

Object-Oriented Programming

Arrays, Exceptions, File Processing



1. Array

- 1.1 Introduction
- 1.2 Array in C
- 1.3 Array in Java
- 1.4 Array as a Parameter
- 1.5 Detour: String[] in main() method
- 1.6 Returning an Array
- 1.7 Common Mistakes
- 1.8 2D Array
- 1.9 Drawback

2. Exceptions

- 2.1 Motivation
- 2.2 Exception Indication
- 2.3 Exception Handling
- 2.4 Execution Flow
- 2.5 Checked vs Unchecked Exceptions
- 2.6 Defining New Exception Classes

1. Array

- 1.1 Introduction
- 1.2 Array in C
- 1.3 Array in Java
- 1.4 Array as a Parameter
- 1.5 Detour: String[] in main() method
- 1.6 Returning an Array
- 1.7 Common Mistakes
- 1.8 2D Array
- 1.9 Drawback

2. Exceptions

- 2.1 Motivation
- 2.2 Exception Indication
- 2.3 Exception Handling
- 2.4 Execution Flow
- 2.5 Checked vs Unchecked Exceptions
- 2.6 Defining New Exception Classes

Using arrays to organise data

- **Array** is the simplest way to store a collection of data of the same type (homogeneous)
- It stores its elements in contiguous memory
 - Array index begins from zero
 - Example of a 5-element integer array *A* with elements filled

A

24	7	-3	15	9
<i>A</i> [0]	<i>A</i> [1]	<i>A</i> [2]	<i>A</i> [3]	<i>A</i> [4]

sum_array.c

```
#include <stdio.h>
#define MAX 6

// To read values into arr and return
// the number of elements read.
int scan_array(double[], int);
void print_array(double[], int);
double sum_array(double[], int);

int main(void)
{
    double list[MAX];
    int size = scan_array(list, MAX);
    print_array(list, size);
    printf("Sum = %f\n",
        sum_array(list, size));

    return 0;
}
```

sum_array.c

```
int scan_array(double arr[], int max_size)
{
    printf("How many elements? ");
    int size;
    scanf("%d", &size);
    if (size > max_size) {
        printf("Exceeded max; you may only enter");
        printf(" %d values.\n", max_size);
        size = max_size;
    }
    printf("Enter %d values: ", size);

    int i;
    for (i = 0; i < size; i++) {
        scanf("%lf", &arr[i]);
    }

    return size;
}
```

sum_array.c

```
// To print values of arr
void print_array(double arr[], int size)
{
    int i;
    for (i = 0; i < size; i++) {
        printf("%f ", arr[i]);
    }

    printf("\n");
}
```

sum_array.c

```
// To compute sum of all elements in arr
double sum_array(double arr[], int size)
{
    double sum = 0.0;
    int i;
    for (i = 0; i < size; i++) {
        sum += arr[i];
    }

    return sum;
}
```

TestArray1.java

```
public class TestArray1 {  
    public static void main(String[] args) {  
        int[] arr; // arr is a reference  
  
        // create a new integer array with 3 elements  
        // arr now refers (points) to this new array  
        arr = new int[3];  
  
        // using the length attribute  
        System.out.println("Length = " + arr.length);  
  
        arr[0] = 100;  
        arr[1] = arr[0] - 37;  
        arr[2] = arr[1] / 2;  
        for (int i = 0; i < arr.length; i++) {  
            System.out.println("arr[" + i + "] = " + arr[i]);  
        }  
    }  
}
```

Declaring an array:
datatype[] arrayName

Constructing an array:
arrayName = **new datatype[size]**

Accessing individual
array elements.

Length = ?
arr[0] = ?
arr[1] = ?
arr[2] = ?

- In Java, **array is an object**.
- Every array has a **public length** attribute (it is not a method!)

TestArray2.java

```
public class TestArray2 {  
    public static void main(String[] args) {  
        // Construct and initialise array  
        double[] arr = { 35.1, 21, 57.7, 18.3 };  
  
        // using the length attribute  
        System.out.println("Length = " + arr.length);  
  
        for (int i = 0; i < arr.length; i++) {  
            System.out.print(arr[i] + " ");  
        }  
        System.out.println();  
  
        // Alternative way  
        for (double element : arr) {  
            System.out.print(element + " ");  
        }  
        System.out.println();  
        System.out.println(Arrays.toString(arr));  
    }  
}
```

Length = 4
35.1 21.0 57.7 18.3
35.1 21.0 57.7 18.3
[35.1, 21.0, 57.7, 18.3]

Syntax (enhanced for-loop): **for** (datatype e: array_name)

Go through all elements in the array. "e" automatically refers to the array element sequentially in each iteration

Using `toString()`
method in `Arrays` class

- Alternative loop syntax for accessing array elements
- Illustrate `toString()` method in `Arrays` class to print an array

TestArray3.java

```
public class TestArray3 {  
    public static void main(String[] args) {  
        int[] list = { 22, 55, 33 };  
  
        swap(list, 0, 2);  
  
        for (int element: list)  
            System.out.print(element + " ");  
        System.out.println();  
    }  
  
    // To swap arr[i] with arr[j]  
    public static void swap(int[] arr, int i, int j) {  
        int temp = arr[i];  
        arr[i] = arr[j];  
        arr[j] = temp;  
    }  
}
```

