Session 4

Exception in Java

Outline

- Exception Handling
  - Throwing Exception
  - Try, catch and finally
- Exception as objects
  - Creating new exception classes
- Exception hierarchy

Exception Handling

- An exception in Java is a signal that indicates the occurrence of some important or ***unexpected condition*** during execution.

- Example exceptions
  - A requested file cannot be found
  - An array index is out of bounds
  - A network link failed

- Java provides an exception handling mechanisms for systematically dealing with such error conditions.

Types of errors

- **User Input error**
  - e.g enter character instead of integer
  - request URL which is syntatically wrong
- **Device error**
  - e.g., the printer may be turn off/ run out of paper during printing.
  - The requested web page is not available
- **Physical limitation**
  - e.g., disk can fill up / out of available memory
- **Code error**
  - e.g., invalid array index, non-existence method call, pop empty stack

throw-and-catch paradigm

throw – to throw an exception is to signal that an unexpected error condition has occurred.

catch – to catch an exception is to take appropriate action to deal with the exception.

- Done by exception handler.

Handling Exceptions in two ways

1. **Implicit** Exception Handling
   - To let Java handle such errors automatically
   - use **throws** construct

2. **Explicit** Exception Handling
   - To let programmer to provide code dealing with errors
   - use **try-catch-finally** construct

throws construct

- A java method can throw an exception if it encounters a situation it cannot handle.

```
method(…) throws exception-name1, exception-name2, ….
{
    //statement


}
```

Example:

public String readLine() throws IOException

Example using throws

```
class Calculator
{
    public static void main(String[] args) throws ArithmeticException
    {
        int num1=100;
        int num2= 0;
        int result=num1/num2;
        System.out.println("result = " + result);
    }
}
```

try-catch construct

```
method(…)
{
    try{
                    < statements>
         }catch(exception-name1    parameter1){
                    < statements>
         }catch(exception-name2    parameter2){
                    < statements>
         }
         ………………..
          catch(exception-nameN    parameterN){
                    < statements>
         }
}
```

Example using **try-catch**

```
class Calculator{
    public static void main(String[] args) {
        try{
                int num1=100;
                int num2= 0;
                int result=num1/num2;
                System.out.println("result = " + result);
          }catch(ArithmeticException  e)
          {
                System.err.println("Error occurred in division!");
          }
    }
}
```

