Generics and Collections

OOP Lecture 5

Sjaak Smetsers

This week

- Looking back: OO concepts
- Java generics
- Collections

OO concepts

- *encapsulation*: information hiding (private properties/behaviour)
- *realization*: `implements` (interfaces)
- *composition*: has-a (fields)
- *inheritance*: is-a, `extends (base and derived classes)`
- *polymorphism*: method overriding (different from *overloading*)
- *types*: type = set of values
  - primitive/reference types
  - subtypes
  - static (compile-time)/dynamic (run-time) types

Collections: ArrayLists

Introduction to Collections: class `ArrayList`

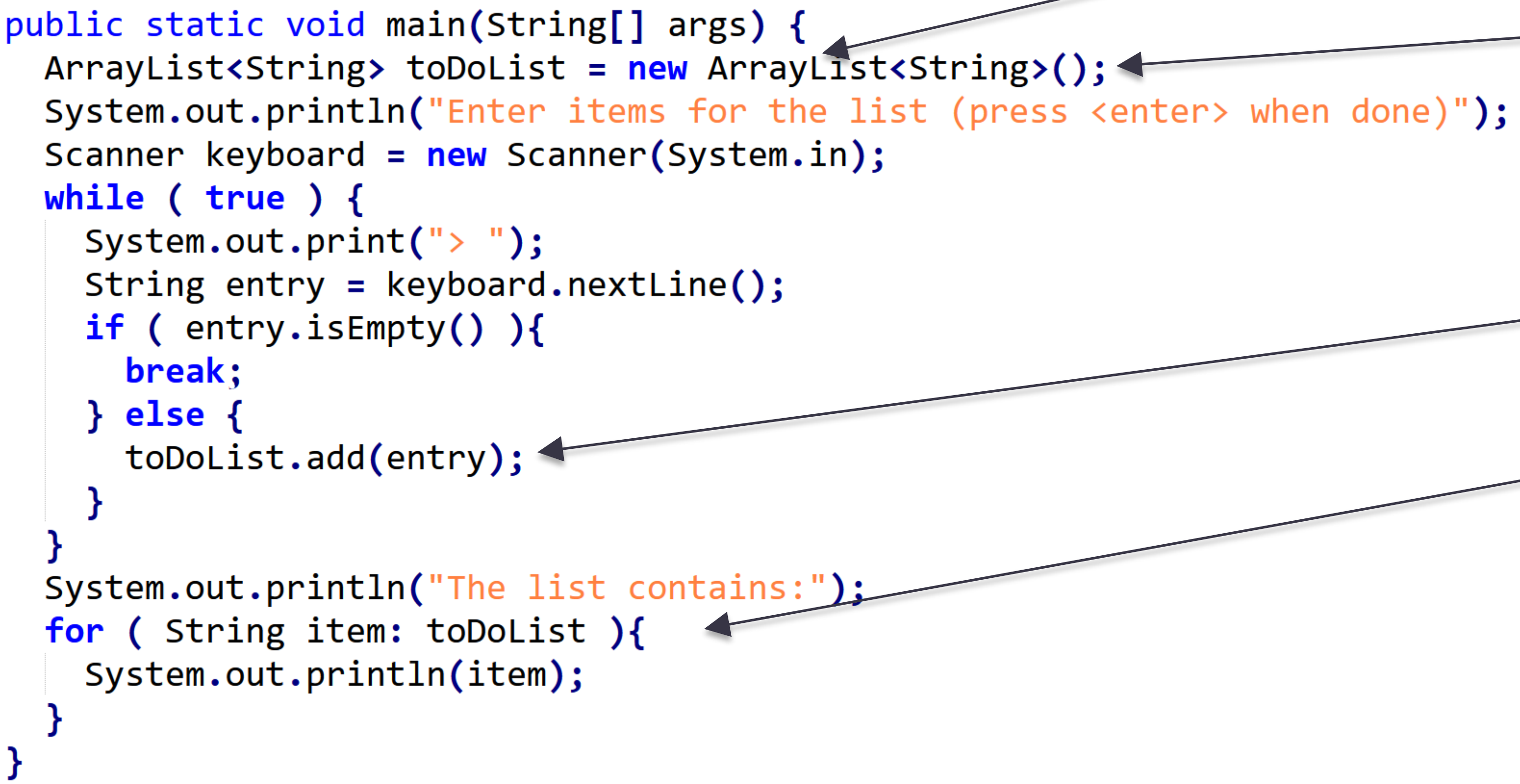
- Java API provides several predefined data structures, called collections, used to store groups of *homogeneous objects*.
  - Each provides efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
- Ordinary arrays do not automatically change their size at execution time to accommodate additional elements.
- `ArrayList<T>` (package `java.util`) can dynamically change its size to accommodate more elements.
  - T is a placeholder for the *type of element* stored in the collection.
- Classes with this kind of placeholder that can be used with any type are called generic classes (more about this later).

