Programming in Java – Day 9 Recap

Java Collections, wildcards

School of Computing and Mathematical Sciences
Birkbeck, University of London

Topics on Day 9

- Java Collections.
- Maps.
- Type wildcards.

Java Collections

- Java Collections Framework was introduced with Java 1.2.

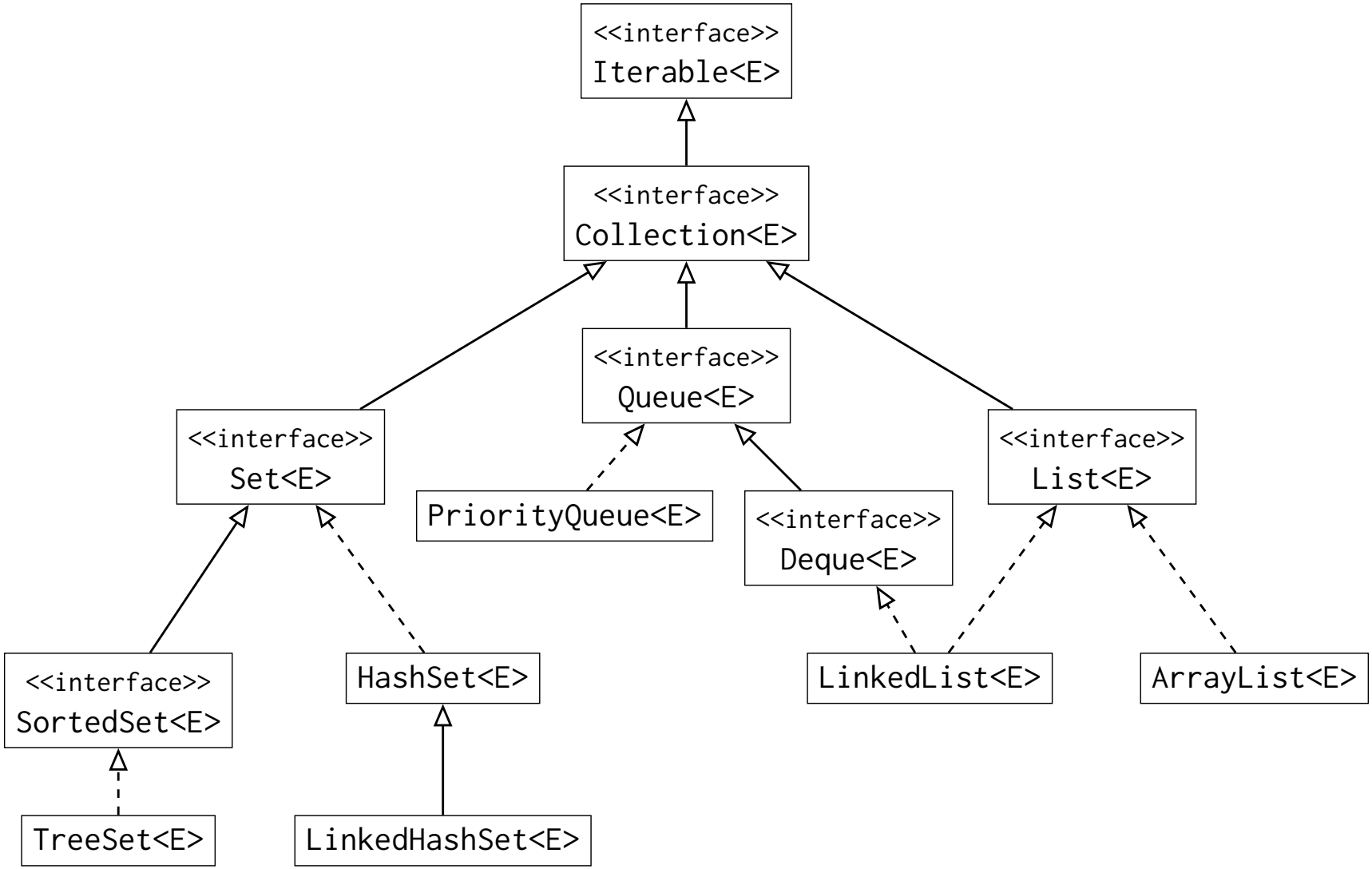
Java Collections

- Java Collections Framework was introduced with Java 1.2.
- It provides collections and maps to store elements grouped together.

Java Collections

- Java Collections Framework was introduced with Java 1.2.
- It provides collections and maps to store elements grouped together.
- It sets some requirements on the elements that are worth knowing.

Inheritance Relations in the Collections Framework



Let's walk through these ...

Interface Iterable<E>

```
1 public interface Iterable<E> {
2     // give me an iterator over this object
3     Iterator<E> iterator();
4     ...
5 }
```

Interface Iterable<E>

```
1 public interface Iterable<E> {
2     // give me an iterator over this object
3     Iterator<E> iterator();
4     ...
5 }
```

Ok, and what's an Iterator<E>?

Interface Iterable<E>

```
1 public interface Iterable<E> {
2     // give me an iterator over this object
3     Iterator<E> iterator();
4     ...
5 }
```

Ok, and what's an Iterator<E>?

Iterators allow us to access the elements in a container data structure, one by one.

```
1 public interface Iterator<E> {
2     boolean hasNext(); // are there elements left to visit?
3     E next();          // give me the next element
4     void remove();     // remove last visited element
5     ...
6 }
```

Interface Iterator<E> I

Standard use until Java 1.4:

```
1 // Assuming there is a collection (list, set, etc) of MyClass
2 // called "elements"
3 for (Iterator<MyClass> iter = elements.iterator(); iter.hasNext(); ) {
4     MyClass next = iter.next();
5     next.doSomething();
6 }
```

Interface Iterator<E> I

Standard use until Java 1.4:

```
1 // Assuming there is a collection (list, set, etc) of MyClass
2 // called "elements"
3 for (Iterator<MyClass> iter = elements.iterator(); iter.hasNext(); ) {
4     MyClass next = iter.next();
5     next.doSomething();
6 }
```

Since Java 1.5:

used implicitly in **enhanced for loop over Iterable<E>**

(e.g., ArrayList<E>) as shorthand:

```
1 // Assuming there is a collection (list, set, etc) of MyClass
2 // called "elements"
3 for (MyClass next : elements) {
4     next.doSomething();
5 }
```

Side note: enhanced **for** loop for arrays

Given: `Foo[] myArray`.

