Chapter 25

Exceptions

Algorithm design can be tricky because the details are crucial. It may be straightforward to write an algorithm to solve the problem in the general case, but there may be a number of special cases that must all be addressed within the algorithm for the algorithm to be correct. Some of these special cases might occur rarely under the most extraordinary circumstances. For the algorithm to be robust, these exceptional cases must be handled properly; however, adding the necessary details to the algorithm may render it overly complex and difficult to construct correctly. Such an overly complex algorithm would be difficult for others to read and understand, and it would be harder to debug and extend.

Ideally, a developer would write the algorithm in its general form including any common special cases. Exceptional situations that should rarely arise along with a strategy to handle them could appear elsewhere, perhaps as an annotation to the algorithm. Thus, the algorithm is kept focused on solving the problem at hand, and measures to deal with exceptional cases are handled elsewhere.

Java's exception handling infrastructure allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face. This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.

An *exception* is a special object that can be created when an extraordinary situation is encountered. Such a situation almost always represents a problem, usually some sort of runtime error. Examples of exceptional situations include:

- Attempting to read past the end of a file
- Evaluating the expression `A[i]` where `i` $\geq$ `A.length`
- Attempting to remove an element from an empty list
- Attempting to read data from the network when the connection is lost (perhaps due to a server crash or the wire being unplugged from the port).

Many of these potential problems can be handled by the algorithm itself. For example, an `if` statement can test to see if an array index is within the bounds of the array. However, if the array is accessed at many different places within a method, the large number of conditionals in place to ensure the array access safety can quickly obscure the overall logic of the method. Other problems such as the network connection problem are less straightforward to address directly in the algorithm. Fortunately, specific Java exceptions are available to cover each of the above problems.

Exceptions represent a standard way to deal with runtime errors. In programming languages that do not support exception handling, programmers must devise their own ways of dealing with exceptional situations. One common approach is for methods to return an integer that represents that method's success. This kind of error handling has its limitations, however. The primary purpose of some methods is to return an integer result that is not an indication of an error (for example, `Integer.parseInt()`). Perhaps a `String` could be returned instead? Unfortunately, some methods naturally return `String`s (like `next()` in the `Scanner` class). Also, returning a `String` would not work for a method that naturally returns an integer as its result. A completely different type of exception handling technique would need to be developed for methods such as these.

Other error handling strategies are possible. The main problem with these ad hoc approaches to exception handling is that the error handling facilities developed by one programmer may be incompatible with those used by another. Another weakness of programmer-devised error detection and recovery facilities is that the compiler cannot ensure that they are being used consistently or even at all. A comprehensive, uniform exception handling mechanism is needed. Java's exceptions provide such a framework. Java's exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors. Exceptions are used in the standard Java API, and programmers can create new exceptions that address issues specific to their particular problems. These exceptions all use a common mechanism and are completely compatible with each other. Also, exceptions are part of the Java language specification, and so the compiler has been made to properly enforce the proper use of exceptions.

25.1 Exception Example

A simple example introduces Java's exception handling features. This example uses the following classes and client program:

- `Collection.Collection` (25.3),
- `Collection.CollectionFullException` (25.2),
- `Collection.InvalidCollectionSizeException` (25.1), and
- `CollectTest` (25.4).

The client code creates and uses a simple custom data structure called a *collection*. A collection has the following characteristics:

- It has three private instance variables:
  - an array of `Object` types; therefore, any reference (that is, nonprimitive) type can be stored in the collection
  - an integer representing a programmer-specified fixed maximum capacity of the collection
  - an integer keeping track of the current number of valid elements in the collection
- It has three public methods:
  - `insert()` allows client code to add elements to the collection
  - `toString()` (overridden method inherited from `java.lang.Object`) renders the contents of the collection in a form suitable for display
  - `isEmpty()` returns true if the collection contains no elements; otherwise, it returns false
  - `isFull()` returns true if the collection has no more room to insert more elements; otherwise, it returns false

- Its constructor accepts a single integer parameter indicating the collection's maximum capacity; initially a collection is empty

Client code can check to see if a collection is empty before trying to print it (although printing an empty collection is not harmful). Client code *should* always check to see that the collection is full before attempting to insert a new element. What if the client programmer fails to include the check for full before insertion? Without exceptions, an array access out of bounds runtime error occurs that terminates the program.

Should a programmer be allowed to create a collection with a zero or negative capacity? Such a collection is useless, so this should be disallowed.

Exceptions can be used to address the above two issues. Two custom exception classes are devised:

- `Collection.InvalidCollectionSizeException` (25.1) is used when client code attempts to create a collection instance of nonpositive capacity and
- `Collection.CollectionFullException` (25.2) is used when client code attempts to insert an element into a full collection.