- The reference to the array is passed into a method
 - Any modification of the elements in the method will affect the actual array

TestCommandLineArgs.java

```
public class TestCommandLineArgs {  
    public static void main(String[] args) {  
  
        for (int i = 0; i < args.length; i++)  
            System.out.println("args[" + i + "] = " + args[i]);  
    }  
}
```

- The main() method contains a parameter which is an array of `String` objects
- We can use this for command-line arguments

```
java TestCommandLineArgs The "Harry Potter" series has 7 books.  
args[0] = The  
args[1] = Harry Potter  
args[2] = series  
args[3] = has  
args[4] = 7  
args[5] = books.
```

TestArray4.java

```
public class TestArray4 {  
    public static void main(String[] args) {  
        double[] values = makeArray(5, 999.0);  
  
        for (double value: values) {  
            System.out.println(value + " ");  
        }  
    }  
  
    // To create an array and return it to caller  
    public static double[] makeArray(int size, double limit) {  
        double[] arr = new double[size];  
  
        for (int i = 0; i < arr.length; i++) {  
            arr[i] = limit/(i+1);  
        }  
        return arr;  
    }  
}
```

Return type:
datatype[]

999.0
499.5
333.0
249.75
199.8

- Array can be returned from a method

- `length` versus `length()`
 - To obtain length of a `String` object `str`, we use the `length()` method
 - Example: `str.length()`
 - To obtain length (size) of an array `arr`, we use the `length` attribute
 - Example: `arr.length`
- Array index out of range
 - Beware of `ArrayIndexOutOfBoundsException`

```
public static void main(String[] args) {  
    int[] numbers = new int[10];  
    . . .  
    for (int i = 1; i <= numbers.length; i++)  
        System.out.println(numbers[i]);  
}
```

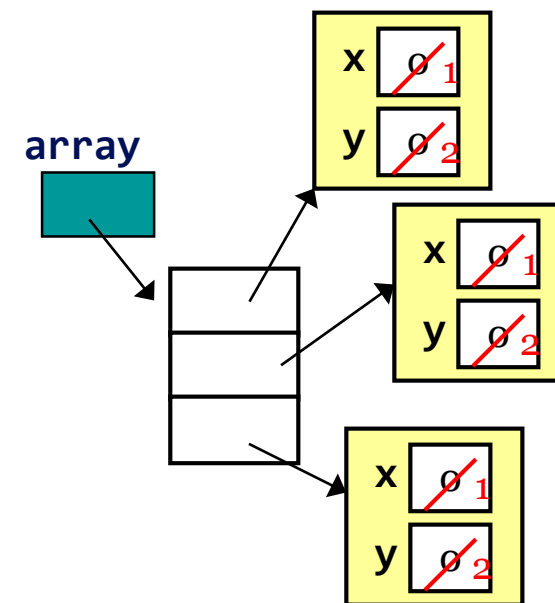
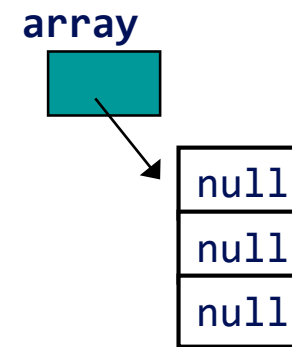
- When you have an **array of objects**, it's very common to forget to instantiate the array's objects.
- Programmers often instantiate the array itself and then think they're done – that leads to `java.lang.NullPointerException`
- Example on next slide
 - It uses the **Point** class in the API
 - Refer to the API documentation for details

```
Point[] array = new Point[3];  
for (int i = 0; i < array.length; i++) {  
    array[i].setLocation(1, 2);  
}
```

There are no objects referred to by `array[0]`, `array[1]`, and `array[2]`, so how to call `setLocation()` on them?!

Corrected code:

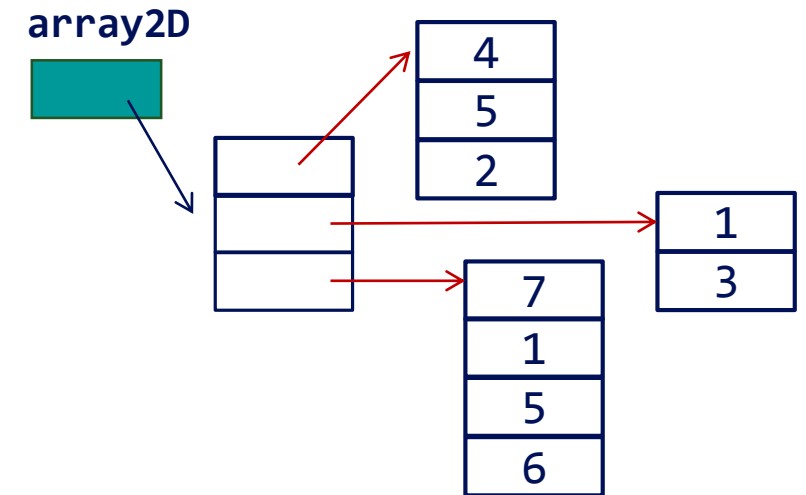
```
Point[] array = new Point[3];  
for (int i = 0; i < array.length; i++) {  
    array[i] = new Point();  
    array[i].setLocation(1, 2);  
}
```



- A two-dimensional (2D) array is an array of array.
- This allows for rows of **different lengths**.

