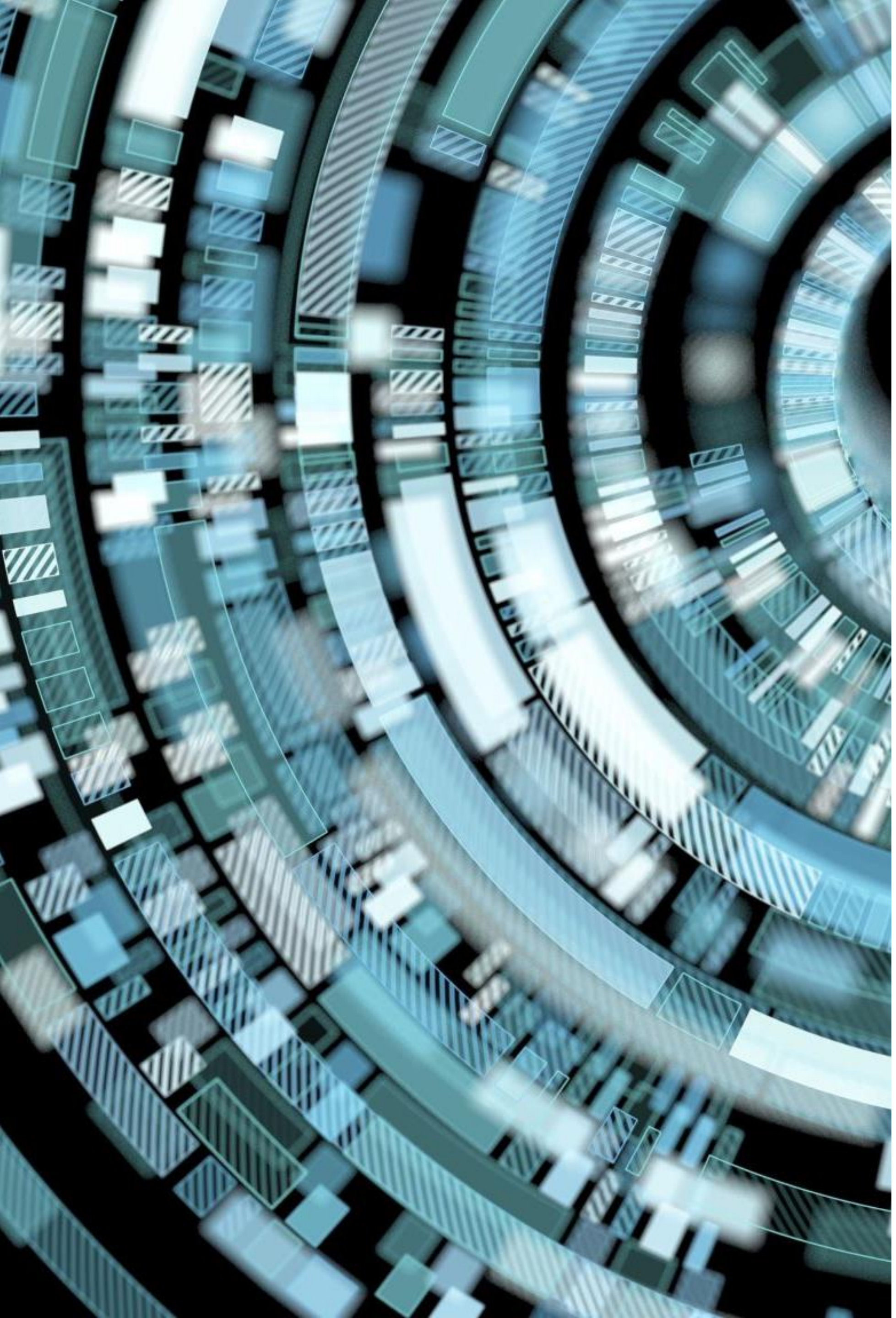
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）

taoyd@sustech.edu.cn

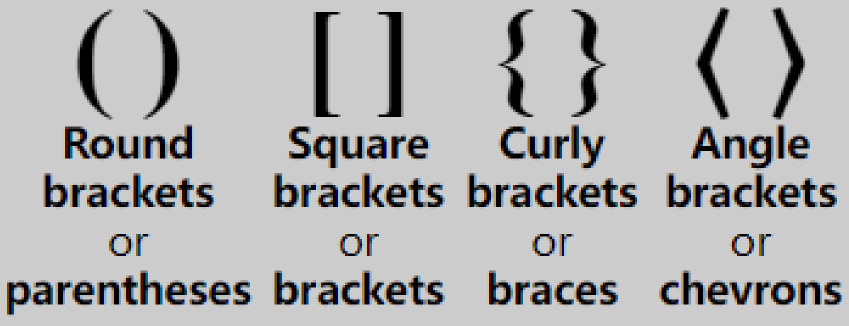


Lecture 3

- Generics
- Abstract Data Type (ADT)
- Collections

Generics (泛型)

- Introduced in JDK 5.0
- Generics mean parameterized types: types (classes and interfaces) can be used as parameters (将类型参数化)
- Consider it as a *template*



( )	[ ]	{ }	⟨ ⟩
Round brackets or parentheses	Square brackets or brackets	Curly brackets or braces	Angle brackets or chevrons

```
List <?>
```

Motivating Example I

Explicit casting could be inefficient and not good for readability.

```
List list = new ArrayList();
list.add("hi");
```

Compiler error:
Type mismatch: cannot convert
from Object to String

✗

```
String s = list.get(0);
```

Need to explicitly cast to String

√

```
String s = (String)list.get(0);
```

Motivating Example II

Error-prone: may cause type-related runtime errors if a programmer makes a mistake with the explicit casting.

```
List list = new ArrayList();
list.add("Hello");
list.add(2022);

for(int i=0;i<list.size();i++) {
❌  String elem = (String)list.get(i);
    System.out.println(elem);
}
```

Program throws ClassCastException: class java.lang.Integer cannot be cast to class java.lang.String

