

# Software Development (CS2500)

Lectures 47–49: Generics

M.R.C. van Dongen

February 16–21, 2011

## Contents

1	Outline	2
2	Boxing and Unboxing	2
2.1	Examples . . . . .	2
2.2	Caching . . . . .	3
3	Motivation	4
4	First Solution	5
5	Comparable	5
6	Simple Generics	6
7	Subtyping	7
8	Extends Wildcards	8
9	Super Wildcards	9
10	Get and Put	9
11	Linked Lists	10
12	Generic Lists	14
13	For Wednesday	16
14	Acknowledgements	16

## 1 Outline

This lecture studies *generic types*. Generic types help us detect certain kinds of errors. In addition they remove the need for certain run-time checks. They allow class-reuse for specialised versions of the classes. 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 start 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 into an equivalent object type. This turns `int` into `Integer`, turns `double` into `Double`, and so on. Boxing may be done explicitly or implicitly:

**Explicit:** Explicit boxing is done with the constructors of the boxed classes: `new Integer( 42 )`, `new Double( 3.14 )`, ...

**Implicit:** Implicit boxing happens when an object is expected where a primitive type is provided. In this case Java *automatically* translates the unboxed value to its boxed equivalent. This is called *autoboxing*. The box type is determined by the primitive type. For example, if the primitive type is `int` then the type of the resulting box is `Integer`, and so on.

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

**Explicit:** There are two methods to explicitly unbox a boxed value. The first method uses the instance methods of the boxed classes which return the boxed value. These methods are called `shortValue()`, `intValue()`, `doubleValue()`, and so on. The second method to unbox a value is casting.

**Implicit:** Implicit unboxing is done by using a boxed value where a primitive value is expected. If this technique is used, the boxed value is unboxed to the type of the boxed value. 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 = Integer( 42 );           // explicit boxing
boxedValue = intValue;                // auto boxing
intValue = boxedValue.intValue();    // explicit unboxing
intValue = boxedValue;                // implicit unboxing
```

Notice that automatic unboxing only works if a primitive type value is expected and a value of its boxed type equivalent is provided. This explains why the following *doesn't* work.

```
int intValue = 1;
Object boxedValue = intValue; // auto boxing
intValue = boxedValue; // unboxing doesn't work
```

Don't Try this at Home

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.

```
int intValue = 1;
Object boxedValue = intValue; // auto boxing
intValue = (int)boxedValue; // unboxing doesn't work
```

Don't Try this at Home

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
```

Java

It is recalled that with autoboxing the type of the primitive value determines the type of the boxed value. For explicit boxing (using the constructors) the 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 );
```

Java

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 result of the autoboxing 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

The following doesn't work because 3 is an int literal, which triggers autoboxing and results in an Integer. Since Integers cannot be assigned to Doubles this 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.

```
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 we try to get values from the array, all the compiler knows is that they're Objects. If we try using such values as Integers then we have to use a cast, which is inconvenient. What is worse, the compiler cannot check the invalidity of the last statement and the program will fail at run time. The previous example is a common cause of many problems.

The following explains why it is symptomatic. 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 avoid certain errors at compile time.
- This would increase efficiency.

## 4 First Solution

*Generic types* provide a solution to our problem. Roughly speaking a generic type is a parameterised type. 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 collections classes are implemented as generic types.

A generic class, `G`, is parameterised over another class, `T`. The resulting class is written `G<T>`. This is pronounced: *G of T*. The generic class guarantees that objects in `G` are instances of `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 allow the compiler to detect errors at compile time.
- They eliminate the need for adding certain runtime checks.
- They allow the compiler to avoid duplication of code.

The following example demonstrates the use of generic types. This time the programmer is allowed to state exactly what's in the collection: `Integers`. Notice that this time no casts are needed when getting things from the collection.

```
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 );
    }
}
```

Don't Try this at Home

This class `CompileTimeError` is equivalent to the class `RunTimeException` on Page 4, except that now we're using an `ArrayList` of `Integer` instead of an array of `Object`. Using generics the compiler can detect the error at compile time, which was impossible with the previous example. In general using generics may help detect many similar kinds of errors.

## 5 The Comparable Interface

An important interface is `Comparable`. As suggested by the name, a `Comparable` object can be compared. To implement `Comparable<T>` you must implement `int compareTo( T that )`. The method `compareTo( )` may be used to implement a deep comparison. The result of the call is an `int` which determines how that compares to this. There are three possible cases:

**Negative:** this is less than that.

**Zero:** this is equal to that.

**Positive:** this is greater than that.

The following is a simplified example. The class `Example` compares two instances by 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 which are not `Example` objects.

