OOPJAV – Object-Oriented Programming in Java

Seminar 2 - Lecture

Congratulations on surviving week 1! There was a lot of terminology introduced last week, so make sure you are familiar with terms such as: *class*, *instance*, *attributes*, and *method,* as well as the distinction between *class attributes* and *instance attributes*, and *class methods* and *instance methods*. If we envisage this module as a climb to the top of a mountain with various gradients of ascent, we unfortunately still have a steep slope to climb this week before we reach a plateau where we can grab a short break to collect ourselves.

This week we are going to look at the nature of *data* in java programs, *data types* and (most importantly) *data input* (we did output last week). We will be using the JOptionPane as our input and output medium during the week.

I. Data

All computer software works with data of some form. Literally, the word data means "that which is given." In computer terms, *data* is the name given to the items operated upon by a computer program. Any data item used in a program has a number of attributes as listed below. The nature of these attributes is defined by the *type* of the data item.

1. **Address.** The *address* or *reference* of a data item is the physical location in *memory* (computer store) of that data item. The amount of memory required by a particular data item is dictated by its *type*.

2. **Value.** The binary number stored at an address associated with a data item is its *value*. This is the **internal representation** of the value which has an **interpretation** associated with it. Consider the binary number 01011010. This can be interpreted as:
    a. The decimal integer 90
    b. The hexadecimal integer 5A
    c. The octal integer 132
    d. The ASCII character (capital) Z
    e. Etc.

3. The required **interpretation** is again dictated by the type of the data item. The possible values that are associated with a data type (such as the values a, b, c, etc. associated with the char type) are called *literals*.

4. **Operations**. The type of a data item also dictates the operations that can be performed on it. For example, it makes sense to allow the standard mathematical operations to be applied to *numeric data types,* while the same cannot be said for *string data types*.

5. **Identifiers**. To do anything useful with a data item, all the above must be tied together by giving the data item an identifying name (sometimes referred to as a label or symbol). In Java (like many other programming languages):
    a. Names are made up of alpha-numeric characters and the underscore (_) character
    b. White space is not allowed

c. Case is significant – Java is a case-sensitve language
d. Cannot use key/reserved words as identifiers (e.g. class, new, import, etc.)

Data items can be:

1. Instance or class fields (included in a class declaration)
2. Arguments (included in a method's *signature*)
3. Local (defined within the body of a method)

A data item is introduced using a data declaration. Java, like most modern programming languages, is what is known as a *strictly typed* language (and *strictly classed*). This means Java insists that programmers be precise when defining (declaring) and using data items. This, in turn, ensures that errors resulting from incorrectly defining or using data items are avoided.

II. Data Declarations, Assignment and Initialization

A data *declaration* introduces a data item. In Java, this is achieved by associating the item's identifier with a data type:

```
int integerItem;
```

Here we have declared a data item integerItem, which is of type **int**. A declaration statement can occur in either: (1) a class definition, (2) an argument (the signature of a method) or (3) within a method declaration (the body of a method).

Assignment is the process of associating a value with a data item. This is achieved using an *assignment operator* (=). The Java example:

```
integerItem = 2;
```

Here we have assigned the *literal* value 2 to the data item integerItem. Alternatively, on the right hand side of the assignment operator, we could have used an identifier, an expression or the result of a method call. Examples:

```
integerItem = myDataItem;
integerItem = (myDataItem+2) * someIncrement;
integerItem = myInstance.myMethod(argument1,argument2);
```

Note that the last example assumes we are calling a public instance method from outside the class where it is defined. When calling a method from inside the class where it is defined we omit the instance identifier, e.g.:

```
integerItem = myMethod(argument1,argument2);
```

Initialisation is the process of assigning an initial value to a (field or local) data item on declaration. For example:

```
int integerItem = 2;
```

Here we have declared a data item, integerItem, which is of type int and takes an initial value of 2. In Java this cannot be done where data items are used as arguments to methods.

