Exceptions

Algorithm Design

- Tricky because the details are crucial
- The general case may be straightforward
- 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
  - Adding the necessary details to the algorithm may render it overly complex
    - Difficult to get correct
    - Harder to debug and extend
    - Difficult for others to read and understand

Keep the Algorithm Focused

- Ideally a developer would write the algorithm in its general form
  - Include any *common* special cases.
- Exceptional situations that should should appear elsewhere
  - The strategy to handle these exceptions appears elsewhere
  - Could be 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

Exception Handling in Java

- 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
  - encourages the development of code that
    - cleaner
    - easier to maintain
    - easier to debug

Exception Objects

- 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; for example:
  - 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

Exception Situation—or Not?

- Many potential problems can be handled by the algorithm itself
  - An `if` statement can test to see if an array subscript is within the bounds of the array
  - 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 may be less straightforward to address directly in the algorithm.

Standardized Error Handling

- 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 (the C way) is for methods to return an integer that represents that method's success

Standardized Error Handling

- The ad hoc approach has problems—the error handling facilities developed by one programmer may be incompatible with those used by another.
- A comprehensive, uniform exception handling mechanism supported by the language is required
- Java's exceptions provide such a framework
- Exceptions are used in the standard Java API
- Programmers can create new exceptions that address issues specific to their particular problems

Java exceptions all use a common mechanism and are completely compatible with each other.

Exception Objects

- In Java, runtime errors are encapsulated in *exception objects*
- An exception object is an instance of a subclass of `Exception`
- An exception object stores important information about the runtime error:
  - **What**?
  - **Where**?

Exception Information

An exception object stores important information about the runtime error:

- **What**? An exception object has a name indicating what type of error has been detected

  Examples include:
  - `ArrayIndexOutOfBoundsException`: array subscript negative or too large
  - `ArithmeticException`: division by zero
- **Where**? All exception objects have a method that prints the chain of method calls that led up to the error

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at ArrayAccessException.filter(ArrayAccessException.java:3)
        at ArrayAccessException.compute(ArrayAccessException.java:7)
        at ArrayAccessException.main(ArrayAccessException.java:12)
```

What is displayed is called a *stack trace*.

`try/catch/finally` Block

```
try {
        Code that might throw an exception
} catch ( exception type1 var ) {
        Code that handles exceptions of type type1
         using exception object var
} catch ( exception type2 var ) {
        Code that handles exceptions of type type2
         using exception object var
              .
              .
              .
} catch ( exception typen var ) {
        Code that handles exceptions of type typen
         using exception object var
} finally {
        Code that is to be executed regardless
        of whether or not an exception is thrown
}
```

`catch` vs. `finally`

- The series of `catch` clauses are optional if a `finally` block is present
- The `finally` block is optional if at least one `catch` clause is used
- It is illegal to omit both the `catch` clauses and the `finally` block; that is, a `try` block may not stand alone
- Only one `finally` block may be used with each accompanying `try` block
- It is illegal to use a `catch` clause outside the context of a `try` block

Exception Specification

- A method that can throw an exception must say that it can
- The exception specification lists the kinds of exceptions that the method can throw
- The exception specification is placed between the parameter list and method body

```
public class Collection {
    . . .
    public void insert(Object newObj) throws CollectionFullException {
        . . .
    }
    . . .
}
```

The Client's Responsibility

- Since the method specifies what exceptions it can throw, the compiler can check to see if client code makes provision for the exception(s)
- If is a compile-time error if client code does not handle the potential exception

```
col = new Collection(10);
try {
    col.insert("Wow");
} catch ( CollectionFullException ex ) {
    System.out.println("Collection full; nothing inserted");
}
```

Handling an Exception

To handle an exception, 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)

```
public static void doInsert(Collection col, Object obj)
                throws CollectionFullException {
    col.insert("Wow");
}
```

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

Specifying Multiple Exceptions

- 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

`throw`

- 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" means the method that is closest in the chain of method calls

```
public void insert(Object newObj) throws CollectionFullException {
    if ( currentSize > MAX_SIZE ) {
        throw new CollectionFullException();
    }
    . . .
}
```

Checked and Unchecked Exceptions

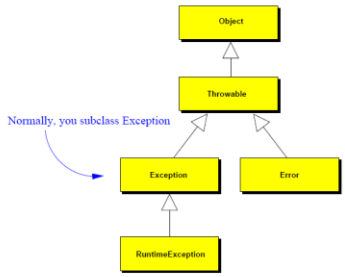
Most programmer-defined exception types are *checked* exceptions

- This means client code is required to check for and handle the exception
- The compiler enforces the check

Some exception types (subclasses of `RuntimeException`) are *unchecked*

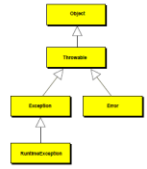
- This means client code is **not** required to check for and handle the exception
- The compiler will not issue an error if code that can throw a `RuntimeException` subclass does not handle the exception

Exception Class Hierarchy



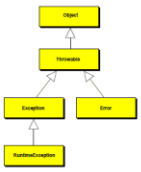
Exception Class Hierarchy

- `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



Exception Class Hierarchy (cont.)

- `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
  - Methods that can throw subclasses of `RuntimeException` are not required to declare so and client
  - 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



Checked Exceptions Preferred

- Checked exceptions lead 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 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

`catch`ing an Exception

- Each `catch` clause specifies a type and a variable
  - The variable is similar to a formal parameter in a method definition
  - Code within the `try` block throws an exception object, and this object is assigned to the variable specified at the beginning of the `catch` clause
- Two useful pieces of information can be extracted from the exception object variable:
  - **What**? The `toString()` method returns a printable version of the type of the exception
  - **Where**? The `printStackTrace()` method prints a trace of the method execution call chain; it also provides the same information as `toString()`
- Sometimes there will be only one `catch` clause since only one particular type of exception is expected

Rethrowing an Exception

- An exception handler within a `catch` clause may take a few steps to handle the exception and then *rethrow* or throw a completely different exception
- This allows the handler to address the exception with any local information that it may have and then pass the exception off to its caller for more caller-specific handling
- The caller may only wish to handle a different set of exceptions, so the handler may need to repackage the given exception into a new type of exception expected by the caller

Catching Order is Important

- When an exception is thrown, the `catch` clauses within the nearest handler are checked in order
- The first `catch` clause that matches the type of the exception thrown is the one executed
- When inheritance is involved, the situation becomes more interesting:
  - The comparison performed in each `catch` clause is a test for *assignment compatibility*
  - Recall that a reference to an instance of a subtype can be assigned to a variable declared to be a reference to a supertype
  - The *is a* relationship between a subclass and its superclass makes this upcasting operation always legal
- Since the `catch` clauses are checked in the order they appear in the source code, more specific exception types (subclasses) must appear before their more general types (superclasses)

The "Catch-all" Exception

- Since all practical exceptions are derived from the class `Exception`, the type `Exception` can be listed as the last `catch` clause to serve as a "catch-all" handler
  - If code within the `try` block throws an exception, all the `catch` clause types will be checked in order
  - If none up to the `Exception` clause match, then the code in the `Exception` will be executed since all exceptions subclass `Exception`
- Notice that the `Exception` type could be used in a single `catch` clause to catch all exceptions without the need to check for individual exception subtypes
  - This practice should be avoided, however, since multiple `catch` types allow different actions to be taken to deal with the different types of problems that may arise
  - Such a comprehensive exception handler would not be as useful for diagnosing a wayward program

Using Exceptions

Exceptions should be reserved for uncommon errors

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

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