Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

When Things Go Wrong: Exceptions

Chapter Overview

- What happens when something goes wrong?
- How do I create alternate ways to handle atypical circumstances?

This chapter covers mechanisms for dealing with atypical behavior. Sometimes, exceptional circumstances arise and require different mechanisms to cope with them. In this case, the normal entity-to-entity communication in your system may need to be interrupted. Java provides certain mechanisms for creating alternate paths through your community. These include the throw and catch statements as well as special Exception objects that keep track of these atypical circumstances.

This chapter includes sidebars on the syntactic and semantic details of throw and catch statements, exception objects, and the requirement to declare exceptions thrown. It is supplemented by portions of the reference charts on java methods and statements.

Objectives of this Chapter

1. To be able to read, understand, and write `throws` clauses as a part of interface and class contracts.
2. To learn how to `throw` and `catch` Exceptions and other Throwables.
3. To appreciate the role that anticipating exceptional circumstances plays in the design and testing of programs.

Exceptional Events

So far, the code that we have written has addressed "normal" situations in which nothing goes wrong. But sometimes, unusual things happen in our code, and we have to deal with them. In some cases, these unusual things are unexpected errors; in others, their existence is predictable but we may not know in advance when they are likely to happen. An example of this second kind is a network outage, which happens from time to time and can reasonably be anticipated, but is unexpected when it occurs. Planning for these exceptional circumstances and writing code that can cope with them is an important part of robust coding.

When Things Go Wrong

Consider the following example, drawn from the StringTransformers application of the first interlude. In that scenario, entities called StringTransformers are connected by "tin can telephone" entities called Connectors. Each Connector has an end that you can put something into and an end that produces what you put into it. A StringTransformer can write to (or read from) a connector if it is holding the appropriate end. In the user interface of that application, there is a way that a user can specify two StringTransformers to be connected. We are going to look in more detail at how the Connector actually gets attached to these two StringTransformers.

Let's say that the two transformers we're going to connect are transformerA and transformerB. In the code that is making the connection, we invoke the specific Connector constructor with these transformers as arguments:

```
          new StringConnector( transformerA, transformerB )
```

The constructor code for StringConnector asks each of the transformers, in turn, to accept a(n input or output) connection. In fact, strictly speaking, A and B need not be StringTransformers at all; they need only implement the OutputAcceptor or InputAcceptor interfaces, since that is the only aspect of their behavior that we rely on here.

```
     public StringConnector( OutputAcceptor a, InputAcceptor b )
     {

          a.acceptOutputConnection( this );
          b.acceptInputConnection( this );

     }
```

This code is perfectly reasonable assuming that everything goes right. But what happens if transformerA already has an outputConnection in place? It might be that transformerA is a Broadcaster or AlternatingOutputter or some other kind of transformer that can have many outputConnections. It might be that transformerA is willing to throw away its existing outputConnection and replace it with the one currently on offer. But it might also quite reasonably be that transformerA is unwilling and unable to accept an OutputConnection if it already has one in place. In this case, the StringConnector constructor code is in trouble.

This is precisely the sort of situation that we will deal with in this chapter. Something has gone wrong. We can anticipate in our design that this might happen. We want our code to respond appropriately. In other words, we want to design our programs to be able to handle exceptional circumstances.

Expecting the Unexpected

When you are designing a program, it is relatively easy to think about what is supposed to happen. You can act out the interactions that you want your program to have. You can draw out storyboards describing what comes next. You design interfaces and protocols to describe the roles each entity plays and the contracts it makes with others. But this is not enough.

In addition to figuring out what *ought* to happen, you also need to anticipate what *might* happen. That is, you need to understand what happens if a component does something unexpected; if the user does something foolish; if a resource that you depend on becomes unavailable or temporarily out of service; or even if a change that you make to your code inadvertently violates an assumption. In all of these cases, unexpected behavior of one portion of the system needs to be dealt with. Good design involves anticipating these possibilities and explicitly deciding what to do and designing for these circumstances.

Exceptional circumstances can be partitioned into three groups. One is the catastrophic failure. In case of a catastrophic failure, there's really nothing that your program can do. This might happen, for example, if someone tripped over the power cord of the computer on which your program was running. In this case, it is reasonable to expect that your computer program will stop executing immediately. There's really nothing that you can do about a catastrophic failure.[Footnote: At least at the time of failure. There are still things that you can do to plan for recovery from catastrophic failure. For example, a banking system may temporarily lose the functioning of an ATM, but it will not lose track of your bank balance entirely. It has been designed to keep this information safe even in the face of computer crashes.]

