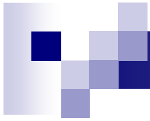
Basics of programming 3

Java collections



Java Generics

Generics

- Objective: code reuse with generic types

- `C` solution
  - `void* malloc(size_t s)`
    - casting is dangerous
    - error may remain hidden
  - `#define max(a,b) ((a)>(b)?(a):(b))`
    - side-effects cause problems

Generics

- Objective: code reuse with generic types

- `C++` solution
  - ```
    template <class T> T max(T a, T b) {
     return (a>b)?a:b;
    }
    ```
    - semantic and syntactic requirements for `T`
      - documentation is important
    - errors might be uncovered at compile time
    ```

Generics

- Objective: code reuse with generic types
- **`Java`** original solution (pre 5.0)
  - all classes are subclasses of *Object*
    - **`C`**’s **`void*`** philosophy in new disgease
    - code is full of casting
    - errors uncovered during runtime
    - type check: operator *instanceOf*

```
if (s instanceOf(String)) ...
```

Generics example: Object cast

```
public class Store {
  Object[] os; int size;
  public Store(int i) {
    os = new Object[i];
    size = 0;
  }
  public void put(Object o) {
    os[size] = o; size++;
  }
  public Object get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

```
Store s = new Store(10);

s.put("hello ");
s.put("world");
s.put("!");

for (int i = 0;
     i < s.size();
     i++) {

  String l = (String)s.get(i);
  System.out.print(l);

}
```

Cast in client code ☹

Java generic classes

- Since Java 5
- Similar to C++ templates
  - But: no class generated for each parametrization
- Parameter interface is definite
  - programmer knows it
- Errors uncovered during compile time
- Cast-free source code
- Template parameter can not be primitive type

Generics example: Template

```
public class Store<T> {
  T[] os; int size;
  public Store(int i) {
    os = (T[])(new Object[i]);
    size = 0;
  }
  public void put(T o) {
    os[size] = o; size++;
  }
  public T get(int i) {
    return os[i];
  }
  public int size() {
    return size;
  }
}
```

Workaround for arrays

```
Store<String> s =
      new Store<String>(10);

s.put("hello ");
s.put("world");
s.put("!");

for (int i = 0;
     i < s.size();
     i++) {

  String l = s.get(i);
  System.out.print(l);

}
```

Generics and inheritance (array)

```
Object[] oa = new String[10];
oa[0] = ”Hello”;
oa[1] = new Integer(2);
      // RT error: ArrayStoreException
```

- **`Object[]`** can be replaced by **`String[]`**
- Types compatible with dynamic type can be stored only

Generics and inheritance

```
// add a method to class Store<T>:
void printStore(Store<Object> of) {
  for (int i = 0; i < of.size(); i++) {
    System.out.println(of.get(i));
  }
}
```

```
// let's call it
Store<String> s = new Store<String>();
Store<Integer> i= new Store<Integer>();

s.printStore(i); // CT error!
```

Generics and inheritance

```java
// same from a different angle

Store<String> s = new Store<String>(10);
Store<Object> o = s; // CT error!
o.put(new Integer(13)); // incompaatibility!
```

- **`Store<Object>`** is **not** a superclass of **`Store<String>`!**
- Their types are incompatible

Wildcard: `?`

```
void printStore(Store<?> of) {
 for (int i = 0; i < of.size(); i++) {
    System.out.println(of.get(i));
  }
} // now OK
```

```
Store<String> sf = new Store<String>();
Store<?> of = sf;
of.put(new Integer(13));   // CT error
of.put(”Hello”);           // CT error
```

- `<?>` can be converted to anything, like `<Object>`
  - accessed references are converted to *Object*
  - assignment to references of *?* is prohibited

Bound wildcard: *subclasses*

```
// add to Store<E>
void putAll(Store<E> st) {
  for(int i = 0; i < s.size(); i++) {
      put(st.get(i));
}
```

- What if type of **`st`** is **`Store<Q>`**, where **`Q`** is *subclass* of **`E`**?

```
Store<Person> p  = new Store<Person>();
Store<Person> p2 = new Store<Person();
Store<Student> s = new Store<Student>();
...
p.putAll(p2); // OK
p.putAll(s);  // CT error
```

Bound wildcard: *extends*

```
// add to Store<E>
void putAll(Store<? extends E> st) {
  for(int i = 0; i < s.size(); i++) {
      put(st.get(i));
}
```

