Programming Paradigms

159.272

Collections and Generic Parameter Types

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

1. Java Tutorial, lesson on generic parameter types
   http://docs.oracle.com/javase/tutorial/java/generics/index.html
1. Java Tutorial, collection trail
   http://docs.oracle.com/javase/tutorial/collections/index.html

Overview

- lists
- generic parameter types
- wildcards
- sets
- maps
- hashing
- implementing hashcode
- iterators
- collection utilities and library

Common Datastructures in Java

- Java contains several common data structures in the `java.util` packages
- this package contains interfaces describing abstract data structures, like `List`, `Set` and `Map`, and several implementation classes, like `ArrayList`, `HashSet` and `HashMap`
- there are several popular open source libraries that contain more data structures, including:
    - Apache Commons Collections
    - Google Guava

The Collection Interface

- `java.util.Collection` specifies an interface for all data structures that can contain objects
- this includes methods to add, remove and retrieve objects
- note that `Collection` **does not**:
    - impose a linear order on its elements
    - define how the order of elements is managed
    - define whether duplicate objects (w.r.t. equals) can be elements within the same container

Using Collections

```
Collection collection = ...;
collection.add("first");
collection.add("second");
collection.add("third");
collection.remove("first");
for (Object element:collection) {
      System.out.println(element);
}
collection.size();
```

add objects

remove object

do something for all objects in the container (loop)

check the size of the container

Lists

- lists are special types of containers
- lists **maintain the order** of elements
- i.e., the object added first (second,..), is visited first (second, ..) when iterating over the list
- the list API therefore contains methods to access elements **by position**
- lists **accept duplicates**: the same (as well as equal) objects can be stored multiple times in a list

Lists ctd

- the list API is defined in the interface `java.util.List`
- `java.util.List` extends the interface `java.util.Collection`
- lists are in many situations good replacements for arrays, as they offer more flexibility
- in particular, their size is flexible - they **grow** if more space is needed

Using Lists: Lists are Containers

```
List list = ...;
list.add("first");
list.add("second");
list.add("third");
list.remove("first");
for (Object element:list) {
     System.out.println(element);
}
list.size();
```

add objects

remove object

do something for all objects in the container (loop)

check the size of the list

Using Lists: Access By Position

```
List list = ...;
list.add("first");
list.add("second");
list.add("third", "third
list.get(0);
for (int i=0;i<list.size();i++) {
     System.out.println(list.get(i));
}
```

add objects

add object at a certain position

returns "third"

loop using positions

List Implementations

- `java.util.ArrayList` is based on an internal array that stores objects
- if the maximum capacity of the internal array is reached, the array is replaced by a bigger array and the content is copied ("growing")
- `java.util.LinkedList` is an implementation based on a doubly-linked list
- i.e., entries are wrapped by entry objects that reference their neighbours
- linked lists are fast, but have some memory overhead as for each entry another entry object must be created
- `java.util.Vector` is similar to `java.util.ArrayList`, but thread safe

The Problem With Lists

```
List list = ...;
list.add(new Student(..));
list.add(new Student(..));
for (Object element:list) {
      Student student = (Student)element;
      System.out.println(student.getId());
}
```

The Problem With Lists ctd

- because lists are general purpose containers, elements can be (arbitrary) objects
- this means that when accessing the elements of a list, they are accessed as instances of `Object`
- for instance, `get` has the following signature:
  `Object get(int position)`
- this implies that (potentially unsafe) casts are required!
- from this point of view, arrays are safer, as they can be typed (e.g., `Student[]`)
- what is needed is a language feature to declare "Lists of Students"

Generic Parameter Types

Java supports this, it is possible to declare a list of students using a generic parameter type as **List<Student>**

```
methods in List (selection)

Object get(int position)
boolean add(Object object)
```

```
methods in List<Student> (selection)

Student get(int position)
boolean add(Student object)
```

