Exception Handling

Q1:What is an Exception?

1. An unwanted or unexpected event that disturbs normal flow of the program is called Exception.

Ex: bike break down exception, FileNotFoundException

1. As a good programming practice, compulsory we should handle Exception.
2. Handling an exception doesn’t mean reparing an exception. It means to provide alternative way to continue rest of the program normally.
3. The main objective of exception handling is graceful termination of the program.

Q2: What is Runtime stack mechanism?

For every thread, jvm will create a runtime stack. All the method calls performed by that thread will be stored in the corresponding stack.

Each entry in the stack is known as stack frame or activation record. Once the method terminates normally then the corresponding entry from the stack will be poped out. After completing all the method calls, just before terminating the thread, jvm destroys the corresponding stack.

Ex:

class Test

{

Public static void main(String args[])

{

. fun();

}

Public static void fun()

{

. gun();

}

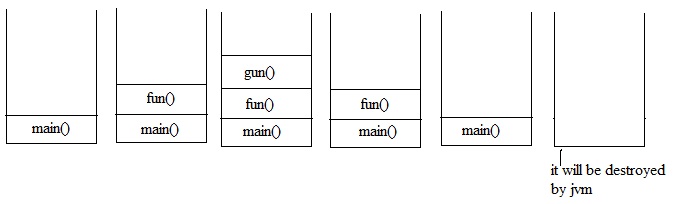
Public static void gun()

{

}

}

For the above example, internally the jvm created stack is the following



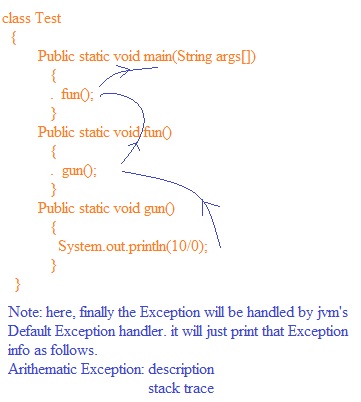
Q2:tell me about default exception handling?

1. Whenever an exception occurs, the method in which it is raised is responsible for the preparation of Exception object by including the following information.
2. Name of the exception
3. Description of the exception
4. Location where exception occurs(stack trace)
5. After creation of Exception object that method handovers it to the jvm.
6. Jvm will check for the required exception handling code in that method. If it is not available, then jvm terminates the method ubnormally and removes corresponding entry from the stack.
7. After that jvm searches for the exception handling code in the caller method, if it doesn’t contain any Exception handling code, then jvm terminates that method also abnormally and removes corresponding entry from the stack.
8. This process will be continue to main() method. If the main() method also doesn’t contain any exception handling code, then jvm terminates the code abnormally and removes corresponding entry from the stack.
9. Just before terminating the program abnormally jvm handovers the responsibility of exception handling to default Exception handler which is a component of jvm.
10. This default Exception handler just prints Exception information in the following format

Name of the Exception: description

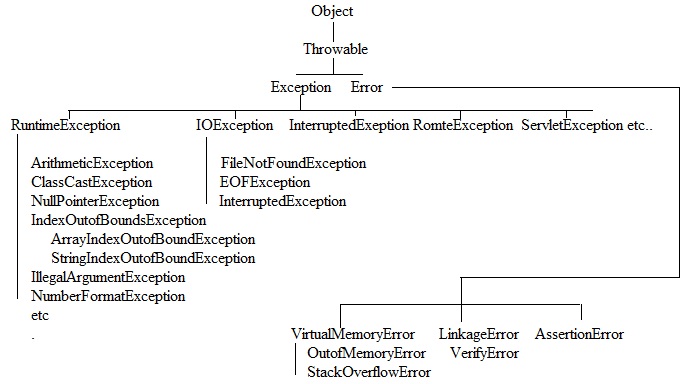
Stack trace

for the above concept, see the following diagram carefully



Q3: few points on Exception hierarchy

1. All exception and errors in java are child classes of Throwable either directly or indirectly. i.e java Exception hierarchy.
2. Javac Exception Hierarchy starts from Throwable.
3. Throwable contain two child classes. They are Exception and Error.
4. Exception: these are mostly caused by our program and these are recoverable.
5. Error: these are not caused by our program and non recoverable. These are due to lack of system resources.
6. See the following diagram

Q4:checked versus unchecked Exception

1. The Exceptions which are checked by compiler for smooth execution of program at runtime are called checked Exception.

Ex: InterruptedException

1. The Exceptions which are not checked by compiler are called unchecked Exception

Ex: ArithmeticException

1. Runtime Exception and its child classes, Error and its child classes are unchecked and all the remaining by default considered as checked Exception.
2. Whether Exception is checked or unchecked, it should occur at run time. There is no chance of occurring at compile time.

