Generics, Exceptions, Lambdas

Dan S. Wallach and Mack Joiner, Rice University
Type restrictions

Sometimes, you want to constrain a type parameter.
For example, a tree is only defined if you can order the objects.

// included in the standard Java library
interface Comparable<T> {
    // a.compareTo(b): returns 0 if a==b, negative if a<b, positive if a>b
    public int compareTo(T o);
}

A Tree would require T to be Comparable:

class Tree<T extends Comparable<T>> { ... }

Note: T extends, not T implements, even though it’s an interface.
class Pair<A, B> {
  public final A a;
  public final B b;

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

Type inference does the right thing.

return new Pair<"Hello", 27>; // inferred type: Pair<String,Integer>
“Primitive” types: int, char, byte, float, double, …
These are not Objects, cannot be used as generic type parameters. But they’re “fast”, often with single-cycle CPU operations.

“Boxed” types: Integer, Char, Byte, Float, Double, …
These are Objects, can be used as generic type parameters. Have to allocate memory to wrap the primitive type.

return new Pair<="Hello", 27>; // automatically boxes int -> Integer
Exceptions

Recall: we don’t want to ever return null. But what do you do when you don’t have an answer to return?

```java
class Empty<T> implements GList<T> {
    public T head() {
        throw new NoSuchElementException("can't take head() of an empty list");
    }
}
```

Later on, you can “catch” this exception and deal with it. For now, these exceptions will cause your program to halt.

We’ll talk about alternative techniques later in the semester.
Two kinds of exceptions in the Java universe:
Runtime Exception (and classes that extend it)
“Declared” exceptions

If a method declares that it throws a type of exception, then you must “catch” it or pass it onward

More on this later. It gets ugly.

If you’re throwing a RuntimeException, no declaration is necessary.
This is what we’ll be doing for now.

Our goal: have a nice way of failing when something unexpected happens.
Should I create my own kind of exception?

Java already has a long list of predefined exceptions in the standard library. Here are the “RuntimeExceptions“:

- AnnotationTypeMismatchException
- ArithmeticException
- ArrayStoreException
- BufferOverflowException
- BufferUnderflowException
- CannotRedoException
- CannotUndoException
- ClassCastException
- CMMException
- CompletionException
- ConcurrentModificationException
- DataBindingException
- DateTimeException
- DOMException
- EmptyStackException
- EnumConstantNotPresentException
- EventException
- FileSystemAlreadyExistsException
- FileSystemNotFoundException
- IllegalArgumentException
- IllegalMonitorStateException
- IllegalPathStateException
- IllegalStateException
- IllformedLocaleException
- ImagingOpException
- IncompleteAnnotationException
- IndexOutOfBoundsException
- JMRuntimeException
- LSException
- MalformedParameterizedTypeException
- MalformedParametersException
- MirroredTypesException
- MissingResourceException
- NegativeArraySizeException
- NoSuchElementException
- NoSuchMechanismException
- NullPointerException
- ProfileDataException
- ProviderException
- ProviderNotFoundException
- RasterFormatException
- RejectedExecutionException
- SecurityException
- SystemException
- TypeConstraintException
- TypeNotPresentException
- UncheckedIOException
- UndeclaredThrowableException
- UnknownEntityException
- UnmodifiableSetException
- UnsupportedOperationException
- WebServiceException
- WrongMethodTypeException

Maybe you don’t need another one...
Another useful JUnit feature: an assertion that always fails

```java
@Test(expected = NoSuchElementException.class)
public void testEmptyHead() throws Exception {
    // we're ignoring the result and expecting this to
    // throw an exception (see @Test declaration above)
    GList.<String>makeEmpty().head();
    fail("Exception should have been thrown!");
}
```
Zen coding rule #2: proper exceptions

Exceptions have a hierarchy
For example, we’re using NoSuchElementException when you try to take the head() of an empty list.

java.lang.Object
   java.lang.Throwable
      java.lang.Exception
         java.lang.RuntimeException
            java.util.NoSuchElementException

Rule of thumb:
• Throw specific exceptions (not just RuntimeException)

Controversial:
• Don’t declare “throws Exception” except where required (e.g., JUnit)

Coming soon: “catching” exceptions and recovering from them
Monday: some simple lambdas:
Function<Integer,Integer> oneplus = x -> x + 1;

Today: can we write a function that operates on other functions?
Simple example: repeat

Functions can take functions as arguments, return more functions!

```java
Function<Integer,Integer> oneplus = x -> x + 1;

static <T> Function<T,T> repeat(Function<T,T> f, int n) {
  if(n == 0)
    return x -> x;
  else
    return x -> f.apply(repeat(f, n-1).apply(x));
}

Function<Integer,Integer> threeplus = repeat(oneplus, 3);
```
Simple example: *repeat*

Functions can take functions as arguments, return more functions!