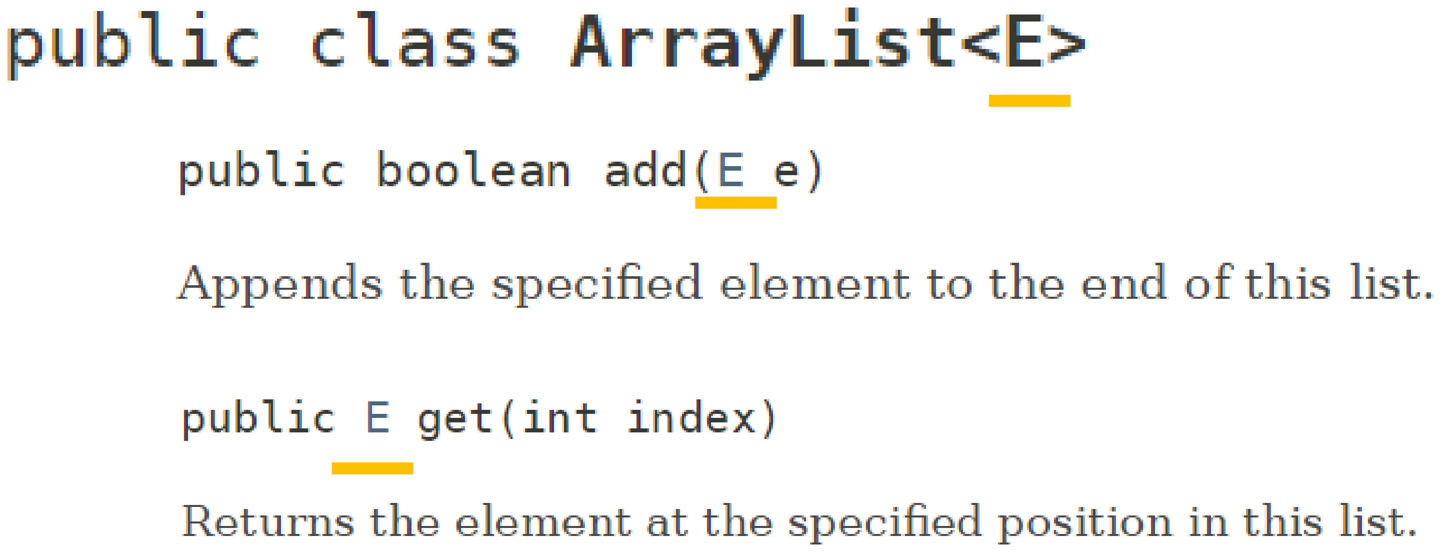
Solution I

What's the problem with this solution?

- Using a dedicated list for each type
  - StringArrayList
  - IntegerArrayList
  - CharArrayList
  - BoolArrayList
  - ……
- Infeasible solution
  - Too many kinds of list (thousands in Java)
  - Too much duplication
  - Hard to scale for user-defined objects

Solution: Generics

- Parameterized types: types like classes and interfaces can be used as parameters



```
public class ArrayList<E>

        public boolean add(E e)
```

Appends the specified element to the end of this list.

```
        public E get(int index)
```

Returns the element at the specified position in this list.

- E stands for "element" (sometimes we use T)
- E could be any non-primitive type
- All elements of the list should be of type E

Example

```
// list of strings
ArrayList<String> strList = new ArrayList<String>();   √
// list of floats
ArrayList<Float> floatList = new ArrayList<Float>();   √
// list of cars
ArrayList<Car> personList = new ArrayList<Car>();      √

strList.add("Hello");  √

strList.add(2022);     ✗   Compilation error: cannot add int to a String list

strList.add(1.23);     ✗   Compilation error: cannot add double to a String list
```

Comparisons

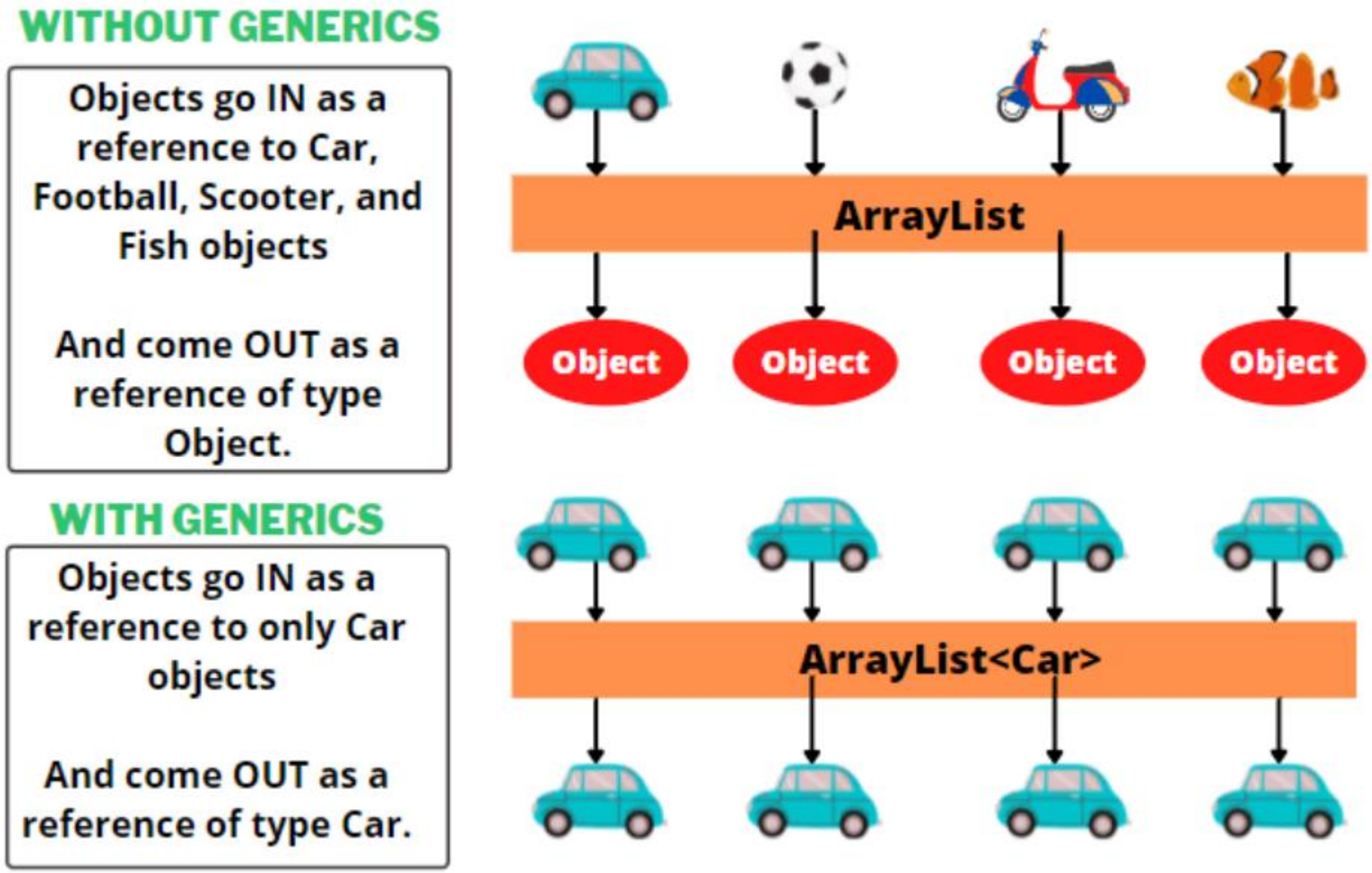
It's better to discover errors as early as possible!

