

Lecture 1

Review of COMP 110

Arrays

- Array is a data structure that represents a collection of the same types of data.
- Passing arrays to methods
- Returning an array from a method
- Searching arrays

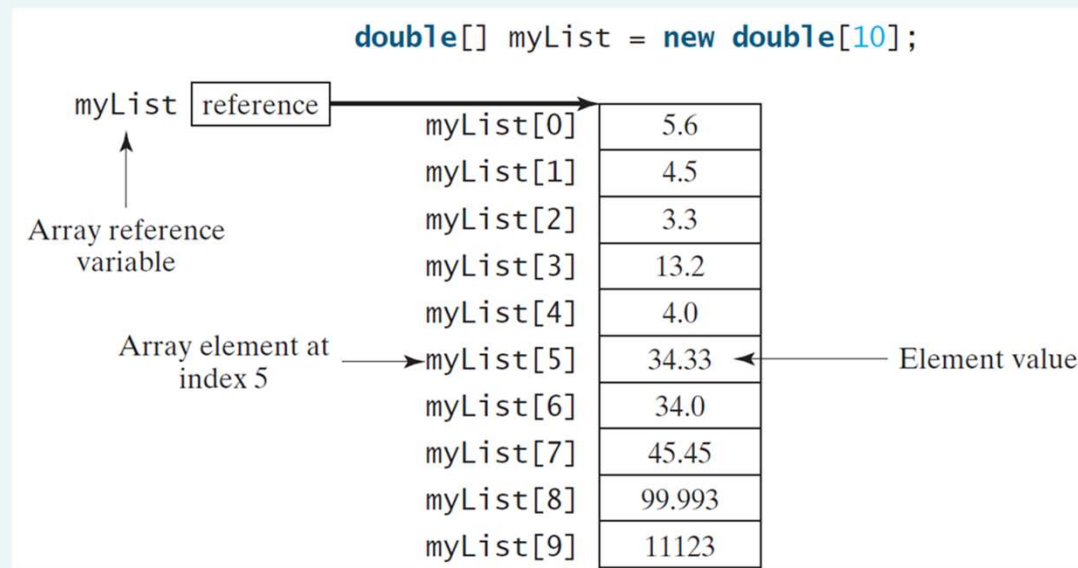
Single-Dimensional Arrays

Objectives

- To describe why arrays are necessary in programming
- To declare array reference variables and create arrays
- To obtain array size using **arrayRefVar.length** and know default values in an array
- To access array elements using indexes
- To declare, create, and initialize an array using an array initializer
- To program common array operations (displaying arrays, summing all elements, finding the minimum and maximum elements, random shuffling, and shifting elements)
- To simplify programming using the foreach loops

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.



Array Initializers

- Declaring, creating, initializing in one step:

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

This shorthand syntax must be in one statement.

Processing Arrays

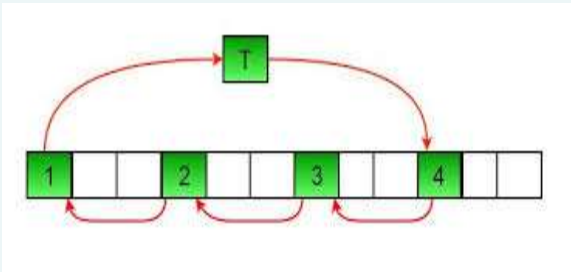
See the examples in the text.

