

```
/*
```

Here I present the solution to the nine nines problem.

I imposed an extra condition on myself since the result was to be emailed, I decided to do the implementation in a single file. Thus some classes have become inner classes where I might have made them separate files normally. This does not pervert the result in any way.

Also I have presented the methods, members and inner class in an order which makes it easier to read from top to bottom. With modern development tools this doesn't matter so much as I have direct access to all methods and definitions through the UI.

The methods most related to the algorithm are near the top, the utility classes and methods near the bottom.

Copyright 2002 by Nicolas Minutillo.
All rights reserved.

```
*/
```

```
import java.util.*;  
public class nn  
{
```

```
    //----- A description of the algorithm -----
```

```
    /*
```

Find the smallest positive integer that cannot be expressed as an arithmetic statement involving nine nines and the operations +, -, * and /.

To solve this problem we first solve the problem of finding all numbers that can be generated by such expressions. Let's call this set $S(9, 9)$. $S(i, 9)$ would be the set of all numbers that can be generated using i nines. $S(8, 5)$ is, of course, the set of all numbers that can be generated by expressions of exactly 8 fives.

Once we have $S(9, 9)$ in hand it is a simple exercise to find the smallest positive integer which is missing.

To find $S(9, 9)$ we observe that each expression can be decomposed into three parts: an operation and two operands, where the one of the operands is an element of $S(i, 9)$ and the other operand is an element of $S(9 - i, 9)$.

For example

$$1 = [(9 - 9)] * [(9 + 9 + 9 + 9 + 9 + 9 + 9)]$$

comprises a multiplication of an element of $S(2, 9)$ $[(9 - 9)]$ and an element of $S(7, 9)$, $[(9 + 9 + 9 + 9 + 9 + 9 + 9)]$.

As a second example, to further clarify the algorithm, consider

$$7 = [(9 + 9) / 9] + [(9 + 9 + 9 + 9 + 9) / 9].$$

7 is generated by adding an element of $S(3, 9)$ $[(9 + 9) / 9]$ to an element of $S(6, 9)$ $[(9 + 9 + 9 + 9 + 9) / 9]$.

Since all elements of $S(9, 9)$ are of this form, if we imagine that we have already determined the sets $S(i, 9)$ for i in $[1, 8]$ then the solution to our problem is obvious.

These observations lead to the following algorithm,

To create the set $S(n, m)$ (where $n > 1$):

```

For each integer i in [1, n - 1]
  For each member L of S(i, m)
    For each member R of S(n - i, m)
      the number L + R is in S(n, m)
      the number L - R is in S(n, m)
      the number L * R is in S(n, m)
      the number L / R (where defined) is in S(n, m)
    End for
  End for
End for

```

$S(1, m)$ is $\{-m, m\}$.

Of course, this algorithm also can produce (recursively) all the sets $S(i, 9)$ with $i = 1..8$ that we need to find $S(9, 9)$.

We are done!

*/

//----- generateSolution -----

/*

This is the method that does all the interesting work, the rest of the file just lays the foundation for this routine. This routine simply implements the algorithm above.

To speed up the calculations we have added the following optimizations. They are listed in order of importance (or benefit):

- 1) We save all results, that is we only calculate a particular $S(i, 9)$ once.
- 2) Because the majority of the operations (+, *) commute the first for loop need only consider i in $[1, (n-1)/2]$. We compensate by calculating both L / R and R / L .
- 3) We need not consider -. Because:
 - We define $S(1, 9)$ to be $\{-9, 9\}$
 - $a - b = a + -b$
 - $-(a * b) = -a * b$
 - $-(a + b) = -a + -b$
 - $-(a / b) = -a / b$

