Chapter 2 Flow Control

2.1 Develop code that implements if or switch statement; and identify legal argument types for these statements

The syntax of the `switch` statement is extended ever-so-slightly. The type of the `Expression` is now permitted to be an `enum` class. (Note that `java.util.Enum` is not an `enum` class.) A new production is added for `SwitchLabel`:

```
SwitchLabel:
        case EnumConst :

EnumConst:
        Identifier
```

The `Identifier` must correspond to one of UNQUALIFIED enumeration constants.

Here is a slightly more complex `enum` declaration for an `enum` type with an explicit instance field and an accessor for this field. Each member has a different value in the field, and the values are passed in via a constructor. In this example, the field represents the value, in cents, of an American coin.

```
public enum Coin {
        PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

        Coin(int value) {
                this.value = value;
        }
        private final int value;
        public int getValue() { return value; }
}
```

Switch statements are useful for simulating the addition of a method to an `enum` type from outside the type. This example "adds" a color method to the `Coin` class, and prints a table of coins, their values, and their colors.

```
import static java.lang.System.out;

public class CoinTest {
        public static void main(String[] args) {
                for (Coin c : Coin.values()) {
                        out.println(c + ":   \t" + c.getValue() + "c  \t" +
                                                                     color(c));
                }
        }

        private enum CoinColor {
                COPPER, NICKEL, SILVER
        }

        private static CoinColor color(Coin c) {
                if (c == null) {
                        throw new NullPointerException();
                }

                switch (c) {
//                      case Coin.PENNY: {}    // Compile error! Must be
                                               // UNQUALIFIED !!!
                        case PENNY:
                                return CoinColor.COPPER;
                        case NICKEL:
                                return CoinColor.NICKEL;
                        case DIME:
                                return CoinColor.SILVER;
                        case QUARTER:
                                return CoinColor.SILVER;
//                      case 2: {}      // Compile error !!!
                                        // Type mismatch: cannot convert from
                                        // int to Coin
                }

                throw new AssertionError("Unknown coin: " + c);
        }
}
```

Running the program prints:

```
PENNY:          1c      COPPER
NICKEL:         5c      NICKEL
DIME:           10c     SILVER
QUARTER:        25c     SILVER
```

2.2 Develop code that implements all forms of loops and iterators, including the use of for, the enhanced for loop (for-each), do, while, labels, break, and continue; and explain the values taken by loop counter variables during and after loop execution.

An enhanced "for" loop

The current `for` statement is quite powerful and can be used to iterate over arrays or collections. However, it is not optimized for collection iteration, simply because the iterator serves no other purpose than getting elements out of the collection. The new enhanced `for` construct lets you iterate over collections and arrays without using iterators or index variables.

The new form of the `for` statement has the following syntax:

```
for ( Type Identifier : Expression ) Statement
```

Note that `Expression` MUST be an array or an instance of a new interface called `java.lang.Iterable`, which is meant to ease the task of enabling a type for use with the enhanced `for` statement. This means that the `java.util.Collection` now extends the `Iterable` interface that has the following signature:

```
package java.lang;

public interface Iterable<E> {
  /**
   * Returns an iterator over the elements in this collection.
   * There are no guarantees concerning the order in which the elements
   * are returned (unless this collection is an instance of some class
   * that provides such a guarantee).
   *
   * @return an iterator over the elements in this collection
   */

   Iterator<E> iterator();
}
```

As an example, consider the following method that uses the conventional `for` statement to iterate over a collection:

```
// Assume we have an instance of StringBuffer "sb"
public void oldFor(Collection c) {
        for(Iterator i = c.iterator(); i.hasNtext(); ) {
                String str = (String) i.next();
                sb.append(str);
        }
}
```

With the addition of generics, the above segment of code can be rewritten using the enhanced `for` statement as follows:

```
// Assume we have an instance of StringBuffer "sb"
public void newFor(Collection<String> c) {
        for(String str : c) {
                sb.append(str);
        }
}
```

The enhanced `for` statement can be used to iterate over an array. Consider the following segment of code for a method that calculates the sum of the elements in an array:

```
public int sumArray(int array[]) {
        int sum = 0;
        for(int i=0; i<array.length; i++) {
                sum += array[i];
        }
        return sum;
}
```

Using the enhanced `for` statement, this can be rewritten as:

```
public int sumArray(int array[]) {
        int sum = 0;
        for(int i : array) {
                sum += i;
        }
        return sum;
}
```

The `for(:)` loop is designed for iteration over *collections* and *arrays*.