Q5:partially checked versus fully checked

1. A checked Exception is said to be fully checked when its every child class is also checked.

Ex: IOException, InterruptedException

1. A checked Exception is said to be partially checked when its child classes need not to be checked.

Ex: Exception, Throwable

Q6:how to handle a exception with try-catch?

We can handle exception by using try-catch, the risky code we have to place inside the try block and corresponding handling code we have to place inside catch block.

. syntax of try-catch

try

{

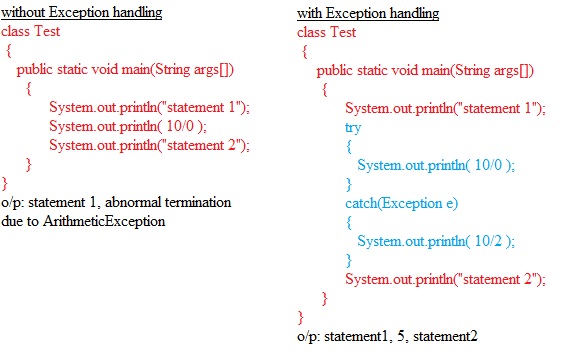
// risky code

}

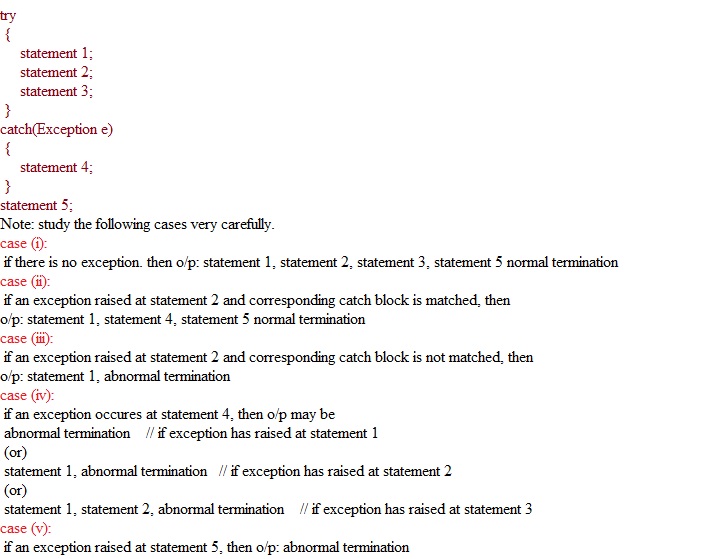
. catch

{

// corresponding handling code

}  


Q7:control flow in try-catch



Q8: what methods are there to print Exception information?

Throwable class defines the following three methods to display Exception information.

1. printStackTrace(): it prints Exception information in the following format.

Name of the exception: description

Stack trace

Ex: ArithmeticException: /by zero

At main()

1. toString(): it displays or it returns exception information in the following format.

Name of the exception: description

Ex: ArithmeticException: /by zero

1. getMessage(): it returns just description of the exception.

Ex: / by zero

Note: default exception handler alwayss uses printStackTrace() method.

Q9: try with multiple catch blocks

The way of handling an exception is varied from Exception to Exception. Hence for every Exception type we have to maintain a separate catch block i.e try with multiple catch block is possible and recommended to use.

If try with multiple catch block present then the order of catch block is always important. It should be from child to parent. Otherwise we will get compile time error.

Ex: see the following valid try with multiple catch blocks.

. try

{

System.out.println( 10/0 );

}

. catch(ArithmeticException e)

{

}

. catch(Exception e)

{

}

Ex: see the following invalid try with multiple catch blocks.

