

## Errors and exception handling

Many programs have more code to handle error condition than to solve the problem for which the program was written. There are many different classes of errors that can occur:

1. User input errors (e.g., mistyped input, wrong filename given, wrong mouse button pressed, etc.)
2. Device errors (e.g., network disconnect, disk crash, modem not turned on, etc.)
3. System resource limitations (e.g., disk is full, heap memory exhausted, file does not exist)
4. Software and hardware component failures (e.g., DNS not available, invalid input, etc.)

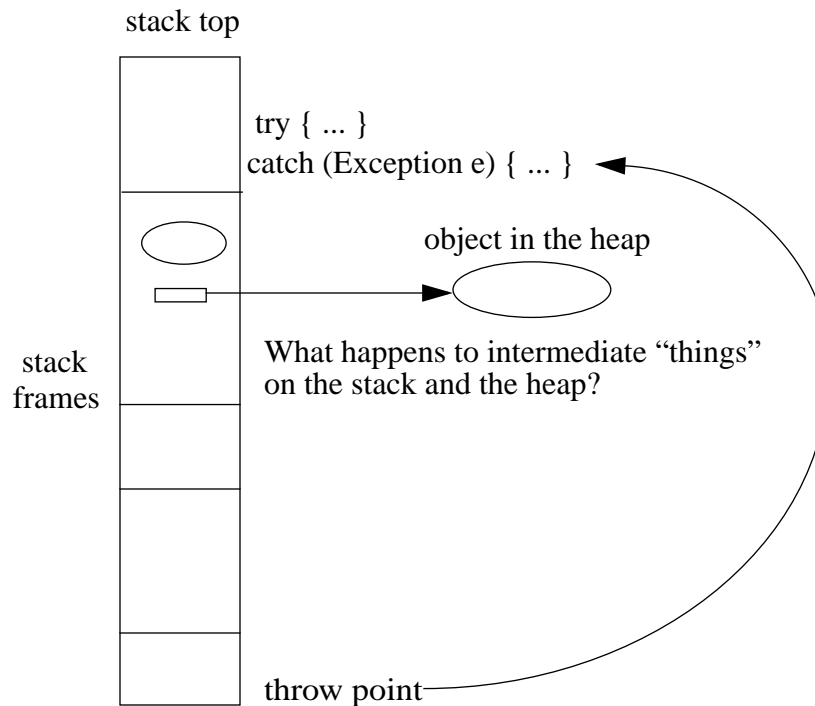
The `java.net.Socket` class is a good example. The class implements an object-oriented wrapper onto Unix sockets for networking (using Java native functions). However, you don't need to know all the low-level details of sockets when implementing a network client application, but you do need to be made aware of error conditions that may arise within the class library. These include I/O errors and network errors, such as unknown host names, broken network connections, etc.

A class library can sometimes handle errors internally, and do something sensible. Other times, the user of the class must be notified of the error. Exceptions provide a *structured* way to communicate error information across a class or procedure abstraction boundary. Many programs are written that do not perform much error checking. Programmers often assume that a program is always given the correct input, there will be no device errors, system resources are always available, and component failures either can't happen because "I wrote the code," or if they do occur, just call "abort." Imagine the flight control computer on a Boeing 777 having been developed in this manner, which just calls "abort" when something goes wrong---that will not be a very satisfying experience!

Implementors of OO class libraries cannot predict how someone might use (or misuse!) your classes. We have to program defensively, which is especially true when developing class libraries that other, perhaps more naive, programmers will use to develop their programs.

## Exceptions and the run-time stack

When an exception is thrown, the stack is “unwound” and intermediate stack frames are deallocated as part of throwing the exception. Keep in mind that any object references in nested scopes between where the exception is thrown and where it is caught are lost, and the objects they pointed to may become garbage (if no other scopes hold references to them). In Java, this is not a problem, because the garbage collector will keep track and collect them eventually. However, in C++, when you throw an exception, you have to be sure to explicitly deallocate any heap allocated objects that may be referenced in intervening stack frames, otherwise you end up with a “memory leak”. The point of throwing an exception is usually that it is caught and some corrective action can be taken or a clean “shutdown” procedure can occur---something more graceful than abort.



## Throwing an exception on the occurrence of an error

If an **exception handling** facility exists, we can **throw** an exception. The way this is done in Java (and ANSI C++) is using a combination of a lexical **try block** and an one or more lexical **catch blocks** that act as exception handlers. An exception is raised by using the throw statement. e.g.:

```
public void some_func() throws SomethingBadHappenedException
{
    int result_code = do_something_that_might_fail();

    if (result_code == OK) {
        ... // hurray! it worked!
    }
    else
        throw new SomethingBadHappenedException("Error: failed on ...",
result_code);
}
```

The throw statement causes a transfer of control **up the call chain** to the “nearest” catch block for the type of object thrown. If no catch block is defined to catch the specific exception thrown, or an exception base class from which the exception type inherits from, then the program aborts.

