

Functional Programming and Error Handling

Jonathan Aldrich

17-363

Functional programming coverage draws heavily from blog post by Amay B:
<https://medium.com/coderhack-com/functional-programming-using-rust-3776c10cfc6>

Functional Programming

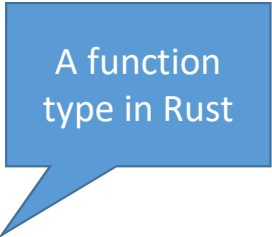
- Benefits
 - Lack of side effects makes programs easier to understand
 - Lack of aliased mutable state makes safe parallelism easier
 - Get a lot of expressiveness from a few language features → programs are short
 - Simplifies some things for the compiler
- Costs
 - Some stateful algorithms are more efficient than purely functional algorithms
 - E.g. Union-find; hash tables
 - Some abstractions have a run-time cost
 - Copying, function calls, pointer use, recursion
 - Sometimes state best fits the programmer's mental model
 - I/O is a good example

Functional Programming Support in Rust

- Tuples and enums for algebraic data types (like ASTs!)
- Pattern matching
- Traits for ad-hoc polymorphism, generics for parametric polymorphism
- Functions are first-class values
- Closures
- Map, filter, fold, etc. library functions for iterators
- Minimal runtime (no GC) and good optimizations lowers cost of recursion, closures by a lot – iterator code is typically as fast as looping
- All mutability is explicit (mut)

First-class Functions

```
fn add(x: i32, y: i32) -> i32 {  
    x + y  
}
```



A function
type in Rust

```
fn call_with_two(func: fn(i32, i32) -> i32, x: i32) -> i32 {  
    func(x, 2)  
}
```

```
call_with_two(add, 1); // Returns 3
```

Functional Operations on Iterators

```
let vec = vec![1, 2, 3, 4, 5];
```



A closure in
Rust

```
let even = vec.iter().filter(|x| x % 2 == 0).collect();  
// even is [2, 4]
```

```
let doubled = vec.iter().map(|x| x * 2).collect();  
// doubled is [2, 4, 6, 8, 10]
```

Closures capture environments

```
let x = 10;
```

We can optionally annotate
parameters and results of
closures

```
let closure = |y : i32| -> i32 {  
  x + y  
};
```

```
let answer = closure(2); // answer is 12
```

Closures can borrow

- Many borrows are immutable, but not all:

```
fn main() {  
    let mut list = vec![1, 2, 3];  
    println!("Before defining closure: {list:?}");  
    let mut borrows_mutably = || list.push(7);  
    // cannot println! from the list here!  
    borrows_mutably();  
    println!("After calling closure: {list:?}");  
}
```

Closures can move, too!

```
use std::thread;
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");
    thread::spawn(move || println!("From thread: {list:?}"))
        .join()
        .unwrap();
}
```


Exercise: What's wrong with this code?

```
struct Rectangle { width: u32, height: u32, }
fn main() {
    let mut list = [ Rectangle { width: 10, height: 1 },
                    Rectangle { width: 3, height: 5 },
                    Rectangle { width: 7, height: 12 }, ];
    let mut sort_operations = vec![];
    let value = String::from("closure called");
    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{list:#?}");
}
```

Different function traits

- FnOnce
 - A closure that may move captured values out of the closure
 - Can only be called once, because we can't move a value out twice
- FnMut
 - A closure that might have a mutable borrow, but doesn't move values out of the closure
- Fn
 - A closure that doesn't do either of the above things

Using FnOnce

```
impl<T> Option<T> {  
    pub fn unwrap_or_else<F>(self, f: F) -> T  
        where F: FnOnce() -> T  
    {  
        match self {  
            Some(x) => x,  
            None => f(),  
        }  
    }  
}
```

Exercise: What kind of closure should `sort_by_key` take?
Fn, FnMut, or FnOnce?

```
struct Rectangle { width: u32, height: u32, }  
fn main() {  
    let mut list = [ Rectangle { width: 10, height: 1 },  
                    Rectangle { width: 3, height: 5 },  
                    Rectangle { width: 7, height: 12 }, ];  
  
    let mut sort_operations = vec![];  
    let value = String::from("closure called");  
    list.sort_by_key(|r| {  
        sort_operations.push(value);  
        r.width  
    });  
    println!("{list:#?}");  
}
```

Watch out!

