In order to analyze a program, it is necessary to know how the program is designed to perform. This information is usually expressed in statements called *assertions*. An assertion states a fact about the program that may be true or false. If the program is behaving correctly, then the assertion will be true. If an assertion is false, however, this signals that the program is not behaving as designed.

In the following code, for example, the programmer has added a comment after the while loop that makes an assertion about the value of n:

  //<code not shown>   
  
  int n = 0;   
  while ( k > 0 )   
  {   
    //<code not shown>   
  
    n++;   
  }   
  
  // At this point, n == count is true

Since n == count is a boolean expression, its value may be true or false. So the assertion "n == count is true" may be true or it may be false. Obviously, if n and count do not have the same value, then the assertion is false, and this may indicate that there is a problem somewhere in the code.

Such an assertion, in the form of a comment within a method definition, usually serves to remind the programmer — or whomever is later charged with the task of maintaining the code — of an essential piece of information that may not be obvious from a cursory inspection. Often, these assertions concern conditions that arise when data are close to or exceed the valid limit. A *boundary condition* is an assertion that describes the limit of valid data or inputs; a datum that is near (and potentially exceeds) the limit of valid data is called a *boundary case*. Boundary cases are used to help test that code segments are valid across the whole range of valid data. For example, in the following code segment, one boundary condition is that x must be non-negative; and boundary cases include values for x of -1, 0 and 1:

public static void B( ArrayList<String> t, int x )   
{   
   String y = t.get( x );   
  
   // ...   
}

On the other hand, if a programmer anticipates that an assertion could be false, he or she can signal a serious problem by deliberately throwing a *runtime exception*. In Java, a runtime exception is an object derived (directly or indirectly) from the RuntimeException class, and it is thrown using the keyword throw as shown in the following code:

public static void main( String[] args )

{

  int n = 2, k = 1;

  // <code not shown>

  // At this point, n == k should be true

  if ( n != k )

    throw new RuntimeException();

  // <code not shown>

  System.out.println( "Done." );

}

Exception in thread "main" java.lang.RuntimeException   
  
at MainClass.main(MainClass.java:14)

**Exercise 201**

1. The RuntimeException class has an alternative constructor that takes a single String argument. Modify the above code so that it tests the behavior when this second constructor is used, and suggest an argument that might help the programmer determine why the exception has been thrown.
2. In the following code, identify the boundary conditions for the input to the method chk. Modify the code so that an exception is thrown when boundary conditions are not met. Suggest boundary cases that should be used to test the code.

public static void main( String[] args )

{

  System.out.println( chk( 3 ) );

  System.out.println( "Done." );

}

public static int chk( int n )

{

  int[] indices = { 10, 6, 4, 3, 9, 14 };

  return indices[ n - 2 ];

}

6   
Done.

 Since programmers are usually interested in exactly what deviation from the expected behavior has caused the exception to be thrown, a String such as the one in the following code might be useful:

  if ( n != k )  
  {  
    String s = "Error: n = " + n + "; k = " + k;  
    s += ". They should be equal!";  
    throw new RuntimeException( s );  
  }

 The boundary conditions arise from the fact that chk returns a value that is obtained by accessing an array by an index. The index used must be greater than or equal to 0 and strictly less than the length of the array. In this case the index is n − 2, so the boundary conditions are that

n − 2 >= 0 && n − 2 < indices.length

or alternatively

n >= 2 && n < 2 + indices.length.

Since the indices array has length 6, suitable boundary cases to test would be 1, 2, and 3 (for the lower limit) and 6, 7, and 8 (for the upper limit).