. try

{

System.out.println( 10/0 );

}

. catch(Exception e)

{

}

. catch(ArithmeticException e)

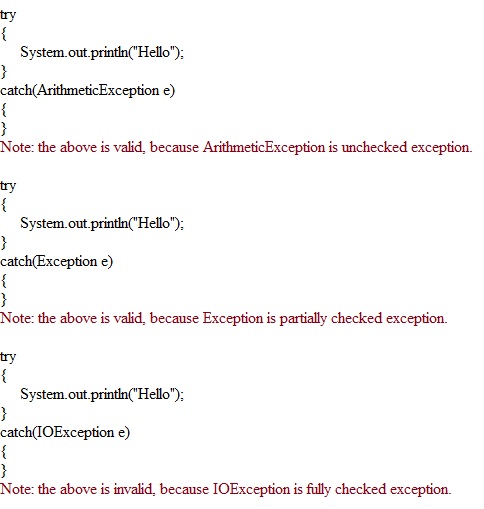
{

}

Here, compile time error will be raised due to not maintain the order.

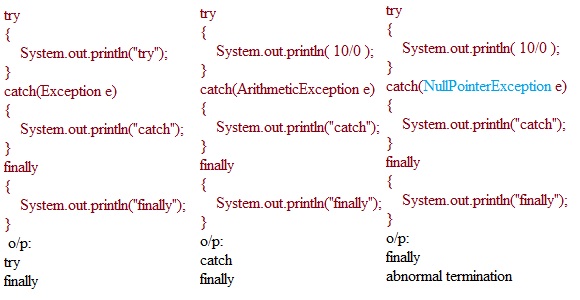
Q10: important point

if there is no chance of raising an exception in try then we are not allowed to define catch block, otherwise we will get compile time error. But this rule is applicable for fully checked exceptions only, but not for unchecked, partially checked.

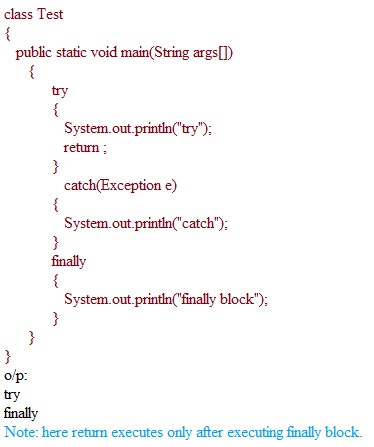


Q11:tell me about finally block?

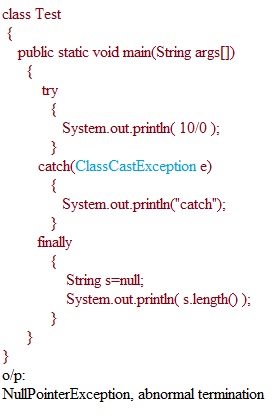
1. Generally, it is not recommended to keep clean-up code inside try, because if any exception is raised, then control skips the try block code.
2. It is not recommended to keep clean-up code inside catch block, because if no exception is raised, then control voilates the catch block code.
3. So we require a place to keep clean-up code which executes always irrespective of whether exception is raised or not. Such type of place is nothing but finally block.
4. Hence the main objective of finally block is to maintain clean-up code which should be executed always irrespective of exception raising or not.
5. For the above concept, see the following example



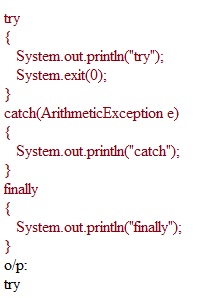
1. With in the try or in catch, if there is any return statement, that return statement will be executed only after executing finally block. See the following example



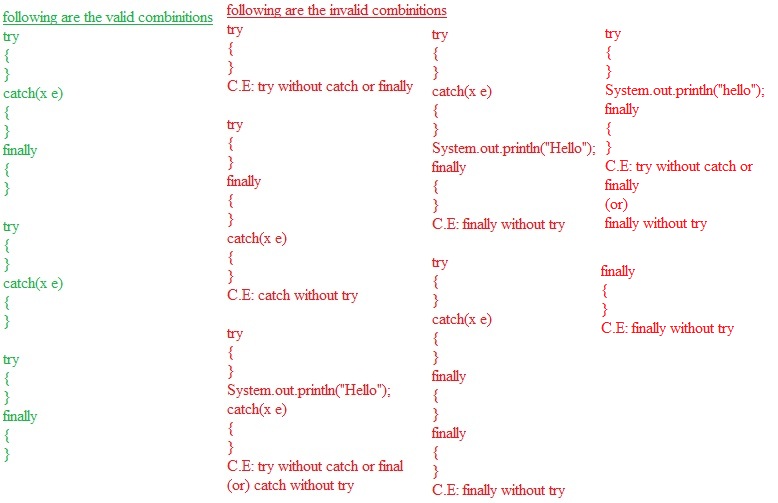
1. Finally dominates return statement. See the following example