- Tail call optimizations are not guaranteed in Rust
 - So deep recursions can result in running out of stack space
 - Use iteration for traversing large sequences or data structures
- Inferred closure types are not polymorphic:

// this code has a type error

```
let example_closure = |x| x;
```

```
let s = example_closure(String::from("hello"));
```

```
let n = example_closure(5);
```

- A lot of Rust standard library data structures are mutable
- Borrowing can be a pain with closures – try currying or function composition

Error Handling

- Recoverable or not?
 - This is a basic distinction both in language constructs and in practical use
 - Details may depend on application. Example: Out of memory errors
 - Usually an unrecoverable error
 - Recovering usually uses memory, so what can we do?
 - But clever code can be used to recover
 - Hold a block of memory in reserve, free it immediately on getting the exception
 - Use this “breathing room” to run code that frees memory in an application specific way
- Basic approaches
 - Exceptions
 - Return values
 - Panic

Exception Handling

- What is an exception?
 - a hardware-detected run-time error or unusual condition detected by software
- Examples
 - arithmetic overflow
 - end-of-file on input
 - wrong type for input data
 - user-defined conditions, not necessarily errors

Exception Semantics

- An exception propagates *up the function-call stack* until the top-level scope is reached or until the exception is caught
- Exception handling with try/catch/finally
 - try: a block of code where an exception might be triggered
 - catch: one or more blocks of code, each associated with an exception type
 - The code is run if the corresponding exception is triggered
 - finally: code that always runs
 - Typically cleans up resources

Try-catch example

```
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[42] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of
bounds");
}
```

- Prints:

Top

Caught index out of bounds

Try-catch example, part 2

```
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Caught index out of bounds");
    }
}
```

- Prints:

Top

Caught index out of bounds

Finally example

```
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[42] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```

- Prints:

Top

Caught index out of bounds

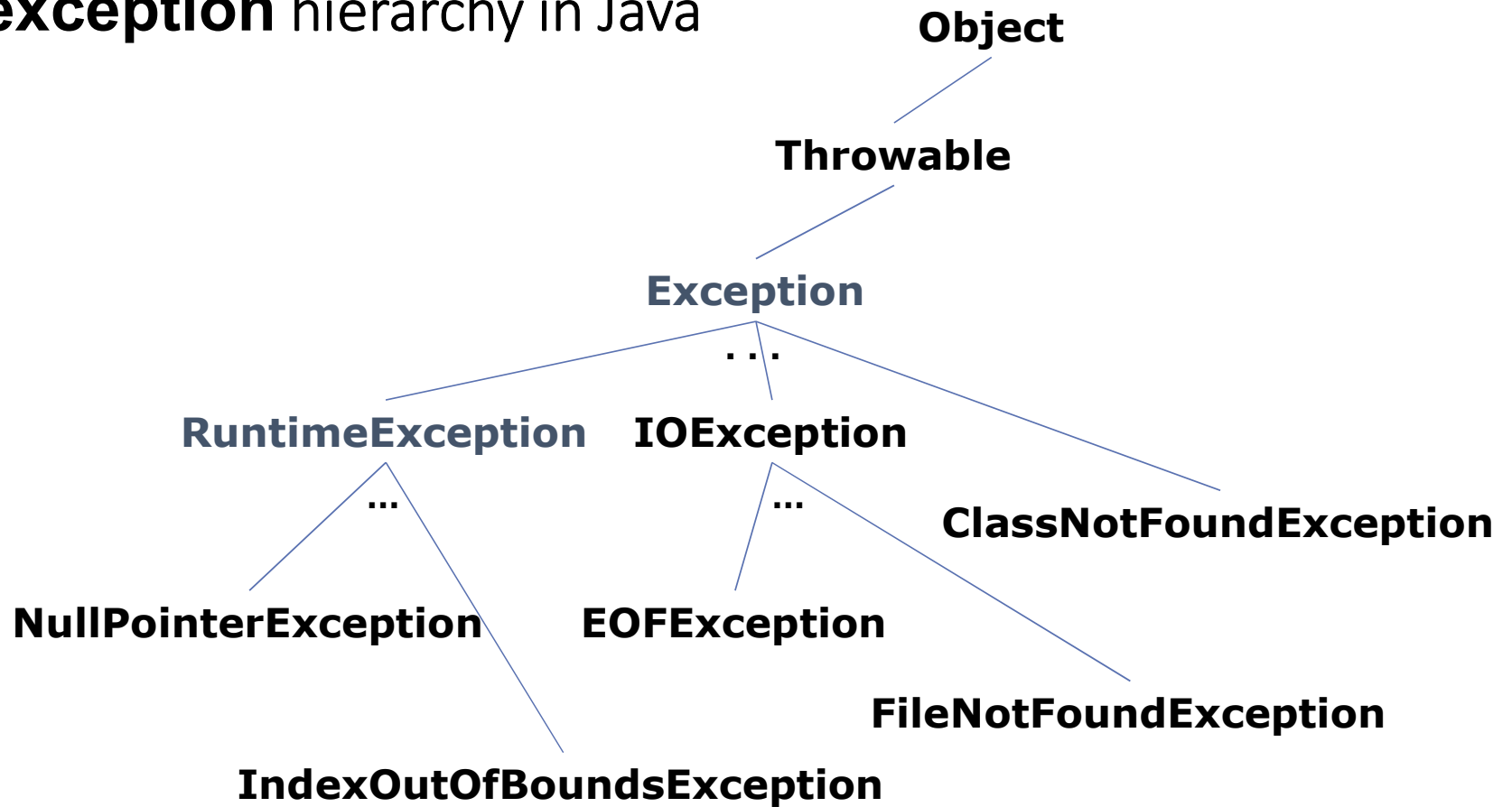
Finally got here

Finally example, part 2

