LESSON X. Exception Handling

Vu Thi Huong Giang

Objectives

- After this lesson, students (learners) can:
  - Learn what an exception is, what have to do to handle exception
  - Distinguish the checked and unchecked exceptions
  - Implement the try-catch-finally blocks for catching and handling exceptions
  - Write methods that propagate exceptions.
  - Write programmer-defined exception classes.

Content

I. **Exception & Exception handling**
II. Exception classes hierarchy
III. try/catch/finally blocks
IV. Throwing exceptions
V. Print a stack trace message
VI. Making custom exceptions

I. Exception and Exception Handling

1. **Exception**
2. Exception Handling

1. Exception-What are errors?

- There are three categories or types of errors:
  - (1) Syntax errors
  - (2) Logic errors (sometimes called: semantic errors)
  - (3) Runtime errors
- Syntax errors arise because the rules of the language have not been followed. They are detected by the compiler.
- Logic errors occur when a program doesn't perform the way it was intended to.
- Runtime errors occur while the program is running if the environment detects an operation that is impossible to carry out

1. Exception-Error examples

- User Input Errors:
  - connect to a URL which is incorrect
  - network exception
- Device Errors:
  - printer is off/ out of paper
  - Network is down
- Physical limitations:
  - Disks run out of memory,
  - quotas fill up
  - an infinite recursion causes a stack overflow.
- Code errors:
  - divide by zero,
  - array out of bound,
  - integer overflow,
  - access a null pointer

1. Exception-Definitions

- *Exception*: an occurrence of an undesirable situation that can be detected during program execution. Examples:
  - Division by zero
  - Trying to open an input file that does not exist
  - An array index that goes out of bounds
- When an exception occurs, or is *thrown*, the normal sequence of flow is terminated. The exception-handling routine is then executed; we say the thrown exception is *caught*

1. Exception-Process when exception occurs

1. When an exception occurs, Java allows a method to **terminate abnormally**
2. It **throws** an object that encapsulates error information
3. It does not return a value
4. It exits immediately
5. Execution does not resume at the point of method call
6. JVM searches for an **exception handler**
7. If none is found, the program crashes.

I. Exception and Exception Handling

1. Exception
2. **Exception Handling**

2. Exception Handling

- Exception handling is generally used to handle abnormal events in a program.
- When a method encounters a problem that disrupts the normal flow, it can cause the program to CRASH ! But with exception handling, the method can throw an exception which informs the caller that a problem occurred allowing the caller to perform alternative actions.
- Exception handling enables more robust programming by providing ways of recovering from errors or problems

2. Exception Handling (within a Program)

- Can use an if statement to handle an exception

```
Perform a task
If the preceding task did not execute correctly
   Perform error processing
Perform next task
If the preceding task did not execute correctly
   Perform error processing
```

2. Exception Handling

- Exception handling streamlines error handling by allowing your program to define a block of code, called an exception handler that is executed automatically when an error occurs
- All exception handling has the same goals:
  - 1. Anticipate the error by the user/system
  - 2. Return the program to a safe state that enables the user to execute other commands
  - 3. Inform the user of the error's cause
  - 4. Allow the user to save work and terminate the program gracefully.

2. Exception Handling-Java's Mechanism of Exception Handling

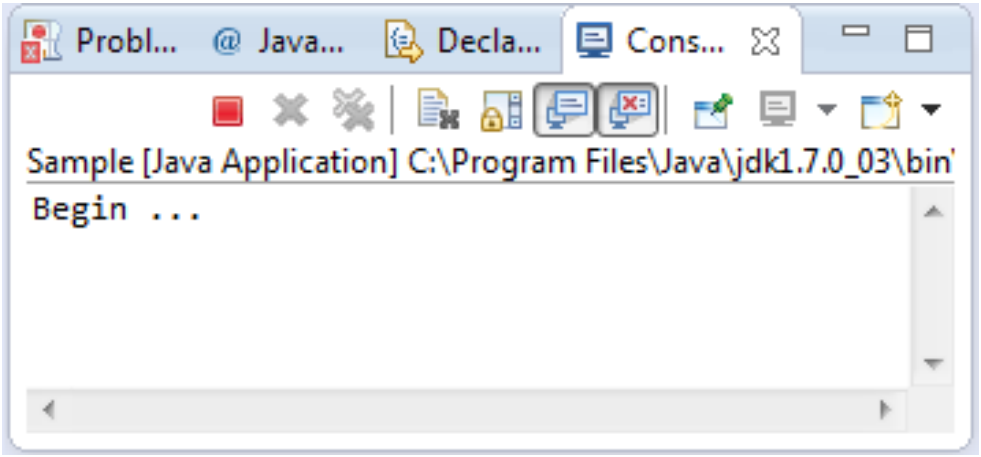
- When an exception occurs, an object of a particular exception `class` is created
- Java provides a number of exception classes to effectively handle certain common exceptions, such as:
  - Division by zero: handled by the `ArithmeticException class`.
  - Invalid input: handled by the `InputMismatchException class.`
  - File not found: handled by the `FileNotFoundException class.`