The new exception classes are very simple:

```
package Collection;

public class InvalidCollectionSizeException extends Exception {}
```

Listing 25.1: `Collection.InvalidCollectionSizeException`—Thrown when creating a `Collection` with a nonpositive size

```
package Collection;

public class CollectionFullException extends Exception {}
```

Listing 25.2: `Collection.CollectionFullException`—Thrown when attempting to insert an element into a full `Collection`

Both exceptions extend class `Exception` but provide no additional functionality. This is not unusual as the main purpose of these definitions is to create new *types*. Notice that both `InvalidCollectionSizeException` and `CollectionFullException` are located in the `Collection` package.

Now that the exception classes have been examined, the `Collection.Collection` class (25.3) itself can be considered.

```
package Collection;
```



```
public class Collection {
    protected Object[] list;   //  Array of elements in the collection
    protected int currentSize; //  Current number of viable elements
    protected int maxSize;     //  Maximum number of elements
    public Collection(int size) throws InvalidCollectionSizeException {
        if ( size > 0 ) {
            list = new Object[size];  //  Allocate collection to maximum size
            maxSize = size;           //  Remember maximum size
            currentSize = 0;          //  Collection is initially empty
        } else {
            throw new InvalidCollectionSizeException();
        }
    }
    public void insert(Object newElement) throws CollectionFullException {
        if ( currentSize < maxSize ) {
            list[currentSize++] = newElement;
        } else {
            throw new CollectionFullException();
        }
    }
    public boolean isEmpty() {
        return currentSize == 0;
    }
    public boolean isFull() {
        return currentSize == maxSize;
    }
    public String toString() {
        String result = "[ ";
        for ( int i = 0;  i < currentSize;  i++ ) {
            result += list[i] + " ";
        }
        for ( int i = currentSize;  i < maxSize;  i++ ) {
            result += "- ";
        }
        return result + "]";
    }
}
```

Listing 25.3: `Collection.Collection`—A simple data structure that uses exceptions

Some notable features of the `Collection` class include:

- The `Collection` class is part of the `Collection` package, as are the custom collection exception classes.
- The constructor and `insert()` method have an additional specification between their parameter lists and bodies:

```
public Collection(int size) throws InvalidCollectionSizeException {
```



```
    . . .
}
```

and

```
public void insert(Object newElement) throws CollectionFullException {
    . . .
}
```

This *exception specification* indicates the types of exceptions that the method or constructor has the potential to *throw*. It specifies that the method can create a new instance of an exception object and pass it (throw it) up to client code that called the method. This has several important consequences to client code that uses such a method:

- Client code is thus warned that the method can throw the indicated type of exception.
- Client code can *catch* the exception object and deal with the problem in some reasonable way. The client code is said to *handle the exception*.
- Client code cannot ignore the possibility of the exception being thrown by the method; client code must do one of two things:
  - handle the exception itself or
  - declare the same exception type in its own exception specification (this basically means that the client code is not handling the method's exception but passing the exception up to the code that called the client code)

It is a compile-time error if client code calls a method with an exception specification but does not address the exception in some way.

The exception specification lists all exceptions that the method can throw. Commas are used to separate exception types when a method can throw more than one type of exception:

```
public void f(int a)
     throws ProtocolException, FileNotFoundException, EOFException {
    //  Details omitted . . .
}
```

Here, method `f()` has the potential to throw three types of exceptions. Any code using `f()` must ensure that these types of exceptions will be properly handled.

- The `throw` keyword is used to force an exception. Ordinarily a new exception object is created, then it is *thrown*. The act of throwing an exception causes the execution of the code within the method to be immediately terminated and control is transferred to the "closest" exception handler. "Closest" here means the method that is closest in the chain of method calls. For example, if method `main()` calls method `A()` which calls method `B()` which finally calls method `C()`, the call chain looks like

  `main()` → `A()` → `B()` → `C()`

  If method `C()` throws an exception and method `A()` has the code to handle that exception, `A()` is the closest method up the call chain that can handle the exception. In this case when `C()` throws the exception:

  1. `C()`'s execution is interrupted by the throw.
  2. Normally control would return from method `C()` back to method `B()` (this is the normal method call return path), but the throw statement causes the return to `B()` to be bypassed.

3. Any code following the call to `B()` within `A()`'s normal program flow is ignored and the exception handling code within `A()` is immediately executed.

