Generics, Collections & Iterators

Lecture 5 (8 March 2022)

Sjaak Smetsers

Software development

goal

- correct, robust, adaptable/extendable software
- fast development, reuse existing parts/libraries

OO tools/principles

- *encapsulation*: information hiding
- *realization*: `implements`
- *composition*: has-a
- *inheritance*: is-a, `extends`
- *polymorphism*: overriding

Java additions

- strong static typing: spot all type problems at compile time
- exception mechanism to handle runtime errors
- generics: increase reusability and type safety by *type parameters*

Generics

The design of `equals` (1)

== operator checks equality of object-locations (pointer comparison)

the default `Object equals` method does the same:

```
public boolean equals (Object o) {
  return this == o;
}
```

typically you want to check equality of (some) fields

- override the `equals` for your class

desirable property: symmetry

- `x.equals(y)` ⇔ `y.equals(x)`

The design of equals using `getClass`

```java
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object objQ) {
    if (objQ == null || getClass() != objQ.getClass()) {
      return false;
    } else {
      final Person q = (Person) objQ;
      return name.equals(q.name);
    }
  }
}
```

```java
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```java
public class Student extends Person {
  private int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object objS) {
    if (objS == null || getClass() != objS.getClass()) {
      return false;
    } else {
      final Student other = (Student) objS;
      return num == other.num;
    }
  }
}
```

RUN

```
p.equals(s) = false
s.equals(p) = false
```

The design of equals using `instanceof`

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object objQ) {
    if ( objQ instanceof Person ) {
      final Person q = (Person) objQ;
      return name.equals(q.name);
    } else {
       return false;
    }
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
public class Student extends Person {
  private int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object objS) {
    if ( objS instanceof Student ) {
      final Student s = (Student) objS;
      return num.equals(s.num);
    } else {
       return false;
    }
  }
}
```

violates x.equals(y) ⇔ y.equals(x)

RUN

```
p.equals(s) = true
s.equals(p) = false
```

Design of `compareTo`

```
interface Comparable {
  int compareTo(Object o);
}
```

Note: this is not the definition of `Comparable` in the standard Java API

desirable property: *inversion*
`sign(x.compareTo(y)) == -sign(y.compareTo(x))`

for **`Person`**, in the style of **`equals`**

```
  public int compareTo(Object objP) {
    if ( objP instanceof Person ) {
      final Person p = (Person) objP;
      return name.compareTo(p.name);
    } else {
      return ??;
    }
  }
```

any integer is wrong!
1) use another result type
2) throw an exception
3) avoid wrong type of argument

A type-safe compare

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

<T>: *formal generic type*

using the formal generic type

generic instantiation: actual concrete type

```
public class Person implements Comparable<Person> {
  private String name;
  public Person(String name) {
    this.name = name;
  }

  @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

RUN

```
p.compareTo(s) = 0
s.compareTo(p) = 0
```

A type-safe compare (2)

```
public class Student extends Person {
  …
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

```
public class Person implements Comparable<Person> {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

Compile-time error: “method does not override or implement a method from a supertype”

`implements Comparable<Student>` also not allowed

```
public class Student extends Person implements Comparable<Student> {
  …
}
```

`equals` with different types

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object objP) {
    if (objP == null || getClass() != objP.getClass()) {
      return false;
    } else {
      final Person p = (Person) objP;
      return name.equals(p.name);
    }
  }
 @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

```
Person p = new Person("Alice");
p.equals("Alice");
```

a) type error

b) true

c) false

`compareTo` with different types

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object objP) {
    if (objP == null || getClass() != objP.getClass()) {
      return false;
    } else {
      final Person p = (Person) objP;
      return name.equals(p.name);
    }
  }
 @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

```
Person p = new Person("Alice");
p.compareTo("Alice");
```

a) type error

b) 0

c) an integer ≠ 0

equals with null

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object objP) {
    if (objP == null || getClass() != objP.getClass()) {
      return false;
    } else {
      final Person p = (Person) objP;
      return name.equals(p.name);
    }
  }
 @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

```
Person p = new Person("Alice");
p.equals(null);
```

a) type error

b) exception

c) true

d) false

compareTo with null

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object objP) {
    if (objP == null || getClass() != objP.getClass()) {
      return false;
    } else {
      final Person p = (Person) objP;
      return name.equals(p.name);
    }
  }
 @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

```
Person p = new Person("Alice");
p.compareTo(null);
```

a) type error

b) exception

c) 0

d) an integer ≠ 0

more uses of generic types: counting word frequencies

```
To be or not to be - that is the question!
```

