**Chapter VIII**

**Focus on OOP, Encapsulation**

**Chapter VIII Topics**

8.1 Introduction

8.2 Creating a New Class

8.3 Restricting Attribute Access

8.4 "Get" and "Set" Methods

8.5 Constructor Methods

8.6 The Cube Class Case Study

8.7 The Consequences of Scope

8.8 Method Summary

8.9 GridWorld’s main Method

**8.1 Introduction**

*Object Oriented Programming* will not be casually mentioned in this chapter. OOP is the front-center topic right now and the intention is to cover some very important OOP ground. Right now you have a variety of basic Java tools under your belt and you are ready to learn about this very fascinating, modern approach to programming.

Now keep in mind that the single most important goal for any program is *reliability*. Speed is great, user-friendly is requested, cheap is desirable, but nothing matters if the software does not work properly. First and foremost a program, or program module, must perform its intended task correctly, the first time, the last time and every time in between. There used to be a time in the early days of programming that the simplicity of the programs and the genius of the computer scientists kept reliability pretty much afloat with hard work and an organized approach to programming. In particular, programming became popular by dividing a program into manageable blocks, and concentrating on completing and testing the smaller blocks. Names like *structured programming*, *modular programming*, *procedural programming*, became important buzzwords and programming tools.

Keep in mind that early programs only produced text output. You should know text output, because many program examples have produced text output so far. With the arrival of the *Graphical User Interface* (*GUI*) the world of programming changed very dramatically, and program complexity has been growing exponentially ever since. A new style of programming came on the scene slowly, at first, but rapidly gained momentum and brought about a whole new way of thinking about programming. Now it matters little if impressive vocabulary is tossed around. The bottom line is still *reliability*, and as you gain more computer science knowledge you will clearly see how Object Oriented Programming in general and Java in particular do an outstanding job to make programs more reliable.

In an earlier chapter you were told that the three corner stones of OOP are *encapsulation*, *class interaction*, and *polymorphism*. Some brief introduction to encapsulation happened earlier during the previous two chapters. This chapter will continue to explore encapsulation and pick up the pace dramatically.

Programs involve data and the modules that access and process the data. The problem in the past has been that the *data*and the *modules*processing the data, called *procedures*and *functions*, were separated. This approach caused serious reliability problems. As programs grew in size so did the likelihood that some *procedure*or *function* accessed data unintentionally and made unwanted and often disastrous changes to data values. A new style of thinking helped prevent these kinds of problems. This style of thinking revolves around *objects*. Objects store data and objects also store the modules - called *methods*in Java - which process the data. So far the object seems no different from programs that combine data storage and procedures that process the stored data. What makes objects unique is that every object has its own set of methods, which are unique to the object and the object stores the data that is appropriate for the object. Furthermore, special precautions are designed into the object to insure that data is properly initialized and accessed. As a matter of fact, a proper object will not allow anything outside the object to access its data directly for fear of potential problems. A well-designed object carries its own methods that safely handle the object data.

So what does this all mean? This means you have *encapsulation*, which is a fancy, but very appropriate word. The data and the methods that act upon the data are all contained in one package or capsule, called an object. Multiple objects can be created for multiple purposes. You can also think of this package as a capsule, a container, and hence the word *encapsulation*.

Now we must get clear on the vocabulary used with objects. You have heard talk about actions, subroutines, processes, procedures, functions, and methods. All these words have been used intentionally, because you will probably find every term used somewhere in computer science. Each one of these words implies some program module that performs a task, frequently involving data, but not always. However, some task is always involved that must be accomplished.

Life would be simpler if computer scientists could just make up their minds about the naming of these program modules. FORTRAN and BASIC started with *subroutines*, Pascal followed by having both *procedures* and *functions*, and C++ wanted every module to be called a *function*. You will be pleased to know that the current fashion is to use the word *method*, and Java error messages will be using that terminology. From here on, *method*will be used. Keep the following object vocabulary in mind.

|  |
| --- |
| **Objects, Variables and Methods** |
| Java encapsulates data and action modules that access  the data in one container, called an object.  Object members that perform some task are called *methods*.  Object members that store data are called *attributes*. |

**8.2 Creating a New Class**

Java uses many containers. There are *class*, *methods* *loop* control structure containers and *selection* structure containers. Containers have a heading followed by a set of curly braces **{ }** , which indicate the start and the end of the container.

This chapter is the first *focus* on Object Orient Programming (*OOP*) and this first focus is on *encapsulation*. The introduction stated that the goal of encapsulation is to *encapsulate* data and all the methods that access the data in the same container or capsule. This capsule in Java is officially called a *class*.

It has been stated in earlier chapters that a *class is a toolkit* and a *method is a tool*. Now it is does not matter if a class is called a container, a capsule, or a toolkit, because the creation of a class will need to start by first creating a container, a capsule or a toolkit. A case study follows that will start very simply and steadily increase in complexity as the class become more and more complete.

There are two case studies formats. The first type presents a completed program. An execution of the completed program is demonstrated followed by an investigation of every component in the program. Explanations are given about the design of the program and the decisions that were made to arrive at a final solution that is presented.

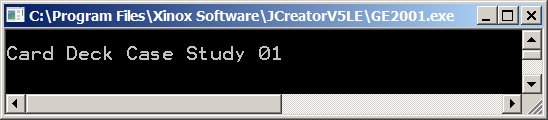
The second style of case study starts with a minimal program. Step after small step the program grows and an explanation is given with each small improvement along the way. There are good arguments for each style of case study and those arguments are not significant right now. The **CardDeck** case study that follows will use the second case study style.

Program **Java0801.java**, in figure 8.1,brings little excitement, but it sets the stage. The main method instantiates a **CardDeck** object, but nothing can be done with this object. However, the program compiles and creates a simple skeleton for the start of our case study.

Note how the **CardDeck** class is not declared **public**. Only one class can be declared **public** and that is the class with the same name as the file name. Proper program design suggests that each class resides in its own file. In this textbook, and especially with case studies, multiple classes are frequently combined in one file for clarity. Our main focus is on the development of the **CardDeck** class, but this growing class needs to be tested at each stage by an outside class that creates an object of the **CardDeck** class.

**Figure 8.1**

|  |
| --- |
| // Java0801.java  // CardDeck Case Study #01  // This shows a minimal class declaration.  // This class has no practical value, but it compiles and executes.  public class Java0801  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 01\n");  CardDeck d = new CardDeck();  System.out.println();  }  }  **class CardDeck**  **{**  **}** |



In the earlier chapter the program output displayed the actual black/white text box that is shown by the program execution. That will change in this chapter. The black/white box is exactly what is shown by JCreator with certain settings, but not necessarily by other IDEs. The graphics screen capture also rapidly adds to the size of the file. From now on all program text output will be placed in a table like the one shown below.

|  |
| --- |
| **Java0801.java Output**  Card Deck Case Study 01 |

**8.3 Restricting Attribute Access**

What type of data needs to be stored in the **CardDeck** class? For this chapter it does not matter since it is not necessary to make the list complete. The purpose of the case study is to present an introduction to Object Oriented Programming, and specifically, the process of *encapsulation*. It is not the goal to make a fully functional **CardDeck** class, but the hope is that the information presented will make it possible to create such a class, if needed. For now there will only be four attributes. The first attribute, **cardGame**, stores the name of the card game. Then the **numDecks**  data field stores the number of card decks, **numPlayers s**tores the number of players in the game, and **cardsLeft** stores the number of cards left in the deck(s). Program **Java0802.java** in figure 8.2 adds these variables, but nothing is done yet besides placing the variables in the correct location. Right now there is not any type of data access at this stage.

**Figure 8.13**

|  |
| --- |
| // Java0802.java  // CardDeck Case Study #02  // Variables, called attributes or data fields, are added to the <CardDeck> class.  public class Java0802  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 02\n");  System.out.println();  CardDeck d = new CardDeck();  System.out.println();  }  }  class CardDeck  {  String cardGame; // name of the card game  int numDecks; // number of decks in a game  int numplayers; // number of players in a game  int cardsLeft; // number of cards left in the deck(s)  } |

|  |
| --- |
| **Java0802.java Output**  Card Deck Case Study 02 |

Simple data types store single values. This may a single integer, a single character, a single decimal number or a single boolean. It is not necessary to indicate which member of the variable is being used. There is only one member. It is not an issue. You learned with the **Math** class that both the class identifier and the method or field identifier must the used, like **Math.sqrt(100)** or **Math.PI**. If object methods are members of a class then an object must be constructed and the object name is used with the method identifier, like **tom.move(next)**. In this case **tom** is an object of the **Bug** class of the GridWorld case study.

Using this same data access approach this means that if **d** is an object of the **CardDeck** class, and **numDecks** is a field identifier in the class, then you would use syntax like **d.numDecks** to access a field inside the object.

This is precisely what is done by program **Java0803.java** in figure 8.3. First, values are assigned to the four **CardDeck** variables and then those values are displayed. You will find that the program compiles just nicely and it also gives the desired, correct output.

**Figure 8.14**

|  |
| --- |
| // Java0803.java  // CardDeck Case Study #03  // <CardDeck> variables are accessed directly by the <main> method.  // This program violates encapsulation, even though it compiles, and executes.  // This approach greatly compromises program reliability.  public class Java0803  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 03\n");  CardDeck d = new CardDeck();  d.cardGame = "Poker";  d.numDecks = 1;  d.numPlayers = 5;  d.cardsLeft = 208;  System.out.println("Name of Card Game: " + d.cardGame);  System.out.println("Number of Decks: " + d.numDecks);  System.out.println("Number of Players: " + d.numPlayers);  System.out.println("Number of Cards Left: " + d.cardsLeft);  System.out.println();  }  }  class CardDeck  {  String cardGame;  int numDecks;  int numPlayers;  int cardsLeft;  } |

**Figure 8.14 Continued**

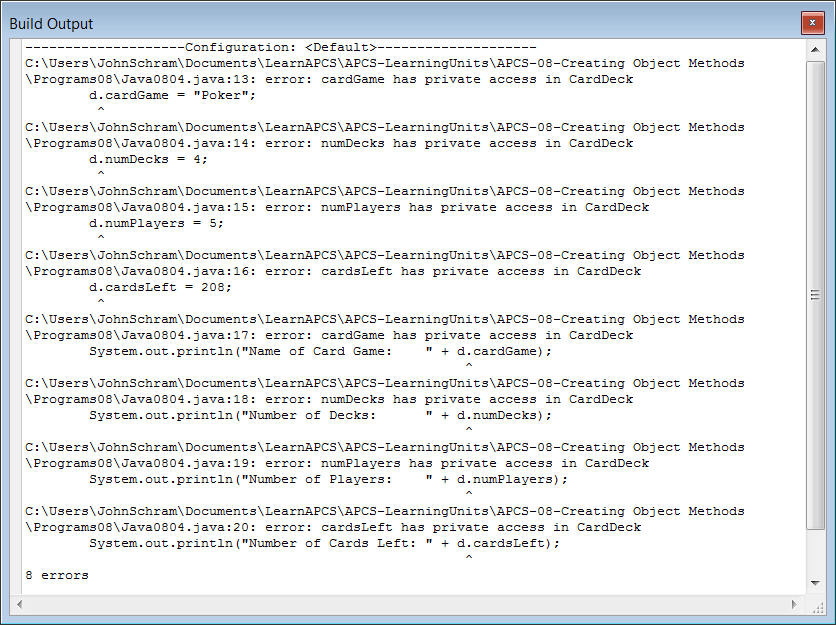
|  |
| --- |
| **Java0803.java Output**  Card Deck Case Study 03  Name of Card Game: Poker  Number of Decks: 1  Number of Players: 5  Number of Cards Left: 208 |

Compiling and nice output are lovely, but it is not sufficient. We committed a serious offense and the *OOP Police* will probably give us several tickets. The essence of *encapsulation*is to package the data and the methods that access the data in one container. However, it is not sufficient to package data and methods together; you also need to insure that only a select number of methods can access the data. Accessing the data directly, from outside the class, is a very serious crime in *OOP Land*. This type of access can create serious unwanted data access and this will be more likely as programs are larger. The next stage, shown by program **Java0804.java**, in figure 8.4, shows how to insure against that problem by declaring the data attributes as **private**.

**Figure 8.4**

|  |
| --- |
| // Java0804.java  // CardDeck Case Study #04  // All the variables in the <CardDeck> class are now declared as private access.  // This prevents improper, public access to the data variables.  public class Java0804  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 04\n");  CardDeck d = new CardDeck();  d.cardGame = "Poker";  d.numDecks = 4;  d.numPlayers = 5;  d.cardsLeft = 208;  System.out.println("Name of Card Game: " + d.cardGame);  System.out.println("Number of Decks: " + d.numDecks);  System.out.println("Number of Players: " + d.numPlayers);  System.out.println("Number of Cards Left: " + d.cardsLeft);  System.out.println();  }  }  class CardDeck  {  private String cardGame;  private int numDecks;  private int numPlayers;  private int cardsLeft;  } |

**Figure 8.4 Continued**



The next section will explain how data can be properly accessed. There may be a few exceptions, but in most cases declare attributes **private**. You have seen some exception, such as **Math.PI** and **Math.E** and in both cases the data fields store constant values that are never meant to change.

|  |
| --- |
| Private and Public Members |
| Members in a class need to be declared as **private** or **public**.  **Private** members can only be accessed by program statements inside the class.  Data attributes usually need to be declared **private**.  **Public** members can be accessed by any program statement, both inside and outside the class.  Methods usually need to be declared **public**. |

**8.4 "Get" and "Set" Methods**

Consider a typical bank. The bank has a lobby that is open to the public. From the lobby, customers can go to tellers to make deposits, withdrawals and loan payments. There are also lobby offices where bank officers take loan applications, handle investments and handle customer concerns. Every area that is mentioned is open to the public.

The bank also has areas that are open only to the bank employees. These *private* areas include the teller area, which contains the cash drawers and containers that store all the daily check deposits. There is the very private bank vault where large deposits of cash are stored and there is also the private bathroom that is meant for bank employees only.

These physical *private* and *public* areas may seem a good analogy to the **private**and **public**access that is used with the design of a new class. The bank has more and that is not physical, but electronic and it also involves levels of access. Every bank has a web site. The home page of the bank is very public and draws attention to many services available to the existing customer or potentially new customer. At this area different bank accounts are explained, as well as available loans and credit cards.

Existing customers can go further and enter a *private access* personal account. Once proper user id and password are entered, a customer can see bank account transactions, pay bills, make transfers and apply for a loan or credit card. Perhaps it is in this area that we truly see the virtue of private access. All bank accounts have private access. It does not make any sense to let the general public have access to all the many customer bank accounts.

