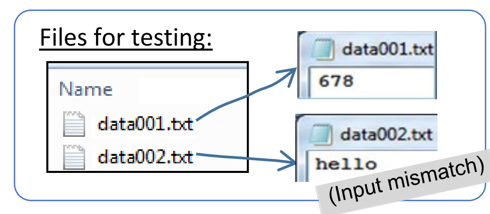
Contents **Topic 06 -Exception Handling**

I. Introductory Examples (Example 1-5)
II. Exception Handling – Basic Idea
III. Exception Hierarchy
IV. The Try-throw-catch Mechanism
V. Define our own exception classes (Example 6)
VI. Pitfall: Catch the more specific exception first (Example 7)
VII. The throws clause, throws clause in derived class (Example 8)
VIII. The Catch-or-Declare Rule, Checked Exceptions, Unchecked Exceptions
IX. The Finally Block (Example 6')

I Introductory Examples

Requirement: **Read one positive integer from a file**



Example 1 - Problems checked by Java Virtual Machine

```
public static void main(String[] args) throws FileNotFoundException
{
    Scanner in = new Scanner(System.in);
    System.out.print("Input the file pathname: ");
    String fname = in.next();

    Scanner inFile = new Scanner(new File(fname));   [Line 13]
    int x = inFile.nextInt();                        [Line 14]
    System.out.println("Data is: "+x);
    inFile.close();

    in.close();
}
```

```
Rundown 1.1:
Input the file pathname: c:\data001.txt
Data is: 678
```

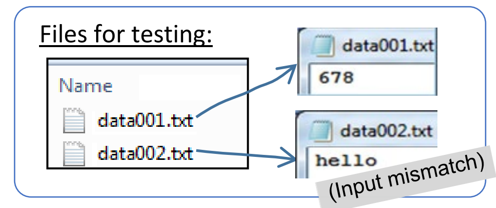
```
Rundown 1.2:
Input the file pathname: c:\data02.txt
Exception in thread "main" java.io.FileNotFoundException
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(Unknown Source)
    at java.util.Scanner.<init>(Unknown Source)
    at Main.main(Main.java:13)
```

```
Rundown 1.3:
Input the file pathname: c:\data002.txt
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at Main.main(Main.java:14)
```

- Problems are checked by **Java Virtual Machine**
- java.io.FileNotFoundException and java.util.InputMismatchException are JAVA classes, each means a type of exception.
- JVM outputs the message by calling:

  `public void printStackTrace()`

  [a method of the `Java.lang.Throwable` class]



Example 2 - We check and handle the problems ourselves

Below shows how that can be done, using try-catch blocks

Program:

```
public static void main(String[] args)
{
    try
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Input the file pathname: ");
        String fname = in.next();
        Scanner inFile = new Scanner(new File(fname));
        int x = inFile.nextInt();
        System.out.println("Data is: "+x);

        inFile.close();
        in.close();
    }
    catch (FileNotFoundException e)
    {
        System.out.println("Cannot open the file. Please check or ask CS2312 helpers.");
    }
    catch (InputMismatchException e)
    {
        System.out.println("Cannot read the required number from the opened file. "+
          " Please download from Helena's website again.");
    }
}
```

- A Try block tells what to do when everything goes smoothly.
- A catch block handles a kind of caught problems (exceptions)

`catch (InputMismatchException e)`

Declare the exception type (Data type) and exception object variable (other names are ok, but "e" is often used)

Testing:

```
Rundown 2.1:
Input the file pathname: c:\data001.txt
Data is: 678
```

```
Rundown 2.2:
Input the file pathname: c:\data02.txt
Cannot open the file.  Please check or ask
CS2312 helpers.
```

```
Rundown 2.3:
Input the file pathname: c:\data002.txt
Cannot read the required number from the
opened file.  Please download from Helena's
website again.
```

```
try
{
  ..
}
catch (FileNotFoundException e)
{
  System.out.println("Cannot open ..");
}
catch (InputMismatchException e)
{
  System.out.println("Cannot read ..");
}
```

We write code to check and take action (here simply output the situation)

A common case: we handle more processing:

1) Put the actions in one method. The actions may have problems.
2) Place the *try-catch* block in the caller. The caller will handle the problems.