## Exceptions should be exceptional

Consider the following:

1. Should an exception be raised when you iterate beyond the end of an array or vector?
2. Should an exception be raised when you reach the end of a file?
3. Should an exception be raised when you pop an empty stack?

In general, use exceptions to cope with unpredictable or exceptional events, and not as a way to cause control to jump around in a program to various handler routines.

Java does not have a goto statement, but you could abuse the exception handling mechanism and implement a non-local goto mechanism, which is a very BAD idea. This style of event processing is better done using other mechanisms, such as “call backs”, which are commonly found in window managers and network servers.

**Q: What about retrying code that raised an exception that was caught? When is this okay?**

```
boolean success = false;
while (!success)
{
    try { ...; success = true; }
    catch (SomeException e) { ... /* try to correct the problem */ }
    // fall out and try again
}
```

## Exception Inheritance Hierarchy

Typically, it is a good idea to implement an *exception class hierarchy* of error types and throw an object that represents all the information known about the error at the time it occurred. Then, a catch block higher up in the call chain can interface with the exception object to find out more about what went wrong, instead of just passing an opaque error value (i.e., -1), which by itself is not very informative. For example, provide as much info as possible:

Where `SomethingBadHappenedException` is but one of many subclasses of the **java.lang.Exception** class. Subclasses of the `Exception` class typically just implement constructors that allow an exception object to be constructed, passing a string representation of its error message up to the base class constructor.

```
class SomethingBadHappenedException extends Exception {  
  
    SomethingBadHappenedException() { super(); }  
    SomethingBadHappenedException(String s) { super(s); }  
    SomethingBadHappenedException(String s, int err) {  
        super(s + String.valueOf(err));  
    }  
}
```

Since all basic exception types inherit from the class `Exception`, you can always write a “catch all” block as:

```
try { ... }  
catch (Exception e) { ... }
```

## The `java.lang.Exception` and `java.lang.Throwable` classes

```
public class Exception extends Throwable {
    public Exception() { super(); }
    public Exception(String s) { super(s); }
}

public class Throwable implements java.io.Serializable {
    private String detailMessage;

    public native Throwable fillInStackTrace();
    private native void printStackTrace0(Object s);

    public Throwable() { fillInStackTrace(); }
    public Throwable(String msg) { fillInStackTrace(); detailMessage = msg; }

    public String getMessage() { return detailMessage; }
    public String toString() {
        String s = getClass().getName();
        String message = getMessage();
        return (message != null) ? (s + ": " + message) : s;
    }
    public void printStackTrace() {
        System.err.println(this);
        printStackTrace0(System.err);
    }
}
```

## java.lang.Error class

Unlike subtypes of `Exception`, which are expected to be caught and handled by the application program, Java also supports subtypes of `Throwable` that inherit from the `java.lang.Error` class. This class of exceptions are considered serious run-time errors that are not caught, but which cause the program to terminate.

```
public class Error extends Throwable {
    public Error() { super(); }
    public Error(String s) { super(s); }
}
```

### Examples of serious errors:

```
public class VirtualMachineError extends Error { ... }
public class InternalError extends VirtualMachineError { ... }
public class OutOfMemoryError extends VirtualMachineError { ... }
public class StackOverflowError extends VirtualMachineError { ... }
public class UnknownError extends VirtualMachineError { ... }
```

### Errors that are more likely to be encountered while developing a program:

```
public class LinkageError extends Error { ... }
public class NoClassDefFoundError extends LinkageError { ... }
public class ClassFormatError extends LinkageError { ... }
public class IncompatibleClassChangeError extends LinkageError { ... }
```

## Simple Networking: InetAddress and URL objects

The `java.net.InetAddress` class is used to create and manipulate Internet address objects. The reason why a hostname is associated with the `InetAddress` object is because the system has to use DNS (Domain Name System) to dynamically look up host information. The steps are to first get the IP address of the local machine, and then use the address to get the hostname

```
import java.net.*;
...
InetAddress addr = InetAddress.getLocalHost();
try { String host = addr.getHostName(); }
catch (UnknownHostException e) { System.err.println(e); }
```

If the hostname is not found, then an “`UnknownHostException`” is thrown. You can ask any `InetAddress` object to convert itself to a `String` object, which is a simple formatted string method that results in a string formatted as “hostname/%d.%d.%d.%d”

```
public String toString() { return getHostName() + "/" + getHostAddress(); }
String address = addr.toString();
```

URLs are also useful objects used to identify resources in the network, using the `java.net.URL` class:

```
try {
    URL http = new URL("http://www.cs.utexas.edu:80/index.html");
    URL ftp = new URL("ftp", "prep.ai.mit.edu", 21, "/pub/gnu/gcl");
} catch (MalformedURLException e) { ... }
```

