Programmieren II

Sorting Collections

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from T. Bögel)

June 5, 2014

Outline

Outline

1. Recap
2. Sorting
   - Sorting Collections
   - Sorted Collections
3. Summary

Polymorphism I

- Objects of a concrete *sub class* can be used where *super classes* are expected
- All sub classes have complete functionality of super class
- **But:** special functionality implemented in the sub class cannot be accessed via super class

Polymorphism II

Example

```
public Message filterMessage(Message m, GeneralFilter f) {
        f.apply(m);
        // f.printFilterRegex() would not work
}
...
public void runFiltering(Message m) {
        LinkFilter f = new LinkFilter();
        this.filterMessage(m,f);
}
```

- `filterMessage()` expects `GeneralFilter`
- `LinkFilter` **is** also a `GeneralFilter`
- Each sub class of `GeneralFilter` has a `apply()` method
- `filterMessage()` does not need to know which filter's method it is calling!

Interfaces

Interfaces

- Interfaces define protocols for communication between objects
- Interface declarations only contain method signatures & constants, no implementation
- A class implementing an interface must implement all of its methods
- Interfaces can be used just like other (reference) types

Interfaces as types

Using interfaces as types

- Interfaces are reference types
- Interface name can be used just like any other data type
- Reference variable with interface type must **always point to instance that implements interface**
- E.g. `Relatable rect = new Rectangle();`

Abstract classes

Motivation

- Super classes represent an *abstraction* of sub classes
- Sometimes, however, instantiating the super class does not make sense
- Examples:
  - Animal
  - Shape
  - Person
- University library software knows two kinds of `Person`s: `Student` and `Teacher`
- Instantiating `Person` would be strange

Example: linguistic annotation (token-based)

- You want to define linguistic token-based annotations in a document
- Concrete implementations:
  - Token
  - Lemma
  - PoS tag
  - Word sense
  - ...
- Each linguistic annotation has a start and end position (measured in **token** from beginning of the document)

Example: linguistic annotation

- You (as a developer) want to write different (token-based) annotations to a file
- Linguistic annotations should be implemented by others
- To write an annotation, you need its content
- Super class: `TokenAnnotation`

What do we know about each TokenAnnotation object?

- Each Annotation has a start and end position
- Each Annotation object has a content
- We do not know how this content looks like!
- Content could be very complicated to compute
- We just need a string representing the content (for writing)

→ **We need an abstract class!**

Example: linguistic annotation

- You (as a developer) want to write different (token-based) annotations to a file
- Linguistic annotations should be implemented by others
- To write an annotation, you need its content
- Super class: `TokenAnnotation`

What do we know about each TokenAnnotation object?

- Each Annotation has a start and end position
- Each Annotation object has a content
- We do not know how this content looks like!
- Content could be very complicated to compute
- We just need a string representing the content (for writing)

→ **We need an abstract class!**

Example: linguistic annotation

- You (as a developer) want to write different (token-based) annotations to a file
- Linguistic annotations should be implemented by others
- To write an annotation, you need its content
- Super class: `TokenAnnotation`

What do we know about each TokenAnnotation object?

- Each Annotation has a start and end position
- Each Annotation object has a content
- We do not know how this content looks like!
- Content could be very complicated to compute
- We just need a string representing the content (for writing)

→ **We need an abstract class!**

Abstract class for TokenAnnotation

```
public abstract class TokenAnnotation {
    int start, end;

    public Annotation(int start, int end) {
        this.start = start;
        this.end = end;
    }
    public abstract String getContent();
    public int getStart() {
        return start;
    }
    public int getEnd() {
        return end;
    }
}
```

