Software Development (CS2500)

Lectures 43 – 45: Generics

M. R. C. van Dongen

January 27, 29, and 31, 2014

Contents

1 Outline

This lecture studies *generic classes.* A generic class depends on one or several type parameters. Using them prevents many commonly occurring errors. Generic classes remove the need for certain run-time checks. Generic classes also allow class-reuse for specialised versions of the classes. The first part of this lecture is based on [Naftalin, and Wadler 2009]. Some of this lecture is based on the `Java` API documentation.

2 Boxing and Unboxing

This section presents old and new information. It starts by briefly recalling boxing and unboxing. It continues by providing some more detail about the caching mechanism for boxed values.

Primitive types such as `int`, `boolean`, `float`, and `double` in `Java` are not objects. *Boxing* turns a primitive type value into an object. The resulting object can represent the value. This turns `int` into `Integer`, turns `double` into `Double`, and so on. Boxing may be done explicitly or implicitly:

Explicitly: The constructors of the boxed classes do explicit boxing: `new Integer( 42 )`, `new Double( 3.14 )`, ...

Implicitly: When you provide a primitive type value and `Java` expects an object, `Java` *automatically* translates the unboxed value to its boxed equivalent. This is called *autoboxing*. The primitive type determines the type of the box. For example, if the primitive type is `int` then the type of the resulting box is `Integer`, and so on. Note that this is different from widening. For example, when a `Double` (an object type) is expected and an `int` is provided, autoboxing will result in an `Integer` because the object type of the box is determined by the primitive type. With widening, the assignment `double d = 1` widens the `1` to `1.0` and then assigns the `1.0` to `d`.

Unboxing turns a boxed value into its equivalent unboxed value. Unboxing may also be done explicitly and implicitly:

Explicitly: If you want to unbox a value explicitly, you can do this in two different ways. First you can use the instance methods of the boxed classes that return the boxed value. These methods are called `shortValue( )`, `intValue( )`, `doubleValue( )`, and so on. You can also unbox a value with casting: `(int)`⟨`Integer` expression⟩.

Implicitly: Implicit unboxing happens when you use a boxed value and `Java` expects a primitive value. With this technique the boxed value is unboxed to the equivalent primitive type value: `Integer` to `int`, `Double` to `double`, and so on. After this it may be coerced to a wider type. For example, if `i` is an `Integer` variable and `d` is a `double` variable, then you may write `d = i`.

2.1 Examples

The following is an example.

```
int intValue = 1;
Integer boxedValue;
boxedValue = new Integer( 42 );    // explicit boxing
boxedValue = intValue;             // auto boxing
intValue = boxedValue.intValue( ); // explicit unboxing
intValue = boxedValue;             // implicit unboxing
```

Notice that automatic unboxing only works if `Java` expects a primitive type value and you provide a value of the primitive type's boxed equivalent. This explains why the following *doesn't* work.

Don't Try this at Home
```
int intValue = 1;
Object boxedValue = intValue; // auto boxing
intValue = boxedValue;        // unboxing doesn't work
```

Adding a cast still doesn't work. For the cast to work, a primitive or `Integer` value is expected. Since `boxedValue` isn't a primitive value *and* isn't an `Integer` the compiler will complain.

Don't Try this at Home
```
int intValue = 1;
Object boxedValue = intValue; // auto boxing
intValue = (int)boxedValue;   // unboxing doesn't work
```

The following fixes the problem with the previous examples. Notice that the cast in the last statement is perfectly valid but not needed and unclear.

```
int intValue = 1;
Object boxedValue = intValue;       // auto boxing
intValue = (Integer)boxedValue;     // unboxing
intValue = (int)(Integer)boxedValue; // unboxing
```

Remember that with autoboxing the type of the primitive value determines the type of the boxed value. For explicit boxing (using the constructors) the primitive argument may be coerced to a wider type. This explains why each of the following are valid.

```
Object i1 = new Integer( 1 );
Integer i2  = new Integer( 2 );
Object d1 = new Double( 3.0 );
Double d2 = new Double( 4D );
Double d3 = new Double( 42 );
```

