COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

ADVANCED PROGRAMMING COMP231

Instructor :Murad Njoum
Office : Masri322

Chapter 12 Exception Handling and Text IO

- An **Exception is a run-time error** which interrupts the normal flow of program execution. Disruption during the execution of the program is referred as error or exception.

- Errors are classified into two categories
    - **Compile time errors** – Syntax errors, Semantic errors
    - **Runtime errors- Exception**
- A **robust program should** **handle all exceptions** and continue with its normal flow of program execution. Java provides an inbuilt exceptional handling method
- **Exception Handler** is a set of code that **handles an exception**. Exceptions can be handled in Java **using try & catch**.
- **Try block**: Normal code goes on this block.
- **Catch block**: If there is error in normal code, then it will go into this block

Advantage of Exception Handling

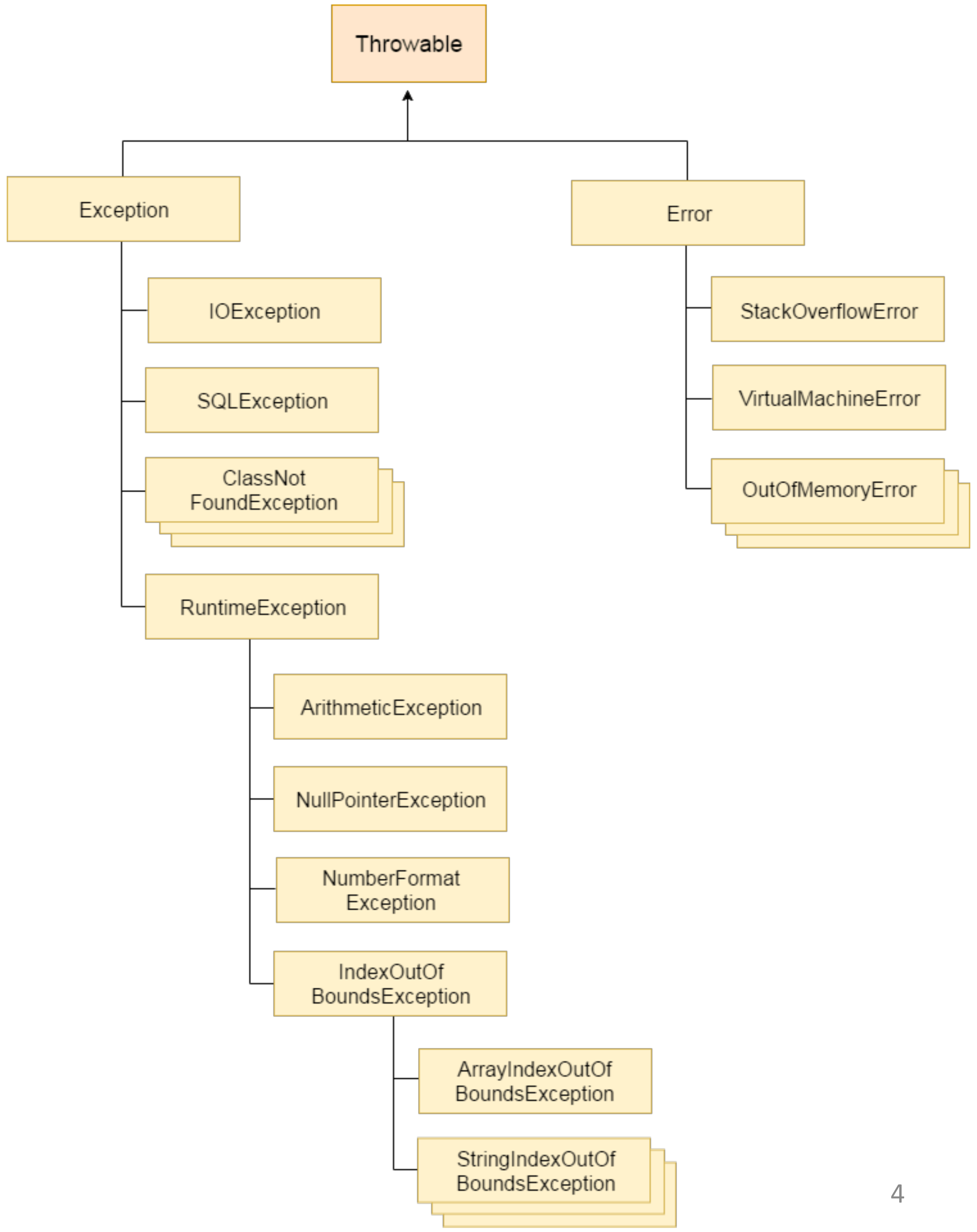
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
1.statement 1;
2.statement 2;
3.statement 3;
4.statement 4;
