SEECS

Object Oriented Programming

Lec: Exception Handling

Hirra Anwar

National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science
Islamabad

Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Introduction

- Exceptional event
- Error that occurs during runtime
- Cause normal program flow to be disrupted
- Examples
  - Divide by zero errors
  - Accessing the elements of an array beyond its range
  - Invalid input
  - Hard disk crash
  - Opening a non-existent file
  - Heap memory exhausted

Introduction : Java Exceptions

- An exception in Java is an object that is created when an abnormal situation arises in your program
- This object has members that stores information about the nature of the problem
- An Exception is always an object of some subclass of the standard class Throwable
- Java provides a very well defined hierarchy of Exceptions to deal with situations which are unusual.
- All standard exceptions are covered by two direct subclasses of the class Throwable
  - Class **Error**
  - Class **Exception**

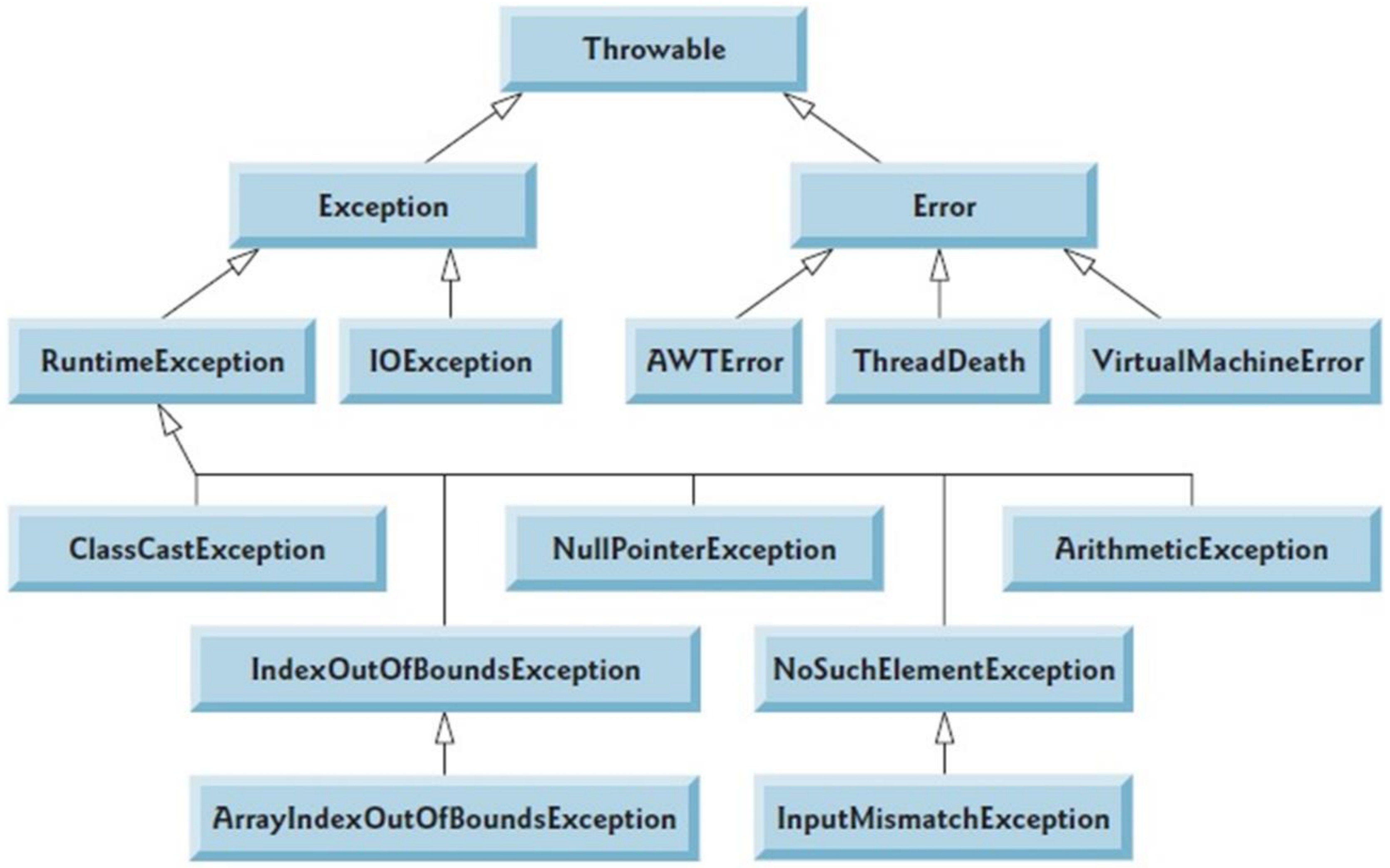
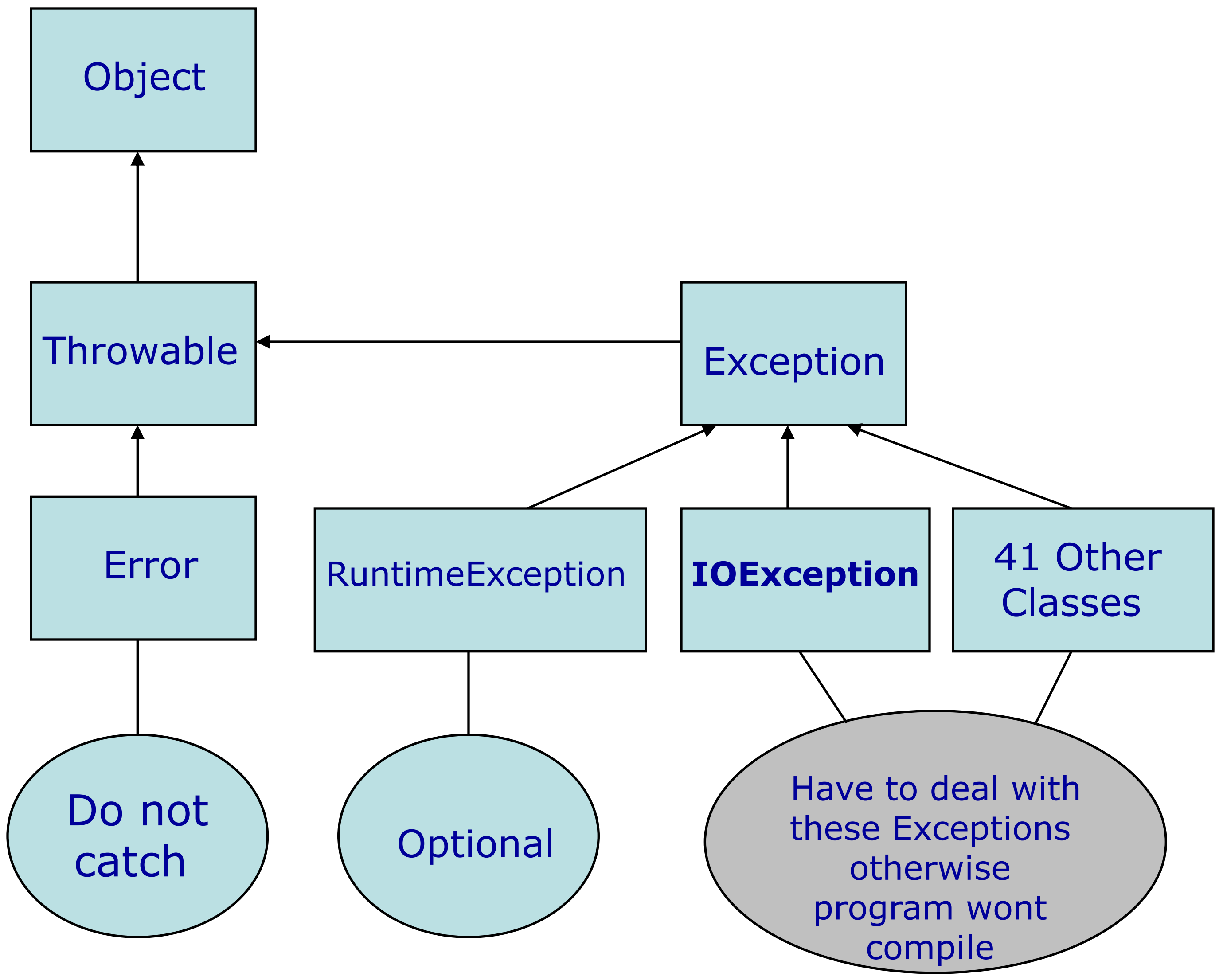