The following are examples of valid autoboxing expressions. For each of the expressions at the right hand side the expected value is an object type. This triggers the autoboxing. The type of the autoboxed instance is completely determined by the type of the constants.

```
Integer i1 = 1;  // equivalent to: Integer i1 = new Integer( 1 );
Object i2 = 2;   // equivalent to: Object  i2 = new Integer( 2 );
Object d1 = 3.0; // equivalent to: Object  d1 = new Double( 3.0 );
Double d2 = 4D;  // equivalent to: Double  d2 = new Double( 4D );
```
Java

In the following `3` is an `int` literal, which is a primitive type value. The primitive value is provided and an object is expected, so this triggers autoboxing. Since the type of the primitive type expression determines the type of the boxed value, the result of the autoboxing is an `Integer`. You cannot assign an `Integer` object reference value to a `Double` object reference variable, so the following results in an error.

```
Double d = 3; // Equivalent to: Double d = new Integer( 3 );
```
Don't Try this at Home

2.2 Caching

The boxing operation turns a primitive value into an object. There is no guarantee that a given primitive value is always mapped to the same object.

```
Integer fst = 12345;
Integer snd = 12345;
assert( fst == snd ); // May fail.
```
Don't Try this at Home

However, for efficiency reason the boxed equivalents of "small" primitive type values are cached. Specifically, the boxed equivalents of the following values are cached.

`boolean`: all.

`char`: `'\u0000'`, `'\u0001'`, …, `'\u007f'`.

`short`: all.

`short`: `-128`, `-127`, …, `127`.

`int`: `-128`, `-127`, …, `127`.

This explains why the following is safe.

```
Integer fst = 12;
Integer snd = 12;
assert( fst == snd );
```
Java

3 Motivation

This section provides a first motivation for generic types. Throughout this section we study different techniques for representing classes of similar objects in "collections." Ideally, the collections should be type-safe so they shouldn't be "polluted" with objects that have the wrong type.

The following shows that an `Object` array is not a good representation for a general-purpose, type-safe datastructure.

```
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.
    }
}
```

Don't Try this at Home

The first three statements in the `main` are pretty obvious. When you take a value from the array, all the compiler knows is that it's an `Object` reference. If you want to use such values as an `Integer`, you have to use a cast it, which is inconvenient. Casting the `Object` to `Integer` at runtime requires a check because the JVM must make sure the conversion is valid. Invalid runtime casts will lead to a runtime error, which is what will happen when the last line is executed: casting a `String` to `Integer` is not allowed. Unfortunately, the compiler cannot check the last statement makes this program fail at run time. The previous example is a specific case of a common cause of many problems:

- To make the collection as flexible as possible, we have to make its type as general as possible.
- Because the type is too general, the implementation cannot be type-safe.

Many applications require collections consisting of type-T objects. (In the previous example, the array played the role of the collection.) A program manipulates a collection, C, using objects of type T. To maximise reuse C is implemented as a 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.

4 First Solution

Generic classes provide a solution to our problem. A *generic class* depends on one or several type parameters. For example: 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. For example: a list of `JButton` objects, a binary tree of `Integer` objects, … Generic types are usually used in combination with *collections*. A collection lets you add objects to and remove objects from the collection. The `Java` collection classes are all implemented as generic classes.

If a generic class, `G`, is parameterised over a type, `T`. The resulting class is written `G<T>`. This is pronounced: `G` *of* `T`. The generic class guarantees that all objects "in" `G` have the same type: `T`. (Note that in general, an instance of `T` may be an instance of a subclass of `T`.)

○ Generic types allow the programmer to state what's in the collection.
○ They help the compiler to detect errors at compile time.
○ They eliminate the need for adding certain runtime checks.
○ They avoid runtime errors.
○ They avoid code duplication.

The following example demonstrates the use of generic types. The example is similar to the example from the previous section but this time we use a generic `ArrayList` instance for the collection. Using the generic class notation the programmer can specify what should in the collection: `Integers`. Notice that this time we need no casts when we take things out of the collection.