You should read the note below on division and the complication it creates.

```
*/
```

```
//----- Partial list of member variables for class nn -----
```

```
private int value;
private int level;
```

```
//----- Compute S(level, value) -----
```

```
private void generateSolution()
```

```
{
    try
    {
        if (level > 1)
        {
            for (int i = 1; i <= level / 2; i++)
            {

                // resultSet is a set
                // memberIterator retrieves the members of that resultSet.

                resultSet leftResults = getSolutionFor(i, value);
                resultSet rightResults = getSolutionFor(level - i, value);

                memberIterator j = leftResults.getIterator();
                while (j.hasNext())
                {
                    double lValue = j.next();

                    memberIterator k = rightResults.getIterator();
                    while (k.hasNext())
                    {
                        double rValue = k.next();

                        // The final three parameters allow us to keep track,
                        // if we wish, of where elements came from.
```

```

// trackingObjects how we do this.

addToSet(lValue + rValue,
        leftResults.getTrackingObject(lValue),
        rightResults.getTrackingObject(rValue),
        "+");

addToSet(lValue * rValue,
        leftResults.getTrackingObject(lValue),
        rightResults.getTrackingObject(rValue),
        "*");

// There are some ambiguities with division.
// See the discussion below. Unfortunately the
// can influence the answer. We allow the user
// to tell us what division means.

if (validDivide(rValue, lValue))
{
    addToSet(computeDivide(rValue, lValue),
            rightResults.getTrackingObject(rValue),
            leftResults.getTrackingObject(lValue),
            "/");
}

if (validDivide(lValue, rValue))
{
    addToSet(computeDivide(lValue, rValue),
            leftResults.getTrackingObject(lValue),
            rightResults.getTrackingObject(rValue),
            "/");
}
}
}
}
}
else if (level == 1)
{
    addToSet(value);
    addToSet(-value);
}
}
catch (Exception e)
{
    System.out.println("Unexpected error in algorithm.");
    System.out.println(e.toString());
    System.exit(1);
}
}
}

```

```
/*
```

Our description of the algorithm is not yet complete. While statements like $L + R$ and $L * R$ are very clear in their meaning (they are operations on integers that yield integer results) there is some ambiguity about L / R . To make this clear consider the following two statements in the Java programming language:

```
double r = 9 / (9 + 9 + 9)
```

and

```
int rr = 9 / (9 + 9 + 9)
```

with these definitions $r = 0.333\dots$ and $rr = 0$.

Which should we use in solving the nine nines problem?

Moreover, one could convincingly argue that $9 / (9 + 9 + 9)$ is no more meaningful than $9 / (9 - 9)$ when interpreted as a statement about integers. Thus $9 / (9 + 9 + 9)$ could be considered undefined.

So which will it be (a) $0.333\dots$, (b) 0 or (c) undefined?

Fortunately (a) and (b) for the nine nines problem yield the same final result: 195 (although very different sets $S(9, 9)$).

Unfortunately (c) yields a different result: 138.

Our solution to this ambiguity is to allow you to decide. If you like (a) then set the `divideType` to `doubleDivide`. If you like (b) then set the `divideType` to `integerRound`. If you like (c) then set the `divideType` to `integerStrict`.

```
*/
```

```
//----- Partial list of member variables for class nn-----
```

```
public static final int doubleDivide = 0;
public static final int integerRound = 1;
public static final int integerStrict = 2;
public static final int defaultDivideType = integerRound;
```

```
private int divideType;
```

```
// Given our definition of division is the statement
// numerator / denominator defined.
```

```
public boolean validDivide(double numerator, double denominator)
{
```

```

    if (divideType == doubleDivide || divideType == integerRound)
    {
        return denominator != 0;
    }
    else
    {
        return (denominator != 0) &&
            (((long) Math.round(numerator) / Math.round(denominator))
            * denominator == numerator);
    }
}

```

// Compute (either as a double or an int) the division.

```

public double computeDivide(double numerator, double denominator)
{
    if (divideType == doubleDivide)
    {
        return numerator / denominator;
    }
    else
    {
        long r = Math.round(numerator) / Math.round(denominator);
        return (double) r;
    }
}

```

/*

Now we are ready to display our main program. It computes $S(9, 9)$ and then searches this solution set for the first missing positive integer.

For the purposes of demonstration the routine does a little more than this.

- A) Display problem being solved and assumptions about division.
- B) Display solution to the problem and elapsed time for the computation.
- C) Recompute (tracking the expressions generated) to display a partial proof.