Side note: enhanced `for` loop for arrays

Given: `Foo[] myArray`.

Instead of

```
1         for (int i = 0; i < myArray.length; i++) {
2             Foo e = myArray[i];
3             // do something with e
4         }
```

Side note: enhanced `for` loop for arrays

Given: `Foo[] myArray`.

Instead of

```
1         for (int i = 0; i < myArray.length; i++) {
2             Foo e = myArray[i];
3             // do something with e
4         }
```

we can use the enhanced `for` loop:

```
1         for (Foo e : myArray) {
2             // do something with e
3         }
```

Side note: enhanced `for` loop for arrays

Given: `Foo[] myArray`.

Instead of

```
1         for (int i = 0; i < myArray.length; i++) {
2             Foo e = myArray[i];
3             // do something with e
4         }
```

we can use the enhanced `for` loop:

```
1         for (Foo e : myArray) {
2             // do something with e
3         }
```

Handy if:

- We want to visit all elements of `myArray` in the order index 0, 1, 2, ..., `myArray.length - 1`.

Side note: enhanced `for` loop for arrays

Given: `Foo[] myArray`.

Instead of

```
1         for (int i = 0; i < myArray.length; i++) {
2             Foo e = myArray[i];
3             // do something with e
4         }
```

we can use the enhanced `for` loop:

```
1         for (Foo e : myArray) {
2             // do something with e
3         }
```

Handy if:

- We want to visit all elements of `myArray` in the order index 0, 1, 2, ..., `myArray.length - 1`.
- We do not need the current value of the index `i` in the loop.

Side note: enhanced `for` loop for arrays

Given: `Foo[] myArray`.

Instead of

```
1         for (int i = 0; i < myArray.length; i++) {
2             Foo e = myArray[i];
3             // do something with e
4         }
```

we can use the enhanced `for` loop:

```
1         for (Foo e : myArray) {
2             // do something with e
3         }
```

Handy if:

- We want to visit all elements of `myArray` in the order index 0, 1, 2, ..., `myArray.length - 1`.
- We do not need the current value of the index `i` in the loop.
- We do not want to write into `myArray` in the loop.

Interface `Iterator<E>` II

...back to `Iterator<E>`!

Interface Iterator<E> II

...back to Iterator<E>!

Can also **remove** elements via an iterator:

```
1  // Assuming there is a collection (list, set, etc) of MyClass
2  // called "elements"
3  for (Iterator<MyClass> iter = elements.iterator(); iter.hasNext(); ) {
4      MyClass next = iter.next();
5
6      // Assuming we have a method shouldBeRemoved(MyClass)
7      // that returns whether we want to remove an element
8      if (shouldBeRemoved(next)) {
9          iter.remove();
10
11         // call iter.remove() only after iter.next(),
12         // but not more than once
13     }
14 }
```

Interface Iterator<E> III

Iterators are sensitive to modifications of their underlying container:

```
1 List<String> container = new ArrayList<>(Arrays.asList("A", "B"));
2 for (String s : container) { // uses Iterator over container to get s
3     container.add("C");      // modifies container
4 }
```

Interface Iterator<E> III

Iterators are sensitive to modifications of their underlying container:

```
1 List<String> container = new ArrayList<>(Arrays.asList("A", "B"));
2 for (String s : container) { // uses Iterator over container to get s
3     container.add("C");      // modifies container
4 }
```

- A ConcurrentModificationException is thrown when we try to get the next element from container after container.add("C"); was executed.

Interface Iterator<E> III

Iterators are sensitive to modifications of their underlying container:

```
1 List<String> container = new ArrayList<>(Arrays.asList("A", "B"));
2 for (String s : container) { // uses Iterator over container to get s
3     container.add("C");      // modifies container
4 }
```

- A ConcurrentModificationException is thrown when we try to get the next element from container after container.add("C"); was executed.
- Similar:

```
3 Iterator<String> iter = container.iterator();
4 container.add("C");
5 if (iter.hasNext()) {
6     String s = iter.next(); // ConcurrentModificationException
7 }
```

Interface Iterator<E> III

Iterators are sensitive to modifications of their underlying container:

```
1 List<String> container = new ArrayList<>(Arrays.asList("A", "B"));
2 for (String s : container) { // uses Iterator over container to get s
3     container.add("C");      // modifies container
4 }
```

- A `ConcurrentModificationException` is thrown when we try to get the next element from `container` after `container.add("C");` was executed.
- Similar:

```
3 Iterator<String> iter = container.iterator();
4 container.add("C");
5 if (iter.hasNext()) {
6     String s = iter.next(); // ConcurrentModificationException
7 }
```

- When we are iterating over a collection, we are (usually) supposed to modify it **only** via the iterator.

Interface `Collection<E>` I

- Used to group together several elements.
- Methods to add and remove elements, to check if an element is present, to check a collection's size, to convert to an array.

