Object Oriented Programming

Instructor Name:

Lecture-17

Today's Lecture

- **Defensive Programming**
- **Exception Handling**

Defensive Programming

What is Defensive Programming?

- In a Client – Server Interaction where a server object only perform activities on client request.
- While writing server class, a programmer can adopt two possible ways
  1. They can assume that client objects will know what they are doing and will request services only in a sensible and well-defined way.
  2. They can assume that server objects will operate in an essentially problematic environment in which all possible steps must be taken to prevent client objects from using them incorrectly.
- Extreme views
- Practically, scenario lies some where in between
- A third possibility, of course, is an intentionally hostile client who is trying to break or find a weakness in the server.

Defensive Programming

Questions to Discuss

- **Different views - a useful base to discuss questions such as:**
  - **How much checking should a server's methods perform on client requests?**
  - **How should a server report errors to its clients?**
  - **How can a client anticipate failure of a request to a server?**
  - **How should a client deal with failure of a request?**
- **So far we have written all the classes with implicit trust that client will use these classes appropriately**

Defensive Programming

Parameter Checking

- A server object is most vulnerable when its constructor and methods receive external values through their parameters.
- The values passed to a constructor are used to set up an object's initial state, while the values passed to a method will be used to influence the overall effect of the method call and may change the state of the object and a result the method returns.
- It is vital that a server object knows whether it can trust parameter values to be valid or whether it needs to check their validity for itself.

Defensive Programming

Server Error Reporting

- Having protected a server object from performing an illegal operation through bad parameter values is all that the server writer needs to do.
- Ideally we should like to avoid such error situations from arising in the first place.
- Furthermore, it is often the case that incorrect parameter values are the result of some form of programming error in the client that supplied them.
- Therefore, rather than simply programming around the problem in the server and leaving it at that, it is good practice for the server to make some effort to indicate that a problem has arisen, either to the client itself or to a human user or programmer.
- Here we see three different nature of users

Defensive Programming

Notifying User

- Print an error message
- Two Problems with this Approach
  1. They assume that the application is being used by a human user who will see the error message. There are many applications that run completely independently of a human user. An error message, or an error window, will go completely unnoticed.
  2. Even where there is a human user to see the error message, it will be rare for that user to be in a position to do something about the problem.
- Programs that print inappropriate error messages are more likely to annoy their users rather than achieve a useful outcome.
- Except in a very limited set of circumstances, notifying the user is not a general solution to the problem of error reporting.

Defensive Programming

Notifying Client Object

- **There are two main ways to notify client object**

1. **A server can use a non-void return type (e.g boolean) of a method to return a value that indicates either success or failure of the method call.**
2. **A server can throw an exception if something goes wrong.**

Exceptions

What is an Exception?

- An Exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's Instructions.
- An exception can occur for many different reasons, below given are some scenarios where exception occurs.
  - A user has entered invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.
- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exceptions

