Mahidol University
Faculty of Information
and Communication Technology

LECTURE 11
Exceptions Handling & File I/O

ITCS123 Object Oriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

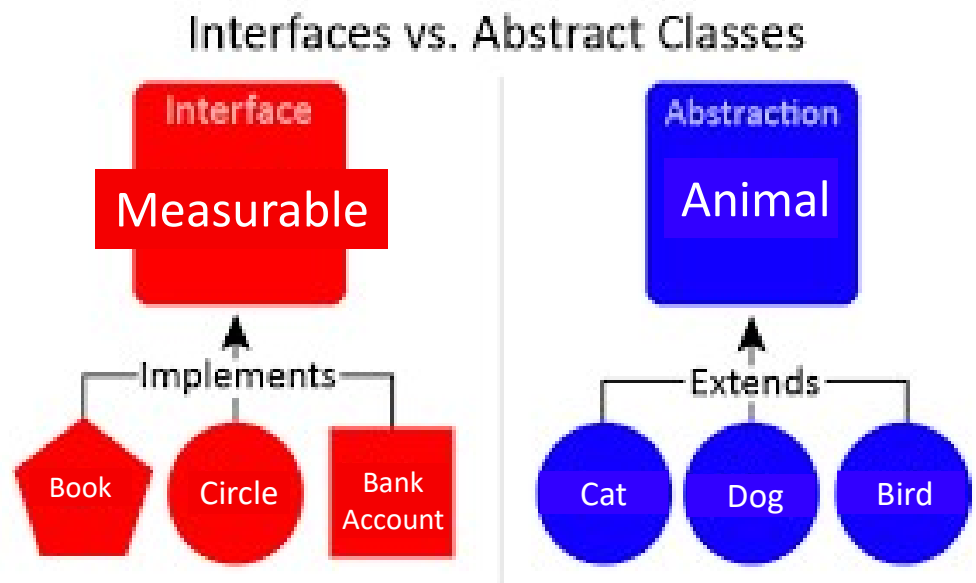
(Some materials in the lecture are done by Aj. Suppawong Tuarob)

Ref: Java Concepts Early Objects by Cay Horstmann

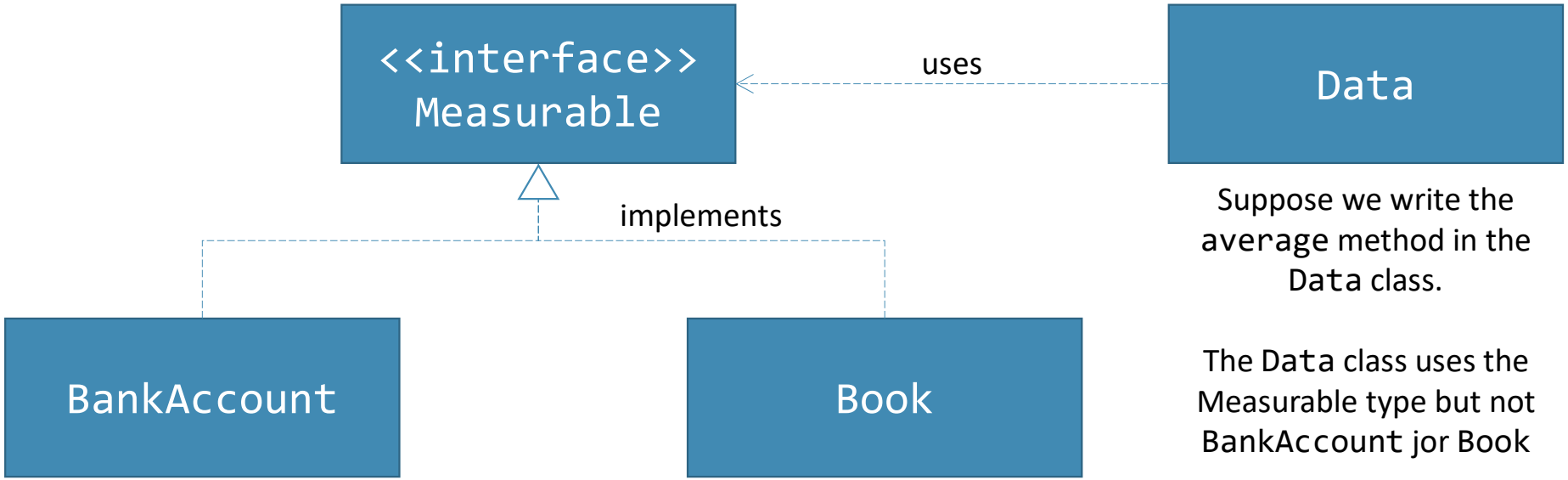
Semester 2/2023 28 Mar 2023

Review Interface

- Declare and implement interfaces
- Using interface for algorithm reuse
  - e.g., average(Measurable[] obj)
- Interface vs Abstract Class
  - Implements many interfaces // OK
- Comparable Java Standard Interface
  - Support sort() method in java



UML Diagram



Suppose we write the `average` method in the `Data` class.

The `Data` class uses the Measurable type but not `BankAccount` jor Book

The `BankAccount` and `Book` classes implement the `Measurable` interface type

```java
public class Data {

    /*
     * finding average measures of any measurable objects
     * @param an array of Measurable objects
     * @return the average of the measures
     */
    public static double average(Measurable[] objects) {
        double sum = 0;
        for(Measurable obj: objects) {
            sum = sum + obj.getMeasure();
        }
        if(objects.length > 0) {
            return sum / objects.length;
        }
        return 0;
    }
}
```

```java
public class App2 {
    public static void main(String[] args) {
        BankAccount2[] accounts2 = new BankAccount2[3];
        accounts2[0] = new BankAccount2(1, 100);
        accounts2[1] = new BankAccount2(2, 200);
        accounts2[2] = new BankAccount2(3, 300);
        System.out.println("Average Measurement of BankAccount2: "
                + Data.average(accounts2));

        Book2[] books2 = new Book2[3];
        books2[0] = new Book2("Java Prog", 200);
        books2[1] = new Book2("OOP Concept", 400);
        books2[2] = new Book2("Python Wow", 600);
        System.out.println("Average Measurement of Book2: "
                + Data.average(books2));
    }
}
```

Checkpoint: Which statements cause ERROR?

