Reading Keyboard Input 3

3.1 Class Scanner

3.1.1 Importing Source Codes

The class Scanner enables reading data from the keyboard, a String object, or a text file using the characters (the white space character ' ', the tab-stop '\t', and the newline '\n') as separators (such characters are called **delimiters**). The character sequences separated by the delimiters can be read not only as String data but also as data of other types (when such interpretations are possible). The class will play an important role in the programs presented in this book. The source code for Scanner is available as part of JDK but not part of java.lang. Therefore, to write a source code for a program that uses Scanner, the source code of Scanner must be included. To include Scanner, an **import statement** must be used. If a class FooBar uses Scanner, the declaration must take the following form:

```
import java.util.Scanner;
public class FooBar
{
...
}
```

Here, java.util is the group called **package** that contains the source code for Scanner.

The source code library of JDK is a hierarchical (tree-like) structure. We call the top-level of the hierarchy the **root**. Underneath the root exists a number of **projects**. A project is a collection of **packages**, where a package is a collection of classes and/or hierarchies of classes serving common purposes. Popular projects include java and javax. Popular packages in the project java are lang, io, and util. The standard Java package lang belongs to the project java. The period . appearing in the import statement refers to these hierarchical relations, so java.util.Scanner means:

"the class Scanner in the package util of the project java".

Some packages have sub-packages, and so the actual number of times that the separator . appears in an import statement is more than two for some packages.

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_3

Multiple import statements may appear in the header. For example, the following code:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.LinkedList;
import java.io.File;
import java.io.FileNotFoundException;
class FooBar
{
...
}
```

imports five distinct classes. Multiple import statements for classes from the same package can be substituted with a universal import statement.

```
import java.util.*;
class FooBar
{
...
}
```

The asterisk, meaning "everything", is called the **wildcard**.

3.1.2 The Instantiation of a Scanner Object

To use the functions of the class Scanner, one needs a Scanner object. A Scanner object is created through a process called **instantiation**. The instantiation of an object of a type CLASS_NAME takes the form of:

```
new CLASS_NAME( PARAMETERS )
```

where CLASS_NAME is a special method called **constructor**. Each object class has its own constructors. The name of a constructor matches the name of the class it belongs to. Here, PARAMETERS represents the information given to the instantiation procedure. Many classes, including Scanner, accept more than one kind of parameter specification. In this book, we study three Scanner constructors: one that takes a String object, one that takes System.in, which refers to the keyboard, and one that takes a File object, which refers to a file.[1] The next code fragment uses the three Scanner constructors and assign them to Scanner variables.

```
Scanner strScanner, fileScanner, keyboard;
strScanner = new Scanner( "My GPA is 4.00!" );
fileScanner = new Scanner( new File( "theFile.txt" ) );
keyboard = new Scanner( System.in );
```

[1] Although we do not use them in this book, String has many constructors (for various examples, see https://docs.oracle.com/javase/7/docs/api/java/lang/String.html).

The first Scanner object strScanner scans the String literal "My GPA is 4.00!", the second Scanner object fileScanner scans a file named "theFile.txt", and the last Scanner object keyboard scans input from the keyboard. We will study the last type extensively in this chapter.

3.2 Reading Data with a Scanner Object

Regardless of whether the input source may be a String, a File, or System.in, the chief function of a Scanner is to read from the input source using a delimiter sequence as a separator. Scanner does this by scanning the input character sequence from the beginning to the end and discovering the longest stretch of characters free of delimiters. The delimiters are our usual suspect of spacing characters: the white space character, the tab-stop, and the newline. We call such a delimiter-free stretch of characters a **token**. When reading from a String and when reading from a File, the contents of the input are fixed, so dividing the contents into tokens and delimiters is easy. When reading from System.in, dividing the contents into tokens is a dynamic process, since the user can use the delete key (or backspace key) to change the contents dynamically until the return key is pressed, upon which no more changes are permitted.

Imagine that a String object myStringSource has contents

```
" Programming\t\tis fun, \nisn't it?\n"
```

where \t and \n are the tab-stop, and the newline and the other gaps consist of the white space characters. We can break this String data into an alternating series of delimiters and tokens, as follows:

1. a delimiter " ",
2. a token "Programming",
3. a delimiter "\t\t",
4. a token "is",
5. a delimiter " ",
6. a token "fun,",
7. a delimiter "\n",
8. a token "isn't",
9. a delimiter " ",
10. a token "it?"

If we have instantiated a Scanner object myInput from this String literal, then myInput produces the five tokens in order of appearance.

To read data from the input source of a Scanner object, we apply a method to the object. Formally, a method applied to an object data is called an **instance method**. To execute an instance method on an object of an object class, we attach a period and the name of the method to the object, and then attach a pair of parentheses after the method name. Each execution of the method is called a **method call**.

This period has the same role as the second periods in `System.out.println` and `System.out.print`, since `System.out` is an object of type `PrintStream` (which we will study in Chap. 15).

The instance method used for reading a token as a `String` data from the input source of a `Scanner` object is called `next`. Here is a code that uses the method:

```java
String myStringSource;
Scanner myInput;
myStringSource = "  Programming\t\tis   fun, \nisn't it?";
myInput = new Scanner( myStringSource );
myInput.next();
```

`myInput.next()` produces the first token of the `String` object, `"Programming"`. If there is no action that utilizes this token, the token disappears. The token that `next` produces in a `String` variable can be saved via assignment; that is, after

```java
String myStringSource, myFirstToken;
Scanner myInput;
myStringSource = "  Programming\t\tis   fun, \nisn't it?";
myInput = new Scanner( myStringSource );
myFirstToken = myInput.next();
```

the variable `firstToken` has the value `"Programming"`.

Earlier we quickly went over the return type of a method. In the case of the method `next` of `Scanner`, its return type is `String`. In the case of the method `next`, the method comes back with a `String`. We can obtain that value by assigning it to a `String` value. In a similar manner, we can place the method call in a `System.out.println`, as shown next:

```java
myInput = new Scanner( myStringSource );
System.out.println( myInput.next() );
```

The `System.out.println` receives the return value of `myInput.next()` and prints it on the screen. Since the first token retrieved is `"Programming"`, the code produces the output:

```
Programming
```

By executing the `next()` four more times, we are able to retrieve all the tokens appearing in `mySourceText`. For example,

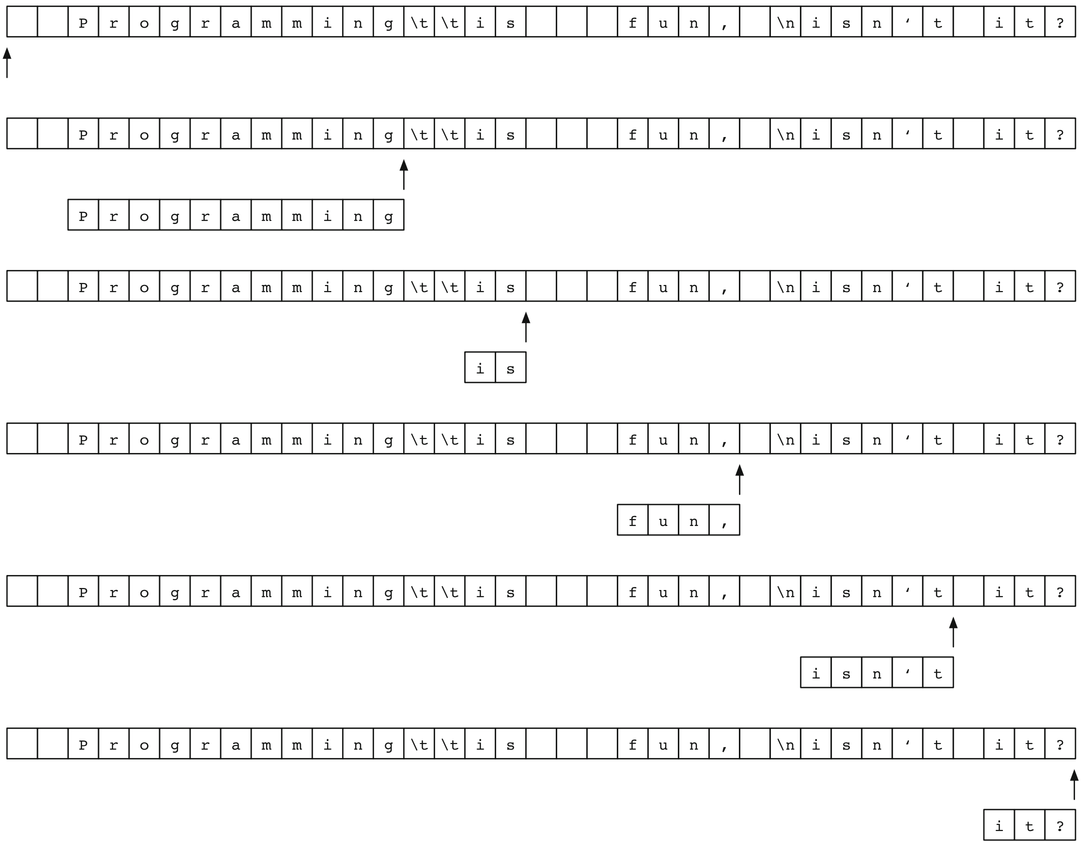

Fig. 3.1 The results of five consecutive calls of `next`. The arrows show the start positions of the `Scanner` object for the next read. The sequences immediately to the left of the arrows are the tokens that have been read

