Designing Robust Classes

Learning Goals

You must be able to:

- ✓ **specify a robust data abstraction**
- ✓ **implement a robust class**
- ✓ **design robust software**
- ✓ **use Java exceptions**

Specifications and Implementations

Software that is robust performs well in a variety of situations. Object-oriented software design aims to structure a solution to a problem in terms of abstractions that model the problem domain. The key abstraction mechanism used to achieve such structure is data abstraction. We define a data abstraction used in an object-oriented program through a specification that focuses on specifying the *operations* provided by the data abstraction.

The specification for an operation defined on a data abstraction places constraints on the situations in which the operation may be used. For example, the `Animal` data abstraction from the Data Abstraction reading included an operation `recordLastFeeding` that required a list of the food eaten to not be empty. Here is the specification of just that operation in the `Animal` data abstraction:

```
// Remember what the animal last ate
// Requires: foodEaten is not an empty list
// Modifies: this
// Effects: this updated with foodEaten
public void recordLastFeeding(List<FeedingRecord> foodEaten)
```

This specification does not say what should or will happen if the operation is called in a situation where the list of food eaten *is* empty. If we can be assured that an implementation of this abstraction will be used only in cases where we can ensure that the parameter `foodEaten` is a non-empty list, such as when we expect to be the only users of the code, this specification is likely adequate. However, the specification may not be adequate if we want to place this abstraction in a library for use by other software systems. Remember the constraints in the REQUIRES clause cannot be enforced automatically by tools, such as a compiler, so we cannot ensure that when others use the abstraction the requirement that `foodEaten` is a non-empty list will be met. If we intend for the abstraction to be used by others, we may want to alter the specification to cover more situations so that implementations that satisfy the specification can operate robustly in more cases. In this reading, we will look at how to improve specifications to be more robust.

Even when a caller of an operation meets all constraints specified in the REQUIRES clause for the operation, there may be times when an implementation cannot behave as specified by the EFFECTS clause for the operation. Imagine our feeding system at the Vancouver aquarium has become more sophisticated and a suitably intelligent animal can now press on a device in their habitat to cause a feeding to occur. We might model this as a `feed` operation on `Animal`. Here is a potential specification for the `feed` operation:

```
// Feed the animal upon request
// Requires: time since last feeding is at least an hour
// Modifies: this
// Effects: correct amount of food is dispensed into the
//   animal's habitat and the time of the last feeding is
//   recorded
public void feed()…
```

Implementing this operation likely requires interacting with a physical system to dispense the appropriate amount of food. It is possible that this physical system might jam or otherwise break. In such a case, it would not be possible for the operation to complete successfully and provide the behaviour specified in the EFFECTS clause. As a result, the implementation is not robust. In this reading, we will also look at how to provide robust implementations of a specification.

Robust Operation Specifications

Let's go back to the first example we used in this course, the `isALeapYear` operation. Here is a specification for that operation:

```
// Determine if a specified year is a leap year
// Requires: year > 0
// Effects: True if specified year is a leap year and false
//   otherwise
boolean isALeapYear( int year)
```

Given this specification, an implementation written to this specification can assume that year is a positive, non-zero value.

But what if we cannot ensure that the value of year is greater than zero when the operation is called? An implementation that satisfies this specification might do or compute anything. If we want the specification to define a robust operation, we should ensure that the specification covers all values that could be passed to the operation.[1] Since `isALeapYear` already returns a value, one choice is to define the behaviour for all inputs in terms of this returned value.

```
// Determine if a specified year is a leap year
// Effects: Returns true if specified year is > 0 and
//   a leap year. Otherwise returns false.
boolean isALeapYear( int year)
```

In this case, we have removed any constraints about when the operation is executed and the EFFECTS clause says that the operation returns false if the given year is not a valid year (i.e., `year <= 0`).