ArrayList methods

add(**value**)	appends value at end of list
add(**index, value**)	inserts given value at the specified position, shifting subsequent values to the right
clear()	removes all elements of the list
indexOf(**value**)	returns first index where given value is found in list (-1 if not found)
get(**index**)	returns the value at given index
remove(**index**)	removes/returns value at given index, shifting subsequent values to the left
set(**index, value**)	replaces value at given index with given value
size()	returns the number of elements in list
toString()	returns a string representation of the list such as `"[3, 42, -7, 15]"`

Programming Example

- A To-Do List maintaining a list of everyday tasks
    - User enters as many as desired
    - Program displays the list



```java
public static void main(String[] args) {
  ArrayList<String> toDoList = new ArrayList<String>();
  System.out.println("Enter items for the list (press <enter> when done)");
  Scanner keyboard = new Scanner(System.in);
  while ( true ) {
    System.out.print("> ");
    String entry = keyboard.nextLine();
    if ( entry.isEmpty() ){
      break;
    } else {
      toDoList.add(entry);
    }
  }
  System.out.println("The list contains:");
  for ( String item: toDoList ){
    System.out.println(item);
  }
}
```

Constructor

`new ArrayList<>();` is also allowed

adding an entry

enhanced for-loop

`ArrayList` limitations

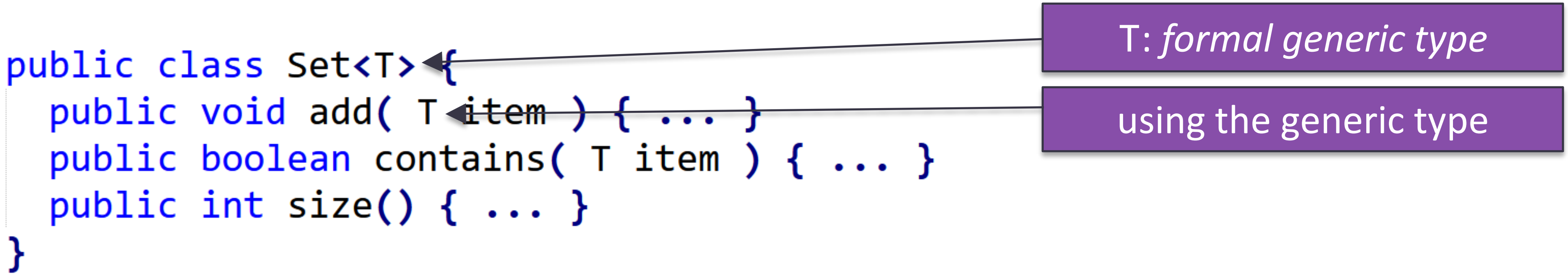
- An `ArrayList`’s *capacity* indicates how many items it can hold without growing.
  - When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array.
    - This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added.
    - An `ArrayList` grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.
- Adding to (or removing from) the end is cheap; but expensive elsewhere

Generics

Generic classes/interfaces

- A generic class (or interface) is a class with one or more type variables/placeholders as parameter.
  - These type variables ar called *generic types*
  - `ArrayList<E>` is an example. E is (formal) generic type.
- A generic (static or non-static) method is a method with one or more generic types

Generic classes: how to define?



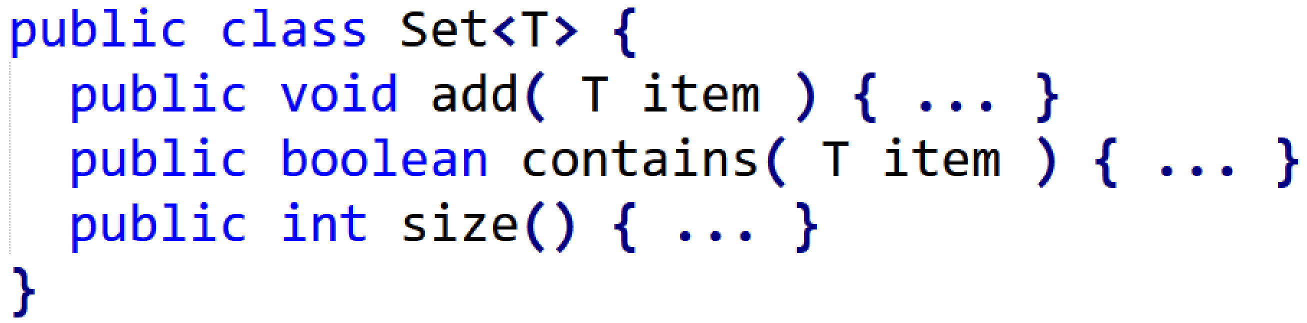
```
public class Set<T> {
  public void add( T item ) { ... }
  public boolean contains( T item ) { ... }
  public int size() { ... }
}
```

T: *formal generic type*

using the generic type

- Naming convention:
  - usually one single uppercase letter, being the first letter of a suitable name for the parameter
  - here: T for type

Generic classes: how to use?



```
public class Set<T> {
  public void add( T item ) { ... }
  public boolean contains( T item ) { ... }
  public int size() { ... }
}
```

```
Set<String> members = new Set<>();
members.add("Sjaak");
// members.add(24); is rejected
```

Every occurrence of the generic variable T in the `Set` class is replaced by `String`