The second kind of exceptional circumstance is at the other end of the spectrum. This is a situation that is not the intended course of your program, but is so benign that it is dealt with almost as a matter of course. These are the unexpected situations that can be handled with a simple conditional or other testing mechanisms. For example, if we are about to perform a division operator, we might check to make sure that the divisor is not 0. In the extreme, these situations can be difficult to distinguish from "normal" operation.

Most exceptional circumstances fall between these two extremes. That is, they admit some intervention or even solution (unlike catastrophic failure), but handling these circumstances requires cooperation among entities or other addtional complexity; it isn't possible or desirable to deal with this situation locally. These are the situations that you must take into account in your design.

When you are planning your program, you will be deciding how to partition the problem among a community of interacting entities and designing how these entities interact. At this point, you should also ask:

- What should happen if one of these entities is unreachable?
- What are all of the ways in which an entity might violate expectations?
- What should happen in each of these cases?
- What should an entity do if it has difficulty fulfilling its contract?

In each of these cases, you should decide whether the circumstance amounts to a catastrophic failure or can be handled by another entity. If it is a catastrophic failure, this circumstance ought to be documented; if not, it provides another set of interactions to build into your system. This exception-handling becomes another part of your system design.

As you break each entity down -- asking what is inside it, decomposing it into further communities of interacting entities -- you should repeat these questions with respect to these entities' mutual commitments. Eventually, you will decompose your problem to the level of individual operations and of interactions with entities outside the system that you are actually building. For these situations, you should ask:

- In what ways might this operation or outside entity fail?
- How else might it violate my expectations?
- Can I test for these circumstances prior to invocation of this operation or resource?
- What should I do if the failure or expectation violation occurs?

If the situation is one that can be ruled out using a simple test -- such as checking for a zero divisor or verifying that the user's input is a legal value and asking for new input if not -- such **error checking** should be introduced into your design. This strengthens the contracts that entities make with one another. Where violations cannot be handled locally, you will need to decide who should handle the issue and how it should behave.

What's Important to Record

At the time that an exceptional circumstance arises, the currently executing code is in the best position to determine what the problem is. It should take pains to record any information that might help other parts of the program (or a human user or debugger) to figure out what happened. So, for example, in the case of a divide-by-zero error, it would be important to know what the expression was whose value was zero, causing the error. In the case of an invalid value entered by a user, it may be important to know what the invalid value is or what the legal values might be. It is also important to know what *kind* of thing went wrong: division by zero or illegal argument passed to a method or a label name that's null and shouldn't be or any of a whole host of possible values.

This information -- what kind of thing went wrong and kind-specific additional information that might be useful for figuring out what the problem was or correcting it -- is, in Java, encapsulated in a special kind of object. These are Exception objects. They signal what's gone wrong. There are many different (more specific) types of Exception objects, such as NullPointerException or IllegalArgumentException. You can also define Exception types of your own (using inheritance). In addition to Exceptions, Java also defines a (similar but distinct) class of Errors, meant to designate conditions of catastrophic failure, such as NoClassDefFoundError. You can (but rarely will) define your own Errors as well.

Since Exceptions are objects, you can use them like any other object. If you define your own Exception classes, you can add any fields or methods that you think might be important to allow your program to handle the exceptional circumstance. One thing that is especially useful for an Exception to have is a String (suitable for printing to a user) that explains something about what has gone wrong. In Java's Exception classes, such a String can be supplied to the constructor and retrieved using the instance's getMessage() method.

It can also be very important to know where the problem occurred. Java's Exception classes record the point at which they were thrown (see below), but it can in addition be useful to record (e.g., in the message or in an additional field that you define) some program-specific indication of which code is reporting the exceptional circumstance and what it was trying to do when the exception occurred.

For example, in our OutputAcceptor code, we might recognize that we can't accept an OutputConnection if we already have one. In this case, we might create a new ConnectionRejectedException recording this circumstance:

```
new ConnectionRejectedException( this.toString()
              + " rejecting redundant OutputConnection" )
```

The ConnectionRejectedException uses the toString() method of the OutputAcceptor within which this code occurs to record who is rejecting the connection. An alternative is just to list the class name and method in a constant String: `"OutputAcceptor.acceptOutputConnection(): "`. The

ConnectionRejectedException might also record the existing OutputConnection and the newly supplied one; in the code fragment above, it does not do this.