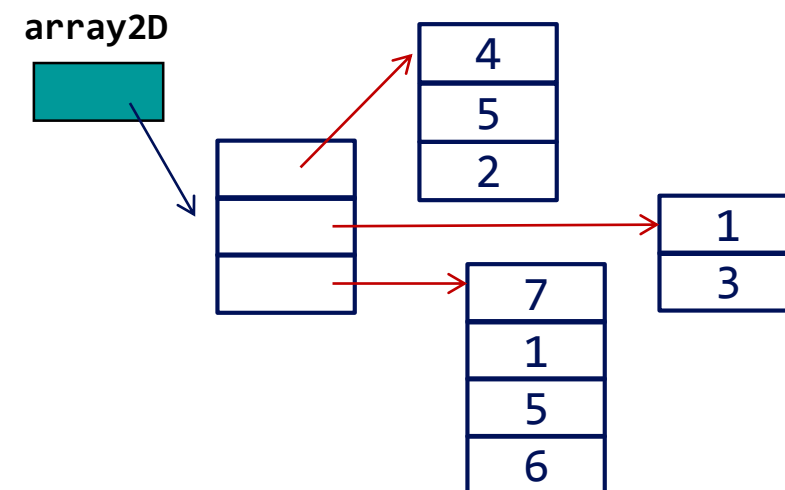
```
// an array of 12 arrays of int  
int[][] products = new int[12][];
```

```
int[][] array2D = {{4,5,2}, {1,3}, {7,1,5,6}};
```



Test2DArray.java

```
public class Test2DArray {  
    public static void main(String[] args) {  
        int[][] array2D = {{4, 5, 2}, {1, 3}, {7, 1, 5, 6}};  
  
        System.out.println("array2D.length = " + array2D.length);  
        for (int i = 0; i < array2D.length; i++) {  
            System.out.println("array2D[" + i + "].length = "  
                               + array2D[i].length);  
        }  
  
        for (int row = 0; row < array2D.length; row++) {  
            for (int col = 0; col < array2D[row].length; col++) {  
                System.out.print(array2D[row][col] + " ");  
                System.out.println();  
            }  
        }  
    }  
}
```



```
array2D.length = 3  
array2D[0].length = ?  
array2D[1].length = ?  
array2D[2].length = ?  
?  
?  
?
```

- Array has one major drawback:
 - Once initialized, the array size is **fixed**
 - Reconstruction is required if the array size changes
 - To overcome such limitation, we can use some classes related to array
- Java has an **Array** class
 - Check API documentation and explore it yourself
- However, we will not be using this **Array** class much; we will be using some other classes such as **Vector** or **ArrayList** (to be covered later)
- Before doing Vector/ArrayList, we will introduce another concept later called **Generics**

1. Array

- 1.1 Introduction
- 1.2 Array in C
- 1.3 Array in Java
- 1.4 Array as a Parameter
- 1.5 Detour: String[] in main() method
- 1.6 Returning an Array
- 1.7 Common Mistakes
- 1.8 2D Array
- 1.9 Drawback

2. Exceptions

- 2.1 Motivation
- 2.2 Exception Indication
- 2.3 Exception Handling
- 2.4 Execution Flow
- 2.5 Checked vs Unchecked Exceptions
- 2.6 Defining New Exception Classes

Exceptions

Handling exceptional events

```
List<Integer> list = new ArrayList<Integer>();

public int getItem(int i) {
    if (list.size() <= i) {
        return -1;
    }
    return list.get(i);
}
```

- When using return codes, developers **must remember value/meaning of return codes.**

```
List<Integer> list = new ArrayList<Integer>();

public int getItem(int i) {
    if (list.size() <= i) {
        System.exit();
    }
    return list.get(i);
}
```

- Using methods that terminate the whole execution lead to code that is **not reusable! Never do this!**

- Terminating execution inside a method must be avoided. The resulting code is not reusable
- The use of return codes implies:
 - Code is **messy to write** and **hard to read**
 - Only the **direct caller** can intercept errors (**no delegation** to any upward method)

```
retval = function();  
if (retval == ERROR_CODE_1) {  
    // handle error 1  
} else if (retval == ERROR_CODE_2) {  
    // handle error 2  
}  
...  

```

1. Open a file
2. Determine file size
3. Allocate the needed memory
4. Read the file into memory
5. Close the file

All these operations can fail! Errors must be checked!

```
void loadFile() {  
    open file;  
    determine file size;  
    allocate memory;  
    read file into memory;  
    close file;  
}
```

- Short, readable BUT
- Not reusable nor dependable
- Errors are not checked at all


```
open file;
if(operationFailed)
    return -1;
determine file size;
if(operationFailed)
    return -2;
allocate memory;
if(operationFailed) {
    close the file;
    return -3;
}
read file into memory;
if (operationFailed) {
    close the file;
    return -4;
}
close file;
if (operationFailed)
    return -5;



return 0;
```

- Reusable, dependable BUT
- Long and obscure
- Error-handling code mixed with functional code
 - To detect errors we have to remember the specification of library calls
 - Each library has its own standards

```
try {  
    open file;  
    determine file size;  
    allocate memory;  
    read file into memory;  
    close file;  
} catch (fileOpenFailed) {  
    doSomething;  
}  
catch(determineSizeFailed) {  
    doSomething;  
}  
catch (memoryAllocationFailed) {  
    doSomething;  
}  
catch (readFailed) {  
    doSomething;  
}  
catch (fileCloseFailed) {  
    doSomething;  
}
```

- Reusable
- Dependable
- Readable
- Functional code, clearly separated from error handling code
- Error handling code separated among different errors
- Possible to delegate error handling to caller methods

Understand how to use the mechanism of exceptions to handle errors or exceptional events that occur during program execution

- Three types of errors
- **Syntax errors**  *Easiest to detect and correct*
 - Occurs when the rule of the language is violated
 - Detected by compiler
- **Run-time errors**
 - Occurs when the computer detects an operation that cannot be carried out (eg: division by zero; x/y is syntactically correct, but if y is zero at run-time a run-time error will occur)
- **Logic errors**  *Hardest to detect and correct*
 - Occurs when a program does not perform the intended task

Example.java

```
import java.util.Scanner;

public class Example {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an integer: ");

        int num = sc.nextInt();
        System.out.println("num = " + num);
    }
}
```

← If error occurs here

← The rest of the code is skipped and program is terminated.

```
Enter an integer: abc
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:909)
    at java.util.Scanner.next(Scanner.java:1530)
    at java.util.Scanner.nextInt(Scanner.java:2160)
    at java.util.Scanner.nextInt(Scanner.java:2119)
    at Example1.main(Example1.java:8)
```

- Consider the **factorial()** method:

- What if the caller supplies a negative parameter?

```
public static int factorial(int n) {  
    int ans = 1;  
    for (int i = 2; i <= n; i++) {  
        ans *= i;  
    }  
    return ans;  
}
```

What if n is negative?

- Should we terminate the program?

```
public static int factorial(int n) {  
    if (n < 0) {  
        System.out.println("n is negative");  
        System.exit(1);  
    }  
    //Other code not changed  
}
```

- Note that factorial() method can be used by other programs

- Hence, difficult to cater to all possible scenarios

- Instead of deciding how to deal with an error, Java provides the **exception** mechanism:
 1. Indicate an error (**exception event**) has occurred
 2. Let the user decide how to handle the problem in a separate section of code specific for that purpose
 3. Crash the program if the error is not handled
- Exception mechanism consists of two components:
 - **Exception indication**
 - **Exception handling**
- Note that the preceding example of using exception for ($n < 0$) is solely illustrative. Exceptions are more appropriate for harder to check cases such as when the value of n is too big, causing overflow in computation.

- To indicate an error is detected:
 - Also known as **throwing an exception**
 - This allows the user to detect and handle the error

SYNTAX	<code>throw <i>ExceptionObject</i>;</code>
--------	--

- Exception object must be:
 - An object of a class derived from **class Throwable**
 - Contain useful information about the error
- There are a number of useful predefined exception classes:
 - **ArithmeticException**
 - **NullPointerException**
 - **IndexOutOfBoundsException**
 - **IllegalArgumentException**

- The different exception classes are used to **categorize the type of error**:
 - There is no major difference in the available methods

Constructor	
	<i>ExceptionClassName</i>(String msg) Construct an exception object with the error message msg
Common methods for Exception classes	
String	getMessage() Return the message stored in the object
void	printStackTrace() Print the calling stack

ExampleImproved.java

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

public class ExampleImproved {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        boolean isError = false;
        do {
            System.out.print("Enter an integer: ");
            try {
                int num = sc.nextInt();
                System.out.println("num = " + num);
                isError = false;
            } catch (InputMismatchException e) {
                System.out.print("Incorrect input: integer required. ");
                sc.nextLine(); // skip newline
                isError = true;
            }
        } while (isError);
    }
}
```

```
do {  
    System.out.print("Enter an integer: ");  
    try {  
        int num = sc.nextInt();  
        System.out.println("num = " + num);  
        isError = false;  
    } catch (InputMismatchException e) {  
        System.out.print("Incorrect input: integer required. ");  
        sc.nextLine(); // skip newline  
        isError = true;  
    }  
} while (isError);
```

```
Enter an integer: abc  
Incorrect input: integer required. Enter an integer: def  
Incorrect input: integer required. Enter an integer: 1.23  
Incorrect input: integer required. Enter an integer: 92  
num = 92
```

```
public static int factorial(int n)
    throws IllegalArgumentException {
```

This declares that method factorial()
may throw IllegalArgumentException

```
    if (n < 0) {
        IllegalArgumentException exObj
            = new IllegalArgumentException(n + " is invalid!");
        throw exObj;
    }
```

Actual act of throwing an exception (Note: 'throw' and not 'throws'). These 2
statements can be shortened to:

```
    throw new IllegalArgumentException(n + " is invalid!");
```

```
    int ans = 1;
    for (int i = 2; i <= n; i++) {
        ans *= i;
    }
    return ans;
}
```

■ Note:

- A method can throw more than one type of exception

- As the user of a method that can throw exception(s):
 - It is your responsibility to handle the exception(s)
 - Also known as **exception catching**