Iterating over arrays:

```
// for(;;) loop
int[] ageInfo = {12, 30, 45, 55};
int sumAge = 0;
for (int i = 0; i < ageInfo.length; i++) {
        sumAge += ageInfo[i];
}

// for(:) loop
int[] ageInfo = {12, 30, 45, 55};
int sumAge = 0;
for (int element : ageInfo) {
        sumAge += element;
}
```

NOTE, that an array element of a primitive value CANNOT be modified in the `for(:)` loop.

Iterating over non-generic `Collection`:

```
// for(;;) loop
Collection nameList = new ArrayList();
nameList.add("Mikalai");
nameList.add("Michael");
nameList.add("Craig");
for (Iterator it = nameList.iterator(); it.hasNext(); ) {
        Object element = it.next();
        if (element instanceof String) {
                String name = (String) element;
                ...
        }
}

// for(:) loop
Collection nameList = new ArrayList();
```

```
nameList.add("Mikalai");
nameList.add("Michael");
nameList.add("Craig");
for (Object element : nameList) {
        if (element instanceof String) {
                String name = (String) element;
                ...
        }
}
```

Iterating over generic `Collection`:

```
// for(;;) loop
Collection<String> nameList = new ArrayList<String>();
nameList.add("Mikalai");
nameList.add("Michael");
nameList.add("Craig");
for (Iterator<String> it = nameList.iterator(); it.hasNext();) {
        String name = it.next();
        ...
}

// for(:) loop
Collection<String> nameList = new ArrayList<String>();
nameList.add("Mikalai");
nameList.add("Michael");
nameList.add("Craig");
for (String name : nameList) {
        ...
}
```

The syntax of the `for(:)` loop does not use an iterator for the collection.

NOTE, the `for(:)` loop DOES NOT allow elements to be removed from the collection.

```
for (Type FormalParameter : Expression) Statement
```

The `FormalParameter` must be declared in the `for(:)` loop.

The `Expression` is evaluated only once.

The type of `Expression` is `java.lang.Iterable` or an array. The `java.util.Collection` interface is retrofitted to extend the `java.lang.Iterable` interface which has the method prototype `Iterator<T> iterator()`.

When `Expression` is an array: the type of the array element MUST be assignable to the `Type` of the `FormalParameter`.

When `Expression` is an instance of a raw type `java.lang.Iterable`: the `Type` of the `FormalParameter` MUST be `Object`.

When `Expression` is an instance of a *parameterized* type `java.lang.Iterable<T>`: the type parameter `T` MUST be assignable to the `Type` of the `FormalParameter`.

Restrictions on the enhanced `for` loop:

- CANNOT be used to remove elements from a collection. The 'for-each' loop hides the iterator, so you CANNOT call `remove()`. Therefore, the 'for-each' loop is not usable for filtering:

```
// This is NOT possible with 'for-each' loop !!!
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
     for (Iterator<String> i = c.iterator(); i.hasNext(); ) {
             if (i.next().length() == 4) {
                     i.remove();
             }
     }
}
```

- CANNOT be used to modify the current slot in a collection or an array. It is not usable for loops where you need to replace elements in a list or array as you traverse it.

```
Collection<String> nameList = new ArrayList<String>();
nameList.add("Mikalai");
nameList.add("Michael");
nameList.add("Craig");
for (String name : nameList) {
     name = "Nicholas"; // WRONG ! Can not modify collection
}
System.out.println(nameList);
```

The output:

```
[Mikalai, Michael, Craig]

int[] numArray = new int[] {1,2,3,4,5};
for (int i : numArray) {
  i = 0; // WRONG ! Can not modify array !
}
System.out.println(Arrays.toString(numArray));
```

The output:

```
[1, 2, 3, 4, 5]
```

However, we CAN modify object itself:

```
Collection<StringBuffer> bufferList = new ArrayList<StringBuffer>();
bufferList.add(new StringBuffer("Mikalai"));
bufferList.add(new StringBuffer("Michael"));
bufferList.add(new StringBuffer("Craig"));
for (StringBuffer name : bufferList) {
  name.reverse(); // CORRECT ! Can modify object itself
}
System.out.println(bufferList);
```

The output:

```
[ialakiM, leahciM, giarC]
```

- CANNOT be used to simultaneously iterate over multiple collections or arrays in parallel.