Just defining a new exception isn't enough, though. Defining an exception is like composing a letter of complaint. In order for it to have any effect, you have to send out the letter. In the case of an Exception, this is accomplished by **throwing** the Exception.

Throwing an Exception

An Exception is an unusual circumstance that requires special handling. In order to understand how an Exception works -- and what it means to throw one -- we first need to look at how method invocation and return normally works.

Let us begin by looking more closely at what happens in our new Connector example, when the user interface calls the StringConnector constructor, which in turn calls the OutputAcceptor's acceptOutputConnection method. We might diagram the normal control flow as follows:

```
    User Interface        Constructor        OutputAcceptor
         -------------------->
                              ------------------->
                                             (records Connection)
                              <-------------------
                      (more activity)
         <--------------------
```

The code from the user interface invokes the StringConnector constructor, then the StringConnector constructor invokes the OutputAcceptor's acceptOutputConnection method. When the acceptOutputConnection method completes, it returns (nothing) to the StringConnector's constructor, which completes its work and provides the newly constructed StringConnector to the User Interface. These arrows are sometimes called the **call path** (and return path) of this execution.

Communication among pieces of code is very simple. Each piece of code can only talk to the other pieces of code about which it knows. In this case, the User Interface knows about the StringConnector's constructor, and the StringConnector's constructor knows about the OutputAcceptor's acceptOutputConnection method. Think of it like an old-fashioned fire-fighting bucket brigade. All of the people line up from the water supply to the fire. A full bucket is passed from hand to hand down the line from the water supply to the fire. The empty bucket must be passed back the same way. In the normal motion of buckets, there is no way for a bucket to skip over a person; it must be passed from hand to hand, returning the way that it came.[Footnote: In this example, the "more activity" line inside the constructor is a shorthand for a more complex picture. This "more activity" actually involves another method call, this one to the InputAcceptor's acceptInputConnection method. So the whole picture is more accurately represented as

```
   User Interface      Constructor      OutputAcceptor     InputAcceptor
        ------------------->
                            ------------------>
                                     (records Connection)
                            <------------------
                            ------------------------------------->
                                                        (records Connection)
```

```
                         <--------------------------------------
         <-------------------
```

This doesn't violate the bucket brigade idea, but it does mean that the bucket brigade has a fork in it. The constructor can pass buckets to (i.e., invoke) both the OutputAcceptor's acceptOutputConnection method and the InputAcceptor's acceptInputConnection method.]

[Insert bucket brigade (throw vs. normal method invocation) pic]

Throwing an Exception is different. What happens in this case looks more like the following:

```
User Interface        Constructor        OutputAcceptor
     -------------------->
                             -------------------->
                                                OH NO!!
```

When the OutputAcceptor's acceptOutputConnection method realizes that it has a problem it generates an Exception object, as we have seen above. Then, it throws the exception as hard as it can back the way it came. The Exception zooms back along the call path, flying too fast to stop and execute any statements waiting for its return. In fact, the Exception keeps going until it encounters a compatible **catch** statement. If necessary, it may exit several method bodies. Or, if the catch is in the same block as the throw, it may not exit any method bodies at all. In other words, a throw statement sets an Exception flying, and the flying Exception can only be caught by a matching catch statement; no other intervening statement along the call path matters.

This is, in fact, just what we want. If the OutputAcceptor can't accept an output connection, we don't want the rest of the Constructor to execute. For example, we don't want it to try to convince the InputAcceptor to accept the input end of the connection, because *this* connection isn't going to work out (since the OutputAcceptor isn't cooperating) and if the InputConnector accepts this one, then it won't later be able to accept a fully operational input connection. So when the OutputAcceptor decides that it has a problem, we want the Exception to propogate all the way back to the user interface code, which should decide that connecting this particular pair of String Transformers may not be such a good idea after all.

The code for the OutputAcceptor might look like this:

```
                            ... implements OutputAcceptor
{

     private OutputConnection out;

     public void acceptOutputConnection( OutputConnection out )
     {
         if ( this.out == null )
         {
             this.out = out;
         }
         else
         {
             throw new ConnectionRejectedException( this.toString()
                          + " rejecting redundant OutputConnection"
                                                 );
```

```
                }
            }
```

This example introduces a new statement type, `throw`, and a new declaration element, `throws`. (Note the s on the declaration element.) The throw statement works just as we have described; it abruptly terminates the execution of this method and causes the Exception to propogate backwards along the return path until a compatible catch statement is encountered. (We will see this below.)

