Core Java

Exception Handling

Lesson Objectives

- On completion of this lesson, you will be able to:
  - Explain the concept of Exception
  - Describe types of Exceptions
  - Handle Exception in Java
  - Create your own Exceptions
  - State best practices on Exception

Why is exception handling used?

- No matter how well-designed a program is, there is always a chance that some kind of error will arise during its execution, for example:
  - ➢ Attempting to divide by 0
  - ➢ Attempting to read from a file which does not exist
  - ➢ Referring to non-existing item in array
- An exception is an event that occurs during the execution of a program that disrupt its normal course.

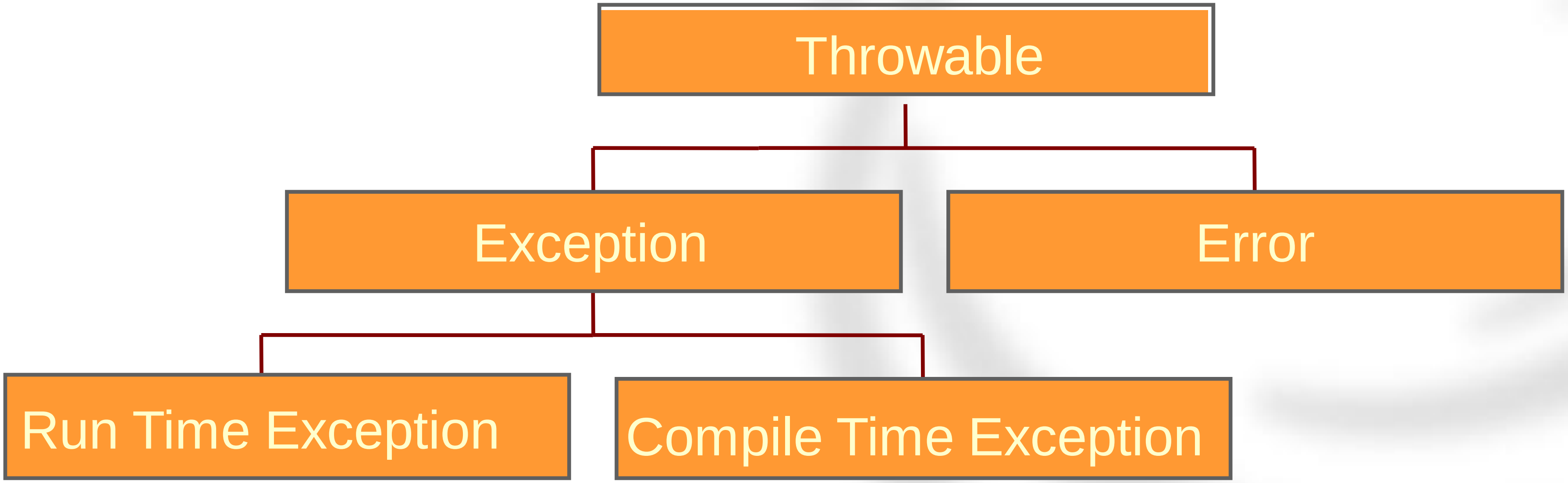
5.1: Exception Handling - Fundamentals

Exception Handling

- Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions:
  - Examples: Hard disk crash; Out of bounds array access; Divide by zero, and so on
- When an exception occurs, the executing method creates an Exception object and hands it to the runtime system – ”throwing an exception”
- The runtime system searches the runtime call stack for a method with an appropriate handler, to catch the exception.

5.2: Exception Types

Hierarchy of Exception Classes



Throwable

Exception

Error

Run Time Exception

Compile Time Exception

Unchecked Exception

Checked Exception

5.2.1: Exception Types -> Errors

Error

- An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch.
- Most such errors are abnormal conditions.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
  - Stack overflow is an example of such an error.

Exception

- The **Exception** class and its subclasses are a form of **Throwables.** They indicate conditions, which a reasonable application may want to catch.
- Two Types:
  - Checked Exception
  - UnChecked Exception

Checked/Compile Time Exceptions

Characteristics of **Checked Exceptions:**

- They are checked by the compiler at the time of compilation.
- They are inherited from the core Java class Exception.
- They represent exceptions that are frequently considered "non-fatal" to program execution.
- They must be handled in your code, or passed to parent classes for handling.
- Some examples of **Checked exceptions** include:
  - **IOException, SQLException, ClassNotFoundException**

5.2.2.1: Exception Types -> Exceptions -> UnChecked Exceptions

Unchecked/Runtime Exceptions

- **Unchecked** exceptions represent error conditions that are considered “fatal” to program execution.
- **Runtime** exceptions are exceptions which are not detected at the time of Compilation.
- They are encountered only when the program is in execution.
- It is called **unchecked exception** because the compiler does not check to see if a method handles or throws these exceptions.

- Only Notes Page

Keywords for handling Exceptions

- try
  - This marks the start of a block associated with a set of exception handlers.
- catch
  - The control moves here if an exceptions is generated.
- finally
  - This is called irrespective of whether an exception has occurred or not.
- throws
  - This describes the exceptions which can be raised by a method.
- throw
  - This raises an exception to the first available handler in the call stack, unwinding the stack along the way.

