 and 13.7. You will probably find the **for** nested loops the simplest approach for this particular program. The **while** loop requires careful attention to manage the loop-control-variable properly.

**Figure 13.6**

|  |
| --- |
| // Java1306.java  // This program displays several multiplication tables using  // a nested <for> loop structure.  public class Java1306  {  public static void main(String args[])  {  System.out.println("Java1306\n");  for (int table = 11; table <= 13; table++)  {  for (int k = 1; k <= 5; k++)  {  System.out.println(k + " \* " + table + " = " + k \* table);  }  System.out.println();  }  }  } |

|  |
| --- |
| **Java1306.java and Java1307 OUTPUT**  Java1306 & Java1307  1 \* 11 = 11  2 \* 11 = 22  3 \* 11 = 33  4 \* 11 = 44  5 \* 11 = 55  1 \* 12 = 12  2 \* 12 = 24  3 \* 12 = 36  4 \* 12 = 48  5 \* 12 = 60  1 \* 13 = 13  2 \* 13 = 26  3 \* 13 = 39  4 \* 13 = 52  5 \* 13 = 65 |

**Figure 13.7**

|  |
| --- |
| // Java1307.java  // This program displays several multiplication tables using  // nested pre-condition <while> loop structures.  public class Java1307  {  public static void main(String args[])  {  System.out.println("Java1307\n");  int k = 1;  int table = 11;  while (table <= 13)  {  while (k <= 5)  {  System.out.println(k + " \* " + table + " = " + k \* table);  k++;  }  System.out.println();  k = 1;  table++;  }  }  } |

**13.8 Compound Conditions**

You are about to use the Boolean logic that was presented earlier in this chapter. Many situations in life present compound conditions and we will start by looking at two programs that decide if a person should be hired. The hiring-criteria are based on years of education as well as years of work experience.

The first program example, **Java1308.java**, in figure 13.8, uses a logical **or** to decide if somebody should be hired. In this case somebody is qualified if they have the required education **or** they have the required work experience. Java uses two vertical lines || located above the <Enter> key to indicate the logical **or**.

**Figure 13.8**

|  |
| --- |
| // Java13.8.java  // This program demonstrates compound decisions with the logical or ( || ) operator.  import java.util.Scanner;  public class Java1308  {  public static void main (String args[])  {  System.out.println("Java1308\n");  Scanner input = new Scanner(System.in);  int education; // years of education  int experience; // years of work experience  System.out.print("Enter years of education ===>> ");  education = input.nextInt();  System.out.print("Enter years of experience ===>> ");  experience = input.nextInt();  if ((education >= 16) || (experience >= 5))  System.out.println("You are hired");  else  System.out.println("You are not qualified");  }  } |

|  |
| --- |
| **Java1308.java Output #1**  Java1308  Enter years of education ===>> 17  Enter years of experience ===>> 8  You are hired |
| **Java1308.java Output #2**  Java1308  Enter years of education ===>> 17  Enter years of experience ===>> 2  You are hired |
| **Java1308.java Output #3**  Java1308  Enter years of education ===>> 13  Enter years of experience ===>> 8  You are hired |
| **Java1308.java Output #4**  Java1308  Enter years of education ===>> 13  Enter years of experience ===>> 2  You are not qualified |

The four output samples of **Java1308.java** showed three hire situations and one “not qualified” response. These four outputs closely resemble the appearance of the truth tables.

The next example, program **Java1309.java**, in figure 13.9**,** is almost identical to the previous program. The only difference is that now the compound statement is decided by a logical **and**. Java uses two ampersands **&&** to indicate a logical **and.** Do not ask why such odd symbols are used for **or** and **and** operators.

The logical **and** is far more demanding. There are four possible hiring combinations, and just like the truth table, only one will qualify. A person must have the proper education and also the required work experience.