Interface Collection<E> I

- Used to group together several elements.
- Methods to add and remove elements, to check if an element is present, to check a collection's size, to convert to an array.

```
1 public interface Collection<E> extends Iterable<E> {
2     boolean add(E e);
3     boolean addAll(Collection<? extends E> c);
```

Interface Collection<E> I

- Used to group together several elements.
- Methods to add and remove elements, to check if an element is present, to check a collection's size, to convert to an array.

```
1 public interface Collection<E> extends Iterable<E> {
2     boolean add(E e);
3     boolean addAll(Collection<? extends E> c);
4
5     boolean remove(Object o);
6     boolean removeAll(Collection<?> c);
7     boolean retainAll(Collection<?> c);
8     void clear();
```

Interface Collection<E> I

- Used to group together several elements.
- Methods to add and remove elements, to check if an element is present, to check a collection's size, to convert to an array.

```
1  public interface Collection<E> extends Iterable<E> {
2      boolean add(E e);
3      boolean addAll(Collection<? extends E> c);
4
5      boolean remove(Object o);
6      boolean removeAll(Collection<?> c);
7      boolean retainAll(Collection<?> c);
8      void clear();
9
10     boolean contains(Object o);
11     boolean containsAll(Collection<?> c);
```

Interface Collection<E> I

- Used to group together several elements.
- Methods to add and remove elements, to check if an element is present, to check a collection's size, to convert to an array.

```
1  public interface Collection<E> extends Iterable<E> {
2      boolean add(E e);
3      boolean addAll(Collection<? extends E> c);
4
5      boolean remove(Object o);
6      boolean removeAll(Collection<?> c);
7      boolean retainAll(Collection<?> c);
8      void clear();
9
10     boolean contains(Object o);
11     boolean containsAll(Collection<?> c);
12
13     boolean isEmpty();
14     int size();
```

Interface Collection<E> I

- Used to group together several elements.
- Methods to add and remove elements, to check if an element is present, to check a collection's size, to convert to an array.

```
1  public interface Collection<E> extends Iterable<E> {
2      boolean add(E e);
3      boolean addAll(Collection<? extends E> c);
4
5      boolean remove(Object o);
6      boolean removeAll(Collection<?> c);
7      boolean retainAll(Collection<?> c);
8      void clear();
9
10     boolean contains(Object o);
11     boolean containsAll(Collection<?> c);
12
13     boolean isEmpty();
14     int size();
15
16     Object[] toArray();
17     <T> T[] toArray(T[] a);
18     ...
19 }
```

Interface Collection<E> II

Two flavours of Collection<E>:

Interface Collection<E> II

Two flavours of Collection<E>:

List<E> are collections
with possible duplicate elements ([1] and [1,1] are **different**)
and an order ([1,2] and [2,1] are **different**).

Examples:
ArrayList<E>, LinkedList<E>.

Interface Collection<E> II

Two flavours of Collection<E>:

List<E> are collections
with possible duplicate elements ([1] and [1,1] are **different**)
and an order ([1,2] and [2,1] are **different**).

Examples:
ArrayList<E>, LinkedList<E>.

Set<E> are collections
without duplicate elements ([1] and [1,1] are **equal**)
whose order does not matter for equality ([1,2] and [2,1] are **equal**).

Examples:
HashSet<E>, LinkedHashSet<E>, TreeSet<E>.

Interface `List<E>`

Implementations of `List<E>`:

same behaviour, but optimised for different purposes.

Interface `List<E>`

Implementations of `List<E>`:

same behaviour, but optimised for different purposes.

`ArrayList<E>` uses an array internally, copying elements over to a larger array when needed. Accessing elements via `get` is **fast** (constant time), iteration efficient, adding elements at **the end** is efficient.

Interface `List<E>`

Implementations of `List<E>`:

same behaviour, but optimised for different purposes.

`ArrayList<E>`	uses an array internally, copying elements over to a larger array when needed. Accessing elements via `get` is **fast** (constant time), iteration efficient, adding elements at **the end** is efficient.
`LinkedList<E>`	uses a doubly-linked list internally. Accessing elements via `get` is **slow** (linear time), but iteration and adding elements at **both ends** is efficient (suitable for queues).

Interface `List<E>`

Implementations of `List<E>`:

same behaviour, but optimised for different purposes.

`ArrayList<E>`	uses an array internally, copying elements over to a larger array when needed. Accessing elements via `get` is **fast** (constant time), iteration efficient, adding elements at **the end** is efficient.
`LinkedList<E>`	uses a doubly-linked list internally. Accessing elements via `get` is **slow** (linear time), but iteration and adding elements at **both ends** is efficient (suitable for queues).

Caveat: for `ArrayList<E>` and `LinkedList<E>`, the `contains` method is **slow** (linear time).

Interface Queue<E> and Deque<E>

- Queue: First-In-First-Out data structure

```
1 Queue<String> myQueue = new LinkedList<>();
2 myQueue.add("Hello");
3 myQueue.add("World");
4 String x = myQueue.remove(); // "Hello"
5 String y = myQueue.remove(); // "World"
```

Interface Queue<E> and Deque<E>

- Queue: First-In-First-Out data structure

```
1 Queue<String> myQueue = new LinkedList<>();
2 myQueue.add("Hello");
3 myQueue.add("World");
4 String x = myQueue.remove(); // "Hello"
5 String y = myQueue.remove(); // "World"
```

- Deque: a "double-ended queue", can also be used as a stack (Last-In-First-Out data structure)

```
1 Deque<String> myDeque = new LinkedList<>();
2 myDeque.push("Hello");
3 myDeque.push("World");
4 String x = myDeque.pop(); // "World"
5 String y = myDeque.pop(); // "Hello"
```

Interface Queue<E> and Deque<E>

- Queue: First-In-First-Out data structure

```
1 Queue<String> myQueue = new LinkedList<>();
2 myQueue.add("Hello");
3 myQueue.add("World");
4 String x = myQueue.remove(); // "Hello"
5 String y = myQueue.remove(); // "World"
```

- Deque: a "double-ended queue", can also be used as a stack (Last-In-First-Out data structure)

```
1 Deque<String> myDeque = new LinkedList<>();
2 myDeque.push("Hello");
3 myDeque.push("World");
4 String x = myDeque.pop(); // "World"
5 String y = myDeque.pop(); // "Hello"
```

- Use `isEmpty()` to check if there is something to retrieve.

Interface Set<E> I

Set<E> corresponds to the mathematical concept of (finite) sets.

- We cannot tell how often an element is contained.

Interface Set<E> I

Set<E> corresponds to the mathematical concept of (finite) sets.

- We cannot tell how often an element is contained.
- We cannot distinguish between different orders of elements (two sets are equal if and only if they have the same size and contain the same elements).

Interface Set<E> I

Set<E> corresponds to the mathematical concept of (finite) sets.

