Introduction to Java (cs2514)

Lecture 15 & 16: Generics

M. R. C. van Dongen

March 6, 2017

Outline

- These lectures study *generic classes*.
  - Generic classes help us detect certain kinds of errors.
  - They remove the need for certain run-time checks.
  - They also allow class-reuse for specialised versions of the classes.
- This lecture is based on [Naftalin, and Wadler 2009].
- At the end we shall implement a generic linked list class.
- Some of this lecture is based on the Java API documentation.

Motivation for Generics

Don't Try This at Home

```
public class RunTimeException {
    public static void main( String[] args ) {
        Object[] things = new Object[ 2 ];
        things[ 0 ] = "mistake";
        things[ 1 ] = 1;
        Integer i = (Integer)things[ 1 ];
        i = (Integer)things[ 0 ]; // bummer.
    }
}
```

What's Going On?

- Many applications require collections of type-T objects.
- Program manipulates a collection, C, using objects of type T.
- To maximise re-use, C is implemented as collection of `Object`.
- Since `Object` is a superclass of T:
  - The compiler cannot assume C consists of type T objects.
  - Run-time errors may occur when taking things from C.
  - Run-time checks have to be added: performance degradation.

What's Going On?

- Many applications require collections of type-T objects.
- Program manipulates a collection, C, using objects of type T.
- To maximise re-use, C is implemented as collection of `Object`.
- Since `Object` is a superclass of T:
  - The compiler cannot assume C consists of type T objects.
  - Run-time errors may occur when taking things from C.
  - Run-time checks have to be added: performance degradation.
- It would be nicer if we could tell the compiler:
  - Trust me, all object in C are instances of (subclasses of) T.

What's Going On?

- Many applications require collections of type-T objects.
- Program manipulates a collection, C, using objects of type T.
- To maximise re-use, C is implemented as collection of `Object`.
- Since `Object` is a superclass of T:
  - The compiler cannot assume C consists of type T objects.
  - Run-time errors may occur when taking things from C.
  - Run-time checks have to be added: performance degradation.
- It would be nicer if we could tell the compiler:
  - Trust me, all object in C are instances of (subclasses of) T.
    - This would help us detect/fix errors at compile time.
    - This would avoid errors at runtime.
    - This would increase efficiency.

Solution: Generic Types

- A *generic class* depends on one or several type parameters:
  - A list with instances of the same class,
  - A binary tree with instances of the same class,
  - ...
- Instances of generic classes must have specific types:
  - A list of `JButton` objects,
  - A binary tree of `Integer` objects,
  - ...
- Generic types are used in combination with *collections*.
  - Lets you add objects to/remove objects from the collection.
- `Java` collection classes are all implemented as generic classes.
- If a generic class, `G`, is parameterised over a type, `T`.
  - You write `G<T>`.
- It guarantees that all objects “in” `G` have the same type: `T`.
  - Allows programmer to state what’s in the collection.
  - Allows the compiler to detect errors at compile time.
  - They eliminate the need for adding certain runtime checks.
  - They avoid runtime errors.
  - Avoids code duplication.

Example

Don't Try This at Home

```
import java.util.*;

public class CompileTimeError {
    public static void main( String[] args ) {
        ArrayList<Integer> nums;
        nums = new ArrayList<Integer>( );
        nums.add( "mistake" ); // compile-time error
        nums.add( 1 );
        Integer i = nums.get( 1 );
        i = nums.get( 0 );
    }
}
```

The Comparable Interface

- An important interface is `Comparable`.
- A `Comparable` object can compare itself to other objects.
- To implement `Comparable<T>` you must override the method
  - `int compareTo( T that )`.
- `compareTo( )` should implement deep comparison.
- Comparison depends on result of `compareTo( that )`:

Negative: `this` is less significant than `that`.
Positive: `this` is more significant than `that`.
Zero: `this` and `that` are equally significant.

