Scanner

The Scanner class is intended to be used for input.

It takes input and splits it into a sequence of **tokens**. A token is a group of characters which form some unit.

For example, suppose the input is
one two three
4 five 7.2

to be clear, this is actually
one<space>two<space>three<return>4<space>five<space>7.2<return>

A scanner splits this into the following tokens
one
two
three
4
five
7.2
All of these are just sequences of one or more characters. One of them (4) can be converted to an int. One (7.2) can be converted into a double. The others, like one, are just Strings.

Note, input units (tokens) are not the same as lines. For example
one two three
4 five 7.2
has 6 tokens, in two lines.

A Scanner splits the input into tokens using some delimiter, which by default is whitespace. Whitespace is one or more spaces, tab characters or newlines.

Scanner also has methods to convert those tokens to primitives using the various 'next' methods. One issue then is what happens if the actual input data does not match with the expected types.

AScanner also uses regular expressions.

Getting runtime input is a pretty basic process which beginners want to do. But Java was not designed as a language for beginners. As a result, Scanner can do a lot more than basic functions.

Basic use

You normally create a Scanner instance like

```
Scanner scanner = new Scanner(System.in);
```

This would need
import java.util.Scanner;
since the Scanner class is in the package java.util

This reads input from System.in, which is the keyboard. Other constructors take input from other sources, such as files.

When asked to do input, a Scanner
1. Skips any leading whitespace
2. Reads an input token up to the next whitespace
3. It might convert the token to an int or whatever.

Note in step 2 it does not use up the next whitespace - that is still in the input stream. The Scanner will block - that is, wait for more input, if it is needed.

It is natural to think of input as a set of lines, separated by newlines. It is better to think of it as a stream, with data separated by whitespace - newlines, spaces and tabs.

Some basic methods are then:

next()	Get next token, as a String
nextInt()	Get next token and convert to an int
nextDouble, nextFloat, nextLong etc	Get next token and convert to double, float etc
nextLine()	Get whole next line, as a String. This might include spaces etc

For example

```
   Scanner scanner = new Scanner(System.in);
   String word = scanner.next();
   int number = scanner.nextInt();
   double decimal = scanner.nextDouble();
   String line1 = scanner.nextLine();
   String line2 = scanner.nextLine();
   System.out.println(word+ " : "+ number +" : "+decimal+" : "+line1 + " : "+line2);
```

sample input:

```
hello 12
9.1
1 2 3 four
```

and output

```
hello : 12 : 9.1 :  : 1 2 3 four
```

What happened?

The input was in fact this stream:

```
hello<space>12<new line>9.1<new line>1 2 3 four<new line>
```

This happened:

Statement	Input stream after	Value obtained
	hello<space>12<new line>9.1<new line>1 2 3 four<new line>	
String word = scanner.next();	<space>12<new line>9.1<new line>1 2 3 four<new line>	word=”hello”
int number = scanner.nextInt();	<new line>9.1<new line>1 2 3 four<new line>	number=12
double decimal = scanner.nextDouble();	<new line>1 2 3 four<new line>	decimal=9.1
String line1 = scanner.nextLine();	1 2 3 four<new line>	line1=””
String line2 = scanner.nextLine();		line2=”1 2 3 four”

So nextLine() is slightly different. It ignores the use of whitespace as the default delimiter. Instead it just takes the input up to and including the next newline. Often the input stream already has a newline 'left over' from the previous input, so that a nextLine reads an empty string.

However nextLine is convenient for reading input line by line, so that each line can then be processed somehow. This is very useful for reading text files.

Input type mismatch

Methods like nextInt expect to input a given datatype. Suppose the actual input data is the wrong type? Like this:

```
   Scanner scanner = new Scanner(System.in);
   int number = scanner.nextInt();
   System.out.println(number);
```

and when we run it:

```
run:
hello
Exception in thread "main" java.util.InputMismatchException
	at java.util.Scanner.throwFor(Scanner.java:857)
	at java.util.Scanner.next(Scanner.java:1478)
	at java.util.Scanner.nextInt(Scanner.java:2108)
	at java.util.Scanner.nextInt(Scanner.java:2067)
	at test2.Test2.main(Test2.java:10)
Java Result: 1
```

so we get an InputMisMatchException - the input data does not match what the code expected.