Generic sorting

```
public static void main(String[] args) {
  ArrayList<String> items = new ArrayList<> (List.of("me", "myself","i"));
  Collections.sort(items);
  System.out.println(items);
}
```

Output: `[i, me, myself]`

```
public record Song( String title, String artist ) {}
```

```
public static void run(){
  ArrayList<Song> songs = new ArrayList<> ();
  songs.add(new Song("Fortnight", "Taylor Swift"));
  songs.add(new Song("Dance the Night", "Dua Lipa"));
  Collections.sort(songs);
  System.out.println(songs);
}
```

No output: it doesn't compile!

no suitable method for `sort(ArrayList<Song>)`

Intermezzo: records

A simple (data) class with only a couple of fields requires a lot of code

```
public class Song2 {
  private String title;
  private String artist;

  public Song2(String title, String artist) {
    this.title = title;
    this.artist = artist;
  }

  public String getTitle() { return title;}
  public String getArtist() { return artist;}

  @Override
  public int hashCode() { }
  @Override
  public boolean equals(Object obj) { }
  @Override
  public String toString() {}
}
```

Java offers an alternative: records

```
public record Song( String title, String artist ) {}
```

Some remarks

- records are immutable
- no "get" prefix for getters
  - just `song.artist()` i.s.o.
  - `song.getArtist()`

Generic sorting (II)

The `sort` method declaration:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Obviously, `sort` is a generic method, but what does the signature say?

- `<T extends Comparable<? super T>>:` only types that extend/implement Comparable are allowed

What is `Comparable`?

ignore this part for now

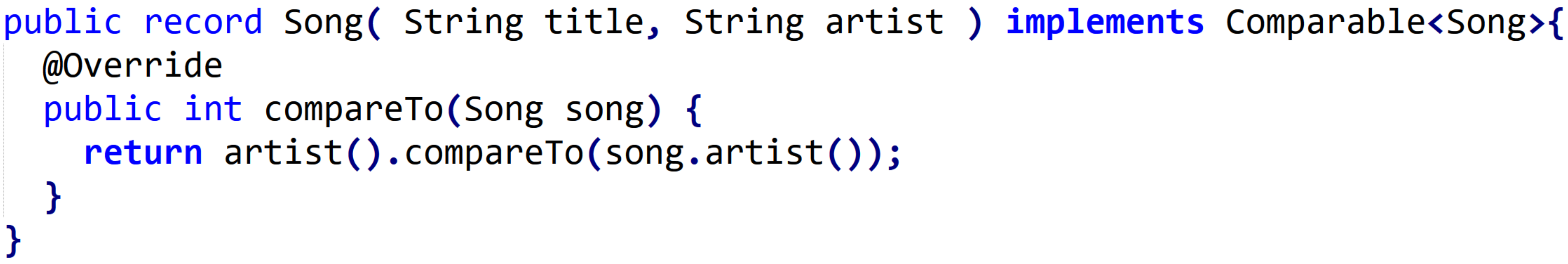
whatever T is must be of type `Comparable`

bounded generic type

(geneneric) Interface Comparable

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

In order to sort songs, the Song class/record must implement Comparable.



```
public record Song( String title, String artist ) implements Comparable<Song>{
  @Override
  public int compareTo(Song song) {
    return artist().compareTo(song.artist());
  }
}
```

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 an *collection of pairs*

1. update collection for each word in input
2. sort words
3. show counts

make the implementation more reusable/flexible

- many programs need pairs
  - often other types than String and int
- many programs need a Map
  - not restricted to Map from String to int
  - StudentNumber to Student
  - Zipcode to Address
- We can make classes more flexible by using generics

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; }
}
```

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

reusable pair (II)

- ... or using a record instead of a class

```
public record Pair<K, V> (K key, V val) {
}
```

allowed instances of generic type variable: any reference type

```
public record Student(String name, int num) {}
```

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

  System.out.println(pss.key());
  System.out.println(pss.val().num());
}
```

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

RUN

```
CS
42
```

allowed instances: what about primitive types?