- Now type of parameter *st* is correct
  - *st* holds *E* or subclasses of *E*

```
Store<Person> p  = new Store<Person>();
Store<Person> p2 = new Store<Person();
Store<Student> s = new Store<Student>();
...
p.putAll(p2); // OK
p.putAll(s);  // OK
```

Bound wildcard: *superclasses*

```
// new method in Store<E>
void put2All(Store<E> st) {
  for(int i = 0; i < size(); i++) {
      st.put(get(i));
}
```

- What if type of **`st`** is **`Store<Q>`**, where **`E`** is *subclass* of **`Q`**?

```
Store<Student> s  = new Store<Student>();
Store<Student> s2 = new Store<Student>();
Store<Person>  p  = new Store<Person>();
...
s.put2All(s2); // OK
s.put2All(p);  // CT error
```

Bound wildcard: *super*

```
// new method in Store<E>
void put2All(Store<? super E> st) {
  for(int i = 0; i < size(); i++) {
      st.put(get(i));
}
```

- Now type of parameter *st* is correct
  - *st* holds *E* or *superclasses* of *E*

```
Store<Student> s  = new Store<Student>();
Store<Student> s2 = new Store<Student>();
Store<Person>  p  = new Store<Person>();
...
s.put2All(s2); // OK
s.put2All(p);  // OK
```

- What if generic type must be **X** and **Y** at the same time?

```
<T extends X & Y>
```

```
public static
<T extends Object & Comparable<? Super T>>
T max(Collection<? extends T> coll)
```

Restrictions on generic types

- No primitive type for generic parameter

```
Store<int> st = new Store<int>(); // ☆
```

- No instance of type parameter may be created

```
T t = new T(); // ☆
```

- No static fields with type parameter allowed

```
static public T test; // ☆
```

```
Store<String>.test = "hello";
Store<Integer>.test = 13;       // ☆
```

Restrictions on generic types

- No instanceOf

```
void foo(Store<T> s) {
      if (s instanceOf(Store<String>)) ... // ☆
```

- No arrays of parameterized types

```
Store<String>[] l1, l2, l3;
l1 = new Store<String>[10];             // ☆
l2 = (Store<String>[])new Object[10]; // ☆
l3 = (Store<String>[])new Store[10];  // ☺
```

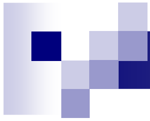
  - Casting is best avoided

Restrictions on generic types

- No method overload with parameter types
  - could map to same type (e.g. *S* = *T* = *String*)

```
class Generic<S,T> {
   public void foo(S s) {...}
   public void foo(T t) {...} // ☆
```

- Limited exception handling
  - No generic exception type
  - No catch for type parameter
  - Can be thrown



Java Collection Framework

Collections: array

- built-in type
- immutable size
  - C/C++ doesn't check, has **`realloc`**
  - Java checks, no **`realloc`**
    - cannot avoid reference copy
- inheritance and type compatibility is problematic

```
Object[] oa = new String[10];
oa[3] = new Double(3.14); // ☆
```

- convenient use

- Linked list
  - single or double-linked, ring, sentinel/guard, comb
- Binary (n-ary) tree
  - balance, red-black, AVL
  - word-tree
- Hash-table
  - associative storage
  - hash function is important

