**Chapter XVI**

**Strings and Re-definingMethods**

**Chapter XVI Topics**

16.1 Introduction to String Methods

16.2 Constructing String Objects

16.3 String Method length

16.4 Working with Substrings

16.5 Converting Strings

16.6 Comparing Strings

16.7 Introduction to Redefining Methods

16.8 The Object Class

16.9 Re-defining the toString Method

16.10 Re-defining the equals Method

16.11 Summary

**16.1 Introduction to String Methods**

Strings are a set of characters in every conceivable arrangement and size. Strings are everywhere, both inside and outside the computer world. A sentence is a set of characters. A page is a set of sentences. A book is a set of pages. A library is a set of books. Given enough computer memory, an entire library can be stored in a computer. Word processing term papers, writing memoirs, sending email messages, responding to surveys, placing online orders and registering products all involve *string processing*. Every software package on the market includes string-processing components. Every programming language has special features that facilitate the manipulation of strings, and Java is no different. Finally, let us not forget that every program you write is one big collection of strings that work together, hopefully, to generate some desired and logical output.

You have actually been using the **String** data type for quite some time and to a large degree you may think that it was a *simple* or *primitive* data type. This is not surprising. Consider the variable declarations in figure 16.1.

**Figure 16.1**

|  |
| --- |
| **int** number;  **char** letter;  **double** gpa;  **boolean** finished;  **String** title; |

You see five declarations and each declaration starts with a data type followed by a variable identifier. It appears that all five of the declarations behave in the same way. You may note one peculiar difference; the **String** declaration is the only data type that starts with an *upper-case* letter. Now look at the value assignments for each one of the variables in figure 16.2.

**Figure 16.2**

|  |
| --- |
| number = 2500;  letter = 'A';  gpa = 3.785;  finished = true;  title = "Exposure Java"; |

The assignment statement seems to give secondary evidence that **String** belongs with the simple data types. By now you have seen that data structures behave quite differently from simple data types. Accessing any element in an array requires not only the array identifier, but also an index to access a specific array element. Access of any field in a record, *which is an attribute of an object in Java*, requires the object identifier and some accessing method.

There appears lots of evidence that the **String** data type is a very convenient and lovely *simple* data type. It is true that a string stores multiple characters, but after all you have treated a string like a single unit. The true nature of strings has intentionally been hidden. During the early chapters you have benefited very nicely by treating and using strings as if they were no different from the other simple data types.

Now that you have started to learn about data structures and you have learned much about Object Oriented Programming, the truth can be told. **String** is a class, which is why it starts with an upper-case letter. All Java classes start with upper-case letters.

Well you are just thrilled to hear this, but you have a fundamental question. For many chapters, many months, and many lab assignments, you have survived so nicely treating **String** as a simple data type. What benefit is there now in revealing and treating **String** like a class? This is a profound question and the answer is quite simple. The **String** class has many methods that facilitate string manipulations.

In this chapter you take a new and fresh look at strings and you will see that there are many powerful methods that will simplify your life with any type of string business. The number of available **String** methods greatly exceeds what will be presented in this chapter, but you will learn the more common **String** methods. First, what exactly is a string?

|  |
| --- |
| **String Definition** |
| A *string*is a set of characters that behaves as a single unit.  The characters in a *string* include upper-case and lower-case letters, numerical characters and a large set of characters for a variety of purposes like:  **! @ # $ % ^ & \* ( ) \_ +** |

|  |
| --- |
| String Literal Definition |
| A *string literal*is a set of characters delimited with double quotations like:  **"Seymour Snodgrass" and "SSN: 123-45-6789"** |

**16.2 Constructing String Objects**

The biggest reason why you may not have suspected that **String** is a class is due to fact that the **new** operator has been absent from our previous **String** programs. You know enough about OOP to realize that the construction, or instantiation of a new object, requires the use of the **new** operator. The **new** operator is quite busy and allocates memory for the new object along with calling the appropriate constructor. Somehow **String** has managed its affairs without using **new**.

Program **Java1601.java**, in figure 16.4, creates five **String** objects that all ultimately store and display the character string "Tango". You will see that some declarations use **new** and others declarations manage to create **String** objects quite easily without any assistance from the **new** operator.

Many program languages have certain requirements that can be omitted, and yet the same results occur. Such is the case with the construction of strings. It really is more correct to use the **new** operator, but its use is often implied. You have already seen that kind of approach with the instantiation of array objects. Consider the two array declarations in figure 16.3.

**Figure 16.3**

|  |
| --- |
| int list1 = new int[10];  int list2 = {11,12,13,14,15,16,17,18,19,20} |

Both **list1** and **list2** are integer array objects. They both have 10 elements. **list1** uses the **new** operator and specifies the size of the new object. **list2** uses an *initializer list* to indicate the size of the array and the **new**operator is implied. Personally, I would be happier if the **new** operator is used for the construction of every new object, but such is not the case. Programming languages are created by human beings, and this means special exceptions and special rules will exist.

## Figure 16.4

|  |
| --- |
| // Java1601.java  // This program demonstrates multiple ways to construct String objects.  // Note that all five string objects store the same information.    public class Java1601  {  public static void main (String args[])  {  System.out.println("\nJava1601.java\n");  String s1 = "Tango";  System.out.println("s1: " + s1);    String s2 = new String(); // Default constructor  s2 = "Tango";  System.out.println("s2: " + s2);    String s3 = new String("Tango"); // Overloaded constructor with String parameter  System.out.println("s3: " + s3);    String s4 = new String(s3); // Same constructor as s3  System.out.println("s4: " + s4);    char dance[ ] = {'T','a','n','g','o'};  String s5 = new String(dance); // Overloaded constructor with char array parameter  System.out.println("s5: " + s5);  System.out.println();  }  } |

|  |
| --- |
| **Java1601.java Output**  Java1601.java  s1: Tango  s2: Tango  s3: Tango  s4: Tango  s5: Tango |

Program **Java1601.java**, in figure 16.4, creates five **String** objects that all ultimately store the value **"Tango"**. You will see that the first **String**, **s1**, is created in the same manner that you learned way back in Chapter 3. The other 4 **String** objects, **s2**, **s3**, **s4** and **s5** are created in a very different manner. **s2** shows that the **String** class has a *default constructor*. This default constructor will construct an empty string. **s3** and **s4** show an *overloaded constructor* being used. This overloaded constructor requires a **String** parameter to initialize the **new String** object. It does not matter if the parameter is a *string variable*, as with **s4**, or a *string literal*, as with **s3**. The final **String** object, **s5**, demonstrates that a completely different overloaded constructor exists. This one uses an array of characters. There are actually many more overloaded **String** constructors available, but I believe the point, that **String** is a **class**, has been made.