Could put anything into the list; compiler won't complain

Need explicit type cast to get element; otherwise runtime errors (crash)

Could only put the specified element; otherwise compiler will complain

No need for type cast since type-safety is already guaranteed in compile time



WITHOUT GENERICS

Objects go IN as a reference to Car, Football, Scooter, and Fish objects

And come OUT as a reference of type Object.

WITH GENERICS

Objects go IN as a reference to only Car objects

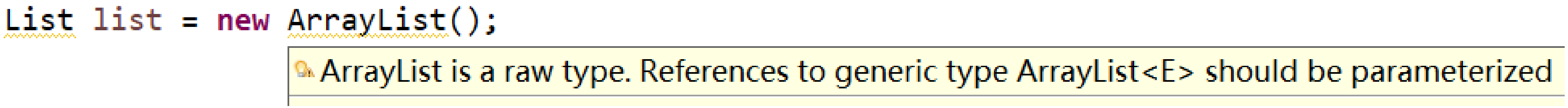
And come OUT as a reference of type Car.

Image source: https://www.scientecheasy.com/2021/10/generics-in-java.html/

Terms

Example	Term
List<E>	Generic type
E	Formal type parameter (类型形参)
List<String>	Parameterized type
String	Actual type parameter (类型实参)
List	Raw type

Avoid using raw types



```
List list = new ArrayList();
```

ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

- By using raw types, we’ll lose all the type-safety and expressiveness benefits of generics
- Question: but the code could still compile (warning instead of error) and run, why?

Backward compatibility (向后兼容): We want to ensure that legacy Java code (遗留代码) that was created before Generics is introduced could still execute

Using Generics

- Generic classes
- Generic interfaces
- Generic methods

Classes in Java Collection (e.g., List, Queue, Set); will be introduced in the next section