2.3 Develop code that makes use of assertions, and distinguish appropriate from inappropriate uses of assertions.

Two forms of the assert statement

1. *Usual form*

   An `assert` statement has two parts separated by a colon. The boolean condition must be `true` for execution to continue. If it is `false`, an `AssertionError` is thrown, which terminates execution and display the message string. Some examples:

   ```
   assert jobQueue.size() == 0 : "process: queue should have been empty.";

   assert connector != null : "merge: Connector null for " + rel;
   ```

   When asserts are enabled, the `assert` statement checks the condition (queue empty, connector is not `null`, etc) which must be `true` for the program to function correctly. If it's `true`, execution continues. If `connector` is `null` (expression is `false`), an exception containing the message is thrown. This message is for the programmer, so it doesn't have to be user friendly.

2. *Abbreviated form*

   The simplest form the `assert` statement specifies only a boolean expression that must be `true`. This is ok when there's not much to say, or the likelyhood of failing seems so remote it isn't worth the extra typing:

   ```
   assert n > 0;
   ```

Enabling assertions at runtime

Assertion checking defaults to off at runtime. You should always turn them on.

`assert` statements are removed by class loader:

```
java MyProg
```

`assert` statements are execued:

```
java -enableassertions MyProg
```

Short form for allowing assertions:

```
java -ea MyProg
```

To disable assertions at various granularities, use:

```
java -disableassertions MyProg
```

or

```
java -da MyProg
```

You specify the granularity with the arguments that you provide to the switch:

- no arguments

  Enables or disables assertions in all classes except system classes.

```
packageName...
```

  Enables or disables assertions in the named package and any subpackages.

- `...`

  Enables or disables assertions in the unnamed package in the current working directory.

- `className`

  Enables or disables assertions in the named class

For example, the following command runs a program, `BatTutor`, with assertions enabled in only package `com.wombat.fruitbat` and its subpackages:

```
 java -ea:com.wombat.fruitbat... BatTutor
```

If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. For example, the following command runs the `BatTutor` program with assertions enabled in package `com.wombat.fruitbat` but disabled in class `com.wombat.fruitbat.Brickbat`:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat BatTutor
```

The above switches apply to all class loaders. With one exception, they also apply to system classes (which do not have an explicit class loader). The exception concerns the switches with no arguments, which (as indicated above) do not apply to system classes. This behavior makes it easy to enable asserts in all classes except for system classes, which is commonly desirable.

To enable assertions in all system classes, use a different switch: `-enablesystemassertions`, or `-esa`. Similarly, to disable assertions in system classes, use `-disablesystemassertions`, or `-dsa`.

For example, the following command runs the `BatTutor` program with assertions enabled in system classes, as well as in the `com.wombat.fruitbat` package and its subpackages:

```
java -esa -ea:com.wombat.fruitbat... BatTutor
```

The assertion status of a class (enabled or disabled) is set at the time it is initialized, and does not change.

To compile using J2SE 1.4:

```
javac -source 1.4 AssertTest.java
```

To compile using J2SE 5.0 (assertions are recognized by default):

```
javac AssertTest.java
```

or (explicitly define source code version)

```
javac –source 1.5 AssertTest.java
```

To test assertions during runtime (both J2SE 1.4 and J2SE 5.0):

```
java –ea AssertTest

class AssertTest {
        public static void main(String[] args) {
                // The following assert statement will stop execution
                // with a message if assertions are turned on.
                assert false : "Assertions are turned on.";

