Core Java

Exception Handling

1

Exception Handling

- A built-in mechanism for trapping & handling errors(Exceptional cases)
- Usually deals with abnormal events or code execution which prevents the program from continuing, like:
  - Array out of bounds accesses
  - Divide by Zero
  - Null pointers & so on…
- Exception Handling handles such cases whenever they happen

2

What is Exception? – Java Language

- An Exception is a Java class
- A variety of subclasses allows handling different kinds of errors & abnormal events
- Basic concept:
  - Whenever an abnormal event occurs, Java *throws* an Exception
  - It means Java instantiates a subclass of the Exception class
  - Whenever an Exception could possibly be thrown, we must provide a mechanism for *catching* it in our code
- Exception Handling Keywords:
  - **try, catch, throw, throws finally**

3

Exception handler Block

```
try {
                    // Code to be monitored for exceptions
}
catch(Exceptionclass object) //A try may have many catch blocks.
{
          //Exception Handler Block
}
finally
{
                    //code to be executed anyways
}
```

4

Catching Exceptions

- A *try* statement executes a block and oversees the execution of enclosed statements for exceptions
- try also defines the scope for exception handlers (defined in catch clause)
- A try block must be accompanied by at least one *catch* block or one *finally* block
- Any method declared as being able to throw an Exception, can have a try / catch block to handle the exception

5

Catching Exceptions (Contd…)

```
try {
    String text = "text";
    System.out.println(text.charAt(10));
} catch(IndexOutOfBoundsException e) {
    System.err.println("Index out of bounds");
    e.printStackTrace();
}
```

- If an Exception is thrown inside of a try block, the returned exception is forwarded as an argument to the catch block where the Exception can be handled

6

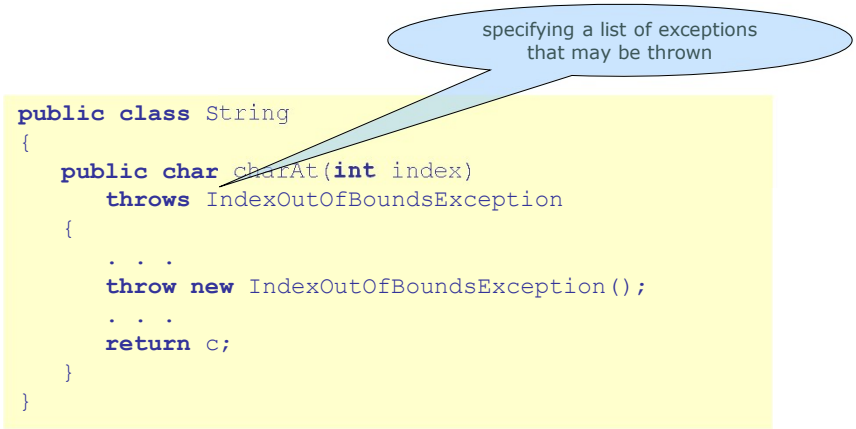
Throwing Exceptions

- If the programmer does not catch the exception, it is thrown automatically to the caller function
- If an exception is thrown from the *main* function, the program is terminated abnormally

7

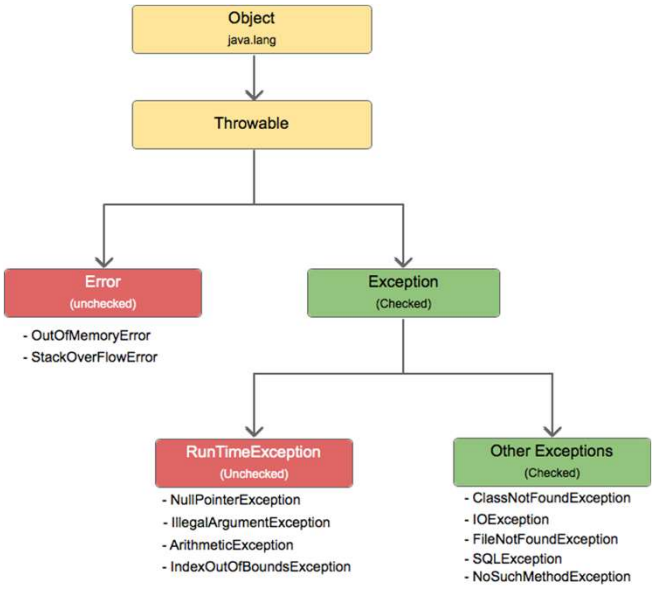
Throwing Exceptions (Contd…)

- Exceptions may be thrown explicitly by using the *throws* keyword
- Throwing exceptions in Java terminates method execution



8

Exception Hierarchy



9

Categories of Exceptions

- Java exceptions fall in two categories:

1. Unchecked
   - Not checked by the compiler at compile time
   - Does not force the client program / method to declare each exception thrown by a method, or even handle it
   - All exceptions are derived from *RuntimeException* class

2. Checked
   - Checked by the compiler to see if these exceptions are properly caught or specified, & if not, the code fails to compile
   - Forces client program to deal with the scenario in which an exception may be thrown
   - All exceptions which are not derived from RuntimeException class