**Figure 13.9**

|  |
| --- |
| // Java1309.java  // This program demonstrates compound decisions with the logical and ( && ) operator.  import java.util.Scanner;  public class Java1309  {  public static void main (String args[])  {  System.out.println("Java1309\n");  Scanner input = new Scanner(System.in);  int education; // years of education  int experience; // years of work experience  System.out.print("Enter years of education ===>> ");  education = input.nextInt();  System.out.print("Enter years of experience ===>> ");  experience = input.nextInt();  if ((education >= 16) && (experience >= 5))  System.out.println("You are hired");  else  System.out.println("You are not qualified");  }  } |

|  |
| --- |
| **Java1309.java Output #1**  Java1309  Enter years of education ===>> 17  Enter years of experience ===>> 8  You are hired |
| **Java1309.java Output #2**  Java1309  Enter years of education ===>> 17  Enter years of experience ===>> 2  You are not qualified |

**Figure 11.16 Continued**

|  |
| --- |
| **Java1309.java Output #3**  Java1309  Enter years of education ===>> 13  Enter years of experience ===>> 8  You are not qualified |
| **Java1309.java Output #4**  Java1309  Enter years of education ===>> 13  Enter years of experience ===>> 2  You are not qualified |

|  |
| --- |
| **Logical Operators in Java** |
| Java uses || to indicate a logical **or**.  Java uses **&&** to indicate a logical **and**.  Java uses **!** to indicate a logical **not**.  Java uses **!=** to indicate a logical **xor**. |

**13.9 Program Input Protection**

One of the most common Boolean laws used in introductory computer science is **DeMorgan’s Law**. Let us start by looking at program **Java1310.java**, in figure 13.10, which shows one approach that uses Boolean logic properly. The purpose of the program is to make sure that proper input is entered from the keyboard. This program example could be part of any type of survey or record that asks the program user to enter his or her gender. The program is expected to prevent any entry, except one of the characters **M** or **F**.

The compound decision first considers what the proper input should be. Then a set of parentheses is placed around the desired **M or F** condition. The **not** decides if the condition **is not** the desired one. Check the program out and notice the combination of the **or** and the **not**.

You might notice a peculiar **charAt(0)** stuck at the end of the **nextLine** method. Appending the **charAt(0)** in this manner allows the **nextLine** method to enter a single character.

**Figure 13.10**

|  |
| --- |
| // Java1310.java  // This program demonstrates compound decision with a do...while loop.  // The program checks for proper data entry.  // The addition of charAt(0) allows the nextLine method to enter a single character.  import java.util.Scanner;  public class Java1310  {  public static void main (String args[])  {  System.out.println("Java1310\n");  Scanner input = new Scanner(System.in);  char gender;  do  {  System.out.print("Enter your Gender [M/F] ===>> ");  **gender = input.nextLine().charAt(0);**  }  **while ( !(gender == 'M' || gender == 'F') );**  System.out.println("Your gender is " + gender);  }  } |

|  |
| --- |
| **Java1310.java Output**  Java1310  Enter your Gender [M/F] ===>> Q  Enter your Gender [M/F] ===>> t  Enter your Gender [M/F] ===>> 1  Enter your Gender [M/F] ===>> f  Enter your Gender [M/F] ===>> m  Enter your Gender [M/F] ===>> F  Your gender is F |

Program **Java1311.java**, in figure 13.11 is very similar to the previous program. The programs are almost identical, but at the same time significantly different. A very important parenthesis is in the wrong place. The result is that the **not** only applies to the **gender == 'M'** component of the condition. This results in a logic error. This type of logic error alters the input protection so that only **M** is accepted as keyboard input. Does it make sense why **M** is the only correct input? Put on your best logic hat and see if you can figure out what is happening.

**Figure 13.11**

|  |
| --- |
| // Java1311.java  // This program demonstrates compound decision with a do...while loop.  // The program does not work properly because of misplaced parentheses.  import java.util.Scanner;  public class Java1311  {  public static void main (String args[])  {  System.out.println("Java1311\n");  Scanner input = new Scanner(System.in);  char gender;  do  {  System.out.print("Enter your Gender [M/F] ===>> ");  gender = input.nextLine().charAt(0);  }  **while ( !(gender == 'M') || gender == 'F' );**  System.out.println("Your gender is " + gender);  }  } |

|  |
| --- |
| **Java1311.java Output**  Java1311  Enter your Gender [M/F] ===>> D  Enter your Gender [M/F] ===>> Q  Enter your Gender [M/F] ===>> F  Enter your Gender [M/F] ===>> M  Your gender is M |

The logical **or** makes a compound statement true if either one of the two conditions is true. With the right parenthesis in the wrong location the condition **gender == 'F'** is not controlled by the **not**. This means that entering **F** makes the condition **true** and the loop body is executed. This is not desirable because the loop body asks to enter **gender** again. In other words, **F** will not stop the loop. Entering **M** does stop the loop. The **not** does apply to the (**gender == 'M'**) condition and the loop works properly for that portion of the overall statement.