```
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[2] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```

- Prints:
Top
Bottom
Finally got here

The **exception** hierarchy in Java



Checked and unchecked exceptions in Java

- **Unchecked exception:** any subclass of `RuntimeException`
 - Indicates an error which is highly unlikely and/or typically unrecoverable
- **Checked exception:** any subclass of `Exception` that is not a subclass of `RuntimeException`
 - Indicates an error that every caller should be aware of and explicitly decide to handle or pass on
- **Methods must declare any checked exceptions they throw**

Benefits of exceptions

- Provide high-level summary of error and stack trace
 - Compare: core dumped in C
- Can't forget to handle common failure modes
 - Compare: using a flag or special return value
- Can optionally recover from failure
 - Compare: calling `System.exit()`
- Improve code structure
 - Separate routine operations from error-handling
- Reuses error-checking and handling code
 - One catch handles errors from multiple sources
- Allow consistent clean-up in both normal and exceptional operation

Rust doesn't use exceptions. Why not?

- Exceptions are expensive
 - Implementations tend to be slow and have unpredictable performance
- Exceptions aren't explicit
 - The Rust “way” is to be explicit about things—including errors
- Rust's error handling mechanism nevertheless supports most of the benefits of exceptions

Unrecoverable errors

- Simply use the panic! macro:
panic!("an error occurred");
 - Prints the given error message
 - Optionally prints the stack
 - Unwinds the stack and drops data, cleaning up as it goes
 - There is an option just to abort directly though
 - Ends the program
- Example: index out of bound errors cause panics
- You can technically catch a panic (see catch_unwind) but it's rare

Recoverable errors with Result

```
enum Result<T, E> { Ok(T), Err(E), }
```

- The Result type represents the result of an operation or an error
- T is the type of the result, if there is one
- E is the type of the error – e.g. an enum or string

- Example:

```
use std::fs::File;
```

```
fn main() {
```

```
    let greeting_file_result = File::open("hello.txt");
```

```
    let greeting_file = match greeting_file_result {
```

```
        Ok(file) => file,
```

```
        Err(error) => panic!("Problem opening the file: {error:?}"),
```

```
    };
```

```
}
```

Rust: Matching on Errors

```
use std::fs::File;
use std::io::ErrorKind;
fn main() {
    let greeting_file_result = File::open("hello.txt");
    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {e:?}"),
            },
            other_error => { panic!("Problem opening the file: {other_error:?}"); }
        },
    };
}
```

Rust: Matching on Errors

```
use std::fs::File;
use std::io::ErrorKind;
fn main() {
    let greeting_file_result = File::open("hello.txt");
    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {e:?}"),
            },
            other_error => { panic!("Problem opening the file: {other_error:?}"); }
        },
    };
}
```

Rust: Shorter error matching

```
use std::fs::File;
use std::io::ErrorKind;
fn main() {
    let greeting_file_result = File::open("hello.txt");
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {error:?}");
            })
        } else {
            panic!("Problem opening the file: {error:?}");
        }
    });
}
```

Makes the
default case
implicit

Shortcut for panic on error

```
use std::fs::File;  
fn main() {  
    let greeting_file = File::open("hello.txt")  
        .expect("hello.txt should be included in this project");  
}
```

Panics with the given message if there is an error result.

Alternative: `.unwrap()` panics with a generic message (not recommended for production code)

Rust Error Propagation

```
use std::fs::File;
use std::io::{self, Read};
fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };
    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

We just propagate the error to the caller. It works because the error type is the same in both Results

Rust Error Propagation Shortcut

```
use std::fs::File;
use std::io::{self, Read};
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt"?);
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

? means match and evaluate to the result, or propagate the error – same as code on previous slide

When to use panic!

- Examples, prototype code, and tests
 - Keeps examples and prototyping code simple and clean
 - panic! is how you signal that a test failed
- Assertions / bugs in program
 - You know something shouldn't be possible, so call panic! if it happens
- When your code could end in a bad state due to an error, and
 - The error is unexpected (as opposed to, say, opening a file that doesn't exist),
 - Or you can't really check for that bad state constantly afterwards
- Otherwise, use Result

Analysis: Rust vs. Java Error Handling

- Rust more cleanly separates recoverable from unrecoverable
- Rust is more explicit
 - Every error is documented, except for unrecoverable panics
 - Propagation is explicit – at minimum documented with a ?
- Rust can be slightly more efficient
 - Exceptions are rare, but even support for them costs something
- Java can achieve marginally more reuse
 - Exception hierarchies
 - Catch exceptions from several throw points w/o wrapping in a function
- Succinctness is debatable
 - Java avoids the ? for propagation
 - But Rust benefits from functional idioms in handling errors succinctly