Iterating Over Generic Lists

```
List students = ..;
for (Object next:students) {
      Student nextStudent = (Student)next;
      ..
}
```

```
List<Student> students = ..;
for (Student nextStudent:students) {
      ..
}
```

Type Inference

```java
List<Student> students =
   new ArrayList<Student>();
```

"traditional" correct syntax

```java
List<Student> students =
   new ArrayList();
```

this will also be compiled, but with a warning: the compiler will insert an unsafe type cast

```java
List<Student> students =
   new ArrayList<>();
```

new syntax from Java 1.7 - the compiler will infer the arguments from the declaration type

Lists and Value Types

- Lists (and similar data structures) can also be used with value types
- in this case the corresponding wrapper type is used when the list is declared
- the compiler applies auto boxing / unboxing to convert value types to reference types and vice versa

Lists and Value Types ctd

```
List<Integer> list =
     new ArrayList<Integer>();
list.add(42);
list.add(43);
int firstElement = list.get(0);
```

the wrapper type is used as generic parameter type

autoboxing: 42 is converted to an instance of Integer

auto unboxing: an instance of Integer is converted to an int value

Generic Parameter Types and Inheritance

```
List<Mammal> mammals = new ArrayList<Zebra>();
```

- assume that `Zebra` is a subclass of `Mammal`
- surprisingly, this is **rejected** by the compiler !
- even if the declared type is changed to `ArrayList<Mammal>`, this is still not compiled
- `List<Mammal>` means a list of exactly the type `Mammal`

Generic Parameter Types and Inheritance ctd

```
List<Mammal> mammals = new ArrayList<Zebra>();
mammals.add(new Lion());
```

- if the compiler allowed this, we could end up in a dangerous situation:
- a lion could get among the zebras !

Wildcards

```
List<?> mammals = new ArrayList<Zebra>();
```

- to deal with this situation, Java allows **wildcards**
- here, ? represents **an unknown type**
- we can now work with the mammals as a list of objects (e.g., to iterate over the list)
- however, **we cannot add elements to the list** (neither zebras nor lions) - as we do not know which type of elements can be added to the list

Covariant Generic Parameter Types

```
List<? extends Mammal> mammals =
    new ArrayList<Zebra>();
```

- a **bounded wildcard** is used to represent an unknown type with some constraints
- here, the unknown type must be a subclass of `Mammal`
- now the list is declared as a list of objects that instantiate `Mammal` or any of its subclasses
- this is called **co-variance** - the inheritance of the generic parameter type follows the same direction as the inheritance of the main type

Contravariant Generic Parameter Types

```
List<? super Mammal> mammals =
   new ArrayList<Object>();
```

- it is also possible to define a **bounded wildcard** with a lower bound
- here, the unknown type must be a supertype (class or interface) of Mammal
- this is called **contra-variance** - the inheritance of the generic parameter type follows the opposite direction as the inheritance of the main type

Declaring Types with Generic Parameters by Example

- the task is to implement a method that searches a list of Mammals for an element that satisfies a certain condition
- the condition is implemented as an interface `Condition` that has only one method: `satifies(Object)` returning a `boolean` (whether the condition is satisfied or not)

Conditions

```
public interface Condition<T> {
     boolean satisfies (T object) ;
}
```

- instances of this interface represent conditions on objects of the type T

Condition Example Implementation

```
public class IsStripy implements Condition<Mammal> {
      @Override
   boolean satisfies (Mammal mammal) {
      returns mammal.hasStripes();
   }
}
```

- note that we have bound the parameter to Mammal in the implements clause

Searching through Lists

```
/**
 * Find the first element in a list matching the
 * condition, or null if no element matches.
 */
public Mammal findMammal(List<Mammal> list,
   Condition <? super Mammal> condition) {
      for (Mammal mammal:list) {
            if (condition.satisfies(mammal)) {
            return mammal;
         }
      }
      return null;
   }
```

note the use of contravariance here: if the condition can be applied to supertypes of Mammal (like Object), it can also be applied to Mammal

Sets

- sets are another type of collection similar to lists
- sets are classes implementing the `java.util.Set` interface
- sets do not rely on / maintain an internal order of their elements
- sets **do not accept duplicates**: if two objects obj1 and obj2 are equal (**identity is not required**), then only one can be added to the list
- lists are optimised for fast lookup: the `boolean contain(Object)` method is fast, O(1) (constant time) for HashSets !