- Suppose there are two classes and two interfaces as follow:
- `public class` *`ClassA`*
- `public abstract class` *`ClassB`*
- `public interface` *`InterfaceC`*
- `public interface` *`InterfaceD`*

Class Declaration

```
a) public class X extends ClassA
b) public class Y extends ClassB
c) public class Z implements InterfaceC
d) public class AC extends ClassA implements InterfaceC
e) public class AB extends ClassA, ClassB
f) public class CD implements InterfaceC, InterfaceD
```

Instantiate Objects

```
1) ClassA var = new ClassA();
2) ClassB var = new ClassB();
3) InterfaceC var = new InterfaceC();
4) ClassA var = new X();
5) ClassB var = new Y();
6) InterfaceC var = new CD();
```

Checkpoint: Which statements cause ERROR?

- Suppose there are two classes and two interfaces as follow:
- `public class ClassA`
- `public abstract class ClassB`
- `public interface InterfaceC`
- `public interface InterfaceD`

Class Declaration

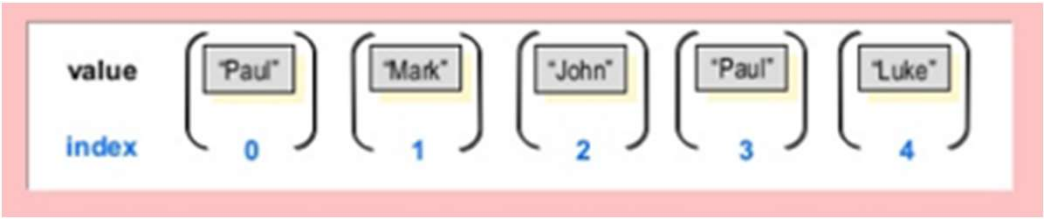
```
a) public class X extends ClassA
b) public class Y extends ClassB
c) public class Z implements InterfaceC
d) public class AC extends ClassA implements InterfaceC
e) public class AB extends ClassA, ClassB
f) public class CD implements InterfaceC, InterfaceD
```

Instantiate Objects

```
1) ClassA var = new ClassA();
2) ClassB var = new ClassB();
3) InterfaceC var = new InterfaceC();
4) ClassA var = new X();
5) ClassB var = new Y();
6) InterfaceC var = new CD();
```

Answer:
Statements e, 2, and 3

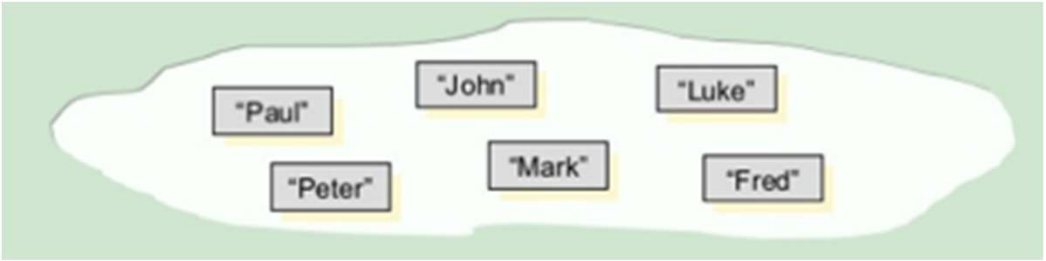
Review Collection



List: Lists of things (classes that implement List)

* cares about the index

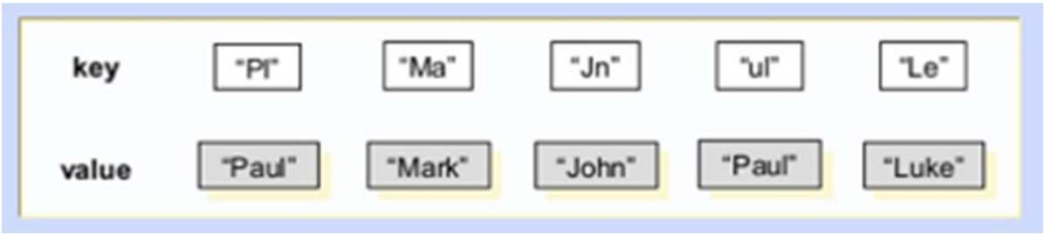
e.g., **ArrayList, Vector, LinkedList**



Set: Unique things (classes that implement Set)

* cares about uniqueness, no duplicate

e.g., **HashSet, LinkedHashSet, TreeSet**



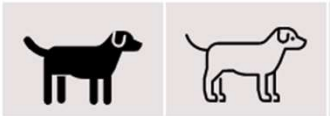
Map: Things with a unique ID (classes that implement Map)

* cares about unique identifiers (key-value pair)

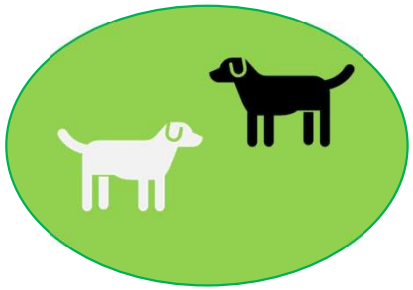
e.g., **HashMap, HashTable, TreeMap**

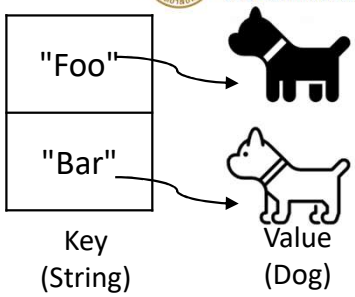
Summary

dogs =



[0] [1]