1. (Initializing arrays with input values)
2. (Initializing arrays with random values)
3. (Printing arrays)
4. (Summing all elements)
5. (Finding the largest element)
6. (Finding the smallest index of the largest element)
7. (**Random shuffling**)
8. (**Shifting elements**) - left

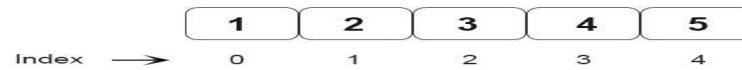
Shifting Elements – shift left



Shifting Elements – shift right



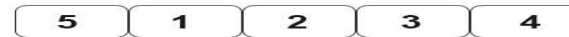
Array



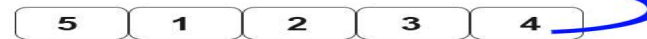
Rotate right by 2



After 1st rotate the new array



Perform 2nd rotate



After 2nd rotate the new array



Enhanced for Loop (for-each loop)

JDK 1.5 introduced a new for loop that enables you to traverse the complete array sequentially without using an index variable. For example, the following code displays all elements in the array `myList`:

```
for (double value: myList)  
    System.out.println(value) ;
```

In general, the syntax is

```
for (elementType value: arrayRefVar) {  
    // Process the value  
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

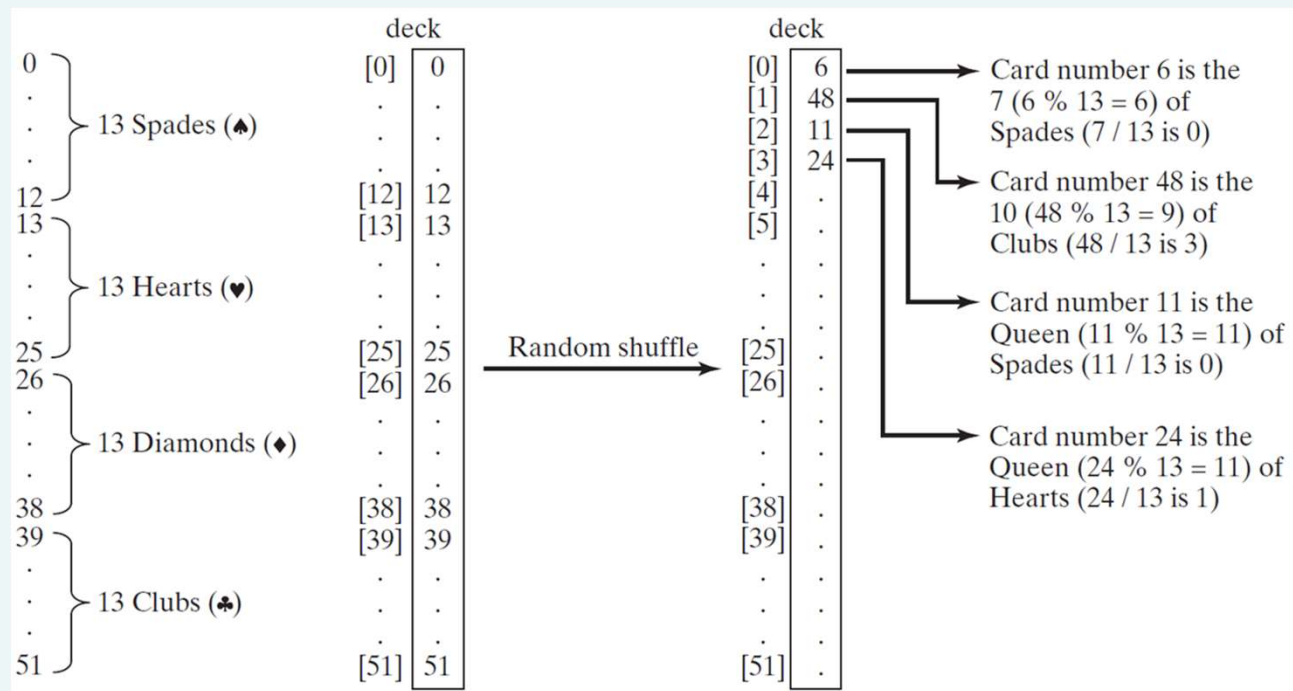
Problem: Deck of Cards (1 of 4)

The problem is to write a program that picks four cards randomly from a deck of 52 cards. All the cards can be represented using an array named `deck`, filled with initial values 0 to 51, as follows:

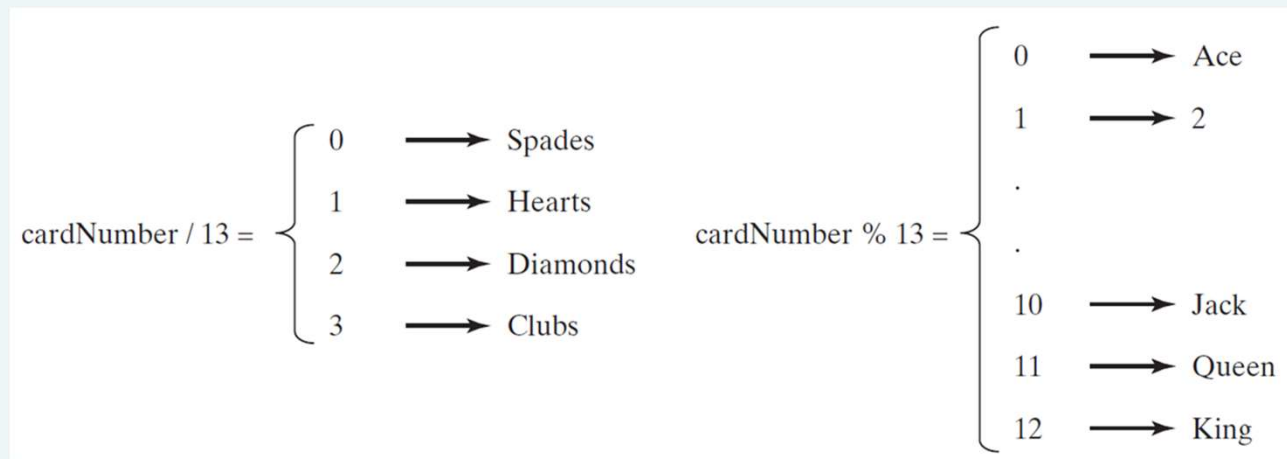
```
int[] deck = new int[52];  
// Initialize cards  
for (int i = 0; i < deck.length; i++)  
    deck[i] = i;
```

[DeckOfCards](#)

Problem: Deck of Cards (2 of 4)



Problem: Deck of Cards (3 of 4)

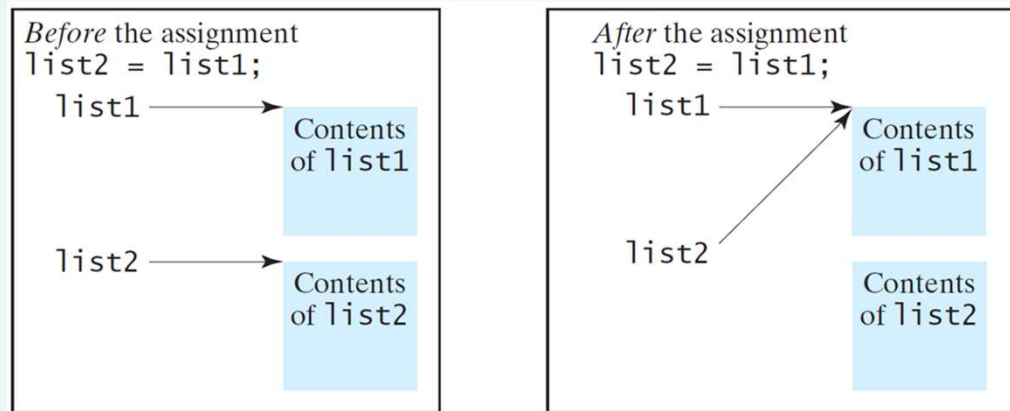


[DeckOfCards](#)

Copying Arrays (1 of 2)

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

```
list2 = list1;
```



Copying Arrays (2 of 2)

Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};  
int[] targetArray = new  
    int[sourceArray.length];  
  
for (int i = 0; i < sourceArray.length;  
    i++)  
    targetArray[i] = sourceArray[i];
```