5.statement 5;//exception occurs
6.statement 6;
7.statement 7;
8.statement 8;
9.statement 9;
10.statement 10;
```

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The *java.lang.Throwable* class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions, Difference between Checked and Unchecked Exceptions

checked and unchecked. Here, an error is considered as the unchecked exception.

1) Checked Exception

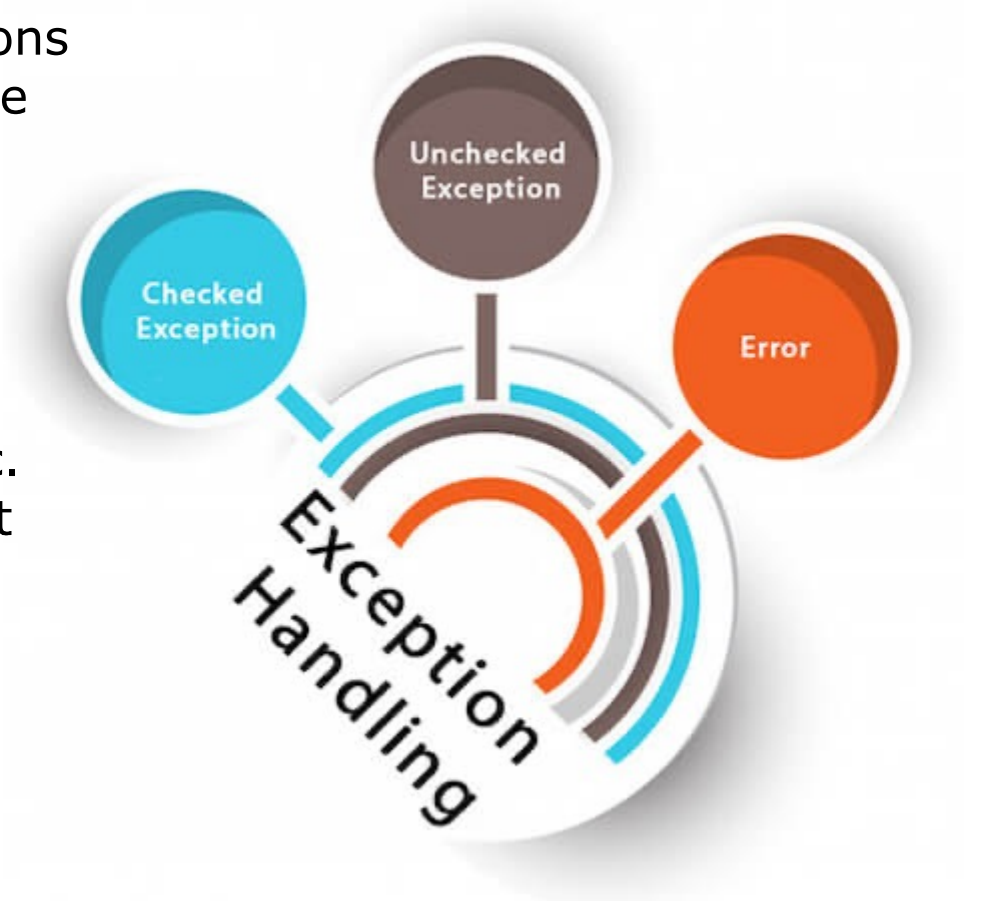
The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

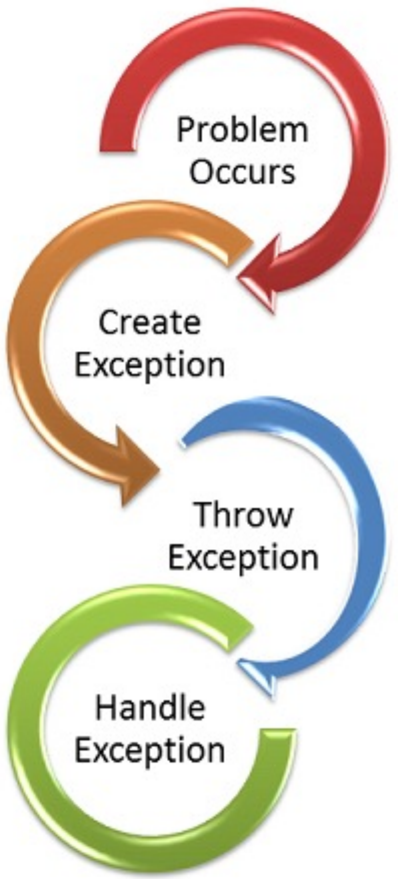
2) Unchecked Exception

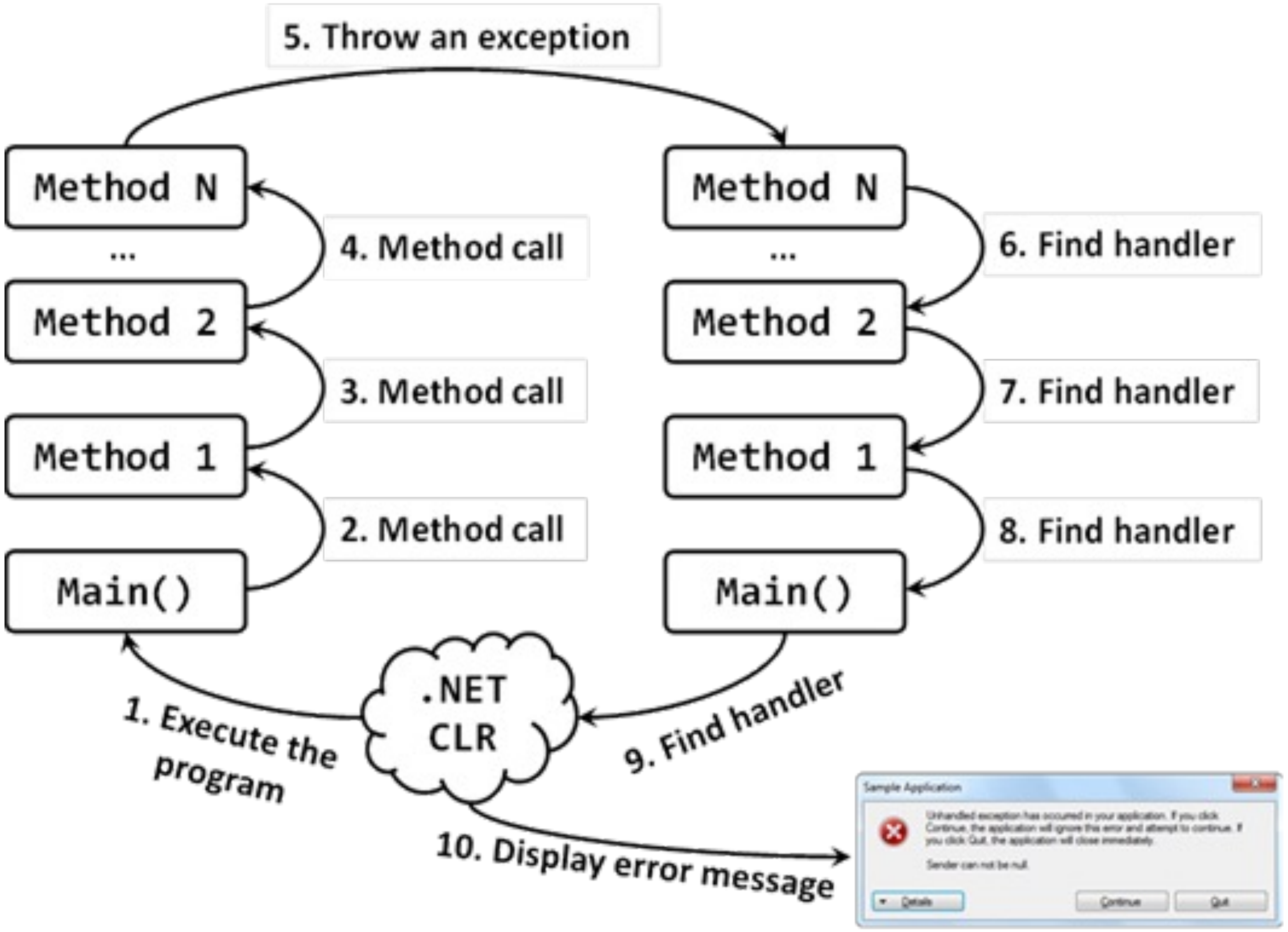
The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.







Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Sender can not be null.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1.int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1.String s=null;
2.System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
1.String s="abc";
2.int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
1.int a[]=new int[5];
2.a[10]=50; //ArrayIndexOutOfBoundsException
```