                // The following statement will only be printed if
                // assertions are turned off because  assertions
                // were not allowed at run time by the –ea parameter.
                System.out.println("Assertions are not active.");
        }
}
```

There are a few situations where you should not use assertions:

1. Do not use assertions for argument checking in `public` methods.

   Argument checking is typically part of the published specifications (or contract) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException`). An assertion failure will not throw an appropriate exception.

2. Do not use assertions to do any work that your application requires for correct operation.

   Because assertions may be disabled, programs MUST NOT assume that the boolean expression contained in an assertion will be evaluated. Violating this rule has dire consequences. For example, suppose you wanted to remove all of the `null` elements from a list names, and knew that the list contained one or more nulls. It would be wrong to do this:

   ```
   // Broken! – action is contained in assertion
   assert names.remove(null);
   ```

   The program would work fine when asserts were enabled, but would fail when they were disabled, as it would no longer remove the `null` elements from the list. The correct idiom is to perform the action before the assertion and then assert that the action succeeded:

   ```
   // Fixed – action precedes assertion
   boolean nullsRemoved = names.remove(null);
   assert nullsRemoved;  // Runs whether or not asserts are enabled
   ```

   As a rule, the expressions contained in assertions should be free of side effects: evaluating the expression should not affect any state that is visible after the evaluation is complete. One exception to this rule is that assertions can modify state that is used only from within other assertions.

There are many situations where it is good to use assertions:

1. Internal Invariants

Before assertions were available, many programmers used comments to indicate their assumptions concerning a program's behavior. You should now use an assertion whenever you would have written a comment that asserts an invariant:

```
if (i % 3 == 0) {
  ...
} else {
  if (i % 3 == 1) {
          ...
  } else {
          assert i % 3 == 2 : i;
          ...
  }
}
```

Another good candidate for an assertion is a `switch` statement with no `default` case. The absence of a `default` case typically indicates that a programmer believes that one of the cases will always be executed. The assumption that a particular variable will have one of a small number of values is an invariant that should be checked with an assertion:

```
default:
  assert false : suit;
```

If the `suit` variable takes on another value and assertions are enabled, the `assert` will fail and an `AssertionError` will be thrown.

An acceptable alternative is:

```
default:
  throw new AssertionError(suit);
```

This alternative offers protection even if assertions are disabled, but the extra protection adds no cost: the `throw` statement won't execute unless the program has failed. Moreover, the alternative is legal under some circumstances where the `assert` statement is not. If the enclosing method returns a value, each case in the switch statement contains a return statement, and no return statement follows the switch statement, then it would cause a syntax error to add a `default` case with an assertion. (The method would return without a value if no case matched and assertions were disabled.)

2. Control-Flow Invariants

   Place an assertion at any location you assume will not be reached. The assertions statement to use is:

```
assert false;
```

   Code now reads:

```
void foo() {
  for (...) {
          if (...) return;
  }
  assert false; // Execution should never reach this point!
}
```

3. Preconditions, Postconditions, and Class Invariants

While the `assert` construct is not a full-blown design-by-contract facility, it can help support an informal design-by-contract style of programming.

Do not use assertions to check the parameters of a `public` method. An `assert` is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, the `assert` construct does not throw an exception of the specified type. It can throw only an `AssertionError`.

You can, however, use an assertion to test a nonpublic method's precondition that you believe will be `true` no matter what a client does with the class.

You can test postcondition with assertions in both public and nonpublic methods.

A class invariants is a type of internal invariant that applies to every instance of a class at all times, except when an instance is in transition from one consistent state to another. A class invariant can specify the relationships among multiple attributes, and should be `true` before and after any method completes. For example, suppose you implement a balanced tree data structure of some sort. A class invariant might be that the tree is balanced and properly ordered.

The assertion mechanism does not enforce any particular style for checking invariants. It is sometimes convenient, though, to combine the expressions that check required constraints into a single internal method that can be called by assertions. Continuing the balanced tree example, it might be appropriate to implement a private method that checked that the tree was indeed balanced as per the dictates of the data structure:

```
// Returns true if this tree is properly balanced
private boolean balanced() {
  ...
}
```

Because this method checks a constraint that should be true before and after any method completes, each `public` method and constructor should contain the following line immediately prior to its return:

```
assert balanced();
```

It is generally unnecessary to place similar checks at the head of each `public` method unless the data structure is implemented by native methods. In this case, it is possible that a memory corruption bug could corrupt a "native peer" data structure in between method invocations. A failure of the assertion at the head of such a method would indicate that such memory corruption had occurred. Similarly, it may be advisable to include class invariant checks at the heads of methods in classes whose state is modifiable by other classes.

In order for the `javac` compiler to accept code containing assertions, you must use the `–source 1.4` command-line option as in this example:

```
javac –source 1.4 MyClass.java
```

This flag is necessary so as not to cause source compatibility problems.

NOTE, in Java 5.0 the compiler accepts assertions BY DEFAULT:

```
javac MyClass.java
```

In both Java 5.0, as in Java 1.4 assertions are disabled by default at runtime. You need explicitly to turn them on.

At a runtime you can check if the program is running with enabled assertions using the following code:

```
boolean assertsEnabled = false;
assert assertsEnabled = true;  // Intentional side-effect !!!
// Now 'assertsEnabled' is set to the correct value
```

2.4 Develop code that makes use of exceptions and exception handling clauses (try, catch, finally), and declares methods and overriding methods that throw exceptions.

try/catch and overridden methods

Write code that makes proper use of exceptions and exception handling clauses (try catch finally) and declares methods and overriding methods that throw exceptions.

An exception condition is a when a program gets into a state that is not quite normal. Exceptions trapping is sometimes referred to as error trapping. A typical example of an exception is when a program attempts to open a file that does not exist or you try to refer to an element of an array that does not exist.

The try and catch statements are part of the exception handling built into Java. Neither C/C++ nor Visual Basic have direct equivalents to Java's built in exceptions. C++ does support exceptions but they are optional, and Visual Basic supports On Error/Goto error trapping, which smacks somewhat of a throwback to an earlier less flexible era of BASIC programming.

Java exceptions are a built in part of the language. For example if you are performing I/O you must put in exception handling. You can of course put in null handling that doesn't do anything. The following is a little piece of code I have used with Borland/Inprise JBuilder to temporarily halt output to the console and wait for any key to be pressed.

```
public class Try{
import java.io.*;
        public static void main(String argv[]){
                Try t = new Try();
                t.go();
        }//End of main
        public void go(){
                try{
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr);
                br.readLine();
                } catch(Exception e){
                        /*Not doing anything when exception occurs*/
                } //End of try
                System.out.println("Continuing");
        }//End of go
}
```

In this case nothing is done when an error occurs, but the programmer must still acknowledge that an error might occur. If you remove the try and catch clause the code will simply not compile. The compiler knows that the I/O methods can cause exceptions and demands exception handling code.

Comparing with Visual Basic and C/C++

This is a little more rigorous than Visual Basic or C/C++ which allows you to throw together "quick and dirty" programs that pretend they are in a world where errors do not occur. Remember that the original version of DOS was called QDOS for Quick and Dirty DOS by it's creator and look how long we have lived with the legacy of that bit of hackery. By the time you have gone to the trouble of putting in a try/catch block and blank braces you may as well put in some real error tracking. It's not exactly bondage and discipline programming, it just persuasively encourages you to "do the right thing".

Overriding methods that throw exceptions

An overriding method in a subclass may only throw exceptions declared in the parent class or children of the exceptions declared in the parent class. This is only true for overriding methods not overloading methods. Thus if a method has exactly the same name and arguments it can only throw exceptions declared in the parent class, or exceptions that are children of exceptions in the parent declaration. It can however throw fewer or no exceptions. Thus the following example will not compile

```
import java.io.*;
class Base{
        public static void amethod()throws FileNotFoundException{}
}

public class ExcepDemo extends Base{
        //Will not compile, exception not in base version of method
        public static void amethod()throws IOException{}
}
```

If it were the method in the parent class that was throwing IOException and the method in the child class that was throwing FileNotFoundException this code would compile. Again, remember that this only applies to overridden methods, there are no similar rules to overloaded methods. Also an overridden method in a sub class may throw Exceptions.

The finally clause

The one oddity that you are likely to be questioned on in the exam, is under what circumstances the finally clause of a try/catch block is executed. The short answer to this is that the finally clause is almost always executed, even when you might think it would not be. Having said that, the path of execution of the try/catch/finally statements is something you really need to play with to convince yourself of what happens under what circumstances.