Collections: general features

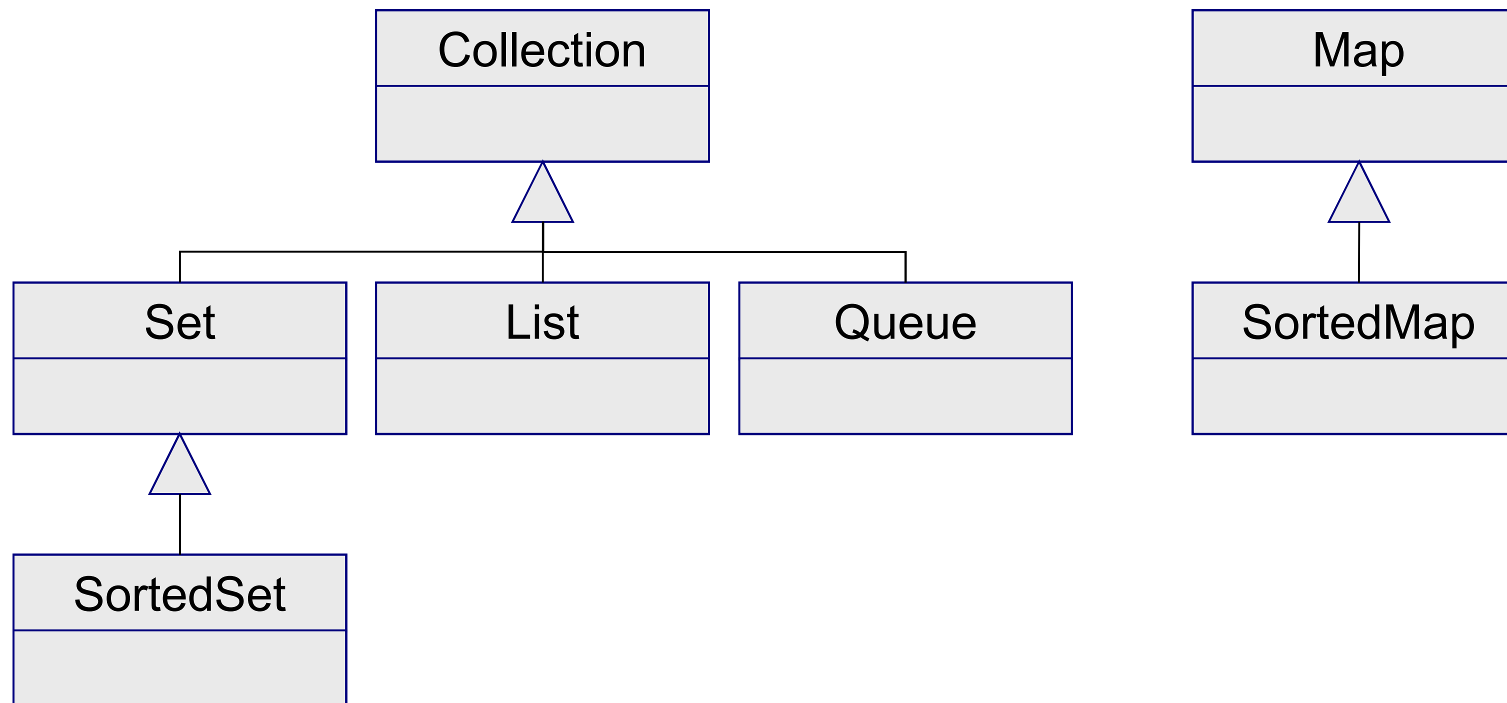
- Common basic functionality
  - insert, search, update, delete
    - CRUD: create, read, update, delete
  - iteration
- Different implementations
  - ordered
  - set/bag
  - deep/shallow copy
  - optimization: e.g. insert vs. search