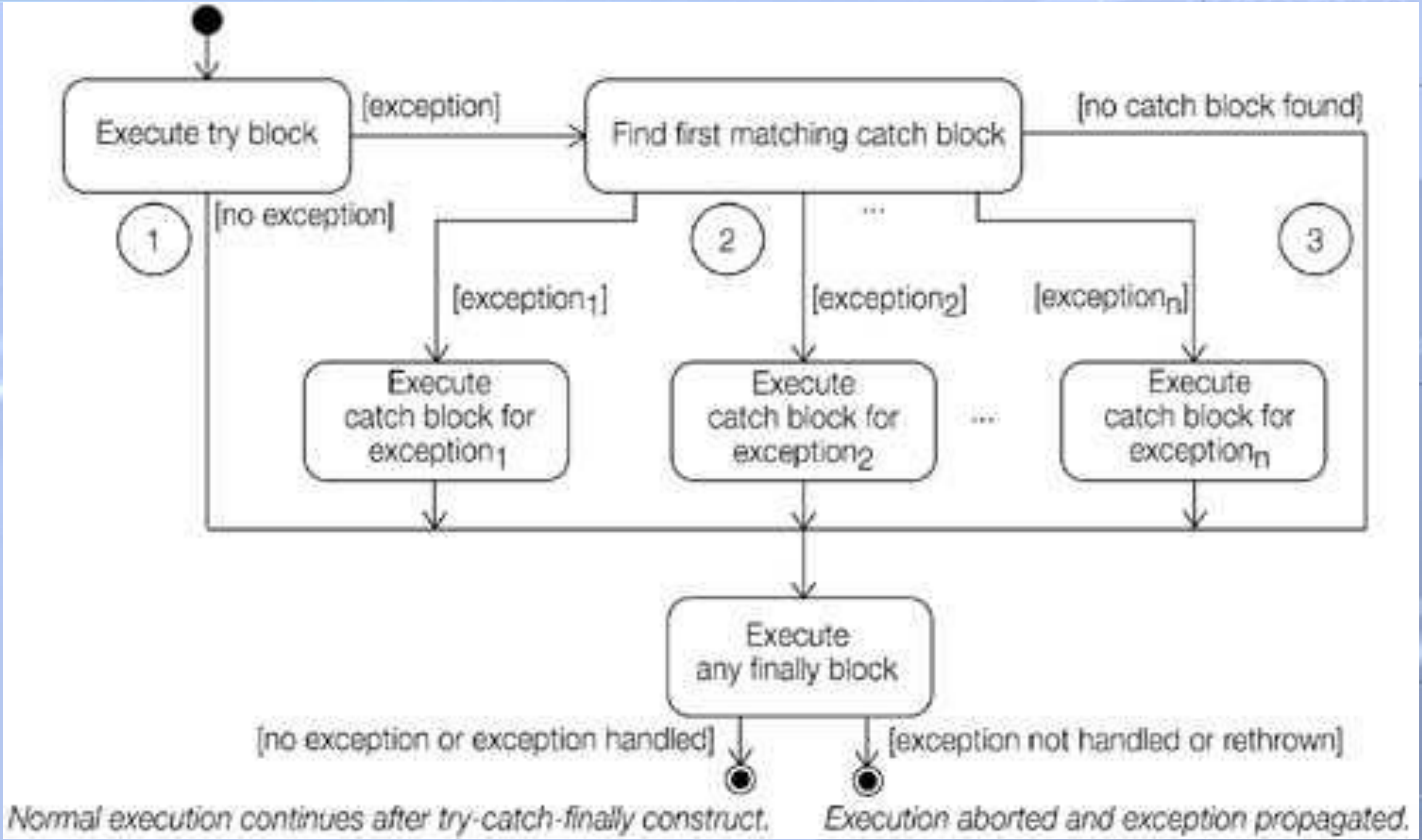
try-catch-finally construct

```
method(…)
{
    try{
                    < statements>
          }catch(exception-name1    parameter1){
                    < statements>
          }
          ……………….
           catch(exception-nameN    parameterN){
                    < statements>
          }finally  {
                    < statements>
          }
}
```

Example using try-catch-finally

```
class Calculator{
    public static void main(String[] args) {
        try{
                int num1=100;
                int num2= 0;
                int result=num1/num2;
                System.out.println("result = " + result);
          }catch(ArithmeticException  e)
          {
                System.err.println("Error occurred in division!");
          }finally
          { System.out.println("exit program."); }
    }
}
```

try-catch-finally construct



Execute try block
[exception]
Find first matching catch block
[no catch block found]
[no exception]
1
2
3
...
[exception1]
[exception2]
[exceptionn]
Execute catch block for exception1
Execute catch block for exception2
Execute catch block for exceptionn
Execute any finally block
[no exception or exception handled]
[exception not handled or rethrown]
Normal execution continues after try-catch-finally construct.
Execution aborted and exception propagated.