1. There is only one situation, where the finally block won’t execute, that occures whenever we use System.exit(0), then there is no chance of executing finally block. See the following example



Q12: possible and impossible combinations of try,catch, finally



Q13: tell me the difference between final, finally and finalize() method?

final

1. Final is the variable applicable for variable, method and class.if a variable declared as final then reassignment is not possible because it is constant.
2. If a method is declared as final then overriding that method in child class is not possible.
3. If a class is declared as final, then we can not create child class for it.

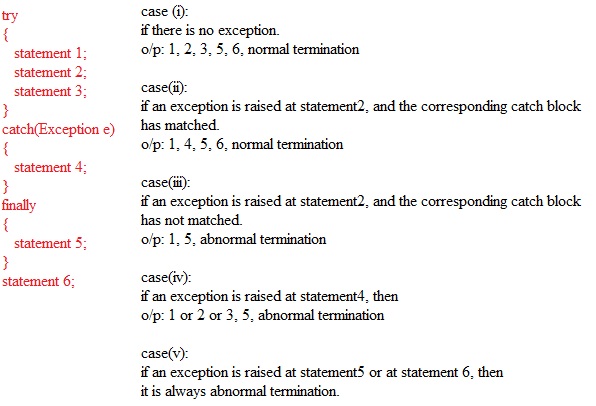
Finally

1. it is a block, always associated with try, catch to maintain clean-up code.
2. Finally block is executed always irrespective of whether exception is raised or not or handled or not handled.

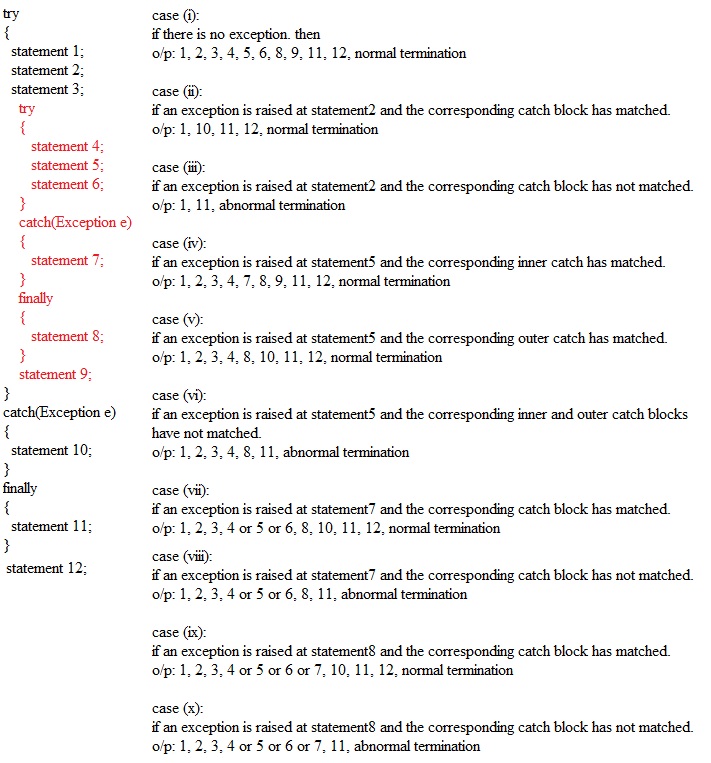
Finalize()

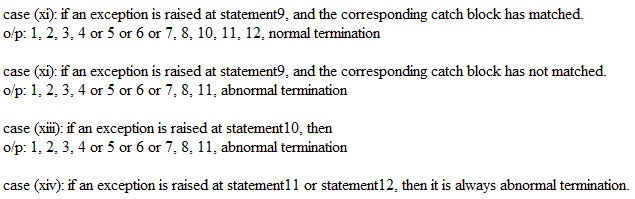
1. It is a method, always executed by garbage collector just before destroying any object in order to perform clean-up activities.
2. When compared with finalize(), finally is always recommended to maintain clean-up code, because we can’t expect exact behavior of the garbage collector.