A. Example Class - Data Initialization

In **Figure 1**, a short Java class definition is presented to illustrate the different forms of data initialisation available in Java.

```
1  //DATA INITIALISATION CLASS
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class DataInitialisation{
9
10     //-------ATTRIBUTES-------
11     private int dataItem1 = 0;
12     private int dataItem2 = 1, dataItem3 = 2;
13     private int dataItem4 = dataItem3 + 1;
14     private int dataItem5 = dataItem2 + dataItem4;
15
16     //-------METHODS-------
17
18     /*-------OUTPUT DATA-------*/
19     /*Method to output values associated with instance fields.*/
20
21     public void outputData(){
22         JOptionPane.showMessageDialog(null, "Item 1 = " + dataItem1);
23                 JOptionPane.showMessageDialog(null, "Item 2 = " + dataItem2);
24                 JOptionPane.showMessageDialog(null, "Item 3 = " + dataItem3);
25                 JOptionPane.showMessageDialog(null, "Item 4 = " + dataItem4);
26                 JOptionPane.showMessageDialog(null, "Item 5 = " + dataItem5);
27             }
28
29             /*-------NEW VALUES-------*/
30             /*Method to assign new values to the instance fields*/
31
32             public void newValues(int value1, int value2, int value3, int value4, int value5){
33                 dataItem1 = value1;
34                 dataItem2 = value2;
35                 dataItem3 = value3;
36                 dataItem4 = value4;
37                 dataItem5 = value5;
38             }
39 }
```

Figure 1: *Data initialization program*

Here we have initialised (declared and assigned values to) a sequence of data fields using a variety of different manners supported by Java. Note that in keeping with the spirit of OOP, all the data fields are declared to be private, i.e. they can only be accessed from within the class where they are declared. This example contains two methods, one to output the instance fields and one to assign new values to the fields. Note how the formal parameters to the second method are declared.

B. Example Application Class - Data Initialization Class

The class presented in **Figure 1** does not include a main() method so we cannot use it directly. We could have included a main() method but in the spirit of object-oriented programming, we should keep application classes and class definitions of the above form apart, as this will encourage desirable OOP features such as reusability, flexibility, adaptability and extension.

Therefore we will write a second application class that will use the above class (note that they must both be in the same directory) as shown in **Figure 2**.

```
1  //DATA INITIALISATION APPLICATION
2  //Y. ing
3  //March 2007
4  //The University of Liverpool, UK
5
6  public class DataInitialisationApp{
7
8      /*-------Main Method-------*/
9
10     public static void main(String args[]){
11         int localData1 = 1;
12         int localData2 = 2, localData3 = 3;
13             int localData4 = localData3 + 1;
14             int localData5 = localData2 * 2 + 1;
15
16             //Create a DataInitialisation Instance
17             DataInitialisation newDataInit = new DataInitialisation();
18
19             //Output the data values assigned on intiailisation
20             newDataInit.outputData();
21
22             /*Change the data values by assigning values of local data items and output*/
23             newDataInit.newValues(localData1, localData2, localData3, localData4, localData5);
24             newDataInit.outputData();
25     }
26 }
```

Figure 2: *Data initialisation class*

Here we have created an application class with a single method, main(), which has a number of local data items associated with it (notice how they are declared). We also created an instance, newDataInit, which is an instance of the class DataInitialisation using the constructor DataInitialisation(). Note that the constructor has not been explicitly defined anywhere. This is because Java automatically creates a *default constructor* for each class if no constructor of this format has been defined. In effect, an instance created by the constructor can be thought of as a compound data item (a data item which has more than one value associated with it) which is of type DataInitialisation. Also be aware of how we have linked the instance, using the dot operator, to the various instance methods in the DataInitialisation class.

C. Execution

To compile the above java source code files we type:

javac DataInitialisation.java
javac DataInitialisationApp.java

This will produce two JBC (Java Byte Code) files: DataInitialisation.class and DataInitialisationApp.class. To interpret the resulting byte code we invoke the JVM (Java Virtual Machine) as follows:

java DataInitialisationApp

The result will be:

```
0 1 2 3 4
1 2 3 4 5
```

Notes:

1. For the above files to work, both the source code files must be in the same directory. It is therefore a good idea to create a dedicated directory for a particular problem solution in which all the source code files pertaining to that solution can be stored.