The arraycopy Utility

```
arraycopy(sourceArray, src_pos,  
          targetArray, tar_pos, length);
```

Example:

```
System.arraycopy(sourceArray, 0,  
                 targetArray, 0, sourceArray.length);
```


Passing Arrays to Methods

```
public static void printArray(int[] array) {  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
}
```

Invoke the method

```
int[] list = {3, 1, 2, 6, 4, 2};  
printArray(list);
```

Invoke the method

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous array

The diagram consists of red lines. A horizontal line is drawn under the anonymous array `{3, 1, 2, 6, 4, 2}` in the `printArray(new int[]{3, 1, 2, 6, 4, 2});` call. A vertical line descends from the center of this horizontal line. From the top of this vertical line, two diagonal lines branch out: one goes up and to the left, pointing to the `list` parameter in the `printArray(list);` call, and the other goes up and to the right, pointing to the `array` parameter in the `printArray(int[] array)` method signature.

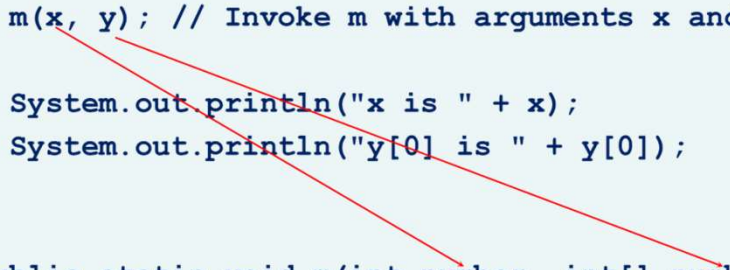
Pass by Value

Java uses **pass by value** to pass arguments to a method. There are important differences between passing a value of variables of primitive data types and passing arrays.

- For a parameter of a primitive type value, the actual value is passed. Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.
- For a parameter of an array type, the value of the parameter contains a reference to an array; this reference is passed to the method. Any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

Simple Example

```
public class Test {  
    public static void main(String[] args) {  
        int x = 1; // x represents an int value  
        int[] y = new int[10]; // y represents an array of int values  
  
        m(x, y); // Invoke m with arguments x and y  
  
        System.out.println("x is " + x);  
        System.out.println("y[0] is " + y[0]);  
    }  
  
    public static void m(int number, int[] numbers) {  
        number = 1001; // Assign a new value to number  
        numbers[0] = 5555; // Assign a new value to numbers[0]  
    }  
}
```

A diagram consisting of two red arrows. The first arrow originates from the variable 'x' in the method call 'm(x, y)' within the 'main' method and points to the parameter 'number' in the method definition 'm(int number, int[] numbers)'. The second arrow originates from the array 'y' in the same method call and points to the parameter 'numbers' in the method definition.

Multidimensional Arrays

Motivations (1 of 2)

Thus far, you have used one-dimensional arrays to model linear collections of elements. You can use a two-dimensional array to represent a matrix or a table. For example, the following table that describes the distances between the cities can be represented using a two-dimensional array.

Distance Table (in miles)

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

Motivations (2 of 2)

```
double[][] distances = {  
    {0, 983, 787, 714, 1375, 967, 1087},  
    {983, 0, 214, 1102, 1763, 1723, 1842},  
    {787, 214, 0, 888, 1549, 1548, 1627},  
    {714, 1102, 888, 0, 661, 781, 810},  
    {1375, 1763, 1549, 661, 0, 1426, 1187},  
    {967, 1723, 1548, 781, 1426, 0, 239},  
    {1087, 1842, 1627, 810, 1187, 239, 0},  
};
```

Objectives

- To give examples of representing data using two-dimensional arrays.
- To declare variables for two-dimensional arrays, create arrays, and access array elements in a two-dimensional array using row and column indexes.
- To program common operations for two-dimensional arrays (displaying arrays, summing all elements, finding the minimum and maximum elements, and random shuffling).
- To pass two-dimensional arrays to methods.