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 );
    }
}
```

As explained before, the class `CompileTimeError` is equivalent to the class `RunTimeException` on Page 3, except that now we're using a generic `ArrayList` of `Integer` objects instead of an array of `Object`. With generics the compiler can detect the error at compile time, which was impossible with the previous example. In general using generics helps detecting many similar kinds of errors.

5 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 )`. The method `compareTo( )` should implement a deep comparison. The result of the call is an `int` that determines how `that` compares to `this`. There are three classes of return values:

Negative: `this` is less significant than `that`.

Positive: `this` is more significant than `that`.

Zero: `this` and `that` are equally significant.

The following is a simplified example. The class `Example` compares two instances by comparing their `attribute` values. The notation `Comparable<Example>` means that the class only implements `Comparable` for `Example`. By implementing `Comparable<Example>` we can compare `Example` objects with `Example` objects. The compiler will complain if you try to compare `Example` objects with objects that are not instances of the `Example` class.

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

Note that overriding `compareTo( )` by letting it return `this.attribute - that.attribute` is not correct because this may result in overflow, e.g. if `this.attribute == Integer.MAX_VALUE` and `that.attribute` is negative.

Also note that we could also have implemented the `Comparable` interface (as opposed to `Comparable<Example>`). Implementing `Comparable` is equivalent to `Comparable<Object>`. So, if you implement `Comparable` or `Comparable<Object>` then the signature of `compareTo` should be `int compareTo( Object that )`. If you implement this interface then `Example` objects can be compared to any kind of `Object`.

6 A Simple Generic Class

The following is an example of a simple generic class. Further on you may find an example that uses this simple 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; }
}
```

The `<T>` after the name of the class on the first line signifies that this is a generic class. Inside the class the `T` acts as a formal type parameter that can only be used for object types. However, `T` isn't a concrete type, so you cannot use it in a cast and you cannot use it as a constructor for arrays, so `new T[ ... ]` is not allowed. In general, you can use the `T` as if it was `Object` (this is how the class is implemented). However, the `T` does provide *some* information, so you can only use a `T` expression if `Java` expects `T` or `Object`.

The body of the class is quite simple, except that you see quite a few `T`s. For example, the instance attribute `attribute` is a `T`.

The class definition is *generic* because you can specialise the `T` for any existing object type when you *use* the class. If you want to declare a `GenericClass` reference variable, `var`, that specialises the `T` to `Integer` then you declare the variable as

```java
GenericClass<Integer> var;
```

Now, here comes the interesting bit. After this declaration, `Java` assumes the existence of a specialised instance of the generic class `GenericClass`. The class is formed by substituting `Integer` for the type parameter `T` in the definition of `GenericClass`.

Java

```java
public class SpecialisedClassInteger {
    private Integer attribute;

    public SpecialisedClassInteger( Integer value ) { attribute = value; }
    public Integer getAttribute( )                  { return attribute; }
    public void setAttribute( Integer value )       { attribute = value; }
}
```

The name of the class is made up; the idea is that you can use `GenericClass<Integer>` as if it was equivalent to `SpecialisedClassInteger`.

After the declaration `GenericClass<Integer> var`, you can use `var` to access everything inside the `GenericClass` class with `Integer` substituted for `T`. So the attribute in the class is now an `Integer`. Likewise the method `setAttribute( )` now takes an `Integer` argument. By default the type parameter `T` is `Object`, so writing `GenericClass var` is equivalent to writing `GenericClass<Object> var`.

The following is a simple class with a `main` that uses our generic class.

Java

```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 );
    }
}
```

The first two lines in the `main( )` declare two generic `GenericClass` reference variables. The first line declares the variable `gi`, which is an instance of `GenericClass` of `Integer`. The declaration on the second line works declares the variable `gd`, which is an instance of `GenericClass` of `Double`.

The fourth and fifth line assign values to the variables. The spell `GenericClass<Integer>( 42 )` calls the `GenericClass` constructor with an actual parameter of `42`, which becomes an `Integer` because of autoboxing. The `<Integer>` after the `GenericClass` in the constructor call defines the actual type, `Integer`, of the generic instancce. When you provide the actual type parameter, you establish a contract with the `javac` compiler: the instance variable of the instance referenced by `gi` should be an `Integer`. You should *always* put the actual type inside the angular brackets when you call a generic class constructor. If you omit the type parameter in the constructor call, `javac` will assume the actual type is `Object`. So, for example, omitting the `<Integer>` in the first constructor call is equivalent to calling `new GenericClass<Object>( )`.

