



Chapter 5: Using Objects

Objectives

Students should

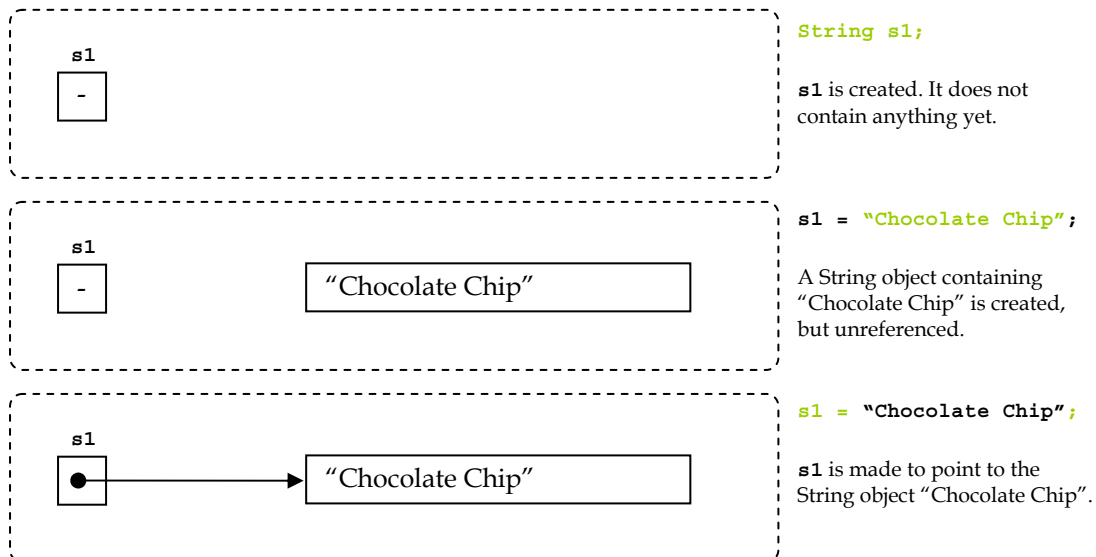
- Understand classes and objects.
- Be able to use class methods and data from existing classes.
- Be familiar with the *String* class and be able to use its methods.
- Be able to use the *BufferedReader* class to get users' input from keyboards.
- Be able to process keyboard input as *String* and numeric values.

Classes and Objects

Classes are non-primitive data types in Java. New classes can be made while there are no such things as new primitive data types. An *object* is an instance of a class. Consider the following Java statements. Note that, as we have mentioned, *String* is a class in Java.

```
String s1;  
s1 = "Chocolate Chip";
```

In the first statement, a variable named `s1` is declared as a variable that is used for storing an object of the class *String*. In the second statement, an object of class `String` is created with the content "Chocolate Chip" and assigned to the variable `s1`. In other words, `s1` is made to point to the *String* object "Chocolate Chip". What have occurred can be depicted as the following figure.

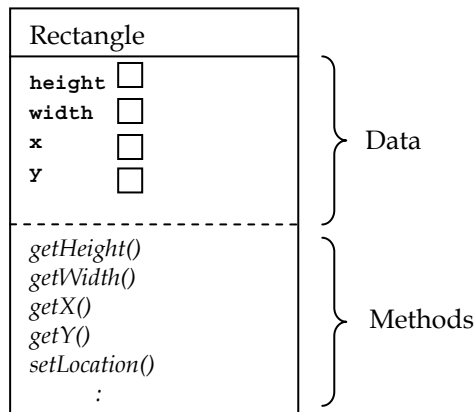


Using Data and Methods provided in Classes

An object of a class contains data and methods. For example, there is a Java class names *Rectangle*, which is a data type used for representing a rectangle. The data contained in each object of this *Rectangle* class are `height`, `width`, `x`, and `y`, which stores necessary attributes that define a rectangle. Apart from data, the class also provides several methods related to using



