java.util.Scanner

`java.util.Scanner` is a class in the Java API used to create a `Scanner` object, an extremely versatile object that you can use to input alphanumeric characters from several input sources and convert them to binary data..

The following three constructors allow you to scan data from the standard input object (`System.in`), from string objects (especially ones provided by an input dialog box) and from external text files.

Three Constructors of java.util.Scanner

`public Scanner( InputStream source )`
`// Creates a scanner that scans values from the input stream.`

`public Scanner( String source )`
`// Creates a scanner that scans values from the string.`

`public Scanner( File source )`
`// Creates a scanner that scans values from the external file.`

Here are some of the many features of `Scanner` objects.

Some Features of java.util.Scanner
They can scan data of all the primitive data types (e.g. `int`, `double`, etc.) except `char`.
They can scan strings.
They easily scan multiple tokens from a single input source.
They can change the characters used to delimit tokens in the input.
They can specify the precise pattern of legal input.
They generate an `InputMismatchException` if the input does not conform to the expected data type, allowing you to use `try` and `catch` constructs to validate the user's input.

Scanning from the Standard Input Object

The first `Scanner` constructor allows you to create a scanner for the standard input object `System.in`, which is an object of class `InputStream`.

Example

The following Java code reads two floating-point values from the standard input object. Below shows the user interaction (the underlined text shows what the user types; the program doesn't underline anything). To the right is a picture of memory after the input values are scanned.

weight	2.0
height	10.0

```
Enter your weight and height: 120 66
```

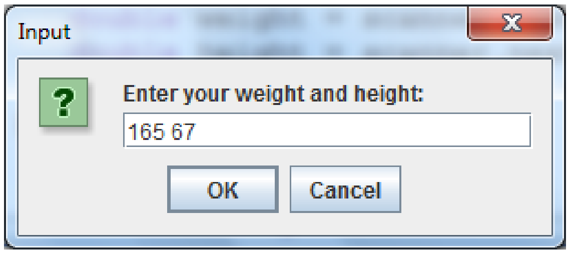
```
import java.util.Scanner;
. . .
double weight, height; // patient stats
Scanner in = new Scanner( System.in );
System.out.print( "Enter your weight and height: " );
weight = in.nextDouble( );
height = in.nextDouble( );
```

Scanning from a String

The second `Scanner` constructor allows you to create a scanner for a string, which you can use to scan data input from a dialog box.

Example

This Java code gets its input from the dialog box shown to the right.



Enter your weight and height:

165 67

```
import java.util.Scanner;
import javax.swing.JOptionPane;
. . .
String prompt = "Enter your weight and height: ";
String input = JOptionPane.showInputDialog( prompt );
Scanner in = new Scanner( input );
double weight = in.nextDouble( );
double height = in.nextDouble( );
```

Scanning from an External Text File

The third **Scanner** constructor allows you to create a scanner that gets its input from a file of text residing on the external disk.

Example

Suppose you have an external file named **data.txt** that contains the values you wish to input. See the picture to the right.

data.txt

```
165 67
```

The following Java code opens the file as a scanner and inputs the two values. It assumes that the data file resides in the same folder as the application's class file.

```
import java.util.Scanner;
import java.io.File;
   . . .
Scanner in = new Scanner( new File( "data.txt" ) );
double weight = in.nextDouble( );
double height = in.nextDouble( );
```

Scanning Primitive Data and Strings

`java.util.Scanner` has methods to scan all the primitive data types except `char`. In addition, it has two methods that scan strings.

Scanner Methods that Scan Primitive Data	

Scans floating-point data	`public double nextDouble( )`
	`public float nextFloat( )`

Scans integer data	`public byte nextByte( )`
	`public int nextInt( )`
	`public long nextLong( )`
	`public short nextShort( )`

Scans `true` and `false`	`public boolean nextBoolean( )`

Scanner Methods that Scan Strings	
Method	**What it Does**
`next( )`	Skips leading white space, collects alphanumeric characters until encountering trailing white space, returns the characters collected as a string.
`nextLine( )`	Collects all characters (visible and invisible) until encountering the next newline character and returns the characters collected as a string.

To scan a primitive value, call the appropriate method within the `Scanner` object and store the scanned value into a variable of the same primitive data type.

Example

```
import java.util.Scanner;
. . .
Scanner in = new Scanner( System.in );
double salary = in.nextDouble( );
boolean isMarried = in.nextBoolean( );
int numberChildren = in.nextInt( );
```

To scan a string, you must a declare reference variable for the **`String`** object that is to be returned. You don't need to initialize it – the object is built and returned by the method.