```
try {  
    statement(s);  
}
```

```
// try block  
// exceptions might be thrown  
// followed by one or more catch block
```

```
catch (ExpClass1 obj1) {  
    statement(s);  
} catch (ExpClass2 obj2) {  
    statement(s);  
}
```

```
// a catch block  
// Do something about the exception  
// catch block for another type of  
// exception
```

```
finally {  
    statement(s);  
}
```

```
// finally block – for cleanup code
```

TestException.java

```
public class TestException {  
  
    public static int factorial(int n) throws IllegalArgumentException {  
        //code not shown  
    }  
  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter n: ");  
        int input = sc.nextInt();  
  
        try {  
            System.out.println("Ans = " + factorial(input));  
        } catch (IllegalArgumentException expObj) {  
            System.out.println(expObj.getMessage());  
        }  
    }  
}
```

We choose to print out the error message in this case. There are other ways to handle this error. See next slide for more complete code.

TestException.java

```
public static int factorial(int n) throws IllegalArgumentException {
    System.out.println("Before Checking");
    if (n < 0) {
        throw new IllegalArgumentException(n + " is invalid!");
    }
    System.out.println("After Checking");
    //... other code not shown
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter n: ");
    int input = sc.nextInt();
    try {
        System.out.println("Before factorial()");
        System.out.println("Ans = " + factorial(input));
        System.out.println("After factorial()");
    } catch (IllegalArgumentException expObj) {
        System.out.println("In Catch Block");
        System.out.println(expObj.getMessage());
    } finally {
        System.out.println("Finally!");
    }
}
```

Enter n: 4
Before factorial()
Before Checking
After Checking
Ans = 24
After factorial()
Finally!

Enter n: -2
Before factorial()
Before Checking
In Catch Block
-2 is invalid!
Finally!

TestExceptionRetry.java

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    int input;  
    boolean retry = true;  
    do {  
        try {  
            System.out.print("Enter n: ");  
            input = sc.nextInt();  
            System.out.println("Ans = " + factorial(input));  
            retry = false; // no need to retry  
        } catch (IllegalArgumentException expObj) {  
            System.out.println(expObj.getMessage());  
        }  
    } while (retry);  
}
```

■ Another version

- Keep retrying if $n < 0$

```
Enter n: -2  
-2 is invalid!  
Enter n: -7  
-7 is invalid!  
Enter n: 6  
Ans = 720
```


- **Checked exceptions** are those that require handling during compile time, or a compilation error will occur.
- **Unchecked exceptions** are those whose handling is not verified during compile time.
 - `RuntimeException`, `Error` and `their subclasses` are unchecked exceptions.
 - In general, unchecked exceptions are due to programming errors that are not recoverable, like accessing a null object (`NullPointerException`), accessing an array element outside the array bound (`IndexOutOfBoundsException`), etc.
 - As unchecked exceptions can occur anywhere, and to avoid overuse of try-catch blocks, Java does not mandate that unchecked exceptions must be handled.

- `InputMismatchException` and `IllegalArgumentException` are subclasses of `RuntimeException`, and hence they are unchecked exceptions. (Ref: `ExampleImproved.java` and `TestException.java`)



- New exception classes can be defined by deriving from class `Exception`:

```
public class MyException extends Exception {  
    public MyException(String s) {  
        super(s);  
    }  
}
```

- The new exception class can then be used in `throw` statements and `catch` blocks:

```
throw new MyException("MyException: Some reasons");
```

```
try {  
    ...  
} catch (MyException e) {  
    ...  
}
```

NotEnoughFundException.java

```
public class NotEnoughFundException extends Exception {  
  
    private double amount;  
  
    public NotEnoughFundException(String s, double amount) {  
        super(s);  
        this.amount = amount;  
    }  
  
    public double getAmount() {  
        return amount;  
    }  
}
```

BankAccount.java

```
class BankAccount {  
    private int accountNumber;  
    private double balance;  
  
    public BankAccount() {  
        // By default, numeric attributes are initialised to 0  
    }  
  
    public BankAccount(int number, double aBalance) {  
        accountNumber = number;  
        balance = aBalance;  
    }  
  
    public int getAccountNumber() {  
        return accountNumber;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
}
```

BankAccount.java

```
public void deposit(double amount) {
    balance += amount;
}

public void withdraw(double amount) throws NotEnoughFundException {
    if (balance >= amount) {
        balance -= amount;
    } else {
        double needs = amount - balance;
        throw new NotEnoughFundException("Withdrawal Unsuccessful", needs);
    }
}
} // class BankAccount
```

TestBankAccount.java

```
public class TestBankAccount {  
  
    public static void main(String[] args) {  
  
        BankAccount bankAccount = new BankAccount(1234, 0.0);  
  
        System.out.println("Current balance: $" + bankAccount.getBalance());  
  
        System.out.println("Depositing $200...");  
        bankAccount.deposit(200.0);  
  
        System.out.println("Current balance: $" + bankAccount.getBalance());  
    }  
}
```

```
Current balance: $0.0  
Depositing $200...  
Current balance: $200.0
```