*/

```

public static void main (String args[])
{
    try
    {
        long startTime = System.currentTimeMillis();

        // Display problem being solved and assumptions about division.
    }
}

```

```

nn problem = new nn(9, 9);

System.out.println("Solving the " + problem.getLevel()
    + " " + problem.getValue() + "s problem: ");

System.out.print("We are assuming that (9 / 27) is ");
if (problem.validDivide(9, 27))
{
    double r = problem.computeDivide(9, 27);
    System.out.println(r + ".");
}
else
{
    System.out.println("undefined.");
}

// Display solution to the problem and elapsed time for the computation.

resultSet v = problem.getSolution();
for (int i = 1; i < 32000; i++)
{
    if (!v.contains(i))
    {
        System.out.println("Cannot produce a " + i + ".");
        break;
    }
}

long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("Elapsed time is " +
    (double) elapsedTime/ 1000.0 + " seconds.");

// Recompute (tracking the expressions generated)
// to display a partial proof.

problem = new nn(9, 9, new expressionTracking());
v = problem.getSolution();
System.out.println("For validation consider the following expressions:");

for (int i = 0; i < 32000; i++)
{
    if (v.contains(i))
    {
        if (problem.isTracking())
        {
            trackingObject proof = v.getTrackingObject(i);
            System.out.println(i + " = " + proof.toString());
        }
    }
    else

```

```

        {
            break;
        }
    }
}
catch (Exception e)
{
    System.out.println("An unavoidable error occurred.");
    System.out.println(e.toString());
}
}

/*

```

The rest of this file simply builds up the foundation we need for what we have seen above to work. There is not much of interest here, while all of it is needed it is relatively straight forward.

Perhaps it is time to run the program!

We proceed in the following order:

- 1) The public interface to class nn.
- 2) The mechanism for saving solutions.
- 3) The mechanism for tracking the results.
- 4) The resultSet and memberIterator objects and related functions.
- 5) Utility methods.

```

*/

//----- Public interface for class nn-----

/*

```

This class nn has the following constructors:

```

problem(int atLevel, int withValue) throws Exception;
problem(int atLevel, int withValue, trackingObject tracker) throws Exception;
problem(int atLevel, int withValue, int withDivideType,
        boolean withLimitedMemUse, trackingObject usingTemplate)
        throws Exception;

```

The class nn has the following public methods:

```

public boolean validDivide(double numerator, double denominator);
public double computeDivide(double numerator, double denominator);

public resultSet getSolution();
public int getValue();
public int getLevel();

```

```
public boolean isTracking();
```

First, we describe the constructors. The primary constructor is

```
problem(int atLevel, int withValue, int withDivideType,  
        boolean withLimitedMemUse, trackingObject usingTemplate)  
        throws Exception;
```

It computes $S(\text{atLevel}, \text{withValue})$.

The parameter `withDivideType` tells us how to interpret the divisions in the expressions. It is one of `doubleDivide`, `integerRound` or `integerStrict`.

The parameter `withLimitedMemUse` tells whether to implement the optimization we recall here:

1) We save all results, that is we only calculate each $S(i, 9)$ once.

If `withLimitedMemUse` is true we turn off this optimization.

The parameter `usingTemplate` is either an object of a class which implement the interface `trackingObject`. If it is null we do no tracking (this does not interfere with solving the problem, it just prevents us from 'showing our work'). Otherwise we use the given object to do the tracking.

If anything goes wrong or if the parameters are invalid an `Exception` is thrown.

The other two constructors just use default values for the missing parameters:

```
withDivideType defaults to defaultDivideType,  
withLimitedMemUse defaults to false,  
usingTemplate defaults to null.
```

Immediately upon construction the object solves the $S(\text{atLevel}, \text{withValue})$ problem.

```
*/
```

```
//----- Partial list of member variables for class nn -----
```

```
private boolean limitMemUse;  
private resultSet solution;  
private trackingObject trackingObjectTemplate;
```

```
//----- The constructors for class nn -----
```

```
nn( int atLevel,  
    int withValue,  
    int withDivideType,
```

```

boolean withLimitedMemUse,
trackingObject usingTemplate) throws Exception
{
    // Assert the various limits on the parameters.

    assert(withValue > 0, "Value is non-positive.");
    assert(atLevel >= 1, "Must allow at least one number.");
    assert(withDivideType >= 0 && withDivideType <= 2,
           "Unknown division algorithm.");

    // Set up the object.

    value = withValue;
    level = atLevel;
    divideType = withDivideType;
    limitMemUse = withLimitedMemUse;
    trackingObjectTemplate = usingTemplate;

    // Create the solution set.

    solution = new resultSet();

    // Solve the S(level, value) problem storing the results.

    try
    {
        generateSolution();
    }

    catch (OutOfMemoryError oRecover)
    {
        // If we are out of memory and we are already in a limited memory regime
        // then there isn't much we can do. Just throw our own exception.

        // If we were not in a limited memory use regime, then retry the problem
        // and limit the amount of memory use. This is slower but may
        // yield a result.

        if (limitMemUse)
        {
            System.gc();
            savedSolutions.clear();
            throw new Exception("Insufficient resources for the computation.");
        }
        else
        {
            limitMemUse = true;
            System.gc();
            savedSolutions.clear();
        }
    }
}

