Programmieren II

Collections

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from T. Bögel)

May 28, 2014

Outline

Outline

Outline

The Path class

java.nio.file.Path

- Primary entry point for file I/O
- Programmatic representation of a path in the file system
- Path reflects underlying OS

Obtaining a Path object

- Helper class: Paths
- `Path p1 = Paths.get("/tmp/foo");`
- `Path p2 = Paths.get("/home/root","Documents","world_formula.txt");`
- `Path p5 = Paths.get(System.getProperty("user.home"),"logs", "foo.log");`

A selection of `Path` methods

`Path` methods

- `toString`: returns a string representation of the `Path`
- `getFileName`: returns the last element of the sequence of name elements
- `getParent`: returns the path of the parent directory

Joining two paths

- Paths can be joined with the `resolve` method
- Partial path given as a parameter is appended to original path

Example

```
Path p1 = Paths.get("/home/joe/foo");
// Result is /home/joe/foo/bar
System.out.format("%s%n", p1.resolve("bar"));
```

Additional methods

Check additional methods in the official Java API:
`http://docs.oracle.com/javase/7/docs/api/`

Files methods

- Class `Files` provides many methods
- Read the Java API to get an overview: `http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html`
- Examples:
  - `exists(Path p)`
  - `isReadable(Path p)`
  - `isDirectory(Path p)`
  - `delete(Path p)`

Creating temporary files

Temporary files

- Platform-specific creation of a temp file
- Two methods:
  - `createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)`
  - `createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`

Example: creating a temp file

```
try {
    Path tempFile = Files.createTempFile(null, ".tmp");
    System.out.format("The temporary file" +
        " has been created: %s%n", tempFile);
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Buffered I/O I

Conveniently reading a file

- Convenience method:
  `Files.newBufferedReader(Path, Charset)`
- Opens a file for reading
- Returns a `BufferedReader`
- Similar for `BufferedWriter`:
  `Files.newBufferedWriter(Path, Charset, OpenOption...)`

Buffered I/O II

Example

```
Charset charset = Charset.defaultCharset();
try (BufferedReader reader = Files.newBufferedReader(file,
    charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Outline

Exceptions: general

- Exceptions define behavior in case of an error
- If error occurs: exception (object) is thrown and needs to be handled

Unchecked exceptions

- Mostly programming errors
- Do not have to be handled $\rightarrow$ program crash

Checked exceptions

- Methods specify that they **can** throw a checked exception
- **If** the exception occurs, `catch` blocks define error behavior
- If the exception is not thrown: code runs normally
- For each (possible) checked exception, there **must** be exception handling
- **The compiler will tell you if you need to handle an exception**

Exception handling I

`try-catch`

- Code that could potentially throw an exception: `try` block
- Code that handles the exception type: `catch` block
- If no exception occurs: `try` block is executed normally
- If exception occurs: remaining `try` statemens are skiped, `catch` is executed
- (Optional) `finally` block: code is *always* executed

Exception handling II

`throws` declaration

Alternative to `try-catch`

- Delegate exception to calling methods
- Add `throws ExceptionType` to method signature
- E.g. `public void riskyMethod() throws FileNotFoundException`
- If exception occurs: it is automatically passed to the calling method
- Method that calls `riskyMethod` must handle checked exception

Outline

Collections

Collections

- Collection is a container
- Collections group multiple elements into a single unit
- Collections allow storing, retrieving and manipulating data
- Represent data items forming a natural group
- Example: mail folder, telephone directory etc.
- Conceptually similar to Arrays

Collections Framework

Collections framework

- Unified architecture for representing and manipulating collections
- Components of a collection framework:
  - Interfaces: abstract data types representing collections (e.g. `List`)
  - Implementations: concrete implementations of interfaces
  - Algorithms: computations on collections (sorting, searching etc.): polymorphic