Program example **Java1312.java**, in figure 13.12 provides correct input protection with the aid of Boolean's *DeMorgan's Law.* Just remember that whenever the **not** is distributed you need to alter the logical operator.

**Figure 13.12**

|  |
| --- |
| // Java1312.java  // This program demonstrates correct use of negative compound  // decision structure using DeMorgan's Law.  import java.util.Scanner;  public class Java1312  {  public static void main (String args[])  {  System.out.println("Java1312\n");  Scanner input = new Scanner(System.in);  char gender;  do  {  System.out.print("Enter your Gender [M/F] ===>> ");  gender = input.nextLine().charAt(0);  }  **while (gender != 'M' && gender != 'F');**  System.out.println("Your gender is " + gender);  }  } |

|  |
| --- |
| **Java1312.java Output #1**  Java1312  Enter your Gender [M/F] ===>> Q  Enter your Gender [M/F] ===>> W  Enter your Gender [M/F] ===>> M  Your gender is M |
| **Java1312.java Output #2**  Java1312  Enter your Gender [M/F] ===>> F  Your gender is F |

The **boolean** data type exists for the purpose of readability. In the area of variety, **boolean** is seriously lacking. There are only two values for **boolean**, and that is **true** and **false**. Yet with those two small data values we can create programs that are much nicer because they create such excellent readability.

The next program example uses the **boolean** data type. In this case the loop continues while the condition is **!correct**. Now that is very readable. Inside the loop there is also a statement that says **if (!correct)** and then a message follows. The beauty of the **boolean** data type is that we can make our programs closer to English. This is not important the day that you write the program. But next week, next month, and especially next year you will not always remember the meaning of some program code. At such times you will appreciate any extra readability that you added to your program source code. The outputs of the previous gender input programs may not have been very satisfying to you. In each program the lower-case characters **m** and **f** were rejected. Program **Java1313.java**, in figure 13.13, not only uses a nice **boolean** data type, but also makes the conditional statement longer to accept lower-case input.

**Figure 13.13**

|  |
| --- |
| // Java1313.java  // This program accepts upper-case as well as lower-case.  // Gender input for [M/F] by using multiple conditional statements.  import java.util.Scanner;  public class Java1313  {  public static void main (String args[])  {  System.out.println("Java1313\n");  Scanner input = new Scanner(System.in);  char gender;  boolean correct;  do  {  System.out.print("Enter your Gender [M/F] ===>> ");  gender = input.nextLine().charAt(0);  **correct = (gender == 'M' || gender == 'F' || gender == 'm' || gender == 'f');**  if (!correct)  System.out.println("Incorrect input; please re-enter");  }  while (!correct);  System.out.println();  System.out.println("Your gender is " + gender);  }  } |

|  |
| --- |
| **Java1313.java Output #1**  Java1313  Enter your Gender [M/F] ===>> q  Incorrect input; please re-enter  Enter your Gender [M/F] ===>> Q  Incorrect input; please re-enter  Enter your Gender [M/F] ===>> 1  Incorrect input; please re-enter  Enter your Gender [M/F] ===>> M  Your gender is M |

**Figure 13.13 Continued**

|  |
| --- |
| **Java1313.java Output #2**  Java1313  Enter your Gender [M/F] ===>> F  Your gender is F |
| **Java1313.java Output #3**  Java1313  Enter your Gender [M/F] ===>> m  Your gender is m |
| **Java1313.java Output #4**  Java1313  Enter your Gender [M/F] ===>> f  Your gender is f |

We will finish this section by looking at a practical situation that occurs in real life. Individuals entering a password to login to a network, or people entering a *PIN* to use an ATM card should not just be able to enter data forever. Somebody who has stolen an ATM card, or somebody who has no business logging in as a different user, should not have unlimited access trying to enter the system.

There have been a variety of movies where clever young kids hook up a computer program to some database and the computer takes all night methodically trying to crack into the system by trying millions of different passwords. Now this looks cute on television or in a movie, but do give me a break. Anybody who uses a program that sits around waiting for millions of break-in attempts deserves to get trouble. There has to be a limit on input attempts.

Our final program in this section shows just such a situation. It is also another nice example of using a compound condition. Program **Java1314.java**, in figure 13.14, shows part of a program that checks the proper **PIN** (Personal Identification Number) of an **ATM** (Automatic Teller Machine) card. The compound condition has been set up to accept three attempts at entering the correct PIN. After three tries it is over. In such a case the machine can simply reject the ATM, but it can also do fun things like keep the card.