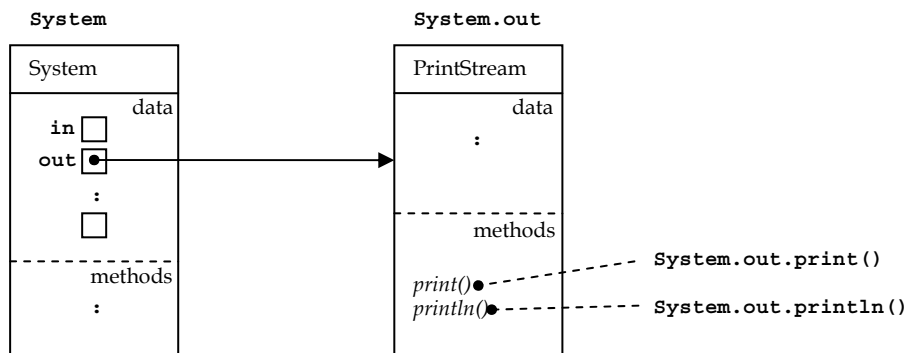
the rectangle, such as `getHeight()`, `getWidth()`, `getX()`, `getY()`, and `setLocation()`. This class might be illustrated as the following figure.



Generally, when we write computer programs in Java, we make use of existing methods and data that have already been defined or made in some existing classes. The dot operator (.) is used for accessing data or methods from a class or an object of a class. We have already seen (and used) two methods that print message onto the screen since earlier chapters. Here, we discuss the meaning of them. Consider the two methods below.

```
System.out.print("Strawberry Sundae");
System.out.println("Banana Split");
```

The four periods seen in both statements above are the dot operator. *System* is a class in a standard Java package. This class contains an object called *out*, whose class is a class called *PrintStream*. Thus, using the dot operator, we refer to this *out* object in the class *System* by using *System.out*. Consequently, the *PrintStream* class contains *print()* and *println()*, and we can access the two methods using *System.out.print()* and *System.out.println()*.



Some data and methods can be accessed by using the dot operator with the name of the class while some can be accessed by using the name of the variable storing the object of that class. Data and methods that are accessed via the class name are called *class* (or *static*) data and *class* (or *static*) methods. Data and methods that are accessed via the object name are called *instance* (or *non-static*) data and *instance* (or *non-static*) methods. At this point, you are not expected to know that whether the data and methods that you have never come across before are associated with classes or instances (objects). Just make sure you understand what you are doing when accessing ones.

Let's look at an example.



```
public class AreaOfCircle
{
    public static void main(String[] args)
    {
        double area, r = 10;
        String s1 = "The Area of a circle with ";
        String s2 = " r = ";
        String s3 = " is ";
        String s4;
        area = Math.PI*Math.pow(r,2);
        s4 = s1.concat(s2);
        System.out.println(s4+area);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
c:>javac AreaOfCircle.java
c:>java AreaOfCircle
The Area of a circle with r = 314.1592653589793
c:>_
```

This Java program calculates the area of a circle with radius r , where r equals 10. On line 10, we calculate the area by multiplying `Math.PI` with `Math.pow(r,2)`. The former expression refers to the π value that is defined in a constant value names `PI` in the `Math` class. The later is the activation of a method called `pow()` that is also defined in the `Math` class. `pow(r,2)` computes the square of r . Notice that we do not need to create an object of the `Math` class but we access the data and method from the name of the class directly.

On line 11, we make use of a method called `concat()`. It is accessed from a variable that contains a `String` object. `s1.concat(s2)` returns a `String` object resulting from the concatenation of the `String` object in `s1` and the `String` object in `s2`. Also, on line 11, the concatenated `String` object is assigned to `s4`.

Useful String methods

Let's look at some methods that we can use from a `String` object. The methods discussed here do not make the complete list of the methods provided by `String`. Examples are given so that you can see what each method does as well as practice your code reading skill.

charAt()

Let `s` be a `String` object and `i` be an `int`. `s.charAt(i)` returns the `char` value at the i^{th} index.

length()

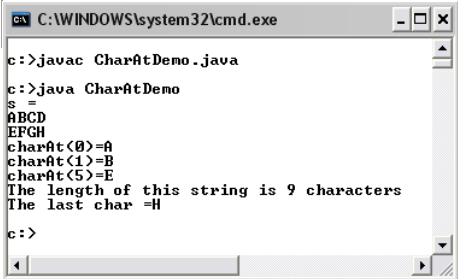
Let `s` be a `String` object. `s.length()` returns the `int` value equals to the length of the `String`.