Example

Subtracting `attribute` May Result in Overflow

Java

```
public class Example implements Comparable<Example> {
    int attribute;
    @Override
    public int compareTo( final Example that ) {
        return ( this.attribute < that.attribute ? -1 :
                 this.attribute > that.attribute ?  1 : 0 );
    }
}
```

Example

Subtracting attribute May Result in Overflow

Java

```
public class Example implements Comparable<Example> {
    int attribute;
    @Override
    public int compareTo( final Example that ) {
        return ( this.attribute < that.attribute ? -1 :
                 this.attribute > that.attribute ?  1 : 0 );
    }
}
```

A Simple Generic Class

Java

```
public class GenericClass<T> {
    private T attribute;

    public GenericClass( T value )      { attribute = value; }
    public T getAttribute( )            { return attribute; }
    public void setAttribute( T value ) { attribute = value; }
}
```

Using the Generic Class

Java

```
public class SimpleMain {
    public static void main( String[] args ) {
        GenericClass<Integer> gi;
        GenericClass<Double>  gd;

        gi = new GenericClass<Integer>( 42 );
        gd = new GenericClass<Double>( 3.14 );

        final Integer oi = gi.getAttribute( );
        final Double od = gd.getAttribute( );

        System.out.println( oi + " " + od );
    }
}
```

Substitution Principle

When Java expects a value of a given type, you may also provide a value of a subtype of that type.

Example

Java

```
import java.util.ArrayList;

public class Example {
    public static void main( String[] args ) {
        final ArrayList<Number> nums;
        nums = new ArrayList<Number>( );
        nums.add( 42 );
        nums.add( 3.14 );
        System.out.println( nums );
    }
}
```

Subtyping

ArrayList<Integer> Does not Extend ArrayList<Number>

Don't Try This at Home

```
final ArrayList<Number> nums = new ArrayList<Number>( );
final ArrayList<Integer> ints;

ints = nums; // compile-time error.
nums.add( 3.14 );
// ints.toString == "[3.14]" ?
```

Subtyping

`ArrayList<Number>` Does not Extend `ArrayList<Integer>`

Don't Try This at Home

```
final ArrayList<Number> nums;
ArrayList<Integer> ints = new ArrayList<Integer>( );

nums = ints; // compile-time error.

nums.add( 3.14 ); // nums is alias of ints.
// ints.toString == "[3.14]" ?
```

Wildcards with `extends`

Java

```java
public interface Collection<T> {
  ...
  public boolean addAll( Collection<? extends T> collection );
  ...
}
```

- `dest.addAll( source )` adds all items from `source` to `dest`.
- Only makes sense if the things in `source` are subtypes of `T`.
- The spell `?` in `Collection<? extends T>` is a *wildcard*.
  - It is any type (class/interface) extending `T`.
- So `Collection<? extends T> collection` guarantees that:
  - Any object in `collection` is-a `T`.
- Assume `Sub` is some subtype of some type `Sup`.
  - Then `Collection<? extends Sup>` is supertype of `Collection<Sub>`.

Example

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );
ArrayList<? extends Number> nums;

ints.add( 42 );

nums = ints; // Not allowed before.
Number num = nums.get( 0 ); // grand
```

Don't Try This at Home

```
nums.add( 3.14 ); // compile-time error
```

Wildcards with super

- We just studied the spell `? extends T`.
- It is for collections consisting of instances from subclasses of `T`.
  - The `?` denotes any subclass of `T`.
  - It lets you safely get things from collections.
  - `Collection<Sub>` is a subtype of `Collection<? extends Sup>`.

Java

```java
final ArrayList<Integer> ints = new ArrayList<>( );
final ArrayList<? extends Number> nums = ints;
```

- `Java` also has a spell `? super T`.
- It is for collections with instances of superclasses of `T`.
  - The `?` denotes any superclass of `T`.
  - The spell `? super T` lets you safely put things into collections.
  - `Collection<? super Sub>` is a supertype of `Collection<Sup>`.

Java