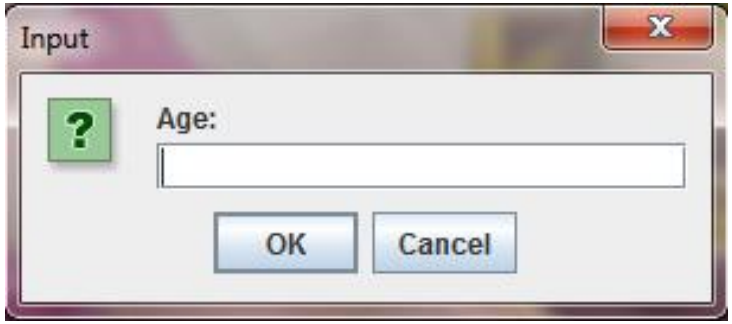
2. Exception Handling-Techniques

- Techniques in Exception handling:
  - Terminate program
  - Fix error and continue
  - Log error and continue

Example:
code without exception handling

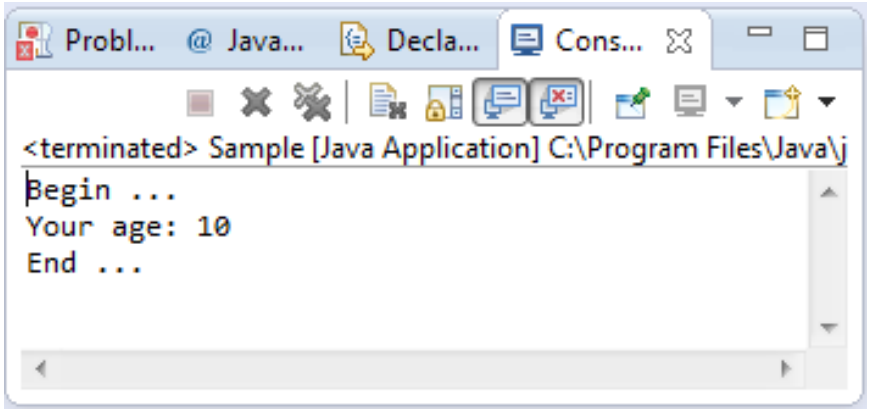
```
import javax.swing.JOptionPane;

public class Sample {
	public static void main(String[] args) {
	// TODO Auto-generated method stub
	System.out.println("Begin ...");
	String inputStr;
	int age;
	    inputStr = JOptionPane.showInputDialog(null, "Age:");
	    age = Integer.parseInt(inputStr);
	    System.out.println("Your age: " + age);
	    System.out.println("End ...");
	}
}
```