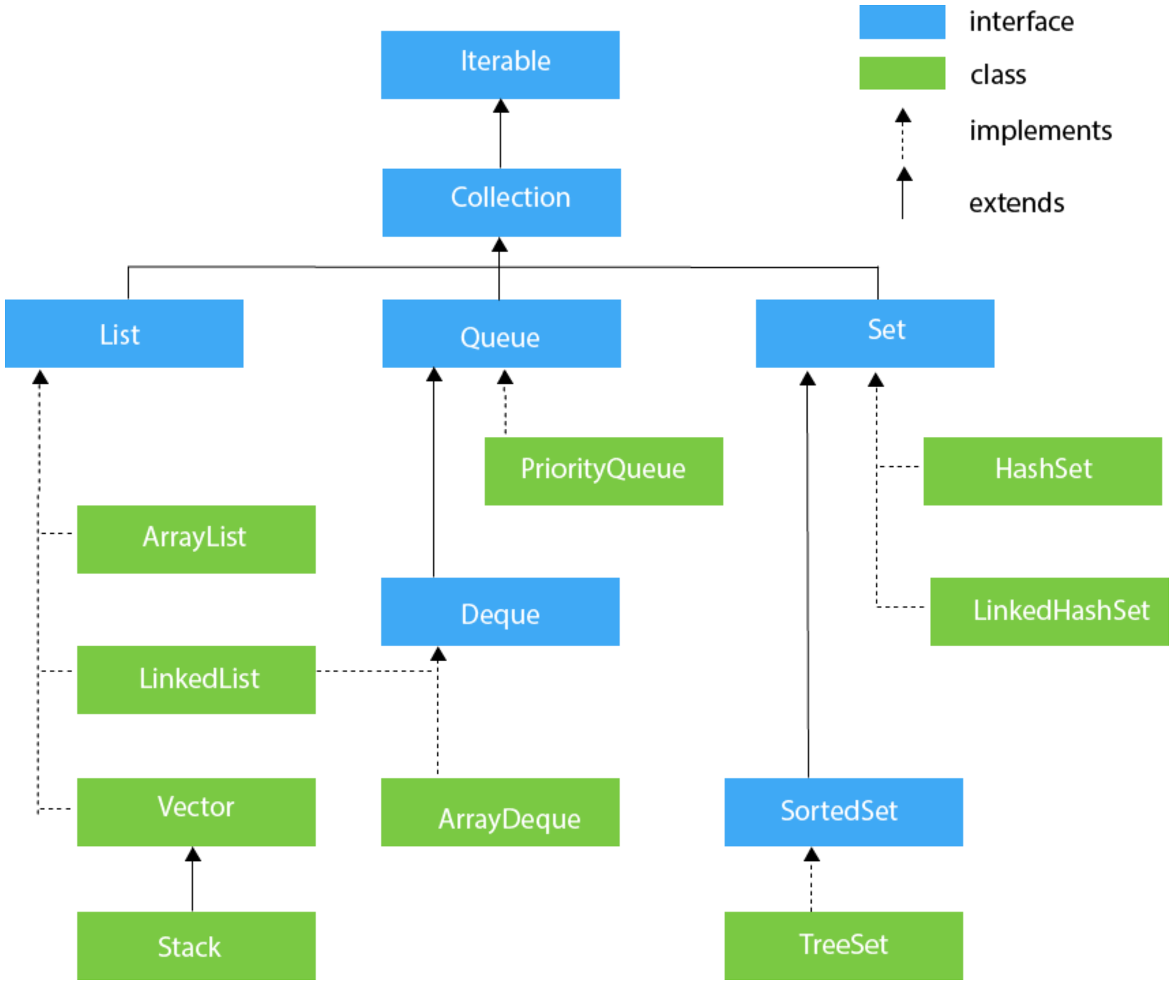
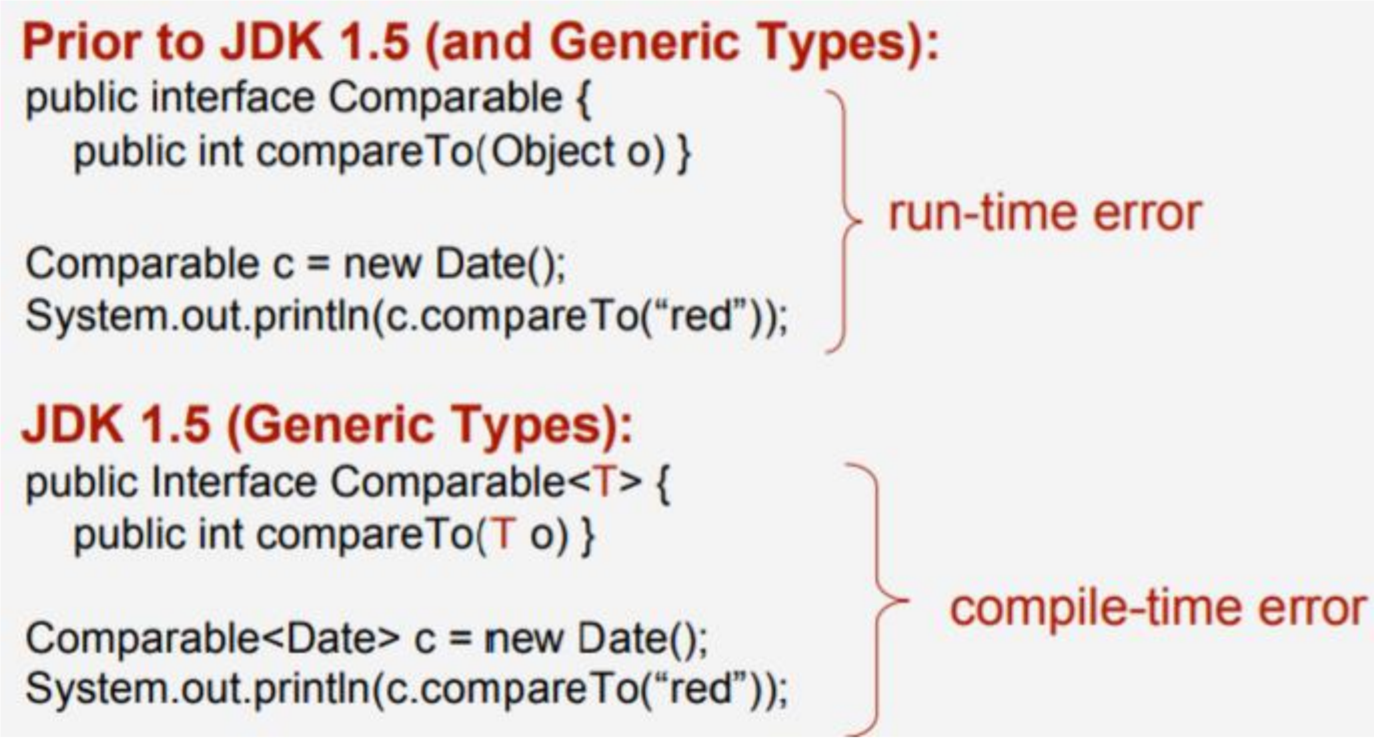


Image source: https://www.javatpoint.com/collections-in-java

Using Generics

- Generic classes
- Generic interfaces
- Generic methods



Prior to JDK 1.5 (and Generic Types):

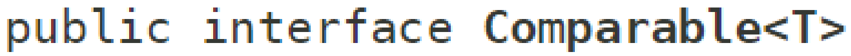
```
public interface Comparable {
    public int compareTo(Object o) }

Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

run-time error

JDK 1.5 (Generic Types):

```
public Interface Comparable<T> {
    public int compareTo(T o) }

Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

compile-time error



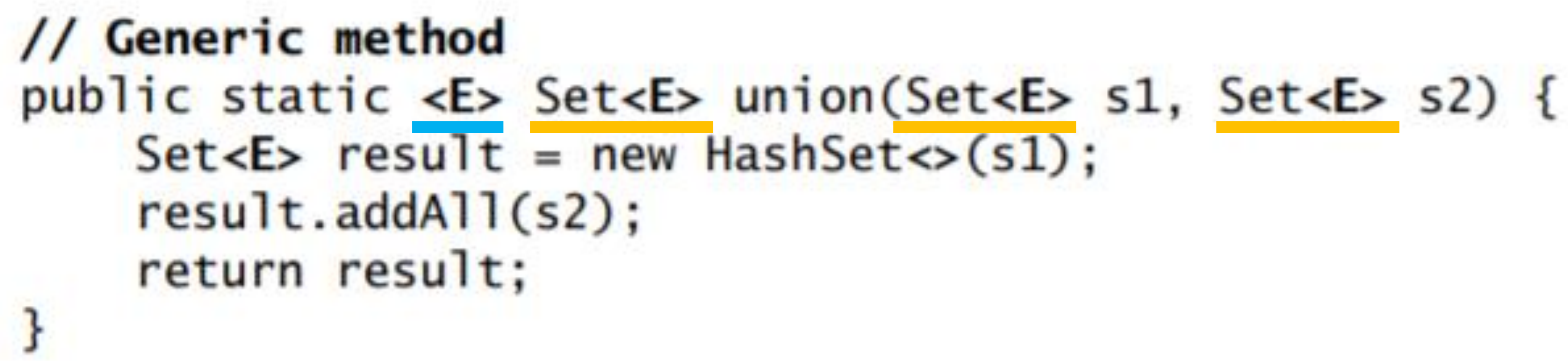
```
public interface Comparable<T>
```

See Lecture 1 notes

Image source: https://www.cs.rit.edu/~rlaz/cs2/slides/CS2_Week5.pdf

Using Generics

- Generic classes
- Generic interfaces
- Generic methods
  - Methods that introduce their own type parameters