	ArrayList	Set	
Add/Insert new element	`dogs.add(new Dog(12, "black”));`	`dogs.add(new Dog(12, "black”));`	`dogs.put("key", new Dog(12, "black”));`
Retrieve element	`dogs.get(0); // index`	Cannot directly get by index	`dogs.get("key"); // key`
Remove element	`dogs.remove(0); // index`	`dogs.remove(dogObject); // Object`	`dogs.remove("key"); // key`
Size / is it empty?	`dogs.size(); dogs.isEmpty()`	`dogs.size(); dogs.isEmpty( );`	`dogs.size(); dogs.isEmpty();`
Iterate (loop through all elements)	`for(Dog d: dogs) { /* do s.th */ };`	`for(Dog d: dogs) { /* do s.th */ };`	Loop using keySet(); method
Find specific element	`dogs.contains(dogObject);`	`dogs.contains(dogObject);`	`dogs.containsKey("key"); // OR dogs.containsValue(dogObject);`

2. Common Ways to Traverse a Collection

- **Normal For Loop**

```
for(int i=0; i < objects.size(); i++) { . . . }
```

- **For-Each Loop**

```
for(Object obj: objects) { . . . }
```

- **Iterator & While Loop**

```
Iterator<Object> it = objects.iterator();
while(it.hasNext()) { . . . }
```

- `forEach()` **method**

```
objects.forEach(obj -> { . . . } );
```

For-Each loop vs forEach() method

```java
//ArrayList
for(Dog dog: dogList){
  System.out.println("List => " + dog);
}

// Set
for(Dog dog: dogSet){
  System.out.println("Set => " + dog);
}

// Map -> have to loop through keySet() instead
for(String key: dogMap.keySet()){
  System.out.println("Map => key: " + key
    + ", value " + dogMap.get(key));
}
```

```java
// ArrayList
dogList.forEach(dog -> {
  System.out.println("List => " + dog);
});

// Set
dogSet.forEach(dog -> {
  System.out.println("Set => " + dog);
});

// Map
dogMap.forEach((k, v)-> {
  System.out.println("key: " + k + ", value " + v);
});
```

Example Iterator: List vs Set vs Map

```
Iterator<Dog> cursorList = dogList.iterator();

while(cursorList.hasNext()) {
  System.out.println("List => " + cursorList.next());
}
```

```
Iterator<Dog> cursorSet = dogSet.iterator();

while(cursorList.hasNext()) {
  System.out.println("Set => " + cursorList.next());
}
```

Move cursor to next element

```
// For Map, we cannot get value (Dog) directly

// 1. Getting a Set of key-value pairs
Set entrySet = dogMap.entrySet();

// 2. Obtaining an interator for the entry set (key-value)
Iterator<Map.Entry<String, Dog>> it = entrySet.iterator();

while(it.hasNext()) {
  Map.Entry mapElement = it.next();
  System.out.println("Map => key: " + mapElement.getKey() +
                     ", value " + mapElement.getValue());
}
```

Map.Entry<key, value>
In this example, key is String and value is Dog.

mapElement.getKey() -> return key
mapElement.getValue() -> return value

OUTPUT
Map => key: Bar, value age: 10, color: white
Map => key: Foo, value age: 12, color: black

Dog Class

```
public String toString(){
  return "age: " + age + ", color: " + color;
}
```

Today Topics

- Errors in the Program
- Exception Handling in Java
- File Input/Output

Class Learning Outcomes (CLOs)

- Students can *explain* different types of errors in the program
- Students can *write* a program to handle errors in the program appropriately
- Students can *write* a program to read and write files

0. Errors

- Syntax errors
  - arise because the rules of the language have not been followed.
  - detected by the compiler.
- *Logic errors*
  - leads to wrong results and detected during testing.
  - arise because the logic coded by the programmer was not correct.
- *Runtime errors*
  - Occur when the program is running and the environment detects an operation that is impossible to carry out.
    - E.g., Divide by zero, Array out of bounds, Integer overflow, Accessing a null pointer (reference)

Example

```java
public static void main(String[] args){
    int[] numbers = new int[10];

    for(int i = 0; i < numbers.length(); i++){
        numbers[i] = 10 - i;
    }

    for(int i = 0; i < numbers.length; i++){
        // print values that higher than 5
        if(numbers[i] < 5)
            System.out.println(numbers[i]);
    }

    // print all values in the array numbers
    for(int i = 0; i <= numbers.length; i++){
        System.out.println(numbers[i]);
    }
}
```

```java
// Invalid input from user might occur !!!

Scanner scan = new Scanner(System.in);
System.out.print("Enter any number: ");
int num = scan.nextInt();
System.out.println("Your number is " + num);
scan.close();
```

```
Enter any number: ing
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Main.main(Main.java:7)
exit status 1
```

Exception Handling

- Throwing Exception
- Catching Exception
- Throws, Finally
- User-Defined Exception Type

1. Exception Handling

- Where there is an *error*
  - the program will *crash* (program terminated unexpectedly)
- There are two aspects to dealing with program errors: *detection* and *handling*.
  - For example, if a user enters an invalid inputs into the Scanner class, the program can terminate the program, or ask the user for a new input. However, the Scanner cannot choose by itself.
- **Exception handling** provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.