2. For the time being, you should create a source code file for each class that you wish to create.

3. All Java source code files must have the extension .java.

4. We can not interpret the DataInitialisation class because it does not contain a main() method, which is why we have created the class DataInitialisationApp that does contain a main() method. Any one application can have only one main() method.

III. Variables and Constants

Given that we can always replace a particular bit pattern with another, the value of a data item can always be changed. Data items that are intended to be used in this way are referred to as *variables*. We sometimes refer to instance or class variables.

The value associated with a variable can be changed using what is called an *assignment operation*. In the case of Java, this is indicated by the symbol (=) as illustrated above. In Java, by default (as is the case with many programming languages), data items are assumed to be variables. All the integer data items in the above example are interpreted in this way. The DataInitialisation instance, although also a data item, is interpreted in a different way because it is a *reference* to a *compound data item*, but more on this in section 4.

It is sometimes desirable to define a data item whose value cannot be changed. Such data items are referred to as *constants*. In Java, they are indicated by incorporating the key word **final** into the declaration:

```
final int constantDataItem = 2;
```

Here we have declared a constant, constantDataItem, which has a value of 2.

However it should be appreciated that what we are doing here is telling the compiler to flag the data item as a constant (i.e. we are instigating software protection). Theoretically, we can still change the bit pattern representing the value if we can ascertain its address, i.e. find where it is stored.

The use of *named* or *symbolic* constants offers three advantages:

1. By defining a symbolic constant in a single place. If that constant needs to be changed or corrected, this need only be applied to the definition and not through out the program.

2. The significance of a symbolic constant is easily understood, where as the use of a 'magic number' is not so obvious.

3. The risk of incorrectly typing values is reduced. For example, defining a constant

```
final double PI = 3.14159;
```

reduces the risk of incorrectly typing 3.14159 every time it is used.

Named constants therefore enhance the maintainability of a program. Note also that attributes defined within a class may be variables or constants. We sometimes refer to instance variables/constants or class variables/constants.

IV. Object References

An instance or class data item is a data item the same as any other (albeit an extremely complex one). When we create a data item, a section of memory is set aside on the *heap* for this item. When we create an instance of a class, this is the same as creating any other data item except that the immediate value associated with the data item name is an address rather than a value. This address indicates the start of the section of memory reserved in the heap for the object. The reasoning behind this is because an object is what we call a *compound data item,* in that it can have more than one value associated with it, and it therefore needs a sequence of memory slots (one for each value). The immediate value associated with an instance name is thus the address of the start of this block of memory. We refer to such address values as *object references* (because they reference the start address in the heap for the values associated with the indicated compound data item). In some languages (e.g. C and C++) such data items are called *pointers* in that they point to the start of a section of memory in the heap.
For example if we create an instance, newInst1, of the class ExampleClass:

ExampleClass newInst1 = new ExampleClass();

This will create a data item which, like any other data item, has an address and a value, which is a reference value that points to the section of the heap where the instance fields for newInst1 are stored.

V. Uninitialized Objects

Last week we used a constructor to create an instance of the class GrandParent as follows:

GrandParent Louise = new GrandParent();

Here we have created the object Louise, which is an instance of the class GrandParent, and which has been initialised with a reference value indicating the storage location of all the associated data items. It is possible to create an object such as Louise and leave the initialisation till later. We do this using a data declaration of the form:

Grandparent Louise = null;

The null indicates a special value of nothing (*not zero but nothing!*), which means the reference value does not point to anything.

VI. Renaming Objects

We can have two or more names which identify the same object. For example, given the object newInst1 created above, we can assign its immediate value (which, remember, is an object reference, i.e. an address) to another label as follows:

newInst2 = newInst1;

This is called *aliasing* and should be avoided as it causes confusion since we have not made a copy of the object but created a second means of accessing it!