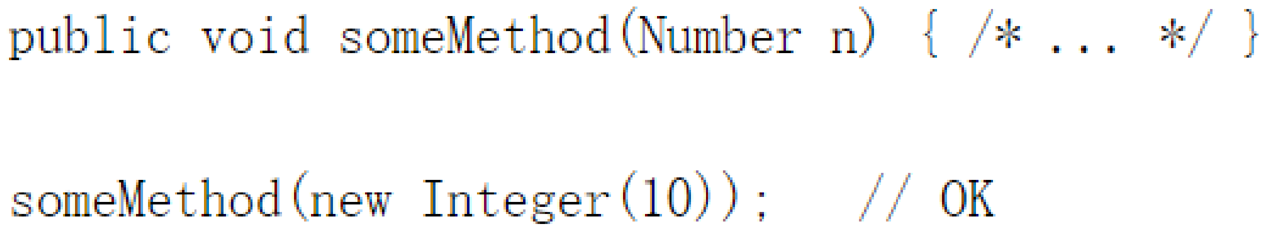


```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

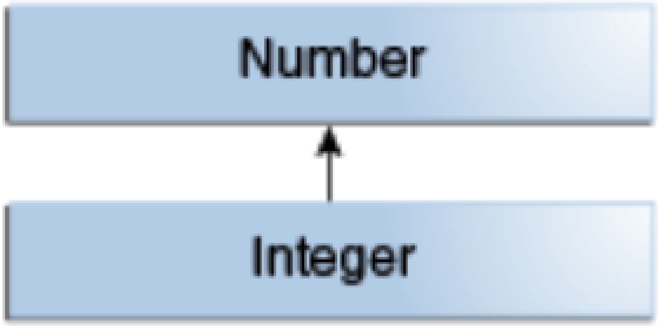
Example from “Effective Java”

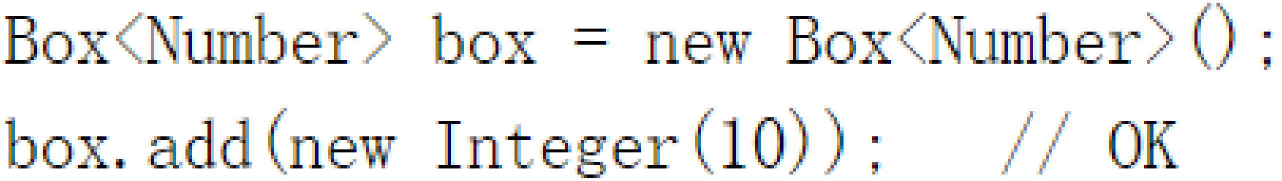
Generics & Inheritance

The code works since Integer **is a** Number (is-a relationship in OO terms)



```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
```





```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
```

Example from the official Java Doc: https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html

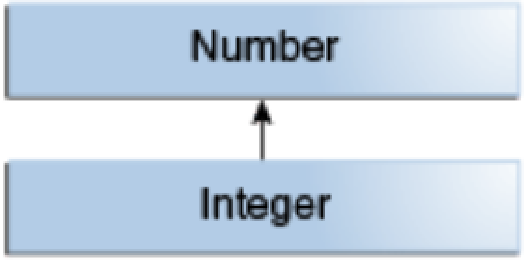
Generics & Inheritance

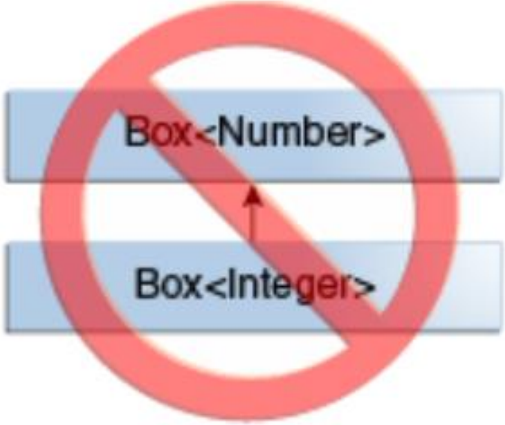
```
public void numberTest(Box<Number> l) {
     /*……*/
}