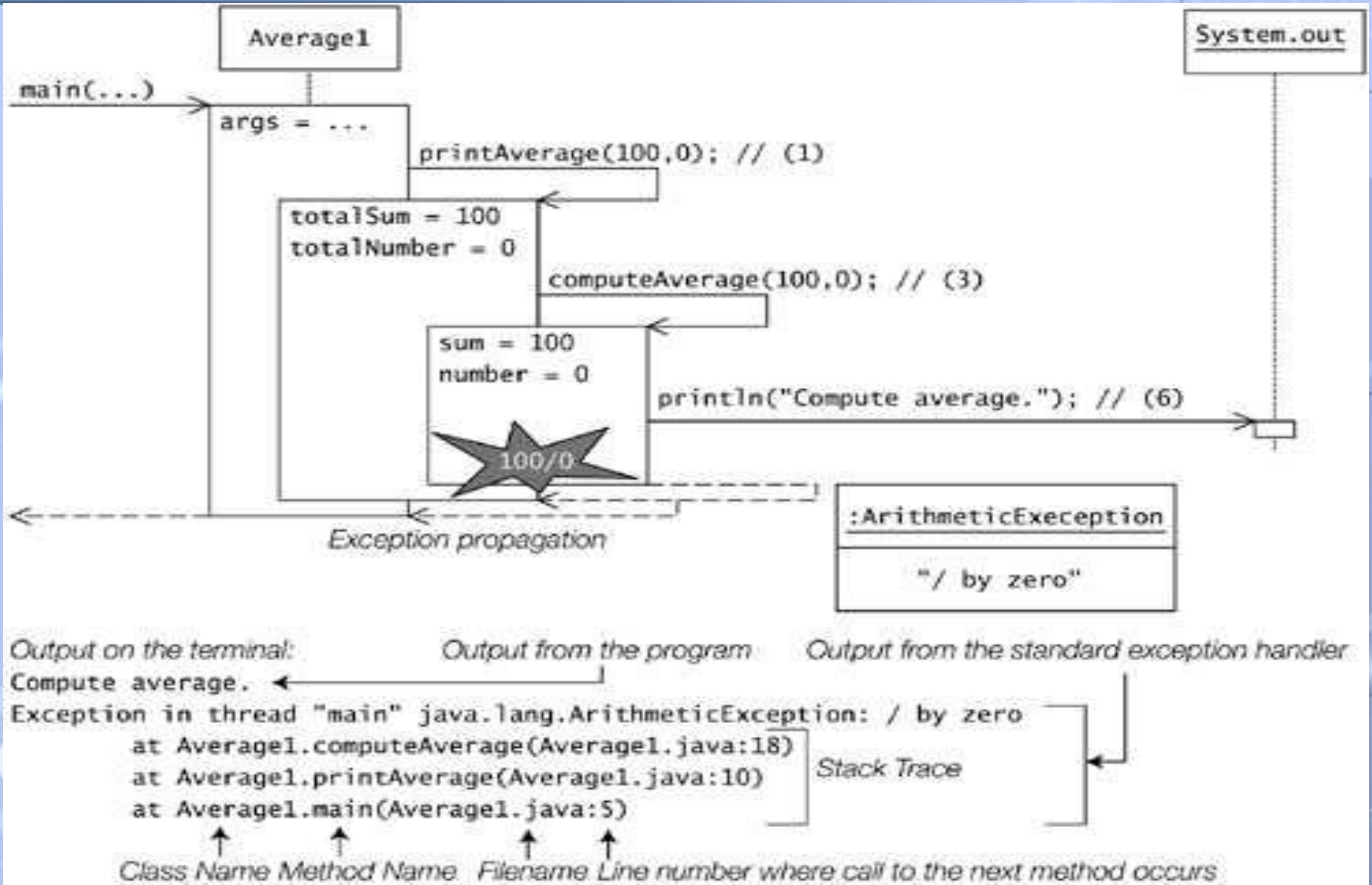
Exception Propagation – let see!

```
public class Average1 {
    public static void main(String[] args) {
        printAverage(100,0); // (1)
        System.out.println("Exit main()."); // (2)     }
    public static void printAverage(int totalSum, int totalNumber) {
        int average = computeAverage(totalSum, totalNumber); // (3)
        System.out.println("Average = " +  totalSum + " / " + totalNumber + " = "
                            + average); / /(4)
        System.out.println("Exit printAverage()."); // (5)        }
    public static int computeAverage(int sum, int number) {
        System.out.println("Computing average."); // (6)
        return sum/number; // (7)
    }
}
```

Result after execution

```
Computing average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Average1.computeAverage(Average1.java:18)
at Average1.printAverage(Average1.java:10)
at Average1.main(Average1.java:5)
```

Exception Handling



```
Compute average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```

throw statement

- A program can explicitly throw an exception using the throw statement.
- Syntax is

  throw new exception-name;

- Example

```
throw new ArithmeticException();

throw new ArithmeticException(“Division by zero!”);
```

Example using **throw**

```
class Calculator
{
    public static void main(String[] args)
    {
        int num1=100;
        int num2= 0;
        if(num==0)
            throw new ArithmeticException();
        int result=num1/num2;
        System.out.println("result = " + result);
    }
}
```

Defining user-defined Exception

- To provide fine-grained categorization of exceptional conditions, instead of using existing exception classes with descriptive detail messages to differentiate between the conditions.
- New exception usually extends the **Exception** class or one of its checked subclasses , there by making the new exception checked.
- They can declare fields and methods.

Example

