

# COMP 322: Parallel and Concurrent Programming

## Lecture 2: Functional Programming Basics

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>



# What is Functional Programming?

---

- Programming Paradigm
- Treats programming as evaluating mathematical functions
- Avoids state
- Avoids mutation (no side effects)
- Recursion
- First-order functions
- Higher-order functions
- Closures
- Composition



# Why Functional Programming?

- Main focus: avoiding mutation of state
- A methodology for solving computation problems without mutating state
- State mutation is one of the biggest source of headaches and complexity in parallel and concurrent programming (more on this later in the course)
- Functional programming paradigm makes programs easier to design and manage when concurrency and parallelism are the goal
- FP is easier to think about before you start writing your code
- FP is easier to test and debug
  - Same inputs yield same outputs every time
- FP abstractions are much easier to run concurrently
- Not a silver bullet!



# Simple example: Lists

```
//  
// Mutating lists  
//  
public class MList {  
    public void push(Object o) { ... }  
    public boolean contains(Object o) { ... }  
    public Object pop() { ... }  
  
    public boolean isEmpty() { ... }  
}
```

```
MList ml = new MList();  
ml.push("Hello");  
ml.push("Rice");  
ml.push("Owls");
```

```
System.out.println(ml.pop()); // Owls  
System.out.println(ml.pop()); // Rice  
System.out.println(ml.pop()); // Hello
```

```
//  
// Functional lists  
//  
public class ObjectList {  
    public ObjectList prepend(Object o) { ... }  
    public boolean contains(Object o) { ... }  
    public Object head() { ... }  
    public ObjectList tail() { ... }  
    public boolean isEmpty() { ... }  
}
```

```
ObjectList l = ObjectList.empty()  
    .prepend("Hello")  
    .prepend("Rice")  
    .prepend("Owls");
```

```
System.out.println(l.head()); // Owls  
System.out.println(l.tail().head()); // Rice  
System.out.println(l.tail().tail().head()); // Hello
```



# Better Idea: Generic Functional Lists

```
//  
// Generic functional lists  
//  
interface GList<T> {  
    GList<T> prepend(T o);  
    boolean contains(T o);  
    T head();  
    GList<T> tail();  
    boolean isEmpty();  
    ...  
  
    static <T> GList<T> empty() { ... }  
}  
  
GList<String> list = GList.<String>empty()  
    .prepend("Hello")  
    .prepend("Rice")  
    .prepend("Owls");  
  
String s = list.head(); // no typecasting!
```

Type T is the parameter type for empty()

Declare type before static method call



# Two Kinds of Lists: A Cons and an Empty

```
/** Interface for a functional list over generic types. */
public interface GList<T> {
  // Data definition: a GList is one of two things:
  // - Cons: an element of type T, and another GList<T>
  // - Empty

  /** Returns the value of the first element in the list. */
  T head();

  /**
   * Returns a new list equal to the old list without its head() element. If the list is empty, this
   * will throw an exception.
   */
  GList<T> tail();
  ...
class Cons<T> implements GList<T> {
  private final T headVal;
  private final GList<T> tailVal;

  private Cons(T value, GList<T> tailList) {
    this.headVal = value;
    this.tailVal = tailList;
  }
  ...
}
class Empty<T> implements GList<T> {
  ...
}
}
```