Syntax of Java try-catch

```
  try{
     //code that may throw an exception
   }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
   //code that may throw an exception
 }finally{}
```

Trace a Program Execution

Suppose no exceptions in the statements

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Next statement in the method is executed

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

Example: What is the output?

```
int []a= {1,2,3};
try {
  System.out.println(a[2]/2);;
  System.out.println(a[2]/0);;
  System.out.println(a[0]/2);;

}
catch(IllegalArgumentException ex) {
  System.out.println(ex.getMessage());
    }
catch(ArithmeticException ex) {
System.out.println(ex.getMessage());
  throw new IllegalArgumentException("2 . Welcome");
   }
finally {
  System.out.println("3. finalStatements");
}
```

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
    }catch(ArithmeticException e){System.out.println(e);
  }
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

```java
public class TryCatchExample5 {

    public static void main(String[] args) {
        try
        {
        int data=100/0; //may throw exception
        }
            // handling the exception
        catch(Exception e)
         {
                // displaying the custom message
             System.out.println("Can't divided by zero");
         }
      }

}
```

```
public class TryCatchExample9 {

  public static void main(String[] args) {
    try
     {
      int arr[]= {1,3,5,7};
     System.out.println(arr[10]); //may throw exception
      }
         // handling the array exception
     catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
     }
     System.out.println("rest of the code");
   }

}
```

```
public class MultipleCatchBlock3 {
public static void main(String[] args) {

   try{
    int a[]=new int[5];
    System.out.println(a[10]);
    a[5]=30/0;
    System.out.println(a[10]);
      }
   catch(ArithmeticException e)
          {System.out.println("Arithmetic Exception occurs");  }
   catch(ArrayIndexOutOfBoundsException e)
          {System.out.println("ArrayIndexOutOfBounds Exception occurs");
}
   catch(Exception e)
          {System.out.println("Parent Exception occurs");  }
 finally{System.out.println("Processed final");}
   System.out.println("rest of the code");
   }
}
```

```
ArrayIndexOutOfBounds Exception occurs
Processed final
rest of the code
```

Try-catch Blocks:

```
public class MultipleCatchBlock3 {

public static void main(String[] args) {
  int a[]=new int[5];
   try{
      System.out.println(a[10]);
      }
   catch(ArrayIndexOutOfBoundsException e)
           {System.out.println("ArrayIndexOutOfBounds Exception occurs");  }

  try{
     a[5]=30/0;
          System.out.println(a[10]);
     }
     catch(Exception e)
        {System.out.println("Parent Exception occurs"); }

      finally{System.out.println("Processed finnal");}
System.out.println("rest of the code");
  }
}
```

Nested try catch:

```
public class MultipleCatchBlock3 {

public static void main(String[] args) {
  int a[]=new int[5];
   try{
      System.out.println(a[10]);

         try{
            a[5]=30/0;
           System.out.println(a[10]);
           }catch(Exception e)
         {System.out.println("Parent Exception occurs"); }

      }
   catch(ArrayIndexOutOfBoundsException e)
           {System.out.println("ArrayIndexOutOfBounds Exception occurs");  }
   finally{System.out.println("Processed finnal");}
System.out.println("rest of the code");
  }
}
```

Fixing With a method

```
public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

```
public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }
    public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }
    System.out.println("Execution continues ..."); }}
```

Handling InputMismatchException

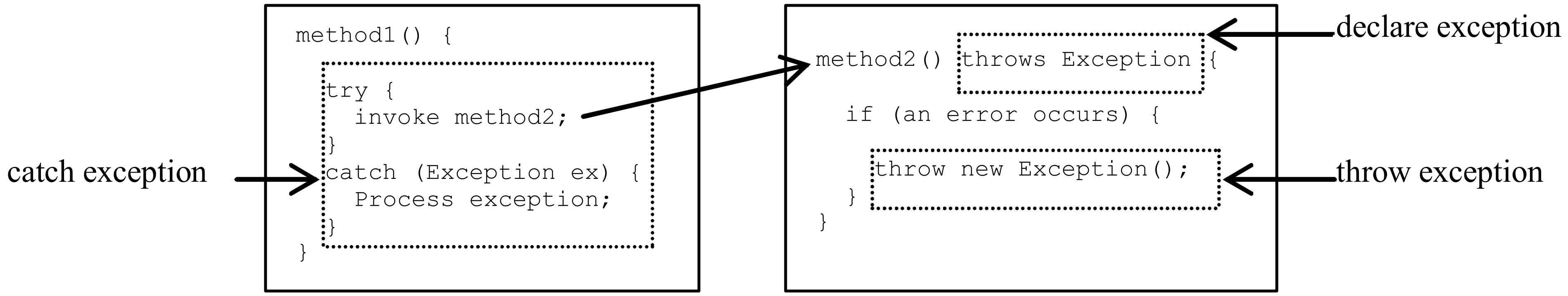
By handling InputMismatchException, your program will continuously read an input until it is correct.