If private access is restricted then it has no practical value. The whole point in combining public local methods with private local attributes is to control data access. In the case of a bank the entry of a *UserID* and a *Password* gains access. Frequently, access is even stricter to protect possible fraud and customers are shown a special picture or phrase that insures a customer that the proper bank web site is in fact being used.

Access to a customer's account is not limited to the customer. A teller can look at the account to determine if the balance is large enough for the requested withdrawal. A bank officer uses access to both the private account of the bank itself and a customer to transfer money from the bank account to the customer account after a loan is approved. In every example stated you see the private attribute, a bank account, and an accessing method, like the bank teller or bank officer, which handles the private account in a proper manner to insure reliability and proper privacy.

Program **Java0805.java**, in figure 8.5 , shows the next stage of the **CardDeck** class, which will include four methods to return the values of the four data attributes. These four methods *get* information for the some program segment outside the class. By convention such methods are called "get methods" and frequently **get** is part of the method identifier, as you see in method **getDecks**. Do not expect some clever program code that insures proper data access as was explained for a customer's bank account. The program examples in this chapter are concerned with explaining the reasoning for encapsulation and how to create proper classes.

**Figure 8.5**

|  |
| --- |
| // Java0805.java  // CardDeck Case Study #05  // The <CardDeck> class now has four "get" methods to return  // the data values of <CardDeck> objects.  // Note that Java assigns initial values to object data.  public class Java0805  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 05\n");  CardDeck d = new CardDeck();  System.out.println("Name of Card Game: " + d.getGame());  System.out.println("Number of Decks: " + d.getDecks());  System.out.println("Number of Players: " + d.getPlayers());  System.out.println("Number of Cards Left: " + d.getCards());  System.out.println();  }  }  class CardDeck  {  private String cardGame;  private int numDecks;  private int numPlayers;  private int cardsLeft;  public String getGame()  {  return cardGame;  }  public int getDecks()  {  return numDecks;  }  public int getPlayers()  {  return numPlayers;  }  public int getCards()  {  return cardsLeft;  }  } |

**Figure 8.5 continued**

|  |
| --- |
| **Java0805.java Output**  Card Deck Case Study 05  Name of Card Game: null  Number of Decks: 0  Number of Players: 0  Number of Cards Left: 0 |

There is something curious about stage 06. There are four methods used to return the values stored by the **CardDeck** object, but there is no evidence that any values were ever stored to any of these four data fields. This is intentional. It shows that Java assigns *default* values when a new object is instantiated. Any of the **int** variables is assigned value **0**.

**Zero** or **0** makes perfect sense, but what is **null** for **cardGame**? A confession must be made here. Java has an interesting data type, called **String**. It is possible to create **String** variables and use **String** variables as if they were simple data types just like **int** and **boolean**. The reality is that **String** is a class, complete with its own set of methods. **null** is not the same thing as *zero*. *Zero*is a value. We state that water freezes at *0 degrees Celsius*. On the other hand, **null** means *no value*. Any object that is not assigned an initial value will return value **null**.

Altering the values of object instance variables is the job of *set methods*. These methods are parameter methods that normally change the value of a single attribute. Like the *get* methods, *set* methods frequently use the word **set** in the method identifier like, **setDecks**. Program **Java0806.java**, in figure 8.6, shows the improved **CardDeck** class with three segments. There are *data attributes, get methods* and *set methods*.

By convention many programmers use complete, self-commenting identifiers for the data fields of the class and then use abbreviations of those names as the formal parameters in the headings of the methods.

You will also note that the earlier get methods are shown in a compact style where each method is written on one line. This style saves space and it is often done for short methods that only have one program statement. It was not done for set methods, because they are emphasized in this section.

**Figure 8.6**

|  |
| --- |
| // Java0806.java  // CardDeck Case Study #06  // The <CardDeck> class adds four "set" methods to alter the data attributes of <CardDeck> objects.  public class Java0806  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 06\n");  CardDeck d = new CardDeck();  d.setGame("Bridge");  d.setDecks(1);  d.setPlayers(4);  d.setCards(52);  System.out.println("Name of Card Game: " + d.getGame());  System.out.println("Number of Decks: " + d.getDecks());  System.out.println("Number of Players: " + d.getPlayers());  System.out.println("Number of Cards Left: " + d.getCards());  }  }  class CardDeck  {  // Data attributes  private String cardGame;  private int numDecks;  private int numPlayers;  private int cardsLeft;  // Get return Methods  public String getGame() { return cardGame; }  public int getDecks() { return numDecks; }  public int getPlayers() { return numPlayers; }  public int getCards() { return cardsLeft; }  // Set void Methods  public void setGame(String cG)  {  cardGame = cG;  }  public void setDecks(int nD)  {  numDecks = nD;  }    public void setPlayers(int nP)  {  numPlayers = nP;  }  public void setCards(int cL)  {  cardsLeft = cL;  }  } |

|  |
| --- |
| **Java0806.java Output**  Card Deck Case Study 06  Name of Card Game: Bridge  Number of Decks: 1  Number of Players: 4  Number of Cards Left: 52 |

**8.5 Constructor Methods**

It is easier to demonstrate the creation of get methods and set methods, but in the real world of designing classes and writing program, you need to create the constructor methods after the data is selected. The job of the constructor is to initialize the attributes of an object. The constructor is automatically called by the **new** operator and increases reliability by assigning correct starting values to the data. Program **Java0807.java**,in figure 8.07, shows the improvement created by adding a constructor to the program.

**Figure 8.16**

|  |
| --- |
| // Java0807.java  // CardDeck Case Study #07  // This <CardDeck> class uses a constructor to initialize variables  // during the instantiation of a new <CardDeck> object.  // This is an example of increasing reliability by an automatic constructor call.  public class Java0807  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 07\n");  CardDeck d = new CardDeck();  System.out.println("Name of Card Game: " + d.getGame());  System.out.println("Number of Decks: " + d.getDecks());  System.out.println("Number of Players: " + d.getPlayers());  System.out.println("Number of Cards Left: " + d.getCards());  System.out.println();  }  }  class CardDeck  {  private String cardGame;  private int numDecks;  private int numPlayers;  private int cardsLeft;  // Constructor  public CardDeck()  {  cardGame = null;  numDecks = 1;  numPlayers = 1;  cardsLeft = 52;  }  public String getGame() { return cardGame; }  public int getDecks() { return numDecks; }  public int getPlayers() { return numPlayers; }  public int getCards() { return cardsLeft; }  public void setGame(String cG) { cardGame = cG; }  public void setDecks(int nD) { numDecks = nD; }  public void setPlayers(int nP) { numPlayers = nP; }  public void setCards(int cL) { cardsLeft = cL; }  } |

**Figure 8.7 Continued**

|  |
| --- |
| **Java0807.java Output**  Card Deck Case Study 07  Name of Card Game: null  Number of Decks: 1  Number of Players: 4  Number of Cards Left: 52 |

It is easy to think that constructors exist for the purpose of initializing object data. Constructors certainly fill that role, but it is not the only role. In more general terms the constructor prepares the new object to fulfill its duty as best as possible. This starts with correct initial values, but there is more. Think card games now. Is it not very important that a deck is shuffled before any game begins? Any computer game that involves cards needs some means to shuffle the decks before they are used. Shuffling seems like a good candidate for a new method, and it is also a terrific candidate for improving the constructor. The constructor is called automatically, and it is capable of calling other methods in the class. By including a call to a **shuffling** method, the constructor insures that any newly constructed **CardDeck** object is shuffled and ready to go.

The **shuffling** method will also be used to illustrate another point. So far you have seen that all the **CardDeck** data variables are declared **private**, and all the **CardDeck** methods are declared **public**. It is easy to draw the conclusion that data is always **private** and methods are always **public**. It is true that most data is declared **private** and most of the methods are declared **public**, but it is not required as you will see in the next program.

Classes include *helper* methods. Helper methods are not meant to perform a procedure that is used by some other class. A helper method exists for the purpose of assisting some other method in the same class. The **shuffling** method is one example of such a helper method. Your lab assignment has an excellent example of a method that is only needed within the class. For the lab assignment you will work with a **Rational** class. The purpose of the rational class is to store and manipulate rational numbers. Rational numbers include whole numbers and fractions. One important capability of the **Rational** class is the ability to reduce fractions, and this can be done with the Greatest Common Factor (*GCF*). The **private**, helper method **computeGCF**, assists the **reduce** method, but it is not needed anywhere else. Program **Java0808.java**, in figure 8.8 demonstrates the use of a private method in a class.

**Figure 8.8**

|  |
| --- |
| // Java0808.java  // CardDeck Case Study #08  // This program adds the <shuffleCards> method, which is a <private> helper method  // used by the <CardDeck> constructor.  public class Java0808  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 08\n");  CardDeck d = new CardDeck();  System.out.println("Name of Card Game: " + d.getGame());  System.out.println("Number of Decks: " + d.getDecks());  System.out.println("Number of Players: " + d.getPlayers());  System.out.println("Number of Cards Left:" + d.getCards());  System.out.println();  }  }  class CardDeck  {  private String cardGame;  private int numDecks;  private int numPlayers;  private int cardsLeft;  public CardDeck()  {  cardGame = "Poker";  numDecks = 1;  numPlayers = 4;  cardsLeft = 52;  shuffleCards();  }  private void shuffleCards()  {  System.out.println("Shuffling Cards");  }  public String getGame() { return cardGame; }  public int getDecks() { return numDecks; }  public int getPlayers() { return numPlayers; }  public int getCards() { return cardsLeft; }  public void setGame(String cG) { cardGame = cG; }  public void setDecks(int nD) { numDecks = nD; }  public void setPlayers(int nP) { numPlayers = nP; }  public void setCards(int cL) { cardsLeft = cL; }  } |

**Figure 8.8 Continued**

|  |
| --- |
| **Java0808.java Output**  Card Deck Case Study 08  Shuffling Cards  Name of Card Game: Poker  Number of Decks: 1  Number of Players: 4  Number of Cards Left: 52 |

Every **CardDeck** object starts its life the same boring way. No card game is decided and then you get *1 Deck, 1 player* and *52 CardsLeft*. There are many card games with more than one player and there are also card games with more than one deck in the game. You have seen overloaded methods before and this is a situation where overloaded methods are very useful. It is allowed, and strongly encouraged, to create multiple constructors in a class. You have a *default*constructor, which is good when no parameters are specified for the new object. After the default constructor you can declare extra constructors for a variety of different situations. In our case it will create a **CardDeck** object with a specified number of **numDecks** and **numPlayers**. **cardsLeft** will not need to be specified because that value is computed as a function of the value of **numDecks**. Program **Java0809.java**, in figure 8.9, shows the two constructors in one class.

**Figure 8.9**

|  |
| --- |
| // Java0809.java  // CardDeck Case Study #09  // A second, overloaded constructor, method is added to the program.  // It is now possible to specify card deck details during instantiation.  public class Java0809  {  public static void main(String args[])  {  System.out.println("\nCard Deck Case Study 09\n");  CardDeck d1 = new CardDeck();  CardDeck d2 = new CardDeck("BlackJack",4,5);  System.out.println();  System.out.println("Name of Card Game: " + d1.getGame());  System.out.println("Number of Decks: " + d1.getDecks());  System.out.println("Number of Players: " + d1.getPlayers());  System.out.println("Number of Cards Left: " + d1.getCards());  System.out.println();  System.out.println("Name of Card Game: " + d2.getGame());  System.out.println("Number of Decks: " + d2.getDecks());  System.out.println("Number of Players: " + d2.getPlayers());  System.out.println("Number of Cards Left " + d2.getCards());  System.out.println();  }  }  class CardDeck  {  private String cardGame;  private int numDecks;  private int numPlayers;  private int cardsLeft;  public CardDeck()  {  System.out.println("Default Constructor");  cardGame = "Poker";  numDecks = 1;  numPlayers = 4;  cardsLeft = 52;  shuffleCards();  }  public CardDeck(String cG, int nD, int nP)  {  System.out.println("Overloaded Constructor");  cardGame = cG;  numDecks = nD;  numPlayers = nP;  cardsLeft = nD \* 52;  shuffleCards();  }  private void shuffleCards()  {  System.out.println("Shuffling Cards");  }  public String getGame() { return cardGame; }  public int getDecks() { return numDecks; }  public int getPlayers() { return numPlayers; }  public int getCards() { return cardsLeft; }  public void setGame(String cG) { cardGame = cG; }  public void setDecks(int nD) { numDecks = nD; }  public void setPlayers(int nP) { numPlayers = nP; }  public void setCards(int cL) { cardsLeft = cL; }  } |

|  |
| --- |
| **Java0809.java Output**  Card Deck Case Study 09  Default Constructor  Shuffling Cards  Overloaded Constructor  Shuffling Cards  Name of Card Game: Poker  Number of Decks: 1  Number of Players: 4  Number of Cards Left: 52  Name of Card Game: Blackjack  Number of Decks: 4  Number of Players: 5  Number of Cards Left: 208 |
| **Instantiation and Construction** |
| A **class** is a template that can form many objects.  An **object** is a single variable **instance** of a class.  **Objects** are sometimes called **instances**.  An **object** is created with the **new** operator. The creation  of a new object is called:  **instantiation** of an object  **construction** of an object  The special method that is called during the instantiation of  an object is called a **constructor**. |

|  |
| --- |
| **Constructor Method Notes** |
| Constructors are methods, which are called during the  instantiation of an object with the new operator.  The primary purpose of a constructor is to initialize all the  attributes of newly created object.  Constructors have the same identifier as the class.  Constructors are neither **void** methods nor **return** methods.  Constructors can be *overloaded* methods.  The method identifier can be the same, but the method  *signature* must be different.  A constructor with no parameters is called a *default constructor*. |

**8.6 The Cube Class Case Study**

The title of this textbook is *Exposure Java*. The content of this chapter is so significant that you will get more than your share of exposure. You will be pleased to know that this chapter has a second case study. The intention is that you have a solid OOP foundation in the area of *encapsulation* when you leave this chapter. The second case study involves manipulation of a cube. A cube is a three-dimensional object and it may be useful to have a special class to assist with some graphics programs that require displaying cubes.

**Cube Case Study, Stage #1**

Program **Java0810.java**, in figure 8.10, starts with a small **Cube** class. This program does not have any output yet. The class has a constructor, which initializes the top-left coordinate of the cube. If you execute the program, you will see an applet window, but the window lacks excitement. Nothing will show.

**Figure 8.10**

|  |
| --- |
| // Java0810.java  // Cube Casestudy #1  // Stage #1 presents a <Cube> class with a default constructor.  // This program does not display a cube.  // The Cube Case Study uses applets. Run the html file to execute.  import java.awt.\*;  import java.applet.\*;  public class Java0810 extends Applet  {  public void paint(Graphics g)  {  Cube cube = new Cube(g);  }  }  class Cube  {  private int tlX; // topleft X coordinate of the Cube's position  private int tlY; // topleft y coordinate of the Cube's position  public Cube(Graphics g)  {  tlX = 50;  tlY = 50;  }  } |

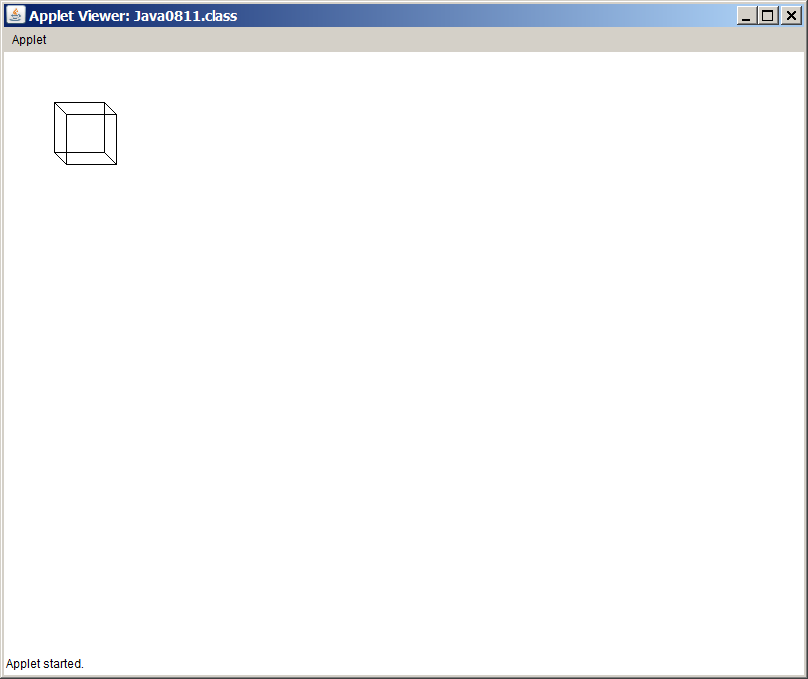
**Cube Case Study, Stage #2**

The constructor initializes the starting coordinates of a cube objects at [50,50]. Is there any magic in those numbers? No, there is not, but some values need to be selected and [50,50] will do nicely. Program **Java0811.java**, in figure 8.11, alters the constructor and includes a call to the new **draw** method. Method **draw** draws a cube at the current value of **tlX** and **tlY**. The result is a single cube.

**Figure 8.11**

|  |
| --- |
| // Java0811.java  // Cube Casestudy #2  // Stage #2 adds a <draw> method to display one cube object.  import java.awt.\*;  import java.applet.\*;  public class Java0811 extends Applet  {  public void paint(Graphics g)  {  Cube cube = new Cube(g);  cube.draw(g);  }  }  class Cube  {  private int tlX; // topleft X coordinate of the Cube's position  private int tlY; // topleft y coordinate of the Cube's position  public Cube(Graphics g)  {  tlX = 50;  tlY = 50;  }  public void draw(Graphics g)  {  int tlX2 = tlX + 12;  int tlY2 = tlY + 12;  g.setColor(Color.black);  g.drawRect(tlX,tlY,50,50);  g.drawRect(tlX2,tlY2,50,50);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+50,tlY,tlX2+50,tlY2);  g.drawLine(tlX,tlY+50,tlX2,tlY2+50);  g.drawLine(tlX+50,tlY+50,tlX2+50,tlY2+50);  }  } |

**Figure 8.11 Continued**

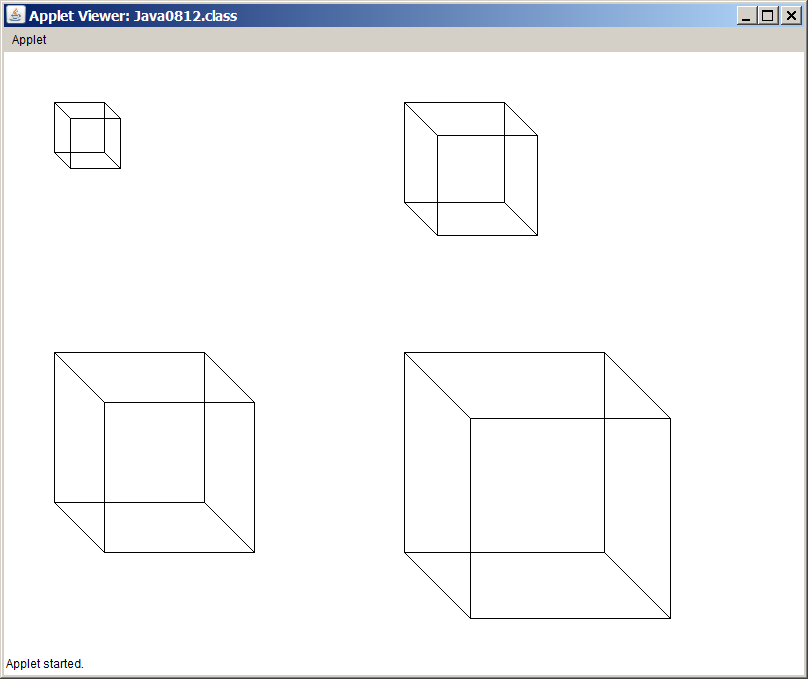


**Cube Case Study, Stage #3**

Program **Java0812.java**, in figure 8.12, adds a second constructor, which can specify the initial coordinate values of the cube object as well as its size. This third stage is not only a matter of adding a second constructor. The **draw** method now has issues. It has many *50s* throughout because every cube it draws is **50** pixels wide and **50** pixels tall. Now the method needs to be more general and handle cubes of any size. The consequence is a new improved **draw** method capable of drawing cubes not only at any location, but also any size.

**Figure 8.12**

|  |  |
| --- | --- |
| // Java0812.java// Cube Casestudy #3// Stage #3 adds a second, overloaded constructor.  // It is now possible to specify the size and the location of the cube.  // The <draw> method needs to be altered to handle different cube sizes. | |
| class Cube{ private int tlX; private int tlY; private int size; public Cube(Graphics g) { tlX = 50; tlY = 50; size = 50; }  public Cube(Graphics g, int x, int y, int s)  {  tlX = x;  tlY = y;  size = s;  }    public void draw(Graphics g)  {  int tlX2 = tlX + size/3;  int tlY2 = tlY + size/3;  g.setColor(Color.black);  g.drawRect(tlX,tlY,size,size);  g.drawRect(tlX2,tlY2,size,size);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+size,tlY,tlX2+size,tlY2);  g.drawLine(tlX,tlY+size,tlX2,tlY2+size);  g.drawLine(tlX+size,tlY+size,tlX2+size,tlY2+size);  }  } | import java.awt.\*;import java.applet.\*;  public class Java0821 extends Applet{ public void paint(Graphics g)  {  Cube cube1 = new Cube(g,50,50,50);  cube1.draw(g);  Cube cube2 = new Cube(g,400,50,100);  cube2.draw(g);  Cube cube3 = new Cube(g,50,300,150);  cube3.draw(g);  Cube cube4 = new Cube(g,400,300,200);  cube4.draw(g);  }  } |



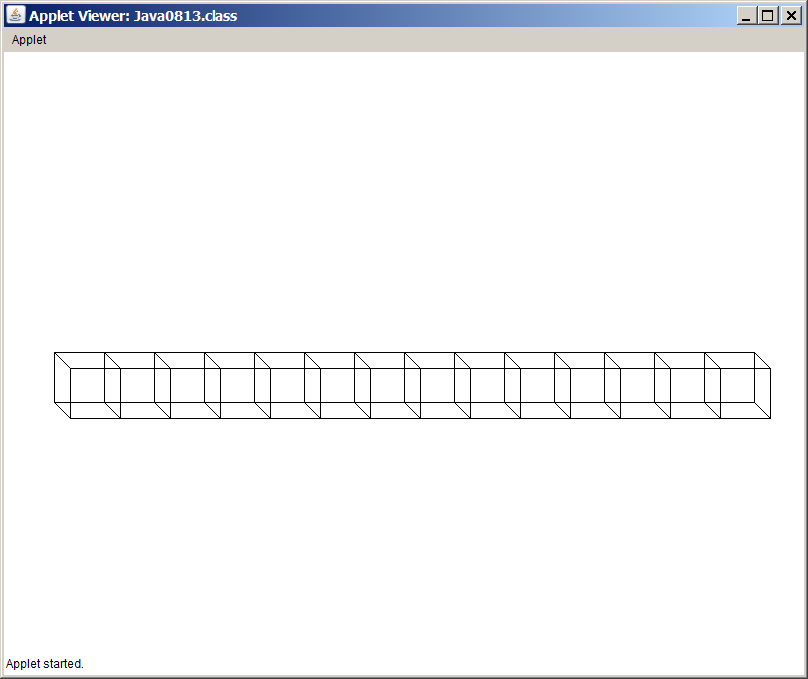
**Cube Case Study, Stage #4**

Program **Java0813.java**, in figure 8.13, adds the **move** method. It is now possible to relocate a cube to a new location after the initial construction of the cube object.

**Figure 8.13**

|  |
| --- |
| // Java0813.java  // Cube Casestudy #4  // Stage #4 adds a <move> method, which updates the cube's coordinates  // and draws a cube at the new location.  import java.awt.\*;  import java.applet.\*;  public class Java0813 extends Applet  {  public void paint(Graphics g)  {  Cube cube = new Cube(g,50,50,50);  for (int x = 50; x < 750; x += 50)  cube.move(g,x,300);  }  }  class Cube  {  private int tlX; // topleft X coordinate of the Cube's position  private int tlY; // topleft y coordinate of the Cube's position  private int size; // the size of the cube along one edge  public Cube(Graphics g)  {  tlX = 50;  tlY = 50;  size = 50;  }  public Cube(Graphics g, int x, int y, int s)  {  tlX = x;  tlY = y;  size = s;  }  public void draw(Graphics g)  {  int tlX2 = tlX + size/3;  int tlY2 = tlY + size/3;  g.setColor(Color.black);  g.drawRect(tlX,tlY,size,size);  g.drawRect(tlX2,tlY2,size,size);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+size,tlY,tlX2+size,tlY2);  g.drawLine(tlX,tlY+size,tlX2,tlY2+size);  g.drawLine(tlX+size,tlY+size,tlX2+size,tlY2+size);  }  public void move(Graphics g, int x, int y)  {  tlX = x;  tlY = y;  draw(g);  }  } |

**Figure 8.13 Continued**



**Cube Case Study, Stage #5**

Let's see if we can have some fun. Everybody likes animation. Moving objects simply is far more exciting than static objects going no place. Program **Java0814.java**, in figure 8.14, shows a **Cube** class with an **erase** method. The aim is to move the cube to a new location and then erase the cube from the previous location. What you see if a simple introduction to animation. Actually, you see absolutely nothing because the move and erase occur so quickly that the last cube location is erased before you had a chance to see anything.

**Figure 8.14**

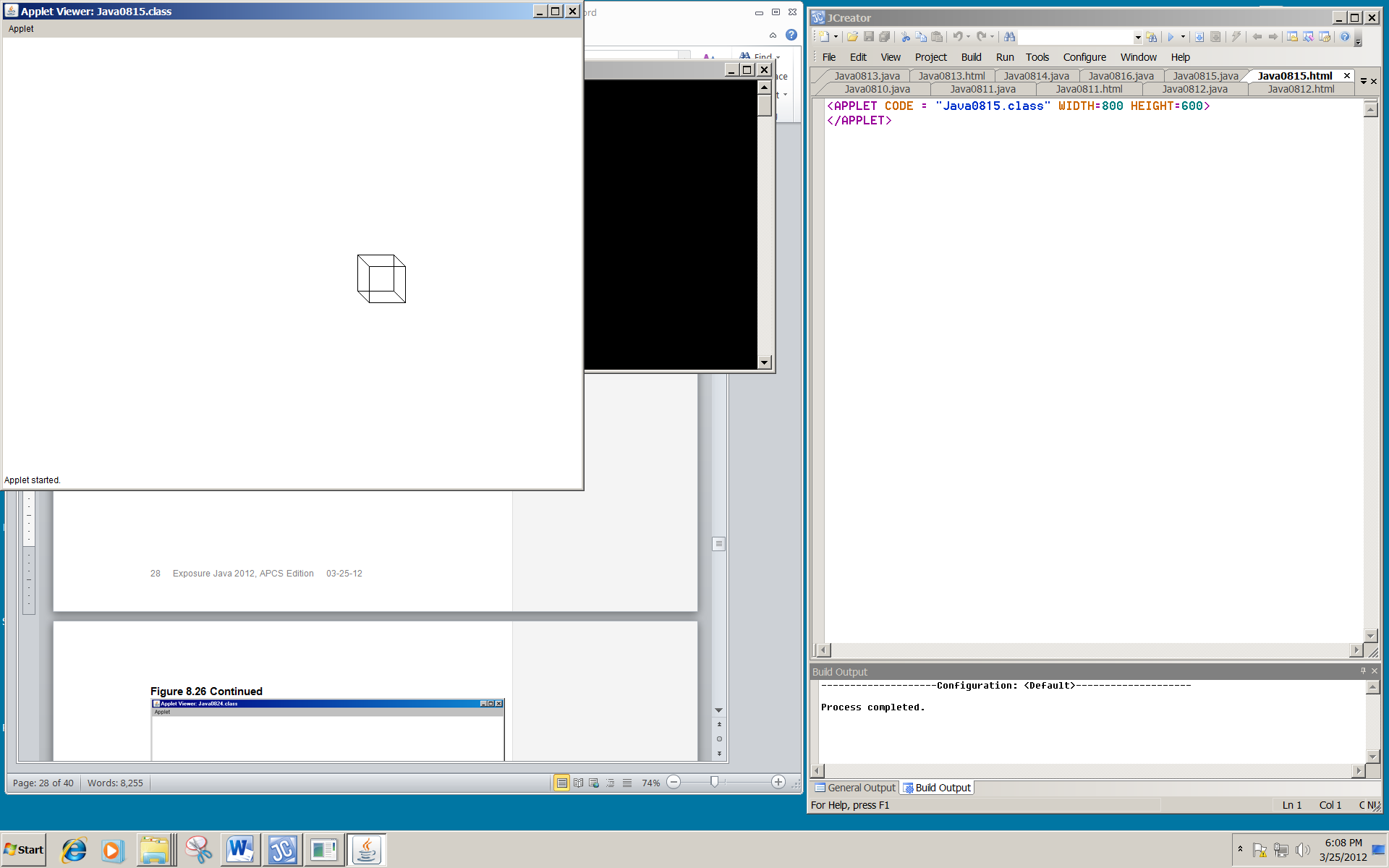
|  |
| --- |
| // Java0814.java// Cube Casestudy #5// Stage #5 adds an <erase> method, which erases the cube at the current [tlX,tlY] coordinates.  // This program has a problem because the cube object is erased immediately after it is drawn.  import java.awt.\*;  import java.applet.\*;    public class Java0814 extends Applet  {  public void paint(Graphics g)  {  Cube cube = new Cube(g,50,50,50);  for (int x = 50; x < 750; x += 50)  {  cube.move(g,x,300);  cube.erase(g);  }  }  }  class Cube  {  private int tlX; // topleft X coordinate of the Cube's position  private int tlY; // topleft y coordinate of the Cube's position  private int size; // the size of the cube along one edge  public Cube(Graphics g)  {  tlX = 50;  tlY = 50;  size = 50;  }  public Cube(Graphics g, int x, int y, int s)  {  tlX = x;  tlY = y;  size = s;  }  public void draw(Graphics g)  {  int tlX2 = tlX + size/3;  int tlY2 = tlY + size/3;  g.setColor(Color.black);  g.drawRect(tlX,tlY,size,size);  g.drawRect(tlX2,tlY2,size,size);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+size,tlY,tlX2+size,tlY2);  g.drawLine(tlX,tlY+size,tlX2,tlY2+size);  g.drawLine(tlX+size,tlY+size,tlX2+size,tlY2+size);  }  public void move(Graphics g, int x, int y)  {  tlX = x;  tlY = y;  draw(g);  }  public void erase(Graphics g)  {  int tlX2 = tlX + size/3;  int tlY2 = tlY + size/3;  g.setColor(Color.white);  g.drawRect(tlX,tlY,size,size);  g.drawRect(tlX2,tlY2,size,size);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+size,tlY,tlX2+size,tlY2);  g.drawLine(tlX,tlY+size,tlX2,tlY2+size);  g.drawLine(tlX+size,tlY+size,tlX2+size,tlY2+size);  }  } |