Example 3

```
public static void processFile(String fname) throws FileNotFoundException, InputMismatchException
{
   Scanner inFile = new Scanner(new File(fname));
   int x = inFile.nextInt();
   System.out.println("Data is: "+x);
   inFile.close();
}

public static void main(String[] args)
{
   try
   {
       Scanner in = new Scanner(System.in);
       System.out.print("Input the file pathname: ");
       String fname = in.next();
       processFile(fname);
       in.close();
   }
   catch (FileNotFoundException e)
   {
       System.out.println("Cannot open the file.  Please check or ask CS2312 helpers.");
   }
   catch (InputMismatchException e)
   {
       System.out.println("Cannot read the required number from the opened file.  Please download from Helena's website again.");
   }
}
```

The throws clause – declare what exceptions might occur.

Rundown: [Same as 2.1, 2.2, 2.3 in Example 2]

We can write an Exception Controlled Loops
- Let the user get things right on a subsequent try

Example 4

```
public static void processFile(String fname) throws FileNotFoundException, InputMismatchException
{
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    System.out.println("Data is: "+x);
    inFile.close();
}

public static void main(String[] args)
{
    Scanner in = new Scanner(System.in);
    boolean shouldEnd=false;
    while (!shouldEnd)
    {
       try
       {
          System.out.print("Input the file pathname: ");
          String fname = in.next();
          processFile(fname);
          shouldEnd=true;
       }
       catch (FileNotFoundException e)
       {
          System.out.println("Cannot open the file.");
          System.out.print("Try another file?  Type your choice [y/n]: ");
          shouldEnd=(in.next().charAt(0)=='n');
       }
       catch (InputMismatchException e)
       {
          System.out.println("Cannot read the required
          System.out.print("Try another file?  Type yo
          shouldEnd=(in.next().charAt(0)=='n');
       }
    }
    in.close();
}
```

Rundown 4.1:
Input the file pathname: c:\data002.txt
Cannot read the required number from the opened file.
Try another file? Type your choice [y/n]: y
Input the file pathname: c:\data02.txt
Cannot open the file.
Try another file? Type your choice [y/n]: y
Input the file pathname: c:\data001.txt
Data is: 678

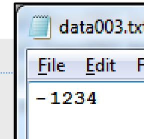
Rundown 4.2:
Input the file pathname: c:\data002.txt
Cannot read the required number from the opened file.
Try another file? Type your choice [y/n]: y
Input the file pathname: c:\data02.txt
Cannot open the file.
Try another file? Type your choice [y/n]: n

Create exception class + throw an exception object of the kind

- Suppose we expect a non-negative integer, however the file contains a –ve integer.
  This is what we want:

If the file content is –ve, the output should tell:

```
Rundown 5.1:
Input the file pathname: c:\data003.txt
Unexpected negative value from the file.
Try another file?  Type your choice [y/n]: n
```



- This problem is not described by JAVA standard exception classes.
  Therefore we create and use our own exception class.

Example 5

```
public class NegativeIntegerException extends Exception
{
      public NegativeIntegerException()
      {
            super("Negative integer!");
      }
      public NegativeIntegerException(String message)
      {
            super(message);
      }
}
```

It is customary to give both a default constructor and a constructor that contains a detailed message.

processFile():

```
public static void processFile(String fname) throws FileNotFoundException,
            InputMismatchException, NegativeIntegerException
{
      Scanner inFile = new Scanner(new File(fname));   // May throw FileNotFoundException

      int x = inFile.nextInt();                         // May throw InputMismatchException
      if (x<0) {
            throw new NegativeIntegerException();       // throw NegativeIntegerException
      }

      System.out.println("Data is: "+x);
      inFile.close();
}
```