**Figure 13.14**

|  |
| --- |
| // Java1314.java  // This program shows the need for a practical compound condition  // used with an input protection loop.  // The program requests the user PIN, but rejects access after three tries.  import java.util.Scanner;  public class Java1314  {  public static void main (String args[])  {  System.out.println("Java1314\n");  Scanner input = new Scanner(System.in);  String pin;  int tries = 0;  do  {  System.out.print("Enter your PIN ===>> ");  pin = input.nextLine();  tries++;  }  while (!pin.equals("8423") && (tries < 3));  if (pin.equals("8423"))  System.out.println("Your PIN is accepted");  else  System.out.println("You have exceeded your PIN entries");  }  } |

|  |
| --- |
| **Java1314.java Output #1**  Java1314  Please enter your PIN ===>> 4325  Please enter your PIN ===>> 4326  Please enter your PIN ===>> 4327  You have exceeded your PIN entries |
| **Java1314.java Output #2**  Java1314  Please enter your PIN ===>> 4325  Please enter your PIN ===>> 4326  Please enter your PIN ===>> 8423  Your PIN is accepted |

|  |
| --- |
| **do . . . while and Input Protection** |
| You will see **do..while** used frequently for input protection  loops. The post-condition loop makes sense for checking  erroneous input because you want the program to enter the  loop body at least one time. |

**13.10 Short-Circuiting Conditions**

Java is pretty clever and takes advantage of something called *short-circuiting* with compound conditions. The name is very appropriate if you think why we use the term *short circuit* with electricity. Electric current takes a path that travels to some appliance like a freezer, television or a light bulb. The path traveled by the electricity through the wires and the appliance is called a circuit. The appliance will only operate properly is a *complete* circuit exists. Sometimes, wires may be faulty and insulation is missing that causes a *short circuit.* Current jumps across the wires and a circuit is completed before the appliance is reached. In such a case a fuse burns or a circuit breaker is switched off to prevent fire damage from the overheated circuits.

The concept applies to computer science decisions as well. A situation exists that stops the normal path in favor of a shorter path. Consider the following, rather complicated, Boolean expression:

A and ( (A or B) and (B or C) and (A or C) or ((B and C) or (A and B)))

This expression can be quite an unpleasant truth table. Now suppose that it is known that **A equals false**. Is it necessary to evaluate the entire expression? No, it is not, because the **and** logical operator guarantees that the final outcome will always be false. In this case we can *short-circuit* the evaluation. Similar logic applies with **or**. If it is given that **A equals true** then the entire expression will evaluate to true. Once again it is not necessary to consider the rest of the expression.

A or ( (A or B) and (B or C) and (A or C) or ((B and C) or (A and B)))

The next two program examples will not only involve compound decisions, but they will also demonstrate that short-circuiting is performed by Java. Each program call uses a method, called **isEven**, shown in figure 13.15. This method is called from the conditional statement.

Calling a method inside a conditional statement is perfectly legal since it is a return method. Return methods return values. You have seen return methods that return integers and real numbers. Method **isEven** returns **true** or **false**, since it is a boolean return method.

The purpose of the method is to return **true** if the method argument **number** is even and **false** if **number** is odd. The method intentionally displays some output so that you can see when isEven is called. Using this method will provide evidence about Java's *short-circuiting* habits.

**Figure 13.15**

|  |
| --- |
| **public static boolean isEven(int Number)**  **{**  **System.out.println();**  **System.out.println("Calling isEven Method");**  **System.out.println();**  **if (Number % 2 == 0)**  **return true;**  **else**  **return false;**  **}** |

Program **Java1315.java**, in figure 13.16, requests two integers from the keyboard, The two entered numbers are used in the parameters of the **isEven** method calls. Now Java claims that it will not consider evaluating a complete compound condition if **true** or **false** can be guaranteed without moving on. In other words, if the first condition is **true** and a logical **or** is used, then the whole condition will be **true**. We can test this theory by running the program several times and seeing if one or both methods are called.