Finally, `CollectTest` (25.4) shows how client code deals with exceptions.

```
import Collection.*;
import java.util.Scanner;

public class CollectTest {
    private static void help() {
        System.out.println("Commands:");
        System.out.println("   n <size>  Create a "
                           + "new collection");
        System.out.println("   i <item>  Insert new "
                           + "element");
        System.out.println("   p         Print "
                           + "contents");
        System.out.println("   h         Show this "
                           + "help screen");
        System.out.println("   q         Quit ");
    }
    public static void main(String[] args) {
        Collection col = null;
        //  Let the user interact with the collection
        boolean done = false;
        Scanner scan = new Scanner(System.in);
        do {
            System.out.print("Command?: ");
            String input = scan.nextLine();
            try {
                switch ( input.charAt(0) ) {
                    case 'N':
                    case 'n':
                        //  Create a new collection with a new size
                        col = new Collection
                                (Integer.parseInt(input.substring(1).trim()));
                        break;
                    case 'I':
                    case 'i':
                        col.insert(input.substring(1).trim());
                        break;
                    case 'P':
                    case 'p':
                        System.out.println(col);
                        break;
                    case 'H':
                    case 'h':
                        help();
                        break;
                    case 'Q':
```



```
                    case 'q':
                        done = true;
                        break;
                }
            } catch ( CollectionFullException ex ) {
                System.out.println("Collection full; nothing "
                                   + "inserted");
            } catch ( InvalidCollectionSizeException ex ) {
                System.out.println("Collection size must be positive");
                System.exit(0);
            } catch ( NullPointerException ex ) {
                System.out.println("Use N command to make a new collection");
            }
        } while ( !done );
    }
}
```

Listing 25.4: `CollectTest`—Exercises the `Collection` class

The expression

```
input.substring(1).trim()
```

returns a new string based on the original string `input`:

1. The call to `substring()` returns a substring of `input`. In this case the substring contains all the characters in `input` beginning at position `1` (which is the second character). Thus, if `input` was `"abcdef"`, then `input.substring(1)` would be `"bcdef"`.

2. The call to `trim()` removes all the leading and trailing whitespace from a string. For example, if string `s` is `" abc "`, then `s.trim()` returns `"abc"`.

If the user types in the string `"i fred"`, then

1. `input` is `"i fred"`

2. `input.substring(1)` is `" fred"`

3. `input.substring(1).trim()` is `"fred"`

Thus the first letter of the string (`input.charAt(0)`) is used to select the command, and `input.substring(1).trim()` is used if necessary to determine the rest of the command—how big to make the collection or what to insert.

`CollectTest` allows users to enter commands that manipulate a collection. When running the program, pressing H displays a help menu:

```
Commands:
   n <size>  Create a new collection
```



```
i <item>  Insert new element
p         Print contents
h         Show this help screen
q         Quit
```

A loop processes commands from the user until the Q command terminates the program. A sample session looks like:

```
Command?: n 5
Command?: p
[ - - - - - ]
Command?: i 10
Command?: p
[ 10 - - - - ]
Command?: i 20
Command?: p
[ 10 20 - - - ]
Command?: i fred
Command?: p
[ 10 20 fred - - ]
Command?: q
```

What are some potential problems to the user?

- Attempting to create a collection with a nonpositive capacity
- Attempting to insert an item into a collection before the collection has been created

Each of these problems should raise an exception and the client code must properly handle them should they arise.

The client-side issues of exceptions in `CollectTest` include:

- Code that has the potential of throwing an exception is "wrapped" by a `try` block. Almost all the code within `main()` is placed within `try { . . . }`. There are three potential exceptions `main()` must be prepared to handle:
  - `insert()` invoked by the I command can throw a `CollectionFullException` exception,
  - The collection creation statement invoked by the N command can throw an `InvalidCollectionSize-Exception` exception,
  - any qualified access using `col` can throw a `NullPointerException` if `col` is `null`; a qualified access is one in which `col` appears to the left of the dot (`.`) operator, as in

    ```
    col.insert(input.substring(1).trim());
    ```

  Notice that two of these exceptions are our custom exceptions, and one is a standard exception that we've seen before (See Section 7.4).
- Three `catch` clauses follow the `try` block. Each `catch` clause provides the code to mitigate a particular type of exception. In this case each remediation simply shows a message to the user, and program execution continues.

If the `try` block (and associated `catch` clauses) were omitted, the compiler would generate an error at the call of `insert()` and the new collection creation. This is because both the `insert()` method and the constructor declare that they can throw an exception in their exception specifications, but the client code would be ignoring this possibility.