Box<Integer> l = new Box<Integer>();
numberTest(l);
```

Compiler will complain on type mismatch:

`Box<Number>` has no relationship to `Box<Integer>`, regardless of whether `Number` and `Integer` are related

Why this is not allowed?

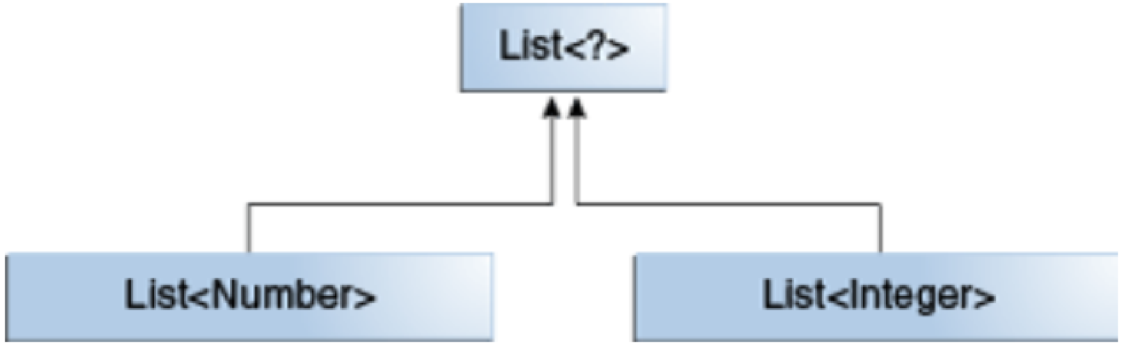




Wildcards (通配符)

- Using "?" to create a relationship between generic types
- List<?> could be `List<Number>`, `List<Integer>`, `List<String>`, etc.

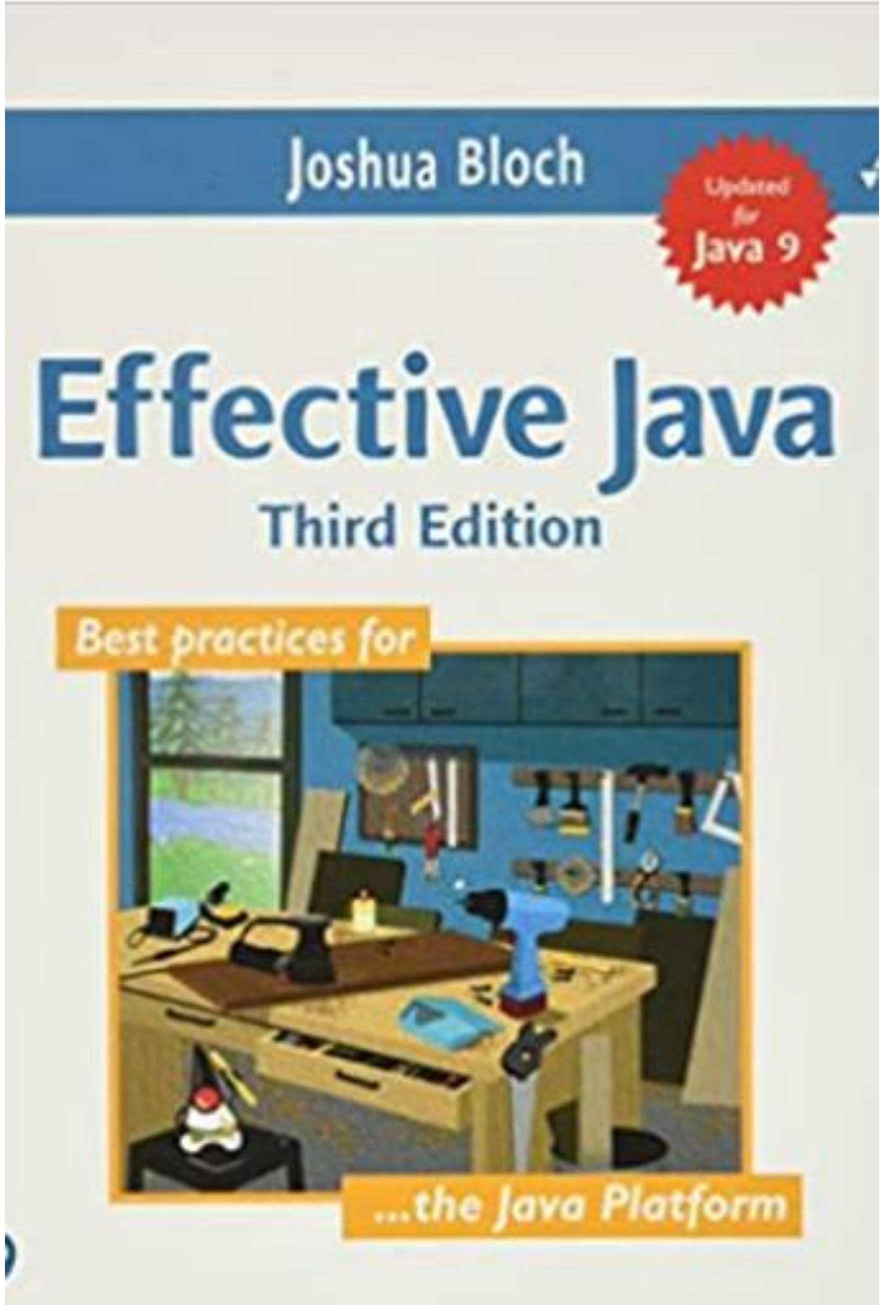
```
public void test(List<?> l) {/*…*/}

List<Integer> l1 = new ArrayList<Integer>();
List<Number> l2 = new ArrayList<Number>();