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

## 6 A Simple Generic Class

The following is an example of a simple generic class. An example which uses this generic class may be found after this simple class.

```
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 which can only be used for object types. However, `T` is not considered a concrete type, so you cannot use it in a cast. 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 `T` expressions where `T` values or `Object` values are expected.

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`, which specialises the `T` to `Integer` then you declare the variable as

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

After this declaration 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 which uses our generic class.

```
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 );

        System.out.println( gi.getAttribute( ) + " " + gd.getAttribute( ) );
    }
}
```

Java

The first two lines in the main( ) declare two generic GenericClass reference variables. The first line declares the variable ‘gi’ which is a ‘GenericClass of Integer’ (GenericClass<Integer>). The declaration on the second line works declares the variable gd which is a 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 due to autoboxing. The ‘<Integer>’ after the ‘GenericClass’ in the constructor call defines the (actual) type parameter ‘Integer’ which is used to specialise the formal type T in the class definition. You usually use it as part of the call to the constructor. Omitting the ‘<T>’ in the constructor call is equivalent to using ‘GenericClass<Object>( )’.

The fifth line works in a similar way as the fourth, except that it creates a GenericClass of Double and assigns it to the variable gd. The last line prints ‘42 3.14’.

## 7 Subtyping

We’ve already seen the Substitution Principle which states that:

Wherever a value of a given type is expected, one may also provide a value of a subtype of that type.

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

```
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 );
    }
}
```

Java

The following is *not* allowed. The reason why this is allowed is that ArrayList<Integer> is *not* a subtype of ArrayList<Number>.

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

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

Don't Try this at Home

Notice that it makes sense to disallow the second assignment. For example, all objects in `ints` should be `Integer` by the fact that `ints` is an `ArrayList<Integer>`. If we allow the assignment then an `ArrayList<Number>` is assigned to `ints`. The `ArrayList<Number>` may contain `Number` instances, including `Doubles`, which are not `Integer` instances.

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

```
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]" ?
```

Don't Try this at Home

Again it makes sense that this is not allowed. For example, if we allow this then the call to `add` would add a `Double` to `nums`, which is an alias of `ints`. This means we can 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.

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

Java

The methods `dest.addAll( source )` adds all items in `source` to `dest`. This only makes sense if the things in `source` are subtypes of `T`. The '?' in '`Collection<? extends T>`' is a *wildcard*. It is any type (class/interface) extending `T`. Therefore, the spell '`Collection<? extends T>`' is any `Collection<S>` such that `S` extends `T`. So writing '`Collection<? extends T> c`' guarantees that: any object in `c` is-a `T`.

Moreover, Java considers '`Collection<? extends T>`' to be subtype of `Collection<T>`. (This is different from what we've seen in the previous section. For example, `Collection<Integer>` is not a subtype of `Collection<Number>`.)

The spell `Collection<? extends T>` puts a constraint on the allowed `Collections`. Specifically, it can be any `Collection<E>`, where `E` is a subtype of `CollectionT`. Since Java considers `Collection<? extends T>` a subtype of `Collection<T>` we can now also use `Collection<E>` where `Collection<T>` is expected, provided `T` extends `E`. This was not possible before. For example in the previous section we could not assign a `Collection<Integer>` reference to a `Collection<Number>` reference variable.

In the following, the new notation allows us to assign `ints` to `nums` because `nums` can be *any* `ArrayList<E>` such that `E` extends `T`. Since `Integer` is a subtype of `Number` this is allowed.



```

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

ints.add( 42 );

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

```

Java

The following is still not allowed. The reason for the error is the same as before.

```

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

```

Don't Try this at Home

## 9 Wildcards with super

We've just studied the spell '*? extends T*'. It is for collections consisting of *subtypes* of T. The '?' denotes any *subtype* of T. It restricts what can be *taken* from collections. In Java `Collection<? extends T>` is a *subtype* of `Collection<T>`.

Java also has a spell '*? super T*'. It is for collections consisting of *supertypes* of T. The '?' denotes any *supertype* of T. It restricts what can be *added* to collections. In Java `Collection<? super T>` is a *supertype* of `Collection<T>`.