Three Categories of Exception

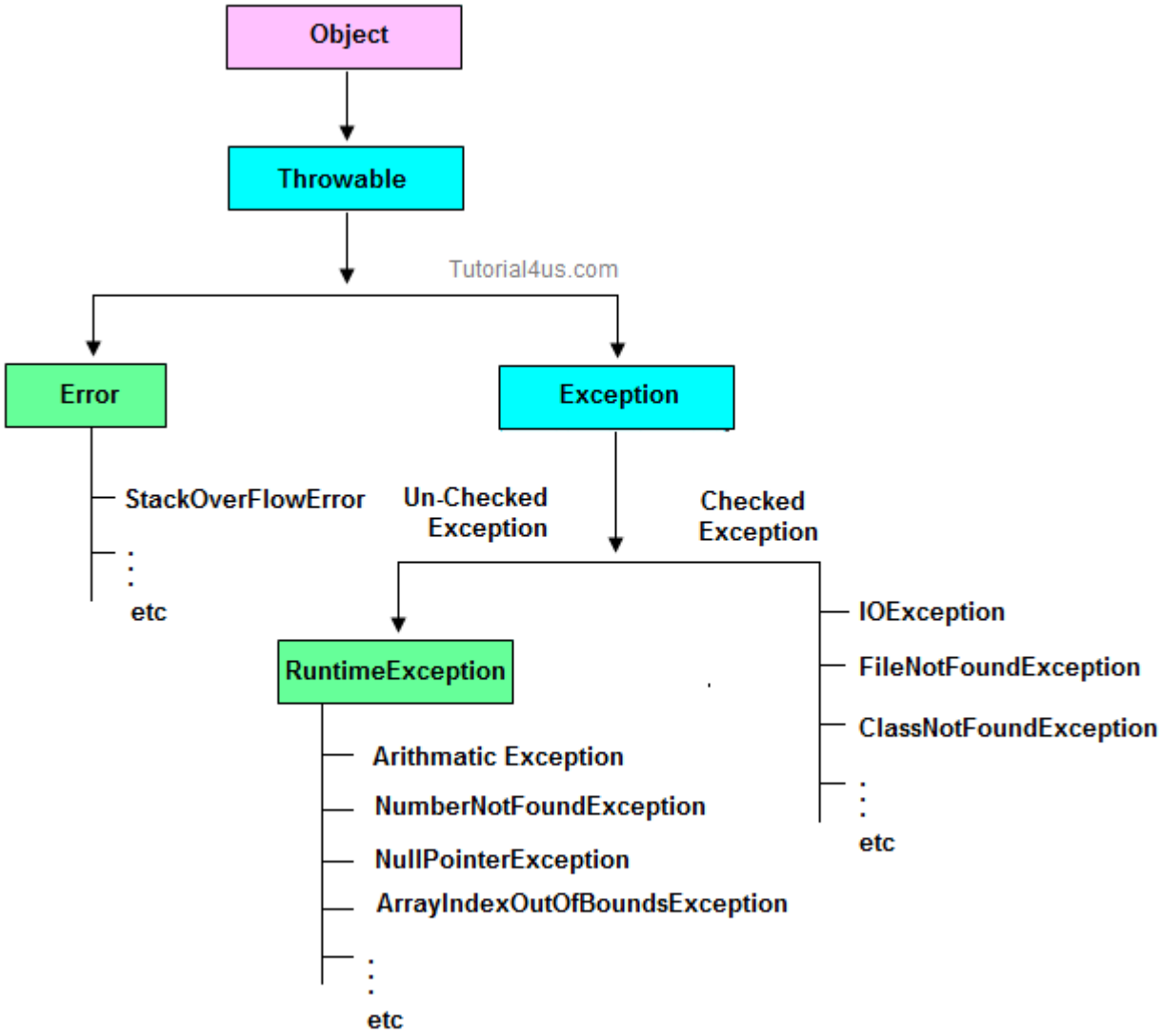
- **Checked Exception :** **A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the Programmer should take care of (handle) these exceptions.**
- **Unchecked Exception :** **An Unchecked exception is an exception that occurs at the time of execution, these are also called as Runtime Exceptions, these include programming bugs, such as logic errors or improper use of an API. runtime exceptions are ignored at the time of compilation.**
- **Error:** **These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.**

Exception Handling

What is Exception Handling?

- **The process of converting system error messages into user friendly error message is known as Exception handling.**
- **This is one of the powerful feature of Java to handle run time error and maintain normal flow of java application.**

Exception Hierarchy



Exception Handling

Methods with Description

- **public String getMessage()**

  Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor

- **public Throwable getCause()**

  Returns the cause of the exception as represented by a Throwable object.

- **public String toString()**

  Returns the name of the class concatenated with the result of getMessage()

- **public void printStackTrace()**

  Prints the result of toString() along with the stack trace to System.err, the error output stream.

There are more method, Google for better understanding of Exception

Exception Throwing

Throwing an Exception

- Throwing an exception is the most effective way a server object has of indicating that it is unable to fulfill a call on one of its methods.
- One of the major advantages this has over using a special return value is that it is (almost) impossible for a client to ignore the fact that an exception has been thrown and carry on regardless.
- Failure by the client to handle an exception will result in the application terminating immediately.
- In addition, the exception mechanism is independent of the return value of a method, so it can be used for all methods, irrespective of the return type
- An Exception is thrown using **throw statement**.