- `Pair<int, Student> p3 = new Pair<>(8, 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<>(8, alice);
```

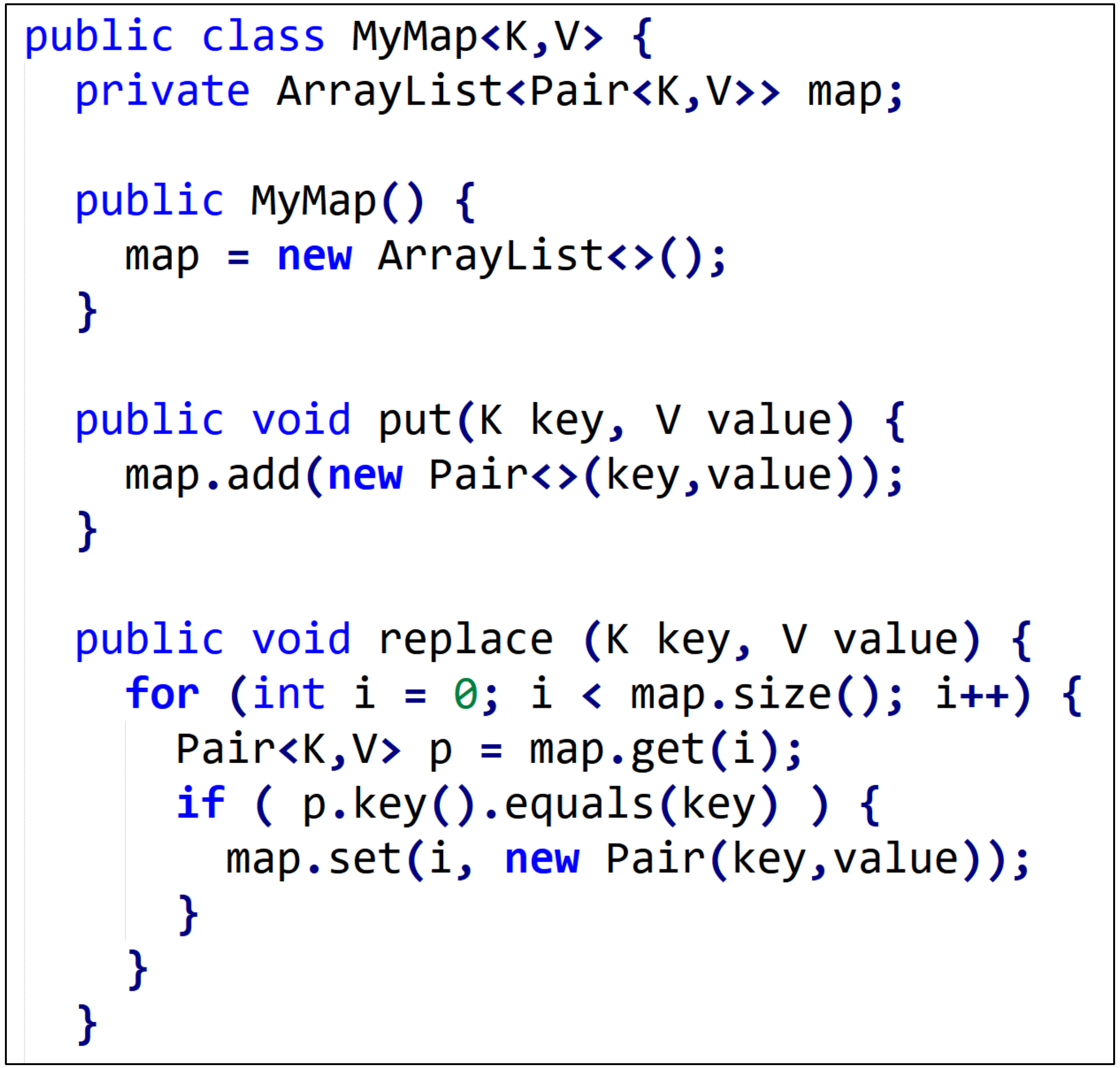
autoboxing / auto-unboxing: automatic conversion between primitive & wrapper

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

instead of

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

Our own map class: **MyMap**

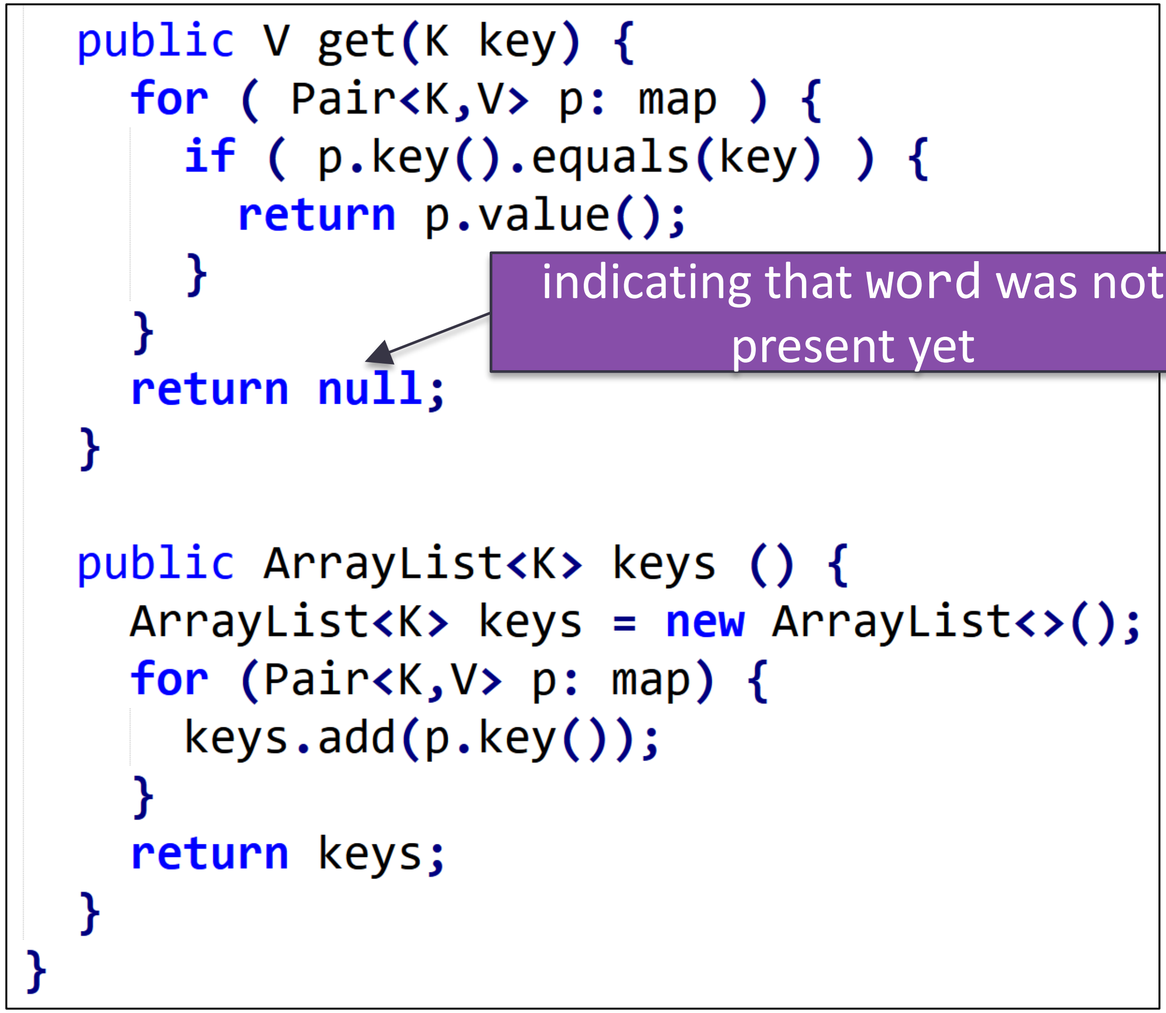