**Cube Case Study, Stage #6**

Our **Cube** class needs a small, clever method to help with this animation stuff. It is called method **delay**. Method delay stops program execution for a specified number of milliseconds. It is now possible to slow down the program execution sufficiently to observe some movement with **Java0815.java**, in figure 8.15.

**Figure 8.15**

|  |
| --- |
| // Java0815.java  // Cube Casestudy #6  // Stage #6 adds a <delay> method which stops program execution for a specified number of  // milli seconds. This makes the cube visible and creates a simple type of animation.  import java.awt.\*;  import java.applet.\*;  public class Java0815 extends Applet  {  public void paint(Graphics g)  {  Cube cube = new Cube(g,50,50,50);  for (int x = 50; x < 750; x += 20)  {  cube.move(g,x,300);  cube.delay(100);  cube.erase(g);  }  }  }  class Cube  {  private int tlX; // topleft X coordinate of the Cube's position  private int tlY; // topleft y coordinate of the Cube's position  private int size; // the size of the cube along one edge  public Cube(Graphics g)  {  tlX = 50;  tlY = 50;  size = 50;  }  public Cube(Graphics g, int x, int y, int s)  {  tlX = x;  tlY = y;  size = s;  }  public void draw(Graphics g)  {  int tlX2 = tlX + size/3;  int tlY2 = tlY + size/3;  g.setColor(Color.black);  g.drawRect(tlX,tlY,size,size);  g.drawRect(tlX2,tlY2,size,size);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+size,tlY,tlX2+size,tlY2);  g.drawLine(tlX,tlY+size,tlX2,tlY2+size);  g.drawLine(tlX+size,tlY+size,tlX2+size,tlY2+size);  }  public void move(Graphics g, int x, int y)  {  tlX = x;  tlY = y;  draw(g);  }  public void erase(Graphics g)  {  int tlX2 = tlX + size/3;  int tlY2 = tlY + size/3;  g.setColor(Color.white);  g.drawRect(tlX,tlY,size,size);  g.drawRect(tlX2,tlY2,size,size);  g.drawLine(tlX,tlY,tlX2,tlY2);  g.drawLine(tlX+size,tlY,tlX2+size,tlY2);  g.drawLine(tlX,tlY+size,tlX2,tlY2+size);  g.drawLine(tlX+size,tlY+size,tlX2+size,tlY2+size);  }  public void delay(int n)  {  long startDelay = System.currentTimeMillis();  long endDelay = 0;  while (endDelay - startDelay < n)  endDelay = System.currentTimeMillis();  }  } |



Make sure that you observe the program execution on a computer. If you look at the textbook page, you will see the cube in one location. The output display captured the cube on its path, but it does move.

**Cube Case Study, Stage #7**

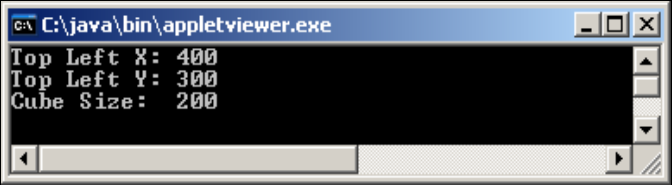
Animation is fun, but we need to return to reality. The last stage in this case study is **Java0816.java**, in figure 8.16. The **Cube** class adds three important *get* methods that return the data of the **Cube** object. In a graphics program this type of information is especially important for debugging purposes when data values need to checked. The program below is not complete.

This program displays a single large cube, but behind the graphics window you will see a text window that displays the data attribute values.

**Figure 8.16**

|  |
| --- |
| // Java0816.java  // Cube Casestudy #7  // Stage #7 adds three get methods that return the instance variable values.  // They are methods <getX>, <getY> and <getSize>.  class Cube  {  private int tlX; // topleft X coordinate of the Cube's position  private int tlY; // topleft y coordinate of the Cube's position  private int size; // the size of the cube along one edge  public int getX()  {  return tlX;  }  public int getY()  {  return tlY;  }  public int getSize()  {  return size;  }  } |

**Figure 8.16 Continued**



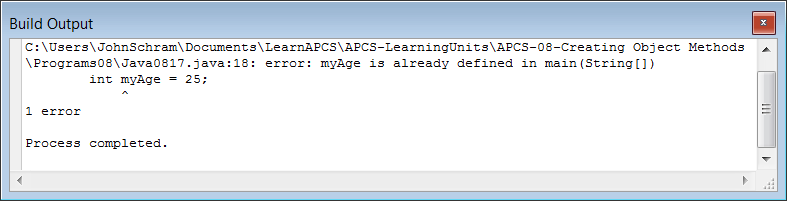
**8.7 The Consequences of Scope**

The next topic in this chapter may seem to be outside the *encapsulation* business. Read carefully and play with the program examples. This is not a "pure" OOP topic, but it is important and *scope* must be understood properly to create correct classes and methods.

Consider program **Java0817.java**, in figure 8.17. This program defines variable **counter** twice, followed by a double definition of **myAge**. The program does not compile. Perhaps you are not surprised, because duplicate definitions are not allowed and yet the error message is only concerned about the duplicate definition of **myAge**. Apparently Java does not get excited about a double **counter**. The issue is a matter of understanding *scope*.

**Figure 8.17**

|  |
| --- |
| // Java0817.java  // This program demonstrates how one variable name <counter> can be  // declared twice correctly.  // It also shows <myAge> declared twice incorrectly.  public class Java0817  {  public static void main(String args[])  {  for (int counter = 1; counter <= 5; counter++)  System.out.print(counter + " ");  for (int counter = 10; counter <= 15; counter++)  System.out.print(counter + " ");  int myAge = 16;  int myAge = 25;  }  } |



Program **Java0818.java**, in figure 8.18, is a little strange looking. There are four variables defined in various locations of the file. This program is odd in the practical sense, but created to demonstrate the scope of four different variables.

**Figure 8.18**

|  |
| --- |
| // Java0818.java  // This program demonstrates the scope of a variable.  public class Java0818  {  public static void main(String args[])  {  int var1 = 10;  System.out.println("var1 in main is " + var1);  System.out.print("var2 inside the main method for loop is ");  for (int var2 = 1; var2 < 10; var2++)  {  System.out.print(var2 + " ");  }  System.out.println();  Boo boo = new Boo(var1);  System.out.println("var4 in Boo is " + boo.getData());  System.out.println();  }  }  class Boo  {  private int var4;  public Boo(int var3)  {  var4 = var3;  System.out.println("var3 in constructor is " + var3);  }  public int getData()  {  return var4;  }  } |

|  |
| --- |
| **Java0818.java Output**  var1 in main is 10  var2 inside the method for loop is 1 2 3 4 5 6 7 8 9  var3 in constructor is 10  var4 in Boo is 10 |

What is scope? The scope of a variable - simple, primitive data type or complex object - is the segment of a program during which a variable is defined, has allocated memory to store values and can be accessed. You can think of scope somewhat like jurisdiction of a police department. Dallas police officers have jurisdiction in the city of Dallas only. The Dallas county sheriff's department has jurisdiction over the entire county of Dallas, which includes the city of Dallas. The Texas state troopers have jurisdiction over the entire state of Texas, which includes Dallas county and the city of Dallas.

Identifier **var1**, in figure 8.19,is declared in **main** and has access for the entire **main** method. Identifier **var2**, in figure v8.20,is also defined in **main**, but the scope is limited to the **for** loop heading and body. Identifier **var3**, in figure 8.21, has a scope limited to the constructor of **Boo** and finally, identifier **var4**, in figure8.22,has scope of the entire **Boo** class.

**Figure 8.19 shows the scope of var1**

|  |
| --- |
| // Java0818.java  // This program demonstrates the scope of a variable.  public class Java0818  {  public static void main(String args[])  {  int var1 = 10;  System.out.println("var1 in main is " + var1);  System.out.print("var2 inside the main method for loop is ");  for (int var2 = 1; var2 < 10; var2++)  {  System.out.print(var2 + " ");  }  System.out.println();  Boo boo = new Boo(var1);  System.out.println("var4 in Boo is " + boo.getData());  System.out.println();  }  }  class Boo  {  private int var4;  public Boo(int var3)  {  var4 = var3;  System.out.println("var3 in constructor is " + var3);  }  public int getData()  {  return var4;  }  } |

**Figure 8.20 shows the scope of var2**

|  |
| --- |
| // Java0818.java  // This program demonstrates the scope of a variable.  public class Java0818  {  public static void main(String args[])  {  int var1 = 10;  System.out.println("var1 in main is " + var1);  System.out.print("var2 inside the main method for loop is ");  for (int var2 = 1; var2 < 10; var2++)  {  System.out.print(var2 + " ");  }  System.out.println();  Boo boo = new Boo(var1);  System.out.println("var4 in Boo is " + boo.getData());  System.out.println();  }  }  class Boo  {  private int var4;  public Boo(int var3)  {  var4 = var3;  System.out.println("var3 in constructor is " + var3);  }  public int getData()  {  return var4;  }  } |

**Figure 8.21 shows the scope of var3**

|  |
| --- |
| class Boo  {  private int var4;  public Boo(int var3)  {  var4 = var3;  System.out.println("var3 in constructor is " + var3);  }  public int getData()  {  return var4;  }  } |

**Figure 8.22 shows the scope of var4**

|  |
| --- |
| class Boo  {  private int var4;  public Boo(int var3)  {  var4 = var3;  System.out.println("var3 in constructor is " + var3);  }  public int getData()  {  return var4;  }  } |

|  |
| --- |
| **Scope Definition** |
| What is scope? The scope of a variable - simple, primitive data type or complex object - is the segment of a program during which a variable is defined, has allocated memory to store values and can be accessed. |
| If two variables have the same identifier and also the same scope, Java will object with a duplicate definition compile error. |

There is more to the scope story. There are programmers who like the actual parameters, the formal parameters and any other variables carrying the same value to all have the same identifiers. Java does not object. Making variables throughout a program use the same identifier, if they carry the same value, clarifies the intent of the variable.

This approach can cause a logic error that is shown by **Java0819.java**, in figure 8.23. When the **Widget** object **w** is constructed, value **100** is passed to the constructor. The intent is that the new object will store 100 widgets. The reality is that not a single widget is stored and **numWidgets** returns an initial default value of **0**.

**Figure 8.23**

|  |
| --- |
| // Java0819.java  // This program shows the logic problem that results from using two variables  // with the same name identifier, but two different scopes.  public class Java0819  {  public static void main(String args[])  {  Widget w = new Widget(100);  System.out.println("Object w has " + w.getWidgets() + " widgets");  }  }  class Widget  {  private int numWidgets;  public Widget(int **numWidgets**)  {  **numWidgets = numWidgets;**  }  public int getWidgets()  {  return numWidgets;  }  } |

|  |
| --- |
| **Java0819.java Output**  Object w has 0 widgets |

The problem is the highlighted **numWidgets = numWidgets** statement. This identifier is identical to the **Widget** private attribute **numWidget**. There is no compile issue with this approach, because **numWidget** in the constructor and **numWidget** as the class attribute have different scope. Java is fine. It is great that everybody is fine, but it also means that value **100** does not go anywhere beyond the constructor.

There are two solutions to solve this logic error and do remember that logic errors will never create any type of compile error message or runtime exception indication. Java lets the programmer deal with logic errors. The first solution is shown by **Java0820.java**, in figure 8.24. Use a different identifier for the formal parameter in the method heading than the attribute identifier. The program output now displays the correct output. The logic error is removed. This solves the problem nicely for most people. Many programmers prefer this style of programming, but the *same identifier* group is not happy.

**Figure 8.24**

|  |
| --- |
| // Java0820.java  // Using different variable names is one solution to the  // problem caused by program Java0819.java.  public class Java0820  {  public static void main(String args[])  {  Widget w = new Widget(100);  System.out.println("Object w has " + w.getWidgets() + " widgets");  }  }  class Widget  {  private int numWidgets;  public Widget(int **nW**)  {  numWidgets = **nW**;  }  public int getWidgets()  {  return numWidgets;  }  } |

|  |
| --- |
| **Java0820.java Output**  Object w has 100 widgets |

There is a way to make our *same identifier* people happy. Java has a special keyword called **this**. **this** is a reference and stores a memory value. Specifically, **this** stores the reference value of the current object in use. **Java0821.java**, in figure 8.25, has a second solution to the logic error of program **Java0819.java**.