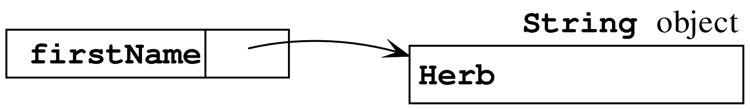
Example

```
1  import java.util.Scanner;
   . . .
2  Scanner in = new Scanner( System.in );
3  System.out.print( "Enter name and address:  " );
4  String firstName = input.next( );
5  String lastName = input.next( );
6  String address = input.nextLine( );
```

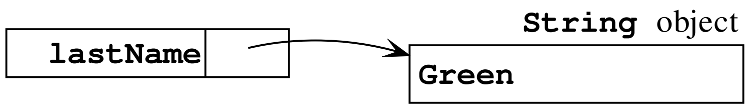
Suppose in response to the user prompt, the user enters the underlined text shown below:

```
     Enter name and address: Herb Green 150 Maple St.
```

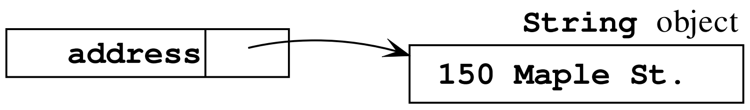
The input is read in three parts. The call to **`next`** on line 4 reads **`Herb`** and ends because of the trailing space. The string reference is placed into **`firstName`**:



Line 5 reads **`Green`** and places it into a string object referenced by **`lastName`**:



Line 6 uses a call to **`nextLine`** to read the entire address including its embedded spaces. **`nextLine`** scans characters until reaching the newline character. The scanned string is placed into a string object referenced by **`address:`**



Specifying Token Delimiters

By default a **`Scanner`** object expects tokens to be delimited by ***white space*** (i.e. spaces, horizontal tabs and newline characters). You can change the delimiter by calling these **`Scanner`** methods:

Scanner Methods to Change Token Delimiters

`Scanner useDelimiter( String pattern )`
`// Set the scanner's token delimiter to the characters specified`
`// by the 'pattern' string.`

`Scanner reset( )`
`// Restores the scanner's token delimiter to its default.`

Although these methods return a **`Scanner`** object, it's the same as the altered scanner so there's no reason to save it.

Example

Line 3 in the Java code below tells the scanner to use a single comma as a token delimiter. For example, the input string

```
     Herb Green,150 Maple St,Kansas City
```

Would be divided into these three tokens:

```
     Herb Green
     150 Maple St
     Kansas City
```

```
1  import java.util.Scanner;
   . . .
2  Scanner in = new Scanner( System.in );
3  in.useDelimiter( "," );
```

A ***pattern*** matches a set of string literals. It looks very much like a string literal itself but some of its characters are ***metacharacters*** that represent different combinations of characters. A full explanation of patterns is an entire topic in itself. Following are some simple, but useful, token delimiting patterns.

Pattern	**What it Means**
`"` *character* `"`	*character* is any character literal (including an escape sequence). The pattern matches the character
`"[` *characters* `]"`	*characters* is a sequence of characters and `[` and `]` are metacharacters. The pattern matches any *single* character in the sequence.
`"[` *characters* `]+"`	`+` is a metacharacter. The pattern matches one or more occurrences of each of the *characters*.

Examples	
This pattern	***Matches***
`" "`	A *single* space
`"\n"`	A *single* newline character
`","`	A *single* comma
`"[, ]"`	Any *single* comma or space
`"[ \n]"`	Any *single* space or newline character
`"[/-]"`	Any *single* slash or hyphen
`"[.,;]"`	Any *single* period, comma or semi-colon
`"[ ]+"`	One or more spaces
`"[, ]+"`	Any combination of one or more commas or spaces

Making the Newline Character Portable

The pattern `"\n"` stands for the newline character, which, unfortunately, is not standard across operating systems. For example, a new line is marked in Microsoft Windows with a combination of two Unicode characters – `\u000D` followed by `\u000A` – whereas MacOS uses a single `\u000`. This can create problems when trying to move code from one platform to another.

To avoid this problem, use the static method **`lineSeparator`** that is found in the API class **`java.lang.System.`**

```
static String lineSeparator( )
// Returns the system-dependent line separator as a string.
```

You can use it by itself or use concatenation to embed it within a more complicated pattern.

Examples	
This pattern	*Matches*
`System.lineSeparator( )`	A single newline character
`"[," + System.lineSeparator( ) + "]"`	A single comma or a single newline character
`"[ " + System.lineSeparator( ) + "]+"`	A combination of one or more spaces and newline characters

Specifying Input Patterns

You can also use a pattern, along with the following method, to specify the precise format of legal input.

```
String next( String pattern )
// Returns the next token if it matches the pattern.
```

If the next input token doesn't match the pattern then the method throws an **`InputMismatchException.`**

Example
This Java code reads a vehicle's license plate number consisting of three digits followed by three upper-case letters and terminated by the system dependent line separator.

```
import java.util.Scanner;
. . .
Scanner in = new Scanner( System.in );
in.useDelimiter( System.lineSeparator( ) );
String input = in.next( "[0-9]{3}[A-Z]{3}" );
```

For a more detailed description of Java patterns, you should refer to the API specification for class **`java.util.regex.Pattern`**.

Input Validation

Generally, **`Scanner`** methods throw an **`InputMismatchException`** if the next input token does not conform to what is expected.