Here is an edited version of the code that incorporates tests for the boundary conditions (throwing exceptions when they are not satisfied) as well as test cases for all the above boundary cases (test these one at a time by uncommenting them one at a time).

public static void main( String[] args )  
{  
  System.out.println( chk( 1 ) );  
  //System.out.println( chk( 2 ) );  
  //System.out.println( chk( 3 ) );  
  //System.out.println( chk( 6 ) );  
  //System.out.println( chk( 7 ) );  
  //System.out.println( chk( 8 ) );  
  //System.out.println( "Done." );  
}  
  
public static int chk( int n )  
{  
  int[] indices = { 10, 6, 4, 3, 9, 14 };  
  
  // n must be >= 2  
  if ( n < 2 )  
    throw new RuntimeException( "n = " + n   
                  + "; should be >= 2" );  
  // n must be < 2 + indices.length  
  if ( n >= 2 + indices.length )  
    throw new RuntimeException( "n = " + n   
          + "; should be < " + (2+indices.length) );  
  
  return indices[ n - 2 ];  
}

Exception in thread "main" java.lang.RuntimeException: n = 1; should be >= 2   
  
at MainClass.chk(MainClass.java:23)   
  
at MainClass.main(MainClass.java:8)

The keyword throw and its use are not in the AP Java subset. Consequently, you will not be tested on them. Even though you are not required to know how to throw exceptions yourself, you are expected to be able to interpret and appropriately react to any exceptions that the execution of a program may cause to be thrown.

Two common forms of assertion are method *preconditions* and *postconditions*. A precondition is an assertion that should be true at the moment when the method is called (and before the method invocation has caused anything to happen). A postcondition is an assertion that should be true when the method completes execution, provided that all the preconditions (if any) were true when the method was called and further provided that the method is correctly implemented. Together, a method's preconditions and postconditions form a contract that the method is designed to fulfill.

Typically, method preconditions are assertions about the method arguments, as in this example:

  // precondition: a is non-empty   
  public static void f( int[] a )   
  {   
    // <code not shown>   
  }

In this case, the method signature requires that the argument, a, must be an int[]. But the precondition reveals that the programmer has in mind the additional requirement that a must be non-empty. Because of the method signature, if a data type other than int[] is passed to this method, the Java compiler will complain by generating a compile-time error. However, unless the programmer deliberately includes code to check that the commented assertion is true, neither the Java compiler nor the Java virtual machine will complain if this method is called on an *empty* int[]. Admittedly, since the programmer has taken the trouble to include the commented assertion, we can expect that something will go wrong if an empty array is passed to this method, and that this misbehavior may result in an exception being thrown. (For example, if the method is called on an empty array and if its body attempts to reference an element of that array, then most likely an *ArrayIndexOutOfBoundsException* will be thrown.) However, such outcomes would usually only be *indirect* indications of the source of the problem.

When it comes to a programmer's attention that an inappropriate argument could disrupt the smooth operation of a program, he or she has the option of manually throwing a runtime exception to announce directly what has caused the problem. For preconditions, it is usual to throw an exception that is derived from the IllegalArgumentException class (which, in turn, is derived from RuntimeException). Here is an example:

// precondition: a is non-empty

public static void f( int[] a )

{

  if ( a.length < 1 )

    throw new IllegalArgumentException();

  // <code not shown>

}

public static void main( String[] args )

{

  int[] a = new int[ 0 ];

  f( a );

  System.out.println( "Done" );

}

Exception in thread "main" java.lang.IllegalArgumentException   
  
at MainClass.f(MainClass.java:10)   
  
at MainClass.main(MainClass.java:18)

Like all runtime exceptions, the IllegalArgumentException class has a second constructor that takes a String argument that is displayed whenever the exception is thrown. As you may have deduced from our solution to [Exercise 201](https://www.eimacs.com/eimacs/mainpage?epid=E2355861952&cid=162149#Exe192), it is helpful if this argument is a dynamically-constructed String that reports the actual condition that has been encountered. See what happens, for example, if, in the code above, the following is used as the argument to the IllegalArgumentException constructor:

"Error - the argument to f has length " + a.length

#### Exercise 203

Exceptions will be thrown by the following code unless the arguments to the method chain have certain qualities. Suggest suitable preconditions for chain, and then modify the code so that an informative exception is thrown when those preconditions are not met.