RUN

```
be 2
is 1
not 1
or 1
question 1
that 1
the 1
to 2
```

Store each word and its count in array of pairs

1. update array for each word in input
2. sort array (lexicographically)
3. show array

make this map of words to count more reusable

- many programs need pairs
  - often other types than String and int
  - using Object instead of String and int spoils type safety
- many programs need a Map
  - not restricted to Map from String to int
  - String to double in expressions
  - StudentNumber to Student
  - Zipcode to Address
- in Java we can make this more reusable by introducing type variables: generic programming
  - available since 2004 in JSE 5.0, SDK 1.5

reusable pair

```
public class Pair<K, V> {
  private K key;
  private V val;

  public Pair(K key, V val) {
    this.key = key;
    this.val = val;
  }

  public K getKey() { return key; }
  public V getVal() { return val; }
  public void setVal(V val) { this.val = val; }
}
```

K and V are generic type variables
typically a single uppercase letter

K and V are used like a type: field

K and V are used like a type: argument of method

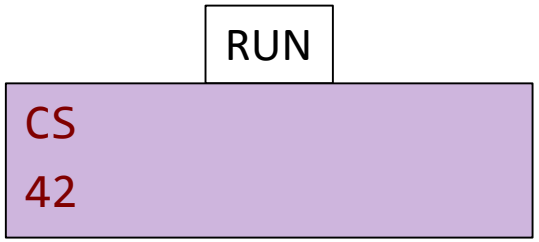
K and V are used like a type: result of method

allowed instances of generic type variable: any reference type

<>: *diamond operator*
instructs the compiler to deduce types automatically

```
private static void run() {
  Pair<String,Student> pss = new Pair<>("CS",new Student(42,"Alice"));

  System.out.println(pss.getKey());
  System.out.println(pss.getVal().getNum());
}
```



allowed instances: what about primitive types?

- `Pair<int, Student> p3 = new Pair<>(42, alice);`

this is **NOT** allowed !

solution: use *wrapper types*

- these are predefined in Java:

  `int`, `double`, `char`, `boolean` wrapped in
  `Integer`, `Double`, `Character`, `Boolean`

use this as

```
Pair<Integer, Student> p3 = new Pair<>(42, alice);
```

autoboxing / auto-unboxing: automatic conversion between primitive & wrapper

```
Integer box = 7;
int plain   = box;
```

instead of

```
Integer box = new Integer(7);
int plain   = box.intValue();
```

Mapping words to counts (1)

```
public class CountMap {
  private Pair<String,Integer>[] map;
  private int nrElems = 0;

  public CountMap(int maxSize) {
    map = new Pair[maxSize];
  }

  public void add(String key) {
    for (int i = 0; i < nrElems; i++) {
      Pair<String,Integer> p = map[i];
      if (p.getKey().equals(key)) {
        p.setVal( p.getVal() + 1 );
        return;
      }
    }
    map[nrElems++] = new Pair<>(key, 1);
  }
```

note: creating an Array of Pairs

no diamond operator here ! ?

postfix ++ returns the value before increment

note: creating a Pair

ignoring size problems

Mapping words to counts (2)

```
  public int get(String key) {
    for (int i = 0; i < nrElems; i++) {
      Pair<String,Integer> p = map[i];
       if (p.getKey().equals(key)) {
         return p.getVal();
      }
    }
    return 0;
  }

  public String[] keys () {
    String[] keys = new String[nrElems];
    for (int i = 0; i < nrElems; ++) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

counting words

one or more “non-word characters”