```java
public class MyMap<K,V> {
  private ArrayList<Pair<K,V>> map;

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

  public void put(K key, V value) {
    map.add(new Pair<>(key,value));
  }

  public void replace (K key, V value) {
    for (int i = 0; i < map.size(); i++) {
      Pair<K,V> p = map.get(i);
      if ( p.key().equals(key) ) {
        map.set(i, new Pair(key,value));
      }
    }
  }
```



```java
  public V get(K key) {
    for ( Pair<K,V> p: map ) {
      if ( p.key().equals(key) ) {
        return p.value();
      }
    }
    return null;
  }

  public ArrayList<K> keys () {
    ArrayList<K> keys = new ArrayList<>();
    for (Pair<K,V> p: map) {
      keys.add(p.key());
    }
    return keys;
  }
}
```

indicating that word was not present yet

Using MyMap to determine word frequencies

```
public static void main(String[] args) {
  run( "To be or not to be - that is the question!");
}

private static void run( String line ){
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  MyMap<String,Integer> map = new MyMap<>();
  while (scan.hasNext()) {
    String nextString = scan.next().toLowerCase();
    Integer val = map.get(nextString);
    if ( val == null ) {
      map.put(nextString, 1);
    } else {
      map.replace(nextString, val + 1);
    }
  }
  var keys = map.keys();
  Collections.sort(keys);
  for (String key : keys) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

one or more “non-word characters”: see IJPDS 12.11.4

RUN

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

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 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.value(), p.key());
}
```

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 generics

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 !

Wild card generic types

- Intro: `Number` is a superclass for the (boxed) numeric classes: `Integer`, `Double`, `BigInteger`, …
- Calculate the total of a List of numbers

```java
public static double sum(List<Number> numbers){
  double total = 0;
  for ( Number nextNumber: numbers ){
    total += nextNumber.doubleValue();
  }
  return total;
}
```

- Using sum

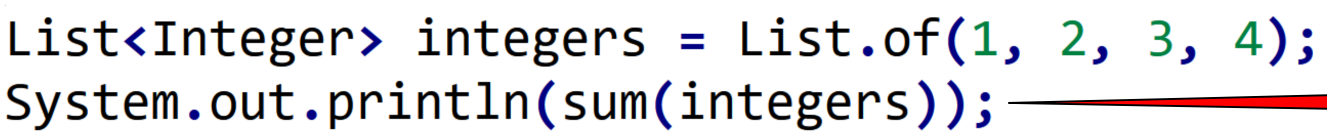
```java
private static void run() {
  List<Number> numbers   = List.of(1, 2.4, 3, 4.1);
  List<Integer> integers = List.of(1, 2, 3, 4);