Consider the following Java program.

```
public class CharAtDemo
{
    public static void main(String[] args)
    {
        String s = "ABCD\neFGH";
        char c;
        System.out.println("s = ");
        System.out.println(s);
        c = s.charAt(0);
    }
}
```



```
System.out.println("charAt(0)=" + c);           10
c = s.charAt(1);                               11
System.out.println("charAt(1)=" + c);           12
c = s.charAt(5);                               13
System.out.println("charAt(5)=" + c);           14
System.out.print("The length of this string is ") 15
System.out.println(s.length() + " characters"); 16
c = s.charAt(s.length() - 1);                  17
System.out.println("The last char =" + c);      18
}                                                19
}                                                20
```



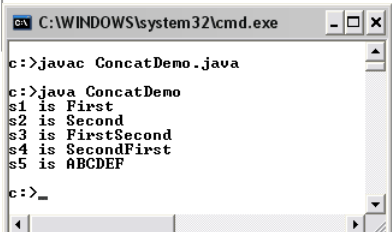
From the above Java program, the *String* *s* contains 9 characters, which are 'A', 'B', 'C', 'D', '\n', 'E', 'F', 'G', and 'H'. Notice that an escape sequence is considered a single character. On line 9, line 11, and line 13, the characters at 0, 1, and 5 which are 'A', 'B' and 'E' are assigned to the *char* variable *c*. Then, *c* is printed out to the screen after each assignment. On line 16, and line 17, the length of the *String* in *s* is extracted via the method *length()*. Be aware that, the first index of a *string* is 0, so the location of the last character is *s.length() - 1*.

concat()

Let *s* be a *String* object and *r* be another *String* object. *s.concat(r)* returns a new *String* object whose content is the concatenation of the *String* in *s* and *r*.

Consider the following example.

```
public class ConcatDemo                         1
{                                                2
    public static void main(String[] args)      3
    {                                            4
        String s1 = "First";                   5
        String s2 = "Second";                  6
        String s3, s4;                         7
                                                8
        s3 = s1.concat(s2);                    9
        s4 = s2.concat(s1);                   10
        System.out.println("s1 is " + s1);    11
        System.out.println("s2 is " + s2);    12
        System.out.println("s3 is " + s3);    13
        System.out.println("s4 is " + s4);    14
                                                15
        String s5 = "AB".concat("CD").concat("EF"); 16
        System.out.println("s5 is " + s5);    17
    }                                           18
}                                               19
```





Notice the difference between `s1.concat(s2)` and `s2.concat(s1)`. Also note that invoking the method `concat()` from a *String* `s` creates a new *String* object based on `s` and the *String* input into the parentheses, it does not change the value of the original *String* object. On line 16, we show two things. Firstly, we can invoke *String* methods directly from a *String* object without having to be referred to by a variable, i.e. `"AB".concat("CD")` can be done without any errors. Secondly, since `"AB".concat("CD")` results in a new *String* object, we can call a *String* method from it directly, e.g. `"AB".concat("CD").concat("EF")`, and the result is `"ABCDEF"`, as expected.

indexOf()

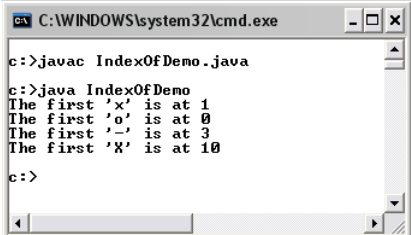
Let `s` be a *String* object and `c` be a `char` value. `s.indexOf(c)` returns the index of the first `c` appearing in the *String*. It returns -1 if there is no `c` in the *String*. If `i` is an `int` value equals to the Unicode value of `c`, `s.indexOf(i)` returns the same result. A *String* `r` can also be used in the place of `c`. In that case, the method finds that *String* inside the *String* `s`. If there is one, it returns the index of the first character of `r` found in the *String* `s`. Again, it returns -1 if `r` is not found in `s`.

lastIndexOf()

`lastIndexOf()` works similarly to `indexOf()` but it returns the index of the last occurrence of the input character or the index of the first character in the rightmost occurrence of the input *String*.