**16.3 String Method length**

The first **String** method is **length**, which returns the numbers of characters in a string. Program **Java1602.java**, in figure 16.5, demonstrates **length** with three different objects. In particular, look at **s3** and see if the space is counted as a character.

**Figure 16.5**

|  |
| --- |
| // Java1602.java// This program demonstrates the use of the <length> method.public class Java1602{ public static void main (String args[]) {  System.out.println("\nJava1602.java\n");  String s1 = "Argentine"; String s2 = "Tango"; String s3 = s1 + " " + s2;  System.out.println(s1 + " has " + s1.length() + " characters.");  System.out.println(s2 + " has " + s2.length() + " characters.");  System.out.println(s3 + " has " + s3.length() + " characters.");  System.out.println();  }  } |

|  |
| --- |
| **Java1602.java Output**  Java1602.java  Argentine has 9 characters.  Tango has 5 characters.  Argentine Tango has 15 characters. |

The output of **Java1602.java** proves that a space is indeed counted as a character. Invisible characters are called *white space* characters, but they are characters nevertheless and need to be considered with string processing.

Java is a programming language that was not created by a single person. It is a language created by a large team of programmers. A team is necessary to create a complex language, like Java, but it also give you some odd quirks, where you get a feeling that one part of the team did not communicate with another part of the team. I am talking about this **length** method and similar methods.

Java has three data structures with a method to return the number of elements in the object. The static array uses the **length** field, the dynamic **ArrayList** uses the **size** method and the **String** uses the **length** method.

|  |
| --- |
| **String method length** |
| **int count = str.length();**  Method **length** returns the *length* or number of characters in the **String** object.  If **str** equals **"Aardvark"** then **count** becomes **8**. |
| The Java static array uses the **length** field.  The dynamic **ArrayList** class uses the **size** method. |

**16.4 Working with Substrings**

Now that we can determine the number of characters in a string with **length**, we are ready to access individual characters. Students with prior knowledge on other programming languages, such as C++, may think that a string is a character array. You may be tempted to use some statement like **c = s1[k];** and think that should work nicely. Please keep in mind that Java strings are not character arrays.

It is possible to construct substrings of larger strings by traversing a string and accessing a specified range of characters. Such an approach works, and you could create your very own substring method. Now you can still charge ahead and make such a method, but Java already has a **substring** method. Please note that it is **substring** and not **subString**, which is what I would have expected following the lower/upper case convention of Java.

The first Java **substring** method uses two parameters: one to indicate the first index and one to indicate the last index. At least a variety of Java reference sources claim that to be the case. My personal experience is that the second parameter is actually one more than the last index. It behaves more like an upper bound than an actual index.

Program **Java1603.java**, in figure 16.6, shows several **substring** examples with the same original string. There are six **substring** commands. Each command has different integer parameters, and therefor displays a different part of the original string. Pay close attention to the two parameters and the actual **substring** that is retrieved for each of the six commands.

**Figure 16.6**

|  |
| --- |
| // Java1603.java// This program demonstrates how to access specified characters of// a string with the <substring(SI,EI)> method, where SI is the StartIndex and// EI is one less than the EndIndex. public class Java1603{ public static void main (String args[])  {  System.out.println("\nJava1603.java\n");  String s = "Racecar";  System.out.println(s.substring(0,4));  System.out.println(s.substring(1,4));  System.out.println(s.substring(2,4));  System.out.println(s.substring(2,6));  System.out.println(s.substring(3,6));  System.out.println(s.substring(4,7));  System.out.println();  }  } |

|  |
| --- |
| **Java1603.java Output**  Java1603.java  Race  ace  ce  ceca  eca  car |

|  |
| --- |
| **String method-1 substring (substring is overloaded)** |
| **s1 = "aardvark";**  **s2 = s1.substring(j,k);**  Method **substring** returns a set of consecutive characters from string **s1**, starting at index **j**, and ending at index **k-1**.  **s3 = s1.substring(4,7);**  **s3** becomes **"var"** |

Java has a second substring method that is less frequently used with a single parameter. This parameter indicates the starting index used to build the substring. It is not necessary to provide a second parameter. This second substring always goes to the end of the string. Program **Java1604.java**, in figure 16.7, has two loops using the same **"Racecar"** string. The first loop uses **substring** with a single parameter to the end of the string. The second loop creates the same results with the first **substring** method. The second parameter is fixed at the length of the source string.

If you realize that the length of a string is one count higher than the final index then you see that the second **substring** method really has the same logic as the first **substring** method. It is almost like there is an invisible second parameter that is fixed at the length of the string.

**Figure 16.7**

|  |
| --- |
| // This program compares the two <substring> methods.  public class Java1604  {  public static void main (String args[])  {  System.out.println("\nJava1604.java\n");  String s = "Racecar";  int n = s.length();  for (int k = 0; k < n; k++)  System.out.println(s.substring(k));  System.out.println();  for (int k = 0; k < n; k++)  System.out.println(s.substring(k,n));  System.out.println();  }  } |

**Figure 16.7 Continued**

|  |
| --- |
| **Java1604.java Output**  Java1604.java  Racecar  acecar  cecar  ecar  car  ar  r  Racecar  acecar  cecar  ecar  car  ar  r |

|  |
| --- |
| **String method-2 substring (substring is overloaded)** |
| **s1 = “Aardvark”;**  **s2 = s1.substring(j);**  Method **substring** returns a set of consecutive characters from **String s1,** starting at index **j**, and continuing all the way to the end of the string.  **s3 = s1.substring(4);**  **s3 becomes "vark"** |

There is more that can be done with substrings. You have just finished specifying a *startindex* and *endindex* within an existing string to get a desired substring. It is also possible to go into the opposite direction. This means that you start with a specified substring and determine if it exists in another string, and if so, where?

Method **indexOf** behaves like the **find** function of a word processor. The method returns the index of the first occurrence of the substring. Program **Java1605.java**, in figure 16.8, uses **"car"** as the search string. First a search is done in string **"racecar"**, where there is only a first occurrence of the substring. Second, **"car"** is used in **"racecar in the carport"**, which has two occurrences of the substring. Third, **"car"** is used with the **"qwerty"** string and will return **-1** since the substring is not found.

**Figure 16.8**

|  |
| --- |
| // Java1605.java  // This program shows the <indexOf> method, which returns the index of the first  // occurrence of the string argument or -1 if the string is not found.  public class Java1605  {  public static void main (String args[])  {  System.out.println("\nJava1605.java\n");  String s1 = "racecar";  String s2 = "racecar in the carport";  String s3 = "car";  int index1 = s1.indexOf(s3);  int index2 = s2.indexOf(s3);  int index3 = s3.indexOf("qwerty");  System.out.println("With \"" + s1 + "\" car starts at " + index1);  System.out.println("With \"" + s2 + "\" car starts at " + index2);  System.out.println("With \"" + s3 + "\" Qwerty shows up at " + index3);  System.out.println();  }  } |

|  |
| --- |
| **Java1605.java Output**  Java1605.java  With "racecar" car starts at 4  With "racecar in the carport" car starts at 4  With "car" Qwerty shows up at -1 |

**16.5 Converting Strings**

In this section we will examine how to change strings into other data types and how to change other data types into strings. Many program languages seem to prefer string keyboard entry and some conversion is necessary to allow numerical computation.