Inside **main():**

```
try
{
      System.out.print("Input the file pathname: ");
      String fname = in.next();
      processFile(fname);
      shouldEnd=true;
}
catch (FileNotFoundException e)
{
      ...
}
catch (InputMismatchException e)
{
      ...
}
catch (NegativeIntegerException e)
{
      System.out.println("Unexpected negative value from the file.");
      System.out.print("Try another file?  Type your choice [y/n]: ");
      shouldEnd=(in.next().charAt(0)=='n');
}
```

II Exception Handling – Basic Idea

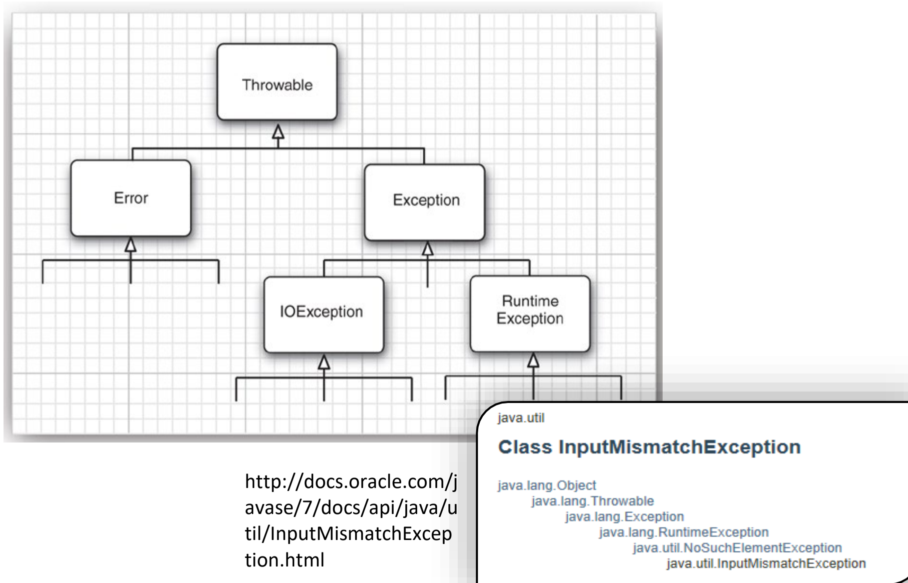
- **Possible errors of a running JAVA program**
  1) User input error (e.g. typing mistake, wrong format of URL)
  2) Device errors (e.g. printer suddenly jammed, webpage temporarily unavailable)
  3) Physical limitation (e.g. disk full, out of memory)
  4) Code error (e.g. invalid array index)

- **Error handling**
  1) [Traditional approach I] Method returns -1, special end-of-file value, etc..
  2) [Traditional approach II] Method returns a null reference.
  3) **JAVA's exception handling:**

     [Mission] **Transfer control from the location where error occurred to an error handler that can deal with the situation**

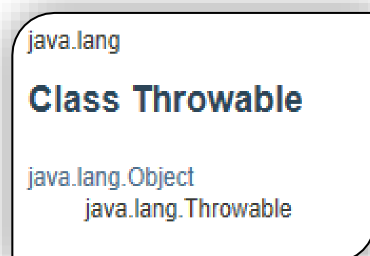
     - throws an object that encapsulates the error information
     - the code section exits immediately
     - it does not return any value (even the method return type is not void)
     - Search for an exception handler that can deal with this particular error.

III Exception Hierarchy in JAVA



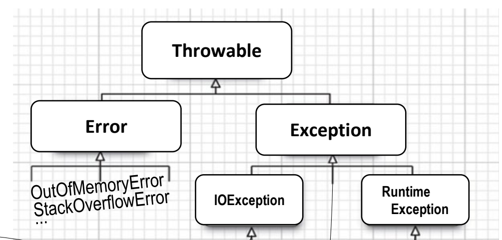
http://docs.oracle.com/javase/7/docs/api/java/util/InputMismatchException.html

The Topmost Class: Throwable



http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

- **`Throwable`** is the superclass of all errors and exceptions in JAVA,
  - Thrown by JVM, e.g.
    `FileNotFoundException`
    `InputMismatchException`
  - Thrown in our code, e.g.
    `NegativeIntegerException` (we defined in Example 5)