**Figure 13.16**

|  |
| --- |
| // Java1315.java  // This program uses "short circuiting" and uses the isEven  // method to demonstrate short circuiting with logical or.  import java.util.Scanner;  public class Java1315  {  public static void main (String args[])  {  System.out.println("Java1315\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter number 1 ===>> ");  int n1 = input.nextInt();  System.out.print("Enter number 2 ===>> ");  int n2 = input.nextInt();  if (isEven(n1) || isEven(n2))  System.out.println("At least one number is even.");  else  System.out.println("Both numbers are odd.");  }  public static boolean isEven(int number)  {  System.out.println();  System.out.println("Calling isEven Method");  System.out.println();  if (number % 2 == 0)  return true;  else  return false;  }  } |

**Figure 13.16 Continued**

|  |
| --- |
| **Java1315.java Output #1**  Java1315  Enter number 1 ===>> 12  Enter number 2 ===>> 24  Calling IsEven Method  At least one number is even**.** |
| **Java1315.java Output #2**  Java1315  Enter number 1 ===>> 12  Enter number 2 ===>> 15  Calling isEven Method  At least one number is even**.** |
| **Java1315.java Output #3**  Java1315  Enter number 1 ===>> 15  Enter number 2 ===>> 31  Calling isEven Method  Calling isEven Method  Both numbers are odd. |

The output of program **Java1315.java** does not disappoint us. As anticipated, only the first part of the compound condition, **(isEven(N1) || isEven(N2))** is evaluated when it is established that the whole condition must be true. This is proven by the single call to the **isEven** method when the first number entered is an even number.

Program example **Java1316.java**, in figure 13.17, is very similar to the previous program, and it uses the same **isEven** method. This time short-circuiting is tested with a logical **and**. The concept of short-circuiting is the same, but in this case Java stops evaluating when it is established that the first condition is false.

You will find in future chapters that short-circuiting becomes a useful tool in preventing incorrect program executions. In particular, short circuiting with the logical **and** will be used frequently. It is appropriate to introduce the concept in a chapter that introduces compound conditions that are evaluated with Boolean logic laws.

**Figure 13.17**

|  |
| --- |
| // Java1316.java  // This program uses "short circuiting" and uses the isEven  // method to demonstrate short circuiting with logical and.  import java.util.Scanner;  public class Java1316  {  public static void main (String args[])  {  System.out.println("Java1316\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter number 1 ===>> ");  int n1 = input.nextInt();  System.out.print("Enter number 2 ===>> ");  int n2 = input.nextInt();  if (isEven(n1) && isEven(n2))  System.out.println("Both numbers are even.");  else  System.out.println("At least one number is odd.");  }  public static boolean isEven(int number)  {  System.out.println();  System.out.println("Calling IsEven Method");  System.out.println();  if (number % 2 == 0)  return true;  else  return false;  }  } |

|  |
| --- |
| **Java1316.java Output #1**  Java1316  Enter number 1 ===>> 12  Enter number 2 ===>> 24  Calling isEven Method  Calling isEven Method  Both numbers are even. |

**Figure 13.17 Continued**

|  |
| --- |
| **Java1316.java Output #2**  Java1316  Enter number 1 ===>> 12  Enter number 2 ===>> 25  Calling isEven Method  Calling isEven Method  At least one number is odd**.** |
| **Java1316.java Output #3**  Java1316  Enter number 1 ===>> 15  Enter number 2 ===>> 31  Calling isEven Method  At least one number is odd**.** |

**13.11 Examining GWCS Critter Class**

This chapter will make a considerable jump into new GridWorld material. You have now seen and used a large number of GridWorld classes. There are the management classes such as **Location** and **Actor**. These classes are used in many GridWorld operations. You have neither seen nor are you expected to make any changes in these fundamental classes. You did see a variety of **Actor** subclasses and each one of the subclasses re-defined or newly-defined one or more of the **Actor** methods.

The biggest change with the new class is that **Critter** objects interact with other objects. This is new. Our previous classes all are pretty tame. **Rock** objects do absolutely nothing, but sit still and become a hindrance to the movement of other objects. **Actor** objects stay in one cell, nervously making 180 turns.

**Flower** objects also stay put and quietly wilt away turning darker with each execution step. **Bug** objects show some excitement as they steadily move forward, dropping flowers, and only turning when other objects block the way. The exception is a flower, which seems not to be an obstacle for a bug.

Pretty much all these **Actor** objects mind their own business. **Bugs** do trample on flowers, but perhaps they feel entitled as bugs drop the flowers on the grid in the first place. You will now see a class that seriously interacts with other objects in a variety of ways.