VII. Data Types

We have seen that the type of a data item defines:

1. The nature of its internal representation (number of bits)
2. The interpretation placed on this internal representation
3. As a result of (2) the allowed set of values for the item
4. Also as a result of (2) the operations that can be performed on it

We have also seen that the process of associating a type with a data item is referred to as a *data declaration*. This binds a data item name to a type definition. The process of assigning a value to a data item on declaration of that item is referred to as *initialisation*. This concept was illustrated in the data initialisation example program presented earlier.

Java supports eight *primitive* (basic) data types details of which are presented in **Figure 3** below. These all have specific storage and interpretation considerations associated with them and both have particular operations which may be applied to them. For example the numeric types have the + and - arithmetic operations associated with them. Note that these operations are *overloaded*, which means there is one plus operation for integer addition and one for real number addition. To mix, for example, integer and real number addition - so called *mixed mode arithmetic* - requires special consideration (more of this later).

Type	Identifier	Storage (bits)	Values
Character	char	16	Unicode 2.0
8-bit signed integer	byte	8	-128 to 127
Short signed integer	short	16 (2 bytes)	-32768 to 32767
Signed integer	int	32 (4 bytes)	-2,147,483,648 to +2,147,483,647
Signed long integer	long	64 (8 bytes)	Maximum of over 10^18
Real number (single precision)	float	32 (4 bytes)	Maximum of over 10^38
Real number (double precision)	double	64 (8 bytes)	Maximum of over 10^308
Boolean	boolean		True or false

Figure 3: *Primitive data types*

VIII. Categorization of Types

1. **Pre-defined and programmer-defined types**. Pre-defined types are types immediately available to the user (they are integral to the language). Programmer-defined types are types derived by the programmer using existing types (pre-defined or otherwise).

2. **Scalar and compound types**. Scalar types define data items that can be expressed as single values (e.g. numbers and characters). Compound types (also referred to as composite or complex types, or object types in some OOP languages) define data items

that comprise several individual values, e.g. the Java type String introduced earlier. Scalar types are generally pre-defined, while compound types are generally programmer-defined (note that the Java compound type String is predefined).

3. **Discrete and non-discrete types**. Discrete (also known as ordinal or linear) types are types for which each value (except its minimum and maximum) has a fixed predecessor and a successor value. The opposite are referred to as non-discrete types.

4. **Primitive and higher level types**. Primitive types (also referred to as basic or simple types) are the standard scalar predefined types that one would expect to find ready for immediate use in any imperative programming language. Higher level types are then made up from such primitive types or other existing higher level types. Higher level types are not necessarily programmer defined - for example many languages (including Java) have a string pre-defined high-level type.

IX. Wrapper Classes

We have already used JOptionPanes to accept input in week 1, but now we need to look at how that process works and how we can manipulate data input so that we may perform calculations.

As we saw in week 1, JOptionPanes use a method called showInputDialog to accept data from the user. The data comes in as a string, so if we want to use the data in calculations, we have to convert the string to the appropriate format. Because this is a very common requirement, Java provides a mechanism whereby this may be achieved. We use something called a *wrapper* class. Each of the primitive types, char, int, long, float, double, etc., has a corresponding wrapper class associated with it: Character, Integer, Long, Float, Double etc. (note the upper case letter for the first character of each identifier in keeping with the Java class naming convention). These classes are designed to wrap the primitive types into a class.