```
String myStringSource, token1, token2, token3, token4, token5
Scanner myInput;
myStringSource = "  Programming\t\tis   fun, \nisn’t it?";
myInput = new Scanner( myStringSource );
token1 = myInput.next();
token2 = myInput.next();
token3 = myInput.next();
token4 = myInput.next();
token5 = myInput.next();
```

retrieves the five tokens as five variables in the order they appear.

The results of the five calls of `next` are shown in Fig. 3.1.

It is impossible to scan beyond the last token. If an attempt is made to read beyond the last token, a run-time error of `NoSuchElementException` occurs. The code appearing in the next source code demonstrates an attempt to read beyond the last token, as well as the resulting run-time error.

```java
import java.util.Scanner;
public class BeyondLimit
{
  public static void main( String[] args )
  {
    String aToken, tokens = new String( "My age is 20" );
    Scanner keyboard = new Scanner( tokens );
    aToken = keyboard.next();
    System.out.println( aToken );
    aToken = keyboard.next();
    System.out.println( aToken );
    aToken = keyboard.next();
    System.out.println( aToken );
    aToken = keyboard.next();
    System.out.println( aToken );
    aToken = keyboard.next();
    System.out.println( aToken );
  }
}
```

Listing 3.1 A program that attempts to read a token that does not exist

Since there are five `keyboard.next()` calls and there are only four tokens, an error occurs at the fifth `keyboard.next()` call, as shown next:

```
1 My
2 age
3 is
4 20
5 Exception in thread "main" java.util.NoSuchElementException
6         at java.util.Scanner.throwFor(Scanner.java:862)
7         at java.util.Scanner.next(Scanner.java:1371)
8         at BeyondLimit.main(BeyondLimit.java:16)
```

The fifth line of the output is the start of the error message. The type of the error appears, at the end, as `java.util.NoSuchElementException`.

It is vital to prevent attempts to read beyond the last token. When reading from the keyboard, because the texts are generated dynamically and the length of the text is indefinite, we usually do not encounter the error. (The error does not occur unless the user simultaneously presses the CTRL-key and the letter `'d'` to indicate the end of the input. This combination is called CTRL-D. We will study the use of CTRL-D in Chap. 11.)

When reading tokens from a `String` data or from a file, however, some proactive measures are needed to prevent the error from happening. There are three possible approaches:

(a) know beforehand how many tokens are in the input source;
(b) check for the existence of a token before attempting to read;
(c) recover from `NoSuchElementException` using a special mechanism, called `try-catch`.

We will study the second approach in Chap. 11 and the third in Chap. 15.

Table 3.1 Selected methods of Scanner

Name	Return type	Action
next	String	Obtains the next token
nextBoolean	boolean	Obtains the next boolean token
nextByte	byte	Obtains the next byte token
nextDouble	double	Obtains the next double token
nextFloat	float	Obtains the next float token
nextInt	int	Obtains the next int token
nextLong	long	Obtains the next long token
nextShort	short	Obtains the next short token
nextLine	String	Obtains the String before the next newline
hasNext	boolean	Checks whether or not the next token exists

All the methods in this table are usually called with an empty parameter, i.e., with () attached after the method names. We will study hasNext in Chap. 11

Using Scanner, we can fetch the next token not only as a String data but also as a token of a specific primitive data type, given that the token can be interpreted as a literal of that type. The type-specific fetch methods are: nextBoolean, nextByte, nextDouble, nextFloat, nextInt, nextLong, and nextShort. Note that there is no method corresponding to reading a char. If one of these methods is called and the next token cannot be interpreted as the type associated with the method, a run-time error of InputMismatchException occurs. For example, suppose the next token in the input sequence is -1.0. The token can be interpreted as a double data, a float data, and a String data, but not as a whole number data or as a boolean data. Thus, the use of nextBoolean, nextByte, nextDouble, nextFloat, nextInt, or nextLong will fail (and a run-time error will subsequently appear).

Table 3.1 presents a list of Scanner methods that appear in this textbook. We will study hasNext in Chap. 11.

There is one particularly interesting "next" method in Scanner and this is nextLine. If the remaining (or upcoming) character sequence has at least one occurrence of the newline character, the nextLine returns the entire character sequence appearing before the first newline character in the upcoming character sequence. After completing nextLine, the Scanner object scans the sequence starting from the character immediately after the identified newline character.

If a Scanner object is instantiated with either a String data or a File object, the remaining sequence may not contain any newline characters. If the nextLine method is called in such a situation, the method returns the String data corresponding to all the remaining characters. After that, no characters remain in the sequence.

Consider the following code:

```
import java.util.*;
public class NextLine
{
  public static void main( String[] args )
  {
    String myStringSource =
        "My lucky number is 17 , \n how about yours?  ";
    Scanner myInput = new Scanner( myStringSource );
    System.out.println( myInput.next() );
    System.out.println( myInput.next() );
    System.out.println( myInput.next() );
    System.out.println( myInput.next() );
    System.out.println( myInput.nextInt() );
    System.out.println( myInput.nextLine() );
    System.out.print( myInput.nextLine() );
    System.out.println( ":::::" );
  }
}
```

Listing 3.2 A program that combines `next` and `nextLine`

The source of the `Scanner` object is the `String` literal

```
"My lucky number is 17 , \n how about yours?  "
```

The fifth token of the input source is the `String` literal `"17"`. We can treat this as either a `String` type or an `int` (or any other number type since 17 can be represented as a `byte` data). The delimiter sequence that follows this number token consists of five characters, represented as `" , \n "`. (the double quotation marks are attached so as to clarify the beginning and the ending of the character sequence. If the method that is called at that point is `next`, then the `Scanner` will skip to the next token `","`. However, if the method that is called is `nextLine` instead, the `"\n"` appearing as the fourth character of the five-character sequence becomes the delimiter. When this happens, the `Scanner` object returns `" , "` as the result of executing `nextLine`, and then the `Scanner` object positions itself to the fifth character of the sequence, i.e., the `" "` after the newline. If `nextLine` and `next` are called successively, the method call to `next` returns `"how"`. If `nextLine` is called twice successively, the first call returns `" , "` as in the previous case, and the second call returns the remainder of the sequence, `" how about yours? "`, because is no additional appearance of `\n`. Based upon this analysis, the output of the program is expected to be:

```
% java NextLine
My
lucky
number
is
17
 ,
 how about yours?  :::::
%
```

The `":::::"` appearing at the end is not part of the `String` literal, but is a marking that this program attaches to indicate the end of its output. Figure 3.2 shows the progress that the `Scanner` object makes during the execution of the program.

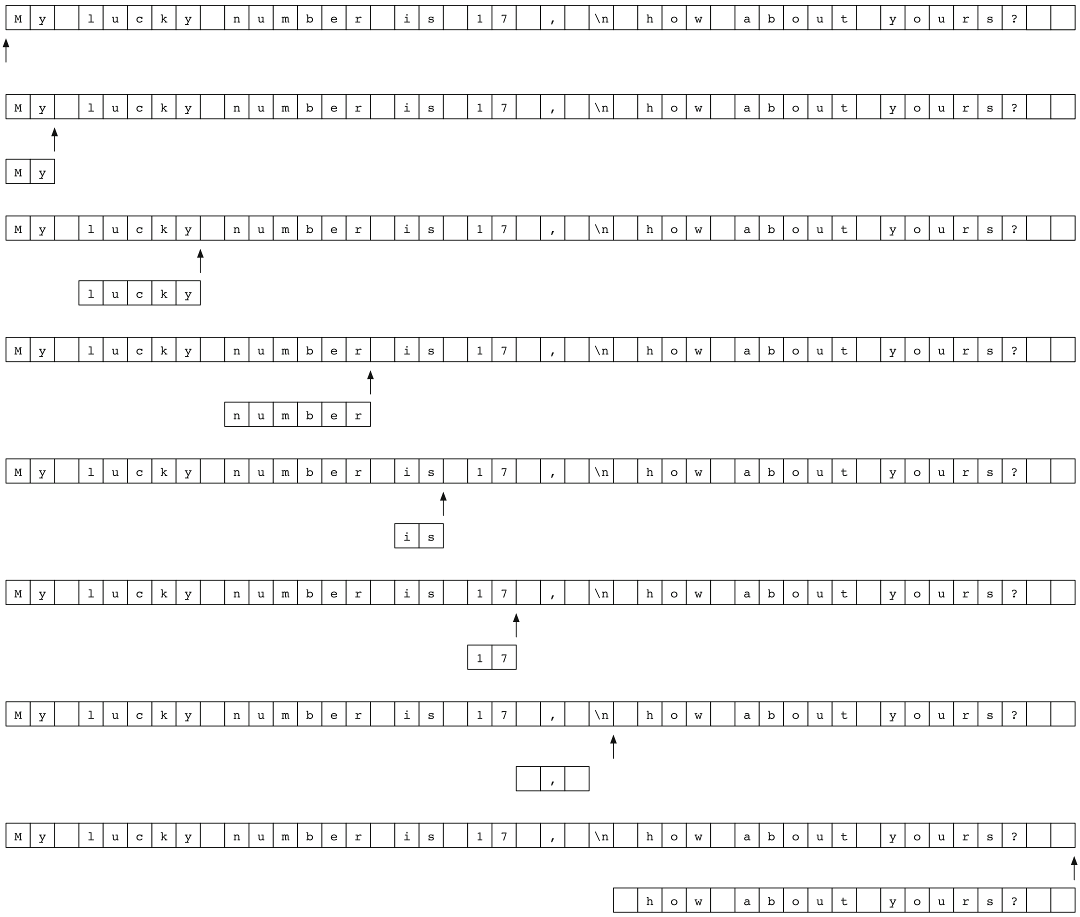
M y l u c k y n u m b e r i s 1 7 , \n h o w a b o u t y o u r s ?
M y
l u c k y
n u m b e r
i s
1 7
,
h o w a b o u t y o u r s ?

Fig. 3.2 The results of executing the program. The arrows show the positions of the `Scanner` object for the next read. The sequences immediately to the left of the arrows are the tokens that have been read. Note that the last two reads are done by `nextLine` so the delimiters other than the newline characters, if any, are included in the returned `String` data that are returned

3.3 Reading Input from the Keyboard

We will now explore how to read data from the keyboard. Note two things:

1. The `Scanner` objects instantiated with `System.in` cannot take action until the return key is pressed. This means that, depending on what actions are awaiting with the `Scanner` object, the single line of input may offer a return value to multiple `next` methods of the `Scanner` object.
2. When reading the output generated by a program that scans keyboard input, the coexistence of the output of the program and the echo of the keyboard input makes it difficult to distinguish between the two different types of screen output.

The next code demonstrates the use of the type-specific token reading methods. The program stores the result of each token it reads in a variable of an appropriate type, and then prints the value.

Notice that `nextLine` appears at the very beginning of the series of calls to `next` methods. This is because if `nextLine` comes after another call to the `next` method, then the result of `nextLine` will be an empty `String`.

Furthermore, when a `Scanner` object is instantiated with `System.in`, because the white space and the tab-stop are erasable after typing, nothing occurs until the return key is entered. For example, if the character sequence `ABC 17 5.4` is typed and the return key is entered, the sequence includes three tokens, but none of the three tokens can be read until the pressing of the return key. Using this

```java
import java.util.Scanner;

public class Nexts
{
  public static void main( String[] args )
  {
    Scanner keyboard;
    boolean myBoolean;
    byte myByte;
    double myDouble;
    float myFloat;
    int myInt;
    long myLong;
    short myShort;
    String myString;
    keyboard = new Scanner( System.in );
    // Nextline
    System.out.print( "Enter any String: " );
    myString = keyboard.nextLine();
    System.out.println( "You've entered: " + myString );
    // String
    System.out.print( "Enter a String with no space: " );
    myString = keyboard.next();
    System.out.println( "You've entered: " + myString );
    // boolean
    System.out.print( "Enter a boolean: " );
    myBoolean = keyboard.nextBoolean();
    System.out.println( "You've entered: " + myBoolean );
    // byte
    System.out.print( "Enter a byte: " );
    myByte = keyboard.nextByte();
    System.out.println( "You've entered: " + myByte );
    // double
    System.out.print( "Enter a double: " );
    myDouble = keyboard.nextDouble();
    System.out.println( "You've entered: " + myDouble );
    // float
    System.out.print( "Enter a float: " );
    myFloat = keyboard.nextFloat();
    System.out.println( "You've entered: " + myFloat );
    // int
    System.out.print( "Enter an int: " );
    myInt = keyboard.nextInt();
    System.out.println( "You've entered: " + myInt );
    // long
    System.out.print( "Enter a long: " );
    myLong = keyboard.nextLong();
    System.out.println( "You've entered: " + myLong );
  }
}
```

Listing 3.3 A program that demonstrates the use of various “next” methods of `Scanner`

information, if the user knows what types of tokens the program is expecting to receive, she can enter multiple tokens in succession.

In the next code, the program expects three input tokens from the keyboard, a `String`, an `int`, and a `double`. The program receives these input tokens and then prints them. Knowing what happens in the code, the user can type `ABC 10 4.5` to enter all the tokens required by the program at once.

