Exception Handling

B.Tech. (IT), Sem-5,
Core Java Technology (CJT)

Dharmsinh Desai University
Prof. (Dr.) H B Prajapati

Updated: 29 August 2020

Error handling in other languages

- Error handling is done manually
- Check return code (error code) of a function
- Find out error description of the matching error code
- Troubleshoot the problem

Exception handling in Java

- Error condition (abnormal condition) is handled by Java Runtime System.
- Exception means
  - Error condition
  - Abnormal condition
- Exception occurs due to the occurrence of unexpected behavior in certain statements
- Java provides run-time error management in the program.

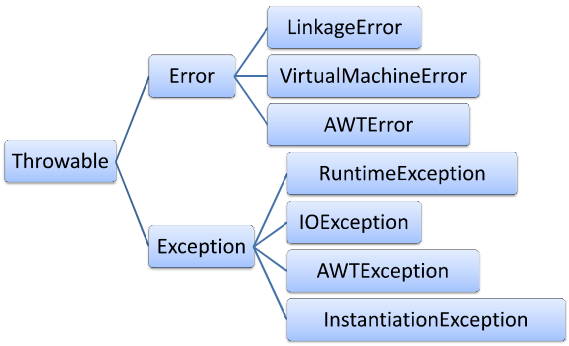
Causes of Exception

- Exceptions can arise due to a number of situations. For example,
  - Accessing array element beyond its size. E.g., trying to access the 11th element of an array when the array contains only 10 element. (ArrayIndexOutOfBoundsException)
  - Division by zero (ArithmeticException)
  - Accessing a file which is not present (FileNotFoundException)
  - Failure of I/O operations (IOException)
  - Illegal usage of null. (NullPointerException)
  - Assigning value of invalid data type to an array (ArrayStoreException)

What is Java Exception?

- A Java Exception is an object that describes exceptional (error) condition that has occurred in our code.
- General working of exception
  - When an exception condition arises:
    - An object representing that error condition is created
    - This exception object is thrown in the method that caused the error.
    - That method may choose to handle that exception or may pass the exception to its caller.
    - This process continues until the exception is handled by the Java system.
    - The default handler prints the name of the exception along with an explanatory message followed by stack trace at the time the exception was thrown and the program is terminated.

Exception hierarchy



Exception classes

- Throwable is the base class of all exception classes. It is kept in java.lang package. All other exception classes are contained in various packages.
- Error class describes internal system errors, which rarely occur.
  - We can only notify about this error, we can't take any action
- Subclasses of LinkageError indicate that a class has some dependency on another class; however, the another class has incompatibly changed after the compilation of the dependent class.
- Subclasses of VirtualMachineError indicate that the JVM is broken or running out of resources.
- AWTError is caused by a fatal error in AWT based programs

Exception classes

- Exception class indicates error caused by our programs and external circumstances. Such errors can be caught, as opposed to Error, and can be handled.
- RuntimeException: indicates programming errors
  - Bad casting
  - Accessing out-of-bound array
  - Numeric conversion errors
- Subclasses of RuntimeException
  - ArithmeticException
  - NullPointerException
  - IllegalArgumentException
  - ArrayStoreException
  - IndexOutOfBoundException

Exception classes

- The IOException class describes errors related to input/output operations:
  - Invalid input (InterruptedIOException)
  - Reading past the end of a file (EOFException)
  - Opening a nonexistent file (FileNotFoundException)
- AWTException class describes errors caused by AWT operations.

Uncaught exception

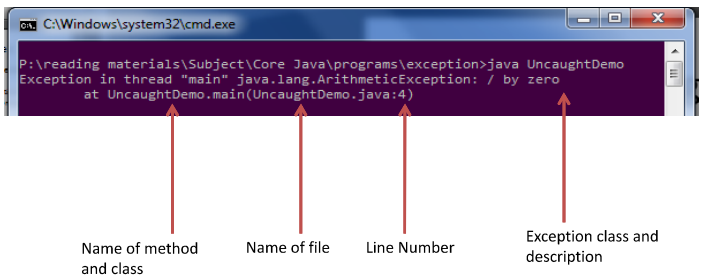
```
class UncaughtDemo{
    public static void main(String[] args){
        int d=0;
        int r=50/d;
    }
}
```