```

```

        try
        {
            generateSolution();
        }
        catch (OutOfMemoryError oFatal)
        {
            savedSolutions.clear();
            throw new Exception(
                "Insufficient resources for the computation");
        }
    }
}

nn(int atLevel, int withValue, trackingObject usingTemplate) throws Exception
{
    this(atLevel, withValue, defaultDivideType, false, usingTemplate);
}

nn(int atLevel, int withValue) throws Exception
{
    this(atLevel, withValue, defaultDivideType, false, null);
}

/*

The remaining public methods are simply accessor methods.
They provide information about the object state and access
to the solution.

*/

//----- Public methods for class nn -----

public resultSet getSolution()
{
    return solution;
}

public int getValue()
{
    return value;
}

public int getLevel()
{
    return level;
}

public boolean isTracking()
{

```

```
    return trackingObjectTemplate != null;
}
```

```
//----- The mechanism for saving solutions -----
```

```
/*
```

We have already seen in the method `generateSolution` that we can retrieve (or compute if it is the first mention) previous solutions with the `getSolutionFor` method.

We present that method here as an introduction to the saved solutions mechanism.

```
*/
```

```
//----- Partial list of member variables for class nn -----
```

```
private static savedObjects savedSolutions = new savedObjects(3);
```

```
/*
```

The method `getSolutionFor` simply requests the appropriate solution from the `savedObjects` class static member variable `savedSolutions`. This member variable is static assure that only one copy of the solutions is maintained. This means that $S(2, 9)$ will only be calculated once, not once by $S(9, 9)$ and then again by $S(8, 9)$ and so forth.

Because the object is static it has no access to the members variables of class `nn` so `getSolutionFor` passes those in as parameters to the `savedObjects` method `get`.

```
*/
```

```
private resultSet getSolutionFor(int level, int value) throws Exception
```

```
{
```

```
    return savedSolutions.get(value,
                               level,
                               divideType,
                               limitMemUse,
                               trackingObjectTemplate);
```

```
}
```

```
/*
```

The public interface for class `savedObjects` is quite simple.

The class `savedObjects` has the following constructor:

```
savedObjects(int divideTypeCount);
```

This constructor allocates sufficient storage for all the different divide types we have defined (3).

It has the following public methods

```
public resultSet get(int withValue, int atLevel, int withDivideType,  
    boolean withLimitedMemUse, trackingObject usingTemplate) throws Exception;
```

```
public void clear();
```

The get method returns the appropriate saved solution.

The clear method sets the main storage in the object to null allowing memory to be freed.

```
*/
```

```
//----- The class savedObjects -----
```

```
private static class savedObjects  
{
```

```
    private HashMap storage[];
```

```
//----- Constructors for class savedObjects -----
```

```
savedObjects(int divideTypeCount)
```

```
{  
    storage = new HashMap[divideTypeCount];  
    for (int i = 0; i < storage.length; i++)  
    {  
        storage[i] = new HashMap();  
    }  
}
```

```
//----- Public methods for class savedObjects -----
```

```
public void clear()
```

```
{  
    for (int i = 0; i < storage.length; i++)  
    {  
        storage[i] = new HashMap();  
    }  
    System.gc();  
}
```

```
public resultSet get(int withValue,  
    int atLevel,  
    int withDivideType,  
    boolean withLimitedMemUse,  
    trackingObject usingTemplate) throws Exception  
{  
    // Assert limits on the parameters.
```



```

{
    // Either retrieve the appropriate HashMap or create it.

    // This HashMap is a collection of solutions (all levels and values) for
    // this trackingType. It is indexed by value.

    HashMap savedObjectsForTracking =
        savedSolutions.getForTracking(withDivideType, usingTemplate);

    // If we haven't computed any solutions for this value yet
    // then allocate the appropriate storage.

    if (!savedObjectsForTracking.containsKey(new Integer(withValue)))
    {
        savedObjectsForTracking.put(new Integer(withValue), new HashMap());
    }

    // Return all the solutions for this value, tracking kind and
    // division rules.

    return (HashMap) savedObjectsForTracking.get(new Integer(withValue));
}

private HashMap getForTracking(int withDivideType,
                               trackingObject usingTemplate)
{
    // The class name of the trackingObject will be used as the index into a
    // HashMap. If we are not tracking then the string "null" will be used.

    String trackingClass = "";
    if (usingTemplate != null)
    {
        trackingClass = usingTemplate.getClass().getName();
    }

    // If we haven't stored solutions with this kind of tracking yet
    // then allocate some storage for them. Space has already been reserved
    // for all of the division rules.

    if (!storage[withDivideType].containsKey(trackingClass))
    {
        storage[withDivideType].put(trackingClass, new HashMap());
    }

    // Return all the solutions with this kind of tracking
    // and division rules.

    return (HashMap) storage[withDivideType].get(trackingClass);
}
}