Collection usage

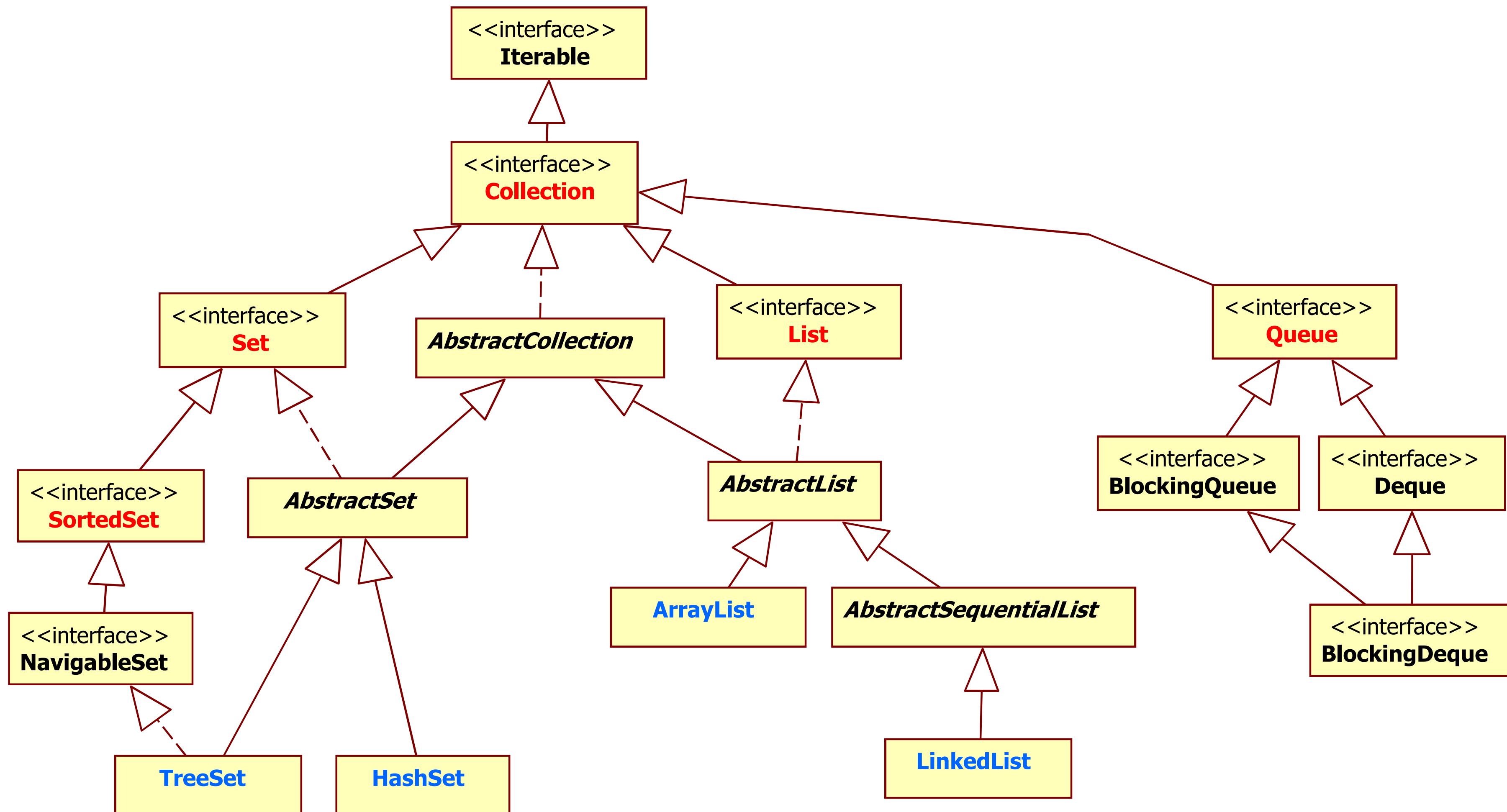
```java
List<Integer> l = new ArrayList<Integer>();

// since J2SE 7
// List<Integer> l = new ArrayList<>();

for (int i = 0; i < args.length; i++) {
        l.add(Integer.parseInt(args[i]));
}

for (int i = 0; i < l.size(); i++) {
        System.out.println(l.get(i)+10);
}
```

Collection framework (simple)

- Interfaces



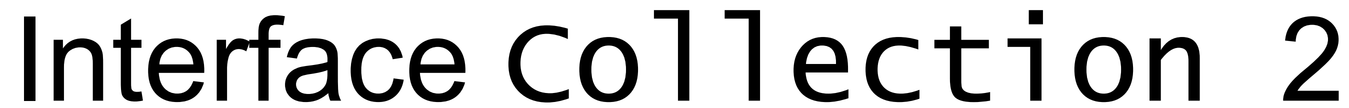
Collection framework (detailed)



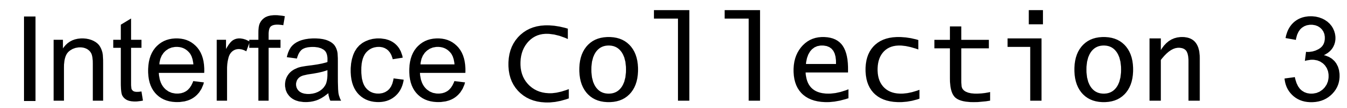
- General collection functionality
- May be
  - ordered or not
  - set or bag
  - some methods might be optional (unimplemented)
- Generic definition:

```
Collection<E>
```