Function\(<\text{Integer, Integer}>\) oneplus = \(x \rightarrow x + 1\);

\[
\text{static} <T> \text{ Function}\langle T, T \rangle \text{ repeat}(\text{Function}\langle T, T \rangle \ f, \text{ int } n) \{ \\
    \text{if}(n == 0) \\
    \quad \text{return} \ x \rightarrow x; \\
    \text{else} \\
    \quad \text{return} \ x \rightarrow f.\text{apply}(\text{repeat}(f, n-1).\text{apply}(x)); \\
\}
\]

Function\(<\text{Integer, Integer}>\) threeplus = \text{repeat}(\text{oneplus}, 3);
Simple example: *repeat*

Functions can take functions as arguments, return more functions!

```java
Function<Integer,Integer> oneplus = x -> x + 1;

static <T> Function<T,T> repeat(Function<T,T> f, int n) {
    if(n == 0)
        return x -> x;
    else
        return x -> f.apply(repeat(f, n-1).apply(x));
}

Function<Integer,Integer> threeplus = repeat(oneplus, 3);
```
Functions can take functions as arguments, return more functions!

Function<Integer,Integer> oneplus = x -> x + 1;

static <T> Function<T,T> repeat(Function<T,T> f, int n) {
    if(n == 0)
        return x -> x;
    else
        return x -> f.apply(repeat(f, n-1).apply(x));
}

Function<Integer,Integer> threeplus = repeat(oneplus, 3);

Simple example: repeat
Apply that function to x
Functions can take functions as arguments, return more functions!

```java
Function<Integer, Integer> oneplus = x -> x + 1;

static <T> Function<T, T> repeat(Function<T, T> f, int n) {
    if (n == 0)
        return x -> x;
    else
        return x -> f.apply(repeat(f, n-1).apply(x));
}

Function<Integer, Integer> threeplus = repeat(oneplus, 3);
```

Simple example: `repeat`
Lexical scope

The lambda hangs onto \( f \) and \( n \), after \textit{repeat} exits!

```java
static <T> Function<T, T> repeat(Function<T, T> f, int n) {
    if (n == 0)
        return x -> x;
    else
        return x -> f.apply(repeat(f, n-1).apply(x));
}
```
Lexical scope rules

A lambda can “capture” any primitive type (int, double, etc.)

A lambda can “capture” any final object type

But if it might mutate, that’s an error (“effectively final” is required)
No mutation? No problem!
Lexical scoping is really useful

GList<String> emptyList = GList.makeEmpty();
GList<String> favoriteMajors =
  emptyList.add("COMP").add("ELEC").add("FWIS");
GList<String> manyClasses =
  emptyList.add("COMP140").add("COMP182").add("ELEC220");

GList<String> favoriteClasses =
  manyClasses.filter(c -> !favoriteMajors
    .filter(m -> c.startsWith(m))
    .empty());
Folding

Let's say we have a list of numbers to add: \{ 1, 2, 3, 4, 5, 6, 7, 8 \}

What order should we add them?
Fold-left: \((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8\nFold-right: 1 + (2 + (3 + (4 + (5 + (6 + (7 + 8)))))))

For integer addition, we get the same answer, but not for others
GList<Integer> intList = ...
int sum = intList.foldr(0, (x,y) -> x + y);
Many uses for folding

List of strings
Join into one string, find longest string, find shortest string, etc.

List of integers
Minimum, maximum, average, sum, etc.

List of bitmap images
Overlay images, concatenate horizontally, etc.

Vocabulary note: “fold” and “reduce” are synonyms. If you’ve heard of “MapReduce”, this is broadly how it works. (More in the coming weeks.)
fold-right, implemented

Fold-right: $1 + (2 + (3 + (4 + (5 + (6 + (7 + 8))))))$

class Cons<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return operator.apply(headVal,
                              tailVal.foldr(accumulator, operator));
    }
}

class Empty<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return accumulator;
    }
}
Fold-right, implemented

Fold-right: \(1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 0)))))))\)

class Cons<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return operator.apply(headVal, tailVal.foldr(accumulator, operator));
    }
}

class Empty<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return accumulator;
    }
}

The accumulator is going to end up over here.
Fold-right, implemented

Fold-right: \(1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 0)))))),)\)

class Cons\(<T>\) implements GList\(<T>\) {
    public T foldr(T accumulator, BinaryOperator\(<T>\) operator) {
        return operator.apply(headVal, tailVal.foldr(accumulator, operator));
    }
}

class Empty\(<T>\) implements GList\(<T>\) {
    public T foldr(T accumulator, BinaryOperator\(<T>\) operator) {
        return accumulator;
    }
}

First, we’re going to recursively fold the tail.
fold-right, implemented

Fold-right: $1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 0)))))))$

```java
class Cons<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return operator.apply(headVal, tailVal.foldr(accumulator, operator));
    }
}

class Empty<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return accumulator;
    }
}
```

Bottoming out at the empty list.
fold-right, implemented

Fold-right: $1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 0)))))))$

class Cons<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return operator.apply(headVal, tailVal.foldr(accumulator, operator));
    }
}

class Empty<T> implements GList<T> {
    public T foldr(T accumulator, BinaryOperator<T> operator) {
        return accumulator;
    }
}

Finally, we can do the addition on headVal
Let’s use folding functions!
(Several of these are in your project this week.)