Evolving Error Models

```
Traceback (most recent call last):
  File "opening_credits.py", line 25, in <module>
    paper________("The Error Model [in Midori]")
  File "opening_credits.py", line 11, in paper________
    author_______("Joe Duffy")
  File "opening_credits.py", line 14, in author_______
    presented_by_("Marvin Humphrey")
  File "opening_credits.py", line 17, in presented_by_
    date_________("December 5, 2019")
  File "opening_credits.py", line 20, in date_________
    for__________("Papers We Love San Diego");
```

Recoverable Errors

- file i/o failures
- network i/o failures
- parser errors
- input validation

Handling Recoverable Errors

- return codes
- exceptions

Return Codes

```
int result = foo();
if (result != 0) {
    fprintf(stderr, "foo failed");
}
```

No language or runtime support necessary!

Exceptions

```
try {
    foo();
}
catch (FooException e) {
    System.err.println("foo failed")
}
```

Stack Traces

```
Traceback (most recent call last):
  File "opening_credits.py", line 25, in <module>
    paper________("The Error Model [in Midori]")
  File "opening_credits.py", line 11, in paper________
    author_______("Joe Duffy")
  File "opening_credits.py", line 14, in author_______
    presented_by_("Marvin Humphrey")
  File "opening_credits.py", line 17, in presented_by_
    date_________("December 5, 2019")
  File "opening_credits.py", line 20, in date_________
    for__________("Papers We Love San Diego");
```

Optional!

`throw`: AN ALTERNATIVE `return`

```
public static int not42(int number)
    throws IllegalArgumentException {
  if (number == 42) {
    throw new IllegalArgumentException();
  }
  return number;
}
```

- success: return `int` to caller.
- failure: return `IllegalArgumentException` to nearest `catch` block.

Checked Exceptions

(Java, Midori, Swift)

```
public static bool guessAnswer(int guess)
    throws IllegalArgumentException {
  guess = not42(guess); // <-- call may throw!
  return guess == theAnswer;
}
```

- method signature includes `throws` clause
- declared exceptions propagate

Unchecked Exceptions

(C#, C++, Python, Ruby, ...)

```
public static bool GuessAnswer(int guess) {
    guess = Not42(guess); // <-- call may throw!
    return guess == theAnswer;
}
```

- no `throws` clause
- no exception propagation
- *ignoring errors makes you faster!*

Invisible Control Flow

C# (and C++, Python, etc...)

```
int number = Not42(42); // kaboom?
```

Swift

```
let number = try not42(42)
```

Signal Handlers

```
void handle_sigint(int sig) {
    fprintf(stderr, "SIGINT detected");
}

int main() {
    signal(SIGINT, handle_sigint);
    while (1) { }
}
```

- global
- asynchronous

Not for general error handling!

Bugs

- invalid memory access
- unintended integer wraparound
- divide-by-zero
- assertion failures

Bugs Are Not Recoverable Errors!

recoverable error

all states accounted for

bug

unanticipated state

Modern Error Models

Project	Recoverable Errors	Bugs
Midori	exceptions	“abandonment”
Go	return codes	`panic`
Rust	return codes	`panic!`
Swift	exceptions	`fatalError`

Abandonment Consequences

Project	Victim
Midori	process (lightweight)
Go	Goroutine
Rust	thread
Swift	process (app)

Degrees of Safety

- Duffy: after bug, address space compromised
- Me: and maybe more!
  - multiple address spaces?
  - persistent storage?

C Error Model

```
int result = do_stuff();
if (result == -1) {
    fprintf(stderr, "foo failed!");
    exit(1);
}
```

- use return codes
- reserve "sentinel" values

C Error Model Flaws

```
int **nums = malloc(1);
*nums[1] = 42;
```

- easy to forget to check return codes
- rarely exercised error handling branches
- magic sentinels (e.g. `NULL`, `-1`) = bad APIs
- single channel for bugs and recoverable errors…
- … but don't forget signals
- unsoundness of C

Java: Exceptions

```
public static int not42(int number)
    throws IllegalArgumentException {
  if (number == 42) {
    throw new IllegalArgumentException();
  }
  return number;
}
```

- both checked *and* unchecked exceptions
- both bugs and recoverable errors are exceptions

Swift: Exceptions

```
enum MyError : Error {
  case ArgIs42
}
func not42(number: Int) throws -> Int {
    if number == 42 {
        throw MyError.ArgIs42
    }
    return number
}
```

- no "invisible control flow"
- `fatalError` for bugs, `throw` for recoverable errors
- func `throws` or not

Go: Return Codes

```
func Not42(num int) (int, error) {
    if num == 42 {
        return 42, errors.New("42 disallowed")
    }
    return num, nil
}
func main() {
    num, err := Not42(42)
    if err != nil {
        panic(err)
    }
    fmt.Println("Hello", num)
}
```

- multiple return values
- `panic` for bugs
- verbose?

Rust: Return Codes

```
fn not_42(num: i32) -> Result<i32, ()> {
    match num {
        42 => Err(()),
        _ => Ok(num)
    }
}

fn main() {
    let number = match not_42(42) {
        Ok(num) => num,
        Err(e) => panic!("{:?}", e),
    };
    println!("Hello, {}!", number);
}
```

Rust: `unwrap`

```
let number = not_42(42).unwrap();
```

- returns value on success
- triggers `panic!` on failure

Rust: `?` Operator

```
let num = not_42(42)?;
```

```
let absolute_path =
    env::current_dir()?.join(relative_path);
```

- evaluates to value on success
- returns `Err` Result on failure

Parallel Evolution

Midori, Rust, Go, Swift all similar

- all distinguish bugs vs. recoverable errors
- all avoid invisible control flow

Evolution Is Slow...

Thank you!

P.S. Hire Me.

- Software Engineer
- Co-organizer of Papers We Love San Diego
- Past Apache Board Member,
  VP Legal Affairs, VP Incubator

Credits

- Toddler biking backwards: Flickr user amseaman, CC-BY-ND 2.0
- Bug: Clker-Free-Vector-Images, Pixabay license.