- We cannot tell how often an element is contained.
- We cannot distinguish between different orders of elements (two sets are equal if and only if they have the same size and contain the same elements).
- Benefit: **Fast** `contains` check.

Interface Set<E> I

Set<E> corresponds to the mathematical concept of (finite) sets.

- We cannot tell how often an element is contained.
- We cannot distinguish between different orders of elements (two sets are equal if and only if they have the same size and contain the same elements).
- Benefit: **Fast** `contains` check.
- Caveat: do not modify elements of a `Set` (for `equals`) while they are in the `Set` — **strange** things may happen to the `Set`.

Class HashSet<E>

- Hash table relies on using

  `int hashCode()`

  (from `Object`) as a **"summary"** of an object (4 bytes for any object).

Class HashSet<E>

- Hash table relies on using

  `int` hashCode()

  (from `Object`) as a **"summary"** of an object (4 bytes for any object).
- If hashCode for the elements is implemented well (different objects "often" get different hash codes), we can expect **constant-time** performance for add, `contains`, `remove`, `size`.

Class HashSet<E>

- Hash table relies on using

  `int` hashCode()

  (from `Object`) as a **"summary"** of an object (4 bytes for any object).
- If hashCode for the elements is implemented well (different objects "often" get different hash codes), we can expect **constant-time** performance for add, `contains`, `remove`, `size`.
- Requires from its elements: **whenever `x.equals(y)` holds, `x.hashCode() == y.hashCode()` must hold, too.**

Class HashSet<E>

- Hash table relies on using

  `int` hashCode()

  (from `Object`) as a **"summary"** of an object (4 bytes for any object).
- If hashCode for the elements is implemented well (different objects "often" get different hash codes), we can expect **constant-time** performance for add, `contains`, `remove`, `size`.
- Requires from its elements: **whenever `x.equals(y)` holds, `x.hashCode() == y.hashCode()` must hold, too.**
- Already required by the contract of `int` hashCode() in `Object`.

Class HashSet<E>

- Hash table relies on using

  `int hashCode()`

  (from `Object`) as a **"summary"** of an object (4 bytes for any object).
- If hashCode for the elements is implemented well (different objects "often" get different hash codes), we can expect **constant-time** performance for add, `contains`, `remove`, `size`.
- Requires from its elements: **whenever `x.equals(y)` holds, `x.hashCode() == y.hashCode()` must hold, too.**
- Already required by the contract of `int hashCode()` in `Object`.
- Holds for `Object`, but when we override `boolean equals(Object)`, we must take care to also override `int hashCode()` accordingly so that the HashSet gives correct answers.

Class HashSet<E>

- Hash table relies on using
  `int hashCode()`
  (from `Object`) as a **"summary"** of an object (4 bytes for any object).
- If hashCode for the elements is implemented well (different objects "often" get different hash codes), we can expect **constant-time** performance for add, `contains`, `remove`, `size`.
- Requires from its elements: **whenever `x.equals(y)` holds, `x.hashCode() == y.hashCode()` must hold, too.**
- Already required by the contract of `int hashCode()` in `Object`.
- Holds for `Object`, but when we override `boolean equals(Object)`, we must take care to also override `int hashCode()` accordingly so that the HashSet gives correct answers.
- Caveat: iteration over HashSet<E> in **chaotic order** (may change for different iterations), inconvenient for users and debugging.

Class LinkedHashSet<E>

Improved subclass of HashSet<E> (since Java 1.4).

- Like HashSet<E>, but iteration over elements is
  in the **same order** as the (first) insertions of the elements.
- LinkedHashSet<E> internally uses an additional linked list between the elements, which hardly affects performance.
- LinkedHashSet<E> usually the better choice over HashSet<E>
  (unless you know for sure that no one will ever iterate over elements).

Interface SortedSet<E> I

Extension of Set<E> with an additional order imposed by its Comparable<E> elements (or a Comparator<? super E>).

Interface SortedSet<E> I

Extension of Set<E> with an additional order imposed by its Comparable<E> elements (or a Comparator<? super E>).

```
1 public interface Comparable<E> {
2     int compareTo(E o); // negative if this object is less than o
3 }
```

is implemented by String, Integer, etc. It's the **"natural"** order.

Interface SortedSet<E> I

Extension of Set<E> with an additional order imposed by its Comparable<E> elements (or a Comparator<? super E>).

```
1 public interface Comparable<E> {
2     int compareTo(E o); // negative if this object is less than o
3 }
```

is implemented by String, Integer, etc. It's the **"natural"** order.

We can use

```
1 public interface Comparator<E> {
2     int compare(E o1, E o2); // negative if o1 is less than o2
3     ...// many useful static and default methods - see Java API
4 }
```

(in java.util package) to provide an **alternative** order,

Interface SortedSet<E> I

Extension of Set<E> with an additional order imposed by its Comparable<E> elements (or a Comparator<? super E>).

```
1 public interface Comparable<E> {
2     int compareTo(E o); // negative if this object is less than o
3 }
```

is implemented by String, Integer, etc. It's the **"natural"** order.

We can use

```
1 public interface Comparator<E> {
2     int compare(E o1, E o2); // negative if o1 is less than o2
3     ...// many useful static and default methods - see Java API
4 }
```

(in java.util package) to provide an **alternative** order, e.g.,

```
1 public class MyComparator implements java.util.Comparator<Integer>() {
2     public int compare(Integer o1, Integer o2) {
3         return o2 - o1; // descending order
4     }
5 }
```

Interface SortedSet<E> II

- SortedSet<E> orders its elements from smallest to largest, also for iteration.

Interface SortedSet<E> II

- SortedSet<E> orders its elements from smallest to largest, also for iteration.
- A method for getting a subset view of elements e with x $\leq e <$ y:
  `SortedSet<E> mySubSet = mySortedSet.subSet(x,y);`

Interface SortedSet<E> II

- SortedSet<E> orders its elements from smallest to largest,
  also for iteration.
- A method for getting a subset view of elements e with x $\leq e <$ y:
  `SortedSet<E> mySubSet = mySortedSet.subSet(x,y);`
- The implementations normally have the following constructors:
  - no parameters — the default (natural) order on elements is used
    (so, E must implement Comparable<E>);
  - with a Comparator<? super E> provided —
    the provided order is used.

