Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java : User defined exceptions/5b1e5d087becc0459deae2f2

TUTORIAL

Java : User defined exceptions

Topics

1.5 Assertion (JDK 1.4)

1.9 Pre-conditions of public methods

Inside the standard package java.lang, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed below:

```
ArithmeticException                    Arithmetic error, such as
divide-by-zero.
ArrayIndexOutOfBoundsException         Array index is out-of-
bounds.
ArrayStoreException                    Assignment to an array
element of an incompatible type.
ClassCastException                     Invalid cast.
IllegalArgumentException               Illegal argument used to
invoke a method.
IllegalMonitorStateException           Illegal monitor
operation, such as waiting on an unlocked thread.
IllegalStateException                  Environment or
application is in incorrect state.
IllegalThreadStateException            Requested operation not
compatible with current thread state.
IndexOutOfBoundsException              Some type of index is out-
of-bounds.
NegativeArraySizeException             Array created with a
negative size.
NullPointerException                   Invalid use of a null
```

```
reference.
NumberFormatException                    Invalid conversion of a
string to a numeric format.
SecurityException                        Attempt to violate
security.
StringIndexOutOfBounds                   Attempt to index outside
the bounds of a string.
UnsupportedOperationException            An unsupported operation
was encountered.
```

Following are those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself:

```
ClassNotFoundException                   Class not found.
CloneNotSupportedException               Attempt to clone an
object that does not implement the Cloneable interface.
IllegalAccessException                   Access to a class is
denied.
InstantiationException                   Attempt to create an
object of an abstract class or interface.
InterruptedException                     One thread has been
interrupted by another thread.
NoSuchFieldException                     A requested field does
not exist.
NoSuchMethodException                    A requested method does
not exist.
```

These are called checked exceptions.

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It

overrides the toString( ) method, allowing the description of the exception to be displayed using println( ).

Java

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

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

```

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the value of the exception. The ExceptionDemo class defines a method named compute( ) that throws a MyException object. The exception is thrown when compute( )'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Assertion (JDK 1.4)

JDK 1.4 introduced a new keyword called `assert`, to support the so-called *assertion* feature. Assertion enables you to *test your assumptions* about your program logic (such as pre-conditions, post-conditions, and invariants). Each assertion contains a boolean expression that you believe will be true when the program executes. If it is not true, the JVM will throw an `AssertionError`. This error signals you that you have an invalid assumption that needs to be fixed. Assertion is much better than using if-else statements, as it serves as proper documentation on your assumptions, and it does not carry performance liability in the production environment (to be discussed later).

The `assert` statement has two forms:

```
assert booleanExpr;
assert booleanExpr : errorMessageExpr;
```

When the runtime execute the assertion, it first evaluates the *`booleanExpr`*. If the value is `true`, nothing happens. If it is `false`, the runtime throws an `AssertionError`, using the no-argument constructor (in the first form) or *`errorMessageExpr`* as the argument to the constructor (in the second form). If an object is

passed as the *errorMessageExpr*, the object's `toString()` will be called to obtain the message string.

Assertion is useful in detecting bugs. It also serves to document the inner workings of you program (e.g., pre-conditions and post-conditions) and enhances the maintainability.

One good candidate for assertion is the switch-case statement where the programmer believes that one of the cases will be selected, and the default-case is not plausible. For example,

```java
class Main
{
  public static void main(String[] args)
  {
    char operator = '%';                 // assumed either '+', '-', '*', '/' only
    int operand1 = 5, operand2 = 6, result = 0;
    switch (operator)
    {
      case '+': result = operand1 + operand2; break;
      case '-': result = operand1 - operand2; break;
      case '*': result = operand1 * operand2; break;
      case '/': result = operand1 / operand2; break;
      default: assert false : "Unknown operator: " + operator;  // not plausible here
    }
    System.out.println(operand1 + " " + operator + " " + operand2 + " = " + result);
  }
}
```

Assertion, by default, are disabled to ensure that they are not a performance liability in the production environment. To enable assertion, use the runtime command-line option –`enableassertions` (or –ea).

In the above example, "`assert false`" always triggers an `AssertionError`. However, the output is different, depending on whether assertion is enabled or disabled.

```
> javac Main.java   // no option needed to compile
> java -ea Main     // enable assertion
Exception in thread "main" java.lang.AssertionError: %
        at Main.main(Main.java:11)
> java Main         // assertion disable by default
5 % 6 = 0
```

In the above example, since the "`assert false`" always triggers an `AssertionError`, you could choose to `throw` an `AssertionError`. "`throw`" is always enabled during runtime.

```
default: throw new AssertionError("Unknown operator: " +
operator);
```

Another usage of assertion is to assert "internal invariants". In other words, to assert the possible values of an internal variable. For example,

```
class AssertionTest
{
  public static void main(String[] args)
  {
    int number = -8;    // assumed number is not negative
    // This assert also serve as documentation
    assert (number >= 0) : "number is negative: " + number;
    // do something
    System.out.println("The number is " + number);
  }
}
```

Output:

```
> java -ea AssertionTest     // enable assertion
Exception in thread "main" java.lang.AssertionError: number is
negative: -8
        at AssertionTest.main(AssertionTest.java:7)

> java AssertionTest
The number is -8
```

Assertion can be used for verifying:

- Internal Invariants: Assert that a value is within a certain constraint, e.g., `assert x > 0`.

- Class Invariants: Assert that an object's state is within a constraint. What must be true about each instance of a class before or after the execution of a method? Class invariants are typically verified via `private boolean` method, e.g., an `isValid()` method to check if a `Circle` object has a positive radius.
- Control-Flow Invariants: Assert that a certain location will not be reached. For example, the `default` clause of a `switch-case` statement.
- Pre-conditions of methods: What must be true when a method is invoked? Typically expressed in terms of the method's arguments or the states of its objects.
- Post-conditions of methods: What must be true after a method completes successfully?

Pre-conditions of public methods

Assertion should not be used to check the validity of the arguments (pre-condition) passed into "public" method. It is because `public` methods are exposed and anyone could call this method with an invalid argument. Instead, use a `if` statement to check the argument and throw an `IllegalArgumentException` otherwise. On the other hand, `private` methods are under your sole control and it is appropriate to assert the pre-conditions. For example,

```
// Constructor of Time class
public Time(int hour, int minute, int second)
{
   if(hour < 0 || hour > 23 || minute < 0 || minute > 59 ||
second < 0 || second > 59)
   {
      throw new IllegalArgumentException();
   }
   this.hour = hour;
   this.minute = minute;
   this.second = second;
}
```