  System.out.println(sum(numbers));
  System.out.println(sum(integers));
}
```

fine

this is **NOT** allowed !

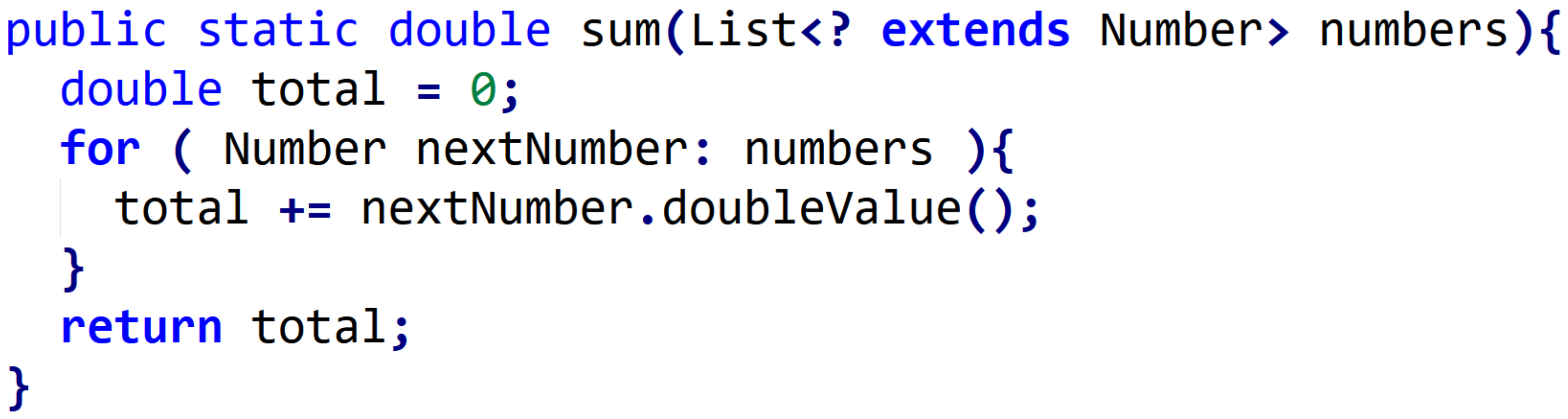
Wild card generic types (II)



```java
List<Integer> integers = List.of(1, 2, 3, 4);
System.out.println(sum(integers));
```

NOT allowed !

- Reason: `List<Integer>` is not a subtype of `List<Number>`
- Solution: use a *wild card generic type* (denoted by a question mark ?)