The statement **this.numWidgets = numWidgets** now makes perfect sense to Java. The first **numWidgets** is the attribute of the object being called, which in this case is being constructed. Java knows this (normal English this) because **this** has the same reference as object **w**.

In method **getWidgets** the **this** reference is also used, but in that case it is optional. There is no possible confusion for Java. Inside method **getWidgets** is no definition of **numWidgets**. To Java this means that the attribute **numWidgets**, which has scope in the entire class is the intended variable. Technically, **numWidgets** has scope everywhere in class **Widget**, but the local declaration of **numWidgets** in the constructor has priority over the global class definition.

**Figure 8.25**

|  |
| --- |
| // Java0821.java  // Using the <this> reference is a second solution to the  // problem in program Java0819.java.  public class Java0821  {  public static void main(String args[])  {  Widget w = new Widget(100);  System.out.println("Object w has " + w.getWidgets() + " widgets");  }  }  class Widget  {  private int numWidgets;  public Widget(int numWidgets)  {  **this.numWidgets = numWidgets;** // required use of this  }  public int getWidgets()  {  **return this.numWidgets;**  // optional use of this  }  } |

|  |
| --- |
| **Java0821.java Output**  Object w has 100 widgets |

The proper use of **this** is very important in encapsulation. If it is not quite clear yet what exactly **this** references, look at program **Java0822.java**, in figure 8.26. In this program you will three objects of the **Widget** class instantiated. In each case the value of the object is displayed. Since objects store a memory reference to the location where the data is stored, the output becomes a memory location. Java usually starts with the name of the class followed by the memory address.

There is also a **println** statement inside the constructor of the **Widget** class and the value printed is **this**. Look closely at the output. You will first see the value of **this**, followed by the value of **w1**, then **w2** and finally **w3.** Three different memory locations will be shown for each one of the **Widget** objects. In each case the memory location is the same as the current object in context. In other words when I am working with **w1** then **this** has the same value as **w1**. Likewise when new memory is located for **w2** you will see that **this** also gets this new address.

**Figure 8.26**

|  |
| --- |
| // Java0822.java  // Comparing the value of the three <Widget> objects demonstrates  // that the <this> reference value is equal to the current object used.  public class Java0822  {  public static void main(String args[])  {  Widget w1 = new Widget(100);  System.out.println("w1 value: " + w1);  System.out.println();  Widget w2 = new Widget(100);  System.out.println("w2 value: " + w2);  System.out.println();  Widget w3 = new Widget(100);  System.out.println("w3 value: " + w3);  System.out.println();  }  }  class Widget  {  private int numWidgets;  public Widget(int numWidgets)  {  this.numWidgets = numWidgets;  System.out.println("this value: " + this);  }  } |

|  |
| --- |
| **Java0822.java Output**  this value: Widget@9664a1  w1 value: Widget@9664a1  this value: Widget@1a8c4e7  w2 value: Widget@1a8c4e7  this value: Widget@1172e08  w3 value: Widget@1172e08 |

The scope and **this** section will finish by looking at the **GridWorld Case Study**. The **moveTo** method of the **Actor** class uses **this** twice. Look at **Java0823.java**, in figure 8.27. The **GridWorld** program is very complex and designed to operate properly in many different situations. Many, many objects can be on the grid at the same time and each one is asked to **act** during each *step* of the program execution. With so many objects moving around the grid a check is made to make sure that the object at the current location is the same one that we plan to move. In the second use of **this** the current object can finally be placed at the specified location. What goes to the specified location? The current object in context. What is the current object? It is **this.**

**Figure 8.27**

|  |
| --- |
| // Java0823.java  // The <moveTo> method of the <Actor> class used by the GridWorld  // case study shows two uses of the <this> reference.  // This file is incomplete and will not compile.  public class Actor  {  public void moveTo(Location newLocation)  {  if (grid == null)  throw new IllegalStateException("This actor is not in a grid.");  if (**grid.get(location) != this**)  throw new IllegalStateException("The grid contains a different actor at location " + location + ".");  if (!grid.isValid(newLocation))  throw new IllegalArgumentException("Location " + newLocation + " is not valid.");  if (newLocation.equals(location))  return;  grid.remove(location);  Actor other = grid.get(newLocation);  if (other != null)  other.removeSelfFromGrid();  location = newLocation;  **grid.put(location, this);**  }  } |

|  |
| --- |
| **this reference** |
| **this** references the object in context. This means that in a statement, such as **tom.makeDeposit(1000)**, the reference address stored by **tom** will be identical to **this** when used inside the **makeDeposit** method. |

|  |
| --- |
| **void Methods With return** |
| A **void** method can have a **return** statement as long as it does not *return* anything.  This **return** simply *exits* the method. |

**8.8 Method Summary**

The primary focus of this chapter was on **creating** **object methods** as opposed to **creating** **class methods** introduced in the last chapter. By now you have already seen an amazing number of different methods. In an effort to clarify all these methods, this chapter will conclude with a review of each type of method. Each method will be briefly explained and then an example of such a method is shown to review the precise Java syntax that is required.

## **Class or Static Methods**

A **class** method calls the class directly, not an object of the class. This is practical when there is no need to make multiple copies of a class. A good example is Java’s **Math** class. Everybody can use the methods of the **Math** class and there is no need to make multiple objects. The value of **PI** will be same with everybody using this class. Use the reserved word **static** at the start of a method heading to indicate that the method is a *class method*. In figure 8.29 notice how each method in the **Piggy** class is called by using the class identifier **Piggy**.

**Figure 8.29**

|  |
| --- |
| public class Demo  {  public static void main(String args[])  {  **Piggy.**initData();  **Piggy.**showData();  **Piggy.**addData(1200);  **Piggy.**showData();  }  }  class Piggy  {  public **static** double savings;  public **static** void initData() { savings = 0; }  public **static** void addData(double s) { savings += s; }  public **static** void showData() { System.out.println("Savings: " + savings); }  } |

## **Object or Non-Static Methods**

Object methods are meant for those situations where multiple copies of a class must be constructed. The essence of the method does not change. What really changes is the fact that now an object must be constructed first with the **new** operator, and then the method can be called. In contrast to the class methods, object methods are called by using the object identifier. In the example below, the same **Piggy** class is used, with the same methods. Now the methods do not use the **static** keyword and figure 8.30 shows that any object must be called with an object identifier, like **tom**.

**Figure 8.30**

|  |
| --- |
| public class Demo  {  public static void main(String args[])  {  Piggy **tom** = new Piggy();  **tom.**initData();  **tom.**showData();  **tom.**addData(1200);  **tom.**showData();  }  }  class Piggy  {  private double savings;    public void initData()  {  savings = 0;  }  public void addData(double s)  {  savings += s;  }  public void showData()  {  System.out.println("Savings: " + savings);  }  } |

Make also a note that the example in figures 8.29 and 8.30 differ in style. The methods in figure 8.29 are written on one line. This is not a class method requirement. Identical methods in figure 8.30 are written on multiple lines. Java is a **free-form** language and programming style is a matter of convention and taste. Frequently, small methods are written on a single line.

## **Private Methods**

**Private** methods can only be accessed by methods of the same class. Normally, such methods are called *helper methods* and they assist other methods that will be used by some client program of the class. The shuffleCards method is an example of a private method in figure 8.31.

**Figure 8.31**

|  |
| --- |
| **private** void shuffleCards()  {  System.out.println("Shuffle Cards");  } |

## **Public Methods**

A **public** method can be accessed by members of the same class, and more importantly public methods can be accessed by any client of the class. The majority of methods are public. Method **getCards**, in figure 8.32 is public.

**Figure 8.32**

|  |
| --- |
| **public** int getCards()  {  return cardsLeft;  } |

## **Void Methods**

There are many ways to classify methods. Methods are not just **public** only, or **private** only, or **static** only, or anything else only. You, personally, have many classifications at the same time. You can be a Sophomore, Junior or Senior. At the same time you can be Male or Female. You can also belong to a religious group and you can be Catholic, Muslim, Jewish, Protestant or non-denominational or no religion at all. Classification also can be based on school organizations. You can be a band member, cheerleader, football player, gymnast, soccer player, academic decathlon member, etc. The point is do not be surprised if you see that a method shows up in many different classifications. The method in figure 8.33 is a public method, but for the purpose of this classification we are interested in the fact that it is a **void** method. Void methods do not return a value and use the reserved word **void** to indicate that no value will be returned.

**Figure 8.33**

|  |
| --- |
| public **void** showData()  {  System.out.println("Name: " + name);  System.out.println("Savings: " + savings);  } |

## **Return Methods**

Return methods are methods that return a value. Two features are necessary for a return method, and look for them in figure 8.34. First, you will see that the method heading indicates a data type, which is the type that the method returns. Second, you see a **return** statement in the method body.

**Figure 8.34**

|  |
| --- |
| public **double** getSavings()  {  **return** savings;  } |

## **Default Constructor Methods**

A constructor is a special method that is automatically called during the instantiation of a new object. If no visible constructor is provided Java will provide its own constructor, called a **default** constructor. I also call a *no-parameter* constructor a **default** constructor, like the example shown in figure 8.35. Constructors are always **public** and they are special methods that are neither **void** nor **return** methods.

**Figure 8.35**

|  |
| --- |
| public CardDeck()  {  System.out.println("Constructing a default CardDeck object");  numDecks = 1;  numPlayers = 1;  cardsLeft = 52;  shuffleCards();  } |

## **Overloaded Constructor Methods**

An **overloaded** constructor is a second, third or more constructor that allows a new object to be instantiated according to some specifications that are passed by parameters. Figure 8.36 shows an overloaded **CardDeck** constructor, which enters **numDecks** and **numPlayers** information.

**Figure 8.36**

|  |
| --- |
| public CardDeck(**int d, int p**)  {  System.out.println("Constructing a CardDeck object with parameters");  numDecks = d;  numPlayers = p;  cardsLeft = d \* 52;  shuffleCards();  } |

## **Accessing or Get Methods**

Methods that only access object data without altering the data are access methods or frequently called **get** methods. Frequently access methods are **return** methods, which return object private data information. The example in figure 8.37 returns the value of **numDecks.**

**Figure 8.37**

|  |
| --- |
| public int getDecks()  {  return **numDecks**;  } |

## **Altering or Modifier or Mutator or Set Methods**

*Altering*methods are sometimes also called *modifier* methods or *mutator*methods or *set*methods. These are methods that not only access the **private** data of an object; they also alter the value of the data. The example in figure 8.21 accesses the **savings** attribute and adds the value of **s** to it.

**Figure 8.21**

|  |
| --- |
| public void savingsDeposit(double s)  {  **savings += s;**  } |

**8.9 GridWorld’s main Method**

This GridWorld section is not directly related to the *OOP, Encapsulation* chapter focus. Perfect integration between the Computer Science curriculum and the GridWorld Case Study is not always possible. It is possible in Chapter IX and GridWorld will do an excellent job helping to explain inheritance. In this chapter it is a different story and GridWorld is standing on its own. This is OK and it will help with future topics that you need to learn.

Each time that you create a project for a GWCS lab assignment or lab experiment it starts with some type of execution display of the GridWorld. You have learned that different classes have different behaviors. **Bug** objects and **Rock** objects have radically different acting behaviors. Unless such acting behaviors are altered you can count on consistent behavior every time you see an object of a certain class on the GridWorld.

A different question is concerned with the initial GridWorld display when a GridWorld program starts. How is that controlled? Also, is it possible to control the manner in which the objects behave from the **main** method?

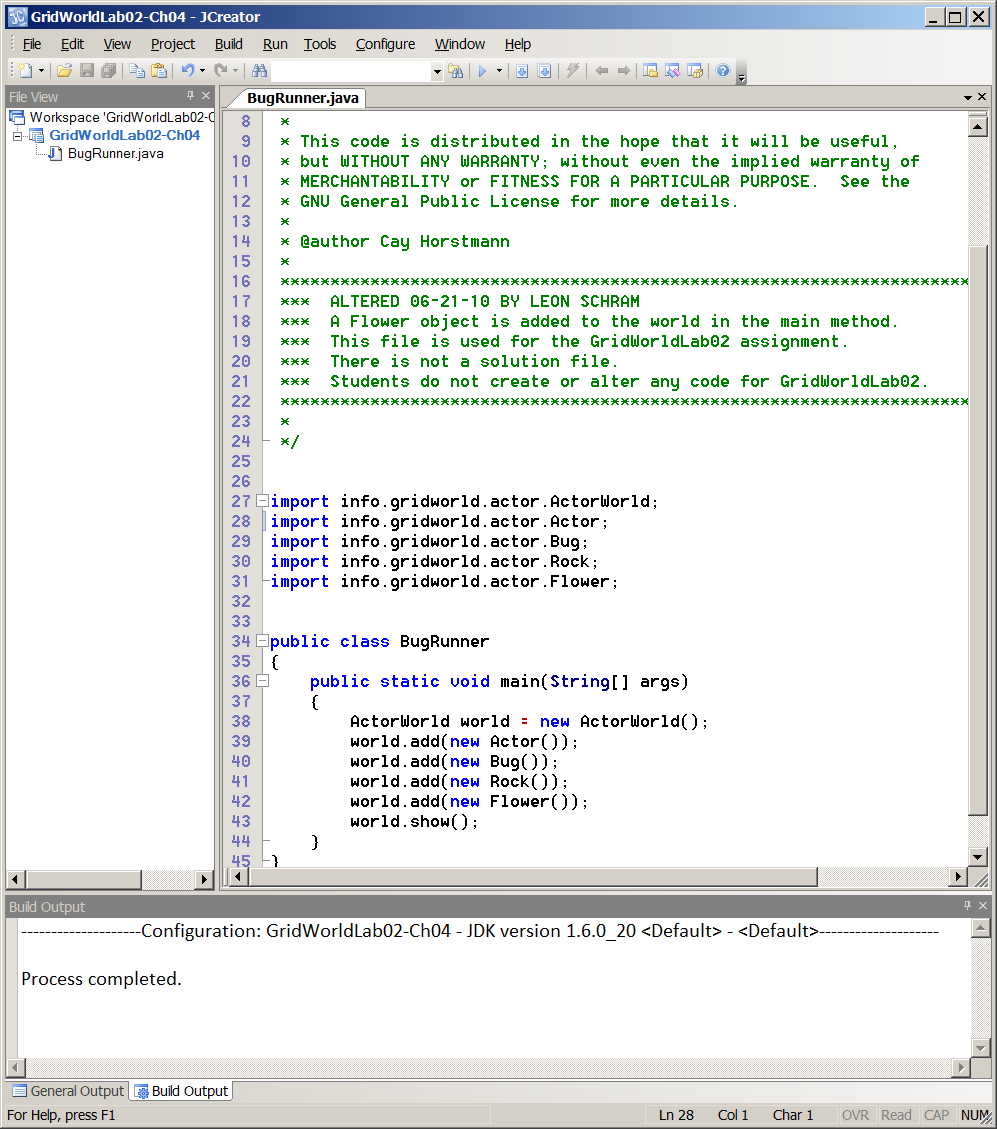
There are many mysteries of GridWorld that happen behind the curtain, but the **main** method is very visible. You can alter the program statements of the **main** method and change the entire manner of program execution. The purpose of this section is to examine the **main** method of GridWorld and learn how to manipulate a variety of different GridWorld executions.