Program **Java1606.java**, in figure 16.9, constructs four strings and each string uses the **valueOf** method. This method is overloaded and the program example shows how to convert an *integer*, *double*, *boolean* and *character* data type. Method **valueOf** is a *static* method, which is evidenced by the **String** class identifier rather than an object identifier.

**Figure 16.9**

|  |
| --- |
| // Java1606.java// This program demonstrates the <valueOf> method of the String class,  // which is shown to convert four data types to a string.  // Note that <valueOf> is a static method and must be called using <String.valueOf>.  public class Java1606  {  public static void main (String args[])  {  System.out.println("\nJava1606.java\n");  String s1 = String.valueOf(1000);  String s2 = String.valueOf(123.321);  String s3 = String.valueOf(true);  String s4 = String.valueOf('A');  String s5 = s1 + s2;  System.out.println("s1: " + s1);  System.out.println("s2: " + s2);  System.out.println("s3: " + s3);  System.out.println("s4: " + s4);  System.out.println("s5: " + s5);  System.out.println();  }  } |

|  |
| --- |
| **Java1606.java Output**  Java1606.java  s1: 1000  s2: 123.321  s3: true  s4: A  s1: 1000123.321 |

|  |
| --- |
| **String static method valueOf** |
| **String s1 = String.valueOf(1000);**  **String s2 = String.valueOf(123.321);**  **String s3 = String.valueOf(true);**  **String s4 = String.valueOf('A');**  Method **valueOf** converts the provided parameter and returns  a string. Four overloaded **valueOf** methods are displayed.  Note that the **valueOf** method is a **static** method that is called with the **String** class identifier. |

Data from the keyword with **readLine** in a text window and data entered in a GUI window are all strings. For many processes this presents no problem. On the other hand, any type of numerical data that requires computation will need to be converted from a string to an integer or a double. Program **Java1607.java**, in figure 16.10, demonstrates both the **parseInt** and the **parseDouble** methods. These methods are shown in this string processing chapter, as well they should, because strings are very much used. However, these two conversion methods are not in the **String** class. They belong to the **Integer** class and the **Double** class. Do not get confused with the simple data types, **int** and **double**. **Integer** and **Double** are bona fide classes, as the starting upper-case letter indicates.

**Figure 16.10**

|  |
| --- |
| // Java1607.java// This program converts string values to integer and double values using// the <parseInt> and <parseDouble> methods of the <Integer> and <Double> classes.  public class Java1607  {  public static void main (String args[])  {  System.out.println("\nJava1607.java\n");  String s1 = "12345";  String s2 = "123.321";  int n1 = Integer.parseInt(s1);  double n2 = Double.parseDouble(s2);  System.out.println(n1 + " + " + n1 + " = " + (n1 + n1));  System.out.println(n2 + " + " + n2 + " = " + (n2 + n2));  System.out.println();  }  } |

|  |
| --- |
| **Java1607.java Output**  Java1607.java  12345 + 12345 = 24690  123.321 + 123.321 = 246.642 |

|  |
| --- |
| **Integer parseInt and Double parseDouble methods** |
| **int n1 = Integer.parseInt(s1);**  **double n2 = Double.parseDouble(s2);**  Method **parseInt** converts a string into an integer.  Method **parseDouble** converts a string into a double. |

**16.6 Comparing Strings**

This section will start by comparing the equality of two strings. Two strings **"Foxtrot"** and **"Waltz"** are literal strings coded into the program. These two strings will be compared to string **"Foxtrot"** that will be entered at the keyboard during executing program **Java1608.java**, in figure 16.11. Would you expect that both strings are considered not equal?

**Figure 16.11**

|  |
| --- |
| // Java1608.java  // This program checks equality of strings using the == operator.  // This program has unexpected results.  import java.util.Scanner;  public class Java1608  {  public static void main (String args[])  {  System.out.println("\nJava1608.java\n");  Scanner input = new Scanner(System.in);  String s1 = "Foxtrot";  String s2 = "Waltz";  System.out.print("Enter a string ===>> ");  String s3 = input.nextLine();  if (s1 == s2)  System.out.println(s1 + " == " + s2);  else  System.out.println(s1 + " != " + s2);  if (s1 == s3)  System.out.println(s1 + " == " + s3);  else  System.out.println(s1 + " != " + s3);  System.out.println();  }  } |

|  |
| --- |
| **Java1608.java Output**  Java1608.java  Enter a string ===>> Foxtrot  Foxtrot != Waltz  Foxtrot != Foxtrot |

Program **Java1609.java**, in figure 16.12, uses the **equals** method to compare two strings. The equals operator **==** is meant for simple data types, like **int**, **char**, **double** and **boolean**. The **equals** method is designed to check object equality. This is yet another example that **String** is not a simple type, but a class.

**Figure 16.12**

|  |
| --- |
| // Java1609.java// This program demonstrates the <equals> method, which is capable of // testing equality of string objects correctly. import java.util.Scanner;  public class Java1609  {  public static void main (String args[])  {  System.out.println("\nJava1609.java\n");  Scanner input = new Scanner(System.in);  String s1 = "Foxtrot";  String s1 = "Foxtrot";  String s2 = "Waltz";  System.out.print("Enter a string ===>> ");  String s3 = input.nextLine();    if (s1.equals(s2))  System.out.println(s1 + " == " + s2);  else  System.out.println(s1 + " != " + s2);  if (s1.equals(s3))  System.out.println(s1 + " == " + s3);  else  System.out.println(s1 + " != " + s3);  System.out.println();  }  } |

|  |
| --- |
| **Java1609.java Output**  Java1609.java  Enter a string ===>> Foxtrot  Foxtrot != Waltz  Foxtrot == Foxtrot |

It is easy to state that the == operator checks equality for simple data types and the **equals** method checks equality for class objects. Does that make any sense? After all, isn't equality the same equality no matter what values are compared? That may be a logical question, but the problem is in the comparing.

When you look at two numbers, like 100 and 100 it seems easy to determine equality. Likewise look at **"Foxtrot"** and **"Foxtrot"** and there appears no difficulty in seeing that those are identical string values. The problem here is that you look at these values like a person and the computer does not see with eyes and compares something entirely different than a person.

Primitive data types store values and objects store references to the memory locations where values are stored. It helps to view these memory locations as storing *shallow* or *immediate* values and *deep* values.