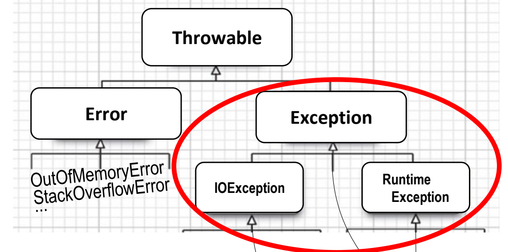


The Exception Hierarchy:

[http://docs.oracle.com/javase/7/docs/api/java/lang/Error.html]

java.lang
Class Exception
java.lang.Object
  java.lang.Throwable
    java.lang.Exception

- The **Exception** hierarchy indicates conditions that a reasonable application might want to catch. (Ref. Examples 1-5)



Example 5 [recalled]

```
public static void processFile(String fname) throws ..
{
   Scanner inFile = new Scanner(new File(fname));
   int x = inFile.nextInt();
   if (x<0)
      throw new NegativeIntegerException();
   System.out.println("Data is: "+x);
   inFile.close();
}
```

JVM may throw FileNotFoundException
JVM may throw InputMismatchException
throw NegativeIntegerException

```
public static void main(String[] args)
{
  ..
  try {
         ...
         processFile(fname);
         ...
  }
  catch (FileNotFoundException e) {..}
  catch (InputMismatchException e) {..}
  catch (NegativeIntegerException e) {..}
  ...
}
```

Catch and handle

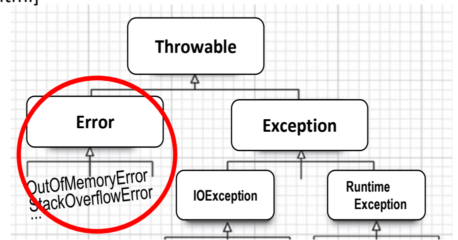
The Error Hierarchy:

[http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html]

java.lang
Class Error
java.lang.Object
  java.lang.Throwable
    java.lang.Error

- The **Error** hierarchy describes internal errors and resource exhaustion situations which are abnormal.

  Indicates serious problems that a reasonable application should NOT try to catch. *(But if you want, you can!!)*



It should be aborted and the system (not our code) **need to take over the control.**
St
Note: Java programmers often call "Errors" as "Exceptions" as well. We speak in the same way in this topic.

Examples:
java.lang.OutOfMemoryError (Previous lecture exercise: out of heap space)

```
Object[] arr1 = new Object[10000000];
Object[] arr = arr1;
for (int i=0;i<200;i++) {
      arr[0]=new Object[10000000];
      arr=(Object[])arr[0];
}
```

Program aborted by JAVA runtime

Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
at Main.main(Main.java:12)

java.lang
Class StackOverflowError
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.StackOverflowError

java.lang.StackOverflowError (Previous lecture exercise: recursion cannot stop – Stack overflow)

```
private static int factorial(int n) {
   if (n==1) return 1;
   else return n*factorial(n+1);
}

public static void main(String[] args) {
   System.out.println(factorial(4));
}
```

Program aborted by JAVA runtime

Exception in thread "main"
java.lang.StackOverflowError
at MainStackError.factorial(MainStackError.java:5)
at MainStackError.factorial(MainStackError.java:6)
at MainStackError.factorial(MainStackError.java:6)
at MainStackError.factorial(MainStackError.java:6)
...

java.lang
Class StackOverflowError
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.StackOverflowError

Catch an error and handle it, continue running.
[For illustration purpose only; recall: "a reasonable application should NOT try to catch"]


```
private static int factorial1(int n)
{
    if (n==1)      return 1;
    else     return n*factorial1(n+1);
}

private static int factorial2(int n)
{
    if (n==1)  return 1;
    else     return n*factorial2(n-1);
}
```

```
public static void main(String[] args)
{
    Scanner in = new Scanner(System.in);
    try
    {
        System.out.println(factorial1(4)); //failed
    }
    catch (StackOverflowError e)
    {
        System.out.printf("Error happens when factorial1 runs\n\n");
        System.out.print("Print stack? Type your choice [y/n]: ");
        if (in.next().charAt(0)=='y')
            e.printStackTrace();

        System.out.print("\nTry another one?  Type your choice [y/n]: ");
        if (in.next().charAt(0)=='y')
            System.out.println(factorial2(4)); //OK: 24
    }
    System.out.println("\nFinished");
}
```