Declare/Create Two-dimensional Arrays

```
// Declare array ref var
dataType[][] refVar;
// Create array and assign its reference to
variable
refVar = new dataType[10][10];
// Combine declaration and creation in one
statement
dataType[][] refVar = new dataType[10][10];
// Alternative syntax
dataType refVar[][] = new dataType[10][10];
```


Declaring Variables of Two-dimensional Arrays and Creating Two-dimensional Arrays

```
int[][] matrix = new int[10][10];
```

or

```
int matrix[][] = new int[10][10];
```

```
matrix[0][0] = 3;
```

```
for (int i = 0; i < matrix.length; i++)
```

```
    for (int j = 0; j < matrix[i].length; j++)
```

```
        matrix[i][j] = (int) (Math.random() *  
        1000);
```

```
double[][] x;
```

Two-dimensional Array Illustration

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

`matrix = new int[5][5];`

(a)

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

`matrix[2][1] = 7;`

(b)

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

```
int[][] array = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9},  
    {10, 11, 12}  
};
```

(c)

`matrix.length?` 5
`matrix[0].length?` 5

`array.length?` 4
`array[0].length?` 3

Declaring, Creating, and Initializing Using Shorthand Notations

You can also use an array initializer to declare, create and initialize a two-dimensional array. For example,

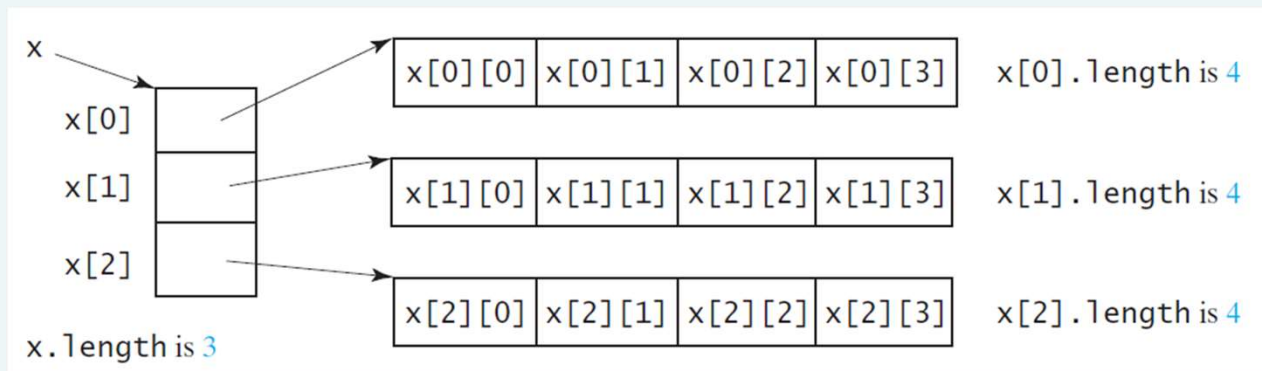
```
int[][] array = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9},  
    {10, 11, 12}  
};
```

Same as

```
int[][] array = new int[4][3];  
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;  
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;  
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;  
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

Lengths of Two-dimensional Arrays (1 of 2)

```
int[][] x = new int[3][4];
```



Lengths of Two-dimensional Arrays (2 of 2)

```
int[][] array = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9},  
    {10, 11, 12}  
};
```

```
array.length  
array[0].length  
array[1].length  
array[2].length  
array[3].length
```