```
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
          "The number entered is " + number);

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        input.nextLine(); // discard input
      }
    } while (continueInput);
  }
}
```

Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.
- For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it;
- an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program.
- Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Declaring, Throwing, and Catching Exceptions



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
  throws IOException

public void myMethod()
  throws IOException, OtherException
```

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```

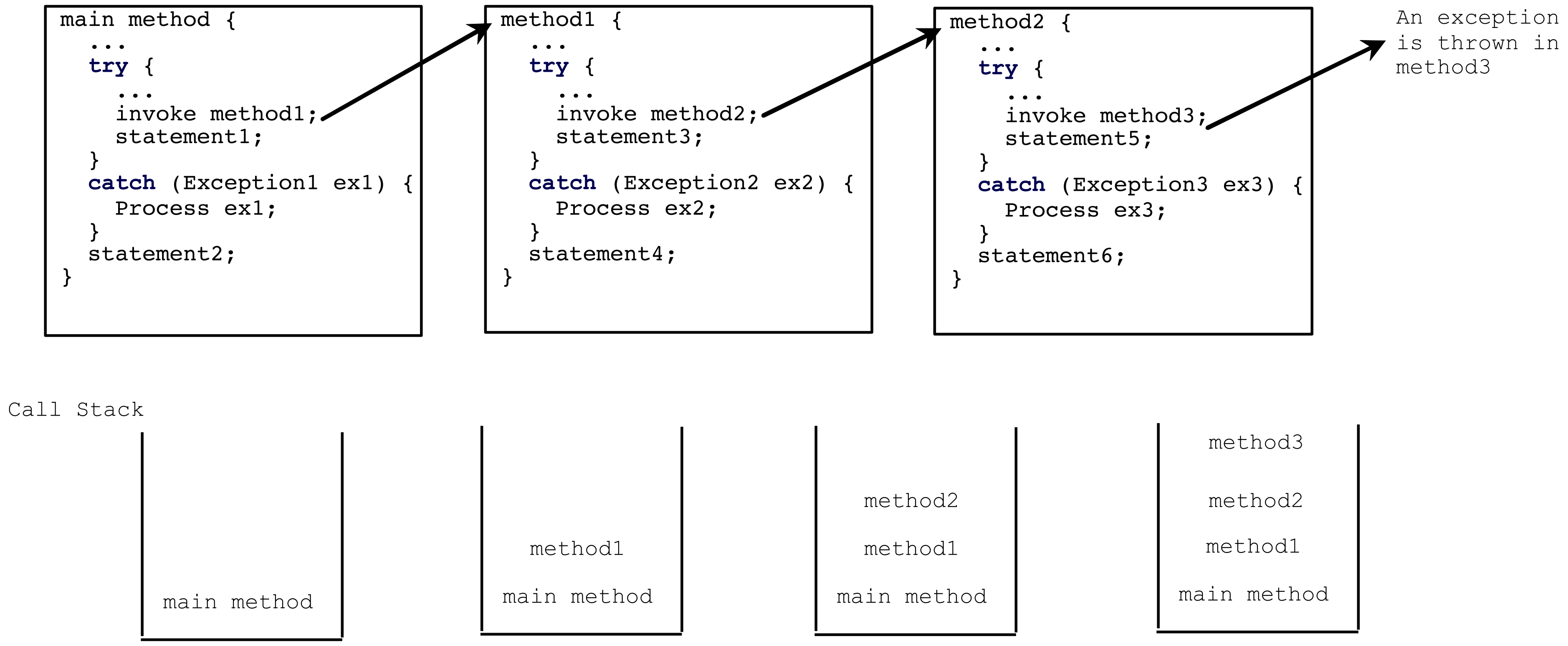
Throwing Exceptions Example

```
  /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
    }
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

Catching Exceptions



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

Catch or Declare Checked Exceptions

Suppose p2 is defined as follows:

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

Catch or Declare Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 9. The new setRadius method throws an exception if radius is negative.

CircleWithException

TestCircleWithException Run

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

public CircleWithException() {
    this(1.0);
  }

public CircleWithException(double newRadius) {
    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 double findArea() {
    return radius * radius * 3.14159;
  }
}
```

```java
public class TestCircleWithException {
  public static void main(String[] args) {
    try {
      CircleWithException c1 = new CircleWithException(5);
      CircleWithException c2 = new CircleWithException(-5);
      CircleWithException c3 = new CircleWithException(0);
    }
    catch (IllegalArgumentException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      CircleWithException.getNumberOfObjects());
  }
}
```

Rethrowing Exceptions

```
try {
  statements;
}
catch(TheException ex) {
  perform operations before exits;
  throw ex;
}
```

The `finally` Clause

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

For each try block there can be zero or more **catch** block, but only one **finally** block.