What to do? For keyboard input, we can ask the user to try again:

```
        boolean tryAgain = true;
        do {
            try {
                int number = scanner.nextInt();
                System.out.println(number);
                tryAgain = false;
            } catch (InputMismatchException ex) {
                System.out.println("Please enter a whole number");
                scanner.nextLine();
            }
        } while (tryAgain);
```

```
run:
q
Please enter a whole number
e
Please enter a whole number
45
45
```

If an exception is thrown, the try block is discarded, and it is as if the nextInt never happened. So we need the nextLine in the catch block to use up the incorrect input.

If the invalid data is in a file, the programmer must decide what they want to happen in the case of reading a corrupt file.

When?

This is what a scanner does - but when does it do it? For example:

```
        Scanner scanner = new Scanner(System.in);

        int number;
        do {
            number = scanner.nextInt();
            System.out.print("*");
        } while (number != -1);
```

```
run:
1 2 3 4
****5 6
**7 -1
**
```

So this is inputting a sequence of ints until a -1 is read. But it does no input until a newline is reached. We has
1 2 3 4<new line>
and then it read in four ints, and outputted four *s. Then another two, the 5 and 6, then the 7 and -1.

The Scanner is taking input from System.in, which is an implementation of the InputStream class. The basic method of this is read() to read a byte, and the API says “This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.” So input from the console is only available after <newline> is pressed.

Input from a File

Suppose we have a file named data.txt which contains:
one
two
three

We can read this by having a Scanner on an input stream from this file, rather than System.in:

```
class Test2 {
    Test2()
    {

            InputStream input = getClass().getResourceAsStream("data.txt");
            Scanner scanner = new Scanner(input);
            try
            {
            while (true)
            {
                String str = scanner.nextLine();
                System.out.println(str);
            }
            }
            catch (NoSuchElementException ex)
            {
                scanner.close();
            }

    }

    public static void main(String args[]) {
        Test2 test = new Test2();
    }

}
```

This expects to find data.txt alongside the Java file. The getClass method is non-static, so we instantiate the class and run it in the constructor. When nextLine hits end of file it throws NoSuchElementException, which we catch and close the scanner.

Changing the delimiter

Suppose our data is a comma separated list of ints, possibly on several lines. For example, if data.txt is
1,2,3,4,5,
6,7,8,9,
One way to read that would be to change the token delimiter from whitespace to a comma:

```
        InputStream input = getClass().getResourceAsStream("data.txt");
        Scanner scanner = new Scanner(input);
        scanner.useDelimiter(",");
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        boolean finished = false;
        while (!finished) {
            try {

                {
                    Integer num = scanner.nextInt();
                    numbers.add(num);
                }
            } catch (InputMismatchException ime) { // found newline
                scanner.nextLine(); // onto next line
            } catch (NoSuchElementException ex) {
                scanner.close(); // end of file
                finished = true;
            }
        }
        for (Integer num : numbers) {
            System.out.print(num+" ");
        }
```

Using regular expressions

Several of the Scanner methods use regular expressions (which maybe you need to read about).

For example, suppose our input data consists of some words, and some integers, like this:
the 65 28 7 cat 8 sat 734
 on 91 the 87 mat
There is a method next(String) which gets the next token which matches the String treated as a regular expression. So for example:

```
        InputStream input = getClass().getResourceAsStream("data.txt");
        Scanner scanner = new Scanner(input);
        boolean finished = false;
        while (!finished) {
            try {
                String str = scanner.next("[a-z]+"); // one or more letters a to z
                System.out.print(str+" ");
            } catch (InputMismatchException ime) { // found non-match
                scanner.next(); // onto next token
            } catch (NoSuchElementException ex) {
                finished = true; // end of file
                scanner.close();

            }
        }
```

which outputs

```
the cat sat on the mat
```