Interface SortedSet<E> II

- SortedSet<E> orders its elements from smallest to largest, also for iteration.
- A method for getting a subset view of elements e with $x \leq e < y$:
  `SortedSet<E> mySubSet = mySortedSet.subSet(x,y);`
- The implementations normally have the following constructors:
  - no parameters — the default (natural) order on elements is used (so, E must implement Comparable<E>);
  - with a Comparator<? super E> provided — the provided order is used.
- The order should be consistent with equals: for all x, y $\neq$ **null**,

Interface SortedSet<E> II

- SortedSet<E> orders its elements from smallest to largest, also for iteration.
- A method for getting a subset view of elements e with $x \leq e < y$:
  `SortedSet<E> mySubSet = mySortedSet.subSet(x,y);`
- The implementations normally have the following constructors:
  - no parameters — the default (natural) order on elements is used (so, E must implement Comparable<E>);
  - with a Comparator<? super E> provided — the provided order is used.
- The order should be consistent with equals: for all x, y $\neq$ **null**,

x.compareTo(y) == 0 if and only if x.equals(y)

Interface SortedSet<E> II

- SortedSet<E> orders its elements from smallest to largest, also for iteration.
- A method for getting a subset view of elements e with $x \leq e < y$:
  `SortedSet<E> mySubSet = mySortedSet.subSet(x,y);`
- The implementations normally have the following constructors:
  - no parameters — the default (natural) order on elements is used (so, E must implement Comparable<E>);
  - with a Comparator<? super E> provided — the provided order is used.
- The order should be consistent with equals: for all x, y $\neq$ **null**,

x.compareTo(y) == 0 if and only if x.equals(y) or
comparator.compare(x,y) == 0 if and only if x.equals(y).

Class TreeSet<E>

Interface SortedSet<E> implemented by TreeSet<E>.

- Internally uses a "red-black tree" (a "binary search tree").

Class TreeSet<E>

Interface SortedSet<E> implemented by TreeSet<E>.

- Internally uses a "red-black tree" (a "binary search tree").
- Needs **logarithmic** time for add, contains, remove
  (better than linear, worse than constant).

Class TreeSet<E>

Interface SortedSet<E> implemented by TreeSet<E>.

- Internally uses a "red-black tree" (a "binary search tree").
- Needs **logarithmic** time for add, contains, remove
  (better than linear, worse than constant).
- So, unless you need the order (and automatic sorting),
  usually LinkedHashSet<E> is the Set<E> you want.

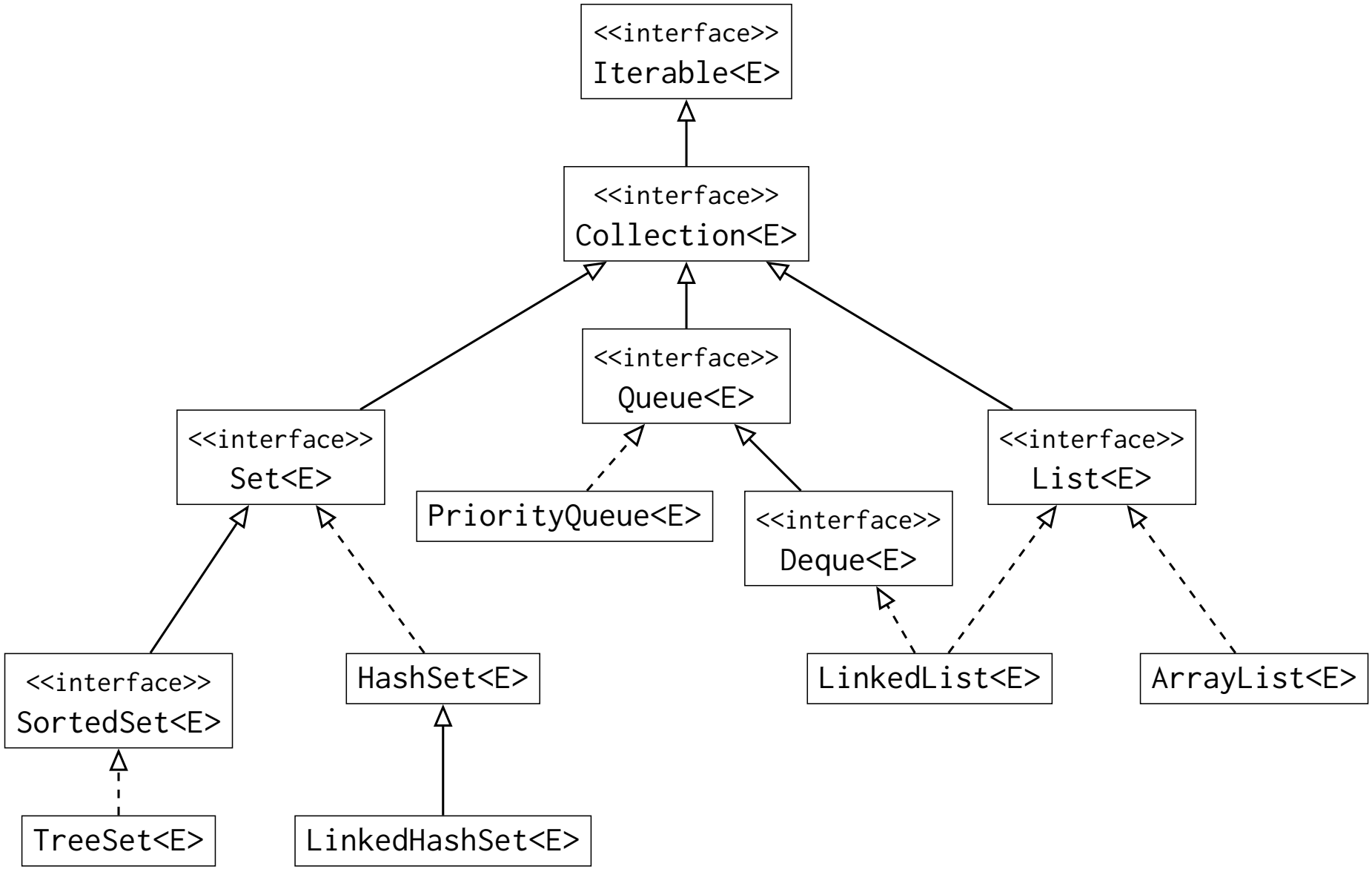
Class TreeSet<E>

Interface SortedSet<E> implemented by TreeSet<E>.