Example:

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
     { // Call constructor of parent Exception
     super(s); }
      }
// A Class that uses above MyException
public class Main
{ // Driver Program
public static void main(String args[])
  { try
   {
   // Throw an object of user defined exception
   throw new MyException("Comp231");
   }
catch (MyException ex)
{
System.out.println("Caught");
// Print the message from MyException object
System.out.println(ex.getMessage()); }
  }
}
```

```
Caught
Comp231
```

```java
import java.util.Scanner;
class MarriageAgeException extends Exception {
 public MarriageAgeException(String message) {
  super(message);
 }
}
public class MyOwnException {

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

  Scanner sc = new Scanner(System.in);

  System.out.println("Enter a person age");

  int age = sc.nextInt();
  if (age <= 30) {
      System.out.println("Valid for Marriage");

  } else {
      throw new MarriageAgeException("Maarige Age is Over Exception");
    }
  }
}
```

```java
class BelowAgeException extends Exception{
BelowAgeException(){
super("Excpetion :Age is under 18 cann't do it");}
}

class Application {
private String name;
private String course;
private int age;

public Application(String name,String course) {
this.name=name;
this.course=course;
age=18;
}
public Application() {
this("","");

}
public void setAge(int age) throws BelowAgeException{
if(age<18)
throw new BelowAgeException();
else
this.age=age;
}
```

```
the name of student :Ali
Applied for Java Programming
Applicant's Ag: 20

Excpetion :Age is under 18 cann't do it
Finally called
Procced job
```

```java
public void displayDetails() {
System.out.println("the name of student :"+name);
System.out.println("Applied for "+course);
System.out.println("Applicant's Ag: "+age);
System.out.println();}}

public class userDefinedExcpetion {
public static void main(String[] args) {
Application app1= new Application("Ali","Java 
Programming");
Application app2= new Application("Ahmad","Java 
Programming");

try {
app1.setAge(20);
app1.displayDetails();

app2.setAge(17);
app2.displayDetails();
}catch(BelowAgeException ex) {
System.out.println(ex.getMessage());
}finally {System.out.println("Finally called");}

System.out.println("Procced job ");
}
}
```

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

When to Throw Exceptions

- An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.

When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

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

When to Use Exceptions

is better to be replaced by

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

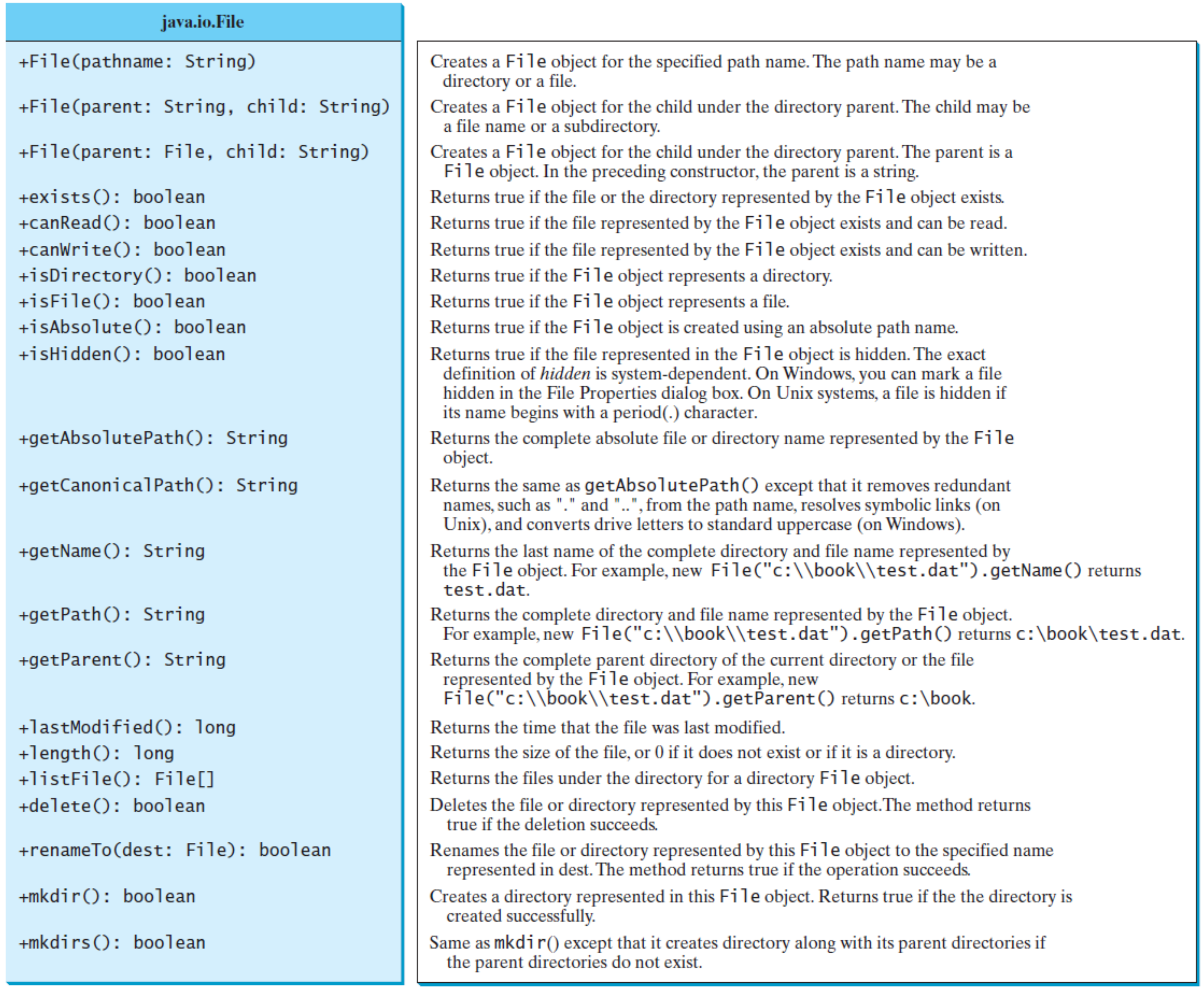
Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

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.

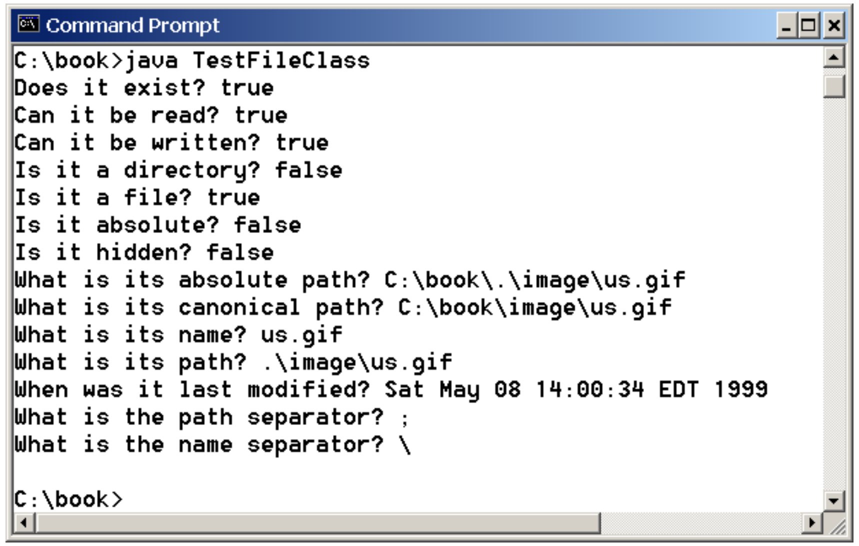
Obtaining file properties and manipulating file



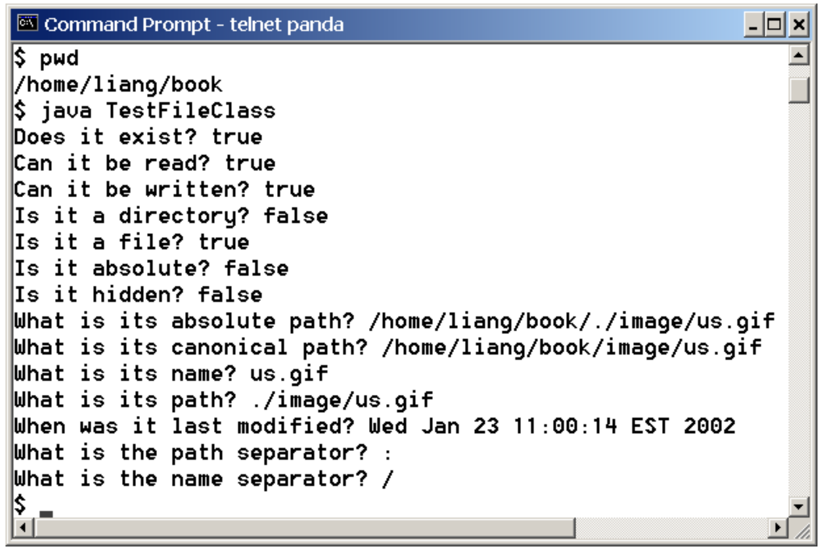
java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: 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 *hidden* 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 `getAbsolutePath()` 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.

Problem: Explore File Properties

Objective: Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. The following figures show a sample run of the program on Windows and on Unix.