You can only appreciate the differences between the critter class and the previous classes by observing the execution of a program that includes **Critter** objects. The **Critter** will show totally new behavior and it will become the superclass for various other critter classes, much as the **Actor** class is the superclass for the earlier subclasses. As you observe the behavior of the objects make sure that you do not use **Run**, but rather use **Step** to slowly observe the changes with each execution.

|  |
| --- |
| **Lab Experiment Java1317**  **Examining the Critter Class** |

|  |
| --- |
| **01. Create a GridWorld Project with Folder Java1317**  **Create a Java1317 project.**  **Make sure to attach the GridWorld library.** |

|  |
| --- |
| **02. Compile and Execute the Java1317 Project**  **Compile and execute the project.**  You should see a grid with 5 bugs, 5 rocks and 2 critters.  You already know the **Rock** and **Bug** objects.  The new blue icons, with curly tails, are **Critter** objects.    Click **Step** and observe the program execution.  Multiple steps will be shown here as well, but remember that  you will have a different starting screen. |

|  |
| --- |
| **03. Observe Object Behavior**  **Click Step seven times.**  Observe each step carefully.  Compare the current step with the previous step.  Watch the pictures closely and also watch your own executions.  The bugs, rocks and flowers behave as you expect. |

|  |  |
| --- | --- |
| **Step 1** | **Step 2** |
| **Step 3** | **Step 4** |

|  |
| --- |
| If you watch closely you will see that objects disappear. Not all objects, but some disappear. There were five bugs in **Step 4** and then there are only four bugs in **Step 5**. **Step 50** shows the GridWorld after the objects have acted 50 times. |

|  |  |
| --- | --- |
| **Step 5** | **Step 6** |
| **Step 7** | **Step 50** |

|  |
| --- |
| **Critter Observation Questions**  Ask yourself the following questions after observing the new Critters objects act from step to step:   * What objects disappear? * Is there a pattern? Can you predict who will disappear next? * How do critters move? * Why do some bugs adjacent to a Critter not get removed? * Are you ready to make general statements about critters? |

**12.8 The Critter Class act Method**

The **Critter** class **extends** the **Actor** class and as all good subclasses in the GridWorld program, Critter**s** re-define the **act** method. This re-definition is not very subtle or simple. Make sure that you have your GridWorld *Quick Reference Guide* handy as you look at the program code that follows.

Usually, you will see a slow, steady sequence that builds up to a conclusion. This time the conclusion will come first. Our thoughts are that you understand the program code better. You have already seen the program execution with several **Critter** objects engaging other objects. The short version of **Critter** behavior can be summed up with three rather simple statements. Digest those three statements first and then look at the **Critter** class declaration in Figure 13.18.

|  |
| --- |
| **Fundamental Critter Behavior Defined in the act Method** |
| * **Get an array of neighbors** * **Process each member in this neighbor array** * **Move to a new location** |

**Figure 13.18**

|  |
| --- |
| public class Critter extends Actor  {  public void act()  {  if (getGrid() == null)  return;  ArrayList<Actor> actors = getActors();  processActors(actors);  ArrayList<Location> moveLocs = getMoveLocations();  Location loc = selectMoveLocation(moveLocs);  makeMove(loc);  }  public ArrayList<Actor> getActors()  {  return getGrid().getNeighbors(getLocation());  }  public void processActors(ArrayList<Actor> actors)  {  for (Actor a : actors)  {  if (!(a instanceof Rock) && !(a instanceof Critter))  a.removeSelfFromGrid();  }  }  public ArrayList<Location> getMoveLocations()  {  return getGrid().getEmptyAdjacentLocations(getLocation());  }  public Location selectMoveLocation(ArrayList<Location> locs)  {  int n = locs.size();  if (n == 0)  return getLocation();  int r = (int) (Math.random() \* n);  return locs.get(r);  }  public void makeMove(Location loc)  {  if (loc == null)  removeSelfFromGrid();  else  moveTo(loc);  }  } |

Perhaps you feel somewhat confused. You might have expected a re-definition of the **act** method only. You now see a pretty good size class with a bunch of foreign methods. Actually, the **Critter** class shows a format that is similar to the **Bug** class. The **Bug** class re-defines **act**, but this new **act** method requires the new definitions of the **canMove**, **turn** and **move** methods. With the explanation of the **Bug** class, it became necessary to put the **act** method on hold and then we returned after the new methods were introduced. The same approach will be used with the **Critter** class.