UncaughtDemo.java

```
1  class UncaughtDemo{
2      public static void main(String[] args){
3          int d=0;
4          int r=50/d;
5      }
6  }
```


Uncaught exception

Running the program



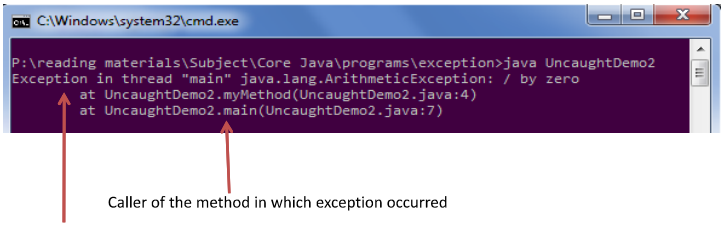
Uncaught exception

UncaughtDemo2.java

```
1  class UncaughtDemo2{
2      static void myMethod(){
3          int d=0;
4          int r=50/d;
5      }
6      public static void main(String[] args){
7          myMethod();
8      }
9  }
```

Uncaught exception

- Running the program
  - The default exception handler shows entire stack trace



Caller of the method in which exception occurred

Position of error: class name and method name (Filename: Line Number)

Exception handling

- Exception handling is managed using five keywords
  - try
  - catch
  - throw
  - throws
  - finally

Use of try-catch

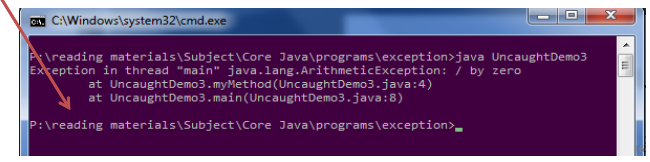
- Format

```
try{
   //statements that could generate error condition
}
catch(Exception e){
   // print exception information
   // take necessary action, if any
}
```


Without use of try-catch

UncaughtDemo3.java

```
1  class UncaughtDemo3{
2      void myMethod(){
3          int d=0;
4          int r=50/d;
5      }
6      public static void main(String[] args){
7          UncaughtDemo3 ob=new UncaughtDemo3();
8          ob.myMethod();
9          System.out.println("After myMethod() call");
10     }
11 }
```



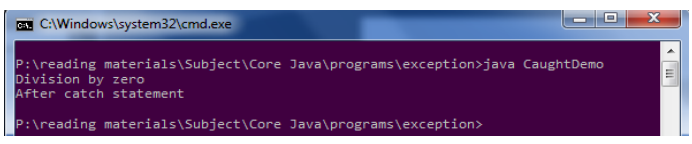
Using try-catch

CaughtDemo.java

```
1  class CaughtDemo{
2      void myMethod(){
3          int d=0;
4          int r=50/d;
5          System.out.println("About to exit from myMethod()");
6      }
7      public static void main(String[] args){
8          UncaughtDemo3 ob=new UncaughtDemo3();
9          try{
10             ob.myMethod();
11             System.out.println("After myMethod() call");
12         }
13         catch(ArithmeticException e){
14             System.out.println("Division by zero");
15         }
16         System.out.println("After catch statement");
17     }
18 }
```


Using try-catch

- Running the program



- Any statement inside try block, following erroneous statement does not get executed
  - i.e., we do not get "After myMethod() call" in output
- But, statements after catch block do get executed.

Multiple catch clauses

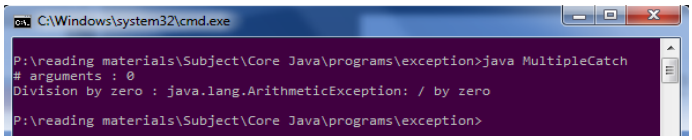
- Passing exception object to println method causes call to its **toString()**

MultipleCatch.java

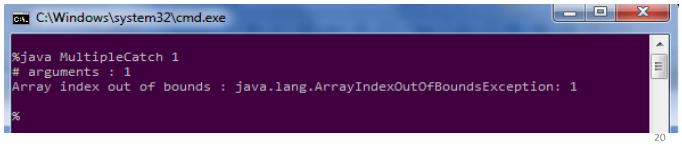
```
1  class MultipleCatch{
2      public static void main(String[] args){
3          try{
4              int ac=args.length;
5              System.out.println("# arguments : "+ac);
6              int r=50/ac;
7              int[] arr={1};
8              arr[ac]=100;
9          }
10         catch(ArithmeticException e){
11             System.out.println("Division by zero : "+e);
12         }
13         catch(Exception e){
14             System.out.println("Array index out of bounds : "+e);
15         }
16     }
17 }
```


Multiple catch clauses

- Running the program: without argument



- Running the program: with argument



Rule for using multiple catch clauses

- When we use multiple catch statements, the exception subclasses must come before any of their super-classes
- A catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus any subsequent catch clauses are never reached – called unreachable code.
- In Java, unreachable code is error. Therefore, such program would not get compiled.

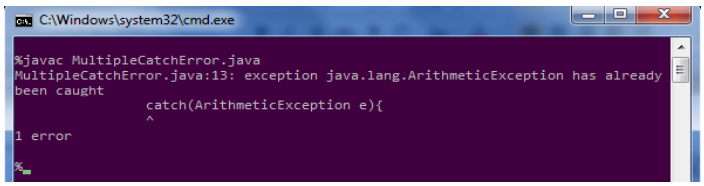
Superclass exception as a first catch

MultipleCatchError.java

```
1  class MultipleCatchError{
2      public static void main(String[] args){
3          try{
4              int ac=args.length;
5              System.out.println("# arguments : "+ac);
6              int r=50/ac;
7              int[] arr={1};
8              arr[ac]=100;
9          }
10         catch(Exception e){
11             System.out.println("Array index out of bound : "+e);
12         }
13         catch(ArithmeticException e){
14             System.out.println("Division by zero : "+e);
15         }
16     }
17 }
```


Superclass exception as a first catch

- The program will not get compiled



Nested try statements

- A try statement can be inside the block of another try
- Working
  - Each time a try statement is entered, the context of that exception is pushed on the stack.
  - If inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's (outer one) catch handlers are inspected for a match.
  - This continues until match for catch succeeds, or all of the nested try statements are exhausted.
  - If no matching catch is found, the Java runtime system will handle the exception.

Nested try

```
class NestedTry{
    public static void main(String[] args){
        try{
            int ac=args.length;
            System.out.println("# arguments : "+ac);
            int r=50/ac;
            try{
                if(ac==1)
                    r=50/(ac-1);
                if(ac==2){
```


Nested try

```
                    int[] arr={1};
                    arr[ac]=100;
                }
            }
            catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Array Index out-of bounds :
"+e);
            }
        }
        catch(ArithmeticException e){
            System.out.println("Division by zero : "+e);
        }
```


Nested try

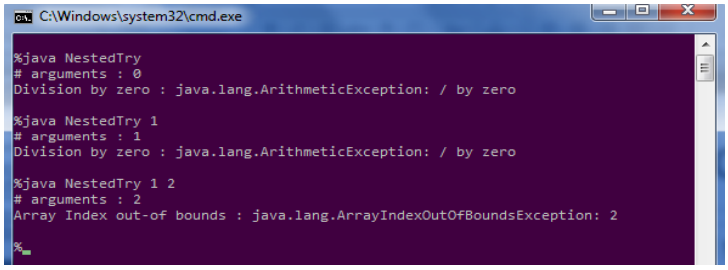
```
    }
}
```


Nested try (with line numbers)

NestedTry.java

```
1  class NestedTry{
2      public static void main(String[] args){
3          try{
4              int ac=args.length;
5              System.out.println("# arguments : "+ac);
6              int r=50/ac;
7              try{
8                  if(ac==1)
9                      r=50/(ac-1);
10                 if(ac==2){
11                     int[] arr={1};
12                     arr[ac]=100;
13                 }
14             }
15             catch(ArrayIndexOutOfBoundsException e){
16                 System.out.println("Array Index out-of bounds : "+e);
17             }
18         }
19         catch(ArithmeticException e){
20             System.out.println("Division by zero : "+e);
21         }
22     }
23 }
```


Running the program: Nested try



Ways to create Java Exception

- Two ways:
  - Exception gets created automatically , due to some operation
    - Done by Java runtime system
  - We can manually create exception and can throw it.
    - Programmers write code, uses throw clause

Exception handling mechanism

- Claiming an exception
- Throwing an exception
- Catching an exception

Exception handling mechanism

- Claiming an exception
  - It tell to the compiler that during execution of this function, something wrong can happen.
    - public void myMethod() throws IOException
  - The caller of such method has to use either try-catch or needs to pass exception to its caller by specifying throws
- Throwing an exception
  - The method that declared an exception can throw exception object
    - throw new ExceptionClass();
- Catching an exception
  - Catching of thrown exception is done using catch statements.

Checked exception and unchecked exception

- Unchecked exceptions
  - These exceptions are not required to be declared or claimed using *throws clause*.
  - Error, RuntimeException, and subclasses are unchecked exceptions.
  - Examples: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, ClassCastException
- Checked exceptions
  - For all other exceptions, method needs to claim/declare exception using *throws clause*.
    - Return-type method-name(parameter-list) *throws exception-list*
  - Examples: ClassNotFoundException, IOException, InterruptedException

Checked Unchecked exception: example

```
import java.io.IOException;
class CheckUncheckExceptionDemo{
    public static void uncheckMethod(){
        System.out.println("> Inside uncheckMethod()");
        throw new RuntimeException("Demo: Runtime error");
    }
    public static void checkMethod() throws IOException{
        System.out.println("> Inside checkMethod()");
        throw new IOException("Demo: IOException");
    }
```

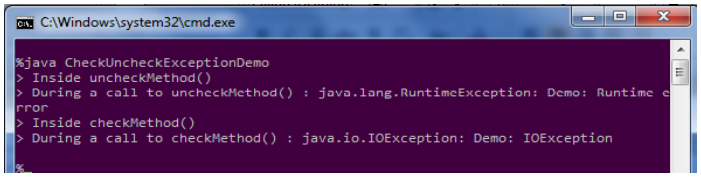

Checked Unchecked exception: example (cont…)

```
public static void main(String[] args){
    try{
        uncheckMethod();
    }
    catch(RuntimeException e){
        System.out.println("> During a call to uncheckMethod()
: "+e);
    }
```


Checked Unchecked exception: example (cont…)

```
    try{
        checkMethod();
    }
    catch(IOException e){
        System.out.println("> During a call to checkMethod() :
"+e);
    }
}
```

Running Checked Unchecked exception: example



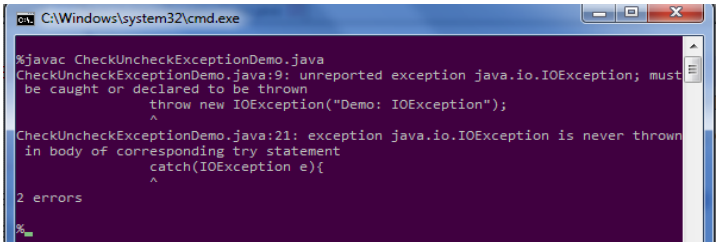
Throwing checked exception without declaration

- Throwing checked exception without declaration will cause compilation error.
- The following code is invalid

```
public static void checkMethod() {
    System.out.println("> Inside checkMethod()");
    throw new IOException("Demo: IOException");
}
```


Throwing checked exception without declaration

- Compiler will generate errors



Methods related to exception

- Some methods of Throwable class
  - public String getMessage()
    - Returns detailed description of the Throwable object
  - public String toString()
    - Returns short description of the Throwable object
  - public String getLocalizedMessage()
    - Returns a localized description of the Throwable object.
  - public void printStackTrace()
    - It prints the Throwable object and its trace information on the console.

Rethrowing exceptions

- If the caller of method wants to do some processing related to the thrown exception, the called method can rethrow the exception, even after the called method has done processing of its part

```
try{
    // statements;
}
catch(TheExceptionClass e){
    // perform operations before exit
    throw e;
}
```


The finally clause

- The finally clause is useful when a method is using resources (file, db connection, network connection) and the method returns prematurely.

```
try{
} catch(Throwable e){
}
finally{
}
```

- The finally block is executed in all circumstances, whether exception occurs or does not occur.
- The finally block will execute even if no matching catch statement is found.