Consider the following examples. Note that the Unicode of `'-'` is 45. Also, make sure you remember that Java is a case-sensitive language.

```
public class IndexOfDemo                                1
{                                                        2
    public static void main(String[] args)              3
    {                                                    4
        String s = "oxx-xo--xoXo";                     5
        System.out.println("The first 'x' is at "+s.indexOf('x')); 6
        System.out.println("The first 'o' is at "+s.indexOf('o')); 7
        System.out.println("The first '-' is at "+s.indexOf(45)); 8
        System.out.println("The first 'X' is at "+s.indexOf('X')); 9
    }                                                    10
}                                                        11
```



```
C:\WINDOWS\system32\cmd.exe
c:\>javac IndexOfDemo.java
c:\>java IndexOfDemo
The first 'x' is at 1
The first 'o' is at 0
The first '-' is at 3
The first 'X' is at 10
c:\>
```

```
public class IndexOfDemo2                                1
{                                                        2
    public static void main(String[] args)              3
    {                                                    4
        String s = "Chulalongkorn University";          5
        System.out.println(s);                          6
        System.out.println("Univ is at "+s.indexOf("Univ")); 7
        System.out.println("0123 is at "+s.indexOf("0123")); 8
    }                                                    9
}                                                        10
```



```
C:\WINDOWS\system32\cmd.exe
c:\>javac IndexOfDemo2.java
c:\>java IndexOfDemo2
Chulalongkorn University
Univ is at 14
0123 is at -1
c:\>_
```

```
public class IndexOfDemo3
{
    public static void main(String[] args)
    {
        String s = "say ABC ABC ABC";
        System.out.println(s);
        System.out.println("lastIndexOf('\\B\\') =" + s.lastIndexOf('B'));
        System.out.println("lastIndexOf(\"AB\") =" + s.lastIndexOf("AB"));
    }
}
```

1
2
3
4
5
6
7
8
9
10

```
C:\WINDOWS\system32\cmd.exe
c:\>javac IndexOfDemo3.java
c:\>java IndexOfDemo3
say ABC ABC ABC
lastIndexOf('\\B\\') =13
lastIndexOf(\"AB\")=12
c:\>
```

startsWith()

Let *s* be a *String* object and *prefix* be another *String* object. *s.startsWith(prefix)* returns *true* if the *String* *s* starts with *prefix*. Otherwise, it returns *false*.

endsWith()

Let *s* be a *String* object and *suffix* be another *String* object. *s.endsWith(suffix)* returns *true* if the *String* *s* ends with *suffix*. Otherwise, it returns *false*.

trim()

Let *s* be a *String* object. *s.trim()* returns a new *String* object, which is a copy of *s*, but with leading and trailing whitespaces omitted.

Consider the following example.

```
public class TrimDemo
{
    public static void main(String[] args)
    {
        String s1 = " Computer Engineering ";
        String prefix = "Computer";
        String suffix = "ing";
        System.out.print("\""+s1+"\" has \""+prefix);
        System.out.println("\n as a prefix:\t"+s1.startsWith(prefix)+".");
        System.out.print("\""+s1+"\" has \""+suffix);
        System.out.println("\n as a suffix:\t"+s1.endsWith(suffix)+".");

        String s2 = s1.trim();
        System.out.print("\""+s2+"\" has \""+prefix);
        System.out.println("\n as a prefix:\t"+s2.startsWith(prefix)+".");
        System.out.print("\""+s2+"\" has \""+suffix);
        System.out.println("\n as a suffix:\t\t"+s2.endsWith(suffix)+".");
    }
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19