Set Implementation Classes

- `java.util.HashSet` is based on hash maps, very fast lookup/add/remove speed (O(1))
- `java.util.TreeSet` sorts its elements, an optional `Comparator` can be passed to the constructor to define how to sort elements, fast lookup/add/remove speed (O(log(n)))
- when iterating over a `TreeSet`, the elements are returned in sort order
- `java.util.LinkedHashSet` is combination of a hash set and a doubly-linked list
- linked hash sets behave like lists, but when iterating, they return elements in the order in which elements were inserted into the set

Maps

- maps are used to store key-value associations between objects
- therefore, maps have two generic parameter types, one for keys, one for values
- example: a map to manage Student instances by id (assume that the id type is `String`):
  `Map<String,Student>`
- the keys in a map are a set, i.e. **duplicate keys** (w.r.t equals) **are not allowed**

Using Maps

```
Map<String,Student> map = ...;
map.put("0156373",new Student(..));
map.put("0156374",new Student(..));
for (String id:map.keySet()) {
     Student student = map.get(id);
     System.out.println(id + ": " +
student);
}
```

- the two most important map API methods are:
  - `put` to add an association
  - `get` to retrieve a value by key

Map Implementation Classes

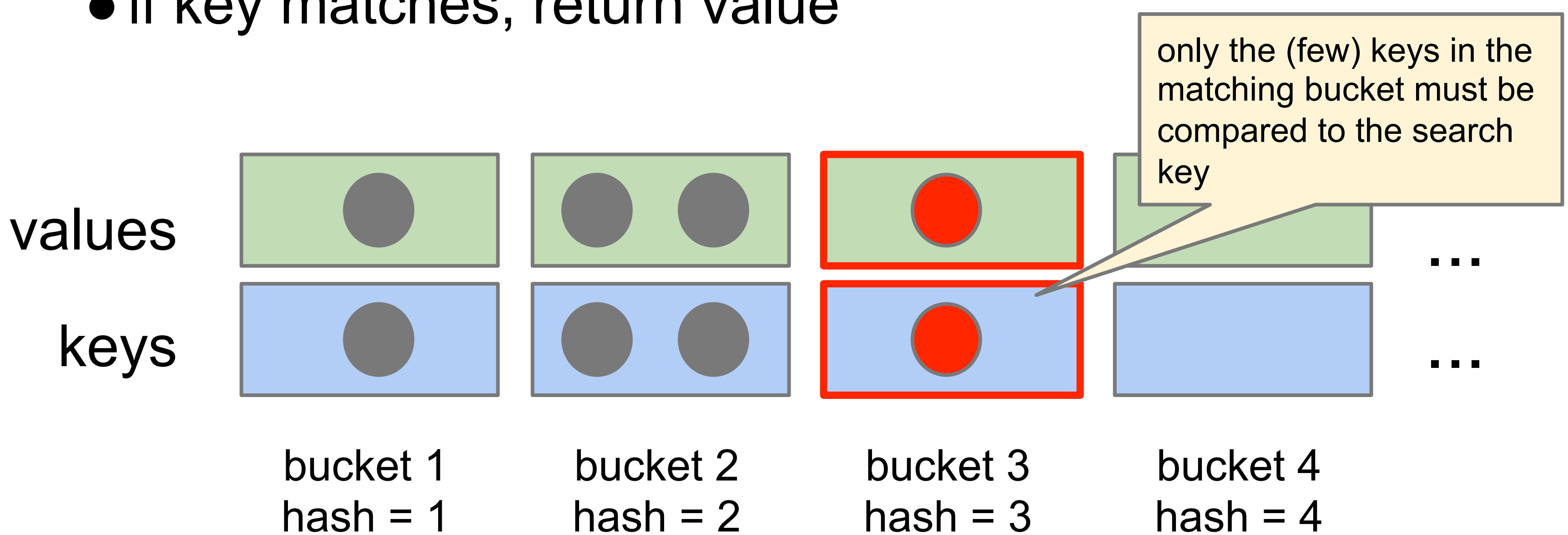
- the standard implementation classes of Map are similar to the implementation classes for Set
- `java.util.HashMap` uses hashing, and has very fast (constant time O(1)) performance for put/remove/get
- `java.util.TreeMap` sorts entries by key, and has fast (log(n)) performance for put/remove/get
- `java.util.LinkedHashMap` is a hash map with an additional list to keep track of the insertion order of entries
- `java.util.WeakHashMap` is a map implementation that uses weak references that are ignored by the garbage collector