### Exercise 203

The following two conditions are necessary as far as the first argument to chain is concerned:

// precondition: s is non-empty and, if  
// s[ 0 ] is "fred", s.length is at least 2

In addition, the value of the second argument, n, must not be greater than the length of the first argument. A definition of chain that verifies that these three conditions are met is as follows. (The inserted code is highlighted on a yellow background.)

// preconditions: s is non-empty and, if  
// s[ 0 ] is "fred", s.length is at least 2;  
// n <= s.length  
public static String chain( String[] s, int n )  
{  
  if ( s.length == 0 )  
  {  
    String err = "chain's first argument is an empty array";  
    throw new IllegalArgumentException( err );  
  }  
  
  if ( s[ 0 ] == "fred" )  
  {  
    if ( s.length < 2 )  
    {  
      String err = "chain's first argument has first element ";  
      err += "\"fred\" and no other elements";  
      throw new IllegalArgumentException( err );  
    }  
    return s[ 1 ];  
  }  
  
  if ( s.length < n )  
  {  
    String err = "chain's second argument is greater than ";  
    err += "the length of its first argument";  
    throw new IllegalArgumentException( err );  
  }  
  String r = "";  
  for ( int i = 0 ; i < n ; i++ )  
    r += s[ i ];  
  
  return r;  
}

A list of the more common runtime exceptions is shown below. Each of them is derived (directly or indirectly) from the RuntimeException class. Moreover, each of these classes has two constructors, one with no arguments and one that takes a single String argument:

|  |  |
| --- | --- |
| **Runtime Exceptions** | |
| ArithmeticException | |
|  | Thrown when an exceptional arithmetic condition has occurred. For example, if an attempt is made to divide an integer by zero, an instance of this class will be thrown. |
| ArrayIndexOutOfBoundsException | |
|  | Thrown when an array is accessed with an illegal index, that is, an index that is either negative or greater than or equal to the length of the array. |
| ClassCastException | |
|  | Thrown when an attempt is made to cast an object to a subclass of which it is not an instance. For example, if an Object is initialized to an Integer and an attempt is then made to cast it to a String, an instance of this class will be thrown. |
| IllegalArgumentException | |
|  | Thrown when a method is passed an illegal or inappropriate argument. For example, if, according to its signature, a method expects an int as its argument and a double is passed to it instead, an instance of this class will be thrown. |
| NullPointerException | |
|  | Thrown when an attempt is made to treat a null object as if it were not null. For example, if an attempt is made to call the length instance method of a null object (as if it were a String), then an instance of this class will be thrown. |
| IllegalStateException | |
|  | Thrown when a method is invoked at an illegal or inappropriate time, that is, when the Java environment or Java application is not in an appropriate state for the requested operation. (See below this table for more information.) |
| NoSuchElementException | |
|  | Thrown when an Object that implements a certain kind of collection of objects uses an instance method that is designed to access the next object in the collection when in fact there are no more objects. (See below this table for more information.) |

As the above descriptions indicate, Java will automatically throw instances of many of these classes when circumstances require it. You have probably seen examples of all but the last two. You do not have to rely on this automatic behavior, however. It is perfectly acceptable for you to include code in your programs that will explicitly throw one or more of these runtime exceptions under circumstances that you specify.

The first of the last two exceptions, IllegalStateException, is intended for use when an instance method is called before the object is ready to execute it. For example, a family tree program might include the following code fragment:

public int getOldest()   
{   
  if ( family.length == 0 )   
    throw new IllegalStateException( "You must first add family members." );   
  
  // ...   
}

The NoSuchElementException is designed for use by certain kinds of data structures; we will see examples of its use in the [*Advanced Topics*](https://www.eimacs.com/eimacs/mainpage?epid=E2346028817&cid=162149#NoSuchElX) section of this course.