Q14: control flow in try,catch,finally



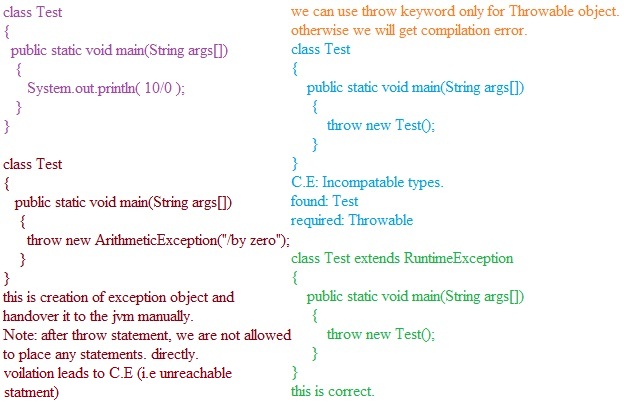
Q15: control flow in nested try, catch, finally

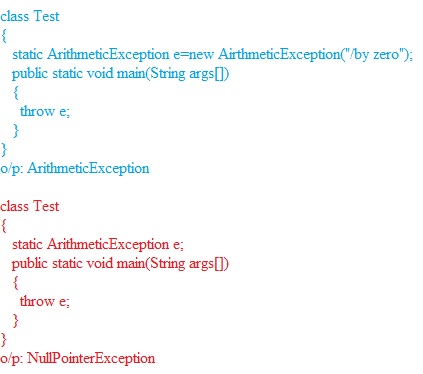




Q16: tell me about throw keyword

Sometimes we can create Exception object explicitly based on our requirement. We can handover the explicitly created object by using throw keyword.





Q17: tell me about throws keyword?

in our code, if there is any chance of raising checked exception then compulsory we should handle it either by try-catch or by using throws keyword. Otherwise we will get compile time error like un reported exception must be caught or declared to be thrown.

Ex: 1

Class Test

{

Public static void main(String args[])

{

Thread.sleep(5000);

System.out.println(“I slept very much”);

}

}

C.E: un reported exception xxx must be caught or declared to be thrown.

Ex: 2

Class Test

{

Public static void main(String args[]) throws InterruptedException

{

Thread.sleep(5000);

System.out.println(“I slept very much”);

}

}

o/p: we won’t get any error.

Actually, the main purpose of throws keyword is to delegate the responsibility of Exception handling to the caller.

Ex: 3

Class Test

{

Public static void main(String args[]) throws InterruptedException

{

Fun();

}

Public static void fun() throws InterruptedException

{

Gun();

}

Public static void gun() throws InterruptedException

{

Thread.sleep(3000);

}

}

o/p: no errors, no exceptions.

Note:

Throws keyword is just for convincing compiler in the case of checked exception only. There is no impact of throws keyword for unchecked exception.

Ex: 4

We can use throws keyword only for throwable class, otherwise we will get compile time error.

Class Test

{

Public static void main(String args[]) throws Test

{

}

}

C.E: incompatable type

Found: Test

Required: Throwable

Q18: summary of exception handling keywords.

Try 🡪 to maintain risky code.

Catch 🡪 to maintain handling code.

Finally 🡪 to maintain clean-up code.

Throw 🡪to handover our explicitly created Exception object to the jvm.

Throws 🡪to deligate the responsibility of Exception handling to the caller.

Various possible compile time errors in Exception handling

1. Exception xxxx has already been caught i.e try with multiple catch blocks.
2. Exception xxxx is never thrown in body of corresponding try statement.
3. Un reported Exception xxxx Exception must be caught or declared to be thrown.
4. try without catch or finally
5. catch without try
6. finally without try
7. incompatable types

found: xxxx

required:Throwable

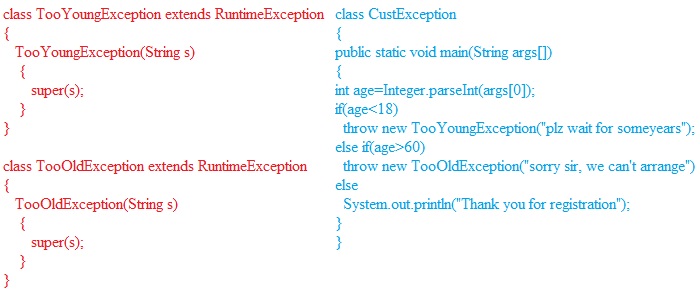
1. unreachable statement

customized Exception:

based on our programming requirement, some times we can create our own exceptions also. Such type of exceptions are called customized exception.