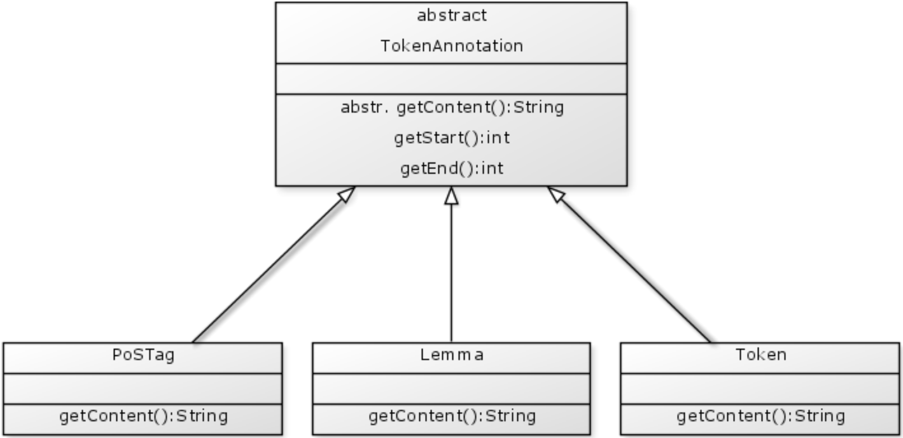
Writer class

Class that writes TokenAnnotation objects

```
public class AnnotationWriter {

    public void writeAnnotations(String fn, List<
        TokenAnnotation> annotations) throws IOException {
        BufferedWriter bw = Files.newBufferedWriter(Paths.get(
            fn), Charset.defaultCharset());
        for (TokenAnnotation a : annotations) {
            bw.write(a.getContent());
        }
        bw.close();
    }
}
```

Class diagram: TokenAnnotation and sub-classes



Advantage of abstract super class

Advantage

- Method that writes an annotation does not have to know which annotation it is dealing with
- Writer method can be implemented at the beginning of the implementation
- Arbitrary annotations can be added easily
- Developer writing the `AnnotationWriter` doesn't need to know anything about the *implementation* of concrete sub-classes

Additional request

Adding parse trees

- You also want to process parse trees
- Parse trees are not token based
- Parse trees have a number of tokens that are spanned by the tree
- Parse trees have a start and an end
- But: positions measured in character positions!
- → we add an alternative super class: `ParseTree`

Inheritance hierarchy

- `ParseTree` as a sub-class of `TokenAnnotation`?
- Not really! A parse tree is not a `TokenAnnotation`!
- → separate inheritance structure

Additional request

Adding parse trees

- You also want to process parse trees
- Parse trees are not token based
- Parse trees have a number of tokens that are spanned by the tree
- Parse trees have a start and an end
- But: positions measured in character positions!
- $\rightarrow$ we add an alternative super class: `ParseTree`

Inheritance hierarchy

- `ParseTree` as a sub-class of `TokenAnnotation`?
- Not really! A parse tree is not a `TokenAnnotation`!
- $\rightarrow$ separate inheritance structure

Additional request

Adding parse trees

- You also want to process parse trees
- Parse trees are not token based
- Parse trees have a number of tokens that are spanned by the tree
- Parse trees have a start and an end
- But: positions measured in character positions!
- → we add an alternative super class: `ParseTree`

Inheritance hierarchy

- `ParseTree` as a sub-class of `TokenAnnotation`?
- Not really! A parse tree is not a `TokenAnnotation`!
- → separate inheritance structure

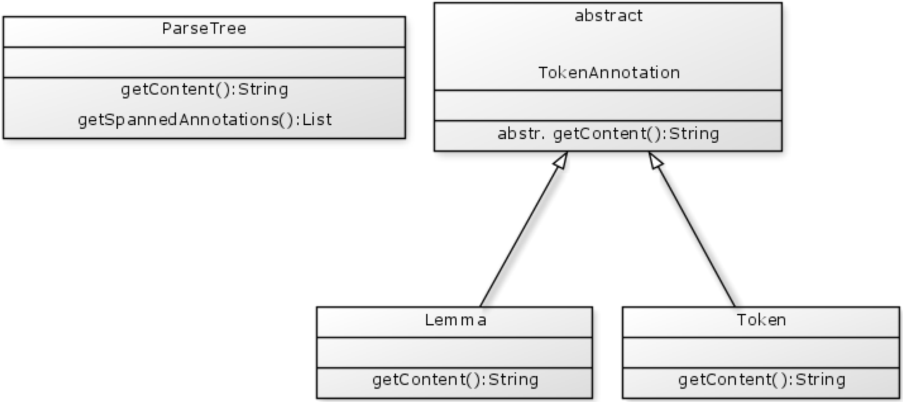
Modeling a parse tree

```java
public class ParseTree {
    // here: _character_ positions
    int start, end;
    List<TokenAnnotation> spannedAnnotations;

    public ParseTree(int start, int end) {
        this.start = start;
        this.end = end;
    }
    public String getContent() {
        // some implementation...
    }
    public List<TokenAnnotation> getSpannedAnnotations() {
        return spannedAnnotations;
    }
}
```

New class structure

- ParseTree completely separate from TokenAnnotation



Reminder: writer class

```
...
    public void writeAnnotations(String fn, List<
        TokenAnnotation> annotations) throws IOException {
        BufferedWriter bw = Files.newBufferedWriter(Paths.get(
            fn), Charset.defaultCharset());
        for (TokenAnnotation a : annotations) {
            bw.write(a.getContent());
        }
        bw.close();
    }
```

- ParseTree is not a TokenAnnotation
- → We cannot write parse trees!