- Internally uses a "red-black tree" (a "binary search tree").
- Needs **logarithmic** time for add, contains, remove
  (better than linear, worse than constant).
- So, unless you need the order (and automatic sorting),
  usually LinkedHashSet<E> is the Set<E> you want.

Let's zoom out again ...

Inheritance Relations in the Collections Framework



Interface Map<K,V>

- Map<K,V> corresponds to the mathematical concept of a **function** / mapping that maps keys from K to values from V.

Interface Map<K,V>

- Map<K,V> corresponds to the mathematical concept of a **function** / mapping that maps keys from K to values from V.
- A Map<K,V> has a finite number of key-value pairs, **at most one per key**.
  For instance, Map<K,Integer> can store "How many articles do I have in stock?"

Interface Map<K,V>

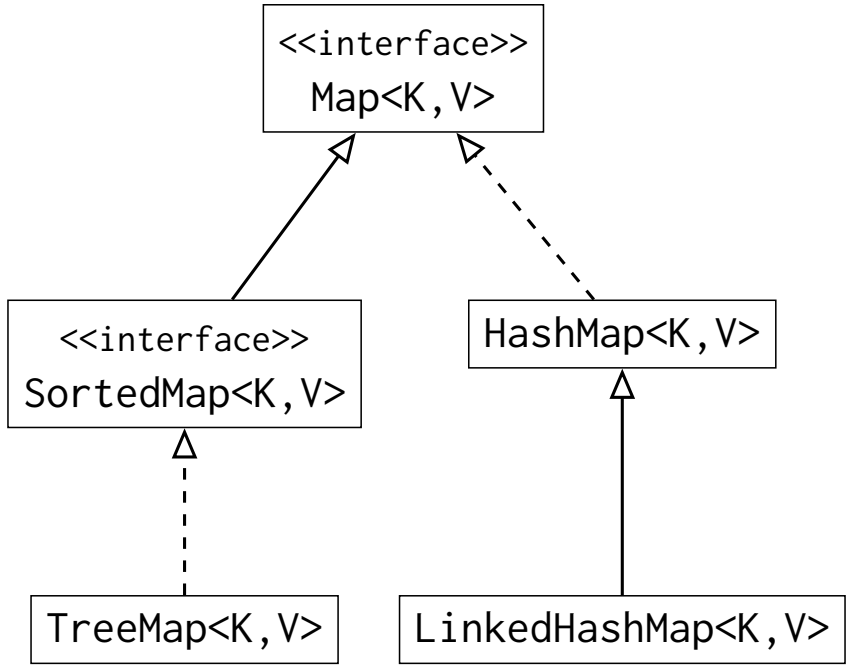
- Map<K,V> corresponds to the mathematical concept of a **function** / mapping that maps keys from K to values from V.
- A Map<K,V> has a finite number of key-value pairs, **at most one per key**.
  For instance, Map<K,Integer> can store "How many articles do I have in stock?"
- In other programming languages: "associative array", a "dictionary" (e.g., Python), a "hash" (Perl), an "array" (PHP), an "object" (JavaScript, JSON).

Interface Map<K,V>

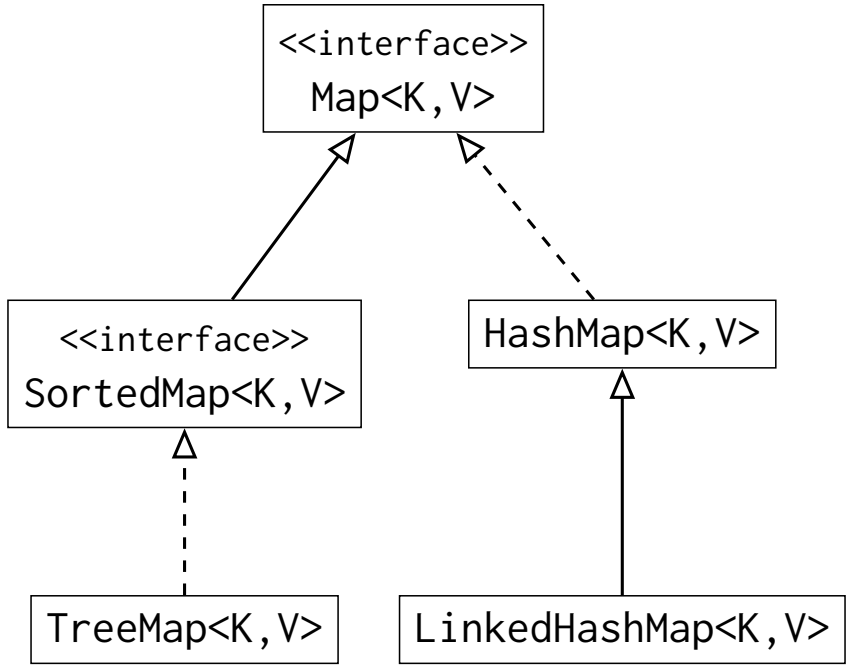
- Map<K,V> corresponds to the mathematical concept of a **function** / mapping that maps keys from K to values from V.
- A Map<K,V> has a finite number of key-value pairs, **at most one per key**.
  For instance, Map<K,Integer> can store "How many articles do I have in stock?"
- In other programming languages: "associative array", a "dictionary" (e.g., Python), a "hash" (Perl), an "array" (PHP), an "object" (JavaScript, JSON).

```
1  public interface Map<K,V> {
2      V get(Object key);      // return the value for key, or null
3      V put(K key, V value); // update so that key is mapped to value
4      V remove(Object key);  // remove key-value pair for key
5      ...
6      // collection views
7      Set<Map.Entry<K,V>> entrySet();
8      Set<K> keySet();
9      Collection<V> values();
10 }
```