```
C:\WINDOWS\system32\cmd.exe
c:>javac TrinDemo.java
c:>java TrinDemo
" Computer Engineering " has "Computer" as a prefix: false.
" Computer Engineering " has "ing" as a suffix: false.
"Computer Engineering" has "Computer" as a prefix: true.
"Computer Engineering" has "ing" as a suffix: true.
c:>
```

The String `s1` contains one whitespace character at the beginning and three of them at the end. A new *String* object is created with these leading and trailing whitespace characters trimmed before being assigned to `s2`. Since `s2` contains "Computer" right at the beginning and "ing" right at the end of the *String*, `s2.startsWith(prefix)` and `s2.endsWith(suffix)` return the boolean value true.

substring()

Let `s` be a *String* object. `s.substring(a,b)`, where `a` and `b` are `int` values, returns a new *String* object whose content are the characters of the *String* `s` from the `a`th index to the `(b-1)`th index. If `b` is omitted the substring runs from `a` to the end of `s`.

toLowerCase()

Let `s` be a *String* object. `s.toLowerCase()` returns a new *String* object which is a copy of `s` but with all uppercase characters converted to lowercase.

toUpperCase()

Let `s` be a *String* object. `s.toUpperCase()` returns a new *String* object which is a copy of `s` but with all lowercase characters converted to uppercase.

Consider the following example.

```
public class SubstringDemo                                1
{                                                         2
    public static void main(String[] args)                3
    {                                                       4
        String s = "Sir Isaac Newton";                     5
        System.out.println(s.substring(10));                6
                                                           7
        int startIdx = 4;                                   8
        int len = 5;                                       9
        System.out.println(s.toUpperCase().substring(startIdx,startIdx+len)); 10
    }                                                       11
}                                                         12
```

```
C:\WINDOWS\system32\cmd.exe
c:>javac SubstringDemo.java
c:>java SubstringDemo
Newton
ISAAC
c:>
```

valueOf()

`valueOf()` is a static or class method provided by the *String* class. It creates a new *String* object whose value is the corresponding *String* representation of the value input to the method. Recall that to use a class method, we use the dot operator with the name of the class.

Reading Input String from Keyboards

It is usually a common requirement to obtain values from the user of the program via keyboards. In Java, this capability is provided by some methods, already defined in classes. A



class called *BufferedReader* provides a method that read characters from keyboard input, until a newline character is found, and store the characters into a *String* object. This method is called *readLine()*. Note that the newline character (`\n`) signaling the end of the input is not included in the *String*.

First, since we are going to use the *BufferedReader* class, which is not in the standard Java packages, we need to let the compiler know where to look for the definition of this class by adding the following statement in to our source code on a line prior to the start of our program's definition.

```
import java.io.*;
```

Then, we need to create an object of class *BufferedReader* by using the following statement.

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
```

This statement creates a variable named `stdin` that refers to a *BufferedReader* object. For simplicity, we will say that `stdin` is a *BufferedReader* object. It is perfectly fine that you use exactly this statement to create a *BufferedReader* object. Detailed explanation is omitted here.

Once a *BufferedReader* object is created, we can access the *readLine()* method from that object. For example, we can use the following statement to read keyboard input to a *String* object called `s`. Note that `stdin` is the object we created in the previous statement.

```
String s = stdin.readLine();
```

Once the statement is executed, the program waits for the user to type in the input until a newline character is entered. This input can be used later in the program from the *String* `s`.

The following program asks the user to input his/her first and last name. Then it prints a message containing the names on to the screen. Notice that another thing that is required to be added is `throws IOException` in the header of the *main()* method. Again, explanation is omitted until you learn about *exceptions* in Java. At this time, make sure you do not forget to add it in your program when *readLine()* is used in the *main()* method.

```
import java.io.*;                                     1
public class Greeting                                 2
{                                                       3
    public static void main(String[] args) throws IOException 4
    {                                                   5
        String firstname, lastname;                   6
        BufferedReader stdin =                         7
            new BufferedReader(new InputStreamReader(System.in)); 8
        System.out.print("Please enter your firstname:"); 9
        firstname = stdin.readLine();                  10
        System.out.print("Please enter your lastname:"); 11
        lastname = stdin.readLine();                   12
        System.out.println("-----");                13
        System.out.println("Hello "+firstname+" "+lastname); 14
        System.out.println("-----");                15
    }                                                   16
}                                                       17
```

```
C:\WINDOWS\system32\cmd.exe
c:>javac Greeting.java
c:>java Greeting
Please enter your firstname:Atiwong
Please enter your lastname:Suchato
Hello Atiwong Suchato
c:>
```