```java
final ArrayList<Number> nums = new ArrayList<>( );
final ArrayList<? super Integer> ints = nums;
```

Wildcards with super

- We just studied the spell `? extends T`.
- It is for collections consisting of instances from subclasses of `T`.
  - The `?` denotes any subclass of `T`.
  - It lets you safely get things from collections.
  - `Collection<Sub>` is a subtype of `Collection<? extends Sup>`.

Java

```java
final ArrayList<Integer> ints = new ArrayList<>( );
final ArrayList<? extends Number> nums = ints;
```

- `Java` also has a spell `? super T`.
- It is for collections with instances of superclasses of `T`.
  - The `?` denotes any superclass of `T`.
  - The spell `? super T` lets you safely put things into collections.
  - `Collection<? super Sub>` is a supertype of `Collection<Sup>`.

Java

```java
final ArrayList<Number> nums = new ArrayList<>( );
final ArrayList<? super Integer> ints = nums;
```

Wildcards with `super`

- We just studied the spell `? extends T`.
- It is for collections consisting of instances from subclasses of `T`.
  - The `?` denotes any subclass of `T`.
  - It lets you safely get things from collections.
  - `Collection<Sub>` is a subtype of `Collection<? extends Sup>`.

Java

```java
final ArrayList<Integer> ints = new ArrayList<>( );
final ArrayList<? extends Number> nums = ints;
```

- `Java` also has a spell `? super T`.
- It is for collections with instances of superclasses of `T`.
  - The `?` denotes any superclass of `T`.
  - The spell `? super T` lets you safely put things into collections.
  - `Collection<? super Sub>` is a supertype of `Collection<Sup>`.

Java

```java
final ArrayList<Number> nums = new ArrayList<>( );
final ArrayList<? super Integer> ints = nums;
```

Wildcards with super

- We just studied the spell `? extends T`.
- It is for collections consisting of instances from subclasses of `T`.
  - The `?` denotes any subclass of `T`.
  - It lets you safely get things from collections.
  - `Collection<Sub>` is a subtype of `Collection<? extends Sup>`.

Java

```java
final ArrayList<Integer> ints = new ArrayList<>( );
final ArrayList<? extends Number> nums = ints;
```

- Java also has a spell `? super T`.
- It is for collections with instances of superclasses of `T`.
  - The `?` denotes any superclass of `T`.
  - The spell `? super T` lets you safely put things into collections.
  - `Collection<? super Sub>` is a supertype of `Collection<Sup>`.

Java

```java
final ArrayList<Number> nums = new ArrayList<>( );
final ArrayList<? super Integer> ints = nums;
```

Using ? super T

Java

```
ArrayList<? super Integer> ints = new ArrayList<Integer>( );
final ArrayList<Number> nums = new ArrayList<Number>( );

ints.add( 42 ); // grand

ints = nums;    // Not allowed before.
nums.add( 1 ); // grand
```

Don't Try This at Home

```
Number num = ints.get( 0 ); // compile-time error.
```

The Get and Put Principle

- Use ? extends E for collections you get Es from.
- Use ? super E for collections you put Es into.
- Use E for collections you get Es from.
- Use E for collections you put Es into.

Java

```
public static <T>
void copy( ArrayList<? super T> destination,
           ArrayList<? extends T> source ) { ... }
```

The Get and Put Principle

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ArrayList<? super Integer> nums;

ints.add( 42 );              // put
Integer i = ints.get( 0 );   // get

nums = ints;
nums.add( 1 );        // put
copy( nums, ints ); // put and get.
copy( ints, ints ); // put and get.
```

Don't Try This at Home

```
copy( ints, nums ); // compile-time error.
```

Linked Lists

Recursive Class Definition

- Java already has an interface called `List`, so we implement our lists as `MyList` instances.
- Each `MyList` instance has an attribute called `nodes`,
  - This attribute represents what's in the list.
  - If the list is empty, the value of `nodes` is `null`.
  - Otherwise, `nodes` represents a non-empty list.
  - Each `Link` instance represents a non-empty lists.
- Each `Link` instance has a *head* and a *tail* attribute.
- The *head* is the first item in the list.
- The *tail* represents the remaining items in the list.