1.1 Throwing Exceptions

- When you detect an error condition, you just need to ***throw*** an appropirate **exception object**.
- For example, someone tries to withdraw too much money from a bank account.

```
if (amount > balance) {
    // now what?
}
```

- So, what is an appropriate exception object? There are bunch of them in Java standard libray (as shown in next slide)

```
if (amount > balance) {
    throw new IllegalArgumentException("Amount exceeds balance");
}
```

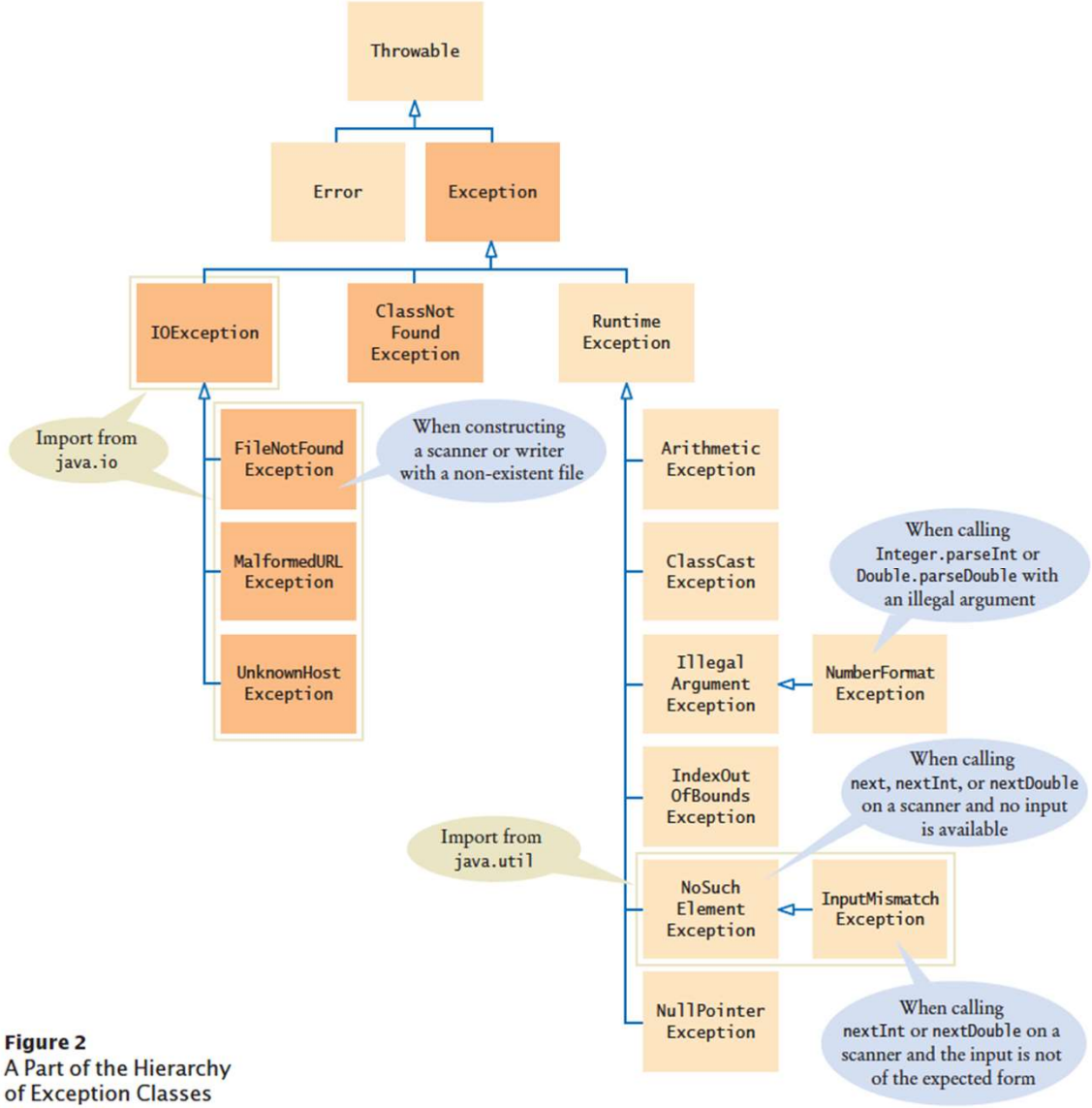


Figure 2
A Part of the Hierarchy of Exception Classes

1.1.1 Thorwing Exception Syntax

Syntax 11.1 Throwing an Exception

Syntax `throw` *exceptionObject*;

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

A new exception object is constructed, then thrown.

This line is not executed when the exception is thrown.

1.2 Catching Exceptions

- Every exception should be handled.
  - If it has no hanlder, and error message is printed, and the program terminates
- **`try/catch`** statement is used here.
- The **`try`** block contains one or more statemetns that may cause an exception that you want to handle.
- Each **`catch`** clause contains the handler for an exception type

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```java
Scanner scan = new Scanner(System.in);
try{
  System.out.print("Enter any number: ");
  int numSure = scan.nextInt();
  System.out.println("Your number is " + numSure);
  System.out.print("Enter any number again: ");
  String stringSure = scan.next();
  double num2 = Double.parseDouble(stringSure);
  System.out.println("Your number is " + num2);

  System.out.println("Good job. Bye!");
  scan.close();

} catch (InputMismatchException e){
  System.out.println("Your input is not a number");
} catch (NumberFormatException e){
  System.out.println("Cannot convert your input to number");
}
```

Question: What will in a user's input is

Option 1: 1 and 2
Option 2: ing
Option 3: 1 and ing
Option 4: ing and ing

1.2.1 Explain Previous Example

- There are two exceptions may be thrown in this try block:
  - The scan.nextInt(); can thorw InputMismatchException
  - The Double.parseDouble(stringSure); can throw NumberFormatException
- If any of these exceptions is actually thrown, then the rest of the instructions in the try block are **skipped**.

No Exception

```
Enter any number: 1
Your number is 1
Enter any number again: 2
Your number is 2.0
Good job. Bye!
```

InputMismatchException

```
Enter any number: ing
Your input is not a number
```

NumberFormatException

```
Enter any number: 1
Your number is 1
Enter any number again: ing
Cannot convert your input to number
```

1.2.2 Getting Information from Exceptions

- Use instance methods of the `java.lang.Throwable` class
- Some useful methods:

```
String toString()       Returns a short description of the exception
String getMessage()     Returns the detail description of the exception
void printStackTrace() Prints the stacktrace information on the console
```

- Exxample

```
catch (NumberFormatException e){
  System.out.println("Cannot convert your input"
                     + " to number");

