Exception Handling

When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully? This is the subject we will introduce in today's session.

Exceptional Event (Exception)

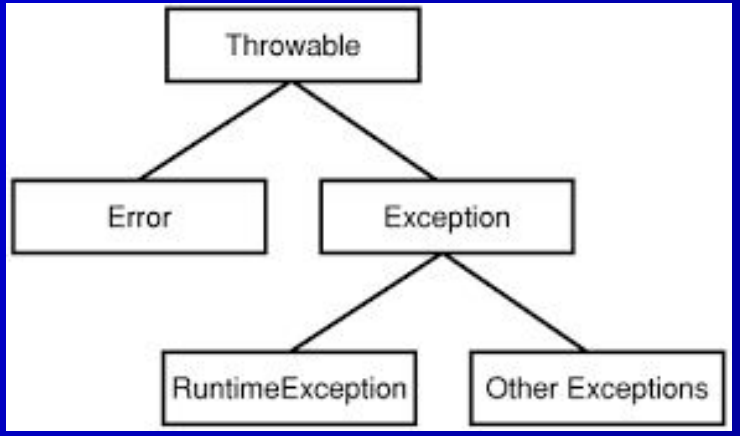
- An *exception* is an event, which occurs during the runtime or execution of a program, that disrupts the normal flow of the program's instructions.
- When an exceptional condition arises in a method, an object representing that exception is created and thrown to the caller of the method , in any case, it is not caught (handled gracefully), that caused the error in the program.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exceptions

- An exception should be ***caught & processed***, otherwise, the program quits abnormally.
- Exceptions thrown by JVM relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via five keywords: ***try, catch, throw, throws, and finally***.

Exception Types

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Exception-Handling Overview

```
Public static void test() {
   int a = 15, b = 0, c;
   c = a/b;
}
```

```
Public static void test(){
   int a = 15, b = 0, c;
   if (b!=0) c = a/b;
}
```

Catching/Handling Exception

```
Public static void test() {
   int a = 15, b = 0, c;
   try {
      c = a/b;
   } catch (Exception x) {
      System.out.println(x);
   }
}
```

Declaring, Throwing, and Catching Exceptions

```
Public static void test() throws ArithmeticException {
    int a = 15, b = 0, c;
    if (b==0)
    throw new ArithmeticException("Zero Divisor");
    c = a/b;
}
```

```
Public static void caller(){
    try {
        test();
    } catch (Exception x) { … … … }
}
```

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
  throws IOException

public void myMethod()
  throws IOException, OtherException
```

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```

Throwing Exceptions Example

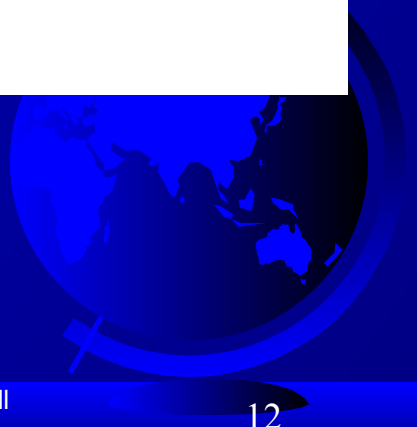
```
/** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

Rethrowing Exceptions

```
try {
  statements;
}
catch(TheException ex) {
  perform operations before exits;
  throw ex;
}
```



The `finally` Clause

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

When to Throw Exceptions

- An exception occurs in a method. If you want the exception to be *processed by its caller*, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.

When to Use Exceptions

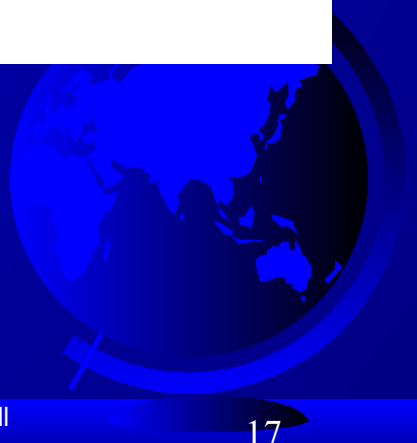
When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

When to Use Exceptions

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```



Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

```
// This program creates a custom exception type.
class GreaterThanTenExp extends Exception {
    private int detail;
    GreaterThanTenExp(int a) {
        detail = a;
    }
    public String toString() {
        return "GreaterThanTenExp[" + detail + "]";
    }
}

public class ExceptionDemo {
    static void compute(int a) throws GreaterThanTenExp {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new GreaterThanTenExp(a);
        System.out.println("Normal exit");
    }
    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (GreaterThanTenExp e) {
            System.out.println("Caught " + e);
        }
    }
}
```

```
//Displays
Called compute(1)
Normal exit
Called compute(20)
Caught GreaterThanTenExp[20]
```