The fifth line in the `main( )` works just as the fourth, except that it creates a `GenericClass` of `Double` instance and assigns it to the variable `gd`.

Lines 7 and 8 get an `Integer` and a `Double` from the generic objects `gi` and `gd`. The last line prints `42 3.14`.

7 Subtyping

We've already seen the (Liskov) Substitution Principle which states that:

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

This explains why the following is allowed. After all, `nums` consists of `Number`s and both `Integer` and `Double` are subtypes of `Number`.

Java

```
import java.util.ArrayList;

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

The following is *not* allowed.

Don't Try this at Home

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

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

To understand the example, remember that the second assignment makes `nums` and `ints` aliases: they reference the same instance, which happens to be an `ArrayList` of `Number`. Disallowing the second assignment makes sense. For example, all objects in `ints` should be `Integer` because `ints` is an `ArrayList<Integer>`. If we allowed the second assignment then `ints` would reference a `ArrayList<Number>` instance, which would be bad news because the `ArrayList<Number>` may contain `Number` instances, including `Double`s, which are *not* `Integer` instances. If that was allowed, we couldn't guarantee `ints` contained `Integer` object references only.

It is also true that `ArrayList<Number>` is not *not* a subtype of `ArrayList<Integer>`. This is why the following is not allowed.

Don't Try this at Home

```
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]" ?
```

Again it makes sense that we don't allow the second assignment. For example, if we allowed the assignment, the call `add( 2.0 )` would add a `Double` to `nums`, which is an alias of `ints`. If this was possible, we could no longer guarantee that `ints` consists of `Integer` instances.

8 Wildcards with `extends`

The following lists part of the `Collection` interface, which is an important `Java` interface. As you can see, the interface is generic.

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

The methods `dest.addAll( source )` adds all items in `source` to `dest`. Adding all items from `source` to `dest` only makes sense if the things in `source` are instances of classes that extend `T`, so the method should pose restrictions on the (generic) types of `source` and `dest`. The `<? extends T>` poses this restriction. The wildcard `?` in `Collection<? extends T>` is a *wildcard*. It is any type (class/interface) extending `T`. So writing `Collection<? extends T> collection` guarantees that any object in `collection` is-a `T`.

In the following, let `Sub` be some subtype of some type `Sup`. Note that `Java` considers `Collection<? extends Sup>` a supertype of `Collection<Sub>`, so you may use a `Collection<Sub>` when `Java` expects a `Collection<? extends Sup>`. This includes assigning `Sub` values to `Collection<? extends Sup>` variables.

We can now use a `Collection<Sub>` where a `Collection<? extends Sup>` is expected. This was not possible before. For example in the previous section we could not use assign a `Collection<Integer>` reference to a `Collection<Number>` variable.

In the following, the new notation lets us assign `ints` to `nums` because `nums` can be *any* `ArrayList` that is *known to* contain instances from classes that extend `Number`. Since `Integer` is a subclass of `Number` this is allowed.

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

ints.add( 42 );

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

As before, the following will still result in a compile-time error. The reason for the error is the same as before: if we allowed it, we can put a `double` in the `ArrayList` of `Integer` using `nums` (because it is an alias of `ints`).

Don't Try this at Home
```
nums.add( 3.14 ); // compile-time error
```

The following are the most important aspects of `C<? extends T>`.

- It is a generic notation that makes the class `C` depend on any combination of instances from `T` or classes that extend `T`.
- Anything in the collection is-a `T`, so you may use the polymorphic type `T` when you get things from the collection: `T instance = collection.get( 0 )`.
- If `Sub` is a subtype of `Sup`, then `Java` considers `C<Sub>` a subtype of `C<? extends Sup>`.
- Because we may assign `C<Sub>` values to `C<? extends Sup>` variables, we may have a situation where a collection, `ct`, of type `C<? extends Sup>` is an alias of a collection, `cs`, of type `C<Sub>`. Because of this possibility, you cannot add `Sup` instances to `ct` because such instances aren't allowed in `cs`.

9 Wildcards with super

In the previous section we studied the spell *? extends T*. The spell is for collections consisting of instances from *subclasses* of T. The ? denotes any *subclass* of T. It lets you safely *get* things from collections. Java considers Collection<Sub> a *subtype* of Collection<? extends Sup>.

As you may have guessed Java also has a spell *? super T*. This time the spell is for collections consisting of instances of *superclasses* of T. The ? denotes any *superclass* of T. The spell ? super T lets you safely *put* things into collections. Java considers Collection<? super Sub> a *supertype* of Collection<Sup>.

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

ints.add( 42 ); // grand

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

Now the following is not allowed because ints may contain instances of classes that extend Number.

Don't Try this at Home
```
Number num = ints.get( 0 ); // compile-time error.
```

The following are the most important aspects of C<? super T>.

- It is a generic notation that makes the class C depend on any combination of instances from T or superclasses of T.
- You may safely put a T instance into a collection of type C<? super T>.
- Java considers C<? super Sub> a supertype of C<Sup>.
- You cannot assume that anything you get from a collection of type ? super Sub is a Sup. For example, the collection may contain plain Object instances.

10 The Get and Put Principle

The *Get and Put Principle* helps you remember which wildcard to use.

- You use ? extends E for collections you to get Ts from.
- You use ? super E for collections you put Ts into.
- You use E for collections you want to get Ts from and put Ts into.

The following is an example.

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

In this example, the <T> before the void provides a context for the two Ts in the generic type of the arguments. You always have to provide such contexts for generic class (static) methods because the "normal" generic parameters only restrict the instances of the class. Without it you get an error.