**Using the Correct import Statements**

Every GridWorld program has a file that tests some new feature. In Chapter VII you made some changes to the **Bug.java** file and altered **Bug** object behavior. It was necessary to have a special class that tests the new **Bug** class. This will always be a class that contains the **main** method. The class that contains the **main** method may be called the *main* class, the *primary* class, the *runner* class or the *testing* class. It does not matter what it is called, but one thing is sure… this primary class needs a correct **main** method along with the necessary **import** statements to make the complete GridWorld project work correctly.

Figure 8.39 shows the **import** statements of some GridWorld file. These import statements are necessary to give access to the classes used by the program. Java has many, many standard libraries that can be accessed by import statements. The GridWorld Case Study has created its own set of packages to organize the many available classes.

**Figure 8.39**



Let's start with a quick review or possibly a first introduction to **import** statements used by Java. Java has hundreds of special toolkits, called *classes*. Each one of these toolkits has a set of tools, called *methods*. The quantity of classes and methods in Java is so large that there simply is not enough memory in a computer to allow access to all the available Java features. There are some very common standard Java classes and methods that are frequently used. They are in the *language* package and this *language* package is automatically loaded in memory. All other classes are stored in packages and must be imported to allow access to the package. So what is a package? Consider the following hierarchy.

|  |
| --- |
| **Java Package Hierarchy** |
| A **keyword** is a word with special meaning in Java, like **void**, **public**, **static**, **main**, **int**, **double**, etc.  A **program statement** is a collection of keywords.  A **method** contains a group of program statements to perform some desired process.  A **class** contains a set of methods with a common purpose, like the **Math** class.  A **package** contains a group of classes of a common purpose. |

The language package includes the simple data types, the **System** class, the **Math** class and a few other commonly used Java features. Access to any classes that are not in the language package requires special **import** statements. Import statements include the package path and the included class name. Look at figure 8.39 again and notice that five separate import statements are used to give access to five GridWorld classes.

**Figure 8.39**

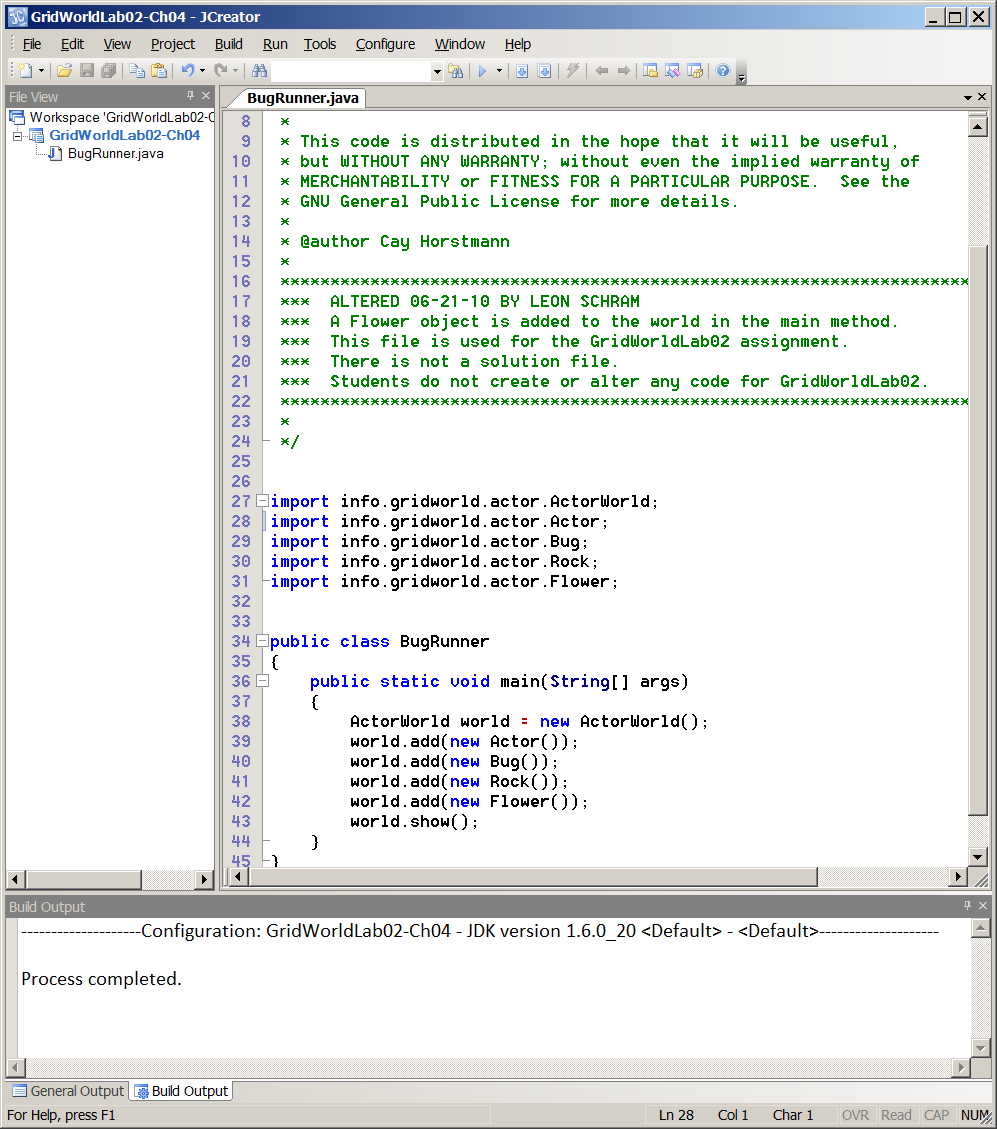
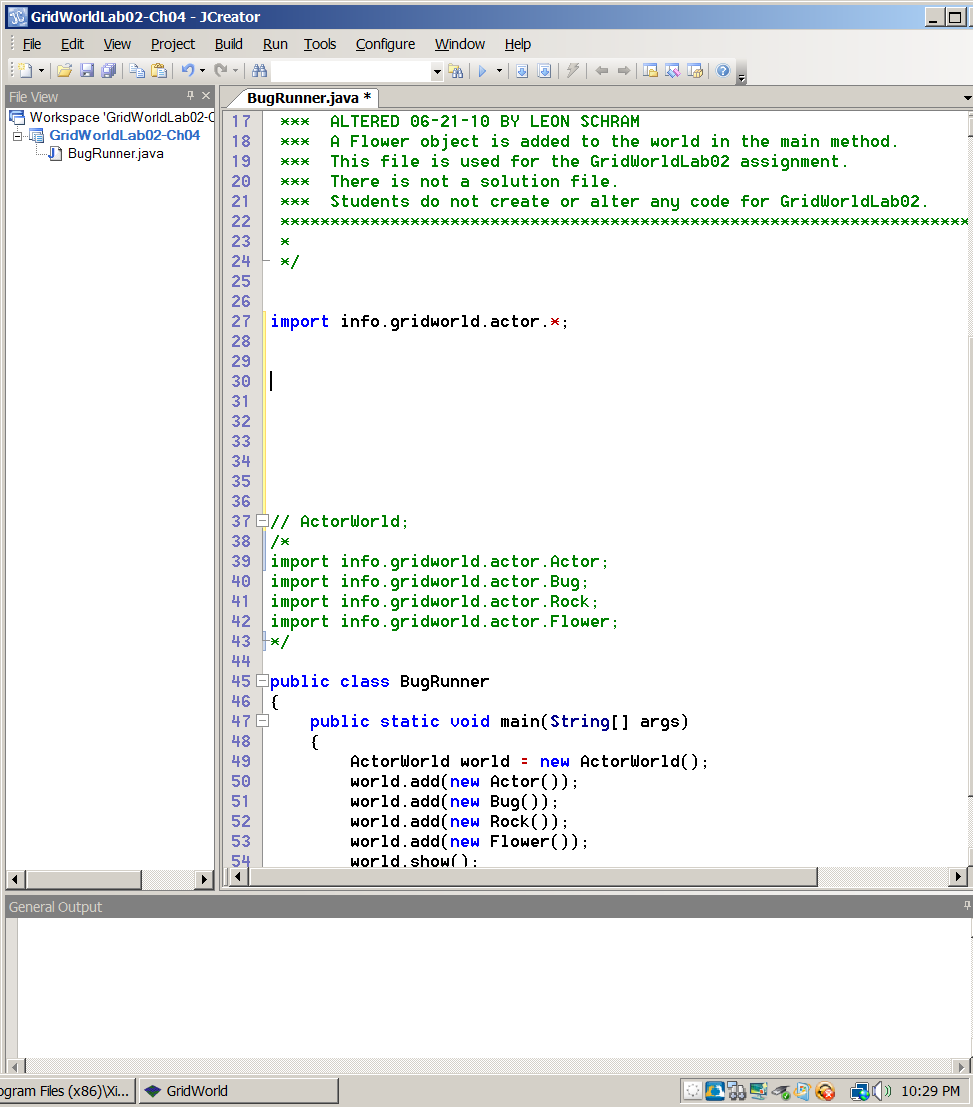


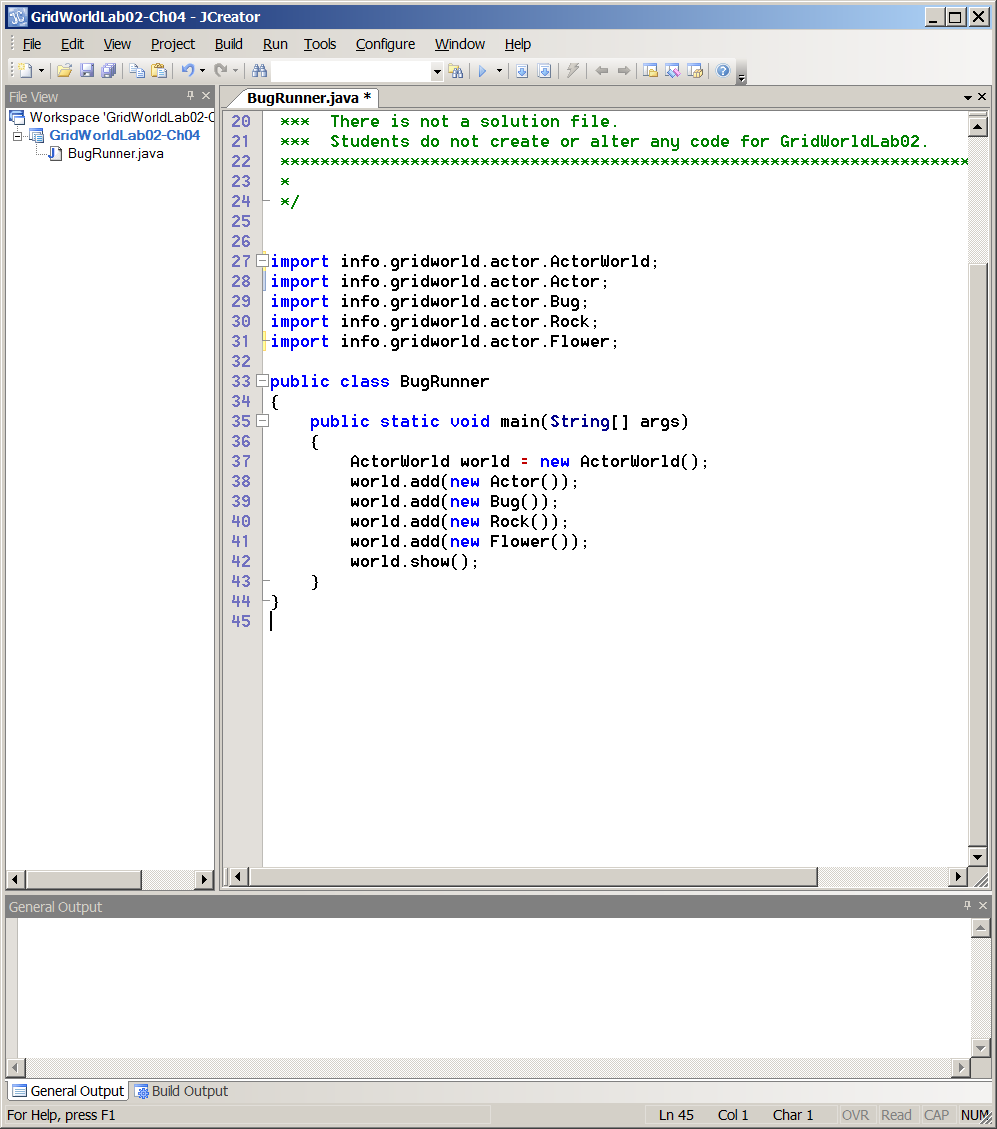
Figure 8.40 gives access to the same five classes and possibly more classes with a single statement that uses the *wildcard* character. It may seem logical to always use wildcard import statements, but with multiple wildcard statements it is difficult to determine where classes are located.

**Figure 8.40**

****

Now look at the complete code of the **BugRunner.java** file in figure 8.41 and compare the import statements with the statements in the main method.