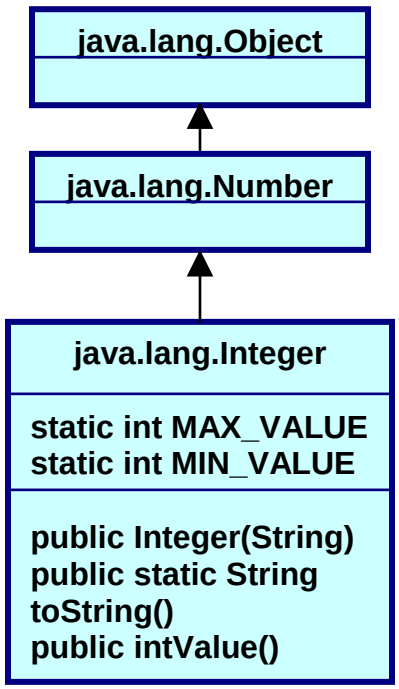


Figure 4: *Class diagram showing API classes associated with the Integer wrapper class*

If we examine the Integer class (**Figure 4**) we will notice that (amongst other things) it contains a constructor that creates an instance of the class Integer that represents the value given by a string argument, i.e. it 'wraps-up' a string data item into an instance of the class Integer:

public Integer(String s) throws NumberFormatException;

(note that this throws an exception in the case of an error – we will discuss error handling in a later seminar). The Integer wrapper class also defines a method to assign the value 'wrapped-up' in an instance of the class Integer to a data item of type int:

public int intValue();

Therefore to input an integer from a JOptionPane requires the following steps:

1. Read in the integer in the form of a string and store it in a string variable.

2. Create an instance of the class Integer and assign a value to it using the Integer(String s) constructor.

3. Assign the instance value to an appropriately defined integer variable using the intValue() method.

The application program presented in **Figure 5** implements this. If, after compilation, we run this program, we will be asked for some input, which will then be echoed to the screen:

java IntegerInput
Please input an integer: 23
Your int value is 23

Try inputting a very large number such as 1000000000000. This will generate an exception because this number is larger than the maximum defined for the integer type:

java.lang.NumberFormatException: 1000000000000
at java.lang.Integer.parseInt(Integer.java)
at java.lang.Integer.(Integer.java)
at IntegerInputApp.main(IntegerInputApp.java:35)

The meaning of the above will be explained later.

```
1  //INTEGER INPUT CLASS
2  //Y. Jing
3  //March 2007
4  //University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class IntegerInput{
9
10     //-------Main Method-------
11
12     public static void main(String[] args){
13         int convertedStr;
14
15         //Accept a string value from the user
16         String intInputStr = JOptionPane.showInputDialog("Please input an integer value:");
17
18         //Create a new instance of Integer with a string argument
19         Integer convertToInt = new Integer(intInputStr);
20
21         //Call the intValue method from the Integer class and assign the result to an int
22         convertedStr = convertToInt.intValue();
23         JOptionPane.showMessageDialog(null, "Your int value is " + convertedStr);
24     }
25 }
```

Figure 5: *Integer input example application program*

The Integer wrapper class also includes the method toString which converts its value to a string. This is most commonly used automatically by the Java interpreter when ever it discovers a concatenation operator (+) in an output statement (e.g. print or println). In Figure 1, the line:

JOptionPane.showMessageDialog(null, "Item 1 = " + dataItem1);

is automatically invoking the toString method on dataItem1 which is an int.

Remember that the concatenation operator joins strings, however the *operands* for the operator do not necessarily have to be strings. Where this occurs the Java interpreter will automatically invoke the appropriate toString method - there is one in each wrapper class.

Finally, Integer wrapper class also includes two class constants called MAX_VALUE and MIN_VALUE. These contain the maximum and minimum value that an integer type can take and are useful when checking inputs to ensure that they do not go out of range.

X. Refinement of the Input Process

The example in **Figure 5** includes unnecessary data items, namely the Integer instance convertToInt. We could rewrite the above as shown in **Figure 6** and save space and verbiage. This provides more succinct encoding and is the approach we will continue to use in our coding.

```
1  //INTEGER INPUT VERSION 2
2  //Y. Jing
3  //March 2007
4  //University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class IntegerInput2{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int convertedStr;
14
15         //Accept a string value from the user
16         String intInputStr = JOptionPane.showInputDialog("Please input an integer value:");
17
18         /*Create a new instance of Integer with a string argument, call the intValue
19         convertedStr = (new Integer(intInputStr)).intValue();
20         JOptionPane.showMessageDialog(null, "Your int value is " + convertedStr);
21     }
22 }
```

Figure 6: *Refined integer input example program*

XI. Type Conversion

It is sometimes necessary to convert a data item of one type to another type; for example, when it is necessary to perform some arithmetic using data items of different types (*mixed mode arithmetic*). Under certain circumstances, type conversion can be carried out automatically, but in other cases it must be forced manually.