Benefits of Java Collections Framework

Benefits

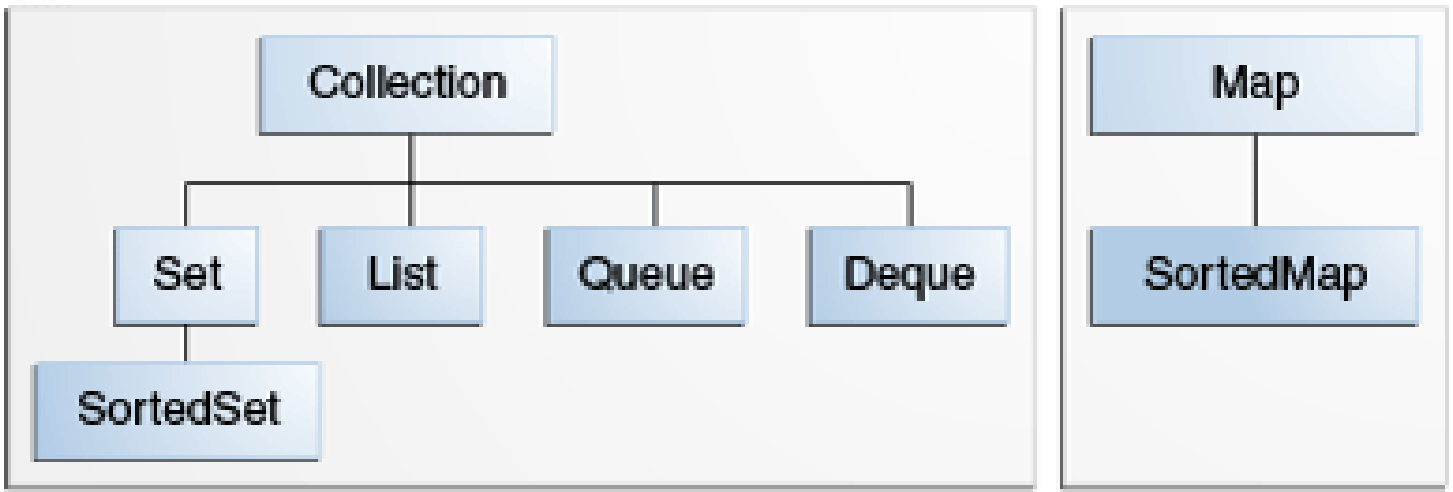
- Reduces programming effort
- Increases program speed and quality: high-performance implementations
- Allows interoperability among unrelated APIs
- Fosters software reuse