Hashing

- the idea of hashing it to divide the internal storage of a map into **buckets**
- it is inexpensive to calculate an integer value (**hash code**) for objects
- this hash code can be used to find a bucket for this object
- finding a bucket can run in constant time (binary search, int numbers have fixed 32 bits!)
- then for lookups (get) only the keys in a bucket must be compared with the search key
- this comparison is using `equals()`, not `==`

Hashing (ctd)

- lookup: `map.get(aKey)`
- compute hash code of `aKey`
- find bucket for this hashcode
- compare object with keys in this bucket (equals)
- if key matches, return value



Hash Codes

- the hash code is computed by the objects themselves!
- maps ask object for it: `hashCode()` is supported by all classes as it is implemented in `Object`
- hashCode should be overridden in subclasses
- hashCodes must be **consistent with equals** for maps to work: otherwise we may not be able to retrieve associations stored in the map!

Map Lookup

```
Map<String,Student> map = new HashMap<>();
String key =  "0156373";
Student value = new Student(..);
map.put(key,value);
...
String lookupKey = "0156373";
Student student = map.get(lookupKey);
```

- assume that `key!=lookupKey`
- but of course, `key.equals(lookupKey)` yields true!
- should this return the `Student` instances added to the map?

Map Lookup

```
Map<String,Student> map = new HashMap<>();
String key =  "0156373";
Student value = new Student(..);
map.put(key,value);
...
String lookupKey = "0156373";
Student student = map.get(lookupKey);
```

- the lookup will succeed as expected: i.e., the `Student` instance added before will be retrieved
- equality, not identity of keys is required to lookup a map entry

equals and hashCode

- this implies the following rule:
- **if two objects are equal, then they have to have the same hash code**
- if this rule is violated, the wrong buckets are checked, and data structure like `HashMap` and `HashSet` will exhibit unexpected behaviour such as:
  - lookups fail when they should succeed
  - duplicate keys can be stored
- this is an example of a **(semantic) contract** between two methods
- an easy way to ensure is to use a code generator, and generate consistent code for both `equals()` and `hashCode()` at the same time
  Eclipse: Source > Generate hashCode() and equals()

Good Hash Codes

- if two objects are not equal, but have the same hash codes, a **collision** occurs
- for a good hashcode, the buckets are as small as possible: only few collisions occur
- if the bucket size is 1, the hash code rule is reversed: if two objects are not equal, then they have different hash codes - this means that there are no collisions

Secure Hash Codes

- in security, hash codes are used for signatures and verification
- many authentication systems do not store passwords but the secure hashes of passwords
- a good secure hash code function is a function that cannot be reversed: it is computationally expensive (and therefore practically impossible) to find an object obj such that hashCode(obj) yields a given hash code X
- Examples for secure hash code functions are MD5 and SHA1