Look at the diagram in figure 16.13 where you see two memory locations. That memory is allocated for two **int** variable **n1** and **n2.** At each one those two memory locations the same **1000** is stored. The statement **if (n1 == n2)** checks to see what values are stored at **n1** and **n2** and compares them. We can say that the *shallow* or *immediate* values are compared.

**Figure 16.13**

**n1 (int) n2 (int)**

1000

1000

It is a different story for **String** objects. Figure 16.14 shows greater complexity in storing the **String** values. The immediate values of **s1** and **s3** are not a **String** value, but a memory reference. This memory reference is the deeper memory location where the actual **String** values of **"Foxtrot"** are stored.

When the statement **if (s1 == s3)** is used, Java checks the equality of the immediate values of **s1** and **s3** just like it was done with **n1** and **n2**. The immediate values are a base-16 memory reference and **dff6ccd** and **601bb1** are not equal. This is what the computer sees, compares and then concludes that **s1** is not equal to **s3.**

The **equals** method has been designed to ignore the shallow values and compare the deep values of **s1** and **s3**, which is where values **"Foxtrot"** are stored. The result is that now the equality is properly evaluated.

**Figure 16.14**

**s1 s3**

**dff6ccd 601bb1**

@601bb1

@dff6ccd

**Foxtrot**

**Foxtrot**

|  |
| --- |
| **String equals method** |
| **if (s1.equals(s2))**  Method **equals** returns **true** if **s1** equals **s2**, and **false** otherwise. |

The Java **String** class has a **compareTo** method, which is similar to the **equals** method. Both methods determine equality, but the difference is that **compareTo** is able to indicate the relative distance between strings that are not equal. You may find *distance-between-strings* a rather peculiar concept. The relative distance is an integer value based on the difference of the character values. This means that equal strings have a distance of **0**. Strings that start with **a** and **b** have a distance of **1** and strings that start with **A** and **Z** have a distance of **25**.

The **compareTo** method also has the ability to identify whether a compared string is greater or lesser and uses a negative sign to indicate lesser strings. Program **Java1610.java**, in figure 16.18, will help to clarify this negative business.

**Figure 16.18**

|  |
| --- |
| // Java1610.java  // This program demonstrates the <compareTo> method, which returns an integer value.  // The returned value indicates which string alphabetically goes before the other.  // If the value is negative, the original string goes first.  // If the value is positive, the parameter string goes first.  // If the value is zero, both strings are equal.  public class Java1610  {  public static void main (String args[])  {  System.out.println("\nJava1610.java\n");  String s1 = "AARDVARK";  String s2 = "ZEBRA";  String s3 = "AARDVARK";  String s4 = "BART";  int value1 = s1.compareTo(s2);  int value2 = s1.compareTo(s3);  int value3 = s2.compareTo(s1);  int value4 = s1.compareTo(s4);  System.out.println("value1: " + value1);  System.out.println("value2: " + value2);  System.out.println("value3: " + value3);  System.out.println("value4: " + value4);  System.out.println();  }  } |

**Figure 16.18 Continued**

|  |
| --- |
| **Java1610.java Output**  Java1610.java  Value1: -25  Value2: 0  Value3: 25  Value4: -1 |

The **compareTo** method checks its own **String** relative to the **String** argument. If its own string value is less (alphabetically speaking) than the string argument, the integer returned is a negative value, and otherwise it is positive.

|  |
| --- |
| **String compareTo method** |
| **int distance = s1.compareTo(s2);**  Method **compareTo** returns **0** if **s1** equals **s2**,  otherwise an integer is returned based on the difference between **s1** and **s2**.  If **s1 < s2**, the returned value is negative.  If **s1 > s2**, the returned value is positive. |

|  |
| --- |
| **AP Computer Science Examination Alert** |
| Java has a large selection of String methods and some of them were shown in this chapter.  Not all of these methods will be tested. Only the following methods are part of the AP Java testing Subset:  **compareTo - equals - length - substring - indexOf** |

**16.7 Introduction to Re-defining Methods**

This chapter has combined two earlier, short chapters. The topics are related, but they are different. For this reason you will now have a second introductory section that used to be the first introduction of a chapter.

There are some mysteries in the Java programming language. For example, how is it that it is possible to use the **print** method with an **ArrayList** object and a **String** object and the array members are all printed or the **String** is printed? On the other hand, the same approach with a static array does not work. It requires a loop to display every member of a static array. There is a similar problem when you create your own class. You cannot simply use **print** to display the values stored in an object of your class. Or can you?

This is a short, but vitally important, chapter. The mystery of output display will be explained and you will learn how to create your own class in such a manner that any desired values will be displayed using **print** without benefit of any special loop structure.

There seems a similar mystery in the areas of equality checking. Some variables, like simple data types can be checked for equality with the equality operator and others, like **String** objects require the use of the **equals** method. The question once again about user-defined classes pops up. Can user-defined objects be tested for equality, and if so how is this done?

It turns out that there are two methods, **toString** and **equals** that have been defined for many Java standard classes. These methods can also be defined for any user-defined classes.

Methods **toString** and **equals** are short, humble methods, but they carry a punch. User-defined classes can be personalized with the aid of these two methods for very specific properties and convenient of use of the new classes.

You may recall from studying the GridWorld Case Study that at the conclusion of the **Actor** class is a **toString** method. You also see a **toString** method in the **AbstractGrid** class. In the GridWorld section it was stated that method **toString** assisted in debugging programs. As a rule **toString** is written such that the display of an object will display the values of its attributes. Knowing the values of an object's attributes is an aid in debugging programs.

**16.8 The Object Class**

There exists an invisible class that is present in every program you write. This magical class is the superclass of all the Java standard library classes. It is the superclass of any class that you have created and it will be the superclass to any class that you create in the future. Program **Java1611.java**, in figure 16.19, shows a very short program with two classes. There is nothing practical about this program. Do note that both classes extend the **Object** class.

**Figure 16.18**

|  |
| --- |
| // Java1611.java  // This program intentional extends the two classes with  // Object as the superclass. This is done automatically.  import java.util.ArrayList;  public class Java1611 **extends Object**  {  public static void main (String args[])  {  System.out.println("\nJava1611.java\n");  Qwerty q = new Qwerty();  }  }  class Qwerty **extends Object**  {  public Qwerty()  {  System.out.println("Constructing Qwerty Object");  }  } |

|  |
| --- |
| **Java1611.java Output**  Java1611.java  Constructing Qwerty Object |

Personally, I am not real thrilled with the name **Object** for a class. In an earlier chapter there were many examples, using GridWorld, that showed the difference between a class and an object. A class is a category and an object is one instance of the category. **Student** is a class and **tom** is one instance of **Student**. So now we have a class, which cleverly is named Object. Please do not get confused. Also do not use the syntax shown in program **Java1611.java**. You do not write a program and extend the Object class. That is a done deal. It was only shown so that you see that Java is not at all excited about this **Object** class.