```

```
//----- The mechanism for tracking the results -----
```

```
/*
```

In the method `generateSolution` we have already seen calls to the method `addToSet`. We display this method to introduce the concept of tracking the generation of a solution.

`addToSet` comes in two flavors:

```
private void addToSet(double result)
```

which is used only at level one. It inserts the 9 and -9 into the solution and tracking systems.

```
private void addToSet(double result, trackingObject proofLeft,  
    trackingObject proofRight, String operation) throws Exception
```

which is used everywhere else. It inserts the result into the solution and then, if we are tracking, creates a new tracking object out of the `proofLeft`, `proofRight` and `operation` objects.

Recall that $result = L + R$ (let's say as an example), then `proofLeft` is a record of how we computed `L`, `proofRight` is a record of how we computed `R` and the new tracking object will be a record of how we computed `result` (it will record that we added `L` and `R`).

We don't know the class of the `trackingObject`--all we know is that it implements `trackingObject`.

For simplicity we have made `trackingObjects` their own class factories. Thus we can call `create` and get a new object of the same class as the template object. More formally we would have an interface `trackingObject` and then a class (say) `expressionObjectFactory` with a static method that returned a `trackingObject` whose class is `expressionObject`. But that seems a bit much for this exercise.

```
*/
```

```
private void addToSet(double result)  
{  
    trackingObject proof = null;  
  
    if (trackingObjectTemplate != null)  
    {  
        // trackingObjectTemplate plays the role is a class factory here.  
        // Creating a new object of its own class.  
        proof = trackingObjectTemplate.create(result);  
    }  
  
    // Store this away in the solution set.
```

```

        solution.insertIfNew(result, proof);
    }

    private void addToSet(double result,
                        trackingObject proofLeft,
                        trackingObject proofRight,
                        String operation) throws Exception
    {
        trackingObject proof = null;

        if (trackingObjectTemplate != null)
        {
            // trackingObjectTemplate plays the role is a class factory here.
            // Creating a new object of its own class.
            proof = trackingObjectTemplate.create(proofLeft, proofRight, operation);
        }

        // Store this away in the solution set.

        solution.insertIfNew(result, proof);
    }
}

```

/*

Here is the interface that all trackingObjects (including the expressionTracking object below) implement.

The methods that support tracking results:

public String toString() represents with trackingObject as a string.

public boolean isNew(double result, resultSet solution) determines whether this is a new result with new tracking data.

Class factory methods:

public trackingObject create(double result) an initial tracking object.

public trackingObject create(trackingObject proofLeft, trackingObject proofRight,
 String anOperation) throws Exception

creates a tracking object combining proofLeft, proofRight and the operation.

*/

```

public interface trackingObject
{
    public String toString();
    public boolean isNew(double result, resultSet solution);
}

```

```

    public trackingObject create(double result);
    public trackingObject create(trackingObject proofLeft,
                                trackingObject proofRight,
                                String anOperation) throws Exception;
}

```

```
/*
```

A trackingObject implementation that tracks and displays simple equations for each result.

The goal of this trackingObject is to assure that we recall the simplest expression for each generated number. It makes the proof easier to read.