```
array[4].length    ArrayIndexOutOfBoundsException
```

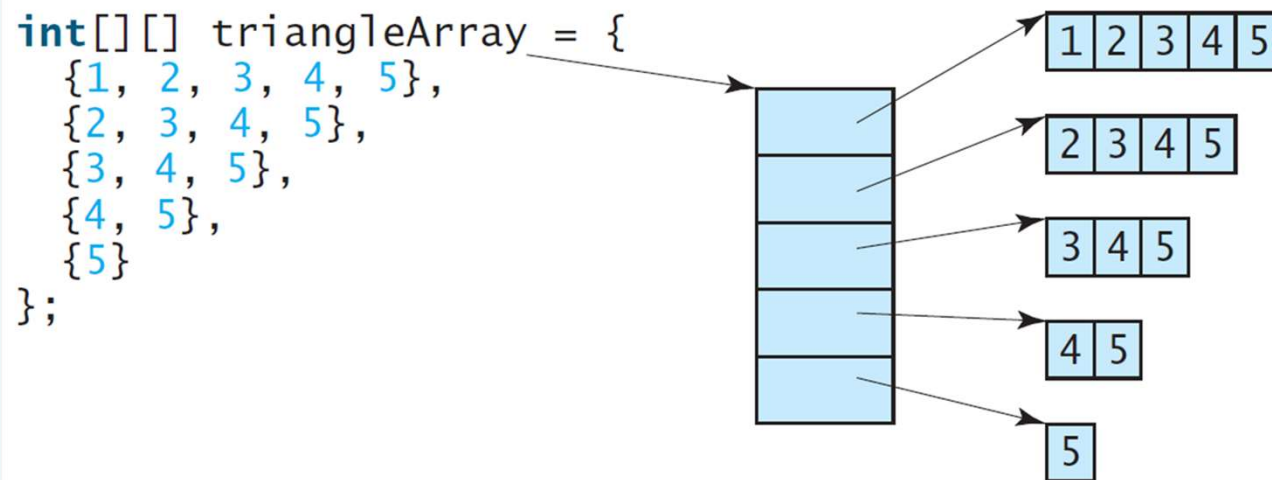
Ragged Arrays (1 of 2)

Each row in a two-dimensional array is itself an array. So, the rows can have different lengths. Such an array is known as a **ragged array**. For example,

```
int[][] matrix = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```

```
matrix.length is 5  
matrix[0].length is 5  
matrix[1].length is 4  
matrix[2].length is 3  
matrix[3].length is 2  
matrix[4].length is 1
```

Ragged Arrays (2 of 2)



Processing Two-Dimensional Arrays

See the examples in the text.

1. (Initializing arrays with input values)
2. (Printing arrays)
3. (Summing all elements)
4. (Summing all elements by column)
5. (Summing all elements by row)
6. (**Random shuffling**)
7. (Which row has the largest sum)
8. (Finding the smallest index of the largest element)