The finally clause

- Any time a method is about to return to the caller, from inside a try-catch block, via an uncaught exception or explicit return statement, the finally clause is executed just before the method returns.
- The finally clause is not compulsory
- But, a try statement requires at least one catch or a finally clause.

Finally demo

```
class FinallyDemo{
    public static void throwExceptionMethod(){
        try{
            System.out.println("> Inside throwExceptionMethod()");
            throw new RuntimeException("Finally Demo: Runtime
error");
        }
        finally{
            System.out.println("> Finally of throwExceptionMethod()");
        }
    }
```


Finally demo (cont…)

```
public static void returnFromTryMethod() {
    try{
        System.out.println("> Inside returnFromTryMethod()");
        return;
    }
    finally{
        System.out.println("> Finally of
returnFromTryMethod()");
    }
}
```

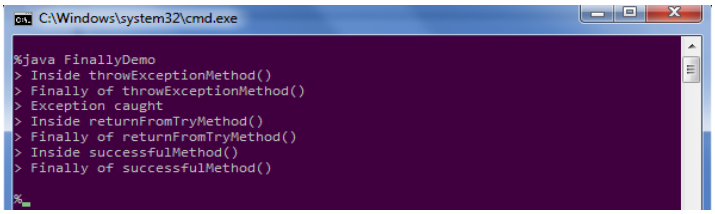

Finally demo (cont…)

```
public static void successfulMethod(){
    try{
        System.out.println("> Inside successfulMethod()");
    }
    finally{
        System.out.println("> Finally of successfulMethod()");
    }
}
```


Finally demo (cont…)

```
public static void main(String[] args){
    try{
        throwExceptionMethod();
    }
    catch(Exception e){
        System.out.println("> Exception caught");
    }
    returnFromTryMethod();
    successfulMethod();
}
}
```

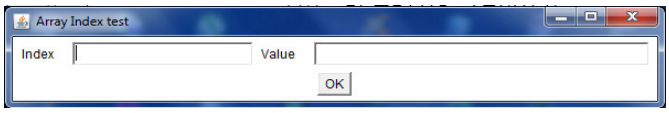

Running: Finally demo



Exception in GUI based program

- If exception of a subclass of Exception occurs in a GUI program?
  - Java prints the error message on the console (program is not terminated)
  - Program continues its user interface processing loop.

Test of ArrayIndexOutOfBoundsException



ArrayIndex: a GUI based application

```
import java.awt.*;
import java.awt.event.*;
class ArrayIndex extends Frame implements ActionListener{
    String[] list={"IT","CE","CL","EC"};
   TextField tf1=new TextField(20);
   TextField tf2=new TextField(40);
   Button b=new Button("OK");
```


ArrayIndex: a GUI based application (cont…)

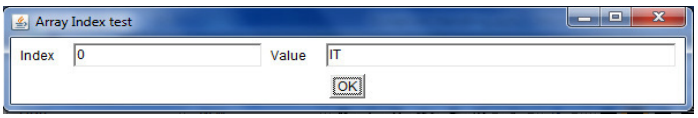
```
public ArrayIndex(String title){
   super(title);
   Panel p1=new Panel();       p1.add(new Label("Index"));
   p1.add(tf1);       p1.add(new Label("Value"));
   p1.add(tf2);       add("North",p1);
   Panel p2=new Panel();       p2.add(b);
   add("South",p2);
   b.addActionListener(this);
   setSize(600,100);
   setVisible(true);
}
```


ArrayIndex: a GUI based application (cont…)

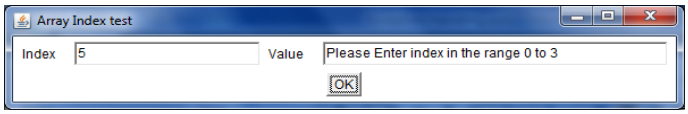
```
  public static void main(String[] args){
          new ArrayIndex("Array Index test");
   }
   public void actionPerformed(ActionEvent ae){
          int index=Integer.parseInt(tf1.getText().trim());
          try{
              String value=list[index];            tf2.setText(value);
          }
          catch(ArrayIndexOutOfBoundsException aie){
                  tf2.setText("Please Enter index in the range 0 to "+(list.length-1));
     }
   }
}
```