```
C:\book>java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? C:\book\.\image\us.gif
What is its canonical path? C:\book\image\us.gif
What is its name? us.gif
What is its path? .\image\us.gif
When was it last modified? Sat May 08 14:00:34 EDT 1999
What is the path separator? ;
What is the name separator? \

C:\book>
```



```
$ pwd
/home/liang/book
$ java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? /home/liang/book/./image/us.gif
What is its canonical path? /home/liang/book/image/us.gif
What is its name? us.gif
What is its path? ./image/us.gif
When was it last modified? Wed Jan 23 11:00:14 EST 2002
What is the path separator? :
What is the name separator? /
$ _
```

TestFileClass Run

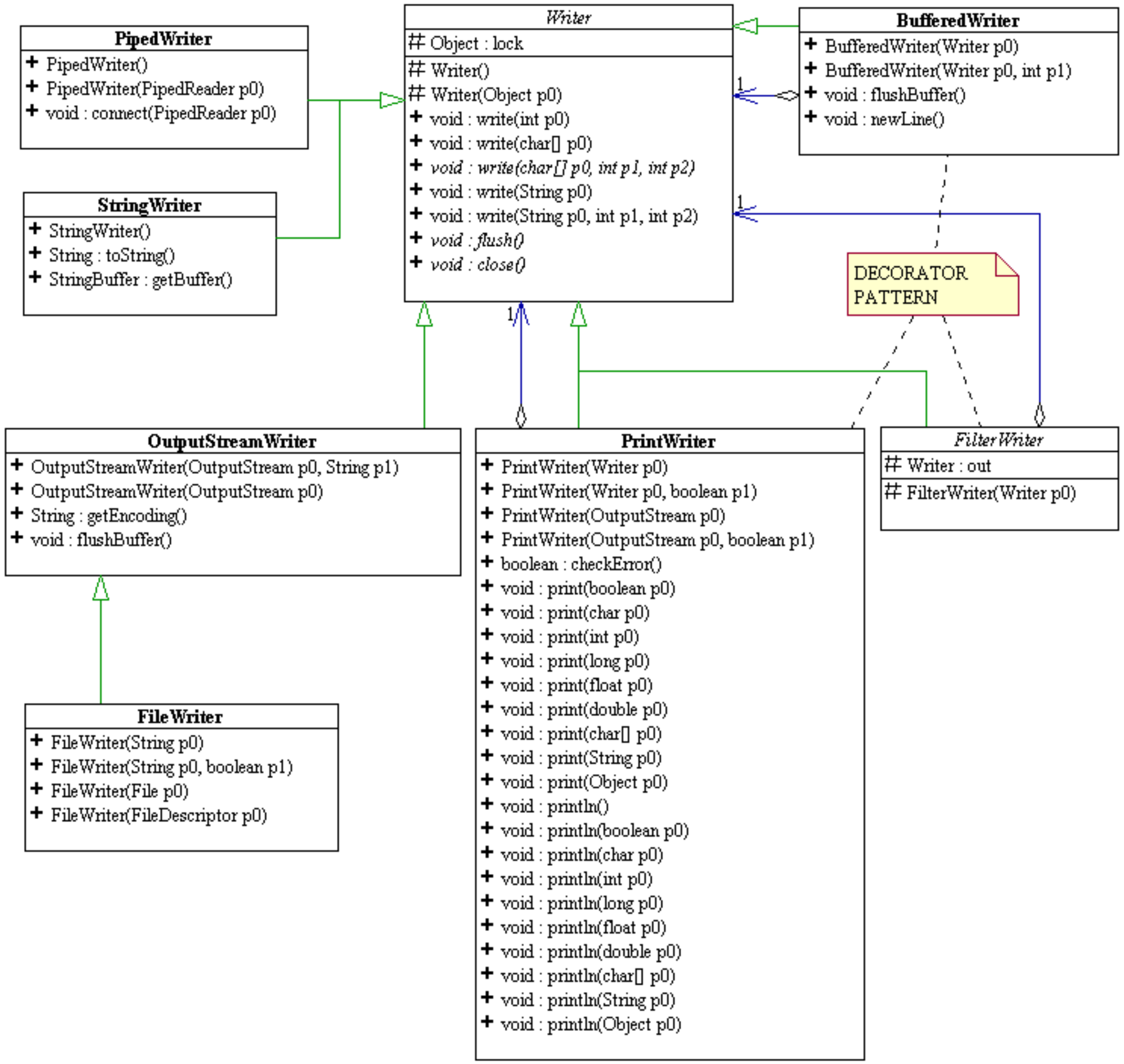
Text I/O

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.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output and Strings."

WriteData Run


PipedWriter
+ PipedWriter()
+ PipedWriter(PipedReader p0)
+ void : connect(PipedReader p0)
StringWriter
+ StringWriter()
+ String : toString()
+ StringBuffer : getBuffer()
Writer
# Object : lock
# Writer()
# Writer(Object p0)
+ void : write(int p0)
+ void : write(char[] p0)
+ void : write(char[] p0, int p1, int p2)
+ void : write(String p0)
+ void : write(String p0, int p1, int p2)
+ void : flush()
+ void : close()
BufferedWriter
+ BufferedWriter(Writer p0)
+ BufferedWriter(Writer p0, int p1)
+ void : flushBuffer()
+ void : newLine()
1
1
1
DECORATOR PATTERN
OutputStreamWriter
+ OutputStreamWriter(OutputStream p0, String p1)
+ OutputStreamWriter(OutputStream p0)
+ String : getEncoding()
+ void : flushBuffer()
PrintWriter