Writer class too concrete

Writing parse trees

- We implemented the writer class to accept each `TokenAnnotation`
- `ParseTree` is not a `TokenAnnotation`
- In `writeAnnotations`, we only access the `getContent` method of `TokenAnnotation`
- `ParseTree` provides the same method
- We need to define that the method can handle all classes that have a `getContent` method!
- $\rightarrow$ We define an interface: `Writable`!

Writer class too concrete

Writing parse trees

- We implemented the writer class to accept each `TokenAnnotation`
- `ParseTree` is not a `TokenAnnotation`
- In `writeAnnotations`, we only access the `getContent` method of `TokenAnnotation`
- `ParseTree` provides the same method
- We need to define that the method can handle all classes that have a `getContent` method!
- → We define an interface: `Writable`!

Writer class too concrete

Writing parse trees

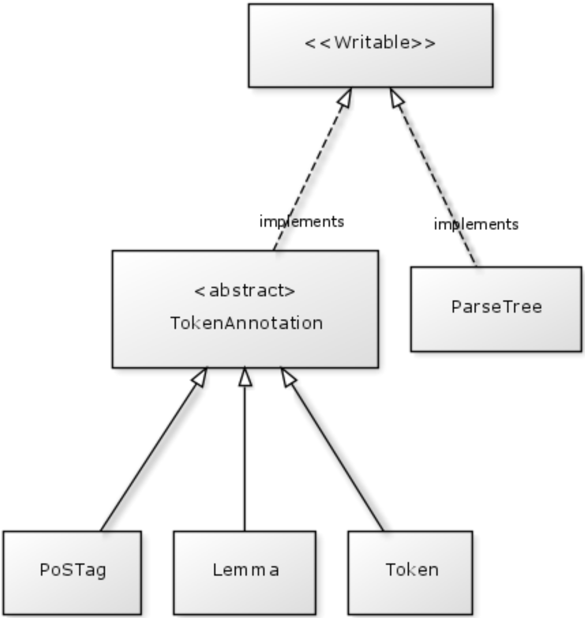
- We implemented the writer class to accept each `TokenAnnotation`
- `ParseTree` is not a `TokenAnnotation`
- In `writeAnnotations`, we only access the `getContent` method of `TokenAnnotation`
- `ParseTree` provides the same method
- We need to define that the method can handle all classes that have a `getContent` method!
- $\rightarrow$ We define an interface: `Writable`!

Writable interface

Simple interface for writable classes

```
public interface Writable {
    public String getContent();
}
```

Class diagram: Writable interface & TokenAnnotation



Implementing the `Writable` interface

- `TokenAnnotation` and `ParseTree` need to implement `Writable`

TokenAnnotation

```
public abstract class TokenAnnotation implements Writable {
    ... }
```

→ No change required (`TokenAnnotation` already implements the `getContent` method)

ParseTree

```
public class ParseTree implements Writable {    ...    }
```

→ No change required (`ParseTree` already implements the `getContent` method)

Applying Writable interface to writer class

- Now, both TokenAnnotation and ParseTree implement Writable
- Both classes (and sub-classes thereof) have a getContent method

```
public void writeAnnotations(String fn, List<
    TokenAnnotation> annotations) throws IOException {
    BufferedWriter bw = Files.newBufferedWriter(Paths.get(
        fn), Charset.defaultCharset());
    for (TokenAnnotation a : annotations) {
        bw.write(a.getContent());
    }
    bw.close();
}
```

→ How can we change this method to accept both classes?