We catch the error and handle it,

make the program continue to run

However, practically it should be aborted and the system (not our code) need to take over the control.

```
Error happens when factorial1 runs

Print stack? Type your choice [y/n]: y
java.lang.StackOverflowError
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:7)
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:8)
  ...
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:8)

Try another one?  Type your choice [y/n]: y
24

Finished
```

IV The Try-throw-catch Mechanism

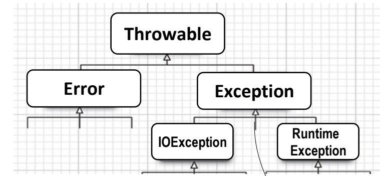
The Try-throw-catch Mechanism:

- The **try** block
  - Tells what to do when everything goes smooth
  - Can have code that throws exceptions for unusual conditions
  - If something goes wrong within the try block, execution in the try block is stoped and an exception is thrown
- The **throw** statement: We can write the throw statement to (create exception object and) throw an exception.
  - The similar is done when JAVA detects problem (e.g., JAVA may throw an object of **StackOverflowError**). **FileNotFoundException, InputMismatchException, etc..**
  - Syntax

    ```
    throw new ExceptionClassName(arguments);
    ```

    exception object is created and thrown
  - After exception is thrown, the flow of control is transferred to a catch block, and the catch block begins to run
- The **catch** block
  - Tells what to do when an exception is caught
  - One parameter, usually named "e" (but other names are also ok)
  - Syntax

    ```
    catch (ExceptionClassName e)
    {
            ..code to handle the exception
    }
    ```



V Define our own exception classes

- Predefined exception class contains:
  - An object field to store a message and a constructor that sets the message
  - An accessor method, `String getMessage()`
- We define our own exception class by inheriting `Exception` (or other throwables)
  - We can add fields and methods in our class
    E.g. `int problemValue`, `int getProblemValue()`
  - Constructors to be implemented:

```
public class NegativeIntegerException extends Exception
{
    private int problemValue;
    public int getProblemValue() { return problemValue; }

    public NegativeIntegerException() { super("Negative integer!"); }
    public NegativeIntegerException(String message) { super(message); }

    public NegativeIntegerException(String message, int v)
    {
        super(message); problemValue=v;
    }
}
```

It is customary to give both a default constructor and a constructor that contains a detailed message.

Add constructor according to our design

  - Which constructor will run is decided by how we create the exception object:

```
throw new NegativeIntegerException();
throw new NegativeIntegerException("-ve number");
throw new NegativeIntegerException("-ve number", x);
```

[Complete Code]

Example 6 Revised NegativeIntegerException: Add a field to store the problem value

NegativeIntegerException:

```
public class NegativeIntegerException extends Exception
{
    private int problemValue;
    public int getProblemValue() {return problemValue;}
    public NegativeIntegerException() {super("Negative integer!"); }
    public NegativeIntegerException(String msg) {super(msg); }

    public NegativeIntegerException(String msg, int v)
    {
        super(msg); problemValue=v;
    }
}
```

Inside main():

```
try
{
    Scanner inFile = new Scanner(new File(fname));     Throw?
    int x = inFile.nextInt();                           Throw?
    if (x<0)
        throw new NegativeIntegerException("-ve number", x);   Throw!

    System.out.println("Data is: "+x);
   inFile.close();
}
catch (FileNotFoundException e) {
    System.out.println("Cannot open ...");
}
catch (InputMismatchException e) {
    System.out.println("Cannot read ...");
}
catch (NegativeIntegerException e) {
   System.out.println(e.getMessage()+
        " ["+e.getProblemValue()+"]");
}
```