```java
import java.util.Scanner;
// an example of using Scanner
public class ScannerExample
{
  public static void main( String[] args )
  {
    Scanner console;
    String theWord;
    int theWholeNumber;
    double theRealNumber;
    console = new Scanner( System.in );
    System.out.print( "Enter a string: " );
    theWord = console.next();
    System.out.print( "Enter an int: " );
    theWholeNumber = console.nextInt();
    System.out.print( "Enter a double: " );
    theRealNumber = console.nextDouble();
    System.out.print( "You have entered " );
    System.out.print( theWord );
    System.out.print( ", " );
    System.out.print( theWholeNumber );
    System.out.print( ", and " );
    System.out.print( theRealNumber );
    System.out.println();
  }
}
```

Listing 3.4 A program that reads a `String` data, an `int` data, and a `double` data using methods of `Scanner`

The next program asks the user to enter two integers and then two real numbers. The program then multiplies the two integers and divides the first real number by the second. Finally, the program produces the output showing the results of the arithmetic operations.

```java
import java.util.Scanner;
// an example of using Scanner
public class ScannerMath
{
  public static void main( String[] args )
  {
    Scanner console;
    int int1, int2, product;
    double real1, real2, quotient;

    console = new Scanner( System.in );

    System.out.print( "Enter int no. 1: " );
    int1 = console.nextInt();
    System.out.print( "Enter int no. 2: " );
    int2 = console.nextInt();
    product = int1 * int2;

    System.out.print( "Received " );
    System.out.print( int1 );
    System.out.print( " and " );
    System.out.println( int2 );
    System.out.print( "The product is " );
    System.out.println( product );

    System.out.print( "Enter double no. 1: " );
    real1 = console.nextDouble();
    System.out.print( "Enter double no. 2: " );
    real2 = console.nextDouble();
    quotient = real1 / real2;

    System.out.print( "Received " );
    System.out.print( real1 );
    System.out.print( " and " );
    System.out.println( real2 );
    System.out.print( "The quotient is " );
    System.out.println( quotient );
  }
}
```

Listing 3.5 A program that performs arithmetic operations on the numbers received from the user

Again, when running this code, all the numbers can be typed at once. Such “ahead of the game” typing saves the wait time.

Summary

- We construct an object by calling a constructor `new CLASS_NAME( PARAMETERS )`.
- A class may have more than one constructor, with each taking a unique set of parameters.
- To use a `Scanner` object in the source code, the class `Scanner` must be imported.
- To execute a method on an object, the method name along with parameters must be attached to the object, with a period before the method name.
- `Scanner` offers a variety of token reading methods: `next`, `nextBoolean`, `nextByte`, `nextDouble`, `nextFloat`, `nextInt`, `nextLine`, `nextLong`, and `nextShort`.

- An attempt to read beyond the last token results in a run-time error.
- The keyboard echo and the output of the program share the same screen.

Exercises

1. **Terminology test**
   (a) To be able to use a `Scanner`, which class must be imported?
   (b) To create a `Scanner` object that receives input from the keyboard, what statement is needed?
   (c) Write the names of the `Scanner` methods necessary for receiving a `String`, an `int`, and a `double`, respectively.
   (d) When a user types a floating point number into a location where the program has just called the `nextInt` method of a `Scanner` method, will an error occur? If so, what kind of error is it?
   (e) Does an `int` value of 15 pass for a `double`?
   (f) What is the formal term that refers to the process of creating an object of an object type?
   (g) What is the special keyword used in the source code when creating an object?
   (h) For one primitive data type, the class `Scanner` does not have the `next` method designated to read tokens of that type. Which type is this?
2. **Scanning errors** Consider the following program:

```java
import java.util.*;
public class SimpleInputOutput
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    String word;
    double real;
    int value;
    System.out.print( "Enter a real number (for a double): " );
    real = keyboard.nextDouble();
    System.out.print( "Enter a word: " );
    word = keyboard.next();
    System.out.print( "Enter an integer (for an int): " );
    value = keyboard.nextInt();
    System.out.print( "Your have entered: " );
    System.out.println( real + ", " + word + ", " + value );
  }
}
```

Suppose the user is considering the following key strokes when the code is compiled and run, where the strokes are presented as `String` literals, with `\t` and `\n` representing the tab-stop and newline respectively. Do they have enough tokens to finish the code? Which ones will run without causing an error? For those that lead to an error, where will its error occur, and what is the nature of the error?

(a) `"\t0.5\tprogramming\t10\n"`
(b) `"\t5\tprogramming\t10.5\n"`
(c) `"\n\n5\n\n5\n"`
(d) `" 0 0 0.5"`
(e) `" 0 0 0.5\n"`

3. **Scanning errors when `nextLine` is involved** Consider the following program:

```java
import java.util.*;
public class SimpleInputOutput
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    String word1, word2;
    double real;
    int value;
    System.out.print( "Enter a real number (for a double): " );
    real = keyboard.nextDouble();
    System.out.print( "Enter a word: " );
    word1 = keyboard.nextLine();
    System.out.print( "Enter an integer (for an int): " );
    value = keyboard.nextInt();
    System.out.print( "Enter another word: " );
    word2 = keyboard.nextLine();
    System.out.print( "Your have entered: " );
    System.out.println(
        real + ", " + word1+ ", " + value + ", " + word2 );
  }
}
```

Suppose the user is considering the following key strokes when the code is compiled and run, where the strokes are presented as `String` literals with `\t` and `\n` representing the tab-stop and newline respectively. Do they have enough tokens to finish the code? Which ones will run without causing an error? For those that lead to an error, where will its error and what is the nature of the error?

(a) `"\t0.5\tprogramming\t10\tJava\n"`
(b) `"\t5\tprogramming\n10.5\tJava\n"`
(c) `"5\nprogramming\n10.5\nJava\n"`
(d) `"\n\n5\n\n5\n"`
(e) `" -3 -3\n-3.7 -3.5\n"`
(f) `" 0.5 0.5\n6 6\n"`

Programming Projects

4. **Inferences** Write a program named `Inference` that receives three names, `name1`, `name2`, and `name3`, from the user and prints the following statements.

```
name1 is senior to name2
name2 is senior to name3
so
name1 is senior to name3

name3 is senior to name2
name2 is senior to name1
so
name3 is senior to name1
```

The output of the program is composed of the names entered, `" is senior to `, and `"so"`. Declare the last two as constants. Use three variables to store the entered names. Use these components to produce the output.

5. **Arithmetic short-hand** Write a program named `ArithmeticShortHand` that receives two `double` numbers `x` and `y` from the user and then executes `x *= y`, `y = x/y`, and `x /= y`. The program should print the values of `x` and `y` after each of the three actions. Try running the program with various values for `x` and `y`. Also, see what happens when the value of `y` is set to 0.
6. **Receiving five numbers** Write a program named `FiveNumbers` that prompts the user to enter five whole numbers, receives five numbers (`u, v, x, y`, and `z`) using `nextInt`, computes the product of the five numbers as a `long` data, and produces the values of the five values and the product on the screen. For example, the result could look like:

```
Enter five integers: 12 34 56 78 90
The five numbers are: 12, 34, 56, 78, 90
The product is: 160392960
```

7. **Adding six numbers with four variables** Write a program named `Miser` that prompts the user to enter six whole numbers and produces the following output:

```
% java Miser
Enter six numbers: 1 10 3 4 -12 -6
You've entered: 1 10 3 4 12 -6 with the total sum of 0
```

   Here, the numbers entered are 1, 10, 3, 4, -12, and -6. The program should be written so that it uses only four variables: one variable for the scanner, one variable for the number received, one variable for the total, and one variable for the message. The initial value of the variable storing the total is 0, and the initial value of the variable storing the message is `"You've entered:"`. After these initializations, the program should receive the input value from the user, add the value to the total, and update the value of the message variable by adding one white space followed by the input value.
8. **Favorite football player, an interactive program** Write a program named `FavoriteFootballPlayer` that receives the name, the position, the team, and the jersey number from the user, and then produces the following message on the screen:

```
Enter name: Larry Fitzgerald
Enter position: Wide Receiver
Enter team: The Arizona Cardinals
Enter jersey number: 11
Your favorite football player is Larry Fitzgerald.
His position is Wide Receiver.
He is with The Arizona Cardinals and wears the jersey number 11.
```

   Here, the first four lines show the interaction with the user. Since we do not need to perform math operations, treat all the four values as `String`. The name may have spaces, so read each input using `nextLine`.
9. **Gravity again** Recall that if an object is released to fall, the speed of the object at t seconds after its release is gt and the distance the object has travelled in the t seconds after release is $\frac{1}{2}gt^2$. Here, g is the gravity constant, which is approximately 9.8 (the unit is m/s^2) on Earth. Write a program named `GravityInput` whose method `main` does the following:
   (a) It declares variables `t` for the travel time, `speed` for the speed, and `distance` for the distance traveled;
   (b) It receives a value for `t` from the user;
   (c) It calculates the speed and the distance;
   (d) It prints the calculated speed and distance.
   (e) Repeat the last three steps two more times.

10. **Quadratic evaluation** Write a program named `QuadraticEvaluation` that declares four variables (`a, b, c, x`), receives values for the four from the user, calculates $ax^2 + bx + c$ and $a/x^2 + b/x + c$, and prints the values.
11. **Adding time** Write a program named `AddTime` that receives minutes and seconds from the user four times, and then computes the total in hours, minutes, and seconds. The user must specify the minutes and the seconds separately. Ideally, we want to be able to check if the values are valid (that is, the minutes and the seconds have to be between 0 and 59), but because we have not learned yet how to write code that does this, we will assume that the user always enters valid values. The interactions with the user can be as follows:

```
Enter time #1: 10 50
Enter time #2: 26 35
Enter time #3: 37 30
Enter time #4: 41 50
The total is 1 hours 56 minutes 45 seconds.
```

We may want to print `"hour"` instead of `"hours"` for this particular case, but again, we do not know yet how to identify that the number of hours is 1, so we will thus use `"hours"` throughout. The task can be accomplished as follows:

- Compute the total seconds and the total minutes from the input.
- Add the quotient of the seconds divided by 60 to the minutes and then replace the seconds with the remainder of the seconds divided by 60.
- Set the quotient of the minutes divided by 60 to the hour and then replace the minutes with the remainder of the minutes divided by 60.

Programming Projects

12. **Computing the tax, again** Write a program named `ComputeTaxAndtotalInteractive` that computes the tax and the total in the following manner:

- The program uses an `int` variable `subtotal` to store the subtotal in cents. The program receives the value from the user.
- The program uses a `double` variable `taxPercent` to store the tax rate in percent. The program receives the value from the user.
- The program then computes the tax amount as a whole number in cents, in an `int` variable `tax`. Using the casting of `(int)` , a floating point number can be truncated to a whole number.
- The program then computes the total and stores it in an `int` variable `total`. (Again, this quantity is in cents.)
- The program reports the result of the calculation in dollars and cents for the subtotal, the tax, and the total, and then reports the tax rate.

The output of the code may look like:

```
Enter subtotal in cents: 11050
Enter tax percentage: 5.5
The subtotal = 110 dollars and 50 cents.
The tax rate = 5.5 percent.
The tax = 6 dollars and 7 cents.
The total = 116 dollars and 57 cents.
```

13. **Speeding fines, part 1** In the town of Silver Hollow, the speeding fines are $20 times the mileage beyond the speed limit. For example, if a driver was driving at 36 mph on a 30 mph road, his fine is $120. Write a program named `SpeedingFineNew1` that receives the speed and the limit from the user, and then computes the fine in Silver Hollow. The result could look like:

```
Enter the speed and the limit: 50 35
The fine for driving at 50 mph on a 35 mph road is 300 dollars.
```

14. **Speeding fines, part 2** In the town of Golden Valley, the speeding fines are $15 times the percentage of the speed exceeding the limit. The percentage is rounded down to a whole number. For example, if a driver was driving at 35 mph on a 30 mph road, the percentage of excess is the integer part of `(35 - 30) / 30`, which is 16. Thus, the fine is 16 times $5 = $80. Write a program named `SpeedingFineNew2` that receives the speed and the limit from the user and then computes the fine. The result could look like:

```
Enter the speed and the limit: 35 30
The fine for driving at 35 mph on a 30 mph road is 80 dollars.
```

15. **Fractional difference** Write a program named `FractionalDifference` that receives four integers `a`, `b`, `c`, and `d` from the user, and then computes the difference `(a/b) - (c/d)`. Compute the difference by treating the four numbers as integers and then as floating point numbers. To convert an integer to `double`, use casting of `(double)`, e.g., `(double) a`. The result could look like:

```
Enter the four numbers: 10 3 20 7
The difference is 1 using int and 0.4761904761904763 using double.
```

16. **Volume of a rectilinear box** Write a program named `RectilinearBox` that receives three quantities, `height`, `width`, and `length`, from the user, and then computes the volume of the rectilinear box whose dimensions are represented by these quantities. Assume that these quantities are `double` (so the volume should be `double`).