10

Dealing with Exceptions

1. By using a try / catch block as seen

2. By indicating that the *calling method* throws the same Exception, essentially forwarding the responsibility of catching the exception to the code that calls your method

```
public void myMethod() throws IOException
{
   //calls a method that throws an IOException
}
```

11

Multiple Catch Blocks

- A method can throw more than one possible Exceptions, or the try block could call two different methods that throw two different Exceptions

```
try {
   String text = "text";
   System.out.println(text.charAt(10));
   int n = Integer.parseInt("abc");
} catch(IndexOutOfBoundsException e) {
   System.err.println("Index out of bounds");
   e.printStackTrace();
} catch(NumberFormatException e) {
   System.err.println("bad number");
   e.printStackTrace();
}
```

12

Multiple Catch Blocks (Contd…)

- Since all Exceptions are subclasses of the Exception class, we can generalize catch blocks to accept multiple different types of Exceptions by using a super class

```
try {
   String text = "text";
   System.out.println(text.charAt(10));
   int n = Integer.parseInt("abc");
} catch(Exception e) {
   System.err.println("Something bad happened");
   e.printStackTrace();
}
```

13

The *finally* Block

- Sometimes, while in a try / catch block, an Exception could be thrown before some important code at the end of the try block
- The *finally* block can be used to run this code
- Code in *finally* always executes (even in case of unhandled exceptions)

```
try {
   String text = "text";
   System.out.println(text.charAt(10));
} catch(IndexOutOfBoundsException e) {
   System.err.println("Index out of
bounds");
   e.printStackTrace();
} finally {
   //important code
}
```

14

Rethrowing Exceptions

- We can *rethrow* an exception after catching it & processing it

```
try {
    String text = "text";
    System.out.println(text.charAt(10));
} catch(IndexOutOfBoundsException e) {
    System.err.println("Index out of bounds");
    e.printStackTrace();
    throw e;
}
```

- If we *rethrow* an Exception, we must specify that the calling method throws the Exception

15

Exception Methods

- What type of information do we get from the Exception objects:
  - getCause()
  - getMessage()
  - printStackTrace()
- Subclasses of Exception can be much more elaborate and contain more information if desired

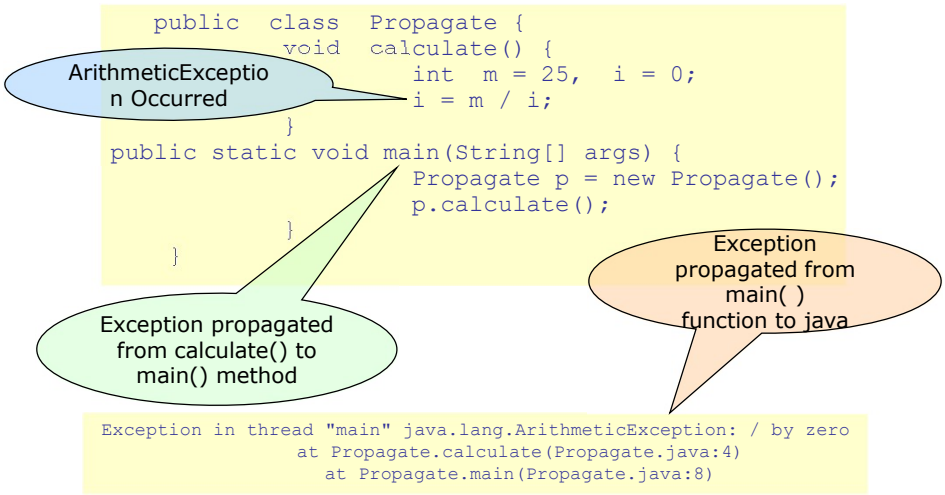
16

Exception Propagation

- Exceptions are always propagated from the *called* method to the *caller* method, if thrown from the *called* method
- If an Exception is thrown from the *main()* method, it will be propagated to the Java Runtime
- In exception propagation, all statement executions are ignored until finding the exception handler

17

Exception Propagation (Contd…)



18

Resource Management (traditional approach)

```
public static void performIO()throws IOException{
        FileInputStream is=null;
        FileOutputStream os=null;
        try{
                is=new FileInputStream(new File("readfile"));
                //.....
                os=new FileOutputStream(new File("writefile"));
                //.....
        }finally{
                if(is!=null) //there is a possibility of not getting
                        //instantiated.
                is.close();
                if(is!=null) //there is a possibility of not getting
                        //instantiated.
                os.close();
        }
}
```

19

Resource Management (Starting java 7 approach)

```
public static void performIO()throws IOException{
        try(FileInputStream is=new FileInputStream(new File("readfile"));
        FileOutputStream os=new FileOutputStream(new File("writefile"));
        ){
        //.....
        //.....
        }
}
```

20

User Defined Exceptions

- A User Defined Exception must be a subclass of Exception or one of its subclasses

```
class AgeException extends Exception
{
 public AgeException(String message)
 {
  super(message);
 }
}
```

```
class Employee
{
 public void setAge(int age) throws AgeException
 {
  if(age<18)
   throw new AgeException("Age must be > 18");
 }
}
```

21

Thank You

22