In this example, expressions in green (lighter-colored) text are what you need to pay attention to. On line 1, the `import` statement tells the compiler about a location that it should look if there appear to be non-standard methods. On line 7 and line 8, a `BufferedReader` object, which we name it `stdin`, is created using the statement mentioned earlier. On line 10 and line 12, the method `readLine()` is used to bring in the keyboard inputs. It is a common practice that messages are shown prior to the execution of `readLine()` in order to instruct users about what they should be doing. Such messages are shown using the `print()` methods on line 9 and line 11.

Converting Strings to numbers

Since the `readLine()` method returns a `String` object and sometimes we expect the keyboard input to be numeric data so that we can process numerically, we need a way to convert a `String` object to an appropriate numeric value. Luckily, Java has provided methods responsible for doing so.

`parseInt()`

`parseInt()` is a static method that takes in a `String` object and returns an `int` whose value associates with the content of that `String`. `parseInt()` is defined in a class called `Integer`. Thus, we should know by now that calling a static method named `parseInt()` from the `Integer` class takes the form: `Integer.parseInt(s)`, where `s` is a `String` object whose content we wish to convert to `int`.

`parseDouble()`

`parseDouble()` is a static method that takes in a `String` object and returns an `double` whose value associates with the content of that `String`. `parseDouble()` is defined in a class called `Double`. Again, calling `parseDouble()` takes the form: `Double.parseDouble(s)`, where `s` is a `String` object whose content we wish to convert to `double`.

Useful Methods and Values in Class `Integer` and Class `Double`

It is necessary to know that `Integer` is a class, not the primitive type `int`, and `Double` is another class, not the primitive type `double`. Furthermore, it might come in handy if you know some of the constants and static methods provided in these two classes (Apart from `parseInt()` and `parseDouble()`, of course).

Here are some of them.

`Integer.MAX_VALUE`

is an `int` holding the maximum value an `int` can have ($2^{31}-1$).

`Integer.MIN_VALUE`

is an `int` holding the minimum value an `int` can have (-2^{31}).

`Integer.toBinaryString(<an int>)`

returns a `String` of the `int` argument as an unsigned integer in base 2.

`Integer.toOctalString(<an int>)`

returns a `String` of the `int` argument as an unsigned integer in base 8.

`Integer.toHexString(<an int>)`

returns a `String` of the `int` argument as an unsigned integer in base 16.



`Integer.toString(<an int>)`
returns the *String* representation of the `int` argument.

`Double.MAX_VALUE`
is the largest positive finite value of type `double`.

`Double.MIN_VALUE`
is the smallest positive nonzero value of type `double`.

`Double.NaN`
is a Not-a-Number (`NaN`) value of type `double`.

`Double.POSITIVE_INFINITY`
is the positive infinite value of type `double`.

`Double.NEGATIVE_INFINITY`
is the negative infinite value of type `double`.

`Double.isInfinite(<a double>)`
returns `true` if the `double` argument is infinitely large in magnitude.

`Double.isNaN(<a double>)`
returns `true` if the `double` argument is an `NaN` value.

`Double.toHexString(<a double>)`
returns the hexadecimal *String* of the `double` argument.

`Double.toString(<a double>)`
returns the *String* representation of the `double` argument.

Example

The program `ShowBinary.java` shown below is used for showing the binary representation of an `int` input by the user. Make sure you go through the program and try to understand all of the statements.

```
import java.io.*;
public class ShowBinary
{
    public static void main(String[] args) throws IOException
    {
        String readStr;
        int i;
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter an integer:");
        readStr = stdin.readLine();
        i = Integer.parseInt(readStr);
        System.out.println("Binary -> "+Integer.toBinaryString(i));
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
c:>javac ShowBinary.java
c:>java ShowBinary
Enter an integer:129
Binary -> 10000001
c:>_
```



Example

Let's look at the following Java program called FunnyEncoder.java. This program uses only what we have learnt so far. The program converts a 4-digit string (E.g. 0345, 1829, etc.) into a specific code by mapping each digit to a specific funny pattern defined by the following rules.

