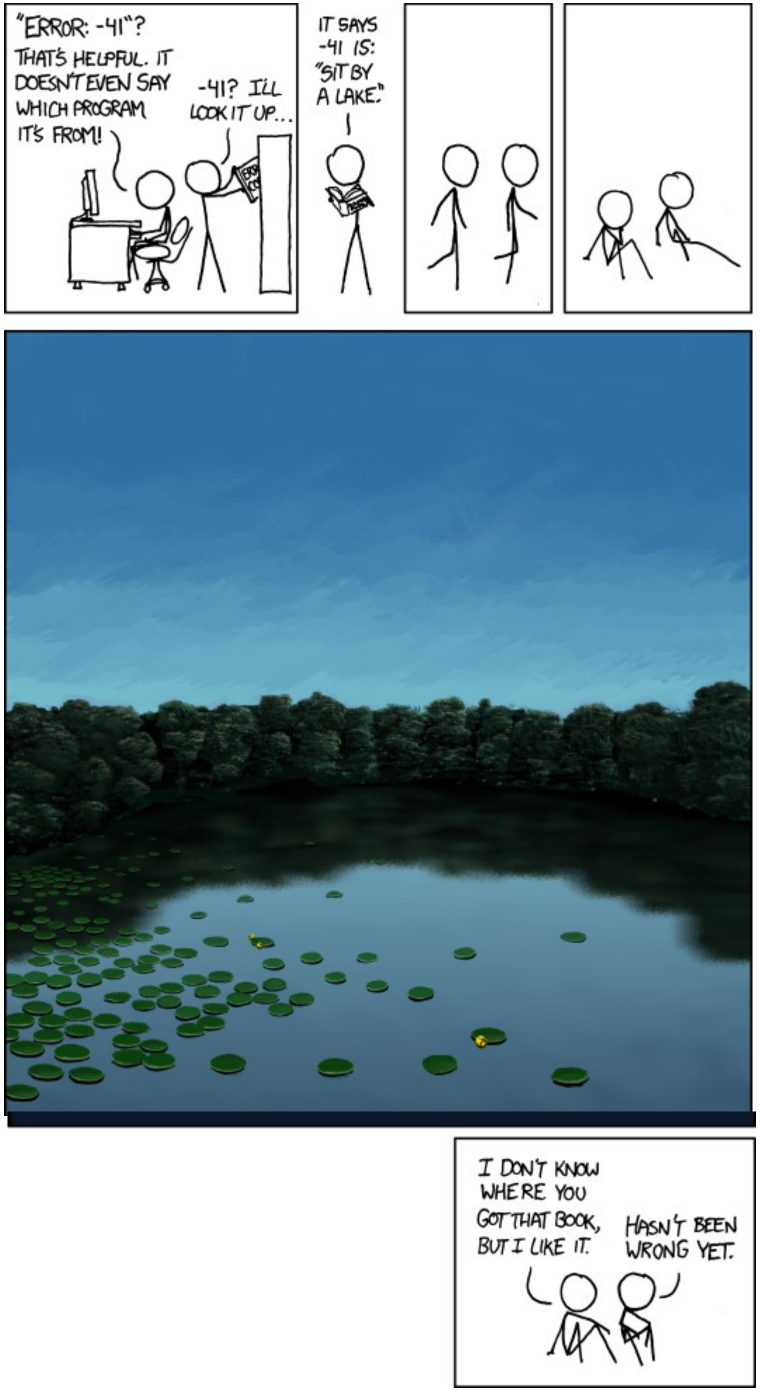
CS 430
Spring 2015

Mike Lam, Professor

Errors and Events



Approaches

- Do nothing
  - Worst possible approach!
  - No indication that anything has gone wrong
  - "Silent propagation" of errors
- Terminate the program
  - I.e. delegate error handling to the operating system
  - Also rather drastic, but at least it provides some kind of notification (OS-dependent)
  - No opportunity to correct problems
  - Most infamous: the segfault

Approaches

- Pass around error handlers
  - Extra function parameters (and associated runtime overhead)
  - Confusing and difficult to reason about
  - What if you pass the wrong error handler?
- Handle all errors at their source
  - Error handling often depends on current context
  - Lots of (possibly duplicate) error handling code

Approaches

- Return an error value (in same variable)
  - Error value must come from variable domain
  - Blurs the line between program logic and program data
  - Burden shifts to callers, who must test for error value
- Return an error value (in separate variable)
  - Cleaner (separation between logic and data)
  - Burden is still on the caller to remember to test for errors