25.2 Checked and Unchecked Exceptions

The `try` block could be omitted in `CollectTest` if only the `NullPointerException` were involved because `NullPointerException` is different from the other two exceptions. Its direct superclass is `RuntimeException`. All subclasses of `RuntimeException` are *unchecked exceptions*. The name `RuntimeException` and the term *unchecked exception* are a bit confusing:

- The name `RuntimeException` is an unfortunate choice, since *all* exceptions are generated at run time. The name comes from the fact that in general error checking can be performed at two separate times: compile time and run time. The compiler makes sure that the rules of the language are not violated (example errors include syntax errors and using an uninitialized variable). It can also check to see if provisions have been made to catch potential exceptions. The compiler can do this because some methods provide exception specifications. The compiler can verify that the invocation of a method with an exception specification appears only within a `try` block that provides a `catch` clause for that exception type. The only exception to this rule: exception types derived from `RuntimeException` are *not* checked at compile time.

  Like all exceptions, they arise at run time, so they are checked at run time only. Thus their name, `RuntimeException`.

- The term *unchecked exception* does not mean that the JRE does not check for these exceptions at run time. It means the compiler does not check to see if code that can throw exceptions of this type properly handles such exceptions. A method that can throw an instance of a subclass of `RuntimeException` is not required to declare that fact in its exception specification. Even if a method does declare it throws a subclass of `RuntimeException`, client code is not required to catch it. "Unchecked" thus means unchecked at compile time.

In contrast to unchecked exceptions, *checked exceptions*, derived from `Exception`, *are* checked at compile time. A method that can throw a checked exception *must* declare so in its exception specification. Client code that invokes a method that can throw a checked exception, such as

```
void f() throws E {
    //  Details omitted
}
```

must do one of two things:

1. it can wrap the invocation in a `try` block and catch that exception type

   ```
   void m() {
       try {
           f();
       } catch ( E e ) {}
   }
   ```

   or

2. it can defer the exception handling to its own caller.


```
void m() throws E {
    f();
}
```

Option 2 requires the client method to include the deferred exception type in its own exception specification.

`RuntimeException` subclasses represent common runtime errors in many programming language environments: out-of-bounds array access, division by zero, dereferencing a `null` reference variable, etc. Many of the `RuntimeException` errors do not arise from explicit method calls; they instead involve Java operators: subscript (`[]`), division (`/`), and dot (`.`).

The hierarchy of `Exception` classes is shown in Figure 25.1.

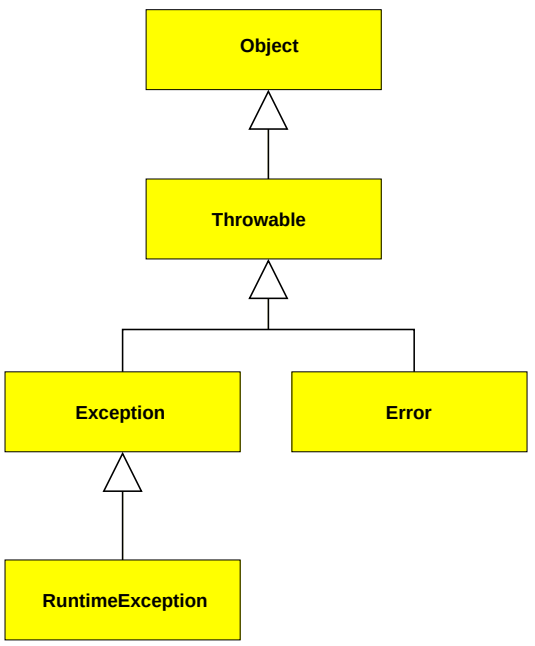

Figure 25.1: The hierarchy of `Exception` classes. These include:

- `Throwable`. The `Throwable` class is the superclass for all exception and error classes. It specifies the methods common to all exception and error classes.
- `Exception`. Programmers normally subclass the `Exception` class when creating custom exceptions. Any method that can throw an instance of a subclass of `Exception` must declare so in its exception specification. This means that client code must be prepared to handle the exception should it arise.
- `Error`. The `Error` class (and any programmer-defined subclasses) represent serious errors that clients are not expected to catch and handle. Programs should unconditionally terminate when an `Error` object is thrown.
- `RuntimeException`. This class and any programmer-defined subclasses represent unchecked exceptions. A method that can throw a `RuntimeException` or any of its subclasses is not required to declare so in its exception specification. Client code is not required to handle `RuntimeException`s. `RuntimeException`s may be caught by client code, but uncaught `RuntimeException`s will terminate the program.