A. Automatic Conversion

In Java, type conversions are performed automatically when the type of the expression on the right hand side of an assignment operation can be *safely* promoted to the type of the variable on the left hand side of the assignment without losing accuracy. Therefore we can safely assign:

Byte -> short -> int -> long -> float -> double

The -> symbol used here should be interpreted as "to a/an." For example:

```
long myLongInteger;   // 64 bit long integer
int  myInteger;       // 32 bit standard integer

myLongInteger = myInteger;
```

The extra storage associated with the long integer in the above example will simply be padded with extra zeros.

B. Explicit Conversion (Casting)

The above will not work the other way around. For example, we cannot automatically convert a long to an int because the first number requires more storage than the second number and consequently, important information may be lost. To force such a conversion, we must carry out an explicit conversion (assuming, of course, that the long integer will fit into a standard integer). This is done using a process known as a *type cast*:

myInteger = (int) myLongInteger

This tells the compiler that the type of myLongInteger must be temporarily changed to an int when the given assignment statement is processed. Therefore, the cast only lasts for the duration of the assignment.

Java type casts have the following form:

(T) N

where T is the name of a numeric type and N is a data item of another numeric type. The result is of type T.

XII. Distinction Between Casting and Rounding

Given a data item of type double or float, we can change the type of this item so that it becomes a data item of type long or int using a cast operation as described above. What happens in this case is that the exponent part of the real number is simply omitted. For example, 99.9999 will become 99. In cases such as this, it might be more desirable to say that the integer equivalent of 99.9999 is 100 (i.e. round up the number). We cannot achieve this using a cast; however the Java API Math class contains methods to achieve this; namely the methods rint and round.

The first rounds a double up or down to its nearest integer equivalent (without converting the type). The second has two versions, one to round and convert a float to an int, and one to round and convert a double to a long. Consider the Java code given in **Figure 7**. The code allows a user to input a double which is then output; first as an integer using a cast, then as an integer using the round method, and finally as a double rounded to the nearest integer.

```
1  //ROUNDING EXAMPLE APPLICATION CLASS
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class RoundingEx{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         double inputDouble;
14
15         //Input
16         String intInputStr = JOptionPane.showInputDialog("Please input an double value:");
17
18         /*Create a new instance of Double with a string argument,
19         call the doubleValue method from the Double class and assign the result to a double*/
20         inputDouble = new Double(intInputStr).doubleValue();
21
22         //Output
23         JOptionPane.showMessageDialog(null, "Your cast int value is " + (int)inputDouble);
24         JOptionPane.showMessageDialog(null, "Your rounded int value is " + Math.round(inputDouble));
25         JOptionPane.showMessageDialog(null, "Your 'rint' int value is " + Math.rint(inputDouble));
26     }
27 }
```

Figure 7: *Rounding example program*

Some example output is given below. Note that a value of 1.5 is rounded upwards.

```
$ java RoundingEx
Input a double precision number = 1.2
Input converted to an int using a cast = 1
Input converted to an int using 'round' method = 1
Input converted to an int using 'rint' method = 1.0

$ java RoundingEx
Input a double precision number = 1.8
Input converted to an int using a cast = 1
Input converted to an int using 'round' method = 2
Input converted to an int using 'rint' method = 2.0

$ java RoundingEx
Input a double precision number = 1.5
Input converted to an int using a cast  = 1
Input converted to an int using 'round' method = 2
Input converted to an int using 'rint' method = 2.0
```

XIII. Strings

We have seen that a string consists of a sequence of characters, so we say that a string is a *character arr*ay (more on the concept of arrays later). As such, the string is also said to have a *length*, i.e. the number of characters in the sequence. Further, each character is said to have an *index number* (or simply an index) associated with it, commencing with the first character which has the index 0, followed by the second character which has the index 1, and so on.

