Error Handling

- The general structure for handling an error that has occurred:

```java
try {
    throw exception();
}
catch (exception e){
    //Fix the error!
}
```
Error Handling

```
try {
    throw exception();
}
catch (exception e) {
    //Fix the error!
}
```

“try” this code and see if any errors happen
Error Handling

try
{
    throw exception();
}
catch (exception e)
{
    //Fix the error!
}
Error Handling

```java
try {
    throw exception();
}
catch (exception e) {
    //Fix the error!
}

If they do, skip the rest of this code and “catch” the error here...
```
Error Handling

```java
try {
    throw exception();
} catch (exception e) {
    //Fix the error!
}
```

... if the error is of this type. (Or one of its subtypes.)
Error Types

• There are actually multiple sorts of errors which can occur within a program and its code.
  – *Compile-time* errors: the interpreter / compiler can’t make sense of your code.
  – *Logical* errors: the program doesn’t crash, but it behaves differently than intended.
Error Types

• There are actually multiple sorts of errors which can occur within a program and its code.
  – *Run-time* errors: Errors which crash a program, but were not intentionally generated by programmer code.
  – Generally, user-generated issues caused by bad inputs. (GIGO, PEBKAC)
  – When a calculator program is told to divide by zero, if it doesn’t check for illegalness, a runtime error will occur.
Error Types

• There are actually multiple sorts of errors which can occur within a program and its code.
  
  – *Generated* errors: the program detects that it is malfunctioning and generates an error to signal it.
  
  – Often generated to prevent run-time errors from crashing the program. Making these gives a chance for recovery if they are caught elsewhere.
Error Types

• When a program hangs (goes unresponsive), it’s typically a logical error.
These license terms are an agreement between Sysinternals (collectively "Sysinternals") and the end user of the software ("you").

1. INSTALLED
   a. You are granted a license to install and use the software on a single computer.
   b. You may not distribute the software to others.
   c. You may make a single backup copy of the software.

2. SCOPE
   a. The software is for your personal, non-commercial use only.
   b. You may not use the software for any commercial purposes.
   c. You may not reverse engineer, decompile, or disassemble the software.

You may not:

* work around any technical limitations in the binary version of the software
* reverse engineer, decompile or disassemble the binary version of the software

* applicable law expressly permits, despite this limitation;

If you close the program, you might lose information.

- Close the program
- Wait for the program to respond
Recovery

• Unfortunately, like in the prior example, not all errors can be detected.
  – Sometimes, an application can get stuck in an infinite loop, rendering it completely unresponsive.
  – Multithreaded applications can also become stuck due to “deadlock” and “livelock” situations.
On the Use of Exceptions

- Exceptions are extremely handy to have as a tool for an object to indicate bad inputs to its constructor or method.

- However, exceptions are quite “expensive”, computationally, to throw.
  - Remember, they interrupt *everything*. 
On the Use of Exceptions

• Objects should throw exceptions when:
  – A constructor receives (bad) inputs that would result in an invalid object.
  – A method receives bad input
  – Out of range index or value
  – Null pointer
On the Use of Exceptions

• Objects should throw exceptions when:
  – A method cannot perform the requested action.
  – Some objects may have different “modes,” where certain actions may only be possible in certain situations.
  – Example: a file must be opened to read or write from/to it.
On the Use of Exceptions

- Objects should throw their own exceptions, rather than relying on a future method to throw exceptions.
  - When debugging, it is better to know the underlying source of the erroneous error.
  - Thus, the sooner code can detect that an error will occur (even if later in the chain), the better.
On the Use of Exceptions

• Objects should throw their own exceptions, rather than relying on a future method to throw exceptions.
  – Failure to do so will make it seem as if the object is miscoded, using the future method incorrectly.
On the Use of Exceptions

• Exceptions are a useful tool for object-orientation, allowing objects to actively prevent actions that would be invalid.
On the Use of Exceptions

- Exceptions are a useful tool for object-orientation, allowing objects to actively prevent actions that would be invalid.
  - They also allow objects to report why those actions are invalid, which aids debugging.
  - Sometimes, it may even be possible to recover from the error, depending on its type.