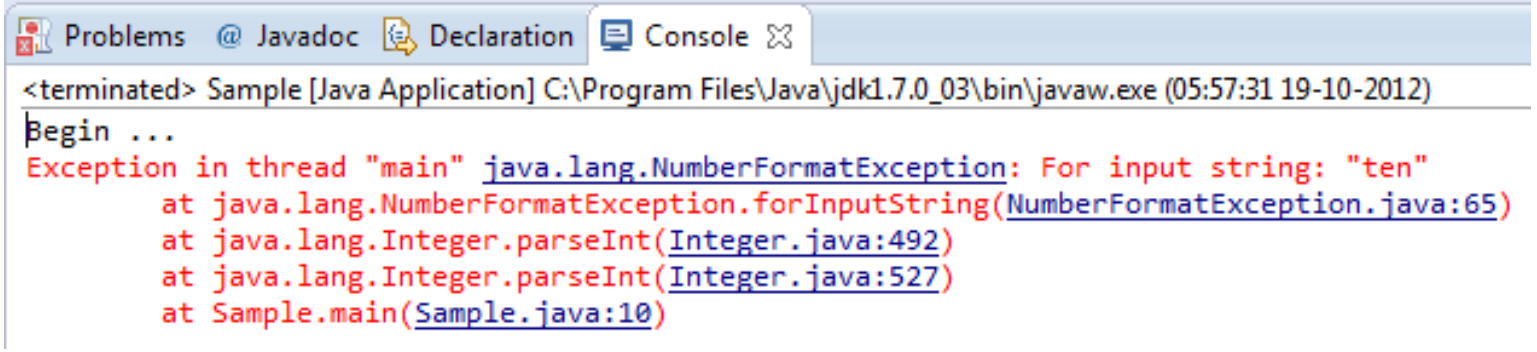




Example: code without exception handling

- What would happen if the user enters an integer such as 10?
- What would happen if the user enters a text “ten” instead of the number 10?

