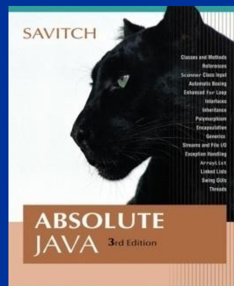
Comp 248
Introduction to Programming
Chapter 2 - *Console Input & Output*

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides has been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.

Copyright © 2007 Pearson Addison-Wesley
Copyright © 2008-2016 Aiman Hanna
All rights reserved



`System.out.println` for console output

- Every invocation of **`println`** ends a line of output

```
System.out.println("The account has
  a balance of " + balance);
```

`println` Versus `print`

- Another method that can be invoked by the **`System.out`** object is **`print`**
- The **`print`** method is like **`println`**, except that it does not end a line
- OutputFormatting1.java (MS-Word file)

Formatting Output with `printf`

- Starting with version 5.0, Java includes a method named **`printf`** that can be used to produce output in a specific format

- **`System.out.printf`** can have any number of arguments
  - The first argument is always a *format string* that contains one or more *format specifiers* for the remaining arguments

- OutputFormatting2.java (MS-Word file)

`printf` Format Specifier

- The code

```
double price = 19.8;
System.out.printf("Price is:
  %10.2f", price);
System.out.println(" each");
```

will output the line

```
  Price is:       19.80 each
```

Right and Left Justification in `printf`

- Specifier such as `%8.2` will *right-justify* the output

- If *left-justification* is needed, then a "`-`" sign is placed after the `%` of the specifier.

```
System.out.printf("Price is: %-10.2f", price);
```

- OutputFormatting3.java (MS-Word file)

Multiple arguments with `printf`

- It is also possible to format different types with **`printf`**
- **`%n`** is used for new line
- **`%%`** is used to escape a specifier
- OutputFormatting4.java (MS-Word file)

Format Specifiers for `System.out.printf`

Display 2.1 Format Specifiers for `System.out.printf`

CONVERSION CHARACTER	TYPE OF OUTPUT	EXAMPLES
d	Decimal (ordinary) integer	%5d %d
f	Fixed-point (everyday notation) floating point	%6.2f %f
e	E-notation floating point	%8.3e %e
g	General floating point (Java decides whether to use E-notation or not)	%8.3g %g
s	String	%12s %s
c	Character	%2c %c

Importing Packages and Classes

- Libraries in Java are called *packages*
  - A package is a collection of classes that is stored in a manner that makes it easily accessible to any program
  - In order to use a class that belongs to a package, the class must be brought into a program using an *import* statement
  - Classes found in the package **`java.lang`** are imported automatically into every Java program

```
import java.text.NumberFormat;
  // import the NumberFormat class only
import java.text.*;
  //import all the classes in package
java.text
```

Console Input Using the `Scanner` Class

- Starting with version 5.0, Java includes a class for doing simple keyboard input named the **`Scanner`** class
- In order to use the **`Scanner`** class, a program must include the following line near the start of the file:

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

- InputScanner1.java (MS-Word file)
- InputScanner2.java (MS-Word file)

Console Input Using the `Scanner` Class

- The method **`next`** reads one string of non-whitespace characters delimited by whitespace

- The method **`nextLine`** reads an entire line of keyboard input

- InputScanner3.java (MS-Word file)

Pitfall: Dealing with the Line Terminator, `'\n'`

- The method **`nextLine`** of the class **`Scanner`** reads the remainder of a line of text starting wherever the last keyboard reading left off
- This can cause problems when combining it with different methods for reading from the keyboard such as **`nextInt, or next`**
- Given the code,

```
Scanner keyboard = new Scanner(System.in);
int n = keyboard.nextInt();
String s1 = keyboard.nextLine();
String s2 = keyboard.nextLine();
```

and the input,

```
2
Heads are better than
1 head.
```

what are the values of **`n`**, **`s1`**, and **`s2`**?

Pitfall: Dealing with the Line Terminator, `'\n'`

- Given the code and input on the previous slide
  `n` will be equal to **`"2"`**,
  `s1` will be equal to **`""`**, and
  `s2` will be equal to **`"heads are better than"`**

- If the following results were desired instead
  `n` equal to **`"2"`**,
  `s1` equal to **`"heads are better than"`**, and
  `s2` equal to **`"1 head"`**
  then an extra invocation of **`nextLine`** would be needed to get rid of the end of line character (**`'\n'`**)

- InputScanner4.java (MS-Word file)

Pitfall: You "should" avoid resource leak - `close()` the Scanner

- You ***should*** avoid leaving an open Scanner object behind when your program exists

- Scanner kb = **new Scanner(System.in);**

  :

  :

  :

  kb.close();

- Java does not enforce it; so you will only get a *warning* if you do not close an opened Scanner object (or objects if you create more than one)
- Nonetheless, you should close it/them before your end your program

Methods in the Class `Scanner` (Part 1 of 3)

Display 2.8 Methods of the Scanner Class

The `Scanner` class can be used to obtain input from files as well as from the keyboard. However, here we are assuming it is being used only for input from the keyboard.