Applying `Writable` interface to writer class

- Now, both `TokenAnnotation` and `ParseTree` implement `Writable`
- Both classes (and sub-classes therof) have a `getContent` method

```
public void writeAnnotations(String fn, List<Writable>
    annotations) throws IOException {
    BufferedWriter bw = Files.newBufferedWriter(Paths.get(
        fn), Charset.defaultCharset());
    for (Writable a : annotations) {
        bw.write(a.getContent());
    }
    bw.close();
}
```

We just use `Writable` **instead of** `TokenAnnotation`**!**

Summary I

Abstract classes

- Begin implementation with most abstract class possible that contains all functionality each subclass should have (`TokenAnnotation`)
- Implement methods that are identical for each sub-class (e.g. getter, setter)
- Mark all other methods as `abstract` methods
- Exploit polymorphism *wherever possible*

Interfaces

- Combine two class hierarchies
- Specify “contract” that defines that all classes have particular methods
- Use interfaces as types (polymorphism) wherever possible

Summary II

Polymorphism

- Always use most abstract type possible
- Advantage: methods etc. can be applied to **all sub-classes**
- Disadvantage: loss of specificity
  $\rightarrow$ special behavior of concrete sub-classes not accessible
- Exception: if a method is overwritten, the most specific method is called (dynamic method lookup)

Outline

Traversing collections

A) Traversing collections with `for-each`

```
for (Object o : collection)
    System.out.println(o);
```

B) Using `Iterators`

- Iterators allow traversing trough collections
- Each collection provides an iterator with the `.iterator()` method

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

- `Iterator.remove()`: *modify the collection* ***during iteration***

Iterator example: filtering a list

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

- Works for **any** Collection

Outline

1. Recap
2. Sorting
   - Sorting Collections
   - Sorted Collections
3. Summary

Sorting Collections

Simple case

- `Collections.sort(l)` (where `l` is a `List`, for instance)
- Natural ordering of elements (works for all standard Java data types out of the box)
- In order to sort a Collection, its elements need to `implement Comparable`
- Overview of classes implementing `Comparable`: `http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html`

Writing Comparable types (classes)

Comparable interface

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

- In order to sort collections with your own classes, you have to implement `Comparable`!

`compareTo` method

- Compares the object with another object (o)
- **returns negative int**, if o is less than the object for which the method is called
- **returns 0**, if both objects are equal
- **returns positive int**, if o is greater

Simple example: comparing Names

```
public class Name implements Comparable<Name> {
    private String firstName;
    private String lastName;

    public Name(String first, String last) {
        this.firstName = first;
        this.lastName = last;
     }

    public int compareTo(Name o) {
        int lastComp = this.lastName.compareTo(o.lastName);
        if (lastComp == 0) {
             return this.firstName.compareTo(o.firstName);
        }
        return 0;
    }
}
```

Comparing Persons

```
public class Person implements Comparable<Person> {
    private Name name;
    private int birthYear;

    public Person(String firstN, String lastN, int birthY) {
        this.name = new Name(firstN, lastN);
        this.birthYear = birthY;
    }

    public int compareTo(Person arg0) {
        int nameComp = this.name.compareTo(arg0.name);
        if (nameComp == 0) {
            return arg0.birthYear - this.birthYear;
        }
        return nameComp;
    }
}
```

equals, hashCode, compareTo

- You (almost always) want to override all three of them
- Hashcode contract: two `equal` objects have the same hash code
- `equals()` should return `true` under the same conditions that `compareTo` return `0`

Example for Name

```
public class Name implements Comparable<Name> {
  ...
    public boolean equals(Object o) {
        Name no = (Name) o;
        return (no.firstName.equals(this.firstName) &&
                no.lastName.equals(this.lastName));
    }

    public int hashCode() {
        return (this.firstName + this.lastName).hashCode();
    }

    public int compareTo(Name o) {
        int lastComp = this.lastName.compareTo(o.lastName);
        if (lastComp == 0) {
            return this.firstName.compareTo(o.firstName);
        }
        return lastComp;
    }
}
```

How to compare objects

- Begin with comparing most specific information
- Proceed with comparing all remaining properties of the object
- Delegate comparisons to `compareTo` methods of single components

Comparator

- Default ordering: natural order
- Different behavior: you need a `Comparator`
- Class that compares two elements of the same type

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Example: Person Comparator

- Normally, sorting persons by their name first is ok
- One scenario: we want to sort them by birthyear for a company anniversary

```java
import java.util.Comparator;

public class YearFirstPersonComp implements Comparator<Person>
    {

    public int compare(Person arg0, Person arg1) {
        // sort persons by their birthyear
        return (arg0.getBirthYear() - arg1.getBirthYear());
    }

}
```

Sorting a list of Persons

```java
Collections.sort(personList, new YearFirstPersonComp());
```