Exception Throwing

Throwing an Unchecked Exception Example

- Here getDetails method is throwing an exception to indicate that passing a nullvalue for the key does not make sense because it is not a valid key.

```
public ContactDetails getDetails(String key)
 {
 if(key == null){
     throw new IllegalArgumentException(
     "null key in getDetails");
 }
   return book.get(key);
 }
```

Exception Throwing

Throwing Exception Principles

- **Exception Mechanism is independent of the return value of method, so it can be used for all methods, irrespective of all return types**
- **The place where an error is discovered will be distinct from where recovery is attempted**
- **Discovery will be in server's method**
- **Recovery will be in the client**
- **If recovery were possible at the point of discovery then there would be no point in throwing an exception**

Exception Throwing

Throwing an Exception

- Two stages involve in throwing exception
  1. First, an exception object is created using the new keyword
  2. the exception object is thrown using the throw keyword
- When an exception object is created, a diagnostic string may be passed to its constructor.
- This string is later available to the receiver of the exception via the exception object's getMessage and toString methods.
- The string is also shown to the user if the exception is not handled and leads to the termination of the program.
- The exception type we have used here, `IllegalArgumentException`, is defined in the `java.lang` package and is regularly used to indicate that an inappropriate actual parameter value has been passed to a method or constructor.

Exception Throwing

Checked vs Unchecked Exceptions

- **Checked exceptions are intended for cases where the client should expect that an operation could fail (for example, if it tries to write to a disk, it should anticipate that the disk could be full).**
- **In such cases, the client will be forced to check whether the operation was successful.**
- **Unchecked exceptions are intended for cases that should never fail in normal operation—they usually indicate a program error.**
- **For instance, a programmer would never knowingly try to get an item from a position in a list that does not exist, so when they do, it elicits an unchecked exception.**

Exception Throwing

The Effect of an Exception

- **Two Effects**
  1. **the effect in the method where the problem is discovered and the exception is thrown**
  2. **the effect in the caller of the problem method**
- **When an exception is thrown, the execution of the current method finishes immediately; it does not continue to the end of the method body**
- **Consider the following contrived call to getDetails:**

```
String phone = details.getPhone();
```

- **The execution of these statements will be left incomplete**

Exception Throwing

Preventing Object from Creation

- An important use for exceptions is to prevent objects from being created if they cannot be placed in a valid initial state
- This will usually be the result of inappropriate parameter values being passed to a constructor.
- The process of throwing an exception from a constructor is exactly the same as throwing one from a method
- An exception thrown from a constructor has the same effect on the client as an exception thrown from a method.

Exception Handling

Checked Exceptions – the throws clause

- The first requirement of the compiler is that a method throwing a checked exception must declare that it does so in a `throws` clause added to the method's header.

```
public void saveToFile(String destinationFile)
   throws IOException
```

- It is permitted to use a throws clause for unchecked exceptions, but the compiler does not require one

Exception Handling

Checked Exceptions – the try statement

- Second requirement, when using checked exceptions
- Method that throws a checked exception must make provision for dealing with the exception
- Use try statement to make a try block
- Syntax

```
try{
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
```