To set things up for keyboard input, you need the following at the beginning of the file with the keyboard input code:

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

You also need the following before the first keyboard input statement:

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

The *Scannner_Object_Name* can then be used with the following methods to read and return various types of data typed on the keyboard.

Values to be read should be separated by whitespace characters, such as blanks and/or new lines. When reading values, these whitespace characters are skipped. (It is possible to change the separators from whitespace to something else, but whitespace is the default and is what we will use.)

Scannner_Object_Name`.nextInt()`

Returns the next value of type `int` that is typed on the keyboard.

(continued)

Methods in the Class Scanner (Part 2 of 3)

Display 2.8 Methods of the Scanner Class

Scannner_Object_Name.nextLong() Returns the next value of type long that is typed on the keyboard.
Scannner_Object_Name.nextByte() Returns the next value of type byte that is typed on the keyboard.
Scannner_Object_Name.nextShort() Returns the next value of type short that is typed on the keyboard.
Scannner_Object_Name.nextDouble() Returns the next value of type double that is typed on the keyboard.
Scannner_Object_Name.nextFloat() Returns the next value of type float that is typed on the keyboard.

(continued)

Methods in the Class `Scanner` (Part 3 of 3)

Display 2.8 Methods of the Scanner Class

Scannner_Object_Name`.next()`

Returns the `String` value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

Scannner_Object_Name`.nextBoolean()`

Returns the next value of type `boolean` that is typed on the keyboard. The values of `true` and `false` are entered as the strings `"true"` and `"false"`. Any combination of upper- and/or lowercase letters is allowed in spelling `"true"` and `"false"`.

Scanner_Object_Name`.nextLine()`

Reads the rest of the current keyboard input line and returns the characters read as a value of type `String`. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.

Scanner_Object_Name`.useDelimiter(`*New_Delimiter*`);`

Changes the delimiter for keyboard input with *Scanner_Object_Name*. The *New_Delimiter* is a value of type String. After this statement is executed, *New_Delimiter* is the only delimiter that separates words or numbers. See the subsection "Other Input Delimiters" for details.

Other Input Delimiters

- The delimiters that separate keyboard input can be changed when using the **`Scanner`** class

- For example, the following code could be used to create a **`Scanner`** object and change the delimiter from white-space to **`"##"`**

```
Scanner keyboard2 = new
  Scanner(System.in);
Keyboard2.useDelimiter("##");
```

- After invocation of the **`useDelimiter`** method, **`"##"`** and not white-space will be the only input delimiter for the input object **`keyboard2`**

- InputScanner5.java (MS-Word file)

Changing the Input Delimiter (Part 1 of 3)

Display 2.10 Changing the Input Delimiter

```
1   import java.util.Scanner;

2   public class DelimiterDemo
3   {
4       public static void main(String[] args)
5       {
6           Scanner keyboard1 = new Scanner(System.in);
7           Scanner keyboard2 = new Scanner(System.in);
8           keyboard2.useDelimiter("##");
9           //Delimiter for keyboard1 is whitespace.
10          //Delimiter for keyboard2 is ##.
```

(continued)

Changing the Input Delimiter (Part 2 of 3)

Display 2.10 Changing the Input Delimiter

```
11            String word1, word2;
12            System.out.println("Enter a line of text:");
13            word1 = keyboard1.next();
14            word2 = keyboard1.next();
15            System.out.println("For keyboard1 the two words read are:");
16            System.out.println(word1);
17            System.out.println(word2);
18            String junk = keyboard1.nextLine(); //To get rid of rest of line.
19
20            System.out.println("Reenter the same line of text:");
21            word1 = keyboard2.next();
22            word2 = keyboard2.next();
23            System.out.println("For keyboard2 the two words read are:");
24            System.out.println(word1);
25            System.out.println(word2);
26        }
27    }
```

(continued)

Changing the Input Delimiter (Part 3 of 3)

Display 2.10 **Changing the Input Delimiter**

SAMPLE DIALOGUE

```
Enter a line of text:
one two##three##
For keyboard1 the two words read are:
one
two##three##
Reenter the same line of text:
one two##three##
For keyboard2 the two words read are:
one two
three
```

Money Formats

- Using the **`NumberFormat`** class enables a program to output amounts of money using the appropriate format
  - The **`NumberFormat`** class must first be *imported* in order to use it

    ```
    import java.text.NumberFormat
    ```

  - An object of **`NumberFormat`** must then be created using the **`getCurrencyInstance()`** method
  - The **`format`** method takes a floating-point number as an argument and returns a **`String`** value representation of the number in the local currency