Example
The following Java code, if the user were to enter the input **`$2,125.50`**, would halt with the run-time error:

```
Exception in evaluation thread java.util.InputMismatchException
```

```
import java.util.Scanner;
. . .
Scanner in = new Scanner( System.in );
System.out.print( "Enter salary: " );
double salary = in.nextDouble( );
```

This behavior enables to you, by using Java's **`try`** and **`catch`** constructs, to control what happens in such situations so that your program responds to bad input in a user-friendly manner.

Example

The following Java code uses a loop to trap user input errors. The loop (lines 7 through 21) continues to cycle so long as the user's input is not a valid floating-point number. If the user enters valid input, the loop quits.

```
 1  import java.util.Scanner;
    import java.util.InputMismatchException;
 2  . . .
 3  double salary;
 4  Scanner in = new Scanner( System.in );
 5  in.useDelimiter( System.lineSeparator( ) );
 6  boolean inputOK = false; // set flag to indicate bad input
 7  while ( ! inputOK )      // while flag indicates bad input
 8  {
 9     try
10     {
11        System.out.println( "Enter salary: " );
12        salary = in.nextDouble( );  // may throw exception
13        inputOK = true;             // no exception thrown
14     }
15     catch ( InputMismatchException ex )
16     {  // exception thrown
17        System.out.println( "Bad input. Try again." );
18        inputOK = false;
19        in.nextLine( );  // drop line separator in input
20     }
21  }
```

Exercises

In each problem below, the first two statements are correct; there is something wrong with method call in the third statement. Circle what's wrong and explain. None of them is correct.

1.
```
import java.util.Scanner;
. . .
Scanner input = new Scanner( System.in );
double x = input.next( );
```

2.
```
import java.util.Scanner;
. . .
Scanner input = new Scanner( System.in );
int m = input.nextDouble( );
```

3.
```
import java.util.Scanner;
. . .
Scanner input = new Scanner( System.in );
int m = nextInt( );
```

4.
```
import java.util.Scanner;
. . .
Scanner input = new Scanner( System.in );
int m = input.nextint( );
```

5.
```
import java.util.Scanner;
. . .
Scanner input = new Scanner( System.in );
String line = input.next;
```

6.
```
import java.util.Scanner;
. . .
Scanner input = new Scanner( System.in );
String line = input.nextline( );
```

Enter the application given below into jGRASP, save it to a file and compile it. For each of the exercises that follow, run the application, enter the input given and explain why the program works or doesn't work.

```java
import java.util.Scanner;

public class Scanning
{
   public static void main( String [] args )
   {
      // declare data
      String name;   // child's first name
      double height; // child's height
      int age;       // child's age
      // build Scanner
      Scanner input = new Scanner( System.in );
      // prompt for and read data
      System.out.println( "Name? Age? Height?" );
      name = input.next( );
      age = input.nextInt( );
      height = input.nextDouble( );
      // print results
      System.out.println( "Name: " + name );
      System.out.println( "Age: " + age );
      System.out.println( "Height: " + height );
   }
}
```

7.	`Tom 12 4.75`
8.	`Tom Jones 12 4.75`
9.	`Tom 12.5 4.75`
10.	`Tom 12 4`

11.	Modify application **`Scanning`** given above so that it will read a child's name that has embedded spaces.
12.	Modify application **`Scanning`** given above so that it reads all three values from a single **`JOptionPane`** input dialog.

If you modified application **`Scanning`** according to previous exercises, then retrieve the original shown on page 12. Add the following after line 12:

```
input.useDelimiter( "," );
```

For each of the following exercises, run the application, enter the input given and explain why the program works or doesn't work.

13.	**`Tom Jones,12,4.75`**
14.	**`Tom Jones,12,4.75,`**

Continuing with the **`Scanning`** application from exercises 19 and 20, change line 13 to:

```
input.useDelimiter( "[," + System.lineSeparator( ) + "]" );
```

For the following exercise, run the application, enter the input given and explain why the program works or doesn't work.

15.	**`Tom Jones,12,4.75`**

For each of the following exercises: (a) Write the Java statements to scan the described values from the standard input object. (b) Write the Java statements to input and scan the described values from a single **`JOptionPane`** input dialog.

In each case, you may have to change the scanner delimiter to something appropriate for the situation.

16.	Input an **`int`** value into a variable named **`score`**.
17.	Input a person's height (a **`double`**), weight (a **`double`**) and age (an **`int`**). Declare appropriate variables for the values.
18.	Input a string and store its reference into **`phone`**. The string has no embedded spaces.
19.	Input a string and store its reference into **`title`**. The title may have embedded spaces and is followed by the newline character.
20.	Input a company's department name (a string with no embedded spaces) and its budget (a **`double`**).
21.	Input a company's department name (this time allowing the department name to have embedded spaces such as INFORMATION SYSTEMS) and its budget (a **`double`**).
22.	Input a person's name (a string that may have embedded spaces), height (a **`double`**), weight (a **`double`**) and age (an **`int`**). Declare appropriate variables for the values.
23.	Scan a person's name (a string that may have embedded spaces), address (another string that may have embedded spaces), marital status (**`true`** or **`false`**) and number of tax exemptions (an **`int`**). Declare appropriate variables for the values.
24.	Scan hours, minutes, seconds in the form *hh:mm:ss*. Read the values into three **`int`** variables.