Exception Handling

- *Exception*: unusual event (possibly erroneous) that requires special handling
- *Exception handler*: code unit that processes the special handling for an exception
- An exception is *raised* when the unusual event is detected, and is *caught* when the exception handler is triggered
- This framework is called *formal* exception handling
  - First introduced in PL/I (1976)

Benefits of Formal Exceptions

- Less program complexity and clutter; increased readability
- Standardized handling mechanisms
- Increased programmer awareness
- Decouples exception handling from program logic
- Handler re-use via exception propagation
- More secure due to compiler analysis

Design Issues

- How and where are exception handlers specified?
- What is the scope of exception handlers?
  - What information (if any) is available about the error?
- Are there any built-in exceptions? If so, what are they?
- Can programmers define new exceptions?
- How is an exception bound to a handler at runtime?
- Where does execution resume (if at all) after an exception handler finishes?

Syntax

- Detection
  - C++/Java: uses "throw"
  - Ruby: uses "raise"
  - Should be located as close as possible to the root cause
- Handling
  - C++/Java: uses "try/catch" (and "finally" in Java)
    - Unhandled exceptions must be declared at the method level in Java (e.g., "throws IOException")
  - Ruby: uses "begin/rescue/else"
  - Usually located at the end of a code unit

Exceptions

- C++: Any variable can be thrown
  - Each handler catches a particular variable type
  - No predefined exceptions
  - Exception handling with references is complicated
- Java: Any Throwable object can be thrown
  - Each handler catches a particular class
  - Built-in exception hierarchy (Error and Exception+descendants)
    - Error class instances are considered to be "system-level" and "reasonable" applications should not worry about them
  - "Unchecked" (Error and RuntimeException + descendants) vs. "unchecked" exceptions
- Ruby: A string or any exception can be thrown
  - Strings are converted to RuntimeError instances
  - Each handler catches a particular class (RuntimeError by default)
  - Built-in exception hierarchy (Exception+descendants)

Binding and Continuation

- When an exception is thrown
  - Look for matching handler in local scope
    - Could be an "else" handler
  - If no handler is found, continue through ancestors (usually via dynamic scoping)
  - If no handler is found, abort the program
- When a handler finishes
  - If the handler threw another error, handle that
    - First execute any "finally" clause if present
  - Continue execution after the handler
    - First execute any "finally" clause if present
  - Changes made by the error handler are visible

Functional Languages

- "Pure" functional handling of errors is very difficult
  - Error handling usually involves side effects
- Haskell: usually handled with monads
  - Encapsulate errors and data inside a single structure
  - Syntax becomes rather complicated

```
data Exceptional e a =
     Success a
   | Exception e
   deriving (Show)
```

Examples

- See handout

Language Debate

- Are formal exceptions any different from GOTO statements? If so, are they just as dangerous? If not, how are they different?

Language Debate

- Are formal exceptions any different from GOTO statements? If so, are they just as dangerous? If not, how are they different?
  - Basic different: formal exceptions are more structured
    - More rules and restrictions governing their uses
  - Language facilities provide (mostly) safe usage
  - Care should be taken to limit their complexity
    - Main issue: proximity of detection and handling

Event Handling

- Similarity between error handling and event handling
  - Both indicate asynchronous events that must be handled by the program
- Primary difference: events are "normal" while errors are "unusual"
- Another difference: events are usually handled in a separate thread
  - Keeps the program feeling "responsive"

Event Loops

- Event loop: code that explicitly receives and handles events
- Traditional form:

```
while(GetMessage(&Msg) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

- Often run in its own thread
- Requires explicit dispatch routine
  - Can become extremely complex and unwieldy

Observer Pattern

- Cleaner solution: Observer pattern (OOP)
  - Single event thread, implemented in language runtime
    - Dispatches events to relevant objects
  - Objects maintain a list of "observers"/"listeners"
  - Upon receiving an event, the object passes it to a designated routine in every registered observer
  - Optional improvement: anonymous functions or event handling classes
    - Very similar to lambda functions or closures!

Example

```
import java.awt.event.*;
import javax.swing.*;

public class EventEx1 extends JFrame {

    private class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, "Clicked!");
        }
    }

    public EventEx1() {

        JButton myButton = new JButton("Click me!");
        myButton.addActionListener(new ButtonHandler());

        getContentPane().add(myButton);
        pack();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        (new EventEx1()).setVisible(true);
    }
}
```

Example

```
import java.awt.event.*;
import javax.swing.*;

public class EventEx2 extends JFrame {

    public EventEx2() {

        JButton myButton = new JButton("Click me!");
        myButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "Clicked!");
            }
        });

        getContentPane().add(myButton);
        pack();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        (new EventEx2()).setVisible(true);
    }
}
```