```
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.
```

Not to our surprise the following is not allowed.

Don't Try this at Home

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

11 Linked Lists

This is the first of two sections that study the *linked list,* which is a sequential data structure that supports adding items and removing items. The great advantage of a linked list is that the add operation is very cheap in terms of time and space (memory). For example, if you want to add an item to the start of a linked list, you can do this in constant time. Some linked list representations also support a constant-time operation for adding an item to the end of a list. If you use an array to represent the things in a list, then adding an item is not a constant-time operation.

In this section we study a non-generic implementation. As part of the implementation we shall make a mistake in the `main( )` that will result in a runtime error. In the next section we turn the non-generic implementation into a generic implementation. With the generic implementation the `Java` compiler can detect the cause of the error at compile time. After these two sections it should be clear which implementation should be preferred: the generic one.

Our linked lists are sortable so they should contain `Comparable` things. For simplicity we shall implement linked lists as follows.

- `Java` already has an interface called `List`, so we implement our lists as `MyList` instances.
- Each `MyList` instance has an attribute called `nodes`, which represents what's in the list.
- If the list is empty, the value of `nodes` is `null`.
- Otherwise, `nodes` references an instance that represents a non-empty list.
- The `Link` class represents these 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.

The following is a top-level implementation of `MyList`:

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

The top-level implementation of `Link` is as follows.

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

The `nodes` attribute of an empty `MyList` instance is equal to `null`. The `nodes` attribute of a non-empty `MyList` instance references a proper `Link` instance. As already indicated, these instances can represent a head and a tail. `Link` *instances* always represent non-empty sublist. A `Link` instance with a `tail` attribute that is `null` represents a list consisting of one element.

Note that the `Link` class is defined in terms of itself because the `Link` instances

- have `Comparable` attribute values; and
- have `Link` attribute values.

Such class definitions are called *recursive*.

11.1 The Top-Level

The following is our class `MyList`. Basically, it forwards each complex task to a dedicated class method (`static` method), which is defined in the `Link` class.