In this situation, including the handling of the non-valid years in the return value might be reasonable. The answer to whether a non-valid year is a leap year should be no. When there are multiple input values to check or the state of the object on which the operation is called matters, this approach does not scale. For example, consider an operation that returns the number of days into a year when given the year, the month and the day of interest. The first specification you might write for such an operation might look like this:

```
// Determine the number of days into a year
// Requires: year is a valid year, month is a valid month and
// day is a valid day for the given month and year
// Effects: Returns the number of days into a year
int nthDayInYear( int year, int month, int day)
```

Given the input of the year 2010 and the 1st month (Jan) and the 1st day, this operation would return 1.

[1] From a mathematical perspective, an operation like `isALeapYear` that assumes an integer value will be greater than zero defines a partial function on the input domain of integers. A robust operation would consider all integer values and would be defined as a total function on the input domain of integers.

What if we decide to make the operation more robust to handle any possible integer value for year, month and day? Should we return 0 indicating that an illegal value (say -1) has been passed as the year? Should we return -1 indicating that an illegal value (say -1) has been passed as the month? Should we return -2 indicating that an illegal value (say 200) has been passed as the day? This approach makes the code that calls `nthDayInYear` complicated. Here is an example.

```
// Call the nthDayInYear operation checking for return values
// Imagine the variables year, month and day have had values
// set for them
int n = nthDayInYear(year, month, day);
if ( n == 0 )
  // do something about the illegal year
else if ( n == -1 )
  // do something about the illegal month
else if ( n == -2 )
  // do something about the illegal day
else
  // do whatever was intended originally because we now know
  // that n is a good value
```

A different approach is to be able to separate normal and unusual—referred to as *exceptional*—conditions. Many programming languages provide explicit support for the handling of such exceptional conditions through an *exception handling* mechanism. From a specification point of view, you can think about representing different exceptional conditions as different exception types. The name of the exception type helps convey what the exceptional condition is. When an operation wants to terminate abnormally and report an exceptional condition to the caller, the operation can *throw* an exception. In the next section, we will see more about how exception handling is supported by Java.

Returning to our example of the `nthDayInYear` operation, instead of using the return value to indicate exceptional conditions, we might instead introduce an exception type, called `IllegalValueException`, which indicates that a parameter has an illegal value.

```
// Determine the number of days into a year
// Effects: If any input parameter is not a valid value throws
//   IllegalValueException, otherwise returns the nth day in
//   the year for the specified year, month and day
int nthDayInYear( int year, int month, int day)
  throws IllegalValueException
```

This specification has removed any constraints on when the operation can be called (the REQUIRES clause is empty) and the EFFECTS clause states explicitly when the operation could respond to an exceptional condition by throwing a `IllegalValueException`.

You may be asking why we did not create a separate exception type for each kind of exceptional condition—one for an invalid year, one for an invalid month and one for an invalid day value. This approach is possible. We'll see in the next section how we can provide the caller of a method that encountered an exceptional situation information about the condition that occurred. You may also be wondering how throwing an exception makes the calling code any cleaner than the messy return value checking code we showed earlier. As the answer to both of these questions requires delving into some details of the exception handling mechanism, let's jump to looking at Java exception handling.

Java Exception Handling

Although the details of exception handling vary between programming languages, the basic concepts are the same. To make this discussion concrete, we will look at Java's exception handling mechanism.

In Java, exceptions are objects of an exception type. We'll see in a bit how we declare an exception type, but just assume for now you can create an exception type, such as `IllegalValueException`, to represent a situation, such as the passing of a non-positive integer value when a positive value is expected. Picking up from the last section, we have a method, `nthDayInYear` (in some class):

```
public int nthDayInYear( int year, int month, int day)
   throws IllegalValueException …
```

When we start coding `nthDayInYear`, we want to check that the values referred to by the year, month and day variables are positive values. If we encounter a value that is not as we expect, we can raise the exceptional condition by throwing an `IllegalValueException`. As exceptions are objects, this means that we need to create an object of type `IllegalValueException` and use the Java `throw` statement to cause the exception to be thrown. Here is a partial implementation of `nthDayInYear` that is checking the value of the year variable:

```
public int nthDayInYear( int year, int month, int day)
  throws IllegalValueException {
     if ( year <= 0 )
        throw new IllegalValueException("Year is <= 0");
     …
```