Why to handle exceptions?

- Without Exception handling

```
class WithoutException {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
System.out.println(“Will not be printed”); } }
```

- With Exception handling

```
class WithExceptionHandling {
public static void main(String args[]) {
int d=0, a;
try {
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) {System.out.println("Division by zero."); }
System.out.println("This will get printed"); } }
```

5.3.1: Handling Exceptions ->Using try and catch

Try and Catch

- The **try** structure has three parts:
  - The **try** block
    - Code in which exceptions are thrown
  - One or more **catch** blocks
    - To respond to various types of Exceptions
  - An optional **finally** block
    - Code to be executed last under any circumstances
- The **catch** Block:
  - If a line in the **try** block causes an exception, program flow jumps to the **catch** blocks.
  - If any **catch** block matches the exception that occurred that block is executed.

5.3.1: Handling Exceptions

Using Try and Catch

```
try {
            // code in which exceptions may be thrown
  } catch (ExceptionType1  identifier) {
            // code executes if an ExceptionType1 occurs
  } catch (ExceptionType2 identifier) {
            // code executes if an ExceptionType2 occurs
  } finally  {
            // code executed last in any case
  }
```

Multiple Catch Blocks

- If you include multiple **catch** blocks, the order is important.
- You must catch subclasses before their ancestors.

```
public void divide(int x,int y)
{
  int ans=0;
  try{
           ans=x/y;
  }catch(Exception e) { //handle }
  catch(ArithmeticException f) {//handle}  //error
```

5.3.3: Handling Exceptions ->Nested try catch

Nested Try Catch Block

```
try {
    int a = arg.length;
    int b = 10 / a;
    System.out.println("a = " + a);
  try {
    if(a==1)
    a = a/(a-a);
    if(a==2) {
      int c[] = { 1 };
      c[42] = 99;
            }
      } catch(ArrayIndexOutOfBoundsException e) {
       System.out.println("Array index out-of-bounds: " + e); }
 } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e); }
```

5.3.4: Handling Exceptions ->finally

The Finally Clause

- The **finally** block is optional.
- It is executed whether or not exception occurs.

```
public void divide(int x,int y)
{
  int ans;
  try{
  ans=x/y;
  }catch(Exception e) {  ans=0;  }
  finally{  return ans;  // This is always executed }
}
```

Throwing an Exception

- You can throw your own runtime errors:
  - To enforce restrictions on use of a method
  - To "disable" an inherited method
  - To indicate a specific runtime problem
- To throw an error, use the **throw** Statement
  - throw ThrowableInstance
- **ThrowableInstance** is any **Throwable** Object

Throwing an Exception

```
class ThrowDemo {
void proc() {
try {
  throw new ArithmeticException(“From Exception”);
} catch(ArithmeticException e) {
  System.out.println("Caught inside demoproc.");
  throw e; // rethrow the exception
} }
public static void main(String args[]) {
ThrowDemo t=new ThrowDemo();
try {
  t.proc();
} catch(ArithmeticException e) {
System.out.println("Recaught: " + e); } } }
```

Using The Throws Clause

- If a method might throw an exception, you may declare the method as “throws” that exception and avoid handling the exception yourself.

```
class ThrowsDemo {
public static void main(String args[]) {
try {
doWork();
} catch (ArithmeticException e) {
System.out.println("Exception: " + e.getMessage()); } }
static void doWork() throws ArithmeticException {
int array[] = new int[100];
array[100] = 100; } }
```

5.4: Creating Application specific Exception

User specific Exception

- Write a class that extends(indirectly) Throwable.
- What Superclass to extend?
  - For unchecked exceptions: RuntimeException
  - For checked exceptions:
    - Any other Exception subclass or the Exception itself

```
class AgeException extends Exception {
private int age;
AgeException(int a) {
age = a;
}
public String toString() {
return age+" is an invalid age"; } }
```

User Specific Exception (Contd.)

- To create exceptions:
  - Write a class that extends(indirectly) **Throwable.**
  - Which Superclass to extend?
    - For unchecked exceptions: **RuntimeException**
    - For checked exceptions: Any other **Exception** subclass or the exception itself

```
class AgeException extends Exception {
private int age;
AgeException(int a) {
age = a;
}
public String toString() {
return age+" is an invalid age"; } }
```

5.6: Best Practices on Exception Handling

The Best Practices

- Avoid empty catch blocks.
- Avoid throwing and catching a generic exception class.
- Pass all the pertinent data to exceptions.
- Use the **finally** block to release the resources

Best Practices

- Avoid throwing unnecessary exceptions.
- Finalize method is not reliable.
- Exception thrown by finalizers are ignored.
- While using method calls, always handle the exceptions in the method where they occur.
- Do not use loops for exception handling.

Review – Match the Following format

1. CheckedException	A. Compulsory to use if a method throws a checked exception and doesn’t handle it
2. finally	B. Inherited from RuntimeException
3. throws	C. Can have any number of catch blocks
4. Unchecked Exception	D. Used to avoid “resource leak”
5. try	E. Inherited from Exception