Fig. 11.3 | Portion of class `Throwable`'s inheritance hierarchy.

Introduction : Java Exceptions



Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Difference between Error and Exceptions

- Error class
  - Used by the Java run-time system to handle errors occurring in the run-time environment
  - Generally beyond the control of user programs
  - Examples
    - Out of memory errors
    - Hard disk crash
- Exception class
  - Conditions that user programs can reasonably deal with
  - Usually the result of some flaws in the user program code
  - Examples
    - Division by zero error
    - Array out-of-bounds error

Exception Example

```
class DivByZero {
 public static void main(String args[]) {
     System.out.println(3/0);
     System.out.println(“Pls. print me.”);
}
```

- **Exception in thread "main“ java.lang.ArithmeticException: \ by zero at DivByZero.main(DivByZero.java:3)**
- Default exception handler
  - Provided by Java runtime
  - Prints out exception description
  - Prints the stack trace
  - Causes the program to terminate

- Two categories of exceptions: checked and unchecked
- Checked exceptions
  - Exceptions that inherit from class `Exception` but not from `RuntimeException`
  - Compiler enforces a **catch-or-declare** requirement
  - Compiler checks each method call and method declaration to determine whether the method `throws` checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a `throws` clause. If not caught or declared, compiler error occurs.
- Unchecked exceptions
  - Inherit from class `RuntimeException` or class `Error`
  - Compiler does not check code to see if exception is caught or declared
  - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
  - Can typically be prevented by proper coding

- It is a syntax error to place code between a try block and its corresponding catch blocks.

- It is a compilation error to catch the same type in two different catch blocks in a single try statement.

Termination Model of Exception Handling

- When an exception occurs:
  - `try` block terminates immediately
  - Program control transfers to the first matching `catch` block

- After exception is handled:
  - Termination model of exception handling – program control does not return to the throw point because the `try` block has expired; flow of control proceeds to the first statement after the **last** `catch` block

Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Exceptions Handling

- When an exception occurs within a method, the method creates an exception object and hands it off to the runtime system
  - Creating an exception object and handing it to the runtime system is called "throwing an exception"
  - Exception object contains information about the error, including its type and the state of the program when the error occurred

Exceptions Handling

- **For all subclasses of Exception Class(except RuntimeException) you must include code to deal with them**
- **If your program has the potential to generate an exception of such a type, you have got two choices**
  - Handle the exception within the method
  - Register that your method may throw such an exception (You are passing the exception on)
- **If you do neither your code won’t compile**

Exceptions Handling : try-catch

Syntax:

```
try
{
    <code to be monitored for exceptions>
}
catch (<ExceptionType1> <ObjName>)
{
    <handler if ExceptionType1 occurs>…
}
catch (<ExceptionTypeN> <ObjName>)
{
    <handler if ExceptionTypeN occurs>…
}
```

Exceptions Handling :Example-1

```
class DivByZero
{
   public static void main(String args[])
    {
        try
        {
          System.out.println(3/0);
          System.out.println(“Please print me.”);
        }
        catch (ArithmeticException exc)
        {
          //Division by zero is an ArithmeticException
          System.out.println(exc);
        }
        System.out.println(“After exception.”);
    }
}
```