Java

```
public class MyList { private Link nodes; ... }
```

Java

```
public class Link { private Link tail; private Comparable head; ... }
```

Linked Lists

Recursive Class Definition

- Java already has an interface called `List`, so we implement our lists as `MyList` instances.
- Each `MyList` instance has an attribute called `nodes`,
  - This attribute represents what's in the list.
  - If the list is empty, the value of `nodes` is `null`.
  - Otherwise, `nodes` represents a non-empty list.
  - Each `Link` instance represents a non-empty lists.
- Each `Link` instance has a *head* and a *tail* attribute.
- The *head* is the first item in the list.
- The *tail* represents the remaining items in the list.

Java

```
public class MyList { private Link nodes; ... }
```

Java

```
public class Link { private Link tail; private Comparable head; ... }
```

The Class `MyList`

Java

```
public class MyList {
    private Link nodes;

    public MyList( )                              { nodes = null; }
    public void add( final Comparable item ) { nodes = new Link( item, nodes ); }
    public Comparable getHead( )                  { return nodes.getHead( ); }
    public void print( )                          { Link.print( nodes ); }
    public void qsort( )                          { nodes = Link.qsort( nodes ); }
}
```

The Link Class

Java

```
public class Link {
    private Comparable head;
    private Link tail;

    public Link( final Comparable item, final Link list ) {
        head = item;
        tail = list;
    }

    public Comparable head getHead( ) {
        return head;
    }

    /* omitted */
}
```

Printing the List

Java

```
public static void print( final Link list ) {
    if (list != null) {
        final String separator = list.tail == null ? "" : " ";
        System.out.print( list.head + separator );
        print( list.tail );
    }
}
```

Java

```
public static void print( final Link list ) {
    Link link = list;
    while (link != null) {
        final String separator = link.tail == null ? "" : " ";
        System.out.print( link.head + separator );
        link = link.tail;
    }
}
```

Printing the List

Java

```
public static void print( final Link list ) {
    if (list != null) {
        final String separator = list.tail == null ? "" : " ";
        System.out.print( list.head + separator );
        print( list.tail );
    }
}
```

Java

```
public static void print( final Link list ) {
    Link link = list;
    String separator = "";
    while (link != null) {
        System.out.print( separator + link.head );
        separator = " ";
        link = link.tail;
    }
}
```

The Method `qsort( )`

- We shall sort our list using the `QuickSort` algorithm.
- We do not have an array, but we use the same idea:

  Base case: If the list is empty then it is already sorted.

  Recursion: Otherwise:

  1. The list is not empty.
  2. Let `head` be the head of the list.
  3. Partition the tail of the list into two lists `leq` and `gt`:
     - `leq` contains members less than or equal to `head`.
     - `gt` contains members greater than `head`.
  4. Sort `leq` and `gt`.
  5. Add `head` to the front of `gt`. Let `gtExtended` be this list.
  6. Append `leq` and `gtExtended`.

Implementing `qsort( )`

Java

```java
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing qsort( )

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing qsort( )

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing `qsort( )`

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing qsort( )

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing qsort( )

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing `qsort( )`

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing `qsort( )`

Java

```java
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing qsort( )

Java

```
public static Link qsort( final Link list ) {
    final Link result;

    if (list == null) {
        result = list;
    } else {
        final NodeList pivot = list.head;
        final Partition partition = new Partition( pivot, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Appending the Lists

Java