**Figure 8.41**

****

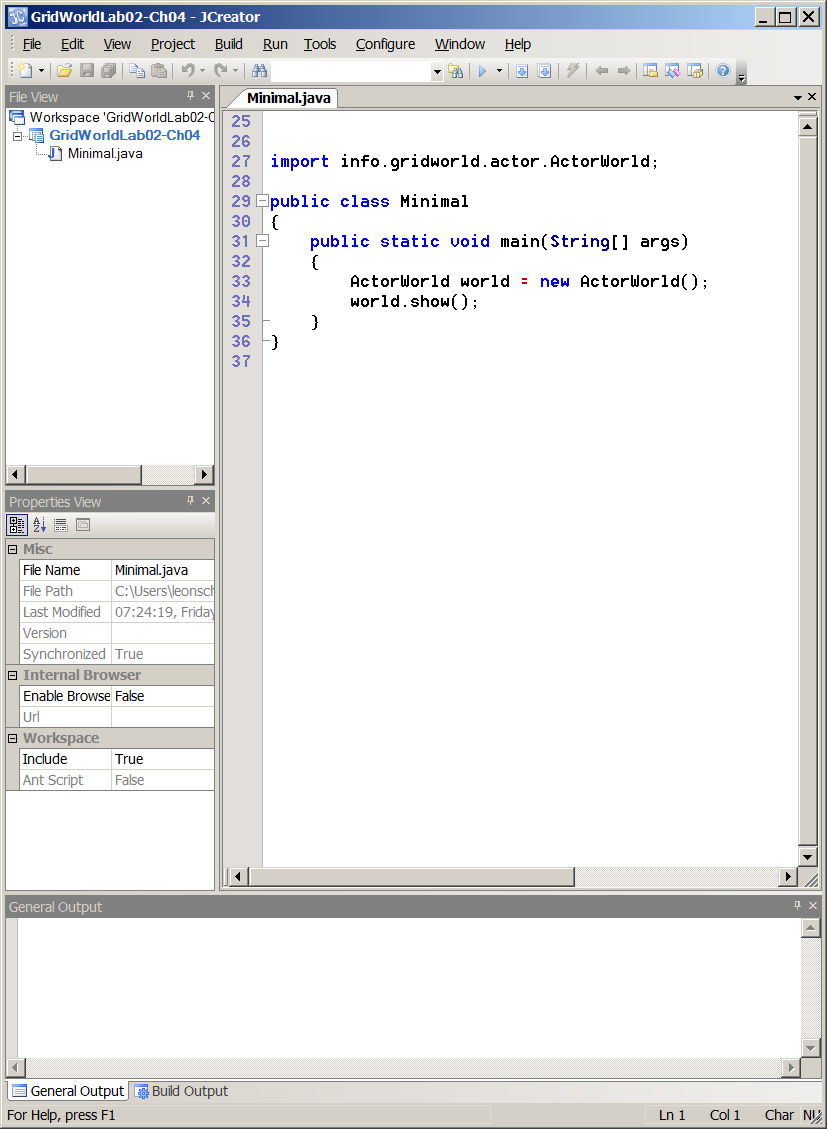
In the **main** method you see five **new** statements used to create five different objects. Note that each of the objects created is one of the classes, which has access courtesy of an **import** statement. Sometimes students use the GridWorld files provided by the College Board, make some changes, and then are surprised when the changes do not compile. Often the problem is that necessary **import** statements are not available.

**The Minimal GridWorld main Method**

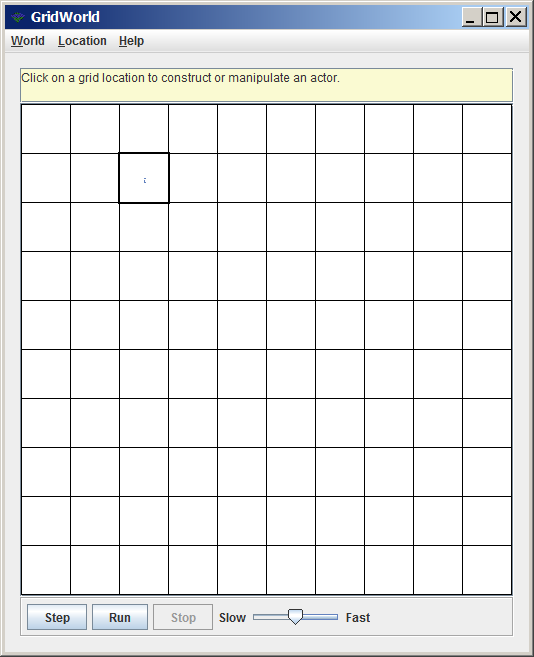
Keep in mind that you are not responsible for using, or even understanding, any of the GUI interface business. In other words, do not worry about how the grid gets on the monitor and how the objects get on the grid. You only need to know how to make the grid appear and how to put objects on the grid. Once that stage is done you can then move on and start to manipulate class behavior.

Figure 8.42 shows a minimal GridWorld **main** method. There needs to be an **ActorWorld** object, called **world** in this case. Also, you need to call the **show** method to make the GridWorld appear. The Execution is shown by figure 8.43.

**Figure 8.42**

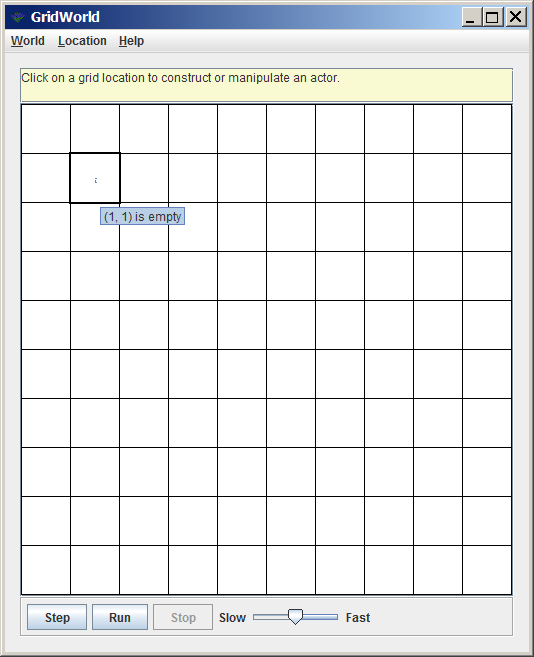
****

**Figure 8.43**

****

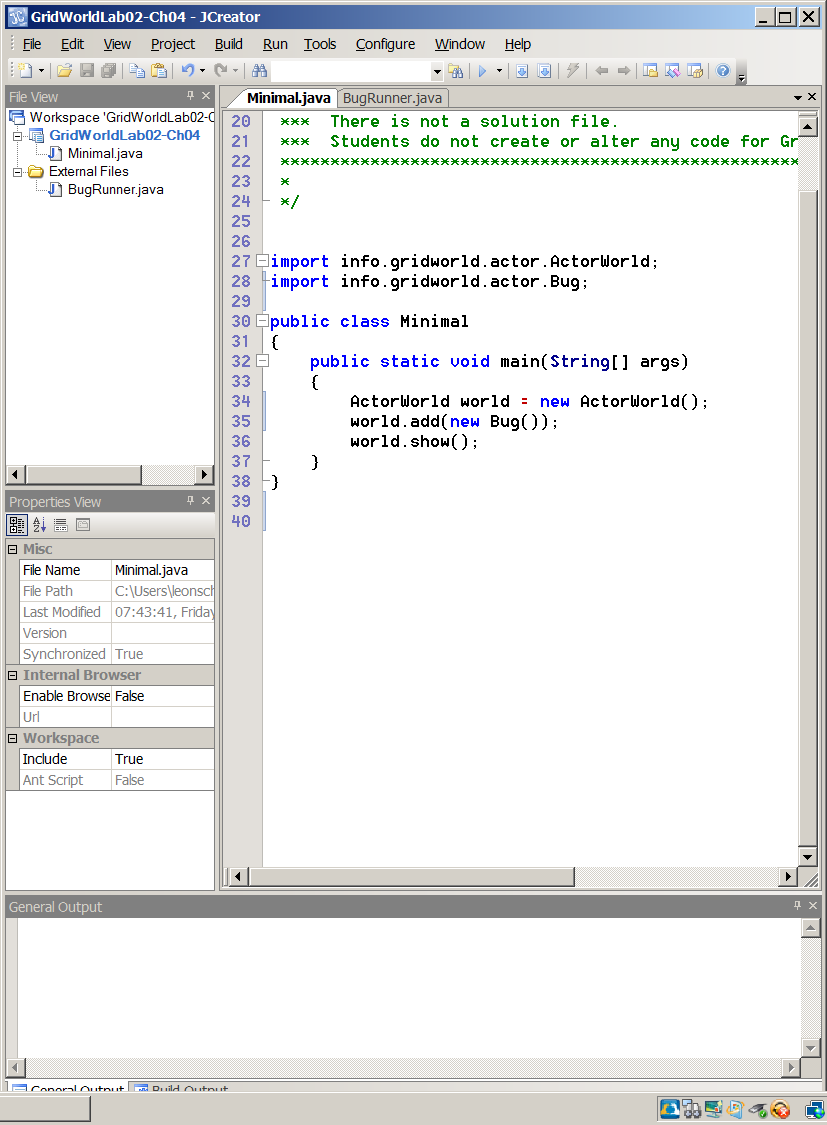
The execution in figure 8.43 is rather plain, but it was mentioned that this was a minimal GridWorld **main** method. Perhaps you are not concerned, because you did learn that you can click on an empty cell and make different objects appear on the GridWorld, ready for action. Clicking on an empty cell may bring a surprise. Figure 8.44 shows some disappointing news. You are told that the cell is empty and there is no list of constructors to select from for the creation of a new object.

**Figure 8.44**

****

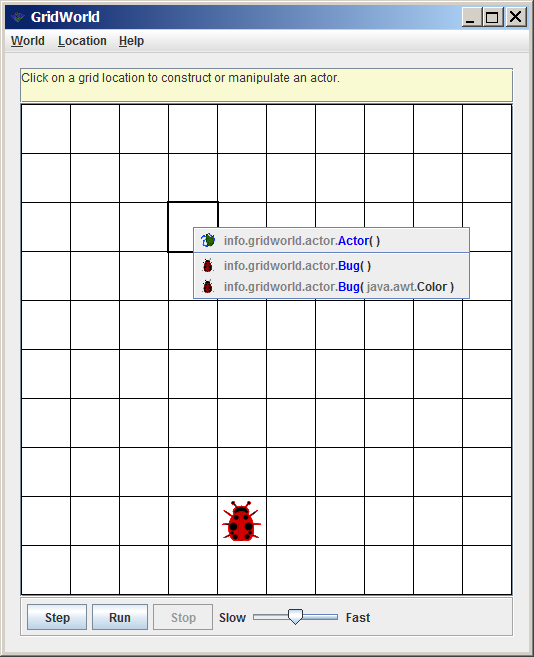
The mystery may be resolved by making a slight change in the **main** method. Figure 8.45 adds one statement to the **main** method. Additionally, there is a second **import** statement to allow access to the **Bug** class.

**Figure 8.45**

****

When the new program is compiled and executed, figure 8.46 appears. There is one **Bug** object shown and clicking on an empty cell makes it now possible to create an **Actor** object or a **Bug** object. You may think that there are two bugs, but that is the same class. There are two constructors for the **Bug** class. The first constructor has no parameters and creates a default bug, which is red. The second constructor allows the opportunity to create a bug of a specified color.

**Figure 8.46**

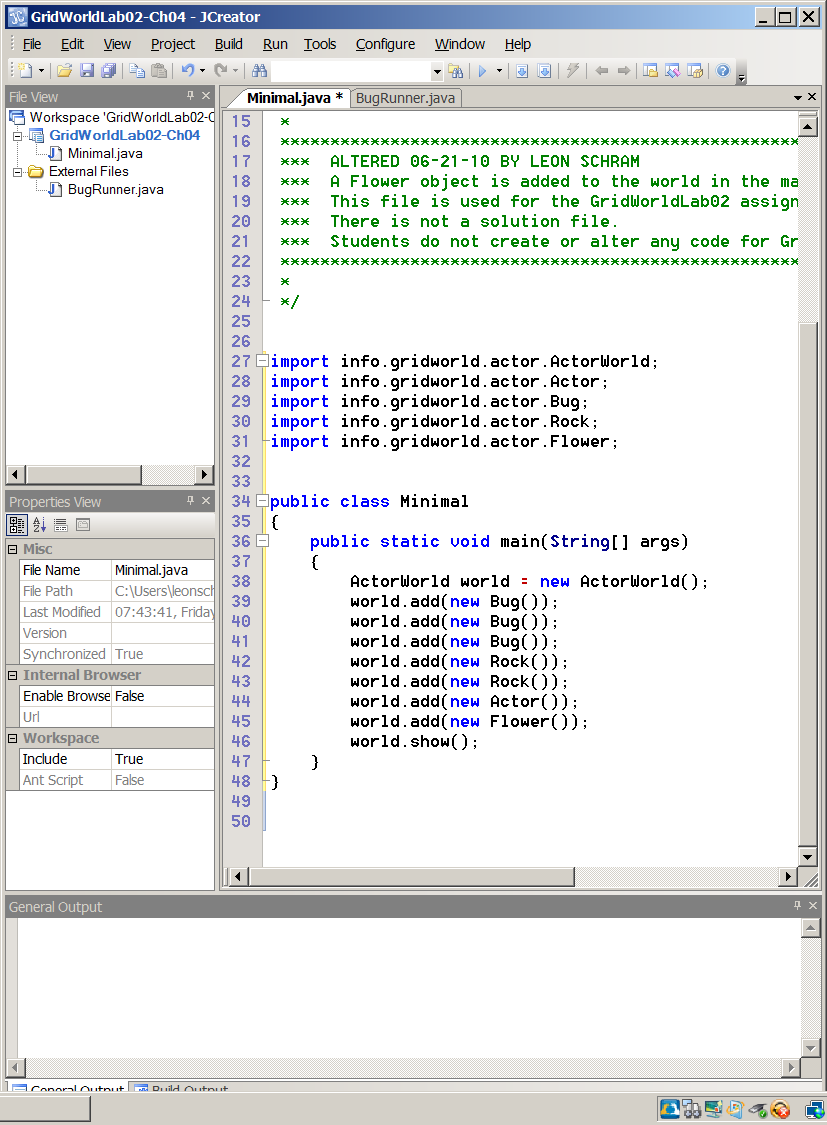
****

**New Objects at Random Locations**

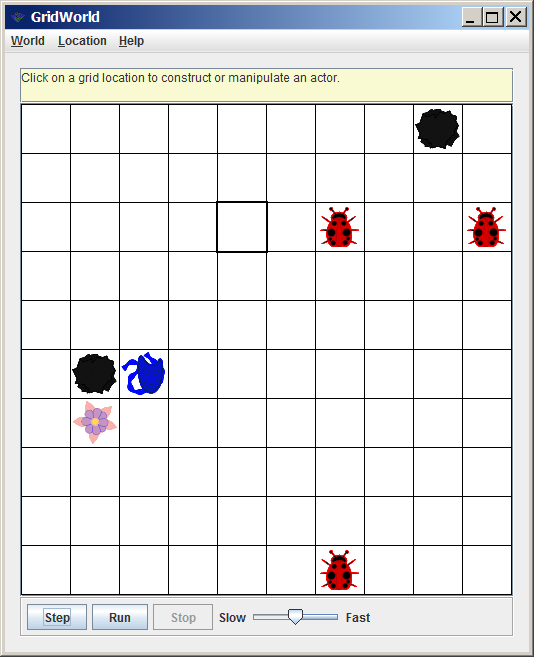
The last section started to give an idea how to add objects to the grid. Method **add** was used to add a new **Bug** object to the GridWorld. Look at figure 8.45 again and you will notice that nothing is stated about the **Bug** object. It only says **world.add(new Bug());** and nothing else. Color is not mentioned and any type of location is not mentioned. When such is the case the new object is placed at a randomly selected location on the grid.