```
class MyException extends Exception
{
    Date date;

    public MyException(String msg, Date d)
     {
            super(msg); //set the detail msg to the Throwable class
            date=d;
    }
}
```

Types of Exception

- Exceptions in Java are objects.
- All exceptions derived from **Throwable** class.
- Two main subclasses:
  - **Exception**
    - It indicates conditions that a reasonable application might want to catch.
  - **Error**
    - It indicates serious problems that a reasonable application should not try to catch.
    - Most such errors are abnormal conditions.

Exception classes

Some subclasses of **Exception** are

- IOException
- RuntimeException
- ClassNotFoundException
- AWTException

❖ IOException and its subclasses are in the java.io package.

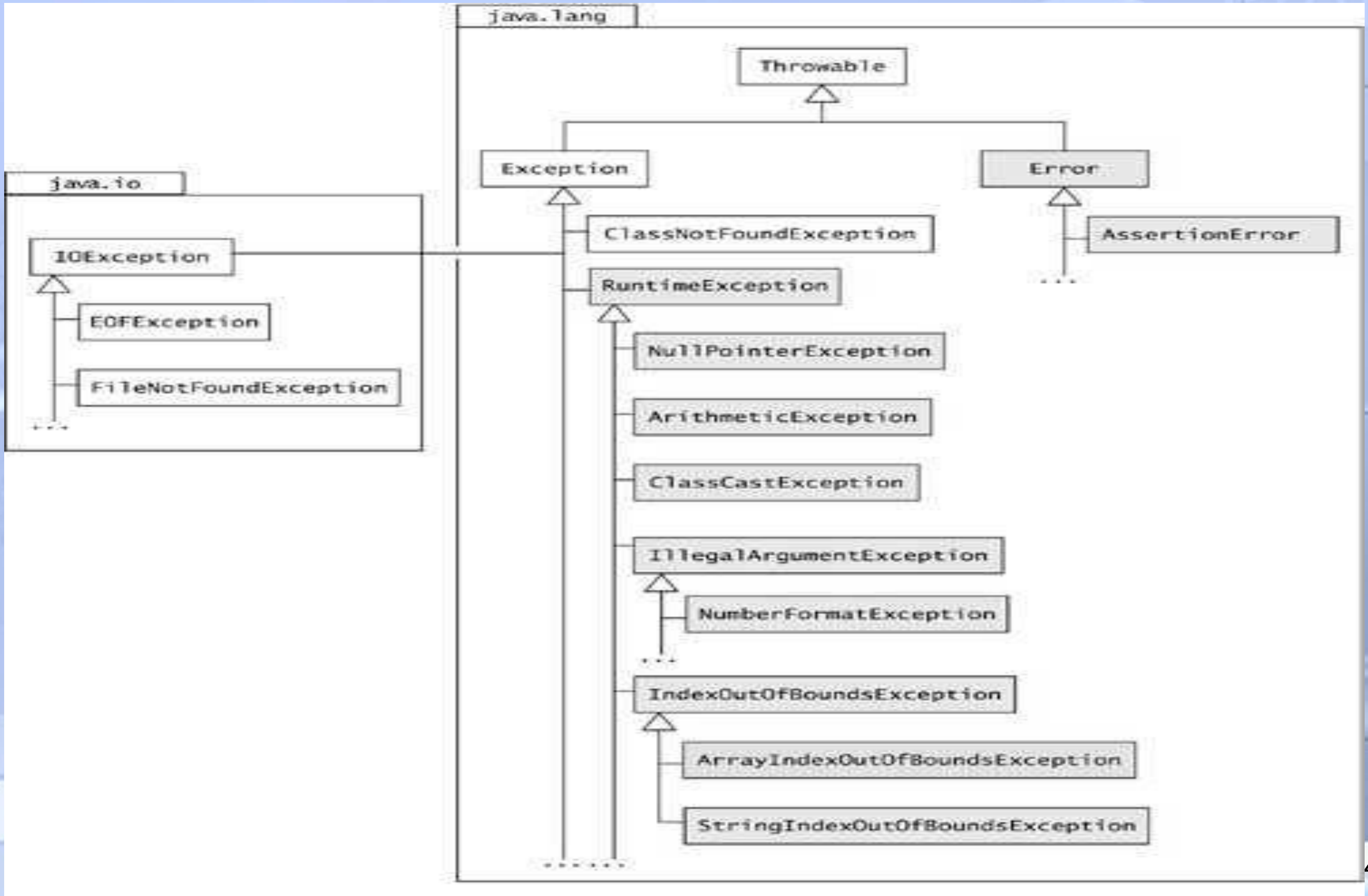
❖ All these exceptions are checked exception.

Error classes

Some subclasses of Error are

- AssertionError
- ThreadDeathError
- LinkageError
- VirtualMachineError

Exception Hierarchy



Classes that are shaded (and their subclasses) represent unchecked exceptions.

Throwable Class

- The Throwable class is the superclass of all errors and exceptions in the Java language.
- Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.
- Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

Throwable Class (cont.)

- Field

  String message;

  - to provide the detail message

- Methods

  String toString()

  - return the short description of the exception

  String getMessage()

  - return the detail message

  void printStackTrace()

  - print the stack trace on the standard error stream

Checked and Unchecked Exceptions

Checked Exception

- The compiler ensures that if a method can throw a checked exception directly or indirectly, then the method must explicitly deal with it.

- The method must either catch the exception and take the appropriate action, or pass the exception on to its caller.

❖ All exceptions are checked exception except for **RuntimeException**, **Error** and their subclasses.

Checked and Unchecked Exceptions

Unchecked Exception

- Exceptions defined by Error and RuntimeException classes and their subclasses are known as unchecked exception, meaning that the method is not obliged to deal with this knid of exceptions.

- They are either irrecoverable and the program should not attempt to deal with them or they are programming errors.

Lets make exercise!