The impact of a single class that is the superclass to any and all classes is quite profound. This means that any class that you ever see anywhere has excess to the method definitions that exist in the **Object** class. Like all subclasses there is the option to re-define any superclass method. However, if a superclass method is not re-defined you will get the version that is defined in the superclass. The Object class has a fair set of method, but we are only concerned with two specific methods, called **toString** and **equals**. We will return to these methods shortly, but first some other program examples need to be presented to help clarify this Object story better.

You have taken the humble **print** and **println** methods for granted for many months. It is such an easy set of methods to use. Everything inside the parentheses is displayed in a nice concatenated fashion. Anything placed between double quotes is displayed literally and variables are displayed according to the value they hold. At least that seems to be the case. You may have been surprised by the output of an **ArrayList** object that was rather casually shoved inside the **println** parentheses. Every one of the object members was neatly, and individually displayed, complete with comma separation.

We will start by investigating the output of various **print** environments and then explain how **print** actually performs its job. Program **Java1612.java**, in figure 16.20, uses **print** with a **String** object and a literal string. The literal string is displayed precisely as it is seen between the quotes and the **String** variable displays the value that it stores.

**Figure 16.20**

|  |
| --- |
| // Java1612.java  // This demonstrates how <String> class objects are printed.  public class Java1612  {  public static void main (String args[])  {  System.out.println("\nJava1612.java\n");  String stringVar = "Tango";  System.out.println(stringVar);  System.out.println();  System.out.println("Literal String");  System.out.println();  }  } |

|  |
| --- |
| **Java1612.java Output**  Java1612.java  Tango  Literal String |

Program **Java1613.java**, in figure 16.21, uses **println** with each one of four different primitive Java data types, **int**, **double**, **char** and **boolean**. Four appropriate values are assigned to the variables and then displayed with **println**. In each case the value assigned to the variable and stored in its allocated memory space is displayed. There are no surprises here.

**Figure 16.21**

|  |
| --- |
| // Java1613.java  // This demonstrates how <int>, <double>, <char> and <boolean>  // variables are printed.  public class Java1613  {  public static void main (String args[])  {  System.out.println("\nJava1613.java\n");  int intVar = 100;  double dblVar = 3.14159;  char chrVar = 'A';  boolean blnVar = true;  System.out.println(intVar);  System.out.println(dblVar);  System.out.println(chrVar);  System.out.println(blnVar);  System.out.println();  }  } |

|  |
| --- |
| **Java1613.java Output**  Java1613.java  100  3.14159  A  true |

Now we will switch to something unusual. We will use **println** with three different static arrays. You have certainly used **println** many times with static arrays, but always with individual elements of an array variable using the **[ ]** index operators. This times we are bluntly placing the actual array variable inside the **println** parentheses and check to see what happens. Will that even compile? I know it compiles and those of you who read this chapter will also know that it compiles in preparation for the discussion on this topic. Program **Java1614.java**, in figure 16.25, features **intList** constructed as an **int** array of 9 integers. There is the **dblList** object constructed as a **double** array of 9 real numbers and the **strList** object, constructed as a **String** array of 4 strings. Objects **intList**, **dblList** and **strList** are each displayed by the **println** method. Check out the result.

**Figure 16.25**

|  |
| --- |
| // Java1614.java  // This program demonstrates how Java <int> and <double> arrays are  // displayed when only the object identifier is printed.  public class Java1614  {  public static void main (String args[])  {  System.out.println("\nJava1614.java\n");  int intList[] = {11,22,33,44,55,66,77,88,99};  System.out.println("intList: " + intList);  double dblList[] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};  System.out.println("dblList: " + dblList);  String strList[] = {"Tom","Joe","Sue","Meg"};  System.out.println("strList: " + strList);  System.out.println();  }  } |

|  |
| --- |
| **Java1614.java Output**  Java1614.java  intList: [I@720eeb  dblList: [D@3179c3  strList: [Ljava.lang.String;@310d42 |

The output display starts with a left-index operator bracket followed by a letter and then some hexadecimal numbers that are computer memory addresses. Earlier in this chapter you saw how there are shallow memory locations and deep memory locations. Primitive data types store the variables at the shallow memory addresses. Objects store a memory reference where the values are stored. Figure 16.26 shows a diagram of how this may appear in memory.

**Figure 16.26**

**intList dblList strList**

**@720eeb @3179c3 @310d42**

**@720eeb**

**@310d42**

**@3179c3**

**Tom**

**1.1**

**11**

Each of the array objects starts with a memory location where the reference is stored. This reference is the first memory location of the *contiguous* block of memory where the actual object data is stored. This means that **intList** stores address **720eeb** where **11** is stored. Only the first array element is shown in each case. Object **dblList** stores **3179c3**, which is the start of **1.1** and the other array elements. Finally there is **strList**, which stores **310d42**, where **Tom** is the first string array element.

What seems to be happening is that a statement like **System.out.println(intList);** displays the actual value stored at **intList**, and that is **@720eeb**. Now the **println** method also adds **[I** , which indicates the class of the object that is stored. In this case the square bracket of the array index operator indicates that this is a static array object and the **I** is an abbreviation for the **int** values that are stored. When simple data types are stored there is only a letter. For classes the path to the class and complete class name is printed.

It is now time to return to our **Object** class. Just a quick review to refresh your memory. Every class in Java derives from the **Object** class. **Object** is the ultimate superclass at the top of all the Java classes. Normally, when you work with a subclass of some superclass, you will see a program statement like **public class Car extends Vehicle.** It is not necessary to state that your class extends **Object**, Java knows this and assumes this to be the case.

The **Object** has a variety of methods that gets Java jump-started. One of these methods is **toString**. The **toString** method is a fascinating method, because you do not see the name anywhere. This modest method lurks in the background and waits for a **println** statement to be executed. The **print** and **println** methods display a String representation of its parameters, but what type of representation will this be? **print** and **println** do not decide what needs to be displayed. Rather our display methods check with the **toString** method to see what must be done with the parameters. **toString** is happy to help and returns the appropriate string representation that needs to be displayed. If this makes sense then we need to ask what instructions are provided by the "hidden" **toString** method. The instructions are started back in the original **Object** classand those instructions basically say to display the value stored at the *shallow* memory location of the variable.

The result of this action is that **toString** returns the string representation of **int**, **dbl**, **char** and **boolean** variables. Primitive data types store their singular values directly in the shallow memory location. Objects store references in the shallow value and that is what **toString** returns along with the name of the object's class.

Our hardworking **toString** method does struggle a little with static arrays, because they do not get a class name. Program **Java1615.java**, in figure 16.27, declares a **Student** class and then constructs four **Student** objects. Each one of the four s**tudent** objects is displayed by the **println** method. Let us see how **toString** handles this type of situation.