Money Formats

```
import java.text.NumberFormat;

public class CurrencyFormatDemo
{
  public static void main(String[] args)
  {
    System.out.println("Default location:");
    NumberFormat moneyFormater =
                    NumberFormat.getCurrencyInstance();

    System.out.println(moneyFormater.format(19.8));
    System.out.println(moneyFormater.format(19.81111));
    System.out.println(moneyFormater.format(19.89999));
    System.out.println(moneyFormater.format(19));
    System.out.println();
  }
}
```

Money Formats

- Output of the previous program

```
Default location:
$19.80
$19.81
$19.90
$19.00
```

Specifying Locale

- Invoking the **`getCurrencyInstance()`** method without any arguments produces an object that will format numbers according to the default location

- In contrast, the location can be explicitly specified by providing a location from the **`Locale`** class as an argument to the **`getCurrencyInstance()`** method

  - When doing so, the **`Locale`** class must first be imported

    ```
    import java.util.Locale;
    ```

Specifying Locale

```
import java.text.NumberFormat;
import java.util.Locale;

public class CurrencyFormatDemo
{
  public static void main(String[] args)
  {
    System.out.println("US as location:");
    NumberFormat moneyFormater2 =
      NumberFormat.getCurrencyInstance(Locale.US);

    System.out.println(moneyFormater2.format(19.8));
    System.out.println(moneyFormater2.format(19.81111));
    System.out.println(moneyFormater2.format(19.89999));
    System.out.println(moneyFormater2.format(19));
  }
}
```

Specifying Locale

- Output of the previous program

```
US as location:
$19.80
$19.81
$19.90
$19.00
```

Locale Constants for Currencies of Different Countries

Display 2.4 Locale Constants for Currencies of Different Countries

`Locale.CANADA`	Canada (for currency, the format is the same as US)
`Locale.CHINA`	China
`Locale.FRANCE`	France
`Locale.GERMANY`	Germany
`Locale.ITALY`	Italy
`Locale.JAPAN`	Japan
`Locale.KOREA`	Korea
`Locale.TAIWAN`	Taiwan
`Locale.UK`	United Kingdom (English pound)
`Locale.US`	United States

The `DecimalFormat` Class

- Using the **`DecimalFormat`** class enables a program to format numbers in a variety of ways
  - The **`DecimalFormat`** class must first be *imported*
  - A **`DecimalFormat`** object is associated with a pattern when it is created using the new command
  - The object can then be used with the method **`format`** to create strings that satisfy the format
  - An object of the class **`DecimalFormat`** has a number of different methods that can be used to produce numeral strings in various formats

The DecimalFormat Class (Part 1 of 3)

Display 2.5 The DecimalFormat Class

```
1   import java.text.DecimalFormat;

2   public class DecimalFormatDemo
3   {
4       public static void main(String[] args)
5       {
6           DecimalFormat pattern00dot000 = new DecimalFormat("00.000");
7           DecimalFormat pattern0dot00 = new DecimalFormat("0.00");

8           double d = 12.3456789;
9           System.out.println("Pattern 00.000");
10          System.out.println(pattern00dot000.format(d));
11          System.out.println("Pattern 0.00");
12          System.out.println(pattern0dot00.format(d));

13          double money = 19.8;
14          System.out.println("Pattern 0.00");
15          System.out.println("$" + pattern0dot00.format(money));
16
17          DecimalFormat percent = new DecimalFormat("0.00%");
```

(continued)

The `DecimalFormat` Class (Part 2 of 3)

Display 2.5 The DecimalFormat Class

```
18          System.out.println("Pattern 0.00%");
19          System.out.println(percent.format(0.308));

20          DecimalFormat eNotation1 =
21             new DecimalFormat("#0.###E0");//1 or 2 digits before point
22          DecimalFormat eNotation2 =
23             new DecimalFormat("00.###E0");//2 digits before point

24          System.out.println("Pattern #0.###E0");
25          System.out.println(eNotation1.format(123.456));
26          System.out.println("Pattern 00.###E0");
27          System.out.println(eNotation2.format(123.456));

28          double smallNumber = 0.0000123456;
29          System.out.println("Pattern #0.###E0");
30          System.out.println(eNotation1.format(smallNumber));
31          System.out.println("Pattern 00.###E0");
32          System.out.println(eNotation2.format(smallNumber));
33      }
34  }
```

(continued)

The `DecimalFormat` Class (Part 3 of 3)

Display 2.5 **The `DecimalFormat` Class**

SAMPLE DIALOGUE

```
Pattern 00.000
12.346
Pattern 0.00
12.35
Pattern 0.00
$19.80
Pattern 0.00%
30.80%
Pattern #0.###E0
1.2346E2
Pattern 00.###E0
12.346E1
Pattern #0.###E0
12.346E-6
Pattern 00.###E0
12.346E-6
```