As already mentioned, the `print` and `println` methods are used for formatted output. Any type can be converted to a `String` suitable for printing by calling its `toString` method; in many cases, this is done automatically. Unlike with C and C++, which have an enormous number of formatting options, output in Java is done almost exclusively by `String` concatenation, with no built-in formatting.

### 2.6.2 the Scanner type

The simplest method for reading formatted input is to use a `Scanner`. A `Scanner` allows the user to read lines one at a time using `nextLine`, `Strings` one at a time using `next`, or `primitive` types one at a time using methods such as `nextInt` and `nextDouble`. Prior to attempting to perform a read, it is customary to verify that the read can succeed by using methods such as `hasNextLine`, `hasNext`, `hasNextInt`, and `hasNextDouble`, which yield boolean results; as such there is typically less need to deal with exception handling. When using a `Scanner` it is customary to provide the import directive

```java
import java.util.Scanner;
```

To use a `Scanner` that reads from the standard input we must first construct a `scanner` object from `System.in`. This was illustrated in Figure 2.11 at line 7. In Figure 2.11, we see that `nextLine` is used to read a `String`, and then the `String` is converted to an `int`. From the discussion of `Scanner` in the previous paragraph, we know there are several other options.

An alternative option, which is perhaps the cleanest, is the following replacement, which avoids exceptions entirely, and uses `nextInt` and `hasNextInt`:

```java
System.out.println( "Enter an integer: " );
if( in.hasNextInt( ) )
  {  
    x = in.nextInt();
    System.out.println( "Half of x is " + ( x / 2 ) );
  }
else
  { System.out.println("Integer was not entered." )
```
Using the various `next` and `hasNext` combinations from the `Scanner` generally works well, but can have some limitations. For instance suppose we want to read two integers and output the maximum.

Figure 2.15 shows one idea that is cumbersome if we want to do proper error checking, without using exceptions. Each call to `nextInt` is preceded by a call to `hasNextInt`, and an error message is reached unless there are actually two `int`s available on the standard input stream.

Figure 2.16 shows an alternative that does not use calls to `hasNextInt`. Instead, calls to `nextInt` will throw a `NoSuchElementException` if the `int` is not available, and this makes the code read cleaner. The use of the exception is perhaps a reasonable decision because it would be considered unusual for the user to not input two integers for this program.

Both options, however, are limited because in many instances, we might insist that the two integers be entered on the same line of text. We might even insist that there be no other data on that line. Figure 2.17 shows a different option. A `Scanner` can be constructed by providing a `String`. So we can first create a `Scanner` from `System.in` (line 7) to read a single line (line 12), and then create a second `Scanner` from the single line (line 13) to extract the two
class MaxTestB
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter 2 ints:");
        try
        {
            int x = in.nextInt();
            int y = in.nextInt();
            System.out.println("Max: \ + Math.max(x, y)";
            catch(NoSuchElementException e)
            { System.err.println("Error: need two ints"); }
        }
    }
}

import java.util.Scanner;
public class MaxTestC
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter 2 ints on one line:");
        try
        {
            String oneLine = in.nextLine();
            Scanner str = new Scanner(oneLine);
            int x = str.nextInt();
            int y = str.nextInt();
            System.out.println("Max: \ + Math.max(x, y)";
            catch(NoSuchElementException e)
            { System.err.println("Error: need two ints"); }
        }
    }
}
Class `java.util.Random`

- `Random()`
  This constructs a new random number generator.

- `double nextDouble()`
  This method returns the next pseudorandom, uniformly distributed floating-point number between 0.0 (inclusive) and 1.0 (exclusive) from this random number generator's sequence.

  **Returns:** The next pseudorandom floating-point number

- `int nextInt(int n)`
  This method returns the next pseudorandom, uniformly distributed integer between 0 (inclusive) and the specified value (exclusive) drawn from this random number generator's sequence.

  **Parameters:** n Number of values to draw from

  **Returns:** The next pseudorandom integer

Class `java.util.Scanner`

- `Scanner(InputStream in)`
- `Scanner(Reader in)`
  These construct a scanner that reads from the given input stream or reader.

  **Parameters:** in The input stream or reader from which to read

- `boolean hasNext()`
- `boolean hasNextDouble()`
- `boolean hasNextInt()`
- `boolean hasNextLine()`
  These methods test whether it is possible to read any non-empty string, a floating-point value, an integer, or a line, as the next item.

  **Returns:** true if it is possible to read an item of the requested type, false otherwise (either because the end of the file has been reached, or because a number type was tested and the next item is not a number)

- `String next()`
- `double nextDouble()`
- `int nextInt()`
- `String nextLine()`
  These methods read the next whitespace-delimited string, floating-point value, integer, or line.

  **Returns:** The value that was read

Interface `java.util.Set<E>`

This interface describes a collection that contains no duplicate elements.