```java
public class MyList {
    private Link nodes;

    public MyList( )                    { nodes = null; }
    public void add( Comparable item ) { nodes = new Link( item, nodes ); }
    public Comparable head( )          { return nodes.head; }
    public void print( )               { Link.print( nodes ); System.out.println( ); }
    public void qsort( )               { nodes = Link.qsort( nodes ); }
}
```

Notice that adding a `node` to the list is implemented by calling the constructor of the `Link` class. Getting the head of a list is implemented by getting the `head` attribute of the `nodes` attribute. (It follows that this method will fail if the current `nodes` attribute is `null`.)

11.2 The `Link` Class

The following is the start of the implementation of the `Link` class. The rest is presented further on.

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

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

    /* omitted */
}
```

Looking at the constructor, we see that each `Link` instance is constructed by combining a `Comparable` instance and (1) `null` or (2) another `Link` instance. The first case corresponds to a list of length 1. The second case corresponds to a list that has a length that is one more than the length of the sublist represented by the other `Link` instance.

This combination mechanism lets us construct a chain of `Comparable` instances that are linked by `Link` instances.

- We can obtain the head of the list that is represented by a `Link` instance by getting the `head` attribute of the instance.
- We can get the tail (rest) of the list by getting the `tail` attribute of the instance.
- We can visit all elements in the list by repeatedly getting heads and tails.

This repeated process of head and tail operations is best illustrated by studying the `print( )` method, which is presented in the next section.

11.3 Printing the List

In this section we study the implementation of the class method `print( )`. We start by studying a recursive definition.

```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 );
    }
}
```

It is easy to see how the method should work.

Base case: If the list is empty (`null`) the method should do nothing.

Recursion: Otherwise, the list is non-empty. The method prints the head of the list and then (recursively) prints the tail.

This is exactly how we'd describe the algorithm in pseudo code.

The following is an iterative implementation of `print( )`. The method works in a similar way as the recursive method.

```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;
    }
}
```

The `link` variable visits all visitable `Link` in chain, starting with `list`. For each visited `Link`, the method prints the head of the `Link`.

11.4 Sorting the List

We shall sort our list using the `QuickSort` algorithm. In Lecture 27 we studied an in-situ implementation that sorted an array of items. The algorithm's divide-and-conquer idea should also work for other data structures. This time our items are in a list. The following describes the basic ideas:

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. Using `head` as the pivot, partition the tail of the list into two lists `leq` and `gt`:
   - The list `leq` should contain the members that are less than or equal to `head`.
   - The list `gt` should contain the members that are 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`.

In the following, we shall implement `qsort( )` in such a way that its reuses the `Link` elements in the original input list. In that sense, the method is destructive, as it usually destroys the spine of the original input list. For many applications a destructive implementation is acceptable because they don't need original input list any more. However, if the original input list is still needed after the sorting, we have to implement a non-destructive version of the algorithm. The most efficient way to do this is to make a copy of the input list and apply the destructive algorithm to the copy. The overhead of making the copy is quite acceptable as it only takes linear time in terms of the length of the input list and because sorting the list will require *at least* the same order of time (but usually more).

The following shows the implementation of the algorithm.

```java
public static Link qsort( final Link list ) {
    final Link result;
    if (list == null) {
        result = list;
    } else {
        final NodeList head = list.head;
        final Partition partition = new Partition( head, list.tail );
        final Link leqSorted = qsort( partition.leq );
        final Link gtSorted = qsort( partition.gt );
        final Link gtExtended = new Link( head, gtSorted );
        result = append( leqSorted, gtExtended );
    }
    return result;
}
```

Let's see if we can understand what is going on.

- The method returns `null` if `list` is empty, which is correct: this is the base case.
- Otherwise, `list` is not `null`, so it represents a non-empty list.
- The call to the `Partition` constructor partitions the tail of `list` into two sublists (possibly empty).
- The first sublist is `partition.leq`. It consists of the members of the tail of the original input list that are less than or equal to the head of the original input list. The second sublist is `partition.gt`. It consists of the members of the tail of the original input list that are greater than `head`;
- The constructor call `new Link( head, gtSorted )` puts the head of the original list in front of the sorted version of the second sublist.
- The resulting list is appended to `leqSorted`.

This is exactly what the method is supposed to do according to the pseudo-code.

Note that we could also have used `list == null || list.tail == null` as a condition for the base case. Arguably, this is a better implementation in terms of efficiency.