If the year value is less than or equal to zero, the `throw` statement is executed and execution in the method is immediately terminated. The flow of control returns to the calling method that either must handle the exception or the caller will be immediately terminated and control will pass to its caller and so on. A caller can provide a handler by embedding the call to the method with the potential exception in `a try-catch` construct. Here is an example of a caller of `nthDayInYear`:

```
public void someMethod() {
   try {
      …
      int n = nthDayInYear(-1, -1, -1);
      …
   } catch( IllegalValueException e ) {
       // Do something smart about the exceptional condition
       // ie, handle the exceptional condition
   }
  * …
}
```

Because the call to `nthDayInYear` is in a `try-catch` construct, when the exceptional condition of `IllegalValueException` is raised by `nthDayInYear`, control will return to the `try` block and flow to the matching `catch` handler. (We will see in a bit how there can be more than one `catch` handler for `a try-catch` construct.) The `catch` clause can try to do something to improve the exceptional condition. Flow of control after the statements in the `catch` block complete is to the statement after the `try` construct (marked with * in `someMethod()` above).

You may have noted above that when the `IllegalValueException` object was created in `nthDayInYear` as part of the `throw` statement, a string value was passed indicating which parameter value was invalid. This approach might be useful if the value came from a user interface where the message could be displayed and the user might input a better value. Since the `IllegalValueException` type is defined by the programmer, operations can be added to the type to record useful information from the exceptional condition, such as recording the invalid value, which can then be passed back to the caller as part of the object and accessed by the caller as part of trying to recover from the exceptional condition.

The above example outlines the Java exception handling mechanism but there are many more important details. We will cover a few here, but you are expected to familiarize yourself with the Exceptions part of the Java tutorials:

- http://download.oracle.com/javase/tutorial/essential/exceptions/index.html

Exception Types

Java has two kinds of exceptions: checked exceptions and runtime (or unchecked) exceptions. A checked exception that is thrown in a method must either be caught and handled within the method or the method must declare that it can throw the checked exception. For example, consider the `nthDayInYear` method:

```
public int nthDayInYear( int year, int month, int day)
  throws IllegalValueException {
     if ( year <= 0 )
        throw new IllegalValueException("Year is <= 0");
     …
```

If `IllegalValueException` is a checked exception and the implementation of `nthDayInYear` has a `throw` statement to throw that exception to a caller, then `nthDayInYear` must declare that the exception can be thrown (see the method signature where it states `throws IllegalValueException`). The Java compiler will check that the declaration of the exception in the method signature exists, if the exception can be thrown from the method. If that declaration is missing, the compiler will signal an error to the programmer. The Java compiler will also force a caller of `nthDayInYear` to either handle the exception or also declare that the exception can be thrown in its method signature. This checking ensures that a method must think about how an exception that reaches the method will be dealt with.

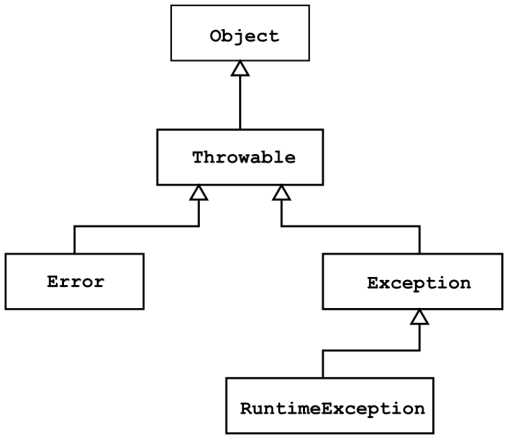

Figure 1 Java Exception Type Hierarchy

Some kinds of exceptional conditions can occur anywhere. For example, if the Java virtual machine runs out of memory as it runs your program, that exceptional condition may occur anywhere in the program. These kinds of exceptional conditions are unchecked exceptions. A method need not declare that it can throw an unchecked exception and the compiler will not warn you that an unchecked exception may occur within some method you are implementing.