Outline

Sorted collections I

`SortedSet` interface

- `head/tailSet(E e)` returns sub-sets of elements less/greater than `e`
- `subSet(E from, E to)` returns a sub-set with values between `from` and `to`
- `first/last()` retrieves first/last element
- Concrete implementation: `TreeSet`
- All elements in a sorted set need to implement `Comparable`
- Optional comparator can be specified to adjust ordering strategy
- Constructors:
  - `TreeSet()`
  - `TreeSet(Comparator comp)`
  - `...`

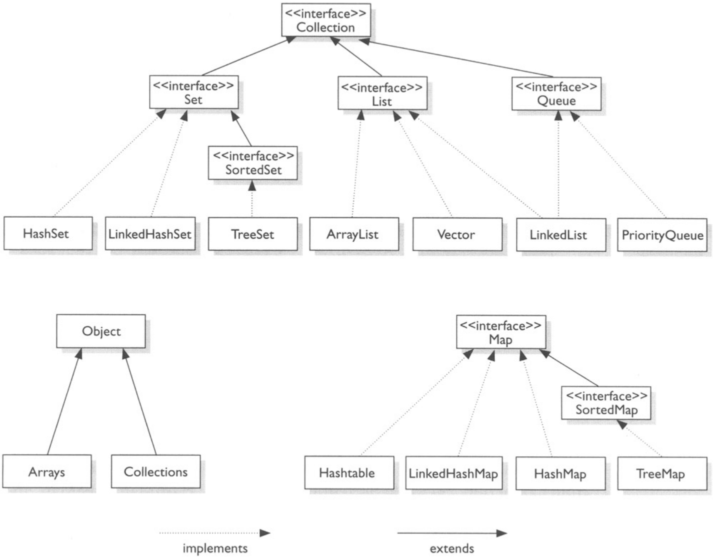
Sorted collections II

`SortedMap` interface

- Keys are ordered
- Concrete implementation: `TreeMap`
- Methods similar to `SortedSet`
  - `firstKey()`
  - `subMap(K from, K to)`
  - ...

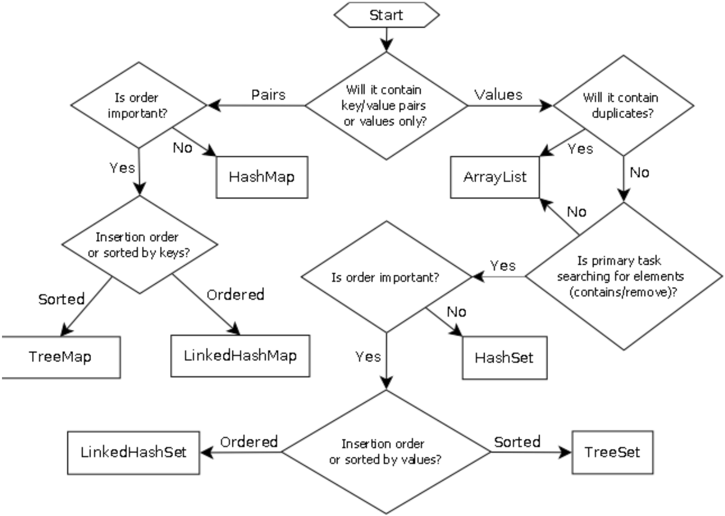
Outline

Overview: collection hierarchy



source: `collectionsjava.blogspot.de`

Choosing the right collection



source: `www.sergiy.ca/guide-to-selecting-appropriate-map-collection-in-java`

Summary

Collections

- `Collection` framework contains multiple classes to conveniently store collections of objects
- Ordered (insertion-order) collections with duplicates: `List` (e.g. `ArrayList`, `LinkedList`)
- Sets of elements without duplicates and no ordering: `Set` (e.g. `HashSet`)
- Sets of elements without duplicates and ordering: `SortedSet` (e.g. `TreeSet`)
- Mapping from keys to values: `Map` (e.g. `HashMap`, `TreeMap`)

Exercises

Source: `http://docs.oracle.com/javase/tutorial/collections/interfaces/QandE/questions.html`

Which collection would you choose?

- Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.
- WTI has decided that each new product will be named after an employee – but only first names will be used, and each name will be used only once. Prepare a list of unique first names.
- WTI decides that it only wants to use the most popular names for its toys. Count the number of employees who have each first name.
- WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.

Literature

Java 7 API
http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

Sierra, K. & Bates, B.
Head First Java. (Chapter 14)
O'Reilly Media, 2005.

Ullenboom, Ch.
Java ist auch eine Insel. (Chapter 13)
Galileo Computing, 2012.