  System.out.println("getMessage():"
                     + e.getMessage() + "\n\n");

  e.printStackTrace();
}
```

```
Enter any number again: ing
Cannot convert your input to number
getMessage():For input string: "ing"


java.lang.NumberFormatException: For input string: "ing"
    at java.base/jdk.internal.math.FloatingDecimal.readJavaForma
tString(FloatingDecimal.java:2054)
    at java.base/jdk.internal.math.FloatingDecimal.parseDouble(F
loatingDecimal.java:110)
    at java.base/java.lang.Double.parseDouble(Double.java:543)
    at Main.main(Main.java:49)
Here in finally block
```

Computing & Society 11.2 The Ariane Rocket Incident

The European Space Agency (ESA), Europe's counterpart to NASA, had developed a rocket model called Ariane that it had successfully used several times to launch satellites and scientific experiments into space. However, when a new version, the Ariane 5, was launched on June 4, 1996, from ESA's launch site in Kourou, French Guiana, the rocket veered off course about 40 seconds after liftoff. Flying at an angle of more than 20 degrees, rather than straight up, exerted such an aerodynamic force that the boosters separated, which triggered the automatic self-destruction mechanism. The rocket blew itself up.

The ultimate cause of this accident was an unhandled exception! The rocket contained two identical devices (called inertial reference systems) that processed flight data from measuring devices and turned the data into information about the rocket position. The onboard computer used the position information for controlling the boosters. The same inertial reference systems and computer software had worked fine on the Ariane 4.

However, due to design changes to the rocket, one of the sensors measured a larger acceleration force than had been encountered in the Ariane 4. That value, expressed as a floating-point value, was stored in a 16-bit integer (like a `short` variable in Java). Unlike Java, the Ada language, used for the device software, generates an exception if a floating-point number is too large to be converted to an integer. Unfortunately, the programmers of the device had decided that this situation would never happen and didn't provide an exception handler.

When the overflow did happen, the exception was triggered and, because there was no handler, the device shut itself off. The onboard computer sensed the failure and switched over to the backup device. However, that device had shut itself off for exactly the same reason, something that the designers of the rocket had not expected. They figured that the devices might fail for mechanical reasons, but the chance of them having the same mechanical failure was remote. At that point, the rocket was without reliable position information and went off course.

Perhaps it would have been better if the software hadn't been so thorough? If it had ignored the overflow, the device wouldn't have been shut off. It would have computed bad data. But then the device would have reported wrong position data, which could have been just as fatal. Instead, a correct implementation should have caught overflow exceptions and come up with some strategy to recompute the flight data. Clearly, giving up was not a reasonable option in this context.

The advantage of the exception-handling mechanism is that it makes these issues explicit to programmers—something to think about when you curse the Java compiler for complaining about uncaught exceptions.

The Explosion of the Ariane Rocket

1.3 Checked Exceptions

- **Unchecked Exception:** indicate errors in your code. You should deal with it. It is your falut if you didn't handle and your program fails. (no one knows better than you – programmers)
  - E.g., `IndexOutOfBoundsException` or `IllegalArgumentException`

- **Checked Eception:** indicate that something has gone wrong from some external reason *beyound your control* such as disk error, broken network connection. There exceptions still need to be handled seriously. So, the compiler will force you to handle these exceptions. Otherwise, your code will not be compiled.
  - E.g., `FileNotFoundException`

```
try
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile); // Throws FileNotFoundException
   . . .
}
catch (FileNotFoundException exception) // Exception caught here
{
   . . .
}
```

1.4 Throws

- It commonly happends that you cannot handle the exception, and you just want your method to be terminated when it occurs. Let the caller of your method to handle it.
- To do so, you can supply the method with a `throws` clause

```
public void readData(String filename) throws FileNotFoundException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   . . .
}
```

- This throws clause informs the caller of your method that it may encounter a FileNotFoundException. The caller then have to decide whether to handle the exception or thorw it again
- This mechanism allow an exception to be sent to the appropriate handler.

Someone detect, someone else can handle -> e.g., you detect a fire, you call the fireman to stop the fire

1.4.1 Thorws Syntax

The throws Clause

Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*
throws *ExceptionClass, ExceptionClass, . . .*

```
public void readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Principle: Throw an exception as soon as a problem is detected.
Catch it only when the problem can be handled.

1.5 The finally Clause

- In some cases, you may want to take some action whether or not an exception is thrown.
- A common example is to close the Scanner after used to avoid resource leak.

```
Scanner scan = new Scanner(System.in);
System.out.print("Enter any number: ");
int num = scan.nextInt();
System.out.println("Your number is " + num);
scan.close();        // May never get here
```

```
Scanner scan = new Scanner(System.in);
try{
  System.out.print("Enter any number: ");
  int num = scan.nextInt();
  System.out.println("Your number is " + num);
  System.out.println("Good job. Bye!");

} catch (InputMismatchException e){
  System.out.println("Your input is not a number");
} finally {
  scan.close();
}
```

```java
Scanner scan = new Scanner(System.in);
try{
  System.out.print("Enter any number: ");
  int numSure = scan.nextInt();
  System.out.println("Your number is " + numSure);
  System.out.print("Enter any number again: ");
  String stringSure = scan.next();
  double num2 = Double.parseDouble(stringSure);
  System.out.println("Your number is " + num2);
  System.out.println("Good job. Bye!");

} catch (InputMismatchException e){
  System.out.println("Your input is not a number");
} catch (NumberFormatException e){
  System.out.println("Cannot convert your input to number");
} finally {
  System.out.println("Here in finally block");
  scan.close();
}
```

Quesiton:
If an exception occurs in the try block, will the program print “Good job. Bye!”

How about “Here in finally block”?