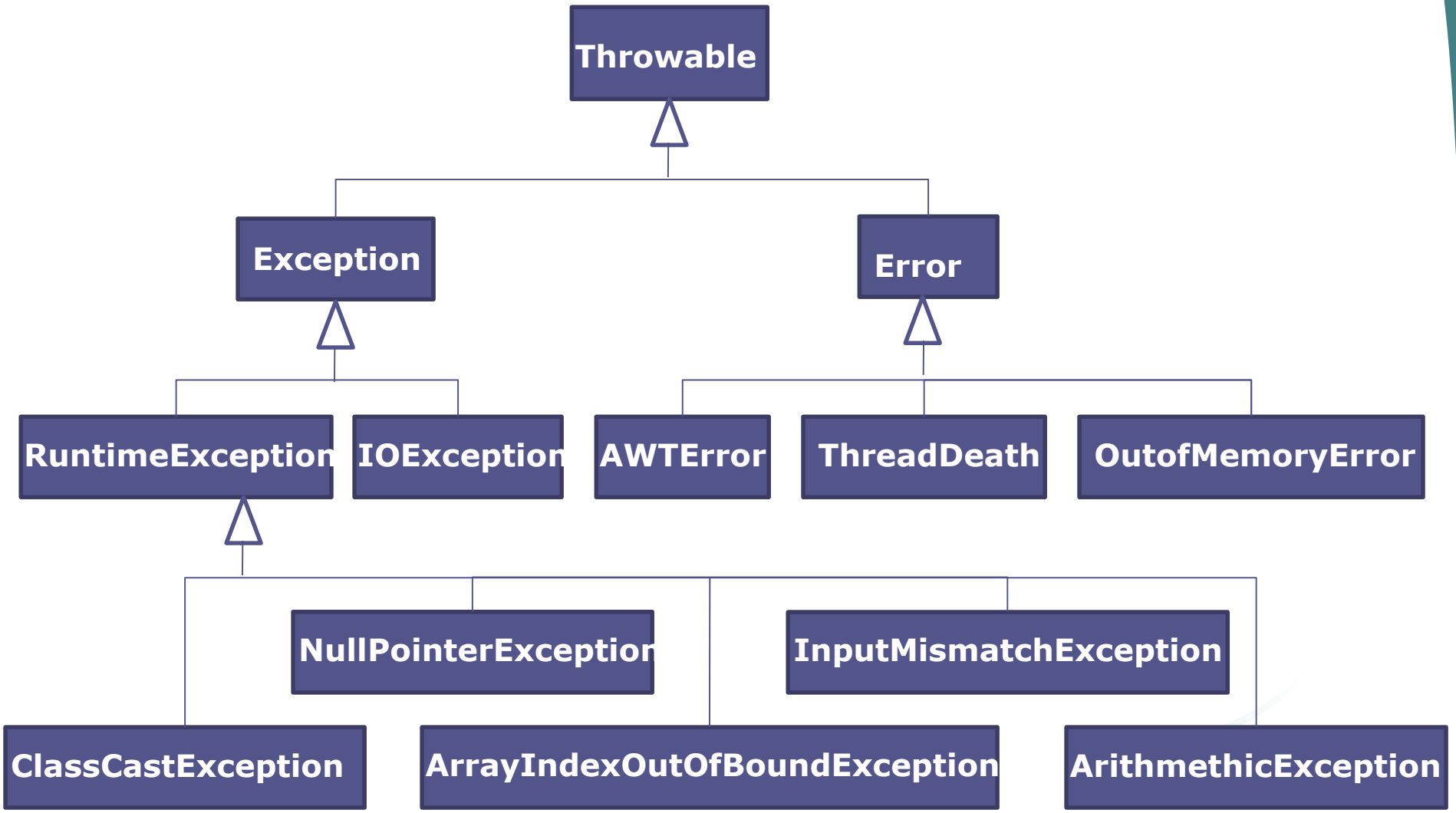




Content

I. Exception & Exception handling
II. Exception classes hierarchy
III. try/catch/finally blocks
IV. Throwing exceptions
V. Print a stack trace message
VI. Making custom exceptions

II. Exception class hierarchy



Constructors and methods of class Throwable

```
public Throwable()
   //Default constructor
   //Creates an instance of Throwable with an empty message string
public Throwable (String strMessage)
   //Constructor with parameters
   //Creates an instance of Throwable with message string
   //specified by the parameter strMessage
public String getMessage( )
   //Returns the detailed message stored in the object
public void printStackTrace( )
   //Method to print the stack trace showing the sequence of
   //method calls when an exception occurs
public void printStackTrace (PrintWriter stream)
   //Method to print the stack trace showing the sequence of
   //method calls when an exception occurs. Output is sent
   //to the stream specified by the parameter stream.
public String toString( )
   //Returns a string representation of the Throwable object
```

class Exception and its Constructors

```
public Exception( )
//Default constructor
//Creates a new instance of the class Exception

public Exception (String str)
//Constructor with parameters
//Creates a new instance of the class Exception.
//The parameter str specifies the message string.
```

Types of Exceptions in Java

- Unchecked exceptions
  - RuntimeException, Error and their subclasses are known as UNCHECKED EXCEPTIONS
- Checked exceptions
  - Other Classes
  - Compiler forces the programmer to check and deal with checked exceptions (or there will be a compilation error)

Some of Java's Exception Classes (1/2)

Exception Class	Description
`ArithmeticException`	Arithmetic errors such as division by zero
`ArrayIndexOutOfBoundException`	Array index is either less than 0 or greater than or equal to the length of the array
`FileNotFoundException`	Reference to a file that cannot be found
`IllegalArgumentException`	Calling a method with illegal arguments
`IndexOutOfBoundException`	An array or a string index is out of bounds
`NullPointerException`	Reference to an object that has not been instantiated

Some of Java's Exception Classes (2/2)

Exception Class	Description
NumberFormatException	Use of an illegal number format
StringIndexOutOfBoundsException	A string index is either less than 0 or greater than or equal to the length of the string
InputMismatchException	Input (token) retrieved does not match the pattern for the expected type, or the token is out of range for expected type

Content

I. Exception & Exception handling
II. Exception classes hierarchy
III. try/catch/finally blocks
IV. Throwing exceptions
V. Print a stack trace message
VI. Making custom exceptions

Keywords for Java Exceptions

- try
- catch
- finally
- throws
- throw

III. try/catch/finally Block

```
try {
   // statements that might
   // generate an exception
   // statements that should not
   // be executed if an exception
   // occurs
}
catch (ExceptionType name){
   // exception handler
}
[… catch (ExceptionType name){}]
[finally { … }]
```

```
try {
   // statements
}
finally {
   // statements that are always
   // executed, regardless of
   // whether an exception occurs,
   // except when the program
   // exits early from a try block
   // by calling the method
   // System.exit
}
```

III. try/catch/finally Block

```
try {
   // statements
}
catch (ExceptionType name){
   // exception handler
}
// …
catch (ExceptionType name){
}
finally {
   // statements
}
```

- If no exception is thrown in a `try` block, all `catch` blocks associated with the `try` block are ignored and program execution resumes after the last `catch` block
- If an exception is thrown in a `try` block, the remaining statements in the `try` block are ignored
- If the type of the thrown exception matches the parameter type in one of the `catch` blocks, the code of that `catch` block executes
- If there is a `finally` block after the last `catch` block, the `finally` block executes regardless of whether an exception occurs

Order of `catch` Blocks

```
try {
    // statements
}
catch (ExceptionType name){
    // exception handler
}
// …
catch (ExceptionType name){
}
finally {
    // statements
}
```