```

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

ints.add( 42 );

superInts = ints; // Not allowed before.
superInts.add( 1 ); // grand
Number num = (Number)superInts.get( 0 ); // grand.

```

Java

Now the following is not allowed.

```

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

```

Don't Try this at Home

It should be clear why this is not allowed. For example, in general we can use `ArrayList<Object>` for `ArrayList<? super Integer>`, but we assigning an `Object` to an `Number` is not valid.

## 10 The Get and Put Principle

The *Get and Put Principle* helps you decide which wildcard to use:

- '*? extends E*' is used for getting things.
- '*? super E*' is used for putting things.
- '*E*' is used for getting and putting.

The following provides an example. The `<T>` before the `void` provides a context for the two `T`s in the wildcards. You have to provide it in the definition for generic static methods. Without it you get an error.

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

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.
```

Java

The reader is invited to confirm why the following is still not allowed.

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

Don't Try this at Home

## II Linked Lists

This is the first of two sections which study *linked lists* in Java. The next section studies a generic implementation but this section studies a non-generic implementation. After studying both sections, it should be clear which implementation should be preferred: the generic one.

Let's implement linked lists in Java. Our linked lists are going to be sortable so they should contain Comparable things. For simplicity we shall implement linked lists as follows.

- Each linked list instance has an attribute `nodes`. This attribute represents what's in the list.
- If the list is empty then `nodes` is `null`.
- Otherwise it 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.

This suggests the following class structure for linked lists.

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

Java

The class structure for `NodeList` is as follows.

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

Java

In what follows, we shall implement the sorting with a class method of `NodeList`. Since inner classes *cannot* have class methods this means we cannot implement `NodeList` as an inner class.

The following is our class `MyList`. Basically, it is a wrapper class for the class `NodeList` which does most of the work. The method `qsort` does the sorting. It is implemented as a class (static) method.

```

public class MyList {
    private NodeList nodes;

    public MyList( ) { nodes = null; }
    public Comparable head( ) { return nodes.head( ); }
    public void print( ) { NodeList.print( nodes ); }
    public void qsort( ) { nodes = NodeList.qsort( nodes ); }
    public void add( Comparable item ) {
        nodes = new NodeList( item, nodes );
    }
}

```

The following is the `NodeList` class. The inner class `Partition` and the method `qsort( )` are presented further on.

```

public class NodeList {
    private Comparable head;
    private NodeList tail;

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

    public Comparable head( ) { return head; }

    public static void print( NodeList list ) {
        while (list != null) {
            System.out.println( list.head );
            list = list.tail;
        }
    }
    /* omitted */
}

```

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. Let  $h$  be the head of the list.
2. Split the tail of the list into two lists  $leq$  and  $gt$ :
  - The list  $leq$  should contain the members which are less than or equal to  $h$ .
  - The list  $gt$  should contain the members which are greater than  $h$ .
3. Sort  $leq$  and  $gt$ .
4. Add  $h$  to front of  $gt$ .
5. Append  $leq$  and  $gt$ .

Unfortunately the append operation is expensive. Fortunately, we can avoid it at the expense of overloading `qsort( )` and adding an extra argument in the (the ?) overloaded version. We overload `qsort( list )`. The overloaded version is `qsort( list, aList )`. Here `aList` is a *continuation* argument. It is

a list that should be appended to the list which is the result of Step 5 of the original algorithm. Then `qsort( list )` is equivalent to `qsort( list, null )`.

The method `qsort( list, aList )` should work as follows:

**Base case:** If `list == null` return `aList`. Clearly this is correct as this is equivalent to appending `list` and `aList`.

**Recursion:** We want to return the result of appending `qsort( list )` and `aList`. The following shows how this may be done.

1. Let `h` be the head of the list.
2. Split the tail of `list` as before.
3. In the original algorithm we have to (1) sort `leq`, (2) sort `gt`, (3) add `h` to `gt`, and (4) append `leq` and `gt`. This time we also have to append `aList`. If we do things in the right order then we can actually achieve this:
  - (a) `gt = qsort( gt, aList )`.
  - (b) Add `h` to front of `gt`.
  - (c) Return `qsort( leq, gt )`.