Some Inheritance Relations on Maps

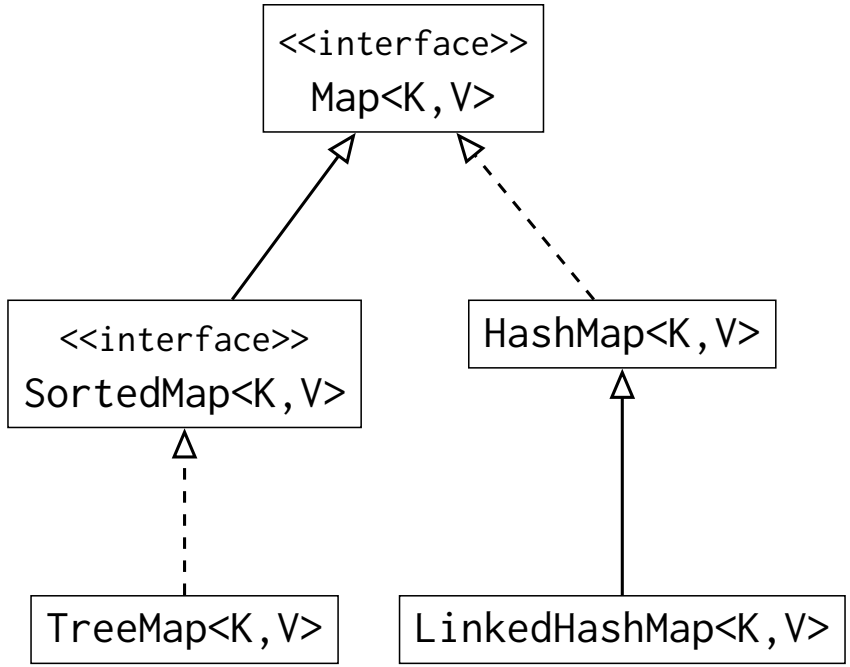


Some Inheritance Relations on Maps



Say, the ...Map<K,V> hierarchy looks like the ...Set<K> hierarchy!

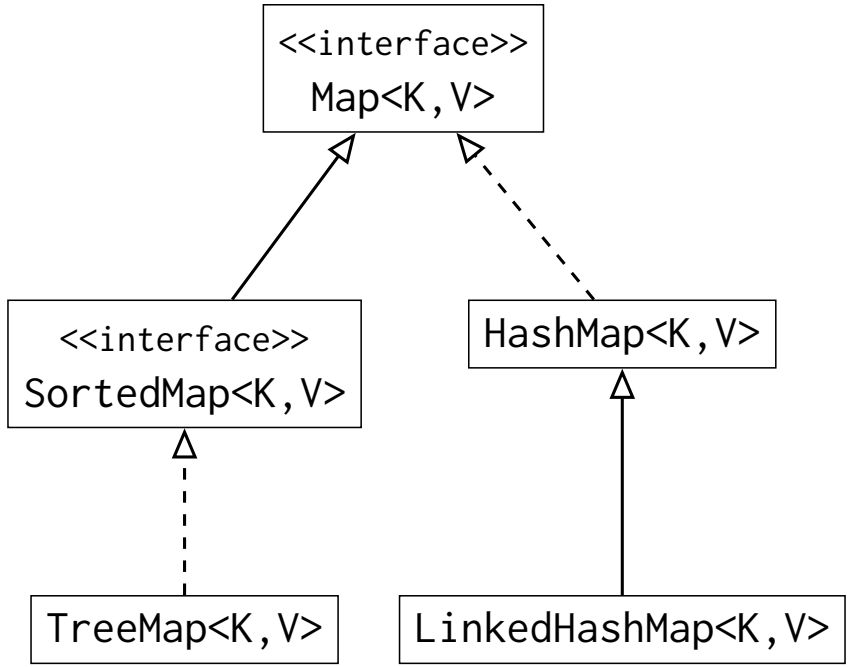
Some Inheritance Relations on Maps



Say, the ...Map<K,V> hierarchy looks like the ...Set<K> hierarchy!

That's no coincidence: the X-Set<K> classes that we've just seen are implemented via a X-Map<K,V> that maps to a dummy object. The map's keys have the role of the set elements.

Some Inheritance Relations on Maps



Say, the ...Map<K,V> hierarchy looks like the ...Set<K> hierarchy!

That's no coincidence: the X-Set<K> classes that we've just seen are implemented via a X-Map<K,V> that maps to a dummy object. The map's keys have the role of the set elements.

So, X-Map<K,V> has similar requirements on its keys as X-Set<K> has on its elements.

Helper class: `Collections`

Class `Collections` in package `java.util` contains many useful **`public static`** helper methods for sorting, checking that there are no common elements, ...

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html

“Cheat Sheet”: Avoiding Collection Pitfalls in Java

- Don’t modify a collection while you are iterating over it (except via the iterator itself).

“Cheat Sheet”: Avoiding Collection Pitfalls in Java

- Don’t modify a collection while you are iterating over it (except via the iterator itself).
- `(Linked)HashSet<E>` requires for `E`: if `x.equals(y)` holds, then `x.hashCode() == y.hashCode()` must hold, too.
- Holds for `equals` and `hashCode` in a new class (that inherits them from `Object`). But if you override `equals(Object o)`, do **remember** to override also `hashCode()`!

“Cheat Sheet”: Avoiding Collection Pitfalls in Java

- Don’t modify a collection while you are iterating over it (except via the iterator itself).
- `(Linked)HashSet<E>` requires for `E`: if `x.equals(y)` holds, then `x.hashCode() == y.hashCode()` must hold, too.
- Holds for `equals` and `hashCode` in a new class (that inherits them from `Object`). But if you override `equals(Object o)`, do **remember** to override also `hashCode()`!
- When in doubt between `HashSet` and `LinkedHashSet`: use `HashSet` only if you know that you (or code called by you, or your users) will **never** iterate over the set — `LinkedHashSet` (since Java 1.4) is almost always the better choice.