Figure 13.19 isolates the **act** method. It is more complex than the **Bug act** method. The **Bug** classuses three new methods. The new methods each had to be studied, but they were pretty simple. The **Critter** class also uses a set of new methods that are re-defined in the **Critter** class.

**Figure 13.19**

|  |
| --- |
| **public void act()**  **{**  **if (getGrid() == null)**  **return;**  **ArrayList<Actor> actors = getActors();**  **processActors(actors);**  **ArrayList<Location> moveLocs = getMoveLocations();**  **Location loc = selectMoveLocation(moveLocs);**  **makeMove(loc);**  **}** |

Everything will be explained systematically, which requires taking one or more statements and explaining the details for just those statements. This process will start with the first program statement shown and explained in Figure 13.20.

**Figure 13.20**

|  |
| --- |
| **if (getGrid() == null)**  **return;** |
| This first statement has nothing to with how a **Critter** object acts. Its mission is to provide protection in the event that the object is not part of a grid.  Method **getGrid** is an **Actor** method, which returns the grid of this actor or **null** if this actor is not in a grid. How can an actor exist without being part of a grid? That will make sense by looking at the three lines below that are used in the **main** method of some GridWorld program. |
| **ActorWorld world = new ActorWorld(); // Line 1**  **Bug busyBee = new Bug(); // Line 2**  **world.add(busyBee); // Line 3** |
| Line 1 creates a new grid. There are no **Actor** objects constructed yet.  Line 2 constructs a new **Bug**, called **busyBee**. The **Bug** object exists, but not as part of any grid. Method **getGrid** will return **null** at this stage.  Line 3 adds **busyBee** to the current grid and acting can now proceed. |

We now get to the actual **act** part of the critters. It was mentioned earlier that critters *get Neighbors, process Neighbors* and *move to a new location****.*** Figures 13.21, 13.22 and 13.23 provide explanations of the three stages of critter acting.

**Figure 13.21 Get the Neighbors**

|  |
| --- |
| **ArrayList<Actor> actors = getActors();** |
| This statement indicates that method **getActors** will return an array of **Actor** members to the **actor** object. Method **getActors** is a **Critter** class method shown in the next table cell. |
| **public ArrayList<Actor> getActors()**  **{**  **return getGrid().getNeighbors(getLocation());**  **}** |
| You do know that **getGrid** returns the current grid of an object.  Methods **getNeighbors** and **getLocation** can be found in the quick reference guide and should be part of your memory by the time you take the AP exam.  **getLocation** returns the grid location of this actor. **getNeighbors** returns an **ArrayList** of objects in the eight compass directions of the actor location. This means that **getActors** can store up to eight actors in the **actors** **ArrayList** object. |

It may seem quite tedious to go to an **act** method, which uses some newly-defined methods and then those new methods in turn use other methods that must be looked up in the quick reference guide. The tedious part goes away when you work more with the GWCS. With each chapter, with each experiment and with each lab assignment you gain more familiarity and you will reach a point where much of what is now very confusing becomes quite simple to you.

|  |
| --- |
| **GridWorld Case Study Reality Check** |
| **Make sure that you tell yourself multiple times that the GridWorld Case Study counts 3/16 of the APCS Exam.** |

**Figure 13.22 Process the Neighbors**

|  |
| --- |
| **processActors(actors);** |
| The method name is self-documenting, but you know little beyond the fact that critters will process neighbors in some fashion. Method **processActors** below contains the details of the method, which are not really so complex. |
| **public void processActors(ArrayList<Actor> actors)**  **{**  **for (Actor a : actors)**  **{**  **if (!(a instanceof Rock) && !(a instanceof Critter))**  **a.removeSelfFromGrid();**  **}**  **}** |
| The **for..each** loop visits each member of the **actors** parameter and causes an individual array member to be moved from the grid depending on some criteria.  A compound condition checks to see if the individual actor is a **Rock** object or if it is a **Critter** object. This means that critters do not remove any rocks or any critters. All other members in the **actors** array are no longer on the grid. |

|  |
| --- |
| **The instanceof Operator** |
| **instanceof** is NOT a method.  **instanceof** is actually a *relational operator* that can be used to compare objects and classes.  The statement: **if (barry instanceof Bug)**  checks if **barry** is a **Bug** object.  In technical terms, we are checking if the object **barry** is an instance of the **Bug** class.  Here are some examples:   |  |  | | --- | --- | | **Object Creation & Condition** | **Condition Evaluates to** | | **Bug barry = new Bug();**  **if (barry instanceof Bug)** | **true** | | **Rock rocky = new Rock();**  **if (rocky instanceof Bug)** | **false** | | **Actor bill = new Actor();**  **if (bill instanceof Actor)** | **true** | | **Bug barry = new Bug();**  **if (barry instanceof Actor)** | **true** |   The reason **barry** is an instance of an **Actor** is that **barry** is an instance of a **Bug** and a **Bug** *is-an* **Actor**. Remember *inheritance*. |