TestBankAccount.java

```
try {
    System.out.println("Withdrawing $150...");
    bankAccount.withdraw(150.0);
    System.out.println("Withdrawing $100...");
    bankAccount.withdraw(100.0);
} catch (NotEnoughFundException e) {
    System.out.println(e.getMessage());
    System.out.println("Your account is short of $" + e.getAmount());
} finally {
    System.out.println("Current balance: $" + bankAccount.getBalance());
}
} // main
} // class TestBankAccount
```

```
Current balance: $0.0
Depositing $200...
Current balance: $200.0
Withdrawing $150...
Withdrawing $100...
Withdrawal Unsuccessful
Your account is short of $50.0
Current balance: $50.0
```


- We learned about **exceptions**, how to raise and handle them
- We learned how to define new exception classes

Input/output on files: reading input from a file and writing data to a file.

1. File Input

1.1 File Objects

1.2 Reading File

1.3 Input Tokens

1.4 Tokenizing a String

1.5 Exercise: Runners

2. File Output

2.1 PrintStream

2.2 System.out and PrintStream

2.3 Exercise: Runners (revisit)

3. Input and Output Streams

3.1 InputStream and OutputStream

3.2 Examples

File Input

- The API `File` class represents files
 - In `java.io` package
 - Creating a `File` object does not actually create that file on your drive
- Some methods in `File` class:

Method	Description
boolean <code>canRead()</code>	Tests whether the application can read the file
boolean <code>canWrite()</code>	Tests whether the application can modify the file
boolean <code>delete()</code>	Deletes the file or directory
boolean <code>exists()</code>	Tests whether the file or directory exists
String <code>getName()</code>	Returns the name of the file or directory
long <code>length()</code>	Returns the length (in bytes) of the file

■ Example:

```
File file = new File("myfile");  
  
if (file.exists() && file.length() > 2048) {  
    file.delete();  
}
```

■ Path

- Absolute path
- Specify a drive or start with the root (/) directory
- Eg: "C:/Documents/CS1020/data"

■ Relative path

- With respect to where the program resides
- Eg: "input/eels3.in"

- Pass a **File** reference when constructing a **Scanner** object

FileExample1.java

```
import java.util.*;
import java.io.*;

public class FileExample1 {
    public static void main(String[] args) throws FileNotFoundException {

        Scanner infile = new Scanner(new File("example"));
        int sum = 0;
        while (infile.hasNextInt()) {
            sum += infile.nextInt();
        }

        System.out.println("Sum = " + sum);
    }
}
```

File "example":
Output:


FileExample2.java

```
import java.util.*;
import java.io.*;

public class FileExample2 {
    public static void main(String[] args) throws FileNotFoundException {
        try {
            Scanner infile = new Scanner(new File("example"));
            int sum = 0;
            while (infile.hasNextInt()) {
                sum += infile.nextInt();
            }
            System.out.println("Sum = " + sum);
        } catch (FileNotFoundException e) {
            System.out.println("File 'example' not found!");
        }
    }
}
```


FileExample3.java

```
import java.util.*;
import java.io.*;

public class FileExample2 {
    public static void main(String[] args) throws FileNotFoundException {
        File file = new File("example");
        if (!file.exists()) {
            System.out.println("File 'example' does not exist!");
            System.exit(1);
        }

        Scanner infine = new Scanner(file);
        int sum = 0;
        while (infine.hasNextInt()) {
            sum += infine.nextInt();
        }
        System.out.println("Sum = " + sum);
    }
}
```

- Input data are broken into tokens when read.
- Scanner view all input as a stream of characters, which it processes with its **input cursor**
- Each call to extract the next input (`next()`, `nextInt()`, `nextDouble()`, etc.) advances the cursor to the end of the current token
- Tokens are separated by whitespace

InputTokens.java

```
import java.util.*;
import java.io.*;

public class InputTokens {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner infine = new Scanner(new File("tokens"));
        int a = infine.nextInt();
        String b = infine.next();
        String c = infine.nextLine();
        double d = infine.nextDouble();

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

File "tokens":

(viewed on screen)

```
123 CS1020 Software Components
456 78.9
```

(internally)

```
123 CS1020 Software Components\n456 78.9\n
```

```
a = 123
b = CS1020
c = Software Components
d = 456.0
```

```
int a = scanner.nextInt();
String b = scanner.next();
String c = scanner.nextLine();
double d = scanner.nextDouble();
```

```
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
```

File “tokens”:

```
123 CS1020 Software Components\n456 78.9\n
```

After `int a = scanner.nextInt();`

```
123 CS1020 Software Components\n456 78.9\n
```

After `String b = scanner.next();`

```
123 CS1020 Software Components\n456 78.9\n
```

After `String c = scanner.nextLine();`

```
123 CS1020 Software Components\n456 78.9\n
```

After `double d = scanner.nextDouble();`

```
123 CS1020 Software Components\n456 78.9\n
```

- A Scanner can tokenize a string

StringTokenize.java

```
import java.util.*;
import java.io.*;

public class StringTokenize {
    public static void main(String[] args) {
        String message = "345 students in CS1020.";
        Scanner scanner = new Scanner(message);

        int a = scanner.nextInt();
        String b = scanner.next();
        String c = scanner.nextLine();