The need for string data items in computer programming is so frequent that many programming languages provided a predefined string type. Java, while remaining with the concept of a string being an array of characters, does not do this. Instead it provides a predefined class called String. We use this in the same manner that we might use any other predefined class. For example, we might declare an instance of the String class as follows:

```
String name = new String();
```

However, in Java, instances of the class String are always constants, so their value is fixed upon construction. In order to assign a value to the instance name, we must do this when it is created. Because instances of the class String are so common, Java provides short-hand to achieve the value assignment as shown below:

```
String name = "Kimberly";
System.out.println(name);
```

XIV. Operations on Strings

The String class provides many methods that may be used to perform useful operations on strings. A selection is presented in **Figure 8**.

Method prototype	Description
char charAt(int index)	Returns the character at the specified index
int compareTo(String anotherString)	Compares two strings lexicographically

String concat(String str)	Concatenates the specified string to the end of a string
str1 + str2	Concatenation operator – achieves the same as concat method
boolean equals(String str)	Compares the string to the specified object
int indexOf(char ch)	Returns the index within the string of the first occurrence of the specified character
int lastIndexOf(char ch)	Returns the index within the string of the last occurrence of the specified character
int length()	Returns the length of the string
String replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in the string with newChar
String substring(int beginIndex, int endIndex)	Returns a new string that is a substring of the original string
String toLowerCase()	Converts all of the characters in the String to lower case
String toUpperCase()	Converts all of the characters in the String to upper case

Figure 8: *String class methods*

There are many more methods as well as variations on the above methods, but in the context of the list presented here, it is worth noting the following:

1. All of the above methods are instance methods and therefore must be called with an instance of the class String.

2. We use the charAt method to return the character value at a particular position. This is analogous to indexing into a true array (more on this later).

3. When using the compareTo method, the method returns an integer indicating the lexicographic (alphabetic) location of the first letter of the first string with the first letter of the second string. If the first letters are both different, an integer is returned indicating the relative alphabetic location of the second start letter compared with the first, negative if the second is lexicographically after the first, and positive otherwise. If both strings are identical, then a 0 value is returned. If both have the same start letter but are not the same, either a 32 or -32 is returned according to whether the second is lexicographically before or after the first. Alternatively, if all we are interested in is direct comparison, we can use the equals method which returns a boolean value.

4. The concatenation method appends its argument (which must be a string) to the indicated string and returns a new instance of the class String.

5. The indexOf and lastIndexOf methods find occurrences of a particular character starting from the start or end of the string respectively, and return the index of that character. If the occurrence cannot be found, the methods return -1. A variation on the indexOf method includes a second argument to indicate a start index for the search.

6. The subString method is used to create new instances of the class String from existing instances. The new string is specified by giving the required index range within the existing string (inclusive of the start index and exclusive of the end index). If no upper

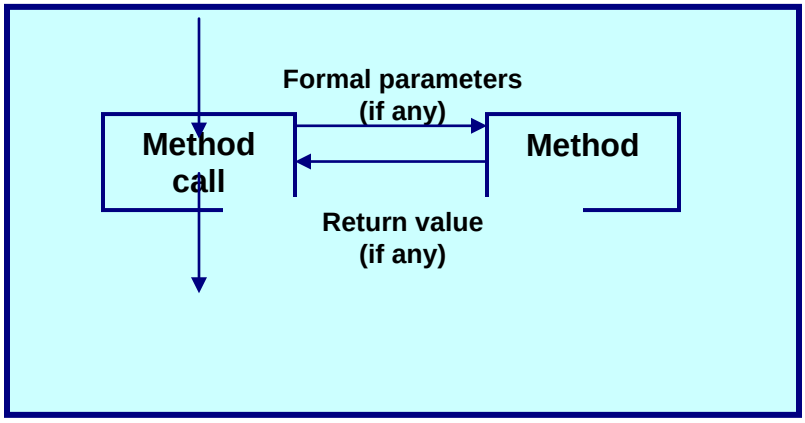
bound is specified for the substring, Java assumes that this is the upper bound of the given string. The method can also be used to create a copy of a string.