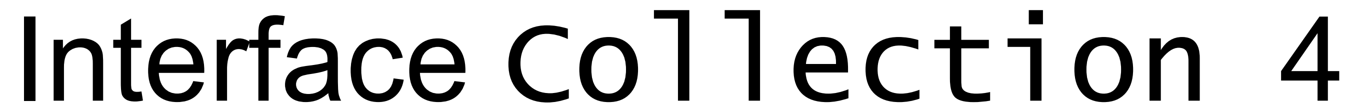
Interface Collection 2



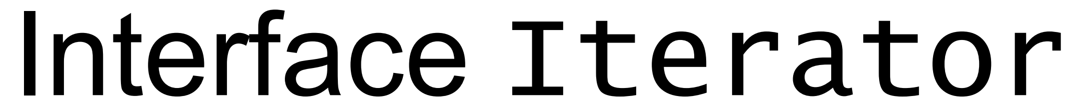
- `void add(E e)` *optional*
  - adding new object
- `void addAll(Collection<? extends E> c)` *optional*
  - **`c`** collection's all objects added
  - only references are copied: shallow copy
- `boolean remove(E e)` *optional*
  - removes object if contained
- `boolean removeAll(`*`Col`*`<?` *`ext`* `E> c)` *optional*
  - all object in **`c`** are deleted from this collection
  - only references are deleted, no direct destruction takes place!



- `boolean contains(E e)` *optional*
  - true if e is stored
- `boolean containsAll(Col<? ext E> c)` *optional*
  - true, if all objects in c are stored
- `int size()`
  - number of stored objects
- `boolean isEmpty()`
  - true if empty
- `void clear()`
  - all references stored are deleted



- boolean retainAll(*Col*<? *ext* E> c) *optional*
  - retaining only those objects stored in c
- boolean equals(E e)
  - equality
  - must be symmetric
- Object[] toArray()
  - collection converted to array
- <T> T[] toArray(T[] ta)
  - collection converted to array of type similat to that of ta
- Iterator<E> iterator()
  - returns an iterator for this collection



Interface Iterator

- Implemented by all iterators
- Iteration over elements
  - different semantics from C++ STL iterators!
- Definition: Iterator<E>
- boolean hasNext()
  - true, if there are uniterated elements
- E next()
  - returns next element, if any
- void remove()
  - removes element last returned by next()

- Typical usage:

```
Collection<Integer> c = ...;
...
Iterator<Integer> i = c.iterator();

while (i.hasNext()) {
        int a = i.next(); // outboxing
        if (a < 0) {
                i.remove();
        }
}
```

- Handling multiple access
  - modifying a collection during iterations causes errors
  - `ConcurrentModificationException` is thrown

```
Collection c = ...;
...
Iterator i1 = c.iterator();
Iterator i2 = c.iterator();
i1.next();
i2.next();
i2.remove();
i1.next();    // here exception is thrown
```

Interface `Set`

- Set: every object stored only once
- Ordering is unknown
- Iterator may iterate in any order
- No extra methods
  - all Collection methods are implemented
- Typical implementation: `HashSet`
  - good hash function needed for efficiency

Interface `SortedSet`

- Set with content ordered
- Content decides ordering
  - natural or `Comparator`
    - Natural: `Comparable.compareTo(Object o)`
    - Comparator: `int compare(Object o1, Object o2)`

      Compares its two arguments for order

      boolean **equals**(Object obj)

      obj is equal to this Comparator
- `Iterator` iterates in order
- Typical implementation: `TreeSet`

Interface `List`

- Sequence of elements
- A single object can appear multiple times
- The index of each element is known
  - access by index is provided
- Searchable (object -> index)
- Provides `ListIterator` for easier access
  - subclass of Iterator, extended functionality
- Typical implementation: `ArrayList`

Interface `List 2`

- Extra methods of `List<E>`
  - `add(int index, E e)`
  - `E get(int index)`
  - `int indexOf(Object)`
  - `int lastIndexOf(Object)`
  - `E remove(int index)`
  - `boolean remove(Object o)`
  - `E set(int index, E e)`
  - `List<E> subList(int from, int to)`
  - `...`

Interface ListIterator

- Additional functionality
  - index-based methods
    - `int nextIndex()`: index of next element
    - `int previousIndex()` : index of previous element
  - reverse iteration
    - `boolean hasPrevious()` : like hasNext but checks for previous
    - `T previous()` : returns previous element in list
  - list modification
    - `set(T t)` : overwrites element that was last returned
    - `add(T t)` : inserts element before next (before cursor)

Interface `Map`

- Stored keys and values
- For a key the value can be set and get
- Using mutable keys are not advised
- Three view
  - set of keys
  - set of values
  - set of key-value pairs
- Typical implementation: `HashMap`

- Definition: Map<K,V>
  - K key types
  - V value type
- Methods similar to Collection methods:
  - void clear()
  - boolean equals()
  - boolean isEmpty()
  - int size()

Interface Map 3

- V put(K key, V value)
  - adding the key-value pair to the map
- void putAll(Map<? ext K, ? ext V> m)
  - adding all pairs in m to this map
- V get(K key)
  - returns value for this key, does not remove it
- V remove(K key)
  - return value for this key, key-value removed

- boolean containsKey(Object key)
- boolean containsValue(Object value)
  - true if contained
- Set<K> keySet()
  - returns set of keys
- Collection<V> values()
  - return collection of values
- Set<Map.Entry<K,V>> entrySet()
  - return set of key-value pairs

Interface `SortedMap`

- Extending Map with sorted keys
  - natural or `Comparator`
  - similar to SortedSet
- Views accordingly
  - `keys(), values(), entrySet()`
  - iterators iterate according to key order
- Typical implementation: `TreeMap`

For-each loop

- Pre Java 5 with *iterator*
  - `hasNext()` and `next()`
- Since Java 5 simplified *for* loop (also for arrays)
  - supports implementations of interface *Iterable*

```
Collection<Integer> c = ...;
...
for (Integer i : c) {
        System.out.println(i);
}
```

```
static public void main(String[] args) {
  for (String s : args) { System.out.println(s); }
}
```

Collections example

- Let's print the frequency of letters

```
SortedMap<Character,Integer> map =
        new TreeMap<>();
Reader r = new InputStreamReader(System.in);
while (true) {
        int i = r.read();
        if (i < 0) break;
        char c = (char)i;
        int n = 0;
        if (map.containsKey(c)) n = map.get(c);
        n++;
        map.put(c, n);
}
for (char c : map.keySet()) {
        System.out.println("Char: " +c+"= \t"+map.get(c));
}
```

J2SE7

boxing

`map.get(c)++` wouldn't work

iterating over keys

For-each-loop vs Iterator

- For-each-loop
  - + more readable code
  - - no collection modification
- Iterator
  - + element removal supported
  - - code overhead, readability (?)

Collections: a helper class

- Sorting, min-max search, etc.
  - sort(List<T> l)
  - sort(List<T> l, Comparator<T> c)
- Reversing
  - reverse(List<T> l)
- Rotation
  - rotate(List<T> l, int distance)
- Shuffling
  - shuffle(List<T> l)
  - shuffle(List<T> l, Random r)

Collections example

- Frequency of letters in order of frequency

```
... // reading chars like previously
class Cmp implements
Comparator<SortedMap.Entry<Character, Integer> > {
    public int compare(Entry<Character, Integer> e1,
                       Entry<Character, Integer> e2) {
        return e2.getValue()-e1.getValue();
  }
}
List<SortedMap.Entry<Character, Integer>> l =
    new ArrayList<>();
l.addAll(map.entrySet());
Collections.sort(l, new Cmp());
for (SortedMap.Entry<Character, Integer> e : l) {
    System.out.println(e.getKey()+"= "+e.getValue());
}
```

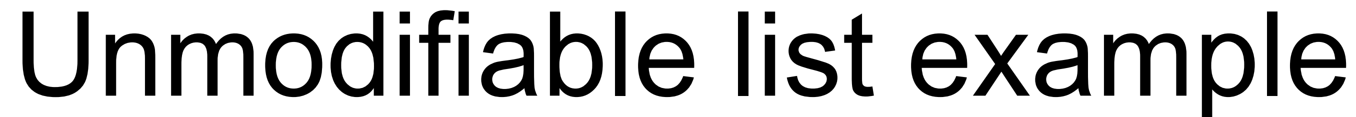
comparator

helper list

list init

sorting

- Collections as parameters
  - Java: shallow copy → mutable by default
  - C++: *const* keyword → immutability
- How to pass immutable collections in Java?
- `java.util.Collections`
  - `Collection unmodifiableCollection(Collection c)`
  - `List unmodifiableList(List c)`
  - `Map unmodifiableMap(Map c)`
  - `Set unmodifiableSet(Set c)`
  - generic types are supported




```
// getting a list from somewhere
List<Student> l = database.getStudents();

// passing the list to some method:
// what happens to content?
doSomethingWithList(l);
```

```
// getting a list from somewhere
List<Student> l = database.getStudents();

// creating an unmodifiable wrapper list
// no copy of the list's contents, generics observed
List<Student> l2 = Collections.unmodifiableList(l);

// passing the wrapper list to some method:
// no modifier methods (UnsupportedOperationException)
checkForSomething(l2);
```