```java
public static Link append( final Link start, final Link end ) {
    final Link result;

    if (start == null) {
        result = end;
    } else if (end == null) {
        result = start;
    } else {
        result = start;
        Link current = start;
        while (current.tail != null) {
            current = current.tail;
        }
        current.tail = end;
    }

    return result;
}
```

Implementing the Inner Class Partition

Java

```
private static class Partition {
    private Link leq; // members less than or equal to the pivot.
    private Link gt;  // members greater than the pivot.

    private Partition( final Comparable pivot, final Link list ) {
        Link leq = null;
        Link gt = null;
        Link link = list;

        while (link != null) {
            // initialise current link
            final Link current = link;
            // prepare link for next iteration
            link = link.tail;
            // add current link to destination partition
            if (pivot.compareTo( current.head ) < 0) {
                current.tail = gt;
                gt = current;
            } else {
                current.tail = leq;
                leq = current;
            }
        }

        this.leq = leq;
        this.gt  = gt;
    }
```

The `main`

`Unix` Session

```
$
```

The `main`

`Unix` Session

```
$ javac Link.java
```

The main

Unix Session

```
$ javac Link.java
Note: Link.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The `main`

Unix Session

```
$ javac Link.java
Note: Link.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Unix Session

```
$
```

The main

Unix Session

```
$ javac Link.java
Note: Link.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Unix Session

```
$ javac -Xlint:unchecked Link.java
```

The `main`

Unix Session

```
$ javac Link.java
Note: Link.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Unix Session

```
$ javac -Xlint:unchecked Link.java
Link.java:62: warning: [unchecked] unchecked call to
                          compareTo(T) as a member of the
                          raw type java.lang.Comparable
                if (item.compareTo( current.head ) < 0) {
                                 ^
...
$
```

The `main`

Don't Try This at Home

```
public class MainSort {
    public static void main( String[] args ) {
        MyList list = new MyList( );

        list.add( 1 );
        list.add( "Bummer!" );
        System.out.println( "Before sort." );
        list.print( );
        System.out.println( );
        list.qsort( );
        System.out.println( "After sort." );
        list.print( );
        System.out.println( );
    }
}
```

Running the Program

Unix Session

```
$
```

Running the Program

Unix Session

```
$ javac *.java
```

Running the Program

Unix Session

```
$ javac *.java
$
```

Running the Program

Unix Session

```
$ javac *.java
$ java MainSort
```

Running the Program

Unix Session

```
$ javac *.java
$ java MainSort
Before sort.
Bummer!
1
Exception in thread "main" java.lang.ClassCastException:
    java.lang.Integer cannot be cast to java.lang.String
        at java.lang.String.compareTo(String.java:108)
        at Link$Partition.<init>(Link.java:62)
        at Link.qsort(Link.java:29)
        at MyList.qsort(MyList.java:9)
        at MainSort.main(MainSort.java:9)
$
```

Generic Linked Lists

Design Options: Instance Method `qsort( )`

Java

```
public class MyList<T extends Comparable<T>> {
    private Link<T> list;

    public MyList( )                   { list = null; }
    public void add( final T item )    { list = new Link<T>( item, list ); }
    public T getHead( )                { return list.getHead( ); }
    public void print( )               { Link.print( list ); }

    public void qsort( ) {
        list = Link.qsort( list );
    }
}
```

Generic Linked Lists

Design Options: Class Method `qsort( )`

Java

```
public class MyList<T> {
    private Link<T> list;

    public MyList( )                  { list = null; }
    public void add( final T item )   { list = new Link<T>( item, list ); }
    public T getHead( )               { return list.getHead( ); }
    public void print( )              { Link.print( list ); }

    public static <S extends Comparable<S>>
    void qsort( final MyList<S> list ) {
        list.list = Link.qsort( list.list );
    }
}
```

Generic Linked Lists

Using T for the Class Method is the Same as Using S

Java