Ex: TooYoungException, TooOldException, InsufficientFundsException

See the following example



Note: it is recommended to keep customized exceptions as unchecked i.e our customized exception class has to extend RuntimeException, but not Exception.

top 10 Exceptions:

all the exceptions are devided into two categories. (i) jvm exceptions (ii) programmatic exceptions

1. jvm exceptions: these are raised automatically by the jvm, whenever a preticular event triggered.

Eg: ArrayIndexOutof BoundsException, NullPointerException

1. programmatic Exception: these are raised explicitly by the programmer or by the api developer.

Eg: IllegalArgumentException, NumberFormatException

The top 10 exceptions are

1. ArrayIndexOutofBoundsException
2. NullPointerException
3. StackOverFlowError
4. ClasscastException
5. NoClassFoundFoundError
6. ExceptionInInitializerError
7. IllegalArgumentException
8. NumberFormatException
9. IllegalStateException
10. AssertionError

Out of the above 10, first 6 are jvm exceptions and the remaining 4 are programmatic exceptions.

1.ArrayIndexOutofBoundsException:

It is child class of RuntimeException, hence it is unchecked exception. It raises when we try to access array elements with out of range index.

Ex:

String s[]=new String[6];

System.out.println(s[0]);

System.out.println(s[6]); // it raises ArrayIndexOutofBoundsException

System.out.println(s[10]); // it raises ArrayIndexOutofBoundsException

2.NullPointerException:

It is child class of RuntimeException, hence it is unchecked exception. It raises when we try to perform any operation on null.

Ex:

String s=null;

System.out.println(s.length()); // it raises NullPointerException

3.StackoverFlowError:

it is child class of Error, hence it is unchecked, it raises whenever a recursive method call is invoked.

Ex:

class Test

{

Public static void m1()

{

m2();

}

Public static void m2()

{

m1();

}

Public static void main(String args[])

{

m1();

}

} // it raises StackoverFlowError

4.ClassCastException:

It is child class of RuntimeException, hence it is unchecked exception. It raises when we try to typecast parent object to child type.

Ex:

String s=”gutta”;

Object o=(Object)s; // valid

Object o=new Object();

String s=(String)o; // it raises ClasscastException

5.NoClassDefFoundError:

it is child class of Error, hence it is unchecked. It raises automatically by jvm, whenever jvm class loader unable to find required .class file.

Ex: java Test

If Test.class file is not available, then we will get NoClassDefFoundError.

6.ExceptionInitializerError:

it is child class of Error, hence it is unchecked. It raises whenever exception occurs during exception of static variable assignments and static block.

Ex:

class Test

{

static int i=10/0;

}

In the above, ExceptionInInitializerError caused by ArithmeticException division by zero.

Ex:

class Test

{

static

{

String s=null;

System.out.println(s.length());

}

}

In the above, ExceptionInInitializerError caused by NullPointerException.

7.IllegalArgumentException:

It is child class of RuntimeException, hence it is unchecked. It raises by the programmer or api developer explicitly to indicate a method has been invoked within appropriate argument.

Ex:

Thread t=new Thread();

t.setPriority(100); // it raises IllegalArgumentException

8.NumberFormatException:

it is the child class of IllegalArgumentException, hence it is unchecked. It raises by the programmer or api developer explicitly to indicate that if we try to convert string to number type, but the string is not properly formatted.

int i=Integer.parseInt(“10”);

int j=Integer.parseInt(“ten”); // it raises NumberFormatException

9.IllegalStateException:

It is child class of RuntimeException, hence it is unchecked. It raises explicitly by the programmer or api when a method has been invoked at an appropriate time.

Example: after starting a thread, we are not allowed to restart the same thread once again.

Thread t=new Thread();

t.start();

t.start(); // it raises ISE

Example: once session expires, we are not allowed to call any method on that session object.

HttpSession session=req.getSession();

System.out.println(session.getId());

session.invalidate(); // here session expires

System.out.println(session.getId()); // it raises ISE

10.AssertionError:

It is child class of Error, hence it is unchecked. It raises explicitly by the programmer to indicate assert statement fails.

T H E E N D