At this stage we've implemented the main part of our sorting implementation. To finish the implementation we have to implement the method `append( )` and the `static` class `Partition`.

11.5 Appending Lists

The method `append( )` is supposed to append two lists `start` and `end`. The links in `start` should be at the start of the resulting list and the links in `end` should be at the end.

Implementing `append( )` is not much more difficult than the implementation of `print( )`. The following idea lets us append `end` to `start`.

- If one of the input lists is empty, the method can return the other.
- Otherwise the method can append `end` to `start` by assigning `end` to the `tail` attribute of the last `Link` in `start`.
- We can locate the last `Link` in `start` using linear search and by stopping as soon as we've located a `Link` whose `tail` attribute is `null`.

```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;
}
```

Exercise 1. *Implement a recursive version of the algorithm.*

11.6 Partitioning

At this stage, we're almost done with our `Link` class. The only thing we need to do is to implement the `static` class `Partition`.

The main task of the `Partition` class is to provide a constructor that destructively partitions an input list into two output lists. One of the lists should consist of the members in the input list that have `head` attributes that are more significant than some given `Comparable` instance. The other list should consist of the remaining members. The order of the `Link` instances in the output lists doesn't matter.

We may implement this by visiting all `Link` instances in the input list and by adding the current `Link` to its proper destination partion (`leq` or `gt`). The following is a possible implementation.

```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( Comparable pivot, final Link list ) {
        leq = null;
        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;
            }
        }
    }
}
```

Notice that the `Partition` class is a `static` class that is defined in the `Link` class. The following briefly outlines how the constructor works.

- It starts by assigning `null` to the attributes `leq` and `gt`. This initialises the partitions of the `Partition`.
- Using the variable `link` it visits each `Link` of the input list `list`.
- After the assignment `current = link`, the variable `current` is the current visited `Link`.
- Each current `Link` is added to the front of the current items in `leq` if its `head` attribute is less than or equal to `pivot`; otherwise it is added to the current items in `gt`.

It is important, **crucial**, you understand that adding the current link to `leq` or `gt` modifies the value of `current.tail`. This is why we must carry out the assignment `link = link.tail` *before* we add the current link to `leq` or `gt`. For example, if we had carried out the assignment after the `if` statement, adding the current link to `leq` or `gt` would also have changed the value of `link.tail` (because `current` and `link` are aliases.

11.7 Compiling

When we compile our classes, the compiler warns about the `Link` class. The following shows the warnings.

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

The compiler recommends we enable the `-Xlint` flag and recompile the `Link` class. (The `-Xlint` flag enables all recommended warnings. You can turn on special warnings by adding them to the flag. For example `-Xlint:unchecked` tells `javac` to warn for so-called "unchecked" calls.) Since the compiler probably has a good reason for giving this advice, we follow its advice. The following shows what happened. (Some of the lines have been edited and rearranged to improve the presentation.)

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) {
                                ^
...
$
```

Looking at the compiler output, we can understand what triggers the warning. The compiler message relates to the statement `item.compareTo( current.head )` in the `partition( )` method. The compiler is worried because `item` and `current.head` are `Comparable`, which doesn't guarantee they are *compatible.* Stated differently, the compiler is worried about the call `item.compareTo( current.head )`: the type of `item.compareTo( )` may not agree with the type of `current.head`. For example, if `item` is an `Integer` and `current.head` is a `String` then both are `Comparable`, but the call `item.compareTo( )` will fail at runtime because the `Integer` class implements `Comparable<Integer>` but not `Comparable<Object>`.

In general, when the compiler issues warnings like the ones show before, then there's something seriously wrong with your class. *You should always use the flag `-Xlint` and never ignore the warning messages. If you do ignore compiler warnings, you may lose lose marks for your assignments.*[1] As we shall see in a moment, the compiler was right: we can write programs that fail at runtime. Specifically, the runtime error occurs because of the call to `item.compareTo( )`.

The following `main` should demonstrate the problem with our `Link` class.

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( );
        list.qsort( );
        System.out.println( "After sort." );
        list.print( );
    }
}
```

When we run the program we get the following error. Some lines have been edited to improve the presentation.