Passing Two-Dimensional Arrays to Methods

PassTwoDimensionalArray

Exception-Handling Overview

Show runtime error

Quotient

Fix it using an if statement

QuotientWithIf

With a method

QuotientWithMethod

Exception Advantages

QuotientWithException

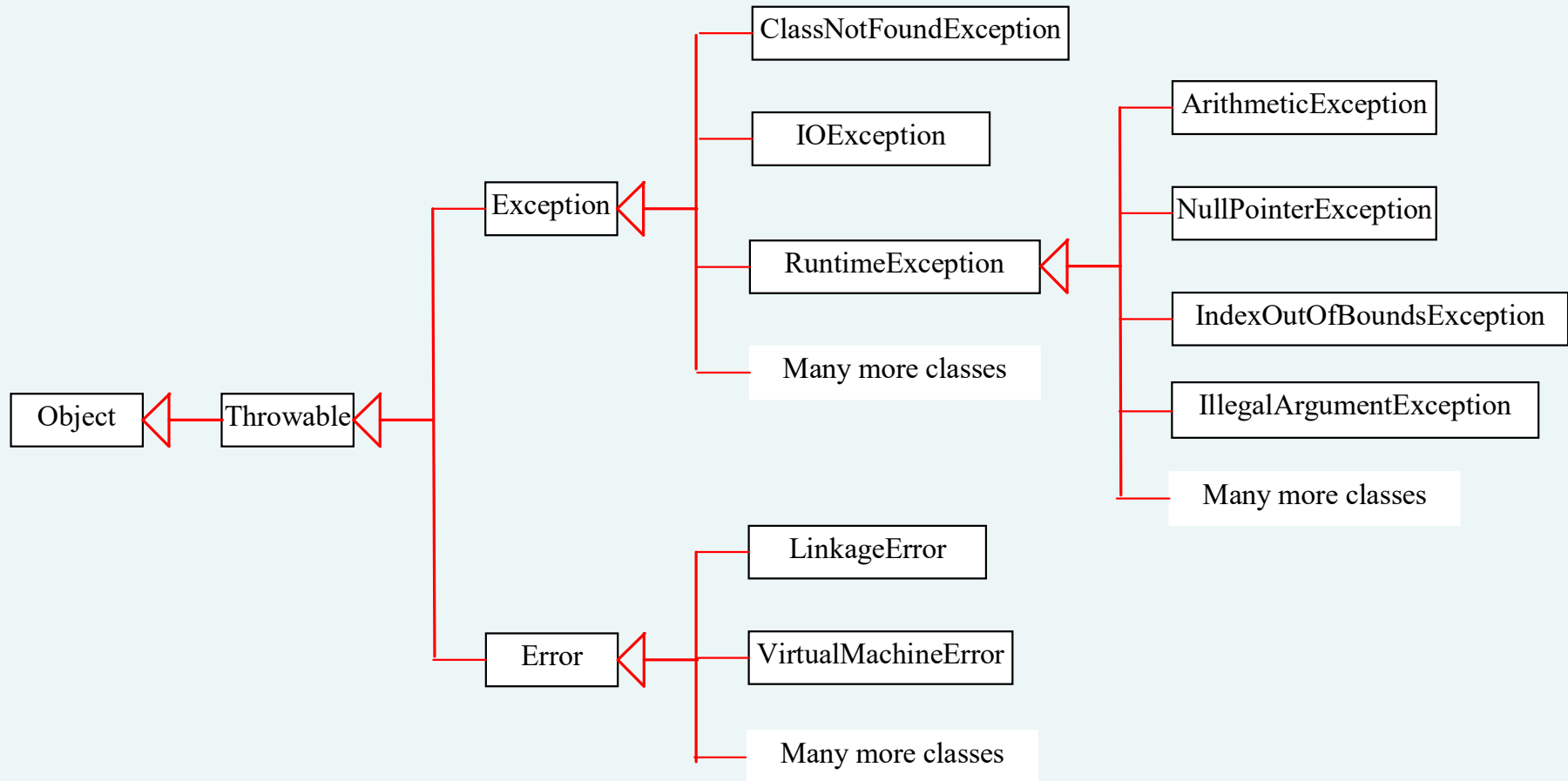
Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.

Handling InputMismatchException

InputMismatchExceptionDemo

By handling InputMismatchException, your program will continuously read an input until it is correct.

Exception Types



Input

- the Scanner class

```
int nextValue;  
int sum=0;  
Scanner kbInput = new Scanner(System.in);  
nextValue = kbInput.nextInt();  
while (nextValue > 0) {  
    sum += nextValue;  
    nextValue = kbInput.nextInt();  
} // end while  
kbInput.close();
```

Input

- The Scanner class (continued)
 - More useful next methods
 - `String next();`
 - **boolean** `nextBoolean();`
 - **double** `nextDouble();`
 - **float** `nextFloat();`
 - **int** `nextInt();`
 - `String nextLine();`
 - **long** `nextLong();`
 - **short** `nextShort();`

Output

- Methods `print` and `println`
 - Write character strings, primitive types, and objects to `System.out`
 - `println` terminates a line of output so next one starts on the next line
 - When an object is used with these methods
 - Return value of object's `toString` method is displayed
 - You usually override this method with your own implementation
 - Problem
 - Lack of formatting abilities

Formatting Output

Use the printf statement.

```
System.out.printf(format, items);
```

Where format is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each specifier begins with a percent sign.