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

1.6 Designing Your Own Exception Types

- If the standard exception types cannot describe your exception well enough. You can design your own exception class.
- For example, to withdraw money from a bank account

```
if (amount > balance) {
    throw new InsufficientFundsException(
          "withdrawal of " + amount + " exceeds balance of " + balance);
}
```

- This new exception type must inherits from an appropriate standard Exception.
- This exception should provide two constructors:
  - `InsufficientFundsException() {..}`
  - `InsufficientFundsException(String message) {..}`

```java
public class InvalidRadiusException extends Exception {
    private double radius;
    public InvalidRadiusException() {
        super("invalid radius!");
    }
    public InvalidRadiusException(double radius) {
         super("Invalid radius ");
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
}
```

```java
public class Circle {
    private double radius;
    private static int numberOfObjects = 0;

    public Circle() {   this(1.0);  }

    public Circle(double newRadius) throws IllegalArgumentException
    {
        setRadius(newRadius);    numberOfObjects++;
    }

    public double getRadius() {    return radius;   }

    public void setRadius(double newRadius)
            throws IllegalArgumentException {

        if (newRadius >= 0)
            radius =  newRadius;
        else
            throw new IllegalArgumentException(
                    "Radius cannot be negative");
    }

    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
}
```

```java
public class TestCircle {
    public static void main(String[] args) {
        try {
            Circle c1 = new Circle(5);
            Circle c2 = new Circle(-5);
            Circle c3 = new Circle(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }
        System.out.println("Number of objects created: "
                        + Circle.getNumberOfObjects());
    }
}
```

Output:

Invalid radius: - 5.0
Number of objects created: 1

1.6.1 When to create Custom Exception classes

- Use the exception classes in the API whenever possible.

- You should write your own exception classes if you answer ‘yes’ to one of the following:

  - ✓ **Do you need an exception type that isn't represented by those in the Java platform?**
  - ✓ **Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?**
  - ✓ **Do you want to pass more than just a string to the exception handler?**

1.7 When to Use Exceptions

- Use it if the event is truly exceptional and is an error
- Do not use it to deal with simple, expected situations.

Example:

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

Can be replaced by:

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Checkpoint

T F 1) When an exception is thrown by code inside a try block, all of the statements in the try block are always executed.

T F 2) `IOException` serves as a superclass for exceptions that are related to programming errors, such as an out-of-bounds array.

T F 3) One `try` block can have many `catch` clauses.

T F 4) All exception classes inherit from `Throwable` class.

T F 5) `finally` block is executed no matter the try block throws an exception or not.

T F 6) Java exception handling is a mechanism for handling exception by *detecting* and *handling* to exceptions in a systematic, uniform and reliable manner.

Mahidol University
Faculty of Information and Communication Technology

Exercise

What's the output?

```
public class Circle {
    private double radius;
    private static int numberOfObjects = 0;

    public Circle() {   this(1.0);  }

    public Circle(double newRadius) throws IllegalArgumentException
    {
        setRadius(newRadius);    numberOfObjects++;
    }

    public double getRadius() {    return radius;   }

    public void setRadius(double newRadius)
            throws IllegalArgumentException {

        if (newRadius >= 0)
            radius =  newRadius;
        else
            throw new IllegalArgumentException(
                    "Radius cannot be negative");
    }

    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
}
```

```
public class TestCircle {
    public static void main(String[] args) {
        try {
            Circle c1 = new Circle(5);
            Circle c2 = new Circle(-5);
            Circle c3 = new Circle(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }
        System.out.println("Number of objects created: "
                        + Circle.getNumberOfObjects());
    }
}
```

Output:

What's the output?

```
public class Circle {
    private double radius;
    private static int numberOfObjects = 0;

    public Circle() {   this(1.0);  }

    public Circle(double newRadius) throws IllegalArgumentException
    {
        setRadius(newRadius);    numberOfObjects++;
    }

    public double getRadius() {    return radius;   }

    public void setRadius(double newRadius)
            throws IllegalArgumentException {

        if (newRadius >= 0)
            radius =  newRadius;
        else
            throw new IllegalArgumentException(
                    "Radius cannot be negative");
    }

    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
}
```

```
public class TestCircle {
    public static void main(String[] args) {
        try {
            Circle c1 = new Circle(5);
            Circle c2 = new Circle(-5);
            Circle c3 = new Circle(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }
        System.out.println("Number of objects created: "
                        + Circle.getNumberOfObjects());
    }
}
```

Output:

java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1

Get more info!

- Java docs: Exception

  http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Exception.html

- Sun Tutorial on Exception Handling

  http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

- Exception Handling @mindprod.com

  http://mindprod.com/jgloss/exception.html

Summary

Two possible ways to deal:

```
void p1() {
  try {
    riskyMethod();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

"Something bad happened. This is how to handle it."

```
void p1() throws IOException {

  riskyMethod();