What about the throws clause? Throwing an exception is actually part of the contract that one object makes with another. It is as much a part of a method's contract as its (normal) return type or the parameters it needs. So a method must declare that it may throw an exception (and what type of exception it may throw). This way, anyone calling the method knows to be prepared for it to throw this exception. The throws clause is the final part of a method signature, and throws clauses may appear in interface (abstract) method declarations as well as in method definitions.

Throws clauses are not restricted to methods. Constructors, too, must declare any exceptions that they throw. A constructor can explicitly throw an exception using a throw statement. A constructor (or method) can also throw an exception by calling something that throws an exception and then not catching it. This is what happens with the StringConnector constructor. Here it is, reprinted from above, with the added `throws` clause underlined.

```
public StringConnector( OutputAcceptor a, InputAcceptor b )
                              throws ConnectionRejectedException
{

     a.acceptOutputConnection( this );
     b.acceptInputConnection( this );

}
```

The StringConnector constructor invokes OutputAcceptor's acceptOutputConnection method. If the OutputAcceptor doesn't accept the output connection, the StringConnector constructor isn't going to be able to fix this. So the StringConnector constructor should itself exit abruptly. In other words, the Exception thrown by acceptOutputConnection flies right out of the StringConnector constructor as well, still waiting to find a compatible catch clause.

Throw Statements and Throws Clauses

A `throw` statement looks a lot like a `return` statement, but it always takes an argument (which can be in parentheses or not), and its argument must be something legal to throw. Anything that `extends Throwable` is legal to throw. In particular, this includes anything that `extends Exception`.

The effect of a `throw` statement is that execution abruptly returns up the call path until a compatible `catch` clause is encountered. Nothing except a compatible `catch` clause can stop the propogation of a thrown object.

If an Exception (except a RuntimeException) is thrown and not caught within a method or constructor body, you must also declare that that method or constructor `throws` the exception. This is a part of the signature, like saying what a method`return`s or what arguments a method or constructor expects.

The `throws` clause appears after the argument list, but before the method/constructor body. The syntax for a `throws` clause is

```
throws ExceptionType1, ExceptionType2, ... ExceptionTypeN
```

Every exception thrown and not caught within the body must match (at least) one of the exception types declared thrown by the method or constructor. If the method or constructor throws only a single exception type, the list contains no commas.

Catching an Exception

We have seen how an Exception can be generated and thrown. We have also seen that a thrown exception keeps flying until it encounters a compatible catch statement. Now, we will look at catch statements and how they work. This code introduces new syntax: the `try/catch` statement type. If `throws` is syntactically like `return`, `try/catch` is a bit like `if/else`.

A catch statement is properly a try/catch statement (or even more properly a try/catch/finally statement). If you are about to execute a statement that might throw an exception that you'd like to catch, you must first enter a try block. This is just like a regular block, except that it is preceded by the Java keyword try. This notifies Java that exceptions may be thrown and that it should be on the lookout for the ones that you want to catch.

At the end of the possible-exception-throwing code, you end the try block and introduce a catch clause. A catch clause contains a parameter declaration of the type that you wish to catch. The catch clause has a block that describes the instructions to execute if one of these is caught.

For example, the code in the user interface that is trying to connect transformerA (here named by `to`) and transformerB (here named by `from`) might say:

```
try
{
    new StringConnector( to, from );
}
catch ( ConnectionRejectedException e )
{
    Console.println( "Sorry, can't make that connection. "
                     + "Please try again." );
}
```

- This `try/catch` statement type has two bodies: one after the keyword `try`, and one after the `catch` parameter.

- The `try` body is a statement or set of statements that may throw an exception. In this case, we know that the `StringConnector` constructor may throw `ConnectionRejectedException`. We can tell this from its declaration, and so can the Java compiler.
- The `catch` portion of the statement has a single parameter, the exception (type and name) that is to be caught. In this case, the exception type is `ConnectionRejectedException`, and the name of the exception is `e.` The name is required, and it may be used inside the `catch` body, just like a method parameter name can be used inside the method body. It is common to name the exception `e`, though there's no particular reason for it; it's just like loop variables are often named `i`.
- The `catch` body contains statements which are executed if and only if the appropriate type of exception is thrown. (The "appropriate type" is the type of the `catch`'s parameter.) Inside the `catch` body, the parameter name may be used to refer to the exception, though there isn't a *whole* lot you can do with an exception other than print its message.

In this case, once the exception is caught, a message is printed to the user. This statement might itself appear inside an animate object's act method, so that something is continually listening to the user and trying to make connections on the user's behalf. This message lets the user know that this particular attempt didn't work. If we had supplied additional information along with the exception, we might use it at this point to give the user more information (perhaps flashing the object that refused the connection) or to try to repair the situation (asking whether the user means to delete the existing connection, for example, and then retrying the connection creation).

One `try` can actually have several `catch` statements. In this case, once something is thrown inside the `try` body, it is compared against the `catch` parameter statements in order until one that matches is found. If a match is found, only the first matching `catch` body is executed; then control continues at the end of the `try/catch` statement. If no match is found, the thrown object continues exiting statement blocks until a corresponding `catch` is found.

For completeness's sake, it is worth mentioning that a `try/catch` statement can have a `finally` clause (so that it's really `try{}catch(){}finally{}`). In this case, no matter how the statement is exited -- regardless of whether something is thrown, and regardless of whether the thrown object is caught -- the `finally` statement *will* be executed. At this point, you shouldn't need to be using `finally`, but if you ever need to know, the gory details are included in the Java language specification.

Try Statement Syntax

A `try/catch/finally` statement has a body after try, a body after each catch clause, and a body after the finally clause if it is present. Each of these bodies is a normal block executed according to the usual block execution rules. If a catch block is executed (i.e., if a matching throwable has been caught), the catch parameter is bound to the caught object during execution of the catch block.

The `try` body is a statement or set of statements that may throw an exception. Although not every execution of the try statement must throw an exception, the try statement must contain at least one expression that is declared as throwing each of the types of exceptions listed in its catch clauses.

Each `catch` clause has a single parameter (type and name) followed by a block. A catch clause matches the thrown object exactly when the thrown object can be named by a name of the catch clause's parameter type. Only the first matching catch clause is executed.

The `try` statement is executed as follows:

- The `try` block is executed in order until something is thrown or the end of the `try` body is reached.
- If nothing is thrown during the `try` body, execution continues after the final `catch` clause of the `try/catch` statement.)
- If something is thrown during the `try` body, it is compared against the parameter of each catch block, in turn, until a match is found. In this case, that catch block is executed (as a normal block) with the parameter bound to the (matching) caught object. At most one catch block of a try statement is executed.

A try statement may also have a single optional finally clause. This is the keyword finally followed by a block. If the try statement is entered, the finally clause is always executed. This leads to somewhat complicated execution rules, described below and further documented in Sun's Java Language Specification. Finally clauses are largely outside the scope of this book and are included here only for completeness.

The following two points explain the special behavior of try statements with finally blocks

- After execution of at most one matching catch block, execution proceeds at the finally block (if it is present). If a try statement is entered, its finally block is always executed, regardless of the execution within the try statement.
- If no uncaught exceptions remain on exiting the finally block, execution proceeds after the end of the try/catch/finally statement. If there is an outstanding thrown object, execution proceeds with the continued flight of that throwable.

Throw vs. Return

There are both similarities and differences between `throw` and `return` statement types. Both involve a single `Thread` following instructions that may take it from one method or constructor to another, often moving across multiple objects. From the perspective of the Thread, the objects (and their methods and constructors) are providing roles that it plays, scripts that it reads, or instructions that it follows.

When a Thread is executing some instructions and reaches a method invocation expression (or an instance creation expression), it carefully records its current place in the script, puts the current script down on the table in front of it, and picks up the invoked method script. If fulfilling that expression in turn involves a further invocation, yet another script will be added to the pile on the table. When an invocation completes, the Thread puts the corresponding script away and returns to the carefully marked pending method invocation (or instance creation) expression on top of the pile.

In other words, to clear off the pile, the Thread must pick up each script in order on its way out and complete any remaining instructions before going on to the next. Every method invocation or instance creation expression eventually returns control to the body of code from which the call was invoked. The Thread eventually returns to the carefully marked spot and continues from there.

A major difference between return and throw statements is in how this execution proceeds, i.e., whether the Thread continues executing one instruction at a time or simply flies over the instructions looking for a matching `catch` statement. When a Thread returns normally from a method, execution continues one instruction at a time. When a Thread encounters a throw statement, it steps back through its pile of carefully marked scripts rather rapidly, scanning down the instructions until an appropriate `catch` statement is encountered. If the current script doesn't contain a matching `catch` statement, it is summarily discarded and the next script is examined in turn.