```java
public static double sum(List<? extends Number> numbers){
  double total = 0;
  for ( Number nextNumber: numbers ){
    total += nextNumber.doubleValue();
  }
  return total;
}
```

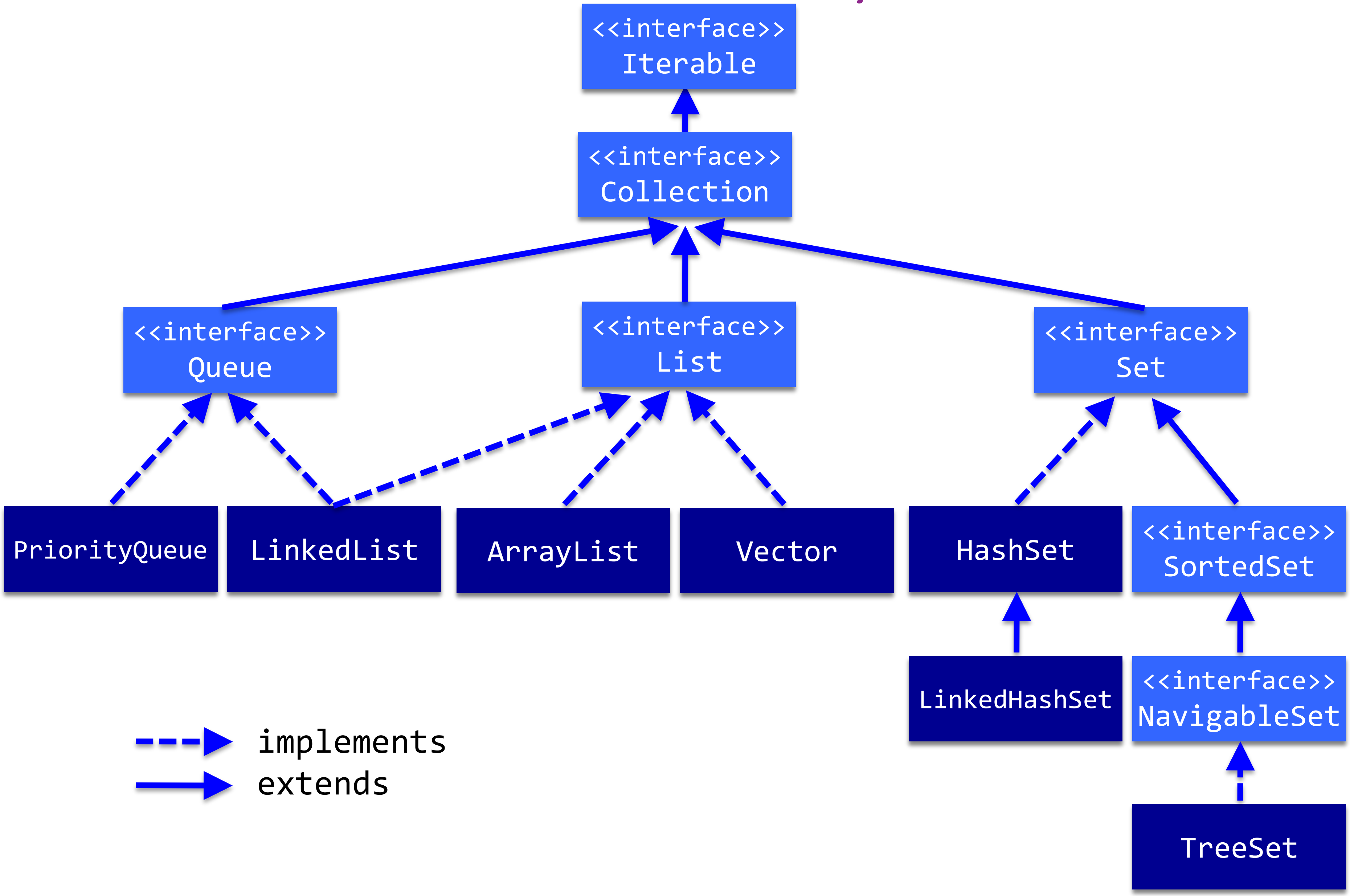
- `List<? extends Number>`: should be read as: the element type can by either a `Number` or a subclass of `Number`

More collections

Java collection framework

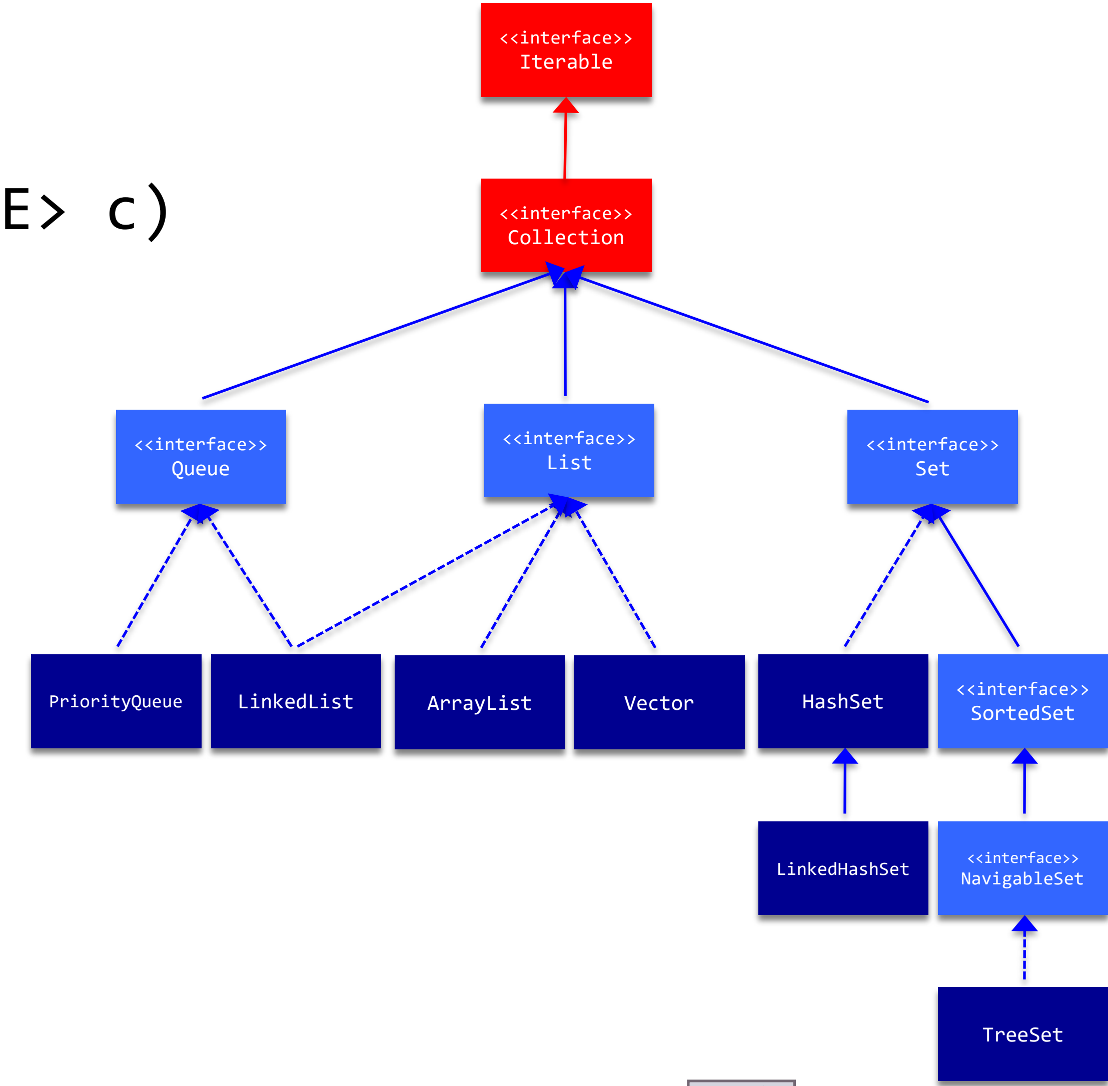
- In the JDK:
  - a set of **interfaces** modeling key concepts
    - with operations (and some "implicit" properties)
  - a set of **classes** implementing those interfaces
- Main interfaces
  - `interface Collection<E>`
    - A collection represents a group of objects, known as its elements
    - E stands for element
  - `interface Map<K,V>`
    - An object that maps keys to values.
    - A map cannot contain duplicate keys; each key can map to at most one value.
    - K stands for key, V stands for value

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 isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int     size()
```