```
private void run( String line ) {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  CountMap map = new CountMap(100);
  while (scan.hasNext()) {
    map.add( scan.next().toLowerCase() );
  }
  for (String key: map.keys()) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

RUN

```
be: 2
is: 1
not: 1
or: 1
question: 1
that: 1
the: 1
to: 2
```

Generics for a single method

Often the generic variables belong to a class;
they can also belong to a *single* method

generic type arguments for method

return type of method with generic types passed in

argument of method with generic types passed in

```
public static <K,V> Pair<V,K> swap (Pair<K,V> p) {
  return new Pair<>(p.getVal(), p.getKey());
}
```

use this like any other method:

```
private static void run(){
  Pair<Integer,String> p = new Pair<>(1,"Foo");
  System.out.println(p);
  var ps = swap(p);
  System.out.println(ps);
}
```

RUN

```
(1,Foo)
(Foo,1)
```

local variable type inference: type of `ps` is inferred by the compiler

Limitations of generic arguments

type parameter <E> cannot be used as a constructor (to create a new objects)

```
E object = new E();
```

this is **NOT** allowed !

You also cannot create an array using <E>:

```
E[] elements = new E[100];
```

this is also **NOT** allowed !

A generic type parameter <E> of a class cannot be used in a static context.

```
private static E statField;
public static void method( E arg ) {…}
```

Both **NOT** allowed !

A generic map (1)

```
public class CountMap {
  private Pair<String,Integer>[] map;
  private int nrElems = 0;
  ...
}
```

The CountMap class can be generalized as follows:

```
public class GenMap<K extends Comparable, V> {
  private Pair<K,V>[] map;
  private int nrElems = 0;

  public GenMap(int size) {
    map = new Pair[size];
  }
  ...
}
```

bounded generic type:
to ensure that we can sort the keys

A generic map (2)

```
public class GenMap<K extends Comparable, V> {
  ...
  public void add(K key, V value) {
    for (int i = 0; i < nrElems; i++) {
      var p = map[i];
      if (p.getKey().equals(key)) {
        p.setVal( value );
        return;
      }
    }
    map[nrElems++] = new Pair<>(key, value);
  }

  public V get(K key) {
    for (int i = 0; i < nrElems; i++) {
      var p = map[i];
      if (p.getKey().equals(key)) {
        return p.getVal();
      }
    }
    return null;
  }
  ...
}
```

A generic map (3)

```
public class GenMap<K extends Comparable, V> {
  ...
  public K[] keys () {
    K[] keys = (K[]) new Object[nrElems];
    for (int i = 0; i < nrElems; i++) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

since new K[count] is not allowed

this gives an exception:
cannot cast Object to Comparable

Example run:

```
private static void run( String line ) {
   Scanner scan = new Scanner(line).useDelimiter("\\W+");
   GenMap<String,Integer> map = new GenMap<>(100);
   while (scan.hasNext()) {
     String word = scan.next().toLowerCase();
     Integer prevVal = map.get(word);
     map.add( word, prevVal == null ? 1 : prevVal + 1 );
   }
   for (String key: map.keys()) {
     System.out.println(key + ": " + map.get(key));
   }
 }
```

Fixing the generic map

good general rule as well; ArrayLists are so much nicer to work with…

return an `ArrayList<K>` instead of an Array

```
public class GenMap<K extends Comparable, V> {
  ...
  public ArrayList<K> keys () {
    ArrayList<K> keyList = new ArrayList<>();
    for (int i = 0; i < nrElems; i++) {
      keyList.add(map[i].getKey());
    }
    Collections.sort(keyList);
    return keyList;
  }
}
```

warning:

this Map class is only to demonstrate generic programming

there is a better reusable solution in Java

never ever implement a Map in your own program

unless you have a very good reason for it

Generics + subtyping (1)

```
public class Person  {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  public String getName() { return name; }
  @Override
  public String toString() {
    return "Person " + name;
  }
}

public class Student extends Person {
  private int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public String toString() {
    return "Student " + getName() + " " + num;
  }
}
```

Generics + subtyping (2)

Inheritance and types

Subtyping: a Derived class is a *subtype* of the Base class.

Subtyping rule:

If a *Base object* is demanded it is safe to offer a *Derived object*.