test(l1);
test(l2);
```

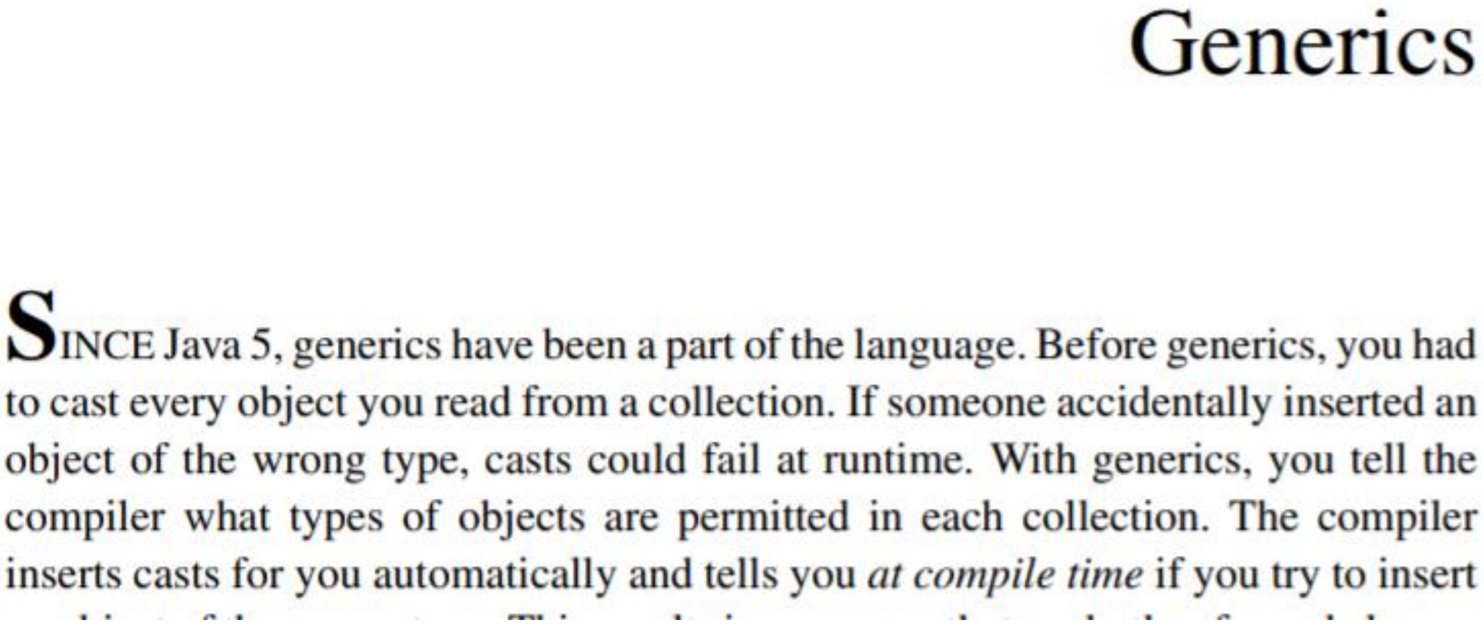


The common parent is List<?>.

Wildcards

- Unbounded:
  - `Box<?>` is a superclass of `Box<T>` for any `T`
- Upper bounded:
  - `Box<? extends T>`: a box of any type that is a subtype of `T`
  - Bounded by the superclass
- Lower bounded:
  - `Box<? super T>`: a box of any type that is a supertype of `T`
  - Bounded by the subclass

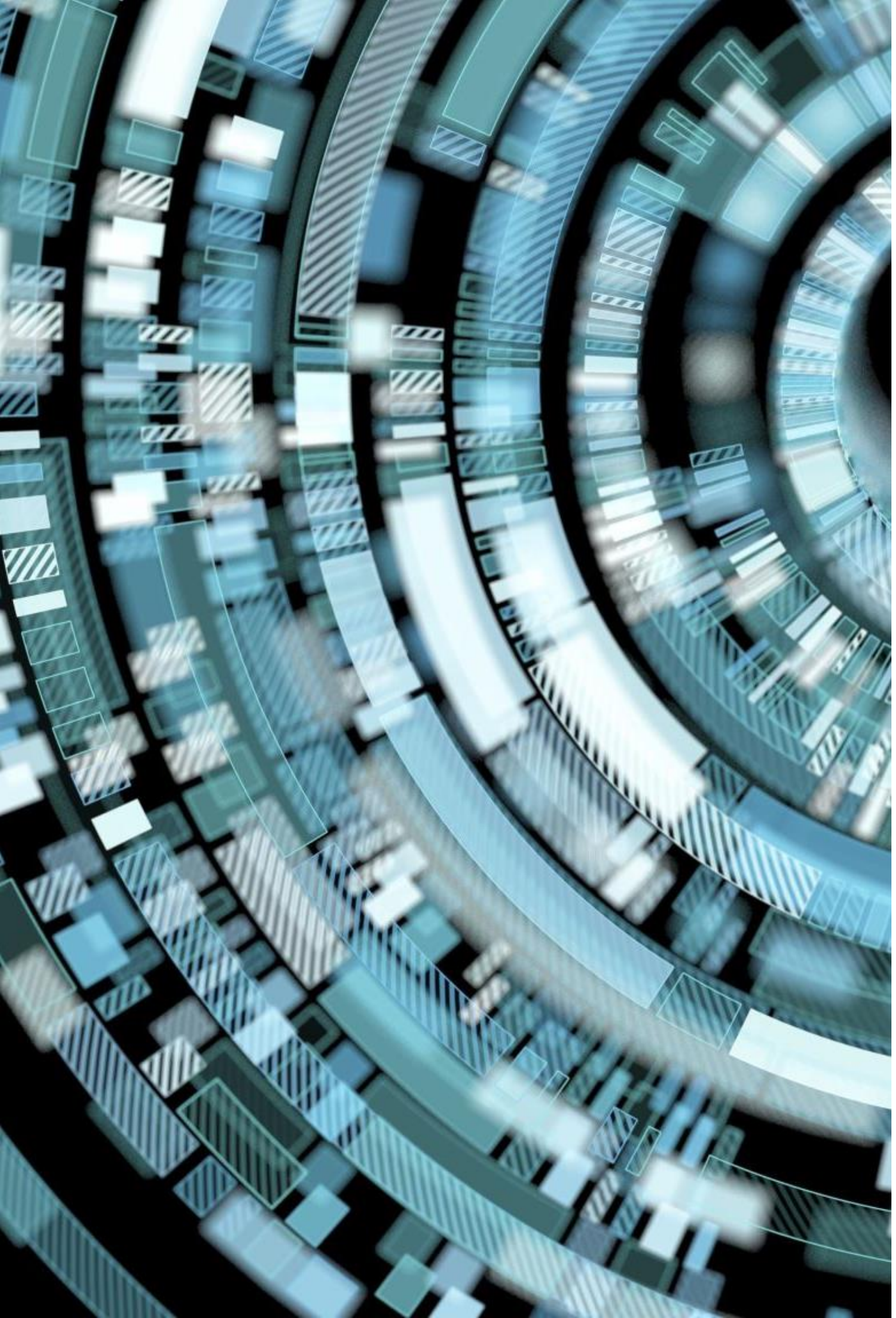




Generics

SINCE Java 5, generics have been a part of the language. Before generics, you had to cast every object you read from a collection. If someone accidentally inserted an object of the wrong type, casts could fail at runtime. With generics, you tell the compiler what types of objects are permitted in each collection. The compiler inserts casts for you automatically and tells you *at compile time* if you try to insert

Further Reading



Lecture 3

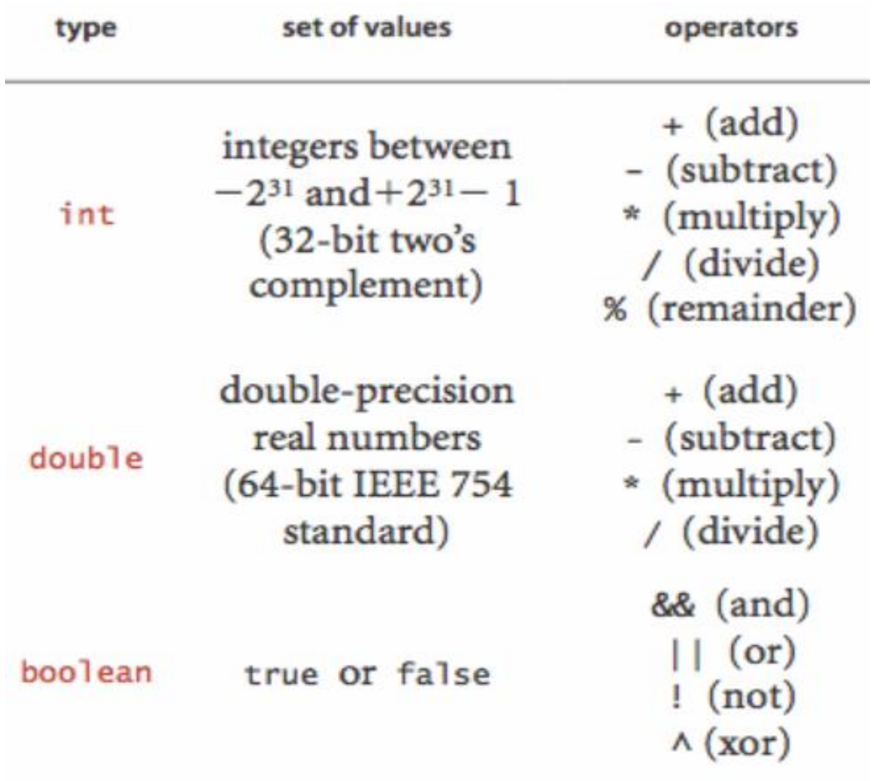
- Generics
- Abstract Data Type (ADT)
- Collections

Data Type

A data type is a set of values and a set of operations on those values

Primitive Types

- values immediately map to machine representations
- operations immediately map to machine instructions



type	set of values	operators