Testing and Benchmarking Hash Codes: Point2D

```
public class Point2D {
      protected int y;
      protected int x;
      public Point2D(int x, int y) {
            super();
            this.x = x;
            this.y = y;
      }
      ..
}
```

- simple class, state consists of two int values
- code: https://oop-examples.googlecode.com/svn/semantics/

Testing and Benchmarking Hash Codes: Point2D.equals

```
@Override
public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        final Point2D other = (Point2D) obj;
        if (x != other.x) return false;
        if (y != other.y) return false;
        return true;
}
```

- straightforward, Eclipse-generated implementation
- points are equal if both coordinates x and y have the same values

Testing and Benchmarking Hash Codes: Implementing Hash Codes

- test different implementations of `hashCode()`
- to do this, implement different subclasses of `Point2D`, and override `hashCode()` in these subclasses

Testing and Benchmarking Hash Codes: Point2D_1

```
@Override
public int hashCode() {
     return System.identityHashCode(this);
}
```

- `System.identityHashCode()` returns an int the system has computed from the memory location of an object
- i.e., different objects (even when equal) will have different caches
- i.e., the contract between `equals` and `hashCode` is violated !

Testing and Benchmarking Hash Codes: Contract Violation

```
Map<Point2D_1,String> map =
   new HashMap<Point2D_1,String>();
Point2D_1 key1 = new Point2D_1(42,42);
Point2D_1 key2 = new Point2D_1(42,42);
map.put(key1,"test");
map.get(key2);
```

- this returns `null` - the lookup fails although the keys are equal !

Testing and Benchmarking Hash Codes: Contract Violation

```
Map<Point2D_1,String> map =
   new HashMap<Point2D_1,String>();
Point2D_1 key1 = new Point2D_1(42,42);
Point2D_1 key2 = new Point2D_1(42,42);
map.put(key1,"test");
map.get(key2);
```

Testing and Benchmarking Hash Codes: Point2D_2

```
@Override
public int hashCode() {
	return x%10;
}
```

Testing and Benchmarking Hash Codes: Point2D_2

```
@Override
public int hashCode() {
     return x%10;
}
```

- this is **correct** with respect to the contract
- equal objects have the same x and y values, and therefore the same hash
- but there are only 10 different hash keys (x%10 means "x modulo 10")!
- i.e., the bucket size is large, and performance is O(n)

Testing and Benchmarking Hash Codes: Point2D_3

```
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
```

- this is `hashCode()` generated by Eclipse
- it uses both x and y, and uses bit shift to increase the probability that unique values will be created
- this means that bucket sizes stay small, and lookup and insert performance is close to constant time O(1)

Testing and Benchmarking Hash Codes: Benchmark Setup

- set problem size MAX
- create MAX x MAX points
- insert an association with a string for each point in a double loop (iterate over x, then y)
- then make MAX x MAX lookups in a second double loop
- measure time using `System.currentTimeMillis()`
- code: https://oop-examples.googlecode.com/svn/semantics/

Testing and Benchmarking Hash Codes: Benchmark Results

	Point2D_2	**Point2D_3**
insert 25000 associations	22,015 ms	448 ms
lookup 25000 associations	19,824 ms	33 ms
insert 1,000,000 associations	787,580 ms = **13.2 min**	1,482 ms
lookup 1,000,000 associations	704,581 ms = **11.7 min**	193 ms

- for experiments with 1,000,000 objects, memory space was increased to 1GB by starting the JVM with -Xmx1g
- System used: PowerMac with JDK 1.7.0_07-b10 on Mountain Lion, 2.8 GHz Intel Core 2 Duo

Loops Revisited: Arrays

```
String[] array = ..;
for (int i=0;i<array.length;i++) {
      String next = array[i];
      // do something with next
}
```

- this is a classical (C-style) loop over an array
- the loop is based on iterating over the positions within the array