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

- Write a program to read in the distances run by a group of runners
- Sample input file “runners_data”:
 - Runner ID (type `int`), name (String, a single word), followed by a list of distances in km (type `double`)
 - You may assume that there are at least one runner and each runner has at least one distance record

```
123 Charlie 6.5 5.2 7.8 5.8 7.2 6.6 9.2 7.2
987 Alex 12.8
312 Jenny 5.7 4 6.2
509 Margaret 3.1 3.4 3.2 3.1 3.5
610 Richard 11.2 13.2 10.8 9.5 15.8 12.4
```

RunnersFlawed.java

```
import java.util.*;
import java.io.*;

public class RunnersFlawed {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner infine = new Scanner(new File("runners_data"));
        int count = 0;
        double totalDist = 0.0;
        while (infine.hasNext()) {
            infine.nextInt(); // read ID
            infine.next();    // read name
            while (infine.hasNextDouble()) {
                count++;
                totalDist += infine.nextDouble();
            }
        }
        System.out.printf("Total distance = %.2f\n", totalDist);
        System.out.printf("Average distance per run = %.2f\n", totalDist/count);
    }
}
```

```
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at RunnersFlawed.main(RunnersFlawed.java:14)
```

- What went wrong?

RunnersFlawed.java

```
int count = 0;
double totalDist = 0.0;
while (infile.hasNext()) {
    infile.nextInt(); // read ID
    infile.next();    // read name
    while (infile.hasNextDouble()) {
        count++;
        totalDist += infile.nextDouble();
    }
}
```

```
123 Charlie 6.5 5.2 7.8 5.8 7.2 6.6 9.2 7.2
987 Alex 12.8
312 Jenny 5.7 4 6.2
509 Margaret 3.1 3.4 3.2 3.1 3.5
610 Richard 11.2 13.2 10.8 9.5 15.8 12.4
```


- Solution: read line by line, then read tokens from each line.

RunnersCorrected.java

```
// Earlier portion omitted for brevity
Scanner infine = new Scanner(new File("runners_data"));
int count = 0;
double totalDist = 0.0;
while (infine.hasNextLine()) {
    String line = infine.nextLine();
    Scanner scanner = new Scanner(line);
    scanner.nextInt(); // read ID
    scanner.next(); // read name
    while (scanner.hasNextDouble()) {
        count++;
        totalDist += scanner.nextDouble();
    }
}

// Later portion omitted for brevity
```

Total distance = 173.40
Average distance per run = 7.54

File Output

- In `java.io` package
- **PrintStream**: An object that allows you to print output to a file
 - Any methods you have used on `System.out` (such as `println()`) will work on a `PrintStream`

```
PrintStream name = new PrintStream(new File("filename"));
```

- Example:

```
PrintStream ps = new PrintStream(new File("greetings"));  
ps.println("Hello world!");  
ps.println("The quick brown fox jumps over the lazy dog.");
```

```
PrintStream name = new PrintStream(new File("filename"));
```

- If the file does not exist, it is created.
- If the file already exists, it is overwritten.
- Note: Do NOT open the same file for reading (Scanner) and writing (PrintStream) at the same time
 - You will overwrite the input file with an empty file

- System.out is actually a PrintStream
- A reference to it can be stored in a PrintStream variable
 - Printing to that variable causes console output to appear

```
PrintStream out1 = System.out;  
PrintStream out2 = new PrintStream(new File("data.txt"));  
out1.println("Hello, console!"); // goes to console  
out2.println("Hello, file!");    // goes to file
```

- Modify RunnersCorrected.java to send its output to the file “running_stat”.