Please do not get confused with the last stage of the **act** method. In stage 3 critters move to a new location. You will be pleased to know that the third stage has three separate parts of its own. The critter has to find possible locations for moving, select one of those locations, and then make the move. It sounds easy enough, but it takes three methods and a fair amount of Java program code to accomplish this important mission.

**Figure 13.23 Move to a New Location**

|  |
| --- |
| **ArrayList<Location> moveLocs = getMoveLocations();**  **Location loc = selectMoveLocation(moveLocs);**  **makeMove(loc);** |
| The first statement creates an array of locations. The locations are provided by the **getMoveLocations**, a new method defined for the **Critter** class. This method is shown below. |
| **public ArrayList<Location> getMoveLocations()**  **{**  **return getGrid().getEmptyAdjacentLocations(getLocation());**  **}** |
| Method **getMoveLocations** uses the **Grid** method **getEmptyAdjacentLocations**, which returns an array of empty cell locations in the eight compass directions around the current object location. |
| **public Location selectMoveLocation(ArrayList<Location> locs)**  **{**  **int n = locs.size();**  **if (n == 0)**  **return getLocation();**  **int r = (int) (Math.random() \* n);**  **return locs.get(r);**  **}** |
| Now that an array of locations is ready, method **SelectMoveLocation** executes next. First the size of the **locs** array is determined. If the size equals zero, it means that there is no place to move. The location that is returned is the current location, otherwise a random integer is generated in the range of the possible array index numbers. |
| **public void makeMove(Location loc)**  **{**  **if (loc == null)**  **removeSelfFromGrid();**  **else**  **moveTo(loc);**  **}** |
| The last step is for the critter to move. If the new location is **null**, the critter has problems and removes itself, otherwise it moves to the newly selected location. |

|  |
| --- |
| **Critter Summary of Behavior** |
| **OK, you heard all the details of all the many methods used by the Critter class. Now look at the short summary.** |
| **1. Create an array of neighbor actors in the 8 compass directions around the critter.**  **2. Process the array of actors by removing all the**  **actors from the grid who are not a rock or a critter.**  **3. Move to a new location.**  **a. Create an array of 8 compass directions empty cell locations.**  **b. Pick one of empty locations at random.**  **c. Move to the random location.** |

**13.13 Summary**

This chapter on Boolean Logic did not present any real new control structures. The same decision and repetition structures were looked at a second time and placed under a more realistic light. Realistic in the sense that real-life programs involve many situations, which are not very simple. Control structures can be nested inside each other in a rather complex manner, and conditions are rarely simple and straightforward, but frequently compounded.

Compound conditions can easily lead to unintentional logic problems in program execution. You need to use Boolean logic to create correct compound conditions.

Boolean Logic provides laws to deal with compound conditions. There are four boolean operators, and, or, xor and not. Each of the logical operators can be placed in a truth table to determine the outcome of a compound condition.

The GridWorld case study is an excellent example of class interaction both with inheritance and with composition. The **Actor** class becomes a container class and has three attributes that are objects of another contained class. The **BoundedGrid** class has a static two-dimensional array that stores all the object on the grid.

This chapter introduced a new GridWorld subclass of the **Actor** class, called the **Critter** class. **Critter** objects are different from previous **Actor** objects in that they interact with other GridWorld objects. The **act** method of the **Critter** class has three distinct stages. A **Critter** objects acts by *getting its neighbors*, *processing its neighbors* and then *moving to a new location*.

The neighbors of a **Critter** are the occupants of all the adjacent cells in the eight compass directions. The **Critter** processes the neighbors are removing all neighbors from the grid that are not a **Rock** or another **Critter**. Finally, the **Critter** moves to a random location, which is an empty cell in eight possible adjacent c compass directions.