To that end, we make sure we do not use excess parantheses. We prefer addition to multiplication, multiplication to division and positive to negative integers.

```
*/
```

```
public static class expressionTracking implements trackingObject
{

```

```

    private String expression;
    private String operation;
    private int complexity;

```

```
//----- Constructors for class expressionTracking -----
```

```
// The default constructor is used to create the template.
```

```

expressionTracking()
{
    expression = "";
    operation = "";
    complexity = 0;
}

```

```
// This constructor corresponds to the create(double result) factory method.
```

```

expressionTracking(double result)
{
    Long simple = new Long(Math.round(result));
    expression = simple.toString();
    operation = "";
    if (result < 0)
    {
        complexity = 1;
    }
    else
    {

```

```

        complexity = 0;
    }
}

// This constructor corresponds to the
// create(trackingObject proofLeft, trackingObject proofRight,
//        String anOperation) throws Exception

expressionTracking(trackingObject proofLeft,
                   trackingObject proofRight,
                   String anOperation) throws Exception
{
    operation = anOperation;

    expressionTracking left = (expressionTracking) proofLeft;
    expressionTracking right = (expressionTracking) proofRight;

    if (operation == "+")
    {
        expression = left.expression + " + " + right.expression;
        complexity = left.complexity + right.complexity;
    }
    else if (operation == "*")
    {
        if (left.operation != "/" && left.operation != "")
        {
            expression = "(" + left.expression + ")";
        }
        else
        {
            expression = left.expression;
        }
        expression += " * ";
        if (right.operation != "/" && right.operation != "")
        {
            expression += "(" + right.expression + ")";
        }
        else
        {
            expression += right.expression;
        }
        complexity = (left.complexity + right.complexity + 1) * 2;
    }
    else if (operation == "/")
    {
        if (left.operation != "")
        {
            expression = "(" + left.expression + ")";
        }
        else

```

```

        {
            expression = left.expression;
        }
        expression += " / ";
        if (right.operation != "")
        {
            expression += "(" + right.expression + ")";
        }
        else
        {
            expression += right.expression;
        }
    }

    complexity = (left.complexity + right.complexity + 1) * 100;
}
else
{
    throw new Exception("Illegal operation detected.");
}
}

public boolean isNew(double result, resultSet solution)
{
    return complexity < ((expressionTracking)
        solution.getTrackingObject(result)).complexity;
}

//----- Public methods for class expressionTracking -----

public String toString()
{
    return expression;
}

//----- Class factory methods for expressionTracking
// as a trackingObject -----

public trackingObject create(trackingObject left,
                            trackingObject right,
                            String anOperation) throws Exception
{
    trackingObject tracker =
        new expressionTracking(left, right, anOperation);
    return tracker;
}

public trackingObject create(double result)
{
    trackingObject tracker = new expressionTracking(result);
    return tracker;
}

```

```
}
```

```
/*
```

This is a simple object to track results and an associated object (in our case the trackingObject).

It is a type safe wrapper for a HashMap, indexed by numbers, with values of trackingObjects.

```
*/
```

```
public class resultSet
{
    HashMap storage;

    resultSet()
    {
        storage = new HashMap();
    }

    public memberIterator getIterator()
    {
        return new memberIterator(storage.keySet().iterator());
    }

    public trackingObject getTrackingObject(double value)
    {
        return (trackingObject) storage.get(new Double(value));
    }

    public boolean contains(double result)
    {
        return storage.containsKey(new Double(result));
    }

    private void insertIfNew(double result, trackingObject tracker)
    {
        if (!contains(result) ||
            ((tracker != null) && (tracker.isNew(result, solution))))
        {
            storage.put(new Double(result), tracker);
        }
    }
}

/*
```

An iterator for the resultSet objects. A type safe wrapper of the Java class Iterator.

```
*/  
  
public class memberIterator  
{  
    Iterator myIterator;  
  
    memberIterator(Iterator j)  
    {  
        myIterator = j;  
    }  
  
    public boolean hasNext()  
    {  
        return myIterator.hasNext();  
    }  
  
    public double next()  
    {  
        return ((Double) myIterator.next()).doubleValue();  
    }  
}  
  
//----- Utility functions -----  
  
private static void assert(boolean guard, String sError) throws Exception  
{  
    if (!guard)  
    {  
        throw new Exception(sError);  
    }  
}  
}
```