On Wednesday, we built a list of Objects. This works. But it sucks.

```java
class FList {
    public FList add(Object o) { ... }
    public boolean contains(Object o) { ... }
    public Object head() { ... }
    public FList tail() { ... }
    public boolean empty() { ... }
}

FList fl = new FList()
    .add("Hello")
    .add("Rice")
    .add("Owls");

System.out.println(fl.head());  // Owls
System.out.println(fl.tail().head());  // Rice
System.out.println(fl.tail().tail().head());  // Hello
```
The Problem with Objects

When all the world’s an Object, what if you want to treat it differently?

Old school answer: explicit typecast to the proper type.

The problem: what if you’re wrong?
Good style would require more error checking, makes code ugly.

“But I know they’re all Strings”!

```java
FList fl = new FList()
    .add("Hello")
    .add("Rice")
    .add("Owls");

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

String foo = (String) fl.head(); // cast to String
```

Yuck!
Generic APIs

Pronounced “List of T” or “T List”

T is a type parameter.

We can now have lists of String, lists of Foo, whatever.

No need for typecasting.

class FList<T> {
    final T value;
    final FList tailList;

    public FList add(T value) { ... }
    public boolean contains(T value) { ... }
    public T head() { ... }
    public FList tail() { ... }
    public boolean empty() { ... }
}

FList<String> fl = new FList<>()
    .add("Hello")
    .add("Rice")
    .add("Owls");

String foo = fl.head(); // no cast necessary!
Generic APIs

Pronounced “List of T” or “T List”

T is a type parameter.

We can now have lists of String, lists of Foo, whatever.

No need for typecasting.

class FList<T> {
    final T value;
    final FList tailList;

    public FList add(T value) { ... }
    public boolean contains(T value) { ... }
    public T head() { ... }
    public FList tail() { ... }
    public boolean empty() { ... }
}

FList<String> fl = new FList<>()
    .add("Hello")
    .add("Rice")
    .add("Owls");

String foo = fl.head(); // no cast necessary!
Generic APIs

Pronounced “List of T” or “T List”
T is a type parameter.

We can now have lists of String, lists of Foo, whatever.
No need for typecasting.

```
class FList<T> {
    final T value;
    final FList tailList;

    public FList add(T value) { ... }
    public boolean contains(T value) { ... }
    public T head() { ... }
    public FList tail() { ... }
    public boolean empty() { ... }
}

FList<String> fl = new FList<>()
    .add("Hello")
    .add("Rice")
    .add("Owls");

String foo = fl.head(); // no cast necessary!
```
Shirt<T>
Vocabulary Alert

Generics = Parametric Polymorphism

(Type) Parameter = Type arguments in the brackets (<T>, etc.)

Polymorphism = Same code can operate over different types

Remember Zen Coding Rule #1: Don’t Repeat Yourself. Generics help.
What about those singletons?

Beforehand...

class FList {
    final Object value;
    final FList tailList;

    ...

    public static class Empty extends FList {
        private final static Empty singleton = new Empty();

        // we don’t want others using this
        protected Empty() {
            super(null, null);
        }

        // call this to “create” an empty list, even
        // though it just returns one we already have
        public static FList create() {
            return singleton;
        }
    }
}
public static class FList<T> {
    final T value;
    final FList tailList;

    ...}

public static class Empty<T> extends FList<T> {
    final static FList<?> singleton = new Empty<>();

    // we don’t want others using this
    protected Empty() {
        super(null, null);
    }

    // call this to “create” an empty list, even though it just returns one we already have
    public static <T> FList<T> create() {
        @SuppressWarnings("unchecked")
        FList<T> result = (FList<T>) singleton;
        return result;
    }

What about those singletons?

Afterward...

<T>’s spread around in various places.

The singleton is a bit weird.
There’s only one? What’s it’s “instance” type?

<?> ?
What about those singletons?

Afterward...

<T>’s spread around in various places.

The singleton is a bit weird. There’s only one? What’s it’s “instance” type?

Compiler directive.
What about those singletons?

Afterward...

<T>’s spread around in various places.

The singleton is a bit weird. There’s only one? What’s it’s “instance” type?

```java
public static class FList<T> {
    final T value;
    final FList tailList;

    ...

    public static class Empty<T> extends FList<T> {

        final static FList<?> singleton = new Empty<>();

        // we don’t want others using this
        protected Empty() {
            super(null, null);
        }

        // call this to “create” an empty list, even
        // though it just returns one we already have
        public static <T> FList<T> create() {
            @SuppressWarnings("unchecked")
            FList<T> result = (FList<T>) singleton;
            return result;
        }
    }
}
```
Understanding Java generics

Rule #1: there’s only one “real” class (FList, etc.)
Java does type erasure, so FList<String> and FList<Foo> etc. become just FList. At runtime, there’s no such thing as FList<String>

Implications
There’s only ever one static member variable of a given name.

Generic: final static FList<?> singleton = new Empty<>();
Erased: final static FList singleton = new Empty ();

We can get away with mixing and matching FList.Empty<A> for FList.Empty<B>
@SuppressWarnings("unchecked")

At runtime, nobody knows what “T” is.
Not allowed to say new T(...) – a giant headache, sometimes really ugly workarounds
Java can’t always read your mind

What you wish you could write:
FList<String> list = FList.Empty.create().add("Hello").add("Rice");

Java can’t (always) figure out the generic return-type of create(). (Sigh.)