% [flags] [width] [.precision] conversion-character (square brackets denote optional parameters)

Frequently-Used Specifiers

Specifier	Output	Example
%b	a boolean value true or false	
%c	a character	'a'
%d	a decimal integer	200
%f	a floating-point number	45.460000
%e	a number in standard scientific notation	4.556000e+01
%s	a string	

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

display count is 5 and amount is 45.560000

Frequently-Used Flags

- : left-justify (default is to right-justify)
- + : output a plus (+) or minus (-) sign for a numerical value
- 0 : forces numerical values to be zero-padded (default is blank padding)
- , : comma grouping separator (for numbers > 1000)
- : space will display a minus sign if the number is negative or a space if it is positive

- **Width:** Specifies the field width for outputting the argument and represents the minimum number of characters to be written to the output. Include space for expected commas and a decimal point in the determination of the width for numerical values.
- **Precision:** Used to restrict the output depending on the conversion. It specifies the number of digits of precision when outputting floating-point values or the length of a substring to extract from a String. Numbers are rounded to the specified precision.

FormatDemo

The example gives a program that uses **printf** to display a table.

FormatDemo

The File Class

The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The File class is a wrapper class for the file name and its directory path.

Obtaining file properties and manipulating file

java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of <i>hidden</i> is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as <code>getAbsolutePath()</code> except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\\book\\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\\book.
+lastModified(): long	Returns the time that the file was last modified.
+length(): long	Returns the size of the file, or 0 if it does not exist or if it is a directory.
+listFile(): File[]	Returns the files under the directory for a directory File object.
+delete(): boolean	Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames the file or directory represented by this File object to the specified name represented in dest. The method returns true if the operation succeeds.
+mkdir(): boolean	Creates a directory represented in this File object. Returns true if the the directory is created successfully.
+mkdirs(): boolean	Same as mkdir() except that it creates directory along with its parent directories if the parent directories do not exist.

Text I/O

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. This section introduces how to read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

Writing Data Using PrintWriter

java.io.PrintWriter

+PrintWriter(filename: String)

Creates a PrintWriter for the specified file.

+print(s: String): void

Writes a string.

+print(c: char): void

Writes a character.

+print(cArray: char[]): void

Writes an array of character.

+print(i: int): void

Writes an int value.

+print(l: long): void

Writes a long value.

+print(f: float): void

Writes a float value.

+print(d: double): void

Writes a double value.

+print(b: boolean): void

Writes a boolean value.

Also contains the overloaded
println methods.

A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix. The printf method was introduced in §4.6, “Formatting Console Output and Strings.”

Also contains the overloaded
printf methods.

WriteData

Try-with-resources

Programmers often forget to close the file. JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {  
    Use the resource to process the file;  
}
```

[WriteDataWithAutoClose](#)

Reading Data Using Scanner

java.util.Scanner

+Scanner(source: File)

Creates a Scanner object to read data from the specified file.

+Scanner(source: String)

Creates a Scanner object to read data from the specified string.

+close()

Closes this scanner.

+hasNext(): boolean

Returns true if this scanner has another token in its input.

+next(): String

Returns next token as a string.

+nextByte(): byte

Returns next token as a byte.

+nextShort(): short

Returns next token as a short.

+nextInt(): int

Returns next token as an int.

+nextLong(): long

Returns next token as a long.

+nextFloat(): float

Returns next token as a float.

+nextDouble(): double

Returns next token as a double.

+useDelimiter(pattern: String):
Scanner

Sets this scanner's delimiting pattern.

[ReadData](#)

Problem: Replacing Text

Write a class named ReplaceText that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

ReplaceText