```
The finally clause of a try catch block will always execute, even
if there are any return statements in the try catch part
```

One of the few occasions when the finally clause will not be executed is if there is a call to

```
System.exit(0);
```

The exam tends not to attempt to catch you out with this exception to the rule.

The exam is more likely to give examples that include return statements in order to mislead you into thinking that the code will return without running the finally statement. Do not be mislead, the finally clause will almost always run.

The try/catch clause must trap errors in the order their natural order of hierarchy. Thus you cannot attempt to trap the generic catch all Exception before you have put in a trap for the more specific IOException.

The following code will not compile

```
try{
    DataInputStream dis = new DataInputStream(System.in);
    dis.read();
} catch (Exception ioe) {
} catch (IOException e) {
        //Compile time error cause
} finally{}
```

This code will give an error message at compile time that the more specific IOException will never be reached.

The throws clause

One of the issues created with the need to include try/catch blocks in code that may throw an exception is that you code can start to appear to more about what might happen than about what should happen. You can pass exceptions "up the stack" by using the throws clause as part of the method declaration. This in effect says "when an error occurs this method throws this exception and it must be caught by any calling method".

Here is an example of using the throws clause

```
import java.io.*;
public class Throws{
        public static void main(String argv[]){
                Throws t = new Throws();
                try{
                        t.amethod();
                }catch (IOException ioe){}
                }
        public void amethod() throws IOException{
                FileInputStream fis = new FileInputStream("Throws.java");
        }
}
```

2.5 Recognize the effect of an exception arising at a specified point in a code fragment. Note that the exception may be a runtime exception, a checked exception, or an error.

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. The `ThreadDeath` error, though a "normal" condition, is also a subclass of `Error` because most applications should not try to catch it.

A method is not required to declare in its `throws` clause any subclasses of `Error` that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Error
```

Exception

The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
```

RuntimeException

`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

A method is not required to declare in its `throws` clause any subclasses of `RuntimeException` that might be thrown during the execution of the method but not caught.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
```

2.6 Recognize situations that will result in any of the following being thrown: ArrayIndexOutOfBoundsException, ClassCastException, IllegalArgumentException, IllegalStateException, NullPointerException, NumberFormatException, AssertionError, ExceptionInInitializerError, StackOverflowError or NoClassDefFoundError. Understand which of these are thrown by the virtual machine and recognize situations in which others should be thrown programatically.

ArrayIndexOutOfBoundsException

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.IndexOutOfBoundsException
                  java.lang.ArrayIndexOutOfBoundsException
```

ClassCastException

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a `ClassCastException`:

```
Object x = new Integer(0);
System.out.println((String)x);

java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.ClassCastException
```

IllegalArgumentException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.IllegalArgumentException
```

IllegalStateException

Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.IllegalStateException
```

NullPointerException

Thrown when an application attempts to use `null` in a case where an object is required. These include:

- Calling the instance method of a `null` object.
- Accessing or modifying the field of a `null` object.
- Taking the length of `null` as if it were an array.
- Accessing or modifying the slots of `null` as if it were an array.
- Throwing `null` as if it were a `Throwable` value.
- Applications should throw instances of this class to indicate other illegal uses of the `null` object.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.NullPointerException
```

NumberFormatException

Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.IllegalArgumentException
                  java.lang.NumberFormatException
```

AssertionError

Thrown to indicate that an assertion has failed. The seven one-argument public constructors provided by this class ensure that the assertion error returned by the invocation:

```
new AssertionError(expression)

java.lang.Object
  java.lang.Throwable
      java.lang.Error
          java.lang.AssertionError
```

ExceptionInInitializerError

Signals that an unexpected exception has occurred in a static initializer. An `ExceptionInInitializerError` is thrown to indicate that an exception occurred during evaluation of a static initializer or the initializer for a static variable.

As of release 1.4, this exception has been retrofitted to conform to the general purpose exception-chaining mechanism. The "saved throwable object" that may be provided at construction time and accessed via the `getException()` method is now known as the cause, and may be accessed via the `Throwable.getCause()` method, as well as the aforementioned "legacy method."

```
java.lang.Object
  java.lang.Throwable
      java.lang.Error
          java.lang.LinkageError
              java.lang.ExceptionInInitializerError
```

StackOverflowError

Thrown when a stack overflow occurs because an application recurses too deeply.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Error
          java.lang.VirtualMachineError
              java.lang.StackOverflowError
```

NoClassDefFoundError

Thrown if the Java Virtual Machine or a `ClassLoader` instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found.

The searched-for class definition existed when the currently executing class was compiled, but the definition can no longer be found.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Error
          java.lang.LinkageError
              java.lang.NoClassDefFoundError
```