+ PrintWriter(Writer p0)
+ PrintWriter(Writer p0, boolean p1)
+ PrintWriter(OutputStream p0)
+ PrintWriter(OutputStream p0, boolean p1)
+ boolean : checkError()
+ void : print(boolean p0)
+ void : print(char p0)
+ void : print(int p0)
+ void : print(long p0)
+ void : print(float p0)
+ void : print(double p0)
+ void : print(char[] p0)
+ void : print(String p0)
+ void : print(Object p0)
+ void : println()
+ void : println(boolean p0)
+ void : println(char p0)
+ void : println(int p0)
+ void : println(long p0)
+ void : println(float p0)
+ void : println(double p0)
+ void : println(char[] p0)
+ void : println(String p0)
+ void : println(Object p0)
FilterWriter
# Writer : out
# FilterWriter(Writer p0)
FileWriter
+ FileWriter(String p0)
+ FileWriter(String p0, boolean p1)
+ FileWriter(File p0)
+ FileWriter(FileDescriptor p0)

Try-with-resources

try-with-resources syntax that automatically closes the files.

try (declare and create resources) {

Use the resource to process the file;

}

```
public class WriteDataWithAutoClose
{ public static void main(String[] args) throws
Exception
    { java.io.File file = new  File("scores.txt");
if (file.exists()) {
    System.out.println("File already    exists");
     System.exit(0); }
 try (PrintWriter output = new PrintWriter(file); )
 {
  output.print("John T Smith "); output.println(90);
  output.print("Eric K Jones "); output.println(85);
    }
    }
 }
```

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 Run

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

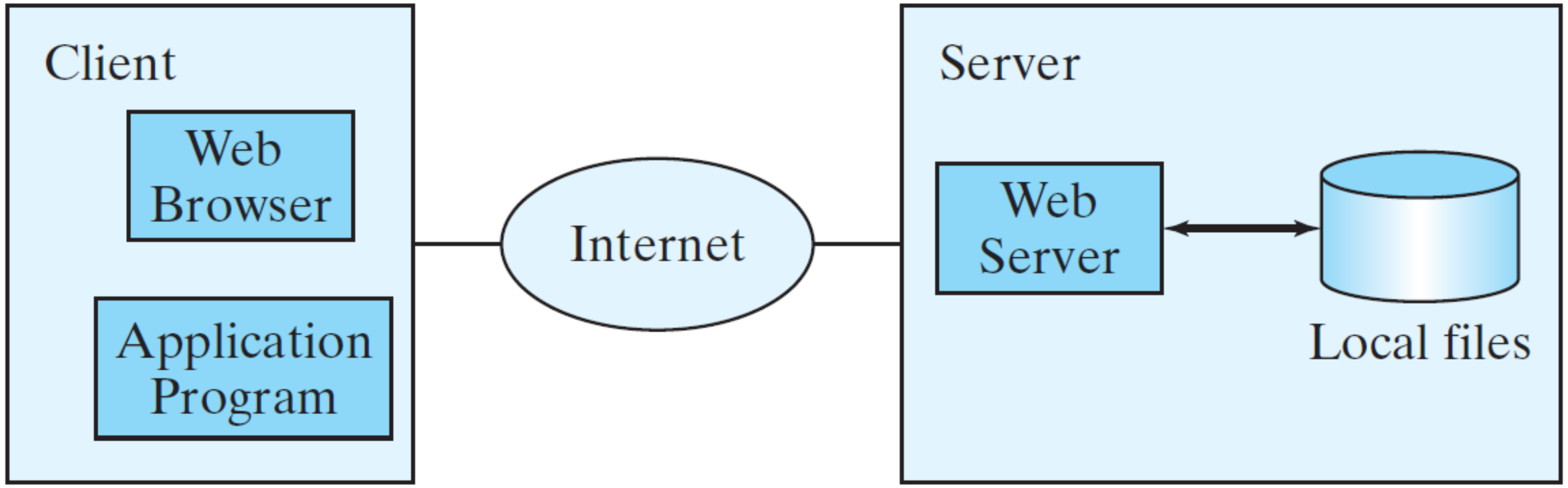
```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