This means that a `return` statement always causes the current method to complete, returning control to whomever called this method. This is true no matter how many statement blocks the return is buried inside. ***A `return` always exits exactly one method invocation.***

In contrast, a `throw` exits one *block* at a time until a `catch` of the appropriate type is found. This means that a `throw` may not exit any methods (if the `throw` occurs directly inside an appropriate `try/catch`), or the `throw` may exit many methods (if the exception is not caught in any of these calling methods). ***A `throw` exits blocks until an appropriate `catch` is encountered.***

Exceptions, Errors, and RuntimeExceptions

In Java, any instance whose class extends the class Throwable can be thrown and caught. Two special subclasses of Throwable are defined for use under exceptional circumstances.

`Error` is the Java class that denotes a catastrophic failure. This is an event from which your program is not expected to be able to recover. A well-designed robust program that is expected to have an extended lifetime (such as a banking system or an airline reservation system) must have ways of dealing with catastrophic failure, but most programs that you write will not have to worry about such circumstances.

`Exception` is the Java class that indicates a non-catastrophic failure. Exceptions are cirucmstances from which your program should recover (or at least exit gracefully).

All Java built-in error and exception classes have two constructors, one that takes no arguments and one that takes a single String argument. This String can be accessed using the getMessage() method of Error or Exception. If you define your own subclass, it is a good idea to define these two constructors there as well.

`RuntimeException` is a special subclass of Exception. RuntimeExceptions are circumstances from which your program should recover, but -- unlike for other Exceptions -- methods and constructors throwing RuntimeExceptions do not have to declare this fact.

All Exceptions other than RuntimeExceptions are called checked exceptions. A method or constructor that may throw a checked exception must declare this fact, allowing the compiler to check for the presence of exception handlers. This can be very helpful in debugging, so you will generally want to extend Exception rather than RuntimeException.

When overriding a superclass method, a subclass method may only throw those checked exceptions also declared by the (overridden) superclass method. In other words, an overriding method may throw fewer things than promised by its superclass, but it may not throw additional (checked) things.

Designing good test cases

One of the most important parts of being a good programmer is knowing how to test your code. To begin this phase, write down all of the assumptions that your code makes. Think of something that violates one of these assumptions; will this break your code? How about something that violates three of these assumptions?

Once you have all of your assumptions written down, think about things that are extreme but within your assumptions. Try to design test cases for these. Think of every feature your code has, and every situation in which this feature could possibly be exercised. Design test cases for these as well. And don't forget the simple cases; it is always worth testing these as well as the pathological ones.

Your goal should be to thoroughly and exhaustively test your code, so you should design your test suite to exercise your program as fully as possible. You should also design test cases to catch bugs you think that other people might make. In particular, you should try to identify any weaknesses or difficult cases and design examples that stress these elements.

Finally, you should keep this test suite around, so that as you modify your code, you can test it again on these same examples, making sure that it still handles all of the old cases.

Chapter Summary

- In designing a program, you should anticipate things that can go wrong and design in mechanisms to deal with them.
  - Catastrophic failures cannot be prevented, but certain systems need to design in mechanisms to minimize the damage that they cause.
  - Some failures can be anticipated and avoided through simple checks and guards.
  - Other failures must be handled as they arise, often using Java's exception handling mechanims.
- Exceptions should record information that is useful for addressing the problem as well as information that is useful for advising the debugger or the human user.
- When an exception is thrown by a method or constructor, it exits each enclosing block in turn until a matching catch statement is encountered.
- Methods and constructors that may throw checked exception types must declare this fact in their signatures.
- A method invocation or constructor that may throw a checked exception may be safely invoked within a try block with a corresponding catch statement. The catch statement is responsible for attempting to recover from the exception.

Exercises

1. Describe the process of baking a cake. Include at least three exceptional circumstances that might arise and how these should be handled.

2. Describe the normal conduct of a soccer game. Include at least three exceptional circumstances that might arise and how these should be handled.

3. Define an Exception type called UnbelievableException. Remember to define two constructors.

4. Using your UnbelievableException type, write an animate object that continually asks the user for the user's age, then throws an UnbelievableException if appropriate. Note: the presence of an unbelievable age should *not* cause the program to terminate.