# Why not just use *null* for an empty list?

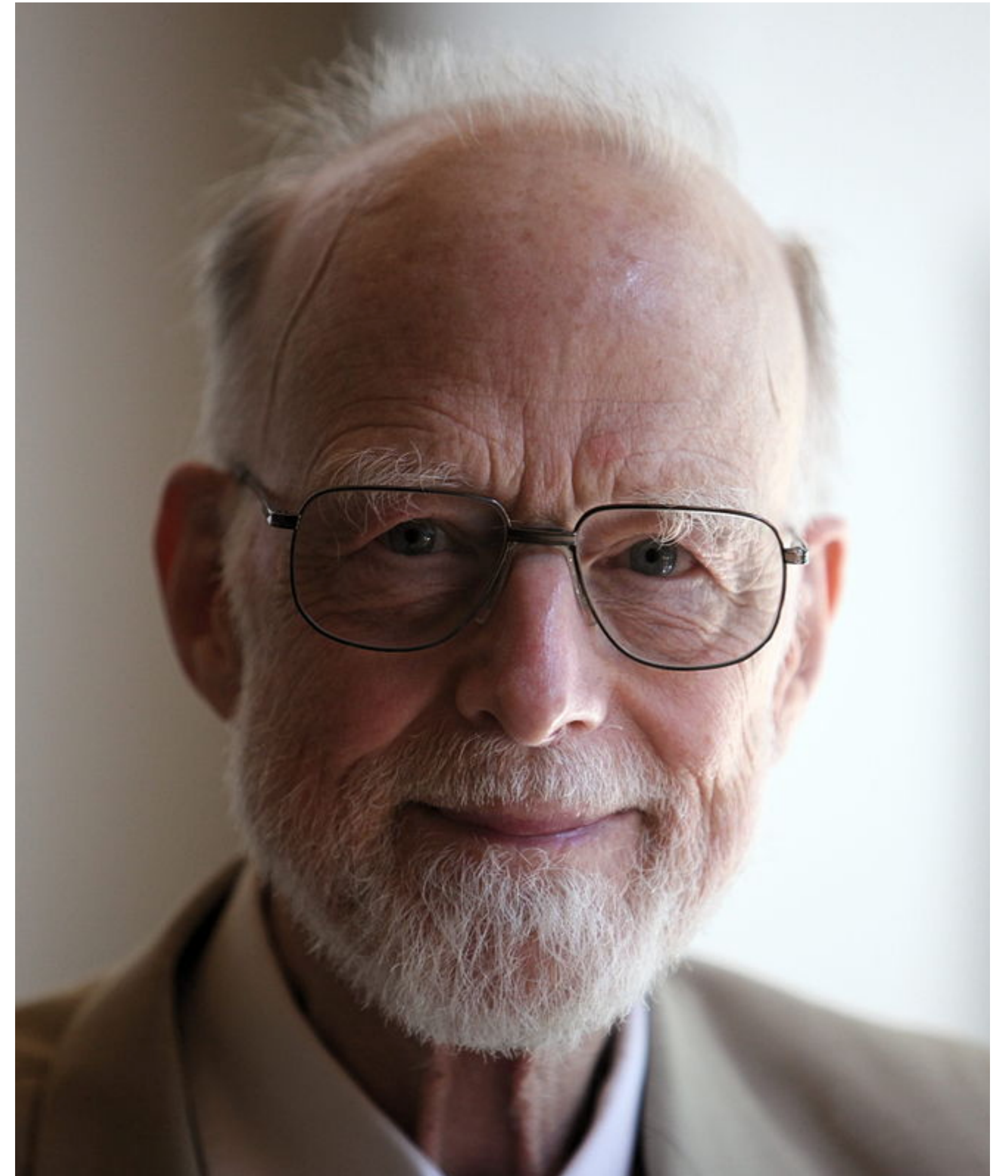
## Sir Tony Hoare on *null* references:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

<http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>

***Java Practices: Avoid null if possible***

<http://www.javapractices.com/topic/TopicAction.do?Id=134>



# There Should Be Only One Empty List!

```
interface GList<T> {  
  
    //Create a new empty list of the given parameter type.  
    @SuppressWarnings("unchecked")  
    static <T> GList<T> empty() {  
        return (GList<T>) Empty.SINGLETON;  
    }  
  
class Empty<T> implements GList<T> {  
    private Empty() { }  
  
    private static final GList<?> SINGLETON = new Empty<>();
```





# Why does this work? *Type erasure!*

- Rule #1: there's only ever one "real" class (GList, etc.)
  - Java only uses type parameters at compile time, so GList<String> and GList<Foo> compile down to just GList
- Implications
  - At runtime, inside GList<T>, we don't know what T actually is
    - Forbidden: `T t = new T();`
  - There's only ever one static method / member of a given name
- `private static final GList<?> SINGLETON = new Empty<>(); // original code`
- `private static final GList SINGLETON = new Empty (); // runtime, after type erasure`



# Why does this work? *Type erasure!*

- Rule #1: there's only ever one "real" class (GList, etc.)