**Note:** `MalformedURLException` inherits from `IOException`.

## Simple Networking: Socket creation and use

To connect to a host on the Internet, create an `InetAddress` of a remote host using its name, and then use it to create a client **socket** for communicating with some application server running on a remote host.

```
import java.net.*;

try {
    InetAddress host = InetAddress.getByName("cs.utexas.edu");
    int port = 25; // the well-known TCP port # for the SMTP service
    Socket sock = new Socket(host, port);
}
catch (UnknownHostException e) { ... }
catch (IOException f) { ... }
```

If you want to write a server application, then you use a **ServerSocket** to listen for incoming client connections:

```
try {
    int port = 25; // SMTP server
    ServerSocket listener = new ServerSocket(port);
    // listen for client connections
    listener.accept(); // blocks thread until client connects
}
catch (IOException e) { ... }
```

## Simple Networking: reading and writing a socket

Sockets provide a duplex byte stream abstraction that can be read and written. So, every socket has two byte streams associated with it: a `java.io.DataInputStream` and a `java.io.DataOutputStream`. However, for reading and writing, we want to “wrap” the raw byte streams using buffered reader and write streams. In order to do so, we have to create a “bridging” I/O object called a `StreamReader`.

```
import java.net.*;
import java.io.*;
...
try {
    InputStreamReader in = new InputStreamReader(socket.getInputStream());
    BufferedReader rdr = new BufferedReader(in);
    String line;
    while ((line = rdr.readLine()) != null) { process(line); }
    rdr.close();
}
catch (IOException e) { ... }

try {
    OutputStreamWriter out = new OutputStreamWriter(socket.getOutputStream());
    BufferedWriter wrt = new BufferedWriter(out);
    wrt.write("some string");
    wrt.flush();
}
catch (IOException e) { ... }
```

## Example: Threaded Network ReaderWriter Class with Exceptions

It is often useful to write a network client that contains two threads: a Reader thread and a Writer thread. The Reader thread listens for data from the server and sends it to the console or a file. The Writer thread takes input from the user or a file, and sends it to the server.

```
public class Client {
    Socket socket;
    private Thread reader, writer;

    public Client (String host, int port) {
        try {
            socket = new Socket(host, port);
            reader = new Reader(this);
            writer = new Writer(this);
            reader.setPriority(6); // reader has priority over the write
            writer.setPriority(5);
            reader.start(); // tart calls reader.run()
            writer.start();
        } catch (Exception e) { System.err.println(e); }
    }

    public static void main(String[] args) {
        try { new Client(args[0], Integer.parseInt(args[1])); }
        catch (NumberFormatException e) { System.exit(-1); }
    }
}
```

## The Reader Thread

```
class Reader extends Thread {
    private Client client;

    public Reader(Client c) { super("Client Reader");this.client = c; }

    public void run() {
        BufferedReader in = null;
        String line;
        try {
            in = new BufferedReader(new InputStreamReader(client.socket.get-
InputStream()));
            while (true) {
                line = in.readLine();
                if (line == null) {
                    System.out.println("server closed connection");
                    break;
                }
                System.out.println(line);
            }
        } catch (IOException e) { System.err.println("Reader Failed: " + e); }
        finally try {
            if (in != null) in.close();
        } catch (IOException e) { System.err.println(e); System.exit(0); }
    }
}
```

## The Writer Thread

```
class Writer extends Thread {
    private Client client;

    public Writer(Client c) { super("Client Writer"); this.client = c; }

    public void run() {
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            String line;
            in = new BufferedReader(new InputStreamReader(System.in));
            out = new PrintWriter(client.socket.getOutputStream());
            while (true) {
                line = in.readLine();
                if (line == null) {
                    break;
                }
                out.println(line);
            }
        } catch (IOException e) { System.err.println("Writer: " + e); }
        finally try { if (out != null) out.close(); }
        catch (IOException e) { System.err.println(e); System.exit(0); }
    }
}
```

## URL Processing

For accessing resources on the web using URLs, you can avoid the details of sockets, and just operate on URL objects:

```
import java.net.*;
import java.io.*;

try {
    URL url = new URL("http://www.cs.utexas.edu:80/index.html");
    String protocol = url.getProtocol(); // "http"
    String host = url.getHost(); // "www.cs.utexas.edu"
    int port = url.getPort(); // 80
    String file = url.getFile(); // "index.html"

    in = new InputStreamReader(url.openStream());
    BufferedReader rdr = new BufferedReader(in);
    String line;
    while ((line = rdr.readLine()) != null) { process(line); }
    rdr.close();
}
catch (MalformedURLException e) { ... }
catch (IOException e) { ... }
```