Running the program

- Valid index



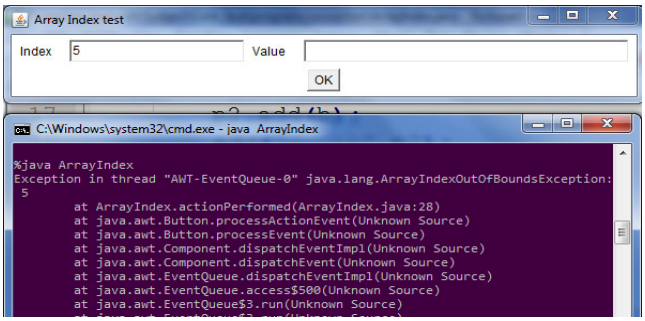
- Invalid index



Without exception handling

```
public void actionPerformed(ActionEvent ae){
        int index=Integer.parseInt(tf1.getText().trim());
        String value=list[index];
        tf2.setText(value);
}
```


Running the program



User defined exception classes

- In some situations, we would like to implement our own exception types to handle application specific situations
  - E.g., entering negative amount in withdraw operation.
  - Try to issue a book (in concurrent environment) that has just become unavailable
- We can do this by defining a subclass of Exception
- Exception class has no methods of its own, it derives from Throwable
- We are not required to implement any method in our own exception class. Their presence indicates that they are exception classes (anything derived from Exception)

Account operation using user defined exception

- We have Account object, which allows the following two operations
  - withdraw()
  - deposit()
- For withdraw() operation,
  - User may pass negative value
  - Sufficient balance is not available
- For deposit operation,
  - User may pass negative value

Account

```
class Account{
    private int ID;
    private double balance;
    Account(int ID,double balance){
        this.ID=ID;
        this.balance=balance;
    }
    public int getID(){
        return ID;
    }
    public double getBalance(){
        return balance;
    }
```


Account (cont…)

```
    public void withdraw(double amount)throws
    NegativeAmountException,
InsufficientFundException{
        if(amount<0)
                throw new
    NegativeAmountException(this,amount,"withdraw");
        if(balance<amount)
                throw new InsufficientFundException(this,amount);
        balance=balance-amount;
    }
```

Account (cont…)

```
    public void deposit(double amount)throws
    NegativeAmountException{
        if(amount<0)
                throw new
    NegativeAmountException(this,amount,"deposit");
        balance=balance+amount;
    }
    public String toString(){
        return "Account: ID="+ID+";Balance="+balance;
    }
}
```


NegativeAmountException

```
class NegativeAmountException extends Exception{
    private Account account;
    private double amount;
    private String transactionType;
    public NegativeAmountException(Account account,double
    amount,String tt){
    super("Negative amount");
        this.account=account;
        this.amount=amount;
        this.transactionType=tt;
    }
}
```


InsufficientFundException

```
class InsufficientFundException extends Exception{
    private Account account;
    private double amount;
    public InsufficientFundException(Account account,double amount){
        super("Insufficient balance");
        this.account=account;
        this.amount=amount;
    }
    public String toString(){
        return "InsufficientFundException: Available balance =
    "+account.getBalance()+" requested amount="+amount;
    }
}
```


TestAccount

```
public class TestAccount{
    public static void main(String[] args){
        Account ac=new Account(1,1000);
        System.out.println("Deposit -100 in "+ac);
        try{
                ac.deposit(-100);
        }
        catch(NegativeAmountException e){
                System.out.println(e);
        }
```


TestAccount (cont…)

```
        System.out.println("Widthdraw 1100 from "+ac);
        try{
                ac.withdraw(1100);
        }
        catch(NegativeAmountException e){
                System.out.println(e);
        }
        catch(InsufficientFundException e){
                System.out.println(e);
        }
    }
}
```


Running the TestAccount