7. The replace method is used to replace each occurrence of the first argument in a string with the second argument.

8. The toUpperCase and toLowerCase methods convert a given string to lower case or uppercase characters respectively.

XV. Method Calls

The process of calling one method from within another method is known as *routine invocation*, where we say that we invoke a method. Routine invocation is activated by a *method call,* which names the method and supplies the *actual parameters* which are assigned to the *formal parameters* associated with the method. On completion of the invocation, control is returned to the point immediately after the invocation, as illustrated in **Figure 9**.



If the method we are calling is defined in some other class then we must call it by either:

1. Linking it to the class name if it is a class method

2. Linking it to an instance of the class if it is an instance method

This is illustrated in the code fragment given in **Figure 10**. Here, we define a class with a private instance variable and a public instance method which returns the instance variable multiplied by 3. The code given in **Figure 11** presents an application class that makes use of the class given in **Figure 10**. Two instances are created of the class ExampleClass; instance1 and instance2, each of which calls the treble method in turn. The resulting output will be as follows:

Instance 1 = 6
Instance 2 = 9

```
1  //EXAMPLE CLASS
2  //Y Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  public class ExampleClass{
7
8      //-------FIELDS-------
9      private int value;
10
11     //-------CONSTRUCTORS-------
12     public ExampleClass(int num){
13         value = num;
14     }
15
16     //-------METHODS-------
17     public int treble(){
18         return(value * 3);
19     }
20 }
```

Figure 10: *Example class*

```
1  //EXAMPLE APPLICATION CLASS
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class ExampleApplicationClass{
9
10     //-------FIELDS-------
11     private int value;
12
13     //-------Main Method-------
14     public static void main(String [ ] args){
15         ExampleClass instance1 = new ExampleClass(2);
16         ExampleClass instance2 = new ExampleClass(3);
17
18         //Output
19         JOptionPane.showMessageDialog(null, "Instance 1 = " + instance1.treble());
20         JOptionPane.showMessageDialog(null, "Instance 2 = " + instance2.treble());
21     }
22 }
```

Figure 11: *ExampleApplication class with method calls*

Reading List:

In **Chapter 2** of the text, read the following sections:

2.5-2.8.

In **Chapter 3** of the text, read the following sections:

3.3-3.5, 3.6-3.9.

In **Chapter 14** of the text, read the following sections:

14.2.

In **Chapter 29** of the text, read the following sections:

29.1-29.3.

You will notice in each chapter various sections entitled Common Programming Errors, Testing and Debugging, Performance Tips and Software Engineering Observation, etc. These sections are meant to be skimmed over or even ignored on a first reading, but bear them in mind as you may wish to return to them later.

Helpful Websites

Some sites you may wish to visit this week for clarification of the ideas presented in the lecture and in the book are:

For data types and mathematic operators:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/variables.html

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/arithmetic.html

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/relational.html

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/assignment.html

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html

For type conversion:

http://java.sun.com/docs/books/tutorial/java/data/numbertostring.html

http://www.arps.org/users/hs/kochn/java/Wrapper.htm

http://www.ibiblio.org/obp/thinkCSjav/app01.htm

For help with methods:

http://java.sun.com/docs/books/tutorial/java/concepts/message.html

http://java.sun.com/docs/books/tutorial/java/data/objectcreation.html

You may notice that many of the above websites come directly from the Sun Microsystems Java web. I encourage you to use this site frequently as there are many tutorials, short courses, white papers, discussion groups, etc., that you may use to your advantage. You may wish to start getting used to the Java API files as soon as possible - many of your questions about syntax can be answered here.