RunnersOutfile.java

```
import java.util.*;
import java.io.*;

public class RunnersOutfile {

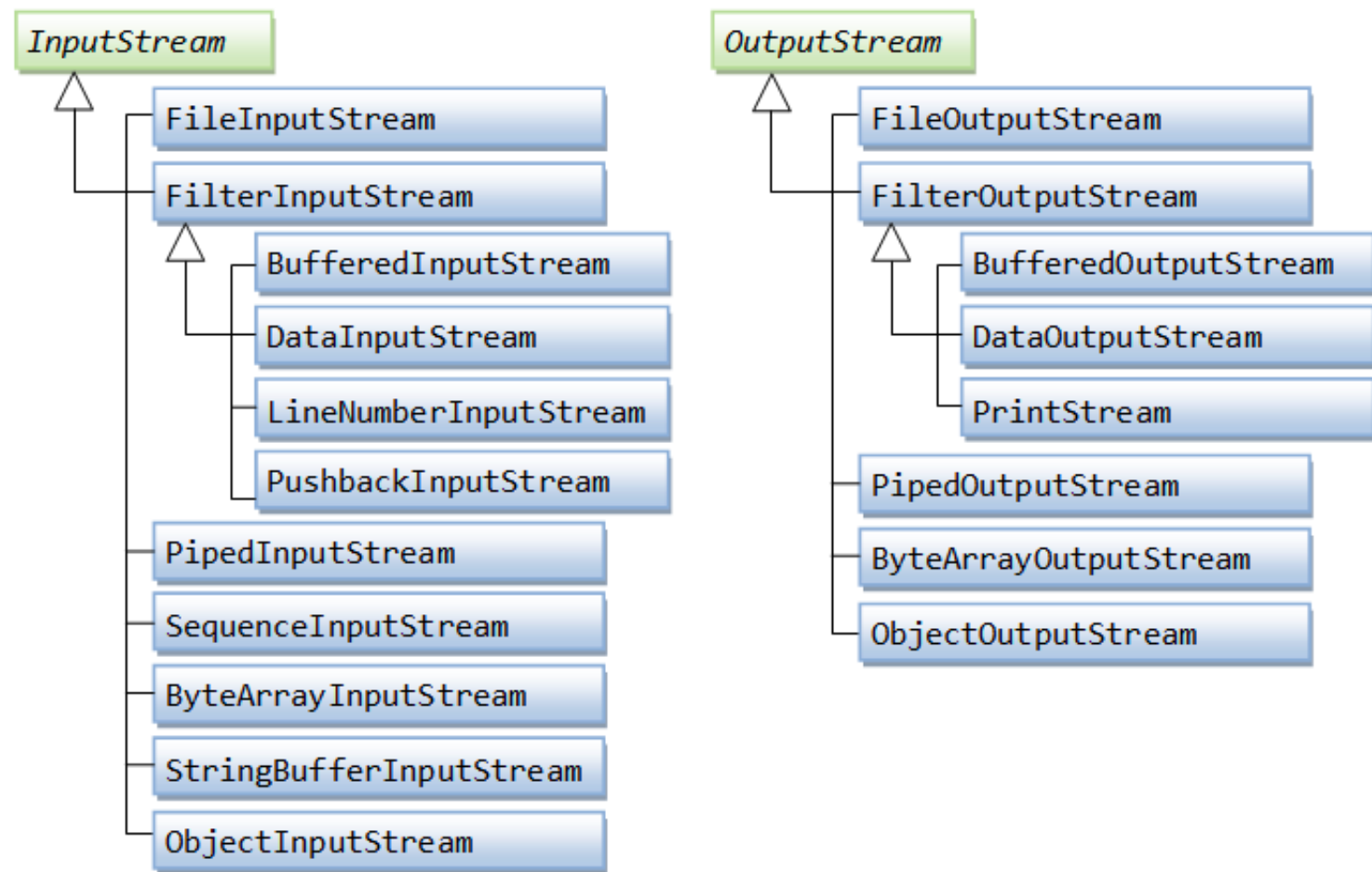
    public static void main(String[] args) throws FileNotFoundException {
        Scanner infine = new Scanner(new File("runners_data"));

        // code omitted for brevity

        PrintStream outfile = new PrintStream(new File("running_stat"));
        outfile.printf("Total distance = %.2f\n", totalDist);
        outfile.printf("Average distance per run = %.2f\n", totalDist/count);
        outfile.close();
    }
}
```

Input and Output Streams

- **InputStream** and **OutputStream** are abstractions of the different ways to input and output data
 - That is, it doesn't matter if the stream is a file, a web page, a video, etc.
 - All that matters is that you receive information from the stream or send information into the stream.
 - **InputStream** is an abstract superclass that provides a minimal programming interface and a partial implementation of input streams. It defines methods for reading bytes, arrays of bytes, etc.
 - **OutputStream** is an abstract superclass that provides a minimal programming interface and a partial implementation of output streams. It defines methods for writing bytes or arrays of bytes to the stream.



- We will use some of the methods in OutputStream below:

Methods	
Modifier and Type	Method and Description
void	<code>close()</code> Closes this output stream and releases any system resources associated with this stream.
void	<code>flush()</code> Flushes this output stream and forces any buffered output bytes to be written out.
void	<code>write(byte[] b)</code> Writes <code>b.length</code> bytes from the specified byte array to this output stream.
void	<code>write(byte[] b, int off, int len)</code> Writes <code>len</code> bytes from the specified byte array starting at offset <code>off</code> to this output stream.
abstract void	<code>write(int b)</code> Writes the specified byte to this output stream.

TestOutputStream.java

```
import java.io.*;

public class TestOutputStream {

    public static void main(String[] args) throws IOException {

        String message = new String("Hello world!");
        OutputStream out = new FileOutputStream("message_file");

        byte[] bytes = message.getBytes();

        out.write(bytes);
        out.write(bytes[1]);
        out.write(10); // ASCII value of newline
        out.write(bytes, 3, 5);
        out.close();
    }
}
```

```
javac TestOutputStream.java
java TestOutputStream
cat msg_file
Hello world!e
lo wo
```

read

```
public abstract int read()  
    throws IOException
```

Reads the next byte of data from the input stream. The value byte is returned as an `int` in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

A subclass must provide an implementation of this method.

Returns:

the next byte of data, or -1 if the end of the stream is reached.

Throws:

`IOException` - if an I/O error occurs.

TestInputStream.java

```
import java.io.*;

public class TestInputStream {

    public static void main(String[] args) throws IOException {
        InputStream in = new FileInputStream("message_file");
        int value;

        while ((value = in.read()) != -1) {
            System.out.print((char)value);
        }

        System.out.println();
        in.close();
    }
}
```

```
javac TestInputStream.java
java TestInputStream
Hello world!e
lo wo
```

Thank you!