Exception Handling

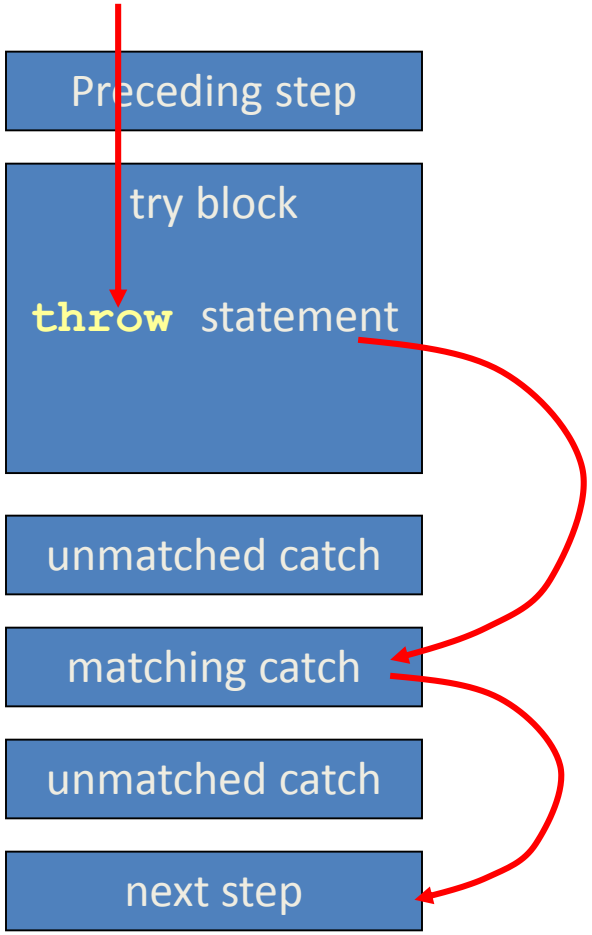
Checked Exceptions – the catch statement

- **catch block catches all exceptions of its type and subclasses of its type**
- **If there are multiple catch blocks that match a particular exception type, only the first matching catch block executes**
- **Makes sense to use a catch block of a superclass when all catch blocks for that class's subclasses will perform same functionality**

Checked Exceptions – the finally Block

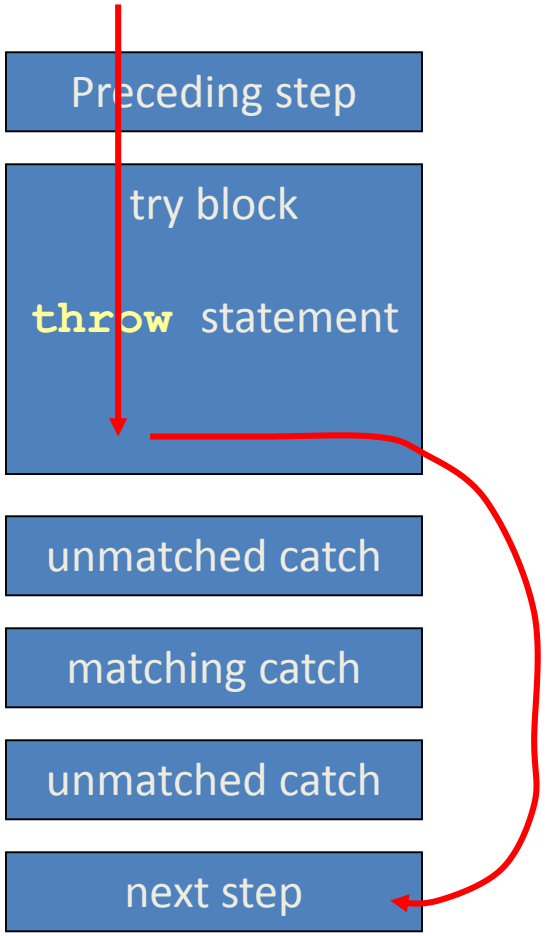
- Consists of finally keyword followed by a block of code enclosed in curly braces
- Optional in a try statement
- If present, is placed after the last catch block
- Executes whether or not an exception is thrown in the corresponding try block or any of its corresponding catch blocks
- Will not execute if the application exits early from a try block via method System.exit
- Typically contains resource-release code