}
```

(b)

"Something bad happened. I failed."

Mahidol University
Faculty of Information and Communication Technology

File IO in Java

Slides modified from Ganesh Viswanathan's

1. File Basics

- Recall that a file is **block** structured. What does this mean?
- What happens when an application **opens** or **closes** a file?
- Every OS (Operating System) has its own EOF (end of file) character.
- Every OS also has its own EOL (end of line) character(s), for each text file

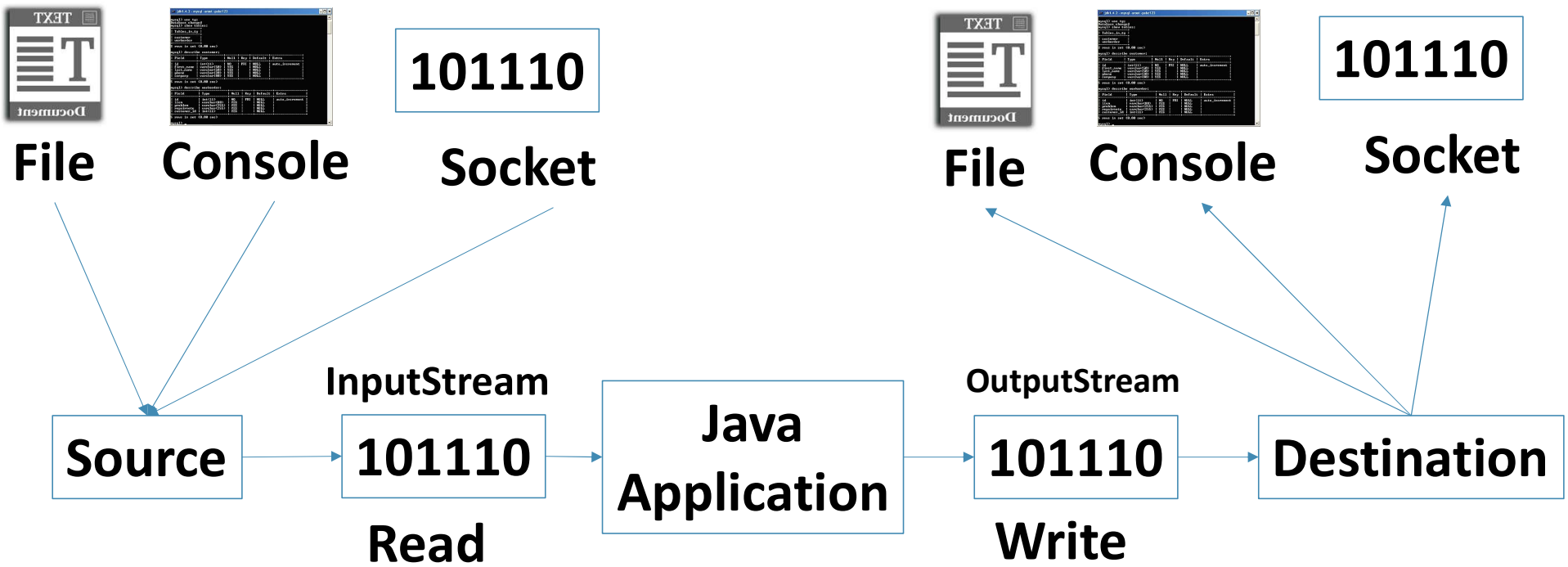
1.1 Streams

- Java file I/O involves **streams**. You write and read data to streams.

- The purpose of the stream abstraction is to keep program code independent from physical devices.

- Three stream objects are automatically created for every application: **`System.in`**, **`System.out`**, and **`System.err`**.

Fundamental concept in Java for handling data input and output (I/O)



1.2.1 Types of Streams

- There are 2 kinds of streams
  - byte streams
  - character streams

	Character Stream	Byte Stream
READ	Class Reader	Class InputStream
WRITE	Class Writer	Class OutputStream

Character Streams

- Character streams create text files.
- These are files designed to be read with a text editor.
- Java automatically converts its internal unicode characters to the local machine representation
  (ASCII in our case).

Byte Streams

- Byte streams create binary files.
- A binary file essentially contains the memory image of the data. That is, it stores bits as they are in memory.
- Binary files are faster to read and write because no translation need take place.
- But they cannot be read with a text editor.

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

2. Reading Text Files

- The most covenient mechanism for reading text file: ‘Scanner’ class
- To read input from a disk file, you need ‘File’ Class as well
  - File describes file’s name and directory
- Then, you can use the Scanner methods such as **`nextInt`**, **`nextDouble`**, and **`next`** to read data from the input file.

```java
File inputFile = new File("input.txt");
Scanner in = null;

try{
  in = new Scanner(inputFile);
  while (in.hasNextDouble()){
    double value = in.nextDouble();
    System.out.println("read: " + value);
  }
} catch(FileNotFoundException e){
  e.printStackTrace();
} finally{
  in.close();
}
```

input.txt

```
10
20
30
40
50
```

```
read: 10.0
read: 20.0
read: 30.0
read: 40.0
read: 50.0
```

2.1 File class and methods

```
File myDir = new File("C:\\ITCS123");
File myFile = new File("C:\\ITCS123\\junk.java");
File myFile = new File("C:\\ITCS123", "junk.java");
File myFile = new File(myDir, "junk.java").
```

- exists()
- isDirectory()
- isFile()
- canRead()
- canWrite()
- isHidden()
- getName()

getPath()
getAbsolutePath()
getParent()
list()
length()
renameTo(newPath)
delete()
mkdir()
createNewFile()

3. Writing Text File

- To write output to a file, you can use 'PrintWriter' object with the specific file name
- If the file already exit, the new data will replace the old one.
- If not, an empty file is created and the data is written.
- You can use the **`print`**, **`println`**, and **`prinf`** methods (similar to PrintStrem class)

```
try {
  out = new PrintWriter("output.txt");
  out.println("Hi, How are you?");
  out.printf("Total: %8.2f\n", 200.22);
} catch(FileNotFoundException e){
  e.printStackTrace();
} finally {
  if(out != null) out.close();
}
```

output.txt ×

```
Hi, How are you?
Total:   200.22
```

4. Process Data and Write Output

```java
File inputFile = new File("input.txt");
Scanner in = null;
PrintWriter out = null;
```

input.txt

```
10
20
30
40
50
```

total.txt

```
                10.00
                20.00
                30.00
                40.00
                50.00
Total:         150.00
```

```java
double total = 0.0;
try{
  in = new Scanner(inputFile);
  out = new PrintWriter("total.txt");
  while(in.hasNextDouble()){
    double value = in.nextDouble();
    out.printf("%15.2f\n", value);
    total += value;
  }
  out.printf("Total:%10.2f\n", total);
} catch(FileNotFoundException e){
  e.printStackTrace();
} finally {
  if(in != null) in.close();
  if(out != null) out.close();
}
```

5. Read Lines

- Many times, each line of a file represents one data record.
- So it is more conveninent to read the whole line with "`nextLine`" method

```
String line = in.nextLine();     // in is a Scanner class
```

- The next input line (without the newline character) is placed into the string line and ready for splitting apart and further processing
- The "`haseNextLine`" method is used to ensure there is another line to process. If all the lines have been read, this method return false.