**Figure 16.27**

|  |
| --- |
| // Java1615.java  // In this program the user-defined <Student> class uses the <toString>  // method inherited from the <Object> class. The <Object> class <toString>  // method returns an actual string representation of the object value,  // which is a memory address.  public class Java1615  {  public static void main (String args[])  {  System.out.println("\nJava1615.java\n");  Student student1 = new Student("Tom",21,3.85);  Student student2 = new Student("Joe",17,3.65);  Student student3 = new Student("Sue",18,2.85);  Student student4 = new Student("Meg",19,3.90);  System.out.println("student1: " + student1);  System.out.println("student2: " + student2);  System.out.println("student3: " + student3);  System.out.println("student4: " + student4);  System.out.println();  }  }  class Student  {  private String name;  private int age;  private double gpa;    public Student(String n, int a, double g)  {  name = n;  age = a;  gpa = g;  }  } |

|  |
| --- |
| **Java1615.java Output**  Java1615.java  student1: Student@3179c3  student2: Student@310d42  student3: Student@5d87b2  student4: Student@77d134 |

**Student** is a class and its objects store references. Precisely according to the instructions started in the **Object** class the value stored by the variable is displayed and **toString** throws in the class name **Student** for free.

|  |
| --- |
| **The Original toString Method** |
| **print** and **println** request display instructions from the **toString** method. Method **toString** is defined by the **Object** class. The **Object** class is the superclass for all Java classes. This means that every class has access to the **toString** method.  The **toString** method, as defined by the **Object** class, returns the actual string representation values of all the primitive types like **int**, **double**, **char** and **boolean**.  **toString** returns the class name followed by the memory reference of any variable object. |

**16.9 Redefining the toString Method**

Everything you have seen thus far about **print** and **toString** does not explain the display of **ArrayList** objects you witnessed in a previous chapter. There you saw the actual, individual array elements displayed and not some kind of memory address. Furthermore, the individuals array elements were neatly displayed, separated by commas and placed inside a set of square brackets.

Program **Java1616.java**, in figure 16.28, displays the **ArrayList** object **names** and you will notice that five individual array elements are displayed. There is no memory location in sight. The output of this program does not follow the instructions that **toString** received inside the **Object** class.

It is certainly convenient to display all the elements of a data structure with a single **System.out.print** statement without the use of any loop structure. However, this is not what the **Object** class specified and it now appears that the instructions of the **toString** can be altered for some specific purpose.

**Figure 16.28**

|  |
| --- |
| // Java1616.java  // This program demonstrates <toString> behaves differently for the <ArrayList>  // class. Note that "referenced" values are displayed, not the references.  import java.util.ArrayList;  public class Java1616  {  public static void main (String args[])  {  System.out.println("\nJava1616.java\n");  ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");  System.out.println("names: " + names);  System.out.println();  }  } |

|  |
| --- |
| **Java1616.java Output**  Java1616.java  names: [Isolde, John, Greg, Maria, Heidi] |

Now we need to get something straight about *inheritance*. It has always been established that subclasses have access to methods of the superclass, but nowhere was it required that superclass methods have to be used. In the case of **toString** you may simply not care for the manner in which **toString** does its job according to the **Object** class. That is just fine and you have the right to create your own methods or redefine any existing methods of the superclass.

In the case of output display you cannot create your own methods. The **print** and **println** methods still look for instructions from the **toString** method, regardless of what your subclass does. This means that **toString** needs to be re-defined if you are not happy with the superclass results. This re-defining is exactly what happened to **ArrayList** and you see the results.

Now we are going to try a little experiment. Program **Java1617.java**, in figure 16.29, once again uses an **ArrayList** object. This time the elements of the object will not be names, but rather **Student** objects. How will **toString** handle this situation?

**Figure 16.29**

|  |
| --- |
| // Java1617.java.java  // The <toString> method of an ArrayList object displays the contents of the <ArrayList>  // object, but the non-Java class <Student> objects display memory addresses.  import java.util.ArrayList;  public class Java1617  {  public static void main (String args[])  {  System.out.println("\nJava1617.java\n");  ArrayList students = new ArrayList();  students.add(new Student("Tom",21,3.85));  students.add(new Student("Joe",17,3.65));  students.add(new Student("Sue",18,2.85));  students.add(new Student("Meg",19,3.90));  System.out.println("students: " + students);  System.out.println();  }  }  class Student  {  private String name;  private int age;  private double gpa;    public Student(String n, int a, double g)  {  name = n;  age = a;  gpa = g;  }  } |

|  |
| --- |
| **Java1617.java Output**  Java1617.java  students: [Student@310d42, Student@5d87b2, Student@77d134, Student@47e553] |

The output looks like some kind of strange hybrid display between the **ArrayList** format and the original **Object** format. Actually, this is very logical. The **students** object does display four individual array elements, separated by commas just like **ArrayList** has done before.

However, **students** is an object of objects and the object at each array element obeys the rules of its own **toString** method. In this case the individual elements are **Student** objects and they display just like they did in the last program with the class name and a memory reference stored at the shallow address location.

|  |
| --- |
| **ArrayList and toString** |
| **ArrayList** objects display the actual individual array elements and not the memory reference of the object.  The output format looks like [Tom, Sue, Joe, Kathy].  This means that the **toString** method is *re-defined* for the  **ArrayList** class or some *superclass* of **ArrayList**. |

The time has come to take matters into our own hands. We have used **toString** as it is defined by the **Object** class. We have also used **toString** in a re-defined manner as we witnessed with **ArrayList** objects. Now we will re-define **toString** to suit our own purpose. Program **Java1618.java**, in figure 16.30, uses the **Student** class again and this time method **toString** is added to the **Student** class. For many months now you know that access to an object method is gained by using *object-identifier ... dot-operator ... method identifier*. We have added **toString** so now it must be possible to call that method. The definition of the **toString** method returns **name**, which is a **String** and completely digestible by the **println** method.

The result of this improved **Student** class is a display of the names stored by each **Student** object. You may not be very impressed because something like a normal OOP getmethod, **getName**, would certainly perform the exact same job that we observe **toString** doing.

**Figure 16.30**

|  |
| --- |
| // Java1618.java  // This program demonstrates defining a <toString> method  // in the <Student> class so that it displays the name field of a <Student> object.  import java.util.ArrayList;  public class Java1618  {  public static void main (String args[])  {  System.out.println("\nJava1618.java\n");  Student student1 = new Student("Kathy Alexander",21,3.75);  Student student2 = new Student("Peter VanVliet",18,2.265);  **System.out.println(student1.toString());**  **System.out.println(student2.toString());**  System.out.println();  }  }  class Student  {  private String name;  private int age;  private double gpa;    public Student(String n, int a, double g)  {  name = n;  age = a;  gpa = g;  }    **public String toString()**  **{**  **return name;**  **}**    } |

|  |
| --- |
| **Java1618.java Output**  Java1618.java  Kathy Alexander  Peter VanVliet |

Program **Java1619.java**, in figure 16.31, may clarify the point of this business. In this program example the **Student** class declaration is completely identical to the previous program. The **main** method is almost identical to the previous program. There is one slight difference. The **toString** method is no longer called. The **Student** object is directly placed inside the **println** method and the results are identical to the output of the previous program.