```
private static void show( Person p ) {
  System.out.println(p);
}

private static void run1() {
  Person p = new Person("Alice");
  Person s = new Student (1, "Bob");
  show(p);
  show(s);
}
```

this is fine: Student is a subclass of Person

polymorphism at work!

RUN

```
Person Alice
Student Bob 1
```

Generics + subtyping (3)

```
public class Box<T> {
  private T elem;
  public Box(T elem) { this.elem = elem; }
  @Override
  public String toString() {
    return "Box{" + elem + "}";
  }
}
```

In some class we have

```
private void run2 () {
  Person al = new Person("Alice");
  Person bo = new Student(1, "Bob");
  Box<Person> ba = new Box<>(al);
  Box<Person> bb = new Box<>(bo);
  show(ba);
  show(bb);
}
```

Box bb contains a Student, not a Person

```
private static void show( Box<Person> p ) {
  System.out.println(p);
}
```

RUN

```
Box{Person Alice}
Box{Student Bob 1}
```

Generics + subtyping (4)

However

```
private static void run3() {
  Person  al = new Person("Alice");
  Student bo = new Student(1, "Bob");
  show(al);
  show(bo);
}
```

still ok

```
private static void show( Person p ) {
  System.out.println(p);
}
```

```
private static void run4() {
  Person  al = new Person("Alice");
  Student bo = new Student(1, "Bob");
  Box<Person> ba  = new Box<>(al);
  Box<Student> bb = new Box<>(bo);
  show(ba);
  show(bb);
}
```

not ok, compiler complains: "no suitable method found"

reason: Box<Student> is not a subtype of Box<Person>

```
private static void show( Box<Person> p ) {
  System.out.println(p);
}
```

Generics + subtyping (5)

fixing the problem

```
private static void show( Box<? extends Person> p ) {
  System.out.println(p);
}
```

bounded wildcard generic

```
private static void run4() {
  Person  al = new Person("Alice");
  Student bo = new Student(1, "Bob");
  Box<Person> ba  = new Box<>(al);
  Box<Student> bb = new Box<>(bo);
  show(ba);
  show(bb);
}
```

works fine

Generics + subtyping (5)

fixing the problem

```
private static void show( Box<? super Student> p ) {
  System.out.println(p);
}
```

lower bound wildcard generic

```
private static void run4() {
  Person  al = new Person("Alice");
  Student bo = new Student(1, "Bob");
  Box<Person> ba  = new Box<>(al);
  Box<Student> bb = new Box<>(bo);
  show(ba);
  show(bb);
}
```

works fine

Collections

the class `Arrays`

- standard Java collections (a collection is a container that stores other data) often provide the same manipulations
  - searching, equality, sorting, ..
- **`Arrays`** provides useful operations on arbitrary arrays
  - fill, sort, binarySearch, equals, ..
  - we used the `Arrays.sort` in the assignments and above
- **`Arrays`** is a *utility class* (e.g. does ***not*** box an ordinary array)
  - it has no attributes
  - static methods of `Arrays` take the array as argument
    e.g. `Arrays.sort( words )`

variable length containers

sometimes the size of a container is not known in advance

- **`Strings`** have a fixed length
- **`StringBuffers`** can be changed
  - it is always possible to add a character

ordinary arrays have a fixed length

`ArrayList` and **`LinkedList`** have a *variable length*

- it is always possible to add an element, at any place
- we can remove an element without the need to shift elements explicitly
- both classes implement the (generic) interface **`List<T>`**

list based map