“Cheat Sheet”: Avoiding Collection Pitfalls in Java

- Don’t modify a collection while you are iterating over it (except via the iterator itself).
- `(Linked)HashSet<E>` requires for `E`: if `x.equals(y)` holds, then `x.hashCode() == y.hashCode()` must hold, too.
- Holds for `equals` and `hashCode` in a new class (that inherits them from `Object`). But if you override `equals(Object o)`, do **remember** to override also `hashCode()`!
- When in doubt between `HashSet` and `LinkedHashSet`: use `HashSet` only if you know that you (or code called by you, or your users) will **never** iterate over the set — `LinkedHashSet` (since Java 1.4) is almost always the better choice.
- `SortedSet<E>` needs an order over `E` that is consistent with `equals`: `x.compareTo(y) == 0` if and only if `x.equals(y)`.

"Cheat Sheet": Avoiding Collection Pitfalls in Java

- Don't modify a collection while you are iterating over it (except via the iterator itself).
- `(Linked)HashSet<E>` requires for `E`: if `x.equals(y)` holds, then `x.hashCode() == y.hashCode()` must hold, too.
- Holds for `equals` and `hashCode` in a new class (that inherits them from `Object`). But if you override `equals(Object o)`, do **remember** to override also `hashCode()`!
- When in doubt between `HashSet` and `LinkedHashSet`: use `HashSet` only if you know that you (or code called by you, or your users) will **never** iterate over the set — `LinkedHashSet` (since Java 1.4) is almost always the better choice.
- `SortedSet<E>` needs an order over `E` that is consistent with `equals`: `x.compareTo(y) == 0` if and only if `x.equals(y)`.
- Same considerations as for `X-Set<E>` also hold for `X-Map<E,V>` (and iteration over their `entrySet()`/`keySet()` views).

Type wildcards

Will the compiler accept this?

```
1 List<Animal> animals1 = new ArrayList<Animal>();
```

Type wildcards

Will the compiler accept this?

```
1 List<Animal> animals1 = new ArrayList<Animal>();
```

And this?

```
1 List<Animal> animals2 = new ArrayList<Dog>();
```

Type wildcards

Will the compiler accept this?

```
1 List<Animal> animals1 = new ArrayList<Animal>();
```

And this?

```
1 List<Animal> animals2 = new ArrayList<Dog>();
```

Problem:

```
1 List<Dog> dogs = new ArrayList<Dog>();
2 List<Animal> animals = dogs; // compiler says "no" here.
3 animals.add(new Cat()); // ok, Cat is a subclass of Animal
4 Dog d = dogs.get(0); // Type error! That's a Cat, not a Dog!
```

Type wildcards

Will the compiler accept this?

```
1 List<Animal> animals1 = new ArrayList<Animal>();
```

And this?

```
1 List<Animal> animals2 = new ArrayList<Dog>();
```

Problem:

```
1 List<Dog> dogs = new ArrayList<Dog>();
2 List<Animal> animals = dogs; // compiler says "no" here.
3 animals.add(new Cat()); // ok, Cat is a subclass of Animal
4 Dog d = dogs.get(0); // Type error! That's a Cat, not a Dog!
```

Solution:

```
1 List<? extends Animal> myAnimals = new ArrayList<Dog>();
```

Type wildcards

Will the compiler accept this?

```
1 List<Animal> animals1 = new ArrayList<Animal>();
```

And this?

```
1 List<Animal> animals2 = new ArrayList<Dog>();
```

Problem:

```
1 List<Dog> dogs = new ArrayList<Dog>();
2 List<Animal> animals = dogs; // compiler says "no" here.
3 animals.add(new Cat()); // ok, Cat is a subclass of Animal
4 Dog d = dogs.get(0); // Type error! That's a Cat, not a Dog!
```

Solution:

```
1 List<? extends Animal> myAnimals = new ArrayList<Dog>();
```

upper-bounded wildcard

Type wildcards

Solution:

```
1 List<? extends Animal> myAnimals1 = new ArrayList<Dog>();
2 List<? extends Animal> myAnimals2 = new ArrayList<Animal>();
```

Type wildcards

Solution:

```
List<? extends Animal> myAnimals1 = new ArrayList<Dog>();
List<? extends Animal> myAnimals2 = new ArrayList<Animal>();
```

Wildcards are useful for method parameters:

```
    public void makeThemBark(List<? extends Dog> dogs) {
        for (Dog d : dogs) {
            d.bark();
        }
    }
```

Type wildcards

Solution:

```
List<? extends Animal> myAnimals1 = new ArrayList<Dog>();
List<? extends Animal> myAnimals2 = new ArrayList<Animal>();
```

Wildcards are useful for method parameters:

```
    public void makeThemBark(List<? extends Dog> dogs) {
        for (Dog d : dogs) {
            d.bark();
        }
    }
```

Elsewhere:

```
List<Dog> dogs = getDogs();
List<Labrador> labradors = getLabradors();
makeThemBark(dogs);      // same method for both List<Dog>
makeThemBark(labradors); // and List<Labrador>!
```

Type wildcards

Solution:

```
1 List<? extends Animal> myAnimals1 = new ArrayList<Dog>();
2 List<? extends Animal> myAnimals2 = new ArrayList<Animal>();
```

Wildcards are useful for method parameters:

```
1     public void makeThemBark(List<? extends Dog> dogs) {
2         for (Dog d : dogs) {
3             d.bark();
4         }
5     }
```

Elsewhere:

```
1 List<Dog> dogs = getDogs();
2 List<Labrador> labradors = getLabradors();
3 makeThemBark(dogs);      // same method for both List<Dog>
4 makeThemBark(labradors); // and List<Labrador>!
```

But:

```
1     public void doSomething(List<? extends Animal> animals) {
2         animals.add(new Cat()); // compiler says "no"
3     }
```

Topics on Day 9

- Java Collections.
- Maps.
- Type wildcards.