Exceptions Handling :Example-2

```
class MultipleCatch
{
    public static void main(String args[])
    {
            try
            {
                        int den = Integer.parseInt(args[0]);
                        System.out.println(3/den);
            }
            catch (ArithmeticException exc)
            {
                        System.out.println(“Divisor was 0.”);
            }
            catch (ArrayIndexOutOfBoundsException exc2)
            {
                        System.out.println(“Missing argument.”);
            }
            System.out.println(“After exception.”);
    }
}
```

Exceptions Handling : Nested try

```
class NestedTryDemo
{
    public static void main(String args[])
    {
        try
        {
                int a = Integer.parseInt(args[0]);
                try
                {
                        int b = Integer.parseInt(args[1]);
                        System.out.println(a/b);
                } catch (ArithmeticException e)
                {
                        System.out.println(“Div by zero error!");
                }
        }
         catch (ArrayIndexOutOfBoundsException e)
        {
                System.out.println(“Need 2 parameters!");
        }
    }
}
```

Exceptions Handling : Nested try

```
class Nest{
   public static void main(String args[]){
     //Parent try block
     try{
    //Child try block1
        try{
           System.out.println("Inside block1");
           int b =45/0;
           System.out.println(b);
        }
        catch(ArithmeticException e1){
           System.out.println("Exception: e1");
        }
        //Child try block2
        try{
           System.out.println("Inside block2");
           int b =45/0;
          }
        catch(ArrayIndexOutOfBoundsException e2){
           System.out.println("Exception: e2");
        }
       System.out.println("Just other statement");
    }
```

Exceptions Handling : Nested try

```
catch(ArithmeticException e3){
     System.out.println("Arithmetic Exception");
         System.out.println("Inside parent try catch block");
    }
    catch(ArrayIndexOutOfBoundsException e4){
    System.out.println("ArrayIndexOutOfBoundsException");
         System.out.println("Inside parent try catch block");
    }
    catch(Exception e5){
    System.out.println("Exception");
         System.out.println("Inside parent try catch block");
    }

    System.out.println("Next statement..");
  }  //Main ends here
}
```