0 → (^_^)	4 → (O_o)	8 → (@_@)
1 → (-_-)	5 → (^v^)	9 → (*_*)
2 → (>_<)	6 → (^o^)	
3 → (o_o)	7 → (^_____^)	

For example, if the input digit string is 0123, the encoded string is (-_-)(>_<)(o_o)(o_o). Here is the source code for the program and some example outputs. Make sure you go through the program and try to understand all of the statements.

```
import java.io.*;
public class FunnyEncoder
{
    public static void main(String[] args) throws IOException
    {
        int loc;
        String input, output = "", s = "";
        s += "(^_^) ";
        s += "(-_-) ";
        s += "(>_<) ";
        s += "(o_o) ";
        s += "(O_o) ";
        s += "(^v^) ";
        s += "(^o^) ";
        s += "(^_____^) ";
        s += "(@_@) ";
        s += "( *_* ) ";

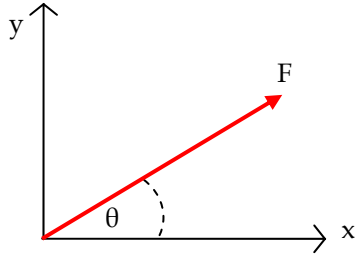
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter a 4-digit string:");
        input = stdin.readLine();
        loc = 9*Integer.parseInt(input.substring(0,1));
        output += s.substring(loc,loc+9).trim();
        loc = 9*Integer.parseInt(input.substring(1,2));
        output += s.substring(loc,loc+9).trim();
        loc = 9*Integer.parseInt(input.substring(2,3));
        output += s.substring(loc,loc+9).trim();
        loc = 9*Integer.parseInt(input.substring(3));
        output += s.substring(loc,loc+9).trim();
        System.out.println("Encoded String -> "+output);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
c:>javac FunnyEncoder.java
c:>java FunnyEncoder
Enter a 4-digit string:0183
Encoded String -> (^_^)(-_-)(@_@)(o_o)
c:>java FunnyEncoder
Enter a 4-digit string:9999
Encoded String -> ( *_* )( *_* )( *_* )( *_* )
c:>java FunnyEncoder
Enter a 4-digit string:2613
Encoded String -> (>_<)(^o^)(-_-)(o_o)
c:>java FunnyEncoder
Enter a 4-digit string:0505
Encoded String -> (^_^)(^o^)(^_^)(^v^)
c:>
```



Example

Now we wish to write a program that calculates the resulting force in the x and y directions, as illustrated in the figure below, from the magnitude of the input force F (in Newton) and the angle between F and the x axis (in Degree).



Problem definition: The program needs to calculate the force in the x and y directions from the magnitude of the input force, F, and the angle, θ .

Analysis: There are two inputs, F and θ . Output, which are the force components in the two directions, are to be shown on screen.

Design:

- Prompt the user to input F, and store the input in `f`.
- Prompt the user to input θ , and store the input in `theta`.
- Convert θ , which is in degree, to radian by $\theta_{\text{radian}} = \theta_{\text{degree}} \times \frac{\pi}{180}$. Then, store the converted angle in `thetaRad`.
- Calculate the force component in the x direction from $F_x = F \cdot \cos(\theta_{\text{radian}})$. Then, store the result in `fx`.
- Calculate the force component in the y direction from $F_y = F \cdot \sin(\theta_{\text{radian}})$. Then, store the result in `fy`.
- Show the `fx` and `fy` on the screen.