Program to an interface, not an implementation

- Instead of

```
public class MyMap<K, V> {
  private final ArrayList<Pair<K,V>> mapData;
  ...
}
```

- use

Collection iso ArrayList

```
public class MyMap<K, V> {
  private final Collection<Pair<K,V>> mapData;
  ...
}
```

Program to an interface … (2)

List iso ArrayList

```java
private static void run(  ) {
  MyMap<String,List<Integer>> oopResults = new MyMap<>();
  oopResults.put("Sjaak", List.of(6,6,7));
  oopResults.put("Sebastian", List.of(9,9,8));
  for (String key: oopResults.keys()) {
    System.out.println(key + ": " + oopResults.get(key));
  }
}
```

RUN

```
Sebastian: [9, 9, 8]
Sjaak: [6, 6, 7]
```

Iterators

The iterator pattern

More about design patterns in week 10

- *Iterator* lets you traverse elements of a collection without exposing its underlying representation
- An iterator offers a standard way to scan and handle all elements of a collection
  - `Iterator` is an interface
  - Every collection provides a *factory method* called `iterator` that creates an `Iterator` object.
  - the class implementing this interface mostly remains hidden
- The Iterator keeps track of the current element in a collection
- There are methods to advance to the next element and to delete the current element from a collection

Iterator interfaces

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

E: generic type of the elements

is there a next object?

yield next object; advance iterator one position

remove last returned object

optional operation, can throw a `NotImplementedException`

```
interface Iterable<E> {
  Iterator<E> iterator()
}
```

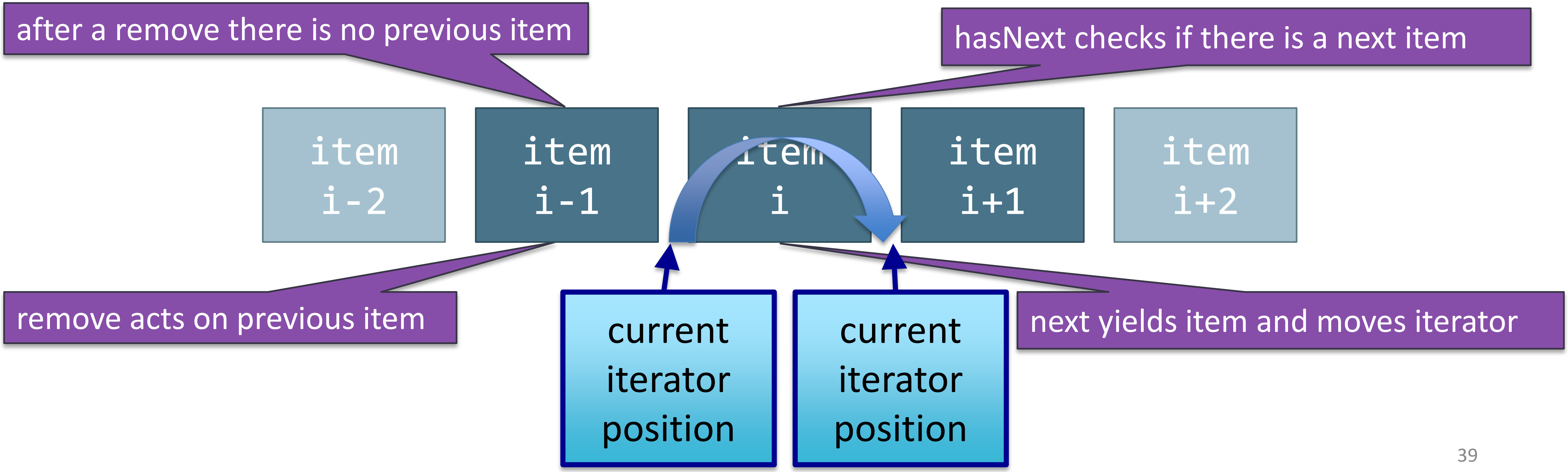
(factory) method for creating an iterator over elements of type E

`interface Collection<E> extends Iterable<E>`

`Iterator` interface

an iterator is conceptually between elements;

- it does not refer to a particular object



after a remove there is no previous item

hasNext checks if there is a next item

remove acts on previous item

next yields item and moves iterator

Iterator example

```
public class MyMap<K,V> {
  private Collection<Pair<K,V>> map;

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

  public void put(K key, V value) {
    map.add(new Pair<>(key,value));
  }

  public boolean replace (K key, V value) {
    boolean contains = false;
    Iterator<Pair<K,V>> mapIt = map.iterator();
    while ( mapIt.hasNext() && ! contains ) {
      Pair<K,V> p = mapIt.next();
      if ( p.key().equals(key) ) {
        mapIt.remove();
        contains = true;
      }
    }
    put(key, value);
    return contains;
  }
}
```

creates an iterator for map

is there another element?

get next element

remove element returned by next from map

NEXT WEEK

Lecture 6: Collections continued