Solution: break it into two lines.
FList<String> emptyList = FList.Empty.create();
FList<String> list = emptyList.add("Hello").add("Rice");
Type restrictions

Sometimes we don’t care, e.g., FList<T>

Sometimes we want to constrain T
Example: for things you put in a tree, they need to be “comparable”

```java
interface Comparable<T> {
    int compareTo(T o);
}
```

If we had a tree class, we might say:

```java
class FTree<T extends Comparable<T>> {
    ...
}
```
Multiple type parameters? No problem.

```java
public class Pair<A,B> {
    final public A a;
    final public B b;

    public Pair(A a, B b) {
        this.a = a;
        this.b = b;
    }
}

We'll use this when we want to return more than one thing at a time.

return new Pair<>(thing1, thing2);
```
Generics + Class Inheritance = Headache

For now, we’ll keep it simple but here’s the issue. Simplified.

```java
class B extends A {
    ...
}
FList<B> flB = ...;
flA = flB; // error!
flB = flA; // error!
```

Anybody who can deal with A can also deal with B, because B extends A.

But, “List of B” doesn’t drop in where “List of A” normally goes.

Why?

```java
flA = flA.add(new A(...)); // fine
flA = flA.add(new B(...)); // also fine, because B extends A
flA = flB; // error, but let’s pretend it’s okay
flA = flA.add(new A(...)); // you just added an A onto a list of B!
```
Generics + Class Inheritance = Headache

FList<B> expects B, not B’s supertype A
If you could assign flA = flB, then you could violate the rule at runtime. Java forbids this.

Anybody who can deal with A can also deal with B, because B extends A.
But, “List of B” doesn’t drop in where “List of A” normally goes.
Why?
flA = flA.add(new A(...)); // fine
flA = flA.add(new B(...)); // also fine, because B extends A
flA = flB; // error, but let’s pretend it’s okay
flA = flA.add(new A(...)); // you just added an A onto a list of B!
Generics + Class Inheritance

For now, just deal with List\(<A>\) and List\(<B>\) not being interchangeable.

Later on, we’ll learn about “type wildcards” that work around this. And, while we’re at it, why functional lists give us more flexibility than mutating.

If you’re bored, go read up on “covariance” and “contravariance”. There’s a lot of deep theory here that we’re totally ignoring today. (Coming in week 6!)
Cool Java Features
Useful Java feature #1: auto-boxing

Java has two kinds of values: primitive types and objects. int, float, double, long, char, byte, boolean

You can only use object types for generic type parameters
Sorry, FList<int> isn’t allowed

The solution? Java object classes specifically made to wrap the primitives
Integer, Float, Double, Long, Char, Byte, Boolean

FList<Integer> emptyIntList = FList.Empty.create();
FList<Integer> list = emptyIntList.add(5).add(3).add(7);

You pass an “int” and it’s automatically wrapped in an “Integer”
Useful Java feature #2: exceptions

What do you do when you don't have something useful to return?
Example: there really is no head() of an empty list.

Typical ugly solution: return null
But, like we said earlier, null sucks.

Better: throw an exception

```java
public T head() {
    throw new RuntimeException("can't take head() of an empty list");
}
```
@Test
public void testHead() throws Exception {
  FList<String> emptyList = List.Empty.create();
  FList<String> list =
      emptyList.add("Alice").add("Bob").add("Charlie");
  assertEquals("Charlie", list.head());

  assertTrue(emptyList.empty());

  // now, verify that we can't take head of an empty list
  try {
    String foo = emptyList.head();
    fail("Exception should have been thrown");
  } catch (RuntimeException e) {
    // good!
  }
}
Zen coding rule #2: proper exceptions

Exceptions have an inheritance hierarchy
Create and throw specific exception types for your problem

```java
class EmptyListException extends RuntimeException { ... }
```

Rules of thumb:
- Don’t catch an exception you’re not prepared to deal with
- Catch the most specific exception(s) that you expect you might see
- Throw specific exceptions (not just “RuntimeException”)​

Declared vs. undeclared exceptions
RuntimeException extends Exception extends Throwable
Anybody can throw a RuntimeException or subclass
Other exceptions thrown must be declared

```java
public T head() throws EmptyListException { ... }
```

Forces the caller to do a try/catch block and catch the exception
Zen coding rule #2: proper exceptions

Exceptions have an inheritance hierarchy
Create and throw specific exception types for your problem

```java
class EmptyListException extends RuntimeException {
    ...
}
```

Rules of thumb:
- Don’t catch an exception you’re not prepared to deal with
- Catch the most specific exception(s) that you expect you might see
- Throw specific exceptions (not just “RuntimeException”)

Declared vs. undeclared exceptions
RuntimeException extends Exception extends Throwable
Anybody can throw a RuntimeException or subclass
Other exceptions thrown must be declared

```java
public T head() throws EmptyListException {
    ...
}
```

*Forces* the caller to do a `try/catch` block and catch the exception.

Java 8 “Optional” is even better. Stay tuned.
Live coding demo

Convert FList from Objects to generics

Make sure *null* is properly forbidden

Write unit tests