Implementation:

```
import java.io.*;
public class FindFComponents
{
    public static void main(String[] args) throws IOException
    {
        double theta, f, thetaRad, fx, fy;
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        // prompt for f
        System.out.print("Enter the magnitude of F (Newton): ");
        f = Double.parseDouble(stdin.readLine());
        // prompt for theta
        System.out.print("Enter the angle between F and the x axis (Degree): ");
        theta = Double.parseDouble(stdin.readLine());
        // convert degree to radian
        thetaRad = theta*Math.PI/180;
        // calculate fx and fy
        fx = f*Math.cos(thetaRad);
        fy = f*Math.sin(thetaRad);
        // show the results
        System.out.println("Fx = "+fx+" N");
        System.out.println("Fy = "+fy+" N");
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
c:>javac FindFComponents.java
c:>java FindFComponents
Enter the magnitude of F (Newton): 5
Enter the angle between F and the x axis (Degree): 30
Fx = 4.330127018922194 N
Fy = 2.4999999999999996 N
c:>java FindFComponents
Enter the magnitude of F (Newton): 5
Enter the angle between F and the x axis (Degree): 0
Fx = 5.0 N
Fy = 0.0 N
c:>_
```

Exercise

- Write valid Java statements that perform the following steps.
 - Declare a variable for storing a *String*. Name it `s1`.
 - Have `s1` refer to a new *String* object whose content is "Java".
 - Declare another variable named `s2` and have it refer to a new *String* object whose content is "Programming".
 - Print the concatenation of `s1` and `s2` on screen.
- Explain in your own words the functionality of the two *dot* operators in the statement `System.out.print("I love eating!");`.
- Given that *Calendar* is a valid class in Java and `c` is a variable referring to an object of the *Calendar* class, which of the following expression involve calling a method in the *Calendar* class. And, which ones simply access some data in the *Calendar* class. (Ignore their meanings for now.)
 - `Calendar.DECEMBER`
 - `Calendar.getInstance()`
 - `Calendar.getAvailableLocales()`
 - `c.isTimeSet`
 - `Calendar.MILLISECOND`
 - `c.clear()`
 - `c.get(1)`
- Write a Java program that calculates and shows the areas and circumferences of three circles, each of which has its radius of 3, 100, and 8.75 centimeters.
- What is the output of the following code segment?

```
String s = "tachygraphometry";
System.out.println(s.charAt(1));
System.out.println(s.charAt(5));
System.out.println(s.charAt(12));
System.out.println(s.charAt(s.length()-1));
```

- What is the output of the following code segment?

```
String s1 = "macaroni penguin";
String s2 = s1.substring(s1.indexOf(' ') + 1, s1.length()).toUpperCase();
System.out.println(s1);
System.out.println(s2);
```



7. What is the output of the following code segment?

```
String s1 = "Houston";
String s2 = "Dallas".concat(s1);
s1 = s2.substring(2,4);
System.out.println(s1.length());
System.out.println(s2.length());
```

8. What is the output of the following code segment?

```
String s = "Jacobian";
System.out.println(s.indexOf('J'));
System.out.println(s.indexOf('c'));
System.out.println(s.indexOf('a'));
System.out.println(s.indexOf('j'));
System.out.println(s.indexOf('c'-1));
```

9. What is the output of the following code segment?

```
String s1 = "A";
String s2 = s1+1;
char c = 'A';
String s3 = c+1+"A";
System.out.println(s2.concat(s3).concat(s1));
```

10. What is the output of the following code segment?

```
String s = "1999";
System.out.println(String.valueOf(s));
System.out.println(String.valueOf(s)+1);
System.out.println(String.valueOf(s+1));
```

11. Explain why the *String* class is available to our program without the use of an *import* statement and why an *import* statement is required when we want to use the *BufferedReader* class in our program.
12. Write a Java program that prompts for and accepts a text message from the user via keyboard and prints it out on screen.
13. Write a Java program that prompts for two text messages from the user via keyboard, connect them together, and print the result on screen.
14. Write a Java program that prompts for and accepts a telephone number of the form xx-xxx-xxxx where each x is a digit (e.g. 02-123-9999), and prints it out in the following form: x-xxxx-xxxx (e.g. 0-2123-9999).
15. Write a Java program that prompts for and accepts an email address and prints the associated account's name and domain name in two separate lines. For example, 2140101@gmail.com should be printed out as:

```
2140101
gmail.com
```

16. Write a Java program that prompts for and accepts two numbers, *a* and *b*, via keyboard, and prints out the results of the following numeric computation:

$$a+b, a \times b, \frac{a}{b}, a^b, \text{ and } \sqrt[b]{a}$$