Exceptions Handling : Nested try

Output:

```
Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

Exceptions Handling : Nested try

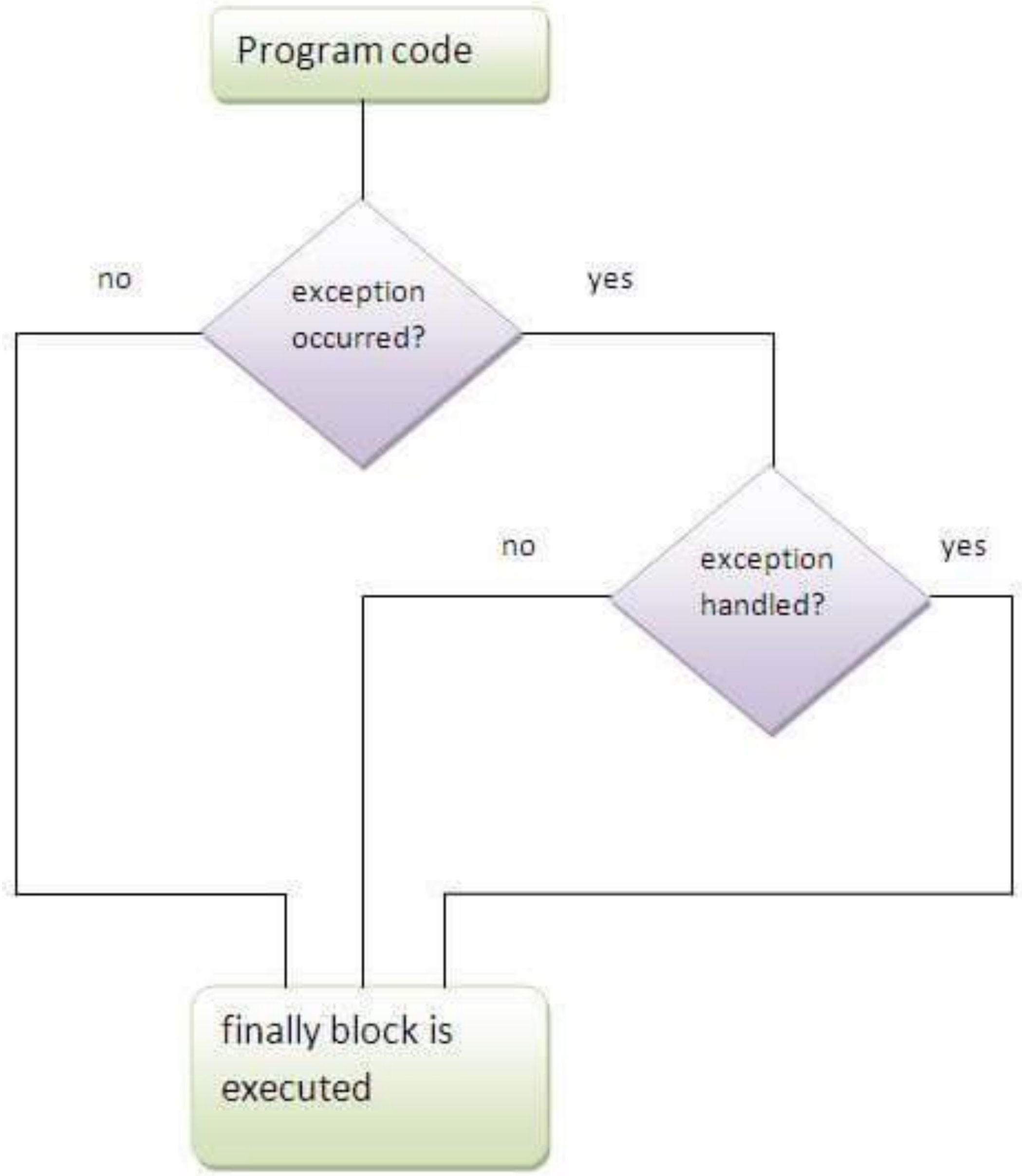
```
class NestedTryDemo2
{
    static void nestedTry(String args[])
    {
        try
        {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                System.out.println(a/b);
         } catch (ArithmeticException e)
        {
                System.out.println("Div by zero error!");
        }
    }
    public static void main(String args[])
    {
        try
        {
                nestedTry(args);
         } catch (ArrayIndexOutOfBoundsException e)
        {
                System.out.println("Need 2 parameters!");
        }
    }
}
```

Exceptions Handling : finally

```
try
{
    <code to be monitored for exceptions>
}
catch (<ExceptionType1> <ObjName>)
{
    <handler if ExceptionType1 occurs>
}
finally
{
    <code to be executed before the try block ends>
}
```

Exceptions Handling : finally

- Java **finally block** is a **block** that is used to execute important code such as closing connection, stream etc.
- Java **finally block** is always executed whether **exception** is **handled** or not.
- Java **finally block** follows try or catch **block**.


Program code
no
exception occurred?
yes
no
exception handled?
yes
finally block is executed

Exceptions Handling : finally

- Block of code is always executed despite of different scenarios:
  - Forced exit occurs using a *return*, *a continue or a break* statement
  - Normal completion
  - Caught exception thrown
    - Exception was thrown and caught in the method
  - Uncaught exception thrown
    - Exception thrown was not specified in any catch block in the method

Exceptions Handling : finally - Example

```
class FinallyDemo {
static void myMethod(int n) throws Exception{
      try {
      switch(n) {
      case 1: System.out.println("1st case");
                return;
      case 3: System.out.println("3rd case");
                throw new RuntimeException("3!");
       case 4: System.out.println("4th case");
                throw new Exception("4!");
       case 2: System.out.println("2nd case");
      }
      } catch (RuntimeException e) {
                System.out.print("RuntimeException: ");
                System.out.println(e.getMessage());
      }
finally{System.out.println("in finally try blk entered");}
System.out.println("after finally");
}
```