```java
File inputFile = new File("order.txt");
Scanner in = null;
PrintWriter out = null;
double totalPrice = 0.0;
try{
  in = new Scanner(inputFile);
  out = new PrintWriter("receipt.txt");
  int lineNo = 1;

  while (in.hasNextLine()) {
    String line = in.nextLine();
    String[] data = line.split(" ");
    // write to file
    out.println(lineNo + ") menu: " + data[0] + ", price: " + data[1] + ", quantity: " + data[2]);
    totalPrice += Double.parseDouble(data[1]) * Integer.parseInt(data[2]);
    lineNo++;
  }
  out.printf("Total Price:%10.2f\n", totalPrice);
} catch(FileNotFoundException e){
  e.printStackTrace();
} finally {
  if(in != null) in.close();
  if(out != null) out.close();
}
```

order.txt

```
Coffee 50 1
Tea 40 2
Mile 30 2
```

receipt.txt

```
1) menu: Coffee, price: 50, quantity: 1
2) menu: Tea, price: 40, quantity: 2
3) menu: Mile, price: 30, quantity: 2
Total Price:    190.00
```

5.1 What if the format in the file is wrong T-T

order-wrong.txt

```
1   Coffee 50 1
2   Tea 40 two
3   Mile 30 2
```

```
1   Coffee 50 1
--> menu: Coffee, price: 50, quantity: 1
2   Tea 40 2
--> menu: Tea, price: 40, quantity: 2
3   Mile 30 two
--> menu: Mile, price: 30, quantity: two
Exception in thread "main" java.lang.NumberFormatException: For input
 string: "two"
    at java.base/java.lang.NumberFormatException.forInputString(Numbe
rFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.main(Main.java:28)
exit status 1
```

How to modify your program, so that the exception is handled.

Assume that the invalid input line will be ignored

5.2 Mix Format

- Note that the **nextInt**, **nextDouble**, and **next** methods do not consume the white space and new line character that follows the number or word.
- This can cause a problem if you alternatively call between **nextInt/nextDouble/next** and **nextLine**.

```
while (in.hasNextLine()) {
        String menu = in.nextLine();
        Double price = in.nextDouble();
        int qty = in.nextInt();    // did not consume newline character
        in.nextLine();  // dummy call to consume leftover newline
character
```

Without this line your menu in the next round will be empty string

```
        totalPrice += price * qty;
}
```

order-complex.txt

```
Coffee
50.0
1
Tea
40.5
2
Mile
30.5
2
```

Other Classes supporting stream I/O

- Java has 6 classes to support stream I/O
  - **`File`**: an object of this class is either a file or a directory.
  - **`OutputStream`**: base class for byte output streams
  - **`InputStream`**: base class for byte input streams
  - **`Writer`**: base class for character output streams.
  - **`Reader`**: base class for character input streams.
  - **`RandomAccessFile`**: provides support for random access to a file.
- Note that the classes `InputStream`, `OutputStream`, `Reader`, and `Writer` are *abstract* classes.

Writing TextFile using FileWriter Class

- The `FileWriter` class is a convenience class for writing character files.
- One version of the constructor take a `String` for a file name, another version takes an object of the `File` class.
- See `FileWriterDemo.java`

```java
public class FileWriterDemo {

    public static void main(String[] args)
    {
        BufferedWriter writer = null;
        File file = new File("log.txt");
        try {
            writer = new BufferedWriter(new FileWriter(file));
            writer.write("Hello World.\n"); //to write
            writer.append("Hello Mars.\n"); //to append

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }finally
        {
                try {
                    if(writer != null)
                        writer.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
    }
}
```

Reading One Char at a Time

- See `StreamReaderDemo.java`
- The `read()` method returns an integer
- This integer should be cast to a `char`
- A value of -1 indicates the end of the stream has been reached.

```java
public class StreamReaderDemo {
    public static void main(String[] args)
    {
        BufferedReader reader = null;
        File file = new File("log.txt");
        try {
            reader = new BufferedReader
                (new InputStreamReader
                (new FileInputStream(file)));

            int c = -1;

            while((c = reader.read()) != -1)
            {
                char character = (char) c;
                System.out.print(character+"|");
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }finally
        {
                try {
                    if(reader != null)
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
    }
```

Reading One Line at a Time

- See `LineReaderDemo.java`
- Use a `BufferedReader`
- The `readLine()` method returns a `String`.
- If the `String` is null, then the end of the stream has been reached.

```java
public class LineReaderDemo {
    public static void main(String[] args)
    {
        BufferedReader reader = null;
        File file = new File("log.txt");
        try {
            reader = new BufferedReader
                (new InputStreamReader
                (new FileInputStream(file)));

            String line = null;

            while((line = reader.readLine()) != null)
            {   line = line.trim();
                if(line.isEmpty()) continue;

                System.out.println(line);
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }finally
        {
            try {
                if(reader != null)
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

Reading One Word (Token) at a Time

- See `TokenReaderDemo.java`
- A word is a sequence of non-whitespace character.
- Use a Scanner
- `hasNext() = true` if there is one more word to read. False otherwise.
- `next()` returns the next word.

```java
public class TokenReaderDemo {
    public static void main(String[] args)
    {
        Scanner reader = null;
        File file = new File("log.txt");
        try {
            reader = new Scanner(file);

            String token = null;

            while(reader.hasNext())
            {   token = reader.next();
                System.out.print(token+"|");
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        finally
        {   if(reader != null) reader.close();
        }
    }
}
```

Shortcuts:
FileUtils from Apache Commons IO

Mahidol University
Faculty of Information and Communication Technology

```
File file = new File("log.txt");
//To write
try {
    FileUtils.write(file, "Hello World.\n");
} catch (IOException e) {e.printStackTrace();}

//To append
try {
    FileUtils.write(file, "Hello Mars.\n", true);
} catch (IOException e) {e.printStackTrace();}

//o read
try {
    List<String> lines = FileUtils.readLines(file);
    for(String line: lines)
    {   System.out.println(line);
    }
} catch (IOException e) {e.printStackTrace();}
```

Apache Commons™
http://commons.apache.org/