—Java only uses type parameters

- Implications

—At runtime, inside GList<T>, we can

- Forbidden: `T t = new T();`

—There's only ever one static method

- `private static final GList<?> SINGLETON`
- `private static final GList SINGLETON`

Our empty-list never returns a T, so we'll get away with our "cheating".

```
public T head() {  
    throw new NoSuchElementException("can't take head() of an  
empty list");  
}
```

But there's no problem with returning GList<T>, since we get those from the Cons class.

```
public GList<T> prepend(T val) {  
    return new Cons<>(val, this);  
}
```



# Java type inference

- Q: Why don't we need to declare the type parameter of the `Cons<>`?

```
interface GList<T> {
```

```
    default GList<T> prepend(T val) {  
        return new Cons<>(val, this);  
    }
```

- A: Java figures it out from context.
  - IntelliJ will tell you if it can't make an inference.
    - You *must* declare type parameters for return types, argument types.
    - You *often* use a “diamond” `<>` for a constructor's type parameter.
    - You *often* leave out the type parameter (no diamond) for method calls.



## Related concept: *java.util.Optional*<T>

---

Container object

May or may not contain a non-null value

*isPresent()* tells us if the value is present in the container

*get()* returns the value if present, throws *NoSuchElementException* if not

**MUCH** better than using *null* to signal that “there is no answer”

Optional is just like a GList with exactly 0 or 1 elements

Some languages (Haskell) actually implement it that way



# More type inference

This code works:

```
GList<Integer> numbers =  
    GList.<Integer>empty().prepend(1).prepend(2).prepend(3);
```

When in doubt, make yourself a separate empty-list of the correct type.

This code also works:

```
GList<Integer> emptyList = GList.empty();  
GList<Integer> numbers = emptyList.prepend(1).prepend(2).prepend(3);  
GList<Integer> numbers = GList.of(3, 2, 1);
```

This code won't compile:

```
GList<Integer> numbers = GList.<>empty().prepend(1).prepend(2).prepend(3);
```

This code won't compile, either:

```
GList<Integer> numbers = GList.empty().prepend(1).prepend(2).prepend(3);
```



# New in Java10+: *var* declarations

This code works:

```
var numbers = GLList.<Integer>empty().prepend(1).prepend(2).prepend(3);
```

```
var empty = GLList.<Integer>empty();
```

```
var numbers = empty.prepend(1).prepend(2).prepend(3);
```

```
var numbers = GLList.of(3, 2, 1);
```

Java can't guess the type parameter

This code doesn't work:

```
var empty = GLList.empty();
```

```
var empty = GLList.of();
```

```
var empty = GLList.<>empty();
```

```
var numbers = GLList.empty().prepend(1).prepend(2).prepend(3);
```



# var vs. explicit types

Great when it works!

But only works for local variables

```
public class Manufacturer {  
    private final String name;  
    private final String homepageUrl;
```

Member variables (static or instance): you still need explicit

```
private static final Map<String, Manufacturer> registry = new HashMap<>();
```

```
static {  
    for (var m : manufacturers) {  
        registry.put(m.getName(), m);  
    }  
}
```

For-each variables: **var** work great

```
public static Manufacturer lookup(String name) {  
    var result = registry.get(name);  
    if (result == null) {  
        throw new NoSuchElementException(name + " not present");  
    } else {  
        return result;
```

Local variables: **var** work great



# Recursion: list length

```
public interface GList<T> {  
  
    class Cons<T> implements GList<T> {  
        private final T headVal;  
        private final GList<T> tailVal;  
  
        public int length() {  
            return 1 + tailVal.length();  
        }  
    }  
  
    class Empty<T> implements GList<T> {  
        public int length() {  
            return 0;  
        }  
    }  
}
```