- The heading of a `catch` block specifies the type of exception it handles
- A reference variable of a superclass type can point to an object of its subclass
- If in the heading of a `catch` block you declare an exception using the `class Exception`, then that `catch` block can catch all types of exceptions because the `class Exception` is the superclass of all exception classes
- In a sequence of `catch` blocks following a `try` block, a `catch` block declaring an exception of a subclass type should be placed before `catch` blocks declaring exceptions of a superclass type

III. `try/catch/finally` Block

```
try {
    resource-acquisition statements
} // end try
catch ( AKindOfException exception1 ) {
    exception-handling statements
} // end catch
catch ( AnotherKindOfException exception2 ) {
    exception-handling statements
} // end catch
finally {
    resource-release statements
} // end finally
```

Nested Try Statement

```
try {
        try {
                // ...
        }
        catch (Exception1 e){
                //statements to handle the exception
        }
}
catch (Exception2 e2){
        //statements to handle the exception
}
```

III. try/catch/finally Block – Example 1

```
import java.io.*;

class Test{
  public static void main(String args[]){
     FileInputStream fis=null;
     try{
       fis = new FileInputStream (new File (args[0]));
     }
     catch (ArrayIndexOutOfBoundsException e){
       System.out.println("Argument missing!");
     }
     catch (FileNotFoundException e){
       System.out.println("File not found!");
     }
     finally{
       fis.close();
     }
  }
}
```

III. try/catch/finally Block – Example 2

```
public static void main(String args[]) {
    try{
        System.out.println("Start try");
        int n = Integer.parseInt(args[0]);
        int i = 9/n;
        System.out.println("End try");
    }catch (ArithmeticException ex){
        System.out.println("Start catch block 1");
        System.out.println("End catch block 1");
    }catch (Exception ex){
        System.out.println("Start catch block 2");
        System.out.println("End catch block 2");
    }finally{
      System.out.println("Finally block always executed");
    }
}
```

Content

I. Exception & Exception handling
II. Exception classes hierarchy
III. try/catch/finally blocks
IV. Throwing exceptions
V. Print a stack trace message
VI. Making custom exceptions

1. Four ways to throw Exceptions

1. Programming error e.g. `a[-1] =0;` generates an ***unchecked exception***

   `ArrayIndexOutOfBoundsException`

2. JVM or runtime library internal error
3. Code detects an error and generates ***checked exception*** with **`throw`** statement
4. Calling a method that `throws` a ***checked exception***

2. Throw an Exception explicitly

- It is possible for your program to throw an exception explicitly
- Syntax:
- `throw exceptionInstance`
  - Here, `exceptionInstance` must be an object of type `Throwable` or a subclass `Throwable`
- Example:

```
SomeException ex = new SomeException("Some message");
throw ex;
```

3. Rethrow an Exception explicitly

- We can handle or rethrow an exception:
  - When an exception occurs in a `try` block, control immediately passes to one of the `catch` blocks; typically, a `catch` block does one of the following:
    - Completely handles the exception
    - Partially processes the exception; in this case, the `catch` block either **rethrows** the same exception or `throws` another exception for the calling environment to handle the exception
    - **Rethrows** the same exception for the calling environment to handle the exception

4. Propagate Exception-throws clause

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception
- Syntax:

```
type method-name parameter-list) throws exception-list
{
   // body of method
}
```

- Example:

```
public void myMethod()
                throws IOException, FileNotFoundException{
   // some code here may or may not throw the declared exceptions
}
```

4. Propagate Exception-throws clause

Caller A (Catcher)

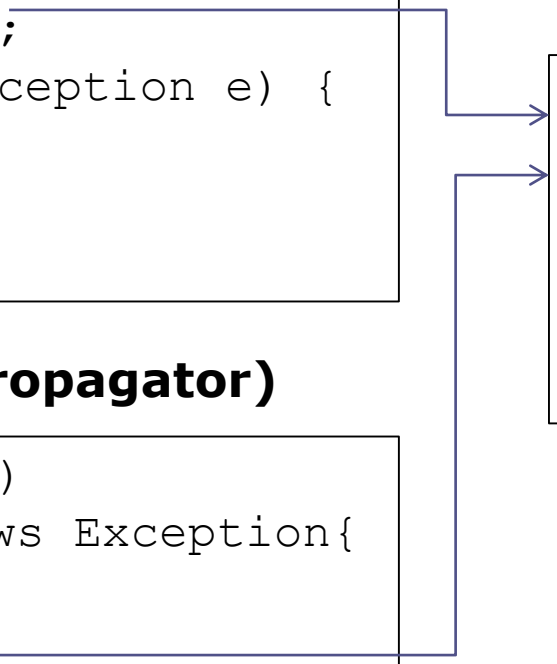
```
void callerA(){
  try{
     doWork();
  } catch (Exception e) {
     ...
  }
}
```