Exception types in Java are organized into a type hierarchy rooted at the type `Throwable` (see Figure 1) defined in the Java library. User-defined exception types may be subtypes of either the Java type `Exception`, in which case the exception type is a checked exception, or of the Java type `RuntimeException`, in which case the exception type is a runtime (or unchecked) exception.

Defining a new exception type is as simple as declaring a class that extends either `Exception` or `RuntimeException`.

It can be difficult to decide whether to make an exception in your program checked or unchecked. If you choose to make the exception checked, you are requiring all methods that throw the exception to handle it or declare it in their method signature and so on all the way up through the callers of the method and their callers, etc., until either some method handles the exception or the program terminates. However, choosing to make

the exception type unchecked means that callers may not be aware the exception can occur.

Bloch (see References and Further Readings below) suggests that checked exceptions should be used for recoverable conditions, meaning that if a caller can potentially fix the problem and run the operation successfully later, then forcing the caller to potentially handle the exceptional condition makes sense. Bloch suggests that unchecked exceptions should be used for programming problems, including checks on the validity of input parameters.

What does this mean for our example of `IllegalValueException`? If we are in a system where we can imagine the caller, perhaps a graphical user interface, can fix the problem by reporting it to the user and getting a new value for year, day or month, we should make `IllegalValueException` a checked exception as in:

```
public class IllegalValueException extends Exception {
}
```

If we can't imagine a caller can fix the exceptional condition, `IllegalValueException` should extend `RuntimeException`.

There is no hard and fast rule here. See Bloch's book, Effective Java, for more pointers on how to choose which kind of exception type to create.

There are standard operations provided on `Exception` and `RuntimeException` that can be overridden. See the Java tutorials on Exception Handling for more details.

Multiple Handlers

A `try-catch` construct is not limited to one `catch` statement. It may have multiple `catch` statements. Imagine that `IllegalValueException` is declared to be a checked exception as shown above. We could have a piece of code such as:

```
int n;
try {
   n = nthDayInYear(-1, -1, -1);
} catch (IllegalValueException ive) {
   // Try again
   n = nthDayInYear(1, 1, 1);
} catch (Exception e) {
   // do nothing
} catch (Error err) {
   // do nothing for now
}
```

The call to `nthDayInYear` within the `try` block should cause a `IllegalValueException` to be thrown. When control returns to the `try` block, it goes through each `catch` block in turn, executing the code associated with the first `catch` block that matches the thrown exception. In this case, the first `catch` block will be executed as the actual type of the exception object, `IllegalValueException`, matches the type specified for the first catch block, `IllegalValueException`. The third `catch` block might be run if the program ran out of memory when `nthDayInYear` was executing causing a system-defined `OutofMemoryError` to be thrown.[2]

Robust Operation Implementations

Sometimes when we start implementing an operation, we realize that there are exceptional conditions that can arise because of how we have chosen to implement the operation. Remember that our `Animal` data abstraction from the Data Abstraction reading had an operation called `recordLastFeeding`. This operation needs to remember when an animal last ate. Here is the specification for the operation:

```
class Animal {

   // Remember what the animal last ate
   // Requires: foodEaten is not an empty list
   // Modifies: this
   // Effects: this updated with foodEaten
   void recordLastFeeding(List<FeedingRecord> foodEaten) {…}

  …
}
```

One way to implement this operation would be to remember the information about the last feeding in a field on the object. In the implementation below, the `feedingRecord` field keeps track in memory of what the animal has eaten and when. Note that this field is declared to be of type `Map`, which is from the Java Collections Framework (JCF). A `Map` stores at a key (in this case a `Date` object) a given value (in this case a `FeedingRecord`). Look up the API of `Map` on the web (web search with "Java 7 API

[2] `OutOfMemoryError` is a subtype of the Java `Error` type. As it is a subtype of `Throwable` it can be caught in a handler.