Rundown 6.1:
Input the file pathname: c:\data003.txt
-ve number [-1234]

VI Pitfall: Catch the more specific exception first

- When we have 2 or more catch-blocks, catch the more specific exception first

Example 7 [based on Example 5]

```
public static void processFile(String fname) throws
FileNotFoundException, InputMismatchException, NegativeIntegerException
 {
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    if (x<0)
        throw new NegativeIntegerException();
    System.out.println("Data is: "+x);
    inFile.close();
 }

 public static void main(String[] args)
 {
    ..
    try {
        ..
        processFile(fname);
    }
    catch (NegativeIntegerException e)
    {
        System.out.println("Unexpected negative value from the file.");
    }
    catch (Exception e)
    {
        System.out.println("Some problem happens.");
    }
    in.close();
 }
```

If you exchange their order, compiler complains:
"Unreachable catch block for NegativeIntegerException. It is already handled by the catch block for Exception"

VII The throws clause, throws clause in derived class

The Throws clause

- When a method may cause exception, but no catch is done within the method,
  Then it may inform the user by using the Throws clause to declare the exception
- The user may catch and handle the exception (or the user itself declares it again).

Example 5 [recall]

```
public static void processFile(String fname) throws FileNotFoundException,
        InputMismatchException, NegativeIntegerException
{
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    if (x<0)
        throw new NegativeIntegerException();
    System.out.println("Data is: "+x);
    inFile.close();
}
```

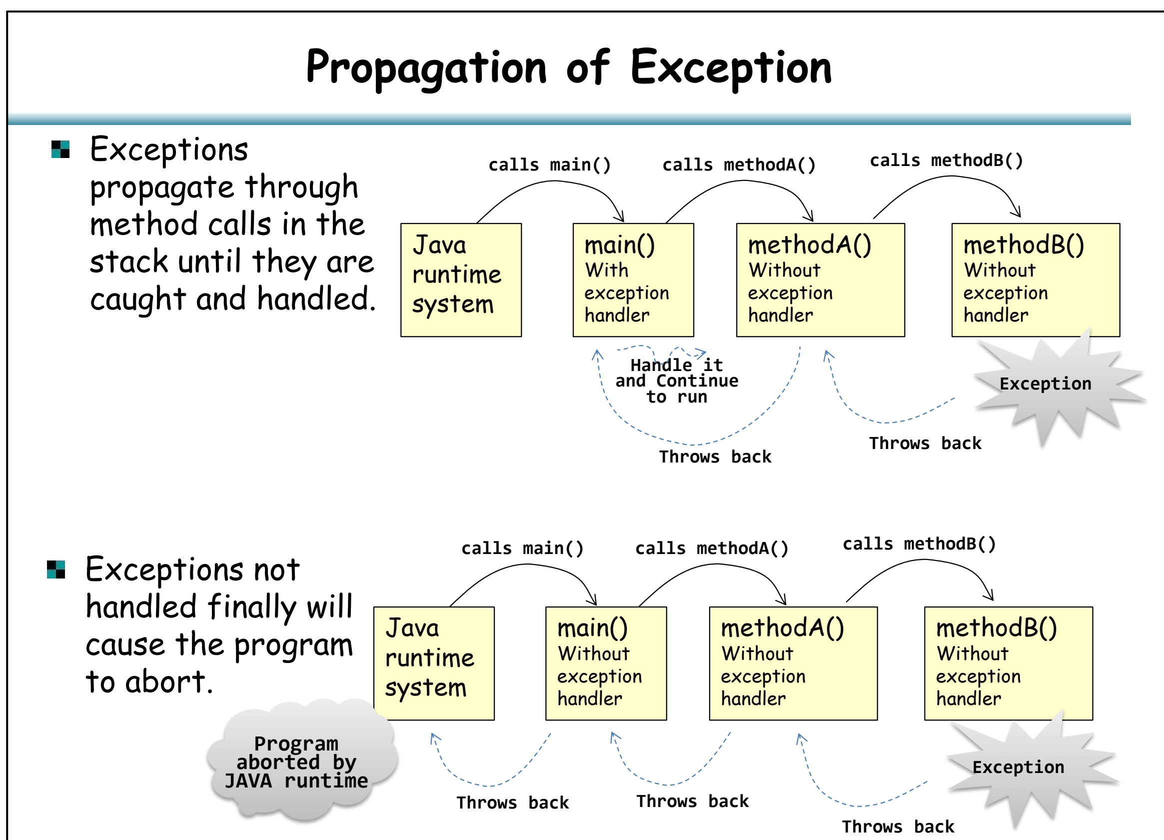
Declaring the exception
The throws clause declares what exceptions might occur.

