==========
Exceptions and Events

What ***is*** an exception?
- an unusual condition detected at run time
  Examples:
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions (not necessarily errors)
- error v. nonlocal return -- different mechanisms? (Common Lisp)

What is an exception handler?
- code executed when exception occurs
- may need a different handler for each type of exception

Why design in exception handling facilities?
- allow user to explicitly handle errors in a uniform manner
- allow user to handle errors without having to check these conditions explicitly in the program everywhere they might occur

Downsides
- may discourage careful checking of boundary conditions (laziness)
- introduces brittleness: caller has to be prepared to handle error
- Some companies (e.g., Google) have banned the use of exceptions in their code.

Consider handling of errors in a (large) recursive-descent compiler.
It's something of pain in languages without exceptions: need extra parameters and checks in every procedure. May be nicer to be able to back out to exactly where you want to.

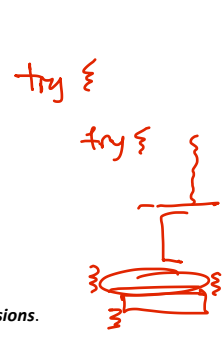
Pioneers:
PL/I: dynamically scoped
CLU: statically scoped, procedure/abstraction oriented
(can't handle locally)

Convergence in modern languages on built-in, statically scoped, "replacement" model: Ada, C++/Java/C#, Modula-3, ML, Common Lisp, python/php/ruby (poorer substitutes in perl, tcl)

Discussion here is for C++/Java.

Handlers local to code in which exception is raised

```
try {
   ...
   // throw obj;
   ...
} catch (end_of_file) {
   ...
} catch (io_error e) {
   ...
} catch (...) {
   ...                    // catch-all
}
```



Handlers must be at the end of a block of code
(but can put blocks around any statement).
ML and Common Lisp allow handlers on arbitrary ***experssions***.

Notion of ***matching*** an exception.
Arguments as members of object.
(Ada has no arguments; Modula-3 and ML make them look like parameters.)

Static (nested) binding within subroutine, then propagate up dynamic chain.
All functions that the exception propagated out of are terminated.
C++ executes destructors as appropriate on the way out.
Execution continues after handler code (which is always at the end of a block)

replacement

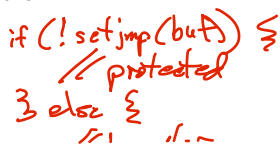
Interesting question: would it make sense to have the handler replace the ***entire*** protected block, rather than the suffix? A group at MSR proposed this a few years back, and called it "`try-all`" (I don't like the name).
It would require a ***roll-back*** mechanism, at least in general.
That has a natural connection to transactional memory.

Implementation of statically-scoped exceptions

(1) Push handler address when entering a protected block, pop when leave.
Sometimes implemented this way in C++, probably to minimize size and complexity of the run-time library.

(2) Do everything via lookup in tables produced by the compiler.
This is the "right" way to do it.

(3) Can sort of fake it in C with `setjmp()` and `longjmp()`. These take state snapshot and restore on throw. Doesn't work right for non-volatile local variables. Very expensive.



Events

Commonly used in interactive programs. In Java:

```
class PauseListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      // do whatever needs doing
   }
}
...
JButton pauseButton = new JButton("pause");
pauseButton.addActionListener(new PauseListener());
```

Or, with anonymous inner classes:

```
pauseButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      // do whatever needs doing
   }
});
```

This is a little ugly because Java insists on making everything a class.
What you really want is a lambda expression to be executed when the even occurs.

In C# you have ***object closures***, which are roughly lambdas, and which the language calls ***delegates***.

```
void Paused(object sender, EventArgs a) {
   // do whatever needs doing when the pause button is pushed
}
...
Button pauseButton = new Button("pause");
pauseButton.Clicked += new EventHandler(Paused);
```

Paused matches the pattern established by a delegate declaration in the library. Can be a static method or a method of a particular object -- even a nested object. In the latter case C# (like C++) allows access to static members of the surrounding class only. Java allows access to nonstatic members, but doesn't have delegate sugar.

or

```
pauseButton.Clicked += delegate(object sender, EventArgs a) {
   // do whatever needs doing
}
```

Note the connection to Runnables in Java, and to subroutine closures in languages with nested subroutines.

Signal handlers

In Swing, as in many GUI packages, events are handled by separate threads, managed by the runtime. If they access data for which consistency is an issue, you have to use explicit synchronization.

In other languages, event handling can happen in the currently running thread, as if it were a spontaneous subroutine call. Because the caller doesn't actually make the call, we need extra machinery, called a ***trampoline***, to fake the calling the sequence: