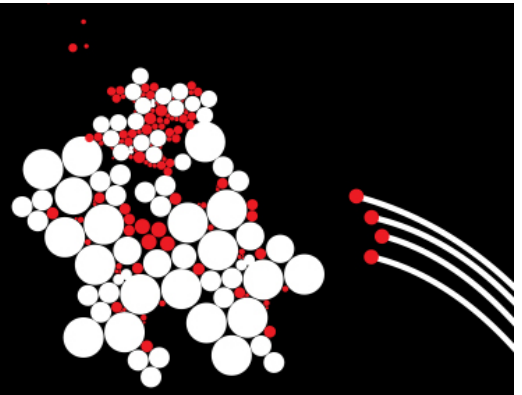
UNIVERSITY OF TWENTE.

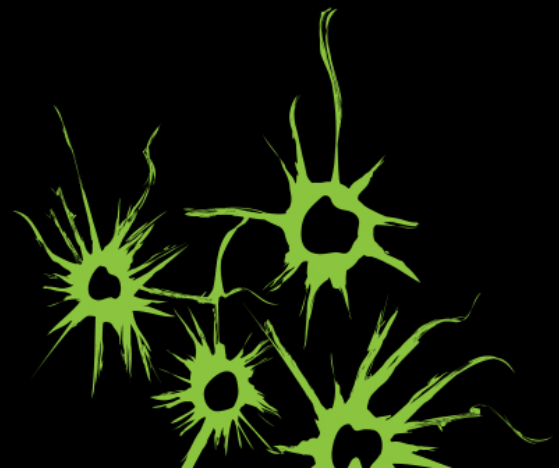


P4.3: COLLECTIONS

LUÍS FERREIRA PIRES

201700117-1B MODULE 2: SOFTWARE SYSTEMS

5 DECEMBER 2019



PROGRAMMING LINE OVERVIEW

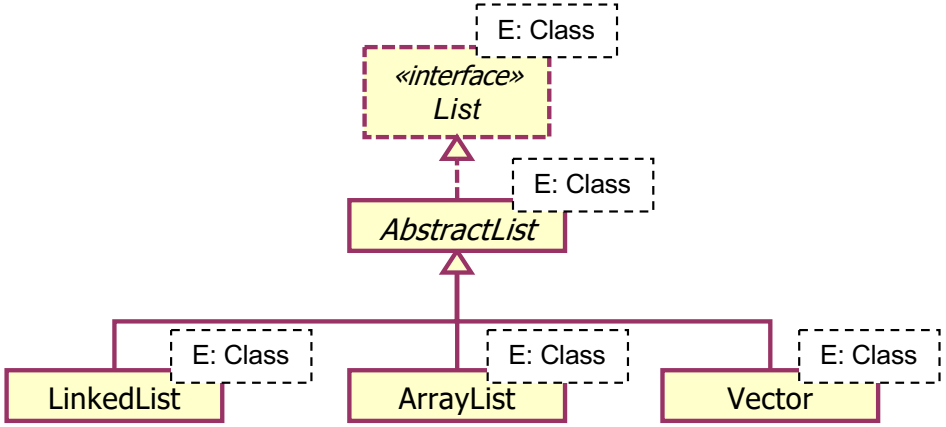
Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Stream I/O and MVC Exceptions Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

JAVA STANDARD INTERFACE: JAVA.UTIL.LIST

- A list implements the concept of sequence
  - List elements have an index and are therefore ordered
- `Lists` are generic
  - Elements in a list have the same time
  - Type must be specified

JAVA STANDARD INTERFACE: JAVA.UTIL.LIST

- Class hierarchy (incomplete)
  - `List` interface: no functionality
  - `AbstractList`: basic methods
- Implementations
  - `ArrayList`: efficient indexing
  - `LinkedList`: efficient manipulation



EXAMPLE
LIST OF STUDENTS

- A Student is identified by a student number and has a name

```
public class Student {
   private final String number;
   private final String name;

   public Student(String number, String name)
{
      …
   }

   // getters and setters
}
```

Number	Name
s0123	Mary
m0246	John
s1345	Kim

EXAMPLE
LIST OF STUDENTS

- Usage

```
List<Student> slist = new ArrayList<>();
slist.add(new Student("s0123", "Mary"));
slist.add(new Student("m0246", "John"));
slist.add(new Student("s1345", "Kim"));
```

Shortcut: type can be omitted

- What is printed in this code?

```
if (slist.contains(new Student("s0123", "Mary"))) {
    System.out.println("Mary's here!");
} else {
    System.out.println("Where's Mary?");
}
```

Method `contains` does not behave as expected → `Student.equals` should be overridden!

```
String hello = new String("hi");
if(hello=="hi") {
   System.out.println("Say hi");
}
```

EXAMPLE
LIST OF STUDENTS

- Override method `equals`
- Method `compare` uses `equals` and compare correctly

```
@Override
public boolean equals(Object obj) {
        if (!(obj instanceof Student)) {
      return false;
    }
    Student other = (Student) obj;
    return other.name.equals(name) && other.nr.equals(nr);
}
```

- You may decide whether it is sufficient to compare student numbers

USING LISTS (AND OTHER COLLECTIONS)

VARIABLE DECLARATION AND INSTANTIATION

- Declared type: as abstract as possible (interface)
- Instantiated type: choose appropriate concrete implementation
- This improves maintainability

Example

```
List<Student> slist = new ArrayList<>();
...
slist.remove(…);
```

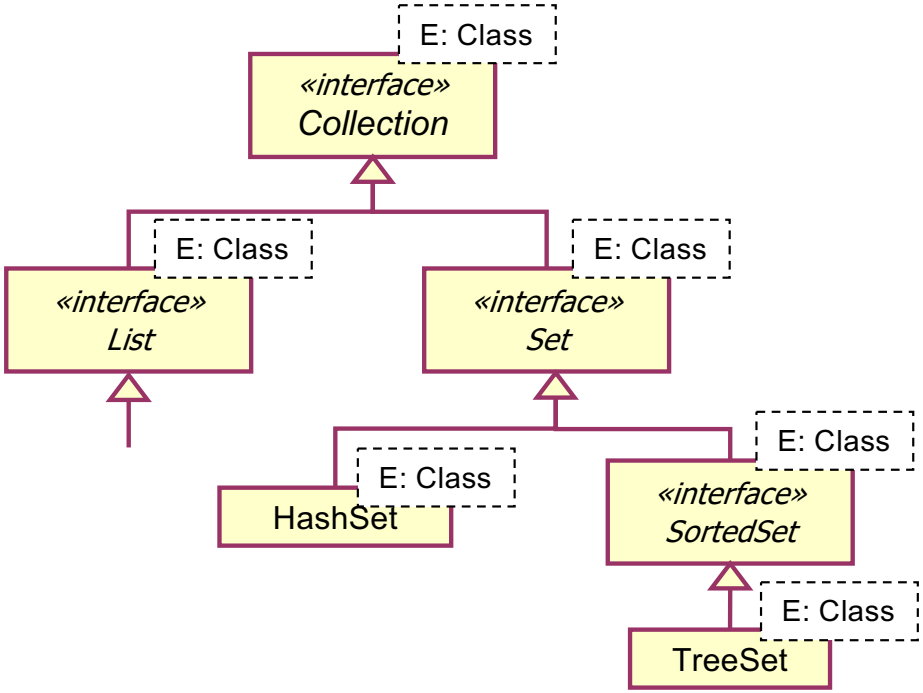
- `remove` is inefficient for `ArrayList`
- Easily changed!

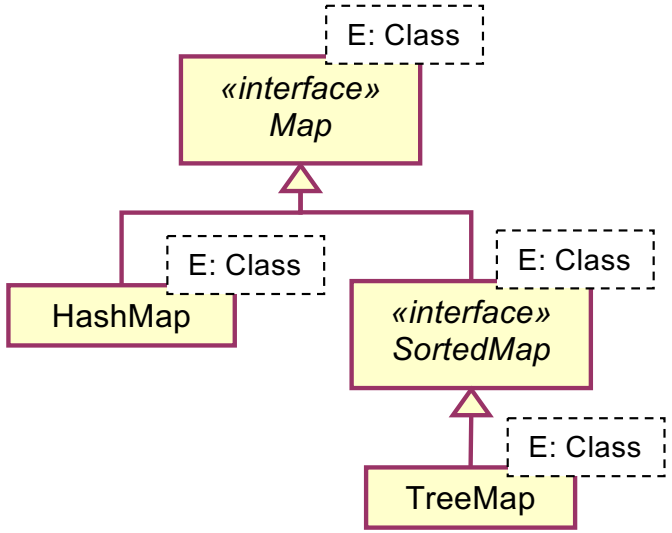
```
List<Student> slist = new LinkedList<>();
...
slist.remove(…);
```

JAVA COLLECTION HIERARCHY

- Besides `List`, there are other fundamental data structures
  - `Set` implements the mathematical concept of a set (surprise…)
  - `Map` implements the mathematical concept of a function
  - Again, both have (many) different implementations

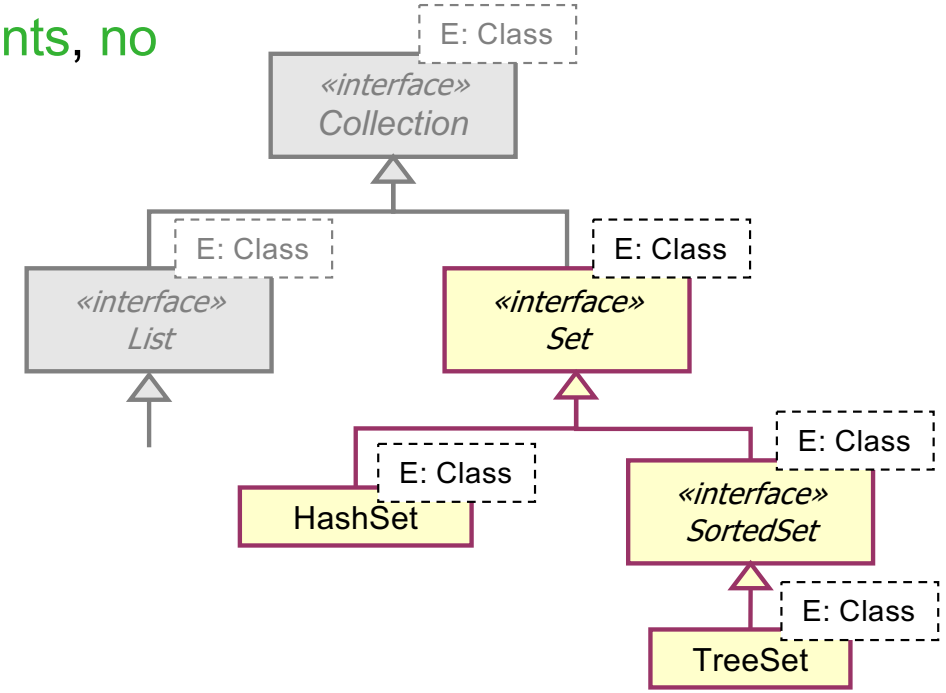
JAVA COLLECTION HIERARCHY





COLLECTIONS: SET

- A `Set` has no duplicate elements, no indexing and no ordering
- `HashSet` is the preferred implementation



HASHING PRINCIPLE

HASH CODES

- Calculate pseudo-random numbers, called hash codes, and assign them to `Object`s
- Store an `Object` using its hash code
  - If you know the hash code, you can find the element in a relatively small almost constant amount of time
  - Unlike when you store `Object`s in a (long) List
- Purpose: fast way to find information

`hashcode()` method of `Object` returns hash code!

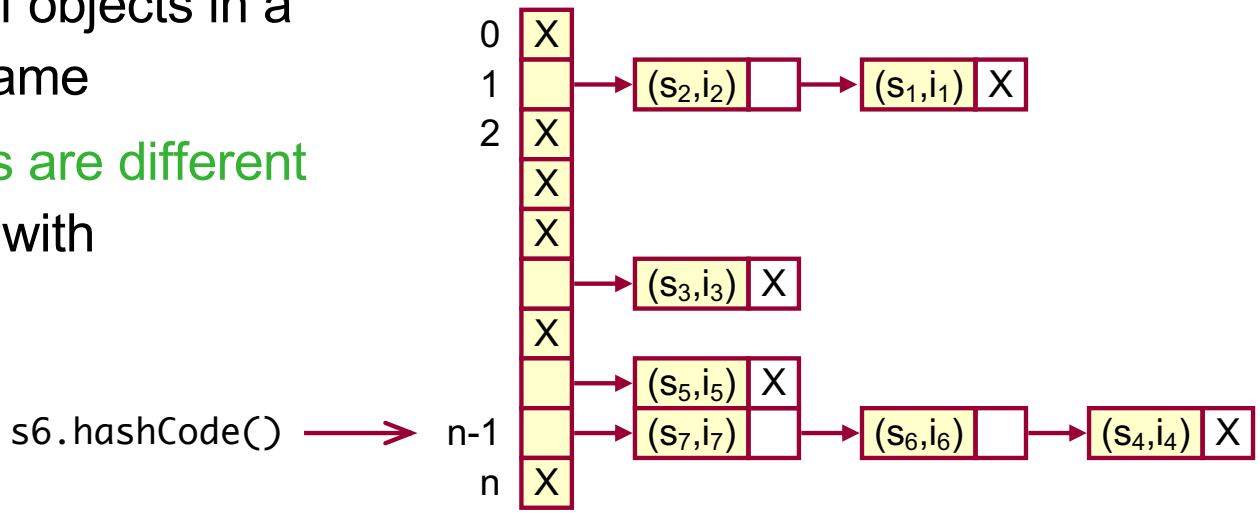
HASHING

TYPICAL IMPLEMENTATION

- Uses a (fixed) number of buckets
- Each bucket contains a linked list
- Method `hashCode()` relates an object to a bucket
- `equals()` can be used to compare elements in the same bucket
- Collision: two different objects have same hash code
  - Efficient hash functions avoid too many collisions
  - Hash codes should be distinct in as many bits as possible

HASHING
BUCKET

- All hash codes of objects in a bucket are the same
- ... but the objects are different when compared with `equals(Object)`



EXAMPLE
LIST OF STUDENTS

- Usage

```
Set<Student> sset = new HashSet<>();
 sset.add(new Student("s0123", "Mary"));
 sset.add(new Student("m0246", "John"));
 sset.add(new Student("s1345", "Kim"));
```

- What is printed here?

```
if (sset.contains(new Student("s0123", "Mary"))) {
   System.out.println("Mary's here!");
} else {
   System.out.println("Where's Mary?");
}
```

`Student` has an appropriate implementation for `equals`

But lacks an implementation for `hashCode`

EXAMPLE

HASH CODE FOR STUDENT CLASS

- Override the `hashCode()` method

```
@Override
public int hashCode() {
  final int PRIME = 37;
  return nr.hashCode()*PRIME + name.hashCode();
}
```

- Alternatively use the existing `hashCode()` implementation for Strings
- To combine `hashCode()` it is customary to multiply them by primes
- A cryptographic hash can be used, if you want to be fancy

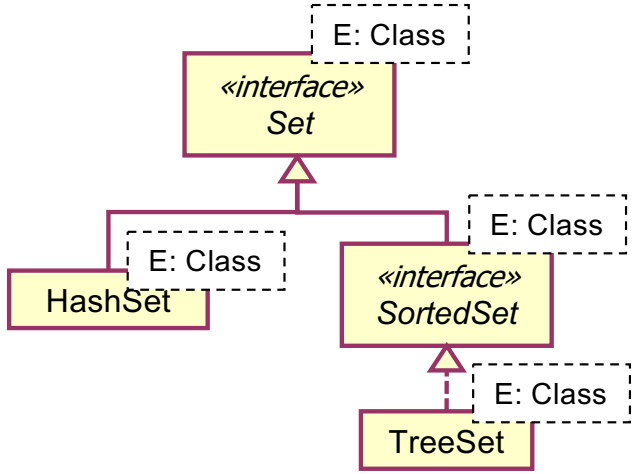
HASH CODE SUPPORT IN JAVA

- Each class has a method `int hashCode()`
  - To use `HashSet`, override `hashCode()` as well as `equals(Object)`
- Equality of objects implies equality of hash codes
  - If `o1.equals(o2)` then `o1.hashCode() == o2.hashCode()`
  - Programmer's responsibility to ensure this, otherwise `HashSet` methods like `add` and `contains` fail unpredictably

Distinct objects may still have the same hash codes, hence the need for linked lists in buckets

COLLECTIONS
SET

- A `Set` has no duplicates, no indexing and no ordering
- `TreeSet` is an alternative implementation for a `SortedSet`
- Sorts the elements with a binary tree
- Class of element must implement the `Comparable` interface




```
public int compareTo(E o)
@ensures if this < o then return < 0
  if this.equals(o) then return == 0
  if this > o then return > 0
```

EXAMPLE
LIST OF STUDENTS

- Usage

```
public class Student implements Comparable<Student>{
   ...
}
```

- Override `compareTo`

Sort the students by using their student numbers

```
@Override
public int compareTo(Student o) {
   return nr.compareTo(o.getNr());
}
```

ITERATOR
REVISITED

- All collection classes have an iterator
- `Collection<E>` has a method `iterator()` that returns an `Iterator<E>`
- Main `Iterator` methods
  - `boolean hasNext()`: returns `true` if iteration has more elements
  - `E next()`: returns next element in the iteration
  - `void remove()`
    - Removes last element returned by this `Iterator`
    - Can be called only once per call to `next()`

ITERATOR TYPICAL USAGE PATTERN

ITERATE THROUGH A LIST AND PRINT THE ELEMENT

```
Iterator<Student> i  = slist.iterator();
while (i.hasNext()) {
  Student s = i.next();
  System.out.printf("Nr: %s, name: %s%n", s.getNr(), s.getName());
}
```

- Same as

```
for (Student s: slist) {
  System.out.printf("Nr: %s, name: %s%n", s.getNr(), s.getName());
}
```

ITERATOR TYPICAL USAGE PATTERN

COLLECTION MANIPULATION

- Sometimes you need an `Iterator`, for example when the collection must be manipulated

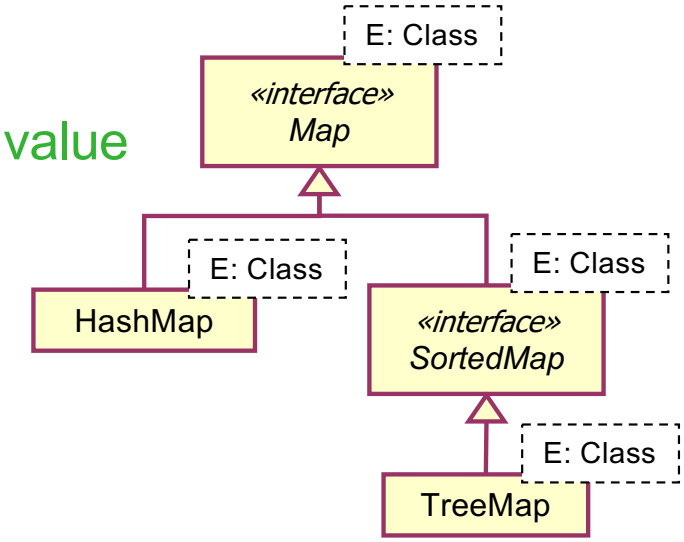
```
public static void removeInvalidNr(Collection<Student> scoll)
{
    Iterator<Student> i = scoll.iterator();
    while (i.hasNext()) {
        if (!i.next().getNr().matches("s[0-9]*")) {
            i.remove(); // Do not use scoll.remove(i)
        }
    }
}
```

Regular expression

Behaviour of `Iterator` is unspecified if underlying collection is modified during iteration not by calling `remove`

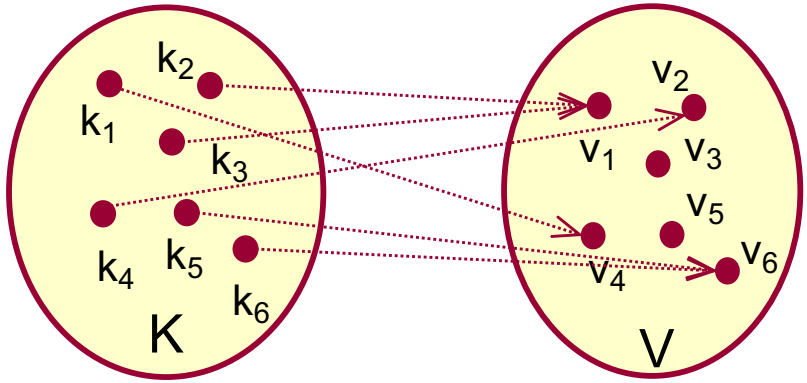
MAPS

- `Set` implements the mathematical concept of a set (surprise…)
- `Map` implements the mathematical concept of a function
- Stores key/value pairs
  - Each key is unique and mapped to one value
- Same value may appear multiple times
- Hierarchy similar to `Set` hierarchy
  - `HashMap` is the preferred implementation
  - `TreeMap` is a sorted map



MAPS

- `Map<K, V>` implements mathematical concept of a function
  - *Map*: *Keys* → *Values*
  - For each key, there is at most one corresponding value



SOME MAP<K,V> METHODS

- `get(K key)`: returns `value` associated with `key` (could be `null`)
- `put(K key, V value)`: associates `key` with `value`
- `remove(K key)`: removes mapping for `key`
- `containsKey(K key)` : `true` if the `key` is used in the `Map`
- `containsValue(V value)`: `true` if `value` is associated with some `key`

SOME MAP<K,V> METHODS
(CONT.)

- From Maps to Sets
  - `keySet()`: returns a `Set` containing all `key`s used by this `Map`
  - `values()`: returns all `values` associated with at least one `key` in this `Map` (returns a `Collection`)
  - `entrySet()`: returns `Set` that contains the mappings of this `Map`

EXAMPLE
STUDENT GRADES

- `Map` of `Student` grades

```
Map<Student, Integer> grades = new HashMap<>();
// Provide some input
grades.put(new Student("s0124", "Adrian"), 7);
Student chris = new Student("s1102", "Chris");
grades.put(chris, 4);
grades.put(chris, 6); //resit
if (grades.get(chris) >= 6) {
    chris.makeCertificate();
}
```

EXAMPLE

STUDENTS GRADES: ACCESS TO MAP PARTS

```
Collection<Student> enrolled= grades.keySet();
for(Student s:enrolled){
    System.out.println(s.toString());
}

Collection <Integer> given= grades.values();
for(Integer g:given){
    System.out.println(g);
}

Set<Entry<Student, Integer>> a = grades.entrySet();
for(Entry<Student, Integer> e:a) {
        System.out.println(e.toString());
}
```

EXAMPLE
STUDENTS GRADES

Typical exercise

- Compute the average grade

```
double avg(Map<Student,Integer> grades) {
   int sum = 0;
      for (Integer c: grades.values()) {
         sum += c;
      }
   return (double)sum / grades.size();
}
```

EXAMPLE
STUDENTS GRADES

Another typical exercise

- Compute the student with the best grade

```
Student best(Map<Student,Integer> grades) {
   Student result = null;
   int best = 0;
   for (Map.Entry<Student,Integer> e: grades.entrySet()) {
      Integer g = e.getValue();
      if (g > best) {
         result = e.getKey();
         best = g;
      }
   }
   return result;
}
```

This is the most efficient way to go through a `Map`!

STATIC METHODS AND TYPE PARAMETERS

- Type parameter is specified when constructor is called
  - For example, `Map<Student, Integer> grades = new HashMap<>();`
- How to use type parameter for other (e.g., static) methods?

```
public static <E> List<E> sort(List<E> list) {
  //...
}
```

Declare type parameter in header of method to specify that the same element type is used for argument and result value!

STATIC METHODS AND TYPE PARAMETERS

- It is possible to constrain the permitted types `E`, for example, to types that extend or implement `T` (`T` is a Class or an Interface, respectively)

```
public static <E extends T> List<E> sort(List<E> list) {
  //...
}
```

- Another example: require `E` to be a subtype of `Comparable<E>`

```
public static <E extends Comparable<E>> List<E> sort(List<E> list)
{
  //...
}
```

POLYMORPHISM AND LISTS

- Everything is an object, so, also lists
  - `Object o = new ArrayList<Student>();`
- But is a list of `Student`s also a list of `Object`s?
  - `List<Object> list = new ArrayList<Student>();`
- Or is the follwowing true?
  - `new ArrayList<Student>() instanceof List<Object>`

POLYMORPHISM AND LISTS

- No!
  - Relationship `X` is-a `Y` means that we can use `X` instead of `Y`
  - A list of `Object`s must allow adding arbitrary objects, e.g., `String`s
  - A list of `Student`s does not allow this!
- Workaround: use type parameter wildcard `?`
  - Out-of-scope for this course...

TAKE HOME MESSAGES

- Java Collection Framework has many useful interfaces and classes for manipulating popular data structures (collections, sets, lists, maps)
- These classes form a hierarchy (reuse!)
- Interfaces and implementations are properly separated (maintainability and reuse!)