Throwing an exception

Inside main():

```
public static void main(String[] args)
{
  ..
  try {
        ...
        processFile(fname);
        ...
  }
  catch (FileNotFoundException e) {..}
  catch (InputMismatchException e) {..}
  catch (NegativeIntegerException e) {..}
  ...
}
```

Handling an exception



Throws clause in derived classes

- When we redefine a method in a subclass, it should have the same exception classes listed in its throws clause that it had in the superclass
- Or it should have a subset of them
- i.e., a subclass may not add any exceptions to the throws clause, but it can delete some

Example 8

```
class ClassY
{
  public void processFile(String fname) throws FileNotFoundException
  {
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    System.out.println("Data is: "+x);
    inFile.close();
  }
}

class ClassZ extends ClassY
{
  public void processFile(String fname) throws FileNotFoundException, NegativeIntegerException
  {
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    if (x<0)
        throw new NegativeIntegerException();
    System.out.println("Data is: "+x);
    inFile.close();
  }
}
```

Exception NegativeIntegerException is not compatible with throws clause in ClassY.processFile(String)

VIII The Catch-or-Declare Rule, Checked Exceptions, Unchecked Exceptions

The Catch-or-Declare Rule

- If an exception may be thrown inside a method,

  the method may deal with it by

  1. placing the concerned code within a try-block, and handle in a catch-block,

  or

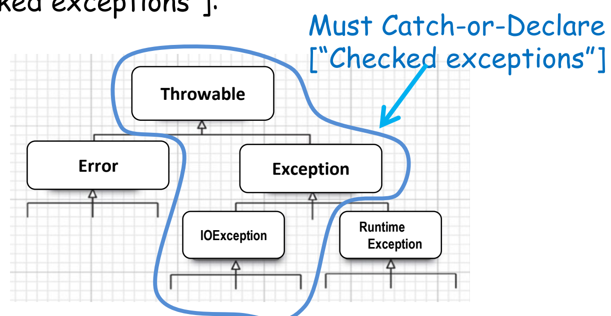
  2. declaring it

```
public void methodX()
{
  ..
  try {
        ... //code which may throw exception
  }
  catch (.. e) {
        ... //catch and handle the exception
  }
}
```

```
public void methodX() throws ..
{
    ..
}
```

- The compiler reinforces the Catch-or-Declare Rule on the following exceptions [Known as "checked exceptions"]:

  For other exceptions/errors, it is okay whether we deal with them or not. [Known as "unchecked exceptions"]



IX The Finally Block

The Finally Block

- The finally block contains code to be executed whether or not an exception is thrown in a try block.

Example 6'

```
public static void main(String[] args)
 {
    Scanner in = new Scanner(System.in);
    System.out.print("Input the file pathname: ");
    String fname = in.next();

    Scanner inFile=null;
    try
    {
        inFile = new Scanner(new File(fname));
        ..
    }
    catch (FileNotFoundException e)
    {
        System.out.println("Cannot open the file.");
    }
    catch (InputMismatchException e)
    {
        System.out.println("Cannot read the required number.");
    }
    catch (NegativeIntegerException e)
    {
        System.out.println(e.getMessage()+" ["+e.getProblemValue()+"]");
    }
    finally {
        if (inFile != null) {
            System.out.println("Closing inFile");
            inFile.close();
        } else {
            System.out.println("inFile not open");
        }
        ..
        in.close();
    }
 }
```

--- end ---