```
public static void main(String args[]){
       for (int i=1; i<=4; i++) {
      try {
             FinallyDemo.myMethod(i);
      }
      catch (Exception e){

      System.out.print("Exception
      caught: ");
      System.out.println(e.getMessage(
      ));
      }
      System.out.println();
}
}
}
```

Exceptions Handling : finally - Example

```
class FinallyDemo {
static void myMethod(int n) throws Exception{
     try {
     switch(n) {
     case 1: System.out.println("1st case");
              return;
     case 3: System.out.println("3rd case");
              throw new RuntimeException("3!");
      case 4: System.out.println("4th case");
              throw new Exception("4!");
      case 2: System.out.println("2nd case");
     }
     } catch (RuntimeException e) {
              System.out.print("RuntimeException: ");
              System.out.println(e.getMessage());
     }
finally{System.out.println("in finally try blk entered");}
System.out.println("after finally");
}
```

Exceptions Handling : finally - Example

```
public static void main(String args[])
{
    for (int i=1; i<=4; i++)
   {
        try
        {
                FinallyDemo.myMethod(i);
        }
        catch (Exception e)
        {
                System.out.print("Exception caught: ");
                System.out.println(e.getMessage());
        }
        System.out.println();
   }
}
```

Exceptions Handling : finally - Example

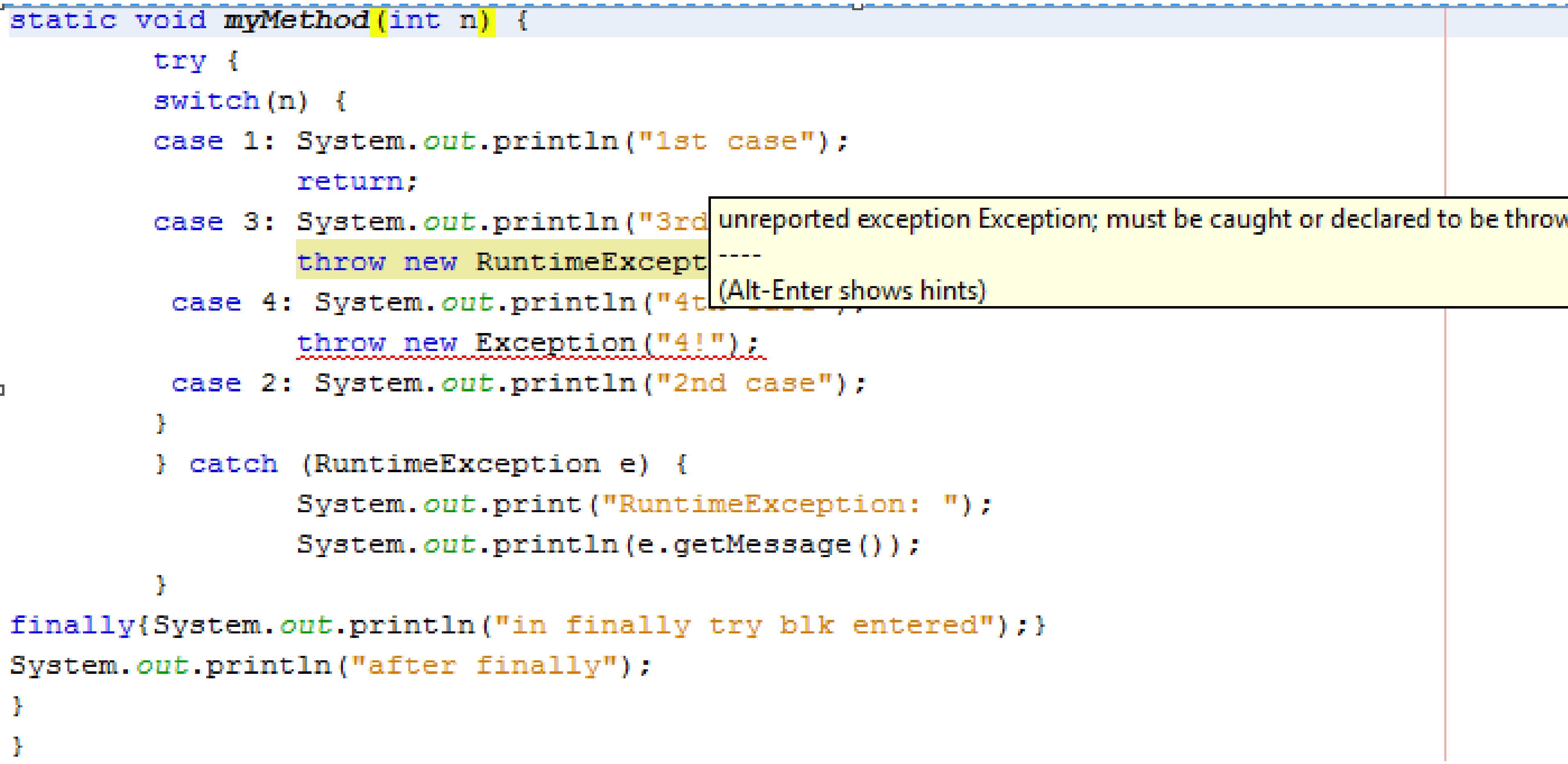
output