Map")[3] if you are interested in learning more. We will see it again several times in the term.

[3] The 1.6 ensures you get a recent version of the Java API. There were some major changes between version 1.4 and 1.6.

```
class Animal {

   private Map<Date,List<FeedingRecord>> feedingRecord;

   // Remember what the animal last ate
   // Requires: foodEaten is not an empty list
   // Modifies: this
   // Effects: this updated with foodEaten
   void recordLastFeeding(List<FeedingRecord> foodEaten) {
      feedingRecord.put(new Date(), foodEaten);
   }

  …
}
```

If for some reason the key or value prevents storage of the information in the `feedingRecord` field, the `Map` data type will throw an `IllegalArgumentException`. Unless we change the specification of the `recordLastFeeding` operation, this method will need to handle the exception and ensure the data is stored in the field as in:

```
class Animal {

   private Map<Date,List<FeedingRecord>> feedingRecord;

   // Remember what the animal last ate
   // Requires: foodEaten is not an empty list
   // Modifies: this
   // Effects: this updated with foodEaten
   void recordLastFeeding(List<FeedingRecord> foodEaten) {
      try {
         feedingRecord.put(new Date(), foodEaten);
      } catch (IllegalArgumentException iae) {
         // make sure the error is corrected and data stored
      }
   }

  …
}
```

Alternatively, we might decide that the specification of this operation, more specifically the EFFECTS clause, must change to include the possibility that the method may throw an `IllegalArgumentException`. Changing the specification may necessitate the need to visit each caller to ensure the caller can handle the change in specification.

In general, as you implement an operation abstraction, you need to consider how to ensure the implementation of the operation is robust. When possible, you want to handle exceptional conditions that a caller should not need to know about and that you can have the implementation recover from. Sometimes, you need to inform the caller and that might require changes to the specification.

Robust Data Abstraction Specifications

To this point in the reading, we have been considering how to make one operation at a time more robust. In object-oriented software, as you've seen, we are more interested in specifying and implementing data abstractions that involve multiple operations working together, sometimes operating on shared state (i.e., fields).

When we considered the specification of a data abstraction in the Data Abstraction reading, we focused on the specification of *operations*. Part of the specification for an operation included an EFFECTS clause that described what the operation alters when it executes. We saw how an EFFECTS clause could affect the state of the object on which the operation (method) may execute. For example, returning to the `Animal` abstraction with the `recordLastFeeding` operation:

```
// Requires: foodEaten must be a non-empty list
// Modifies: this
// Effects: this remembers foodEaten
void recordLastFeeding( List<FeedingRecord> foodEaten ) {…}
```

This operation specifies that as a result of its execution, the object on which the operation occurred has its state altered to remember the food that was eaten.

A more complete specification for a data abstraction also includes clauses about what must always be true—what is *invariant*—for an instance of the data abstraction. In object-oriented software, these clauses are sometimes referred to as a *class invariant*. A class invariant must hold before a method (operation) is executed on an object of that class and must hold immediately after that method finishes execution. As a result, one can think of the class invariant as being added to the REQUIRES clause for each operation and the EFFECTS clause for each operation.

Continuing on with the `Animal` example, imagine that our data abstraction must support returning the number of times the animal has been fed.

```
class Animal {

   // Return the number of times the animal has been fed
   // Effects: Returns a number greater than or equal to zero
   int getTimesFed() {…}

   …
}
```

An invariant for the class may be that the number of times the animal has been fed is always greater than or equal to zero. We can specify this invariant in terms of the `getTimesFed` operation:

```
class Animal {

   // Return the number of times the animal has been fed
   // Effects: Returns a number greater than or equal to zero
   int getTimesFed() {…}

   …
   // Class invariant:
   //   getTimesFed() >= 0
}
```

Conceptually, this class invariant must now be true whenever any operation is called and whenever any operation completes successfully. By specifying the class invariant, we can make more explicit what an operation on the abstraction can assume about an instance (object) of the abstraction and what it must ensure about an instance of the abstraction. Making these assumptions clear helps define potential exceptional cases for the data abstraction and helps ensure the implementation is a correct implementation of the specification.