```
public class MapList<K extends Comparable<K>, V> {
  private final List<Pair<K,V>> map;

  public MapList() {
    map = new ArrayList<>();
  }

  public void add(K key, V value) {
    for (var pair: map) {
      if (pair.getKey().equals(key)) {
        pair.setVal( value );
        return;
      }
    }
    map.add(new Pair<>(key, value));
  }

  public V get(K key) {
    for (var pair: map) {
      if (pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }
  ...
}
```

`List` is an interface

no size argument

`ArrayList` is class implementing `List`

always fits

the interface **`Collection`**

we have several containers in Java

- `String`, `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

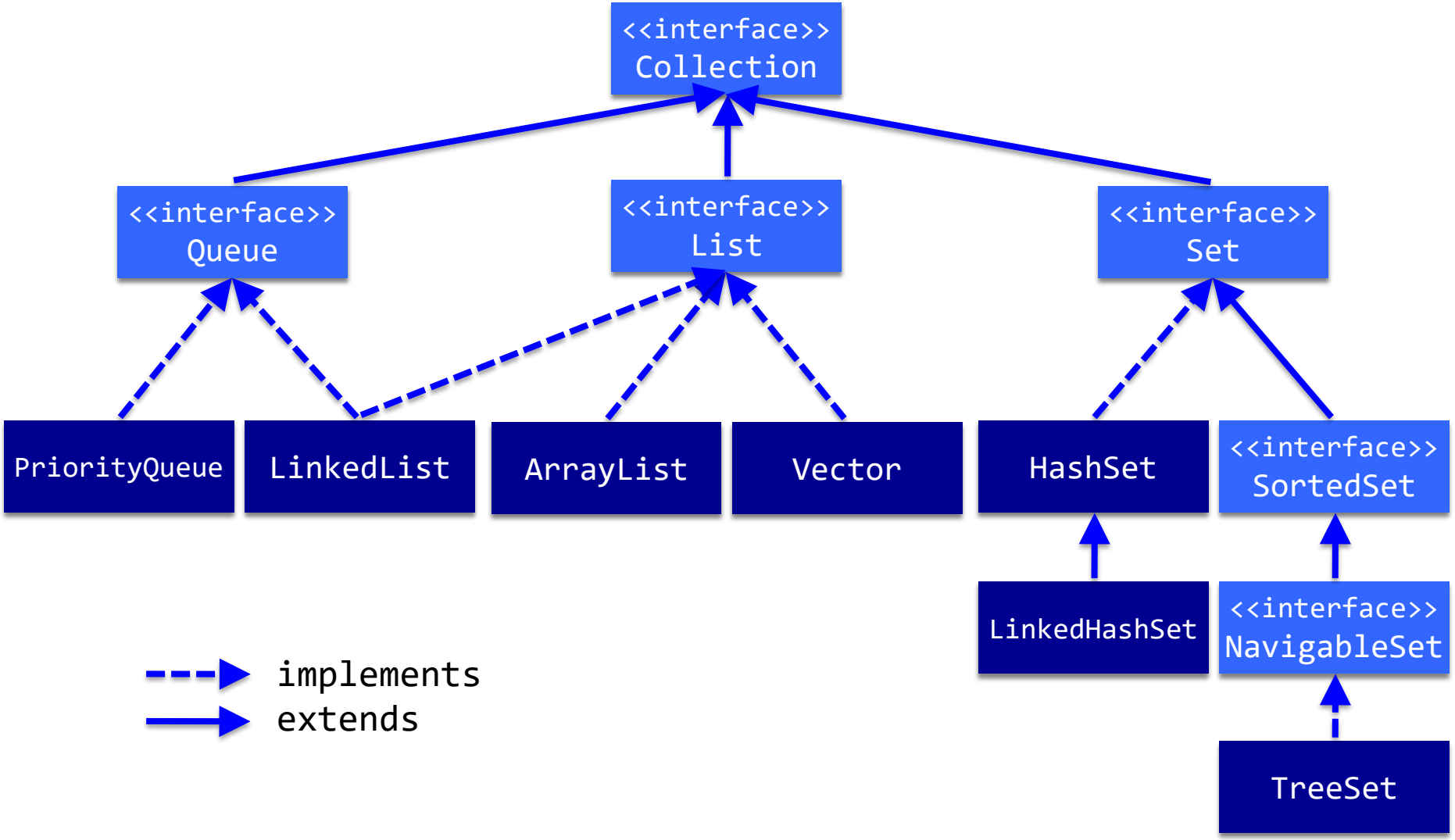
- `isEmpty`, `contains`, `equals`, `size`

the interface **`Collection`** yields a uniform way to handle these kind of operations

warning: there is also a (utility) class **`Collections`**

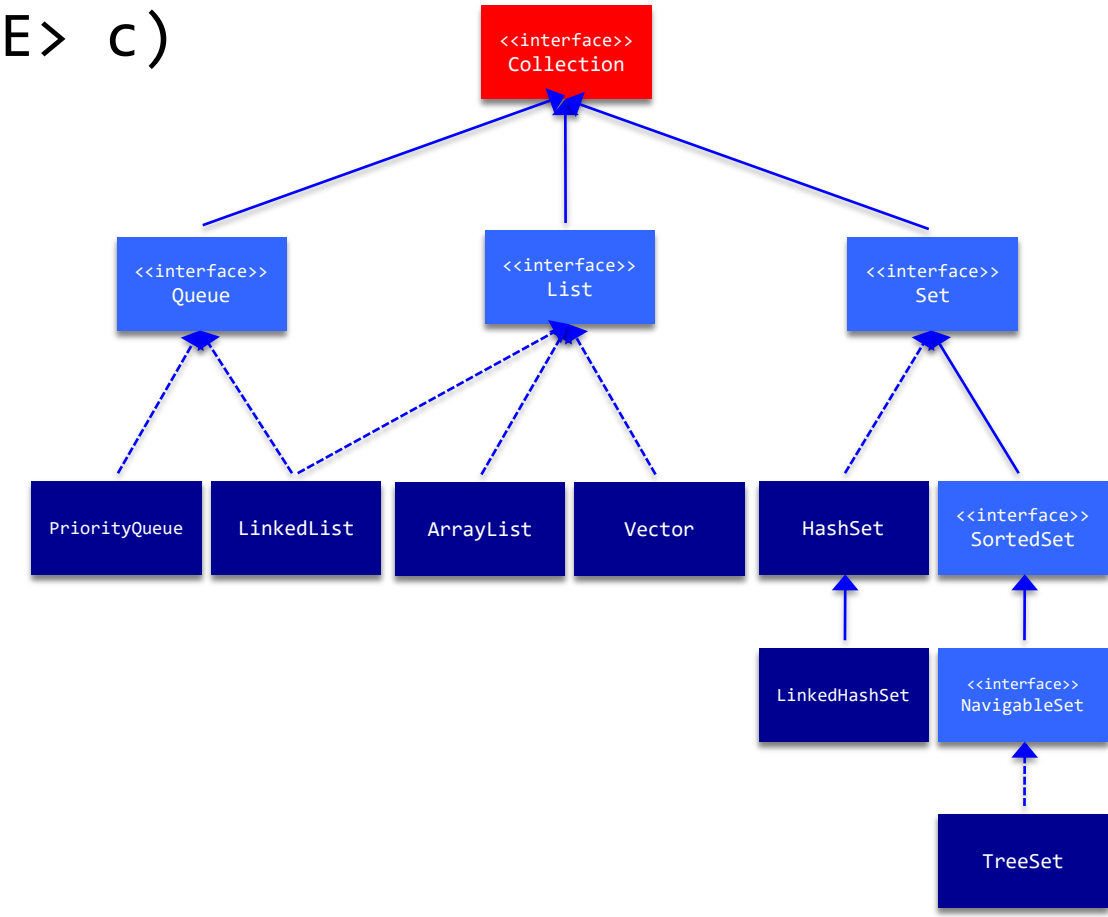
- `Collections` is similar to `Arrays`: set of basic operations provided as static methods
- don't confuse them

Collection interface hierarchy



main methods in interface `Collection<E>`

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int     size()
```



Lecture 6: Collections continued