```
1st case
in finally try blk entered

2nd case
in finally try blk entered
after finally

3rd case
RuntimeException: case 3!
in finally try blk entered
after finally

4th case
in finally try blk entered
Exception caught: 4!
```



```
static void myMethod(int n) {
        try {
        switch(n) {
        case 1: System.out.println("1st case");
                return;
        case 3: System.out.println("3rd
                throw new RuntimeExcept
         case 4: System.out.println("4t
                throw new Exception("4!");
         case 2: System.out.println("2nd case");
        }
        } catch (RuntimeException e) {
                System.out.print("RuntimeException: ");
                System.out.println(e.getMessage());
        }
finally{System.out.println("in finally try blk entered");}
System.out.println("after finally");
}
}
```

unreported exception Exception; must be caught or declared to be throw
----
(Alt-Enter shows hints)

Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Using the throws Clause

- **`throws`** clause – specifies the exceptions a method may throws
  - Appears after method’s parameter list and before the method’s body
  - Contains a comma-separated list of exceptions
  - Exceptions can be thrown by statements in method’s body or by methods called in method’s body
  - Exceptions can be of types listed in `throws` clause or subclasses

Throwing Exception

Java allows you to throw exceptions (generate exceptions)

```
throw <exception object>;
```

- An exception you throw is an object

You have to create an exception object in the same way you create any other object

- Example:

```
throw new ArithmeticException(“testing...”);
```

Throwing Exception

```
class ThrowDemo {

public static void main(String args[]){
     String input = “invalid input”;
     try {
          if (input.equals(“invalid input”)) {
                    throw new RuntimeException("throw demo");
          }
           else {
                    System.out.println(input);
          }
          System.out.println("After throwing");
     } catch (RuntimeException e)  {
          System.out.println("Exception caught:" + e);
     }
}
}
```

Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Creating Your Own Exception

- Steps to follow
  - Create a class that *extends the RuntimeException or the Exception class*
  - Customize the class
- Members and constructors may be added to the class
  - Example:

```
class HateStringExp extends RuntimeException {
        /* some code */
}
```

Creating Your Own Exception

```
class TestHateString
{
   public static void main(String args[])
   {
        String input = "invalid input";
        try
        {
                if (input.equals("invalid input"))
                {
                        throw new HateStringExp();
                }
                System.out.println("Accept string.");
        } catch (HateStringExp e)
        {
        System.out.println("Hate string!”);
        }
   }
}
```

Agenda

- Introduction
- Difference between Error and Exception
- Dealing With Exceptions
- Throwing Exceptions
- Creating Your Own Exceptions

Reading Material

- Reading Material:
  - Chapter 11