Loops Revisited: Lists

```
List<String> list = ..;
for (int i=0;i<list.size();i++) {
      String next = list.get(i);
      // do something with next
}
```

- lists also organise content by index (position)
- this means that it is possible to iterate (loop) over all elements of a list in "array style"

Iterating Over Sets

- how can we iterate over sets, i.e. data structures that do not organise content by index?
- the approach taken is to use an **iterator**, a kind of object that can be used to iterate over elements in collections and similar data structures
- iterators are a classical **design patterns**, and are available in all mainstream programming languages

The Iterator API

`java.util.Iterator<T>` is a generic interface that defines the following methods:

```
// are there more elements to come ?
boolean hasNext();

// return the next element
T next()

// remove the current element
void remove()
```

The Iterator API ctd

- `hasNext()` and `next()` are sufficient to build loops
- all collections (instances of Collection, including sets) support a method `iterator()` that returns an iterator
- `Collection` extends the interface `java.lang.Iterable` that defines `iterator()`
- `remove()` is an **optional method**
- this means that every class implementing `Iterator` must implement this method, but may choose to implement it by throwing an `java.lang.UnsupportedOperationException`
- the purpose of remove is to remove elements from a collection from within the loop that detects the elements to be removed

Looping with Iterators

```
Collection<String> collection = ...;
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
      String next = iterator.next();
      // do something with next
}
```

this could be a list, set or any other collection !

Implementing Iterators

- the collection classes all have their own iterator implementations that are usually invisible to programmers
- this is an example of the advantage we get from abstract type: by calling `collection.iterator()`, we only see the interface, and the implementation complexity is completely hidden!
- implementations usually move some sort of cursor through an internal representation of elements
- `hasNext()` - whether the cursor has not reached the end
- `next()` - move the cursor forward, and return the next element

Problems with Iterators

```
Collection<String> collection = new ArrayList();
collection.add("one");
collection.add("two");
collection.add("three");
Iterator<String> iterator =
collection.iterator();
while (iterator.hasNext()) {
      String next1 = iterator.next();
   String next2 = iterator.next();
}
```

Problems with Iterators

```
Collection<String> collection = new ArrayList();
collection.add("one");
collection.add("two");
collection.add("three");
Iterator<String> iterator =
collection.iterator();
while (iterator.hasNext()) {
      String next1 = iterator.next();
   String next2 = iterator.next();
}
```

- this will fail with a RuntimeException (more precisely, a `java.lang.NoSuchElementException`)
- in the second iteration, there is only one element left, so the second call to `next()` will fail - there is no next element
- rule: `hasNext()` only guards one call to `next()` !

Enhanced for Loops

```
Collection<String> collection = ...;
for (String next:collection) {
      // do something with next
}
```

this is "syntactic sugar" - translated by the compiler to:

```
Collection<String> collection = ...;
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
      String next = iterator.next();
      // do something with next
}
```

Enhanced for Loops ctd

- enhanced for loops work for all iterables, not only collections
- enhanced for loops can also be used with arrays

Iterators ctd

- iterators cannot only be used to iterate over "extensional" collections where all elements are present, but also over "intensional" collections where elements are computed on demand (when `next()` or `hasNext()` are called)
- an example for such an intentional iterator can be found in the database API (`java.sql.ResultSet`)
- external data structure libraries like Google Guava have extended support for iterators, including utilities for tasks such as filtering, searching, mapping and chaining iterators: `com.google.common.collect.Iterators`

More Collections and Utilities

- `java.util.Collections` has many useful static methods to sort and search collections, to create unmodifiable wrappers etc
- `java.util` contains several standard data structures not discussed here, including `Stack` and `Queue`
- `java.util.concurrent` contains several collections optimised for multithreaded Java programs
- there are several open source collection libraries that with more data types and utilities, such as:
  - Google Guava
  - Apache Commons Collections