[1]If you use an IDE to compile your programs, make sure it compiles your classes with the flag enabled.

```
$ 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:8)
        at MainSort.main(MainSort.java:9)
$
```

Unix Session

Remember that Line 62 in `Link.java` is the line that triggered the compiler warning when we compiled the class. As it turns out it is exactly where the program crashed. The reason for the crash is the call (in line 62) `node.compareTo( current.head )` for `String` `node` and for `Integer` `current.head`.

Our program crashed because our class design allowed us to sort lists with incompatible objects. In the following section we shall use generics to overcome this problem.

12 Generic Linked Lists

In this section we shall improve our linked list implementation. We shall improve it in such a way that it will no longer allow us to sort lists with incompatible objects in the same list. We shall make two changes to the previous implementation: we shall make the method `qsort` in the `MyList` class a class method and we shall add generic type information. Besides that we shall not change a single line of code. To shorten the presentation we shall omit the main class.

12.1 Design Options

Before we start, we should consider the consequences of introducing generics and how this may affect the overall functionality of the `MyList` class. In the previous section the elements in the lists *had* to be `Comparable` or otherwise we could no have sorted the lists. If we make our classes generic we have two choices: we can make the classes depend on the generic type `T extends Comparable<T>` or we make the classes depend on the generic type `T`.

- If we make both classes depend on the generic type parameter `T extends Comparable<T>`, our lists will be sortable but we can only put `Comparable` objects in them. This is a serious disadvantage because many applications that don't depend on `Comparable` instances require linked lists. The following option overcomes this issue.
- We parameterise our classes on a generic parameter `T`. To provide sorting, we provide a class method for sorting lists. The class method only accepts lists with members that are comparable and compatible: `public static <S extends Comparable<S>> void qsort( MyList<S> list )`. With this design, we have lists consisting of any kind of members and we can still sort list consisting of `Comparable`, compatible members.

The second design option clearly has more functionality than the first, so we'll base our implementation on that option. As a matter of fact, our approach is also used in the `Collections` framework as it defines a class method `public static <T extends Comparable<? super T>> void sort(List<T> list)` for sorting. This method is slightly more general than our sorting method.

12.2 Implementation

The remainder of this section presents the generic implementation. We start with the `MyList` class.

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

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

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

The following explains the main differences.

- The generic parameter `T` of the generic class `Link` only restricts the actual parameters of *instances* of the class. It doesn't restrict the actual parameters of class methods.
- Since `qsort( )` is now a class method we must define a *context* for the argument. The `static <S extends Comparable<S>>` defines the context: the generic type `S` in the parameter list should be `Comparable`.

The following is the first part of the `Link` class. Notice that the generic type is not restricted by the `Comparable` interface.

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

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

    ...
}
```

To make `print( )` generic, we substitute `S` for `NodeList`. Beside that the method doesn't change.

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

The following is qsort( ). Again, the only difference is that we made the implementation generic. These changes are similar to the changes we made for print( ), except that the context for S is now S extends Comparable<S>, which says that S should be Comparable.

```java
public static <S extends Comparable<S>>
Link<S> qsort( Link<S> list ) {
    final Link<S> result;
    if ((list == null) || (list.tail == null)) {
        result = list;
    } else {
        final S head = list.head;
        final Partition<S> p = new Partition<S>( head, list.tail );
        final Link<S> leqSorted = qsort( p.leq );
        final Link<S> gtSorted = qsort( p.gt );
        result = append( leqSorted, new Link<S>( head, gtSorted ) );
    }
    return result;
}
```

The changes to append( ) are similar:

```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> current = start;
        while (current.tail != null) {
            current = current.tail;
        }
        current.tail = end;
    }
    return result;
}
```

Implementing a generic version of Partition is also straightforward.

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

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

The class is a special case of a generic class. Not only are its members generic, they are also Comparable. The spell Partition<S extends Comparable<S>> states that the S is a generic parameter that is `Comparable`. The remaining `S`s in the class are for the generic parameters.

13 For Monday

Study the lecture notes, and implement the generic list class.

14 Acknowledgements

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

15 Bibliography

References

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