**Figure 16.31**

|  |
| --- |
| // Java1619.java  // This program is almost identical to the previous program.  // This time the <toString> method is not called by any <Student> object.  // Yet the result is the same because <println> uses the string representation of <toString> for its output.  import java.util.ArrayList;  public class Java1619  {  public static void main (String args[])  {  System.out.println("\nJava1619.java\n");  Student student1 = new Student("Kathy Alexander",21,3.75);  Student student2 = new Student("Peter VanVliet",18,2.265);  **System.out.println(student1);**  **System.out.println(student2);**  System.out.println();  }  }  class Student  {  private String name;  private int age;  private double gpa;    public Student(String n, int a, double g)  {  name = n;  age = a;  gpa = g;  }    public String toString()  {  return name;  }    } |

|  |
| --- |
| **Java1619.java Output**  Java1619.java  Kathy Alexander  Peter VanVliet |

The neat thing about redefining **toString** is that it can completely decide the manner in which the object output will be formatted. The last two programs examples displayed the object's **name** field. Program **Java1620.java**, in figure 16.32, continues without trusty **Student** class. The **toString** method is now redefined to return the **name**, **age** and **gpa** fields separated by commas.

**Figure 16.32**

|  |
| --- |
| // Java1620.java  // This program demonstrates "redefining the <toString> method in the  // <Student> class so that it displays every field of a <Student> object.  // Note how this implementation resembles the <ArrayList> format.  import java.util.ArrayList;  public class Java1620  {  public static void main (String args[])  {  System.out.println("\nJava1620.java\n");  Student student1 = new Student("Kathy Alexander",21,3.475);  Student student2 = new Student("Peter VanVliet",18,2.265);  System.out.println(student1);  System.out.println(student2);  System.out.println();  }  }  class Student  {  private String name;  private int age;  private double gpa;    public Student(String n, int a, double g)  {  name = n;  age = a;  gpa = g;  }    **public String toString()**  **{**  **return "[" + name + "," + age + "," + gpa + "]";**  **}**    } |

|  |
| --- |
| **Java1620.java Output**  Java1620.java  [Kathy Alexander,21,3.475]  [Peter VanVliet,18,2.275] |

**16.10 Re-defining the equals Method**

In the same way that the **toString** method can be re-defined, we can also re-define the **equals** method. You have already seen this method in use when we were comparing strings earlier. You may remember that the equality operator == does not work for strings because it merely checks to see if the shallow memory addresses are equal, not the contents in the deeper memories. What we can now say is that the **==** operator merely checks to see if the *shallow**values* are equal, but does not compare the *deep**values*. This issue is not exclusive to **String** objects. It actually applies to all objects. When comparing objects how do you determine if they are equal? There are several different ways, as the next few programming examples will display. Program **Java1621.java**, in figure 16.33, tries to compare two objects with the **==** operator. This approach does not check correctly for equality.

**Figure 16.33**

|  |
| --- |
| // Java1621.java  // This program tries to compares 2 person objects with the == operator.  // This does not work because the == operator only checks the shallow value.  // It also does not give us a way to determine how we will check for equality.  import java.util.ArrayList;  public class Java1621  {  public static void main (String args[])  {  System.out.println("\nJava1621.java\n");  Person tom = new Person("Tom Jones",36,'M',40000);  Person sue = new Person("Sue Smith",29,'F',50000);  Person bob = new Person("Bob Brown",40,'M',50000);  System.out.println(tom);  System.out.println(sue);  System.out.println(bob);  System.out.println();    if (tom == sue)  System.out.println("Tom and Sue are equal.");  else  System.out.println("Tom and Sue are not equal.");  if (tom == bob)  System.out.println("Tom and Bob are equal.");  else  System.out.println("Tom and Bob are not equal.");    if (sue == bob)  System.out.println("Sue and Bob are equal.");  else  System.out.println("Sue and Bob are not equal.");  System.out.println();  }  }  class Person  {  private String name;  private int age;  private char gender;  private double salary;    public Person(String n, int a, char g, double s)  {  name = n;  age = a;  gender = g;  salary = s;  }    public String toString()  {  return "[" + name + ", " + age + ", " + gender + ", " + salary + "]";  }  } |

**Figure 16.33 continued**

|  |
| --- |
| **Java1621.java Output**  Java1621.java  [Tom Jones, 36, M, 40000.0]  [Sue Smith, 29, F, 50000.0]  [Bob Brown, 40, M, 50000.0]  Tom and Sue are not equal.  Tom and Bob are not equal.  Sue and Bob are not equal. |

Program **Java1622.java**, in figure 16.34, attempts to solve the problem of the previous program. This time the **==** is not used and replaced with the **equals** method. You saw that this approach worked correctly when **String** objects are compared for equality.

**Figure 16.34**

|  |
| --- |
| // Java1622.java  // This program tries to compare 2 <Person> objects with the <equals< method.  // This also does not work because we have not "redefined" the <equals> method  // for the <Person> class, and we are simply inheriting the <equals> method from  // the <Object> class, which only checks the shallow value.  import java.util.ArrayList;  public class Java1622  {  public static void main (String args[])  {  System.out.println("\nJava1622.java\n");  Person tom = new Person("Tom Jones",36,'M',40000);  Person sue = new Person("Sue Smith",29,'F',50000);  Person bob = new Person("Bob Brown",40,'M',50000);  System.out.println(tom);  System.out.println(sue);  System.out.println(bob);  System.out.println();    if (tom.equals(sue))  System.out.println("Tom and Sue are equal.");  else  System.out.println("Tom and Sue are not equal.");  if (tom.equals(bob))  System.out.println("Tom and Bob are equal.");  else  System.out.println("Tom and Bob are not equal.");    if (sue.equals(bob))  System.out.println("Sue and Bob are equal.");  else  System.out.println("Sue and Bob are not equal.");  System.out.println();  }  }  class Person  {  private String name;  private int age;  private char gender;  private double salary;    public Person(String n, int a, char g, double s)  {  name = n;  age = a;  gender = g;  salary = s;  }    public String toString()  {  return "[" + name + ", " + age + ", " + gender + ", " + salary + "]";  }  } |

**Figure 16.34 continued**

|  |
| --- |
| **Java1622.java Output**  Java1622.java  [Tom Jones, 36, M, 40000.0]  [Sue Smith, 29, F, 50000.0]  [Bob Brown, 40, M, 50000.0]  Tom and Sue are not equal.  Tom and Bob are not equal.  Sue and Bob are not equal. |

Program **Java2012.java** does what it is told to do and not what it is meant to do. How exactly should two **Person** objects be compared for equality. Should the name be the same? How about the ages, gender or salaries? The original **equals** method, from our **Object** class definition, is used and that definition only concerns itself with shallow values in the same manner as the original **toString** method. The result is not satisfactory.