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

ReplaceText   Run

Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.



Reading Data from the Web

```
URL url = new URL("www.google.com/index.html");
```

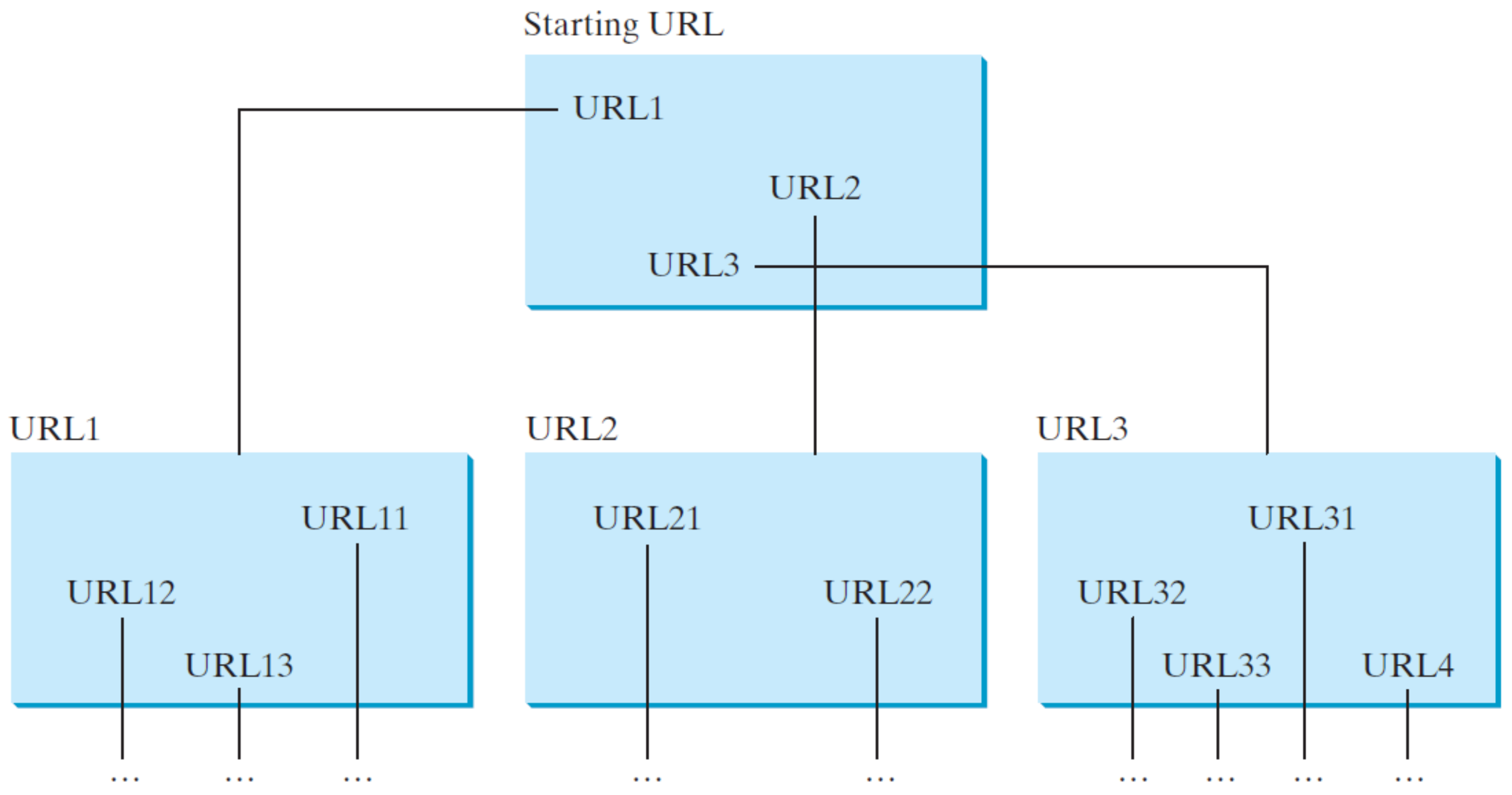
After a **URL** object is created, you can use the **openStream()** method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

```
Scanner input = new Scanner(url.openStream());
```

ReadFileFromURL Run

Case Study: Web Crawler

This case study develops a program that travels the Web by following hyperlinks.



Case Study: Web Crawler

The program follows the URLs to traverse the Web. To avoid that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

Case Study: Web Crawler

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty {
        Remove a URL from listOfPendingURLs;
        if this URL is not in listOfTraversedURLs {
          Add it to listOfTraversedURLs;
          Display this URL;
          Exit the while loop when the size of S is equal to 100.
          Read the page from this URL and for each URL contained in the page {
            Add it to listOfPendingURLs if it is not is listOfTraversedURLs;
          }
      }
 }
```

WebCrawler   Run