```
public class MyList<T> {
    private Link<T> list;

    public MyList( )                  { list = null; }
    public void add( final T item )  { list = new Link<T>( item, list ); }
    public T getHead( )               { return list.getHead( ); }
    public void print( )              { Link.print( list ); }

    public static <T extends Comparable<T>>
    void qsort( final MyList<T> list ) {
        list.list = Link.qsort( list.list );
    }
}
```

Generic Linked Lists

Java

```
public class Link<T> {
    private T head;
    private Link<T> tail;

    public Link( final T item, final Link<T> list ) {
        head = item;
        tail = list;
    }

    public T getHead( ) {
        return head;
    }

    ...
}
```

Implementing `print( )`

Java

```
public static <S> void print( final Link<S> list ) {
    Link<> visitor = list;
    String separator = "";

    while (visitor != null) {
        System.out.print( separator + visitor.head );
        separator = ",";
        visitor = visitor.tail;
    }
    System.out.println( );
}
```

Implementing `qsort( )`

Java

```
public static <S extends Comparable<S>>
Link<S> qsort( final Link<S> list ) {
    final Link<S> result;

    if ((list == null) || (list.tail == null)) {
        result = list;
    } else {
        final S pivot = list.head;
        final Partition<S> p = new Partition<S>( pivot, list.tail );
        final Link<S> leqSorted = qsort( p.leq );
        final Link<S> gtSorted = qsort( p.gt );
        pivot.tail = gtSorted;
        result = append( leqSorted, pivot );
    }

    return result;
}
```

Implementing append( )

Java

```
public static <S extends Comparable<S>>
Link<S> append( final Link<S> start, final Link<S> end ) {
    final Link<S> result;

    if (start == null) {
        result = end;
    } else {
        result = start;
        Link<S> visitor = start;
        while (visitor.tail != null) {
            visitor = visitor.tail;
        }
        visitor.tail = end;
    }
    return result;
}
```

Implementing the Partition Class

Java

```
private static class Partition<S extends Comparable<S>> {
    private Link<S> leq;
    private Link<S> gt;

    public Partition( final S pivot, final Link<S> list ) {
        Link<S> visitor = list;
        while (visitor != null) {
            final Link<S> current = visitor;
            visitor = visitor.tail;
            if (pivot.compareTo( current.head ) >= 0) {
                current.tail = leq;
                leq = current;
            } else {
                current.tail = gt;
                gt = current;
            }
        }
    }
}
```

Iteration

Java

```
Type[] things = ⟨magic⟩;
for (Type thing : things) {
    ⟨use thing⟩
}
```

Iteration

- Generalised `for` loops work for *any* kind of array.
- Also works for collection classes.

Java

```
final ArrayList<Type> things = new ArrayList<Type>( );
⟨magic⟩;
for (Type thing : things) {
    ⟨use thing⟩
}
```

- But `ArrayList`s aren't arrays.
- So how does this work?

Iterable<E>

- The `ArrayList` class implements the `Iterable` interface.
- To implement the interface you only have to do one thing:
  - Override `Iterator iterator( )`.

Java

```
final ArrayList<String> strings = ⟨magic⟩;
for (String str : strings) {
    // Use string.
}
```

Java

```
final ArrayList<String> strings = ⟨magic⟩;
final Iterator<String> iterator = strings.iterator( );
while (iterator.hasNext( )) {
    String string = iterator.next( );
    // Use string.
}
```

Iterator<E>

`boolean hasNext( )`: Returns true if there are more elements.

`E next( )`: Returns the next element in the iteration.

`void remove( )`: Removes last `E` returned by `next( )`.

Question Time

Questions Anybody?

For Monday

- Study the notes,
- Implement the generic list class.

Acknowledgements

- This lecture is based on [Naftalin, and Wadler 2009].
- Some of this lecture is based on the Java API documentation.

Bibliography I

Naftalin, Maurice, and Philip Wadler [2009]. *Java Generics*. O'Reilly. ISBN: 978-0-596-52775-4.

About this Document

- This document was created with `pdflatex`.
- The $\LaTeX$ document class is `beamer`.