This works because

```
qsort( leq ).append( (qsort( gt ).append( aList )).add( h ) )
    = qsort( leq ).append( qsort( gt, aList ).add( h ) )
    = qsort( leq, qsort( gt, aList ).add( h ) ).
```

Here `append( NodeList that )` is a method which appends `that` to this.

```
public static NodeList qsort( NodeList start ) {
    return qsort( start, null );
}

private static NodeList qsort( NodeList list, NodeList tail ) {
    if (list != null) {
        Partition p = new Partition( list.head, list.tail );
        list.tail = qsort( p.gt, tail );
        return qsort( p.leq, list );
    } else {
        return tail;
    }
}
```

Java

```

private static class Partition {
    private NodeList leq;
    private NodeList gt;

    private Partition( Comparable item, NodeList list ) {
        while (list != null) {
            NodeList next = list;
            list = list.tail;
            if (item.compareTo( next.head ) < 0) {
                next.tail = gt;
                gt = next;
            } else {
                next.tail = leq;
                leq = next;
            }
        }
    }
}

```

The static before the class is *needed*. The reason why it is required is that the constructor `Partition()` is called from the *static* method `qsort()`. The method `qsort()` can see the instance attributes of its argument(s), but it's not an instance method: it doesn't have a `this`.

When compiling our classes we notice the compiler warns about the class `NodeList`. The following is what happened. (Some lines have been rearranged to improve the presentation.)

```

$ javac NodeList.java
Note: NodeList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ javac NodeList.java -Xlint:unchecked
NodeList.java:43: warning: [unchecked] unchecked call to
                    compareTo(T) as a member of the
                    raw type java.lang.Comparable
                    if (item.compareTo( next.head ) < 0) {
                        ^
1 warning
$

```

Here the flag `-Xlint` 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. Looking at this output, we can understand what triggers the warning. The compiler can only see that `item` and `next.head` are both `Comparable`, but it doesn't know if they are compatible. More specifically, it is worried that the allowed types of the argument of `item.compareTo()` may not include the type of `next.head` for all allowed uses of the class `NodeList`. For example, if `item` is an `Integer` and `next.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 a warning like this it means that there's something seriously wrong with your class. As we shall see in a moment, the compiler was right: we can write programs which fail at runtime. Specifically, the runtime error occurs because of the call to `item.compareTo()`.

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

```

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( );
    }
}

```

Don't Try this at Home

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

```

$ 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:109)
    at NodeList$Partition.<init>(NodeList.java:43)
    at NodeList$Partition.<init>(NodeList.java:35)
    at NodeList.qsort(NodeList.java:20)
    at NodeList.qsort(NodeList.java:15)
    at MyList.qsort(MyList.java:18)
    at MainSort.main(MainSort.java:10)
$

```

Unix Session

It is recalled that Line 43 in `NodeList.java` is the line which triggered the compiler warning when we compiled the class. as it turns out it is exactly where the program crashed. The reason is that this line called `node.compareTo( next.head )` for `Integer` `node` and for `String` `next.head`.

The reason why our program crashed is that our class design allowed us to put incompatible objects in the same list. 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 put incompatible objects in the same list. We shall not change a single line of code, except for the fact that we shall add generic type information. To shorten the presentation we shall omit the main class.

The following is the wrapper for the `NodeList` class. The '`S` extends `Comparable<S>`' is the context for the '`S`' in the '`MyList<S>`' in the `qsort` method. It states that `S` should be a subtype of `Comparable<S>`: it should be possible to compare `Ss`.

```

public class MyList<T> {
    private NodeList<T> nodes;

    public MyList( )          { nodes = null; }
    public void add( T item ) { nodes = new NodeList<T>( item, nodes ); }
    public void print( )      { NodeList.print( nodes ); }

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

```

The following is the `NodeList` class. The implementations for `head( )` and `print( )` have been omitted.

```

public class NodeList<T> {
    public T head;
    public NodeList<T> tail;

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

    public static <S extends Comparable<S>>
    NodeList<S> qsort( NodeList<S> list ) {
        return qsort( list, null );
    }

    private static <S extends Comparable<S>>
    NodeList<S> qsort( NodeList<S> list, NodeList<S> tail ) {
        if (list != null) {
            Partition<S> p = new Partition<S>( list.head, list.tail );
            list.tail = qsort( p.gt, tail );
            return qsort( p.leq, list );
        } else {
            return tail;
        }
    }

    private static
    class Partition<S extends Comparable<S>> { /* omitted */ }
}

```

The inner class `Partition` is implemented as follows.

```

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

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

```

## 13 For Wednesday

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 and Wadler, 2009] Maurice Naftalin and Philip Wadler. *Java Generics*. O'Reilly, 2009.