Now we can look at another experiment and check the randomness of the objects. First, I will alter the **main** method and add various objects, shown in figure 8.47. Once again, I must be aware that the necessary import statements are added. Figure 8.48 continues and shows the execution at the start with 4 **Bug** objects, 2 **Rock** objects, 1 **Actor** object and 1 **Flower** object.

**Figure 8.47**

****

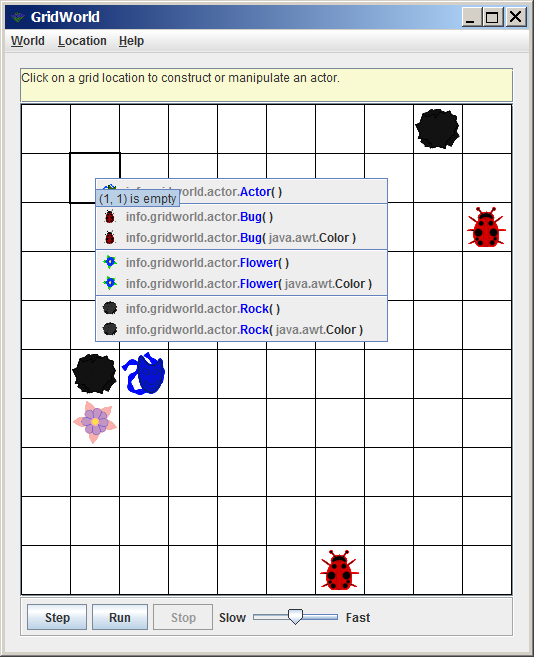
**Figure 8.48**

****

A quick click on an empty cell in figure 8.49 reveals that it is now possible to create any one of four different objects. It appears that the ability to create new objects interactively, with the display during execution, requires that the object is first added to a GridWorld object as part of the start-up execution.

You might be puzzled though, because back in figure 8.46 there was only one **Bug** object added and it was possible to create not only a **Bug** object, but also an **Actor** object. There is a logical explanation. The **Bug** class is a subclass of the **Actor** class and you can only construct a subclass object after first constructing a super class object. Sadly, this is a really correct explanation based on the concept of *inheritance*, which is the topic of the next chapter. If it does not make sense have confidence that you will be appreciate this real soon.

**Figure 8.49**

****

Now back to the random location issue. I stated that a new object added to the GridWorld will go to a random location if the location is not specified. We did see that happen in an earlier chapter, but then the statements in the **main** method were not examined as they are now.

Look at figure 8.50. Four times the program is executed, closed, and then executed again. Keep in mind that this is not the same as clicking run, stop and clicking run again. Four separate times the execution was started from scratch and the displays in figure 8.50 show the object positions at the start of a new execution. None of the objects moved to those locations.

**Figure 8.50**

|  |  |
| --- | --- |
|  |  |
|  |  |

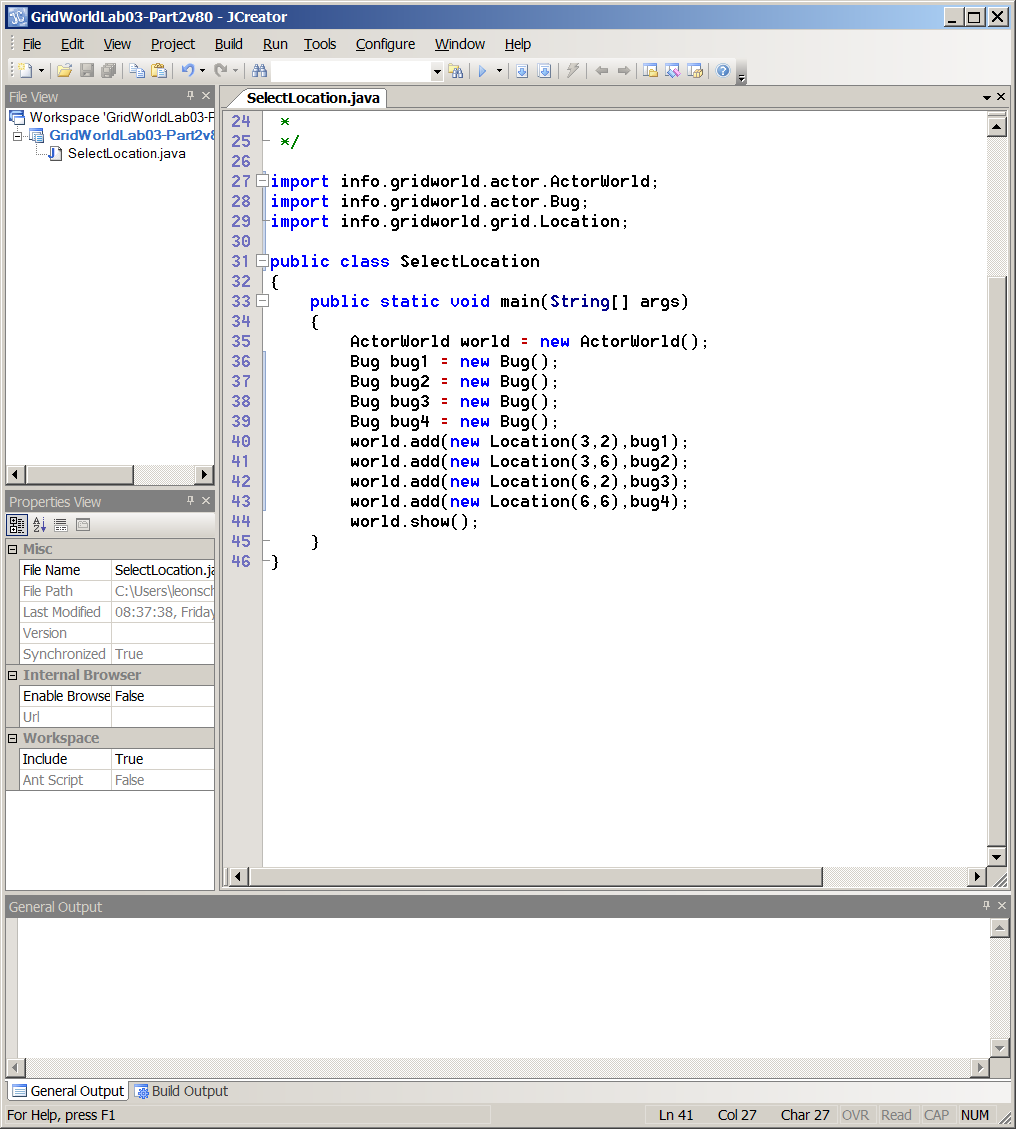
**New Objects at Specified Locations**

This time we want to place our objects at a location of our choosing. In figure 8.51 I have created, or constructed, four **Bug** objects. After the objects are constructed each one is added to the GridWorld at a specified location.

Observe that there is a new class in town, which is the **Location** class. You will also observe that the **add** method is *overloaded* and can be used multiple ways. In the last program, method **add** used only one parameter. This time two parameters are used. One is for the **Location** object and one is for the **Bug** object.

You also must be aware that the GridWorld Case Study uses anonymous objects a lot. An anonymous object is an object that is constructed and used without an identifier. In line 40 of figure 8.51 you see **bug1**, which is a *named* object. In the same line you see **new Location(3,2)**, which is an *anonymous* object.

**Figure 8.51**

****

|  |
| --- |
| **Named Objects and Anonymous Objects** |
| **Bug bug1 = new Bug();** // constructs a *named* **Bug** object  **World.add(new Bug());** // constructs an *anonymous* **Bug** object |

Figure 8.52 shows our four happy bugs at the four specified locations. Multiple start-over executions will result in the exact same startup screen as the matrix demonstrates below. Have you also realized that the GridWorld display starts at the top-left corner in location (0,0), which means *row 0* and *column 0*.

**Figure 8.52**

|  |  |
| --- | --- |
|  |  |
|  |  |

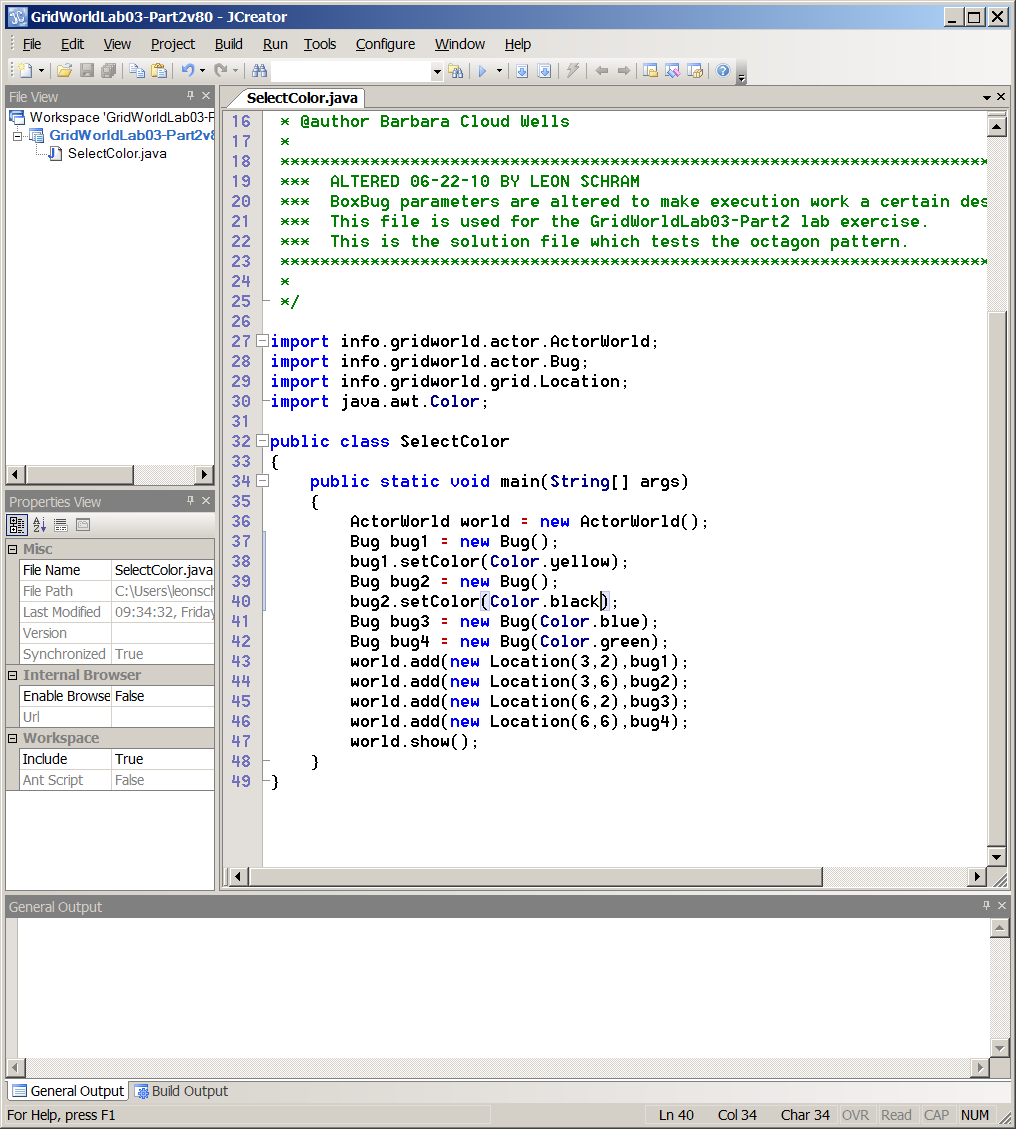
**Controlling Object Color**

You can construct new objects without color specification and you will get the default color. Bugs become **red**, rocks become **black** and actors become **blue**. It is also possible to specify a color during construction and it is possible to alter the default color after object construction.

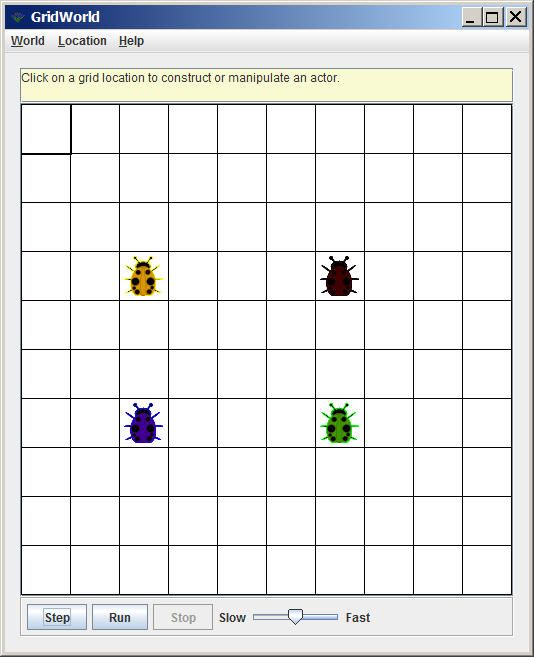
Figure 8.53 shows both approaches. The **Color** class, which is part of the **awt** package - *abstract window toolkit* - must be added to the **import** statements. First two **Bug** objects are constructed or instantiated ... remember that computer science loves to give multiple vocabulary words for the same things. Right after **bug1** and **bug2** are constructed with the default color they are changed with the **setColor** method. This is one approach.

It is also possible to specify the desired color at the construction stage. Objects **bug3** and **bug4** both call the second, overloaded, **Bug** constructor and indicate the required color of the new object. Figure 8.54 shows the start display with four bugs, each with a different color from the default red.

**Figure 8.53**



**Figure 8.54**

****

The GridWorld section in this chapter provided much information about the requirements of the main method in a GridWorld program. You will find that future lab experiments and future lab assignments require little change in any of the **main** methods.

You will primarily be concerned with creating new classes and new methods. The simple reality is that testing these new classes and methods is only possible if the main properly creates and adds the new objects to the GridWorld.