Since most programmer-defined exceptions are derived from the `Exception` class, our focus here is on checked exceptions. The use of checked exceptions leads to more reliable code since the compiler demands that provision be made to handle them. While custom `RuntimeException` subclasses can also be created by programmers, such unchecked exceptions are not nearly as helpful to developers. Such unchecked exceptions may never arise during testing, but show up (by causing the program to terminate with an error message and stack trace) after the software is deployed. This is because client code is not required to catch them and client programmers have forgotten to catch them (or believed they never needed to catch them!). The exceptions may only arise due to unusual circumstances that were not modeled in any of the test cases. Checked exceptions are designed to avoid such problems and should be used instead of unchecked exceptions wherever possible.

25.3 Using Exceptions

Exceptions should be reserved for uncommon errors. For example, the following code adds up all the elements in an integer array named `a`:

```
int sum = 0;
for ( int i = 0;  i < a.length;  i++ ) {
    sum += a[i];
}
System.out.println("Sum = " + sum);
```

This loop is fairly typical. Another approach uses exceptions:

```
sum = 0;
int i = 0;
try {
    while ( true ) {
        sum += a[i++];
    }
} catch ( ArrayIndexOutOfBoundsException ex ) {}
System.out.println("Sum = " + sum);
```

Both approaches compute the same result. In the second approach the loop is terminated when the array access is out of bounds. The statement is interrupted in midstream so `sum`'s value is not incorrectly incremented. However, the second approach *always* throws and catches an exception. The exception is definitely *not* an uncommon occurrence. Exceptions should not be used to dictate normal logical flow. While very useful for its intended purpose, the exception mechanism adds some overhead to program execution, especically when an exception is thrown. (On one system the exception version was about 50 times slower than the exception-less version.) This overhead is reasonable when exceptions are rare but not when exceptions are part of the program's normal execution.

Exceptions are valuable aids for careless or novice programmers. A careful programmer ensures that code accessing an array does not exceed the array's bounds. Another programmer's code may accidentally attempt to access `a[a.length]`. A novice may believe `a[a.length]` is a valid element. Since no programmer is perfect, exceptions provide a nice safety net. In the `CollectTest` program (25.4), a prudent programmer would rewrite the insert case of `switch`:

```
case 'I':
case 'i':
    col.insert(input.substring(1).trim());
    break;
```


as

```
case 'I':
case 'i':
    if ( !col.isFull() ) {
        col.insert(input.substring(1).trim());
    }
    break;
```

This version would avoid the `CollectionFullException`. Since this check may be forgotten, however, the `CollectionFullException` provides the appropriate safety net. Since `CollectionFullException` is a checked exception, the `catch` clause must be provided, and the programmer is reminded to think about how the exception could arise.

Sometimes it is not clear when an exception is appropriate. Consider adding a method to `Collection` (25.3) that returns the position of an element within a collection. The straightforward approach that does not use exceptions could be written:

```
public int find(Object obj) {
    for ( int i = 0;  i < currentSize;  i++ ) {
        if ( list[i].equals(obj) ) {
            return i;  // Found it at position i
        }
    }
    return -1;  //  Element not present
}
```

Here a return value of -1 indicates that the element sought is not present in the collection. Should an exception be thrown if the element is not present? The following code illustrates:

```
public int find(Object obj)
                throws CollectionElementNotPresentException {
    for ( int i = 0;  i < currentSize;  i++ ) {
        if ( list[i].equals(obj) ) {
            return i;  // Found it at position i
        }
    }
    //  Element not there; throw an exception
    throw new CollectionElementNotPresentException();
}
```

In the first approach, an unwary programmer may not check the result and blindly use -1 as a valid position. The exception code would not allow this to happen. However, the first approach is useful for determining *if* an element is present in the collection. If `find(x)` returns -1, then `x` is not in the collection; otherwise, it is in the collection. If the exception approach is used, a client programmer cannot determine if an element is present without the risk of throwing an exception. Since exceptions should be rare, the second approach appears to be less than ideal. In sum,

- The first approach is more useful, but clients need to remember to properly check the result.
- The second approach provides an exception safety net, but an exception always will be thrown when searching for missing elements.

Which approach is ultimately better? The first version uses a common programming idiom and is the better approach for most programming situations. The exception version is a poorer choice since it is not uncommon to look for an element missing from the collection; exceptions should be reserved for uncommon error situations.

As you develop more sophisticated classes you will find exceptions more compelling. You should analyze your classes and methods carefully to determine their limitations. Exceptions can be valuable for covering these limitations. Exceptions are used extensively throughout the Java standard class library. Programs that make use of these classes must properly handle the exceptions they can throw.

25.4 Summary

- Add summary items here.

25.5 Exercises

1. Add exercises here.