Program **Java1623.java**, in figure 16.35, works correctly, provided the mission is to compare equality based on salary. The **equals** method is re-defined for the **Person** class to create the desired result.

**Figure 16.35**

|  |
| --- |
| // Java1623.java  // This program properly compares 2 person objects with the redefined equals method.  // It chooses to define "equality" solely based on a Person's salary, which may be  // overly capitalistic, but still makes a point.  import java.util.ArrayList;  public class Java1623  {  public static void main (String args[])  {  System.out.println("\nJava1623.java\n");  Person tom = new Person("Tom Jones",36,'M',40000);  Person sue = new Person("Sue Smith",29,'F',50000);  Person bob = new Person("Bob Brown",40,'M',50000);  System.out.println(tom);  System.out.println(sue);  System.out.println(bob);  System.out.println();    if (tom.equals(sue))  System.out.println("Tom and Sue are equal.");  else  System.out.println("Tom and Sue are not equal.");  if (tom.equals(bob))  System.out.println("Tom and Bob are equal.");  else  System.out.println("Tom and Bob are not equal.");    if (sue.equals(bob))  System.out.println("Sue and Bob are equal.");  else  System.out.println("Sue and Bob are not equal.");  System.out.println();  }  }  class Person  {  private String name;  private int age;  private char gender;  private double salary;    public Person(String n, int a, char g, double s)  {  name = n;  age = a;  gender = g;  salary = s;  }    public String toString()  {  return "[" + name + ", " + age + ", " + gender + ", " + salary + "]";  }    **public boolean equals(Person temp)**  **{**  **return salary == temp.salary;**  **}**  } |

**Figure 16.35 continued**

|  |
| --- |
| **Java1623.java Output**  Java1623.java  [Tom Jones, 36, M, 40000.0]  [Sue Smith, 29, F, 50000.0]  [Bob Brown, 40, M, 50000.0]  Tom and Sue are not equal.  Tom and Bob are not equal.  Sue and Bob are equal. |

Program **Java1624**, in figure 16.36, shows that the **equals** method can be easily altered for a different comparison. This equality is based on total equality in each of the **Person**'s object fields. You will notice a special keyword, **this**, is included. This keyword (pun intended) is optional. It exists to help distinguish between different objects.

**Figure 16.36**

|  |
| --- |
| // Java1624.java  // This program defines equality differently from the previous programs.  // Now all data fields must match for 2 Person objects to be considered equal.  // A special <this> reference in the equals method helps to distinguish the 2 objects.  // NOTE: It is not uncommon for the equals method from one class to call the  // equals method from another class.  import java.util.ArrayList;  public class Java1624  {  public static void main (String args[])  {  System.out.println("\nJava1624.java\n");  Person tom = new Person("Tom Jones",36,'M',40000);  Person sue = new Person("Sue Smith",29,'F',50000);  Person bob = new Person("Bob Brown",40,'M',50000);  System.out.println(tom);  System.out.println(sue);  System.out.println(bob);  System.out.println();    if (tom.equals(sue))  System.out.println("Tom and Sue are equal.");  else  System.out.println("Tom and Sue are not equal.");  if (tom.equals(bob))  System.out.println("Tom and Bob are equal.");  else  System.out.println("Tom and Bob are not equal.");    if (sue.equals(bob))  System.out.println("Sue and Bob are equal.");  else  System.out.println("Sue and Bob are not equal.");  System.out.println();  }  }  class Person  {  private String name;  private int age;  private char gender;  private double salary;    public Person(String n, int a, char g, double s)  {  name = n; age = a; gender = g; salary = s;  }    public String toString()  {  return "[" + name + ", " + age + ", " + gender + ", " + salary + "]";  }    **public boolean equals(Person that)**  **{**  **return this.name.equals(that.name) &&**  **this.age == that.age &&**  **this.gender == that.gender &&**  **this.salary == that.salary;**  **}**  } |

**Figure 16.36 continued**

|  |
| --- |
| **Java1624.java Output**  Java1624.java  [Tom Jones, 36, M, 40000.0]  [Sue Smith, 29, F, 50000.0]  [Bob Brown, 40, M, 50000.0]  Tom and Sue are not equal.  Tom and Bob are not equal.  Sue and Bob are not equal. |

**16.11 Summary**

A **string** is a set of characters, which behaves as a single unit. The characters in a string include upper-case and lower-case letters, numerical characters and a large set of characters for a variety of purposes like:

! @ # $ % ^ & \* ( ) \_ +

A **string literal** is a set of characters delimited with double quotations like:

"Seymour Snodgrass" and "SSN: 123-45-6789"

**String** method **substring** returns a set of consecutive characters from string **s1**, starting at index **0**, and ending at index **k-1**.

**s2 = s1.substring(0,k);**

If **s1 == "Aardvark"** **and k == 4** then **s2** becomes **"Aard"**

Overloaded **String** method **substring** returns a set of consecutive characters from string **s1**, starting at index **k**, and ending at index **length**.

**s2 = s1.substring(k);**

If **s1 == "Aardvark"** **and k == 4** then **s2** becomes **"vark"**

**String** method **valueOf** converts the provided parameter and returns a string. Four overloaded **valueOf** methods are displayed. Note that the **valueOf** method is a **static** method, which is called with the **String** class identifier.

**String s1 = String.valueOf(1000);**

**String s2 = String.valueOf(123.321);**

**String s3 = String.valueOf(true);**

**String s4 = String.valueOf('A');**

**String** method **parseInt** converts a string into an integer.

Method **parseDouble** converts a string into a double.

**int n1 = Integer.parseInt(s1);**

**double n2 = Double.parseDouble(s2);**

**String** Method **equals** returns **true** if **s1 == s2**, and **false** otherwise.

**if (s1.equals(s2))**

Method **compareTo** returns **0** if **s1 == s2**, otherwise an integer is returned based on the difference between **s1** and **s2**. If **s3 < s4**, the returned value is negative.

**int distance = s3.compareTo(s4);**

It is not possible to simply use a **print** statement with a static array object to display the individual array elements. A loop structure must be used to access each array element.

**ArrayList** object values can be displayed with a **print** statement without using any loop control structure. This is possible, because the **toString** method is re-defined in the **ArrayList** class.

It is also possible to re-define the **equals** method. Both the **toString** method and the **equals** method are originally defined in the **Object** class with minimal capabilities using shallow memory values. Re-definition allows specific processing that is appropriate for the re-defining class.