Sequence of Events for throw



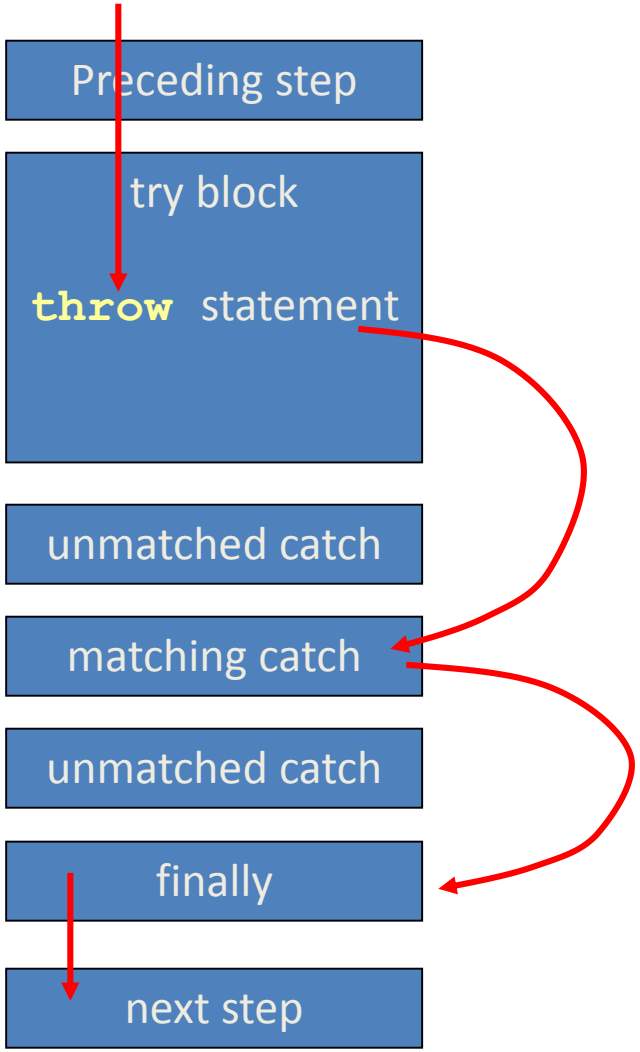
Exception Handling

Sequence of Events for No throw



Exception Handling

Sequence of Events for finally clause



Exception Handling

Exceptional Flow of Control

- **Exceptions break the normal flow of control.**
- **When an exception occurs, the statement that would normally execute next is not executed.**
- **What happens instead depends on:**
  - **whether the exception is caught,**
  - **where it is caught,**
  - **what statements are executed in the 'catch block',**
  - **and whether you have a 'finally block'.**

Exception Handling

Catching a Exception Complete Template

```
try {  // statement that could throw an exception
    }
catch (<exception type> e) {
   // statements that handle the exception
}
catch (<exception type> e) { //e higher in hierarchy
   // statements that handle the exception
}
finally {
   // release resources
}
//other statements
```

- At most one catch block executes
- finally block always executes once, whether there's an error or not

Exception Handling

Compile Time Exception Example

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo {
   public static void main(String args[]){
      File file=new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

C:\>javac FilenotFound_Demo.java FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown FileReader fr = new FileReader(file); ^ 1 error

Exception Handling

Run Time Exception Example

```
public class Unchecked_Demo {

   public static void main(String args[]){

        int num[]={1,2,3,4};
        System.out.println(num[5]);

   }

}
```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)

Exception Handling

Catching a Run Time Exception

```
public class Exception{
   public static void main(String aa[]){
   try{
        int a[]=new int[1];
        System.out.println(a[2]);
   }
   catch(Exception e){
        System.out.println("Exception Caught");
   }
   finally{
        System.out.println("Finally Block");
   }
   System.out.println("After Exception");
   }
}
```

Exception Handling

What may be the behavior of the previous slide program if exception was not handled?

Exception Handling – Multiple Catch Blocks

```
public class Exception{
   public static void main(String aa[]){
   int a[]=new int[5];
   try{
       System.out.println(a[1]/0);
   }
   catch(ArithmeticException e){
       System.out.println("Divided By Zero");
   }
   catch(NullPointerException e){
       System.out.println("Null Pointer Excdption");
   }
   catch(ArrayIndexOutOfBoundsException e){
       System.out.println("Index Out of Bound");
   }
       System.out.println("After Exception");
   }
}
```

RECAP