```
public class MyClass {
    public static void main(String[] args) {
            int k=0;
            try {
                    int i = 5/k;
              } catch (ArithmeticException e) {
                    System.out.println("1");
              } catch (RuntimeException e) {
                    System.out.println("2");
                    return;
             } catch (Exception e) {
                    System.out.println("3");
             } finally {
                    System.out.println("4");   }
            System.out.println("5");
    }
}
```

!!! Answer !!!

- 1
- 4
- 5

Exercise 2

```
public class Exceptions {
    public static void main(String[] args) {
        try {
            if (args.length == 0) return;
                System.out.println(args[0]);
            } finally {
                System.out.println("The end");
            }
        }
}
```

If the program runs with no argument, what is the result???

!!! Answer !!!

- The end

Exercise 3

```
public class MyClass
{
    public static void main(String[] args)
    {
            RuntimeException re = null;
            throw re;
    }
}
```

!!! Answer !!!

```
Exception in thread "main" java.lang.NullPointerException
        at MyClass.main(MyClass.java:4)
```

Exercise 4

```
public class MyClass {
  public static void main(String[] args) {
     try {
             f();
         } catch (InterruptedException e) {
             System.out.println("1");
             throw new RuntimeException();
         } catch (RuntimeException e) {
             System.out.println("2");
             return;
         } catch (Exception e) {
             System.out.println("3");
        } finally {
             System.out.println("4");
         }
             System.out.println("5");
     }

static void f() throws InterruptedException
 {
     throw new
     InterruptedException("Time for
     break.");
      }
}
```

!!! Answer !!!

```
1
4
Exception in thread "main" java.lang.RuntimeException
at MyClass.main(MyClass.java:7)
```

Exercise 5

```
public class MyClass {
    public static void main(String[] args) throws InterruptedException {
        try {
            f();
            System.out.println("1");
        } finally {
            System.out.println("2");
        }
        System.out.println("3");
    }
// InterruptedException is a direct subclass of Exception.
    static void f() throws InterruptedException {
        throw new InterruptedException("Time to go home.");
    }
}
```

!!! Answer !!!

```
2
Exception in thread "main" java.lang.InterruptedException: Time to go home.
at MyClass.f(MyClass.java:13)
at MyClass.main(MyClass.java:4)
```

The Supercomputing People

Corporate Profile

Thank you!