Outline

Collection Interfaces

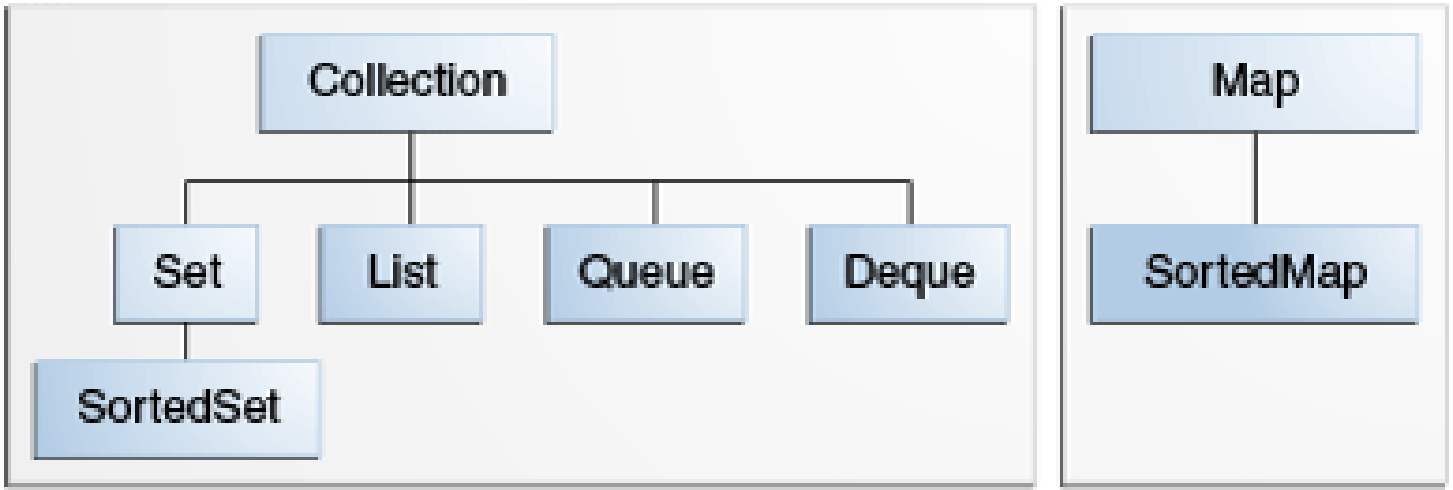
Core Collection Interfaces

- Encapsulate different types of collections
- Manipulation of collections independently of their implementation
- Foundation of Java Collections Framework
- Interfaces form a hierarchy



source: http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

Core Collections hierarchy



source: `http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html`

- `Set` is special kind of `Collection` etc.
- `Map` is separate from `Collection`
- **All** collection interfaces are **generic**: `ArrayList<E> list = new ArrayList<E>;`
- You should specify type of objects within a collection

Description of core collection interfaces

Collection

- Root of collection hierarchy
- Collection contains *elements*

Methods for each `Collection`

- `size(): int` Number of elements in a collection
- `isEmpty(): boolean`
- `contains(Object element): boolean`
- `add(E element): boolean`
- `remove(Object element): boolean`
- `iterator(): Iterator<E>`
- `toArray(): Object[]`
- `equals(): boolean`

Traversing collections

A) Traversing collections with `for-each`

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

B) Using `Iterators`

- Iterators allow traversing through 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

Collection bulk operations

- Methods that operate on *entire* collection

Methods

- `containsAll`: does the collection contain all elements specified in another collection?
- `addAll`: add all elements of one collection to another collection
- `removeAll`
- `clear`: removes all elements from the collection

Outline

Methods for all Lists

Lists

- Ordered collection
- May contain duplicates

Methods for all Lists

- All methods defined in `Collection`
- `void add(int index, Object o)`
- `Object get(int index)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `Object remove(int index)`
- `Object set(int index, Object o)`
- `List subList(int start, int end)`

List implementation: ArrayList

ArrayList

- Similar to static Arrays
- Size of an ArrayList is dynamic
- Based on dynamic arrays that can grow at runtime

Constructor

```
ArrayList()
ArrayList(Collection c)
...
```

ArrayList: Example

```java
package de.uniheidelberg.cl.prog2.collections;

import java.util.ArrayList;

public class ArrayListTest {
  public static void main(String[] args) {
    // create a new array list that contains Strings
    ArrayList<String> list = new ArrayList<>();
    list.add("But you wouldn't listen. No one ever does");
    list.add("Life, don't talk to me about life");

    System.out.format("List contains %s elements.", list.size()
        );
    System.out.format("First element: %s", list.get(0));

    for (String s : list) {
      System.out.println(s);
    }
  }
}
```

code/ArrayListTest.java

Outline

Set interface

Sets

- Set: collection that cannot contain duplicate elements
- Models mathematical sets
- `add(Object obj)` returns `false` if set already contains obj

Set operations

- `s1.addAll(s2)`: s1 = s1 $\cup$ s2
- `s1.retainAll(s2)`: s1 = s1 $\cap$ s2
- `s1.removeAll(s2)`: s1 = s1 $\setminus$ s2
- `s1.containsAll(s2)`: s1 = s1 $\supseteq$ s2

Set implementation: HashSet

HashSet

- Concrete implementation of the `Set` interface
- Efficient access because of hashing
- `int hashCode()` returns unique hash code of an object
- Two identical objects (`equals`) have the same hash code (hash code contract)

Other implementations

- `TreeSet`: ordered set
- `LinkedHashSet`: elements returned in insertion order (initial insertion)