We can use the Java `assert` statement to help check the class invariant. The simple form of the `assert` statement is:

```
assert Expression1;
```

If $\texttt{Expression}_1$ evaluates to true, control flows to the next statement in the method. If $\texttt{Expression}_1$ is false, an `AssertionError` exception is thrown.

If we encode the class invariant into a method on the class, such as

```
class Animal {

   private void hasValidState() {
      assert getTimesFed() >= 0;
   }
}
```

then we can code each implementation of an operation to check this invariant at the start and end of methods. For example, we might write

```
class Animal {
   …
   public void recordLastFeeding( List<FeedingRecord> foodEaten ) {
      hasValidState();
      // do stuff
      hasValidState();
   }
}
```

If for some reason, a method on the `Animal` class breaks the invariant by erroneously setting the number of times an animal is fed to -1, this invalid state will be identified.

If an `AssertionException` (an unchecked exception) is raised during the checking of the invariant, the method may choose to try to handle it or pass it on to the caller. `Assert` statements can also be used to check pre-conditions and post-conditions. In general, it is not necessary to check the invariant at the start of a method if all methods ensure the invariant holds when they complete execution. You can find lots of information on class invariants and asserts on the web, including better ways to express and check invariants. I've chosen to show a simple form to express the basic idea in this reading.

Designing Robust Software

A robust software system is able to run in a variety of conditions. McConnell has described a robust system as "garbage in does not mean garbage out" [2, p. 97]. How do we design a system to be robust? Unfortunately, there is no perfect answer. In very general terms, there are two basic approaches.

First, programmers can choose to code defensively, putting checks everywhere. For instance, each operation (method) can check that each value passed as a parameter is within range and that an object is in an appropriate state before the actual operation is performed. This approach helps ensure that the operations, and the data abstractions composed from the operations, can work in many situations. This style of programming can help create reusable software. The downside of this style of programming is that the code can become polluted with multiple checks of the same value. Sometimes, this can slow the program down. It almost always makes the program code harder to read.

An alternate approach is to program according to design-by-contract, which is also referred to as programming-by-contract. The basic idea is to state, as we have seen through our specification readings, what an operation (supplier) expects from a caller (client). For example, the supplier might simply state that the client must ensure an integer parameter value is greater than zero. Once this is stated, the supplier is under no obligation to check the value. This approach can work well with appropriate tool support. For example, the Eiffel programming language allows pre-conditions (REQUIRES clauses) to an operation to be stated in the programming language. Tool support can then be enabled to automatically check the pre-conditions (and post-conditions) as the code is being developed and tested. When the system is deployed for use, these checks can be turned off so that the system runs more efficiently. Similar tool support is available for many other languages, including Java. The assert statement in Java we saw above can also be used but is not easily turned off and on.

Creating a robust software system typically requires some planning from the start of the design of the system. As you see code through the term, consider whether or not it is robust and how it might be improved to be more robust.

References and Further Reading

Material in this reading is based on:

[1] Program Development in Java: Abstraction, Specification and Object-oriented Design by Barbara Liskov with John Guttag. Addison-Wesley, 2001.

Liskov is the 2008 winner of the ACM A.M. Turing Award which is the most prestigious award in computer science. She was awarded this honour for foundational innovations in programming language design.

This book provides a more comprehensive description of abstraction mechanisms and the kinds of abstractions if you are interested in gaining a deeper understanding of this material.

[2] Code Complete by Steve McConnell. Microsoft Press, 1993.

For further reading on design principles for Java exceptions:

- Effective Java by Joshua Bloch.

For further reading on asserts and class invariants:

- http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html#class-invariants
- http://www.stanford.edu/~pgbovine/programming-with-rep-invariants.htm
- And many others

For further reading on exception handling, read the following section of the online Java tutorial:

http://download.oracle.com/javase/tutorial/essential/exceptions/index.html