Caller B (Propagator)

```
void callerB()
          throws Exception{
  ...
  doWork();
  ...
}
```

doWork throws Exception

```
public void doWork()
          throws Exception{
  ...
  throw new Exception();
  ...
}
```



5. Rule for Method Overriding

- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not.
- The overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method.
- The overriding method can throw narrower or fewer exceptions than the overridden method

Content

I. Exception & Exception handling
II. Exception classes hierarchy
III. try/catch/finally blocks
IV. Throwing exceptions
V. Print a stack trace message
VI. Making custom exceptions

V. Print a stack trace message

- getMessage(): print errors to debug the process
- printStackTrace(): more effective to debug the process

The Method `printStackTrace` – Example (1/3)

```
import java.io.*;
public class PrintStackTraceExample1
{
    public static void methodA() throws Exception{
        methodB();
    }
    public static void methodB() throws Exception{
        methodC();
    }
    public static void methodC() throws Exception{
        throw new Exception("Exception generated "
                            + "in method C");
    }
```

The Method `printStackTrace` – Example (2/3)

```
    public static void main(String[] args)
    {
        try
        {
            methodA();
        }
        catch (Exception e)
        {
            System.out.println(e.toString()
                                + " caught in main");
            e.printStackTrace();
        }
    }

}
```

The Method `printStackTrace` – Example (3/3)

- Sample Run:

```
java.lang.Exception: Exception generated in
  method C caught in main
java.lang.Exception: Exception generated in
  method C
        at PrintStackTraceExample1.methodC
       (PrintStackTraceExample1.java:30)
        at PrintStackTraceExample1.methodB
        (PrintStackTraceExample1.java:25)
        at PrintStackTraceExample1.methodA
        (PrintStackTraceExample1.java:20)
        at PrintStackTraceExample1.main
        (PrintStackTraceExample1.java:9)
```

Content

I. Exception & Exception handling
II. Exception classes hierarchy
III. try/catch/finally blocks
IV. Throwing exceptions
V. Print a stack trace message
VI. Making custom exceptions

VI. Making Custom Exceptions

- You may not find a good existing exception class
- Exception class you define extends class Exception or one of its subclasses
- Give a default constructor and a constructor that takes a message
- Two primary use cases for a custom exception
  - throw the custom exception when something goes wrong
  - wrap an exception that provides extra information by adding our own message

Example (1/2)

```
import java.io.*;
import java.util.*;
class MyException extends Exception{
   public MyException(){
     super("Error! You are not permitted to enter inside");
   }
   public MyException(String message){
     super(message);
   }
}
```

Example (2/2)

```
public class ExcepDemo{
  public static void main(String args[])throws MyException,IOException{
    String temp="";
    try{
       String str="admin";
       System.out.println("Enter the your name");
       BufferedReader br =
                  new BufferedReader(new InputStreamReader(System.in));
       temp=br.readLine();
       if(!temp.equals(str))
         throw new MyException();
       else
         System.out.println("Welcome to the System");
     }
     catch(MyException e){
       System.err.println(e.getMessage());
     }
     catch(Exception e){
       e.printStackTrace();
     }
  }//end of main
}//end of class
```

Quick quiz (1)

1. What exact command can trigger error? Which type of error is it?
   If we run this program, what will be printed on the screen?

```
class StrExceptionDemo  {
  static String str;
  public static void main(String s[])  {
   try {
     System.out.println("Before exception");
      staticLengthmethod();
      System.out.println("After exception");
    }
    catch(NullPointerException ne)  {
      System.out.println("There's error");
   }
    finally {
        System.out.println("In finally");
    }
  }

  static void staticLengthmethod() {
     System.out.println(str.length());
  }
}
```

Quick quiz (2)