Set example

```
package de.uniheidelberg.cl.prog2.collections;

import java.util.HashSet;

public class HashSetTest {

  public static void main(String[] args) {
    HashSet<Integer> accountNumbersRich = new HashSet<>();
    boolean addSuccessful = accountNumbersRich.add(111111);
    addSuccessful = accountNumbersRich.add(111112);
    addSuccessful = accountNumbersRich.add(111112);
    System.out.format("Elements in set: %s.\nLast addition
        successful: %s", accountNumbersRich.size(),
        addSuccessful);
  }
}
```

code/HashSetTest.java

Outline

Maps

Maps

- Map maps keys to values
- Maps cannot contain duplicate keys
- Key-value mapping models mathematical functions

Examples

- Mapping from words to frequencies in a corpus
- Mapping from countries to capital cities
- Mapping from international calling prefix to country
- ...

Methods of the Map interface

Common methods

- `Object put(Object key, Object value)`
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `Collection values()`
- `Set keySet()`

Concrete Map implementation: HashMap

HashMap

- Concrete implementation of the `Map` interface
- `Map m = new HashMap();`
- Methods: identical to `Map`

Example: using Maps

```java
package de.uniheidelberg.cl.prog2.collections;

import java.util.HashMap;

public class HashMapTest {
  public static void main(String[] args) {
    HashMap<String,Integer> heroesFirstApp = new HashMap<>();
    heroesFirstApp.put("Superman", 1938);
    heroesFirstApp.put("Batman", 1939);

    int batmansFirstApp = heroesFirstApp.get("Batman");
    for (String s : heroesFirstApp.keySet()) {
      System.out.format("Superhero: %s, first appearance: %s\n"
          , s, heroesFirstApp.get(s));
    }

  }
}
```

code/HashMapTest.java

HashMap: counting frequencies

```
package de.uniheidelberg.cl.prog2.collections;
...
public class CountingFreqs {

  public static void main(String[] args) {
    String text = "life don't talk to me about life";
    HashMap<String, Double> wordCount = new HashMap<>();
    for (String s : text.split("\\s+")) {
      Double oldCounter = wordCount.get(s);
      if (oldCounter == null)
        oldCounter = 0.;
      wordCount.put(s, ++oldCounter);
    }
    for (String s : wordCount.keySet()) {
      System.out.format("%s\t%s\n",s, wordCount.get(s));
    }
  }
}
```

code/CountingFreqs.java

Data types in collections

- All elements in a collection should be of the same type
- `Object` collections do not enforce this constraint
- `Object` collection: collection without any type parameter (`<Type>`)
- `Object` collections require type casting
- No security about data type in a collection $\rightarrow$ `ClassCastException`

Parameterised collections

Generics

- Collections can (and should) be parameterised
- Type of elements in a collection specified with angled brackets
- We'll learn about generics later

Example

```
List<String> stringList =
        new ArrayList<String>();
Map<Animal,String> animalNames =
        new HashMap<Animal,String>();
```

- HashMap maps Animals (polymorphism) to Strings

Other Map implementations

- `TreeMap`: sorted keys
- `LinkedHashMap`: keys returned in insertion order (initial insertion)

Wrapper types

- Collections can only contain object references
- No primitive data types
- $\rightarrow$ Wrapper classes for primitive data types
- Examples: `Double`, `Integer` ...
- Java API for numeric wrapper classes: `http://docs.oracle.com/javase/7/docs/api/java/lang/Number.html`
- Disadvantage: memory consumption & performance (but see Trove[1])

[1] `http://trove.starlight-systems.com/`

Summary

Collections I

- `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`)
- Mapping from (unordered) keys to values: `Map` (e.g. `HashMap`)

Collections II (later)

- Sorting collections
- Sorted collections
  - Map with sorted keys: `TreeMap`
  - Sets of elements without duplicates and ordering: `SortedSet` (e.g. `TreeSet`)

Literature

Collections tutorial
`http://docs.oracle.com/javase/tutorial/collections/`

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

Sierra, K. & Bates, B.
Head First Java. (Chapter 16)
O'Reilly Media, 2005.

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