2. Which statement is correct?
    a. All statements in try block are always executed
    b. All statements in catch block are always executed
    c. All statements in finally block are always executed
3. Choose the correct answer. Try block can be followed by:
    a. More than 1 catch blocks and 1 finally block
    b. Only 1 catch block
    c. Only 1 finally block
    d. No catch block and no finally block
4. Which type of Exception if we don't catch or propagate, there's no compilation errors (but maybe runtime errors)
    a. Exception
    b. ArithmeticException
    c. ArrayIndexOutOfBoundException
    d. Throwable

Quick quiz (3)

5. How many compilation errors can be found in this methods. What are they?

```
public boolean CheckZero(int j)
                        throw Exception{
  if (j == 0)
     throws new Exception();
}
```

Quiz 1

- Have the following class:

```
import java.util.Scanner;
public class Simple {
  public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    System.out.print("Input your age: ");
    int age = s.nextInt();
    System.out.println("Your age is: " + age);
  }
}
```

- 1. Is there any compilation errors in this application?
- 2. Can this application run properly in every case?
- 3. If not, fix the code to ensure the application run properly

Quiz 1-Solution (1/2)

- 1. There are no compilation errors
- 2. If user enters non-integer value, there will be an `InputMismatchException`, and the application is crashed

Quiz 1-Solution (2/2)

```
import java.util.*;
public class SimpleFix {
  public static void main(String[] args) {
    boolean flag = true;
    while (flag) {
      flag = false;
      try {
        Scanner s = new Scanner(System.in);
        System.out.print("Input your age: ");
        int age = s.nextInt();
        System.out.println("Your age is: " + age);
      } catch (InputMismatchException e) {
        System.out.println("Your age should be an integer. Try again!");
        flag = true;
      } catch (Exception e) {
        flag = true;
      }
    } //end of while
  }//end of main
}//end of class
```

Quiz 2

- Modify the solution in the quiz 1 as following:
  - 1. Write two user-defined exceptions:
    - `AgeUndefinedException`: the object of this exception will be thrown if the input age is not an integer
    - `AgeTooLowException`: the object of this exception will be thrown if the input age<10
    - `AgeTooHighException`: the object of this exception will be thrown if the input age>150
  - 2. Write a static method
    - `public static int inputAge()throws AgeTooLowException, AgeTooHighException, AgeUndefinedExcpetion;`
  - 3. Modify the main method, catch those exceptions

Quiz 2 – Solution (1/3)

```
public class AgeTooLowException extends Exception {
   public AgeTooLowException(){
      super("Your age should >= 10");
   }
}

public class AgeTooHighException extends Exception {
   public AgeTooHighException(){
      super("Your age should <= 150");
   }
}

public class AgeUndefinedExcpetion extends Exception {
    public AgeUndefinedExcpetion(){
      super("Your age should be an integer");
    }
}
```

Quiz 2 – Solution (2/3)

```
import java.util.*;
public class AgeApplication {
  public static int inputAge()throws AgeTooLowException,
                        AgeTooHighException, AgeUndefinedExcpetion{
    int age = 0;
    try {
       Scanner s = new Scanner(System.in);
       System.out.print("Input your age: ");
       age = s.nextInt();
       if (age < 10)
           throw new AgeTooLowException();
       else if (age > 150)
           throw new AgeTooHighException();
     }catch (InputMismatchException e) {
        throw new AgeUndefinedExcpetion();
     }
     return age;
  }
```

Quiz 2 – Solution (3/3)

```
  public static void main(String[] args) {
     boolean flag = true;
     while (flag) {
          flag = false;
          try {
               System.out.println("Your age is: " + inputAge());
          } catch (AgeTooLowException e) {
               System.out.println(e.getMessage());
               flag = true;
          } catch (AgeTooHighException e) {
               System.out.println(e.getMessage());
               flag = true;
          } catch (AgeUndefinedExcpetion e) {
               System.out.println(e.getMessage());
               flag = true;
          }
       }//end of while
  }//end of main
}//end of class
```

Review

- Exception & Exception Handling
  - Definition
- Exception Hierarchy
  - Throwable, Error, Exception, RuntimeException
  - Checked and unchecked exceptions
- Handling exceptions within a program
  - `try/catch/finally` block
  - Order of `catch` blocks
  - Using `try/catch` blocks in a program
  - Throwing an exception
  - Rethrowing an exception

Review

- The method `printStackTrace`
- Exception handling techniques
  - Terminate program
  - Fix error and continue
  - Log error and continue
- Creating your own exception classes