`int`	integers between -2^{31} and $+2^{31}-1$ (32-bit two's complement)	+ (add) - (subtract) * (multiply) / (divide) % (remainder)
`double`	double-precision real numbers (64-bit IEEE 754 standard)	+ (add) - (subtract) * (multiply) / (divide)
`boolean`	`true` or `false`	&& (and) || (or) ! (not) ^ (xor)

Abstract Data Type (ADT)

- A type (or class) for objects whose behavior is defined by a set of values and a set of operations.
  - How values are stored in memory is hidden from the client
  - How operations are implemented internally is hidden from client

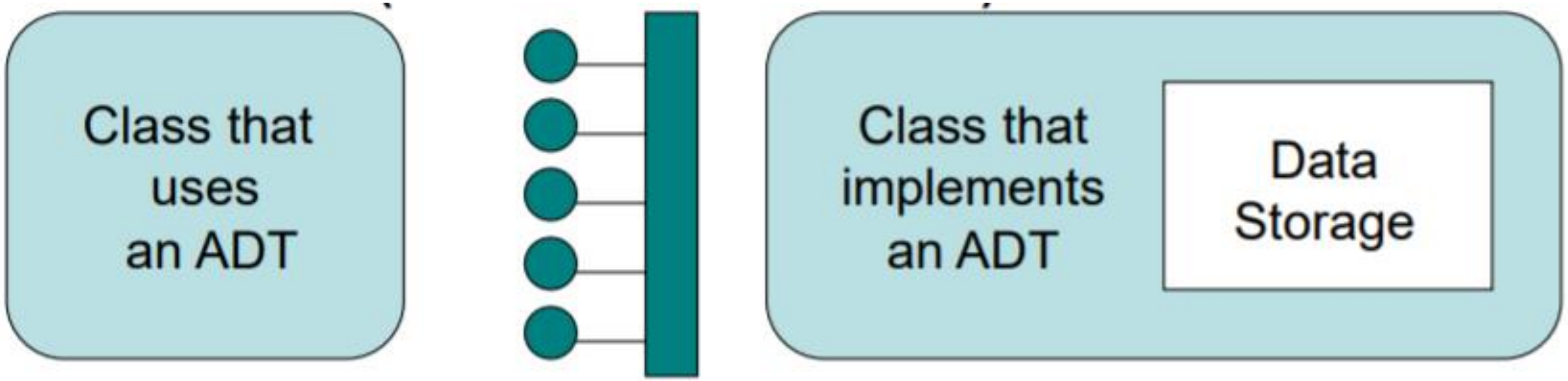


Image source: https://www.cs.umb.edu/~bobw/CS210/Lecture06.pdf

Stack ADT

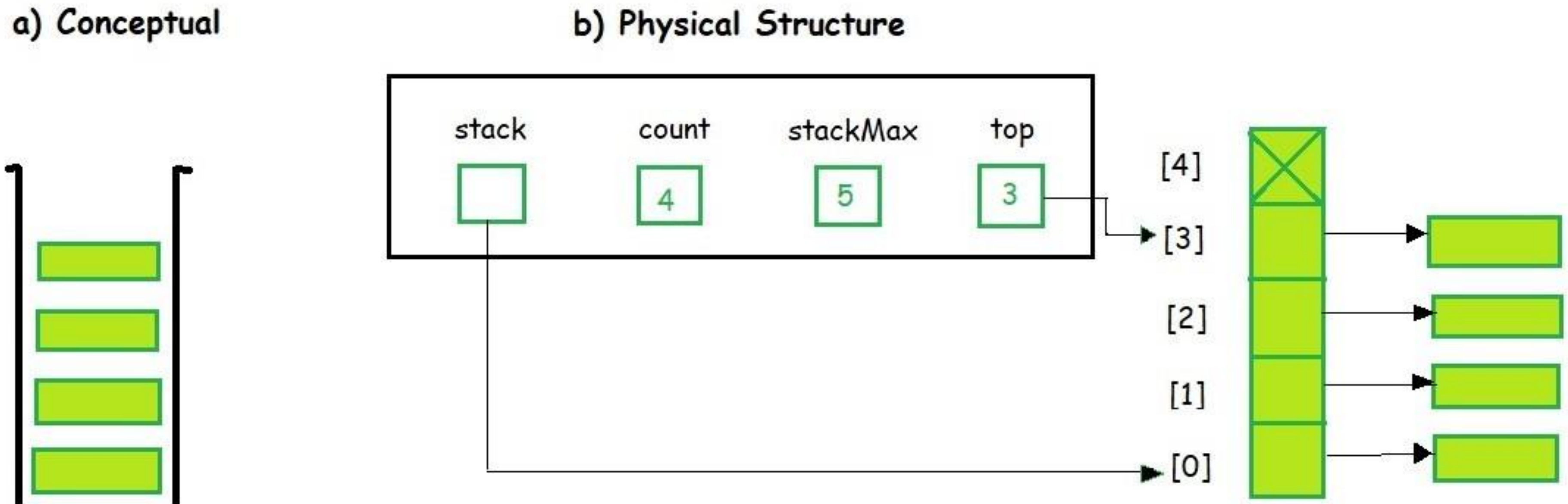


Image source: https://www.geeksforgeeks.org/abstract-data-types/

For clients

- Stack is a last-in-first-out linear collections
- Could push and pop elements

Possible implementations

- Instead of data being stored in each node, the pointer to data is stored
- The program allocates memory for the data and address
- The stack head structure contains a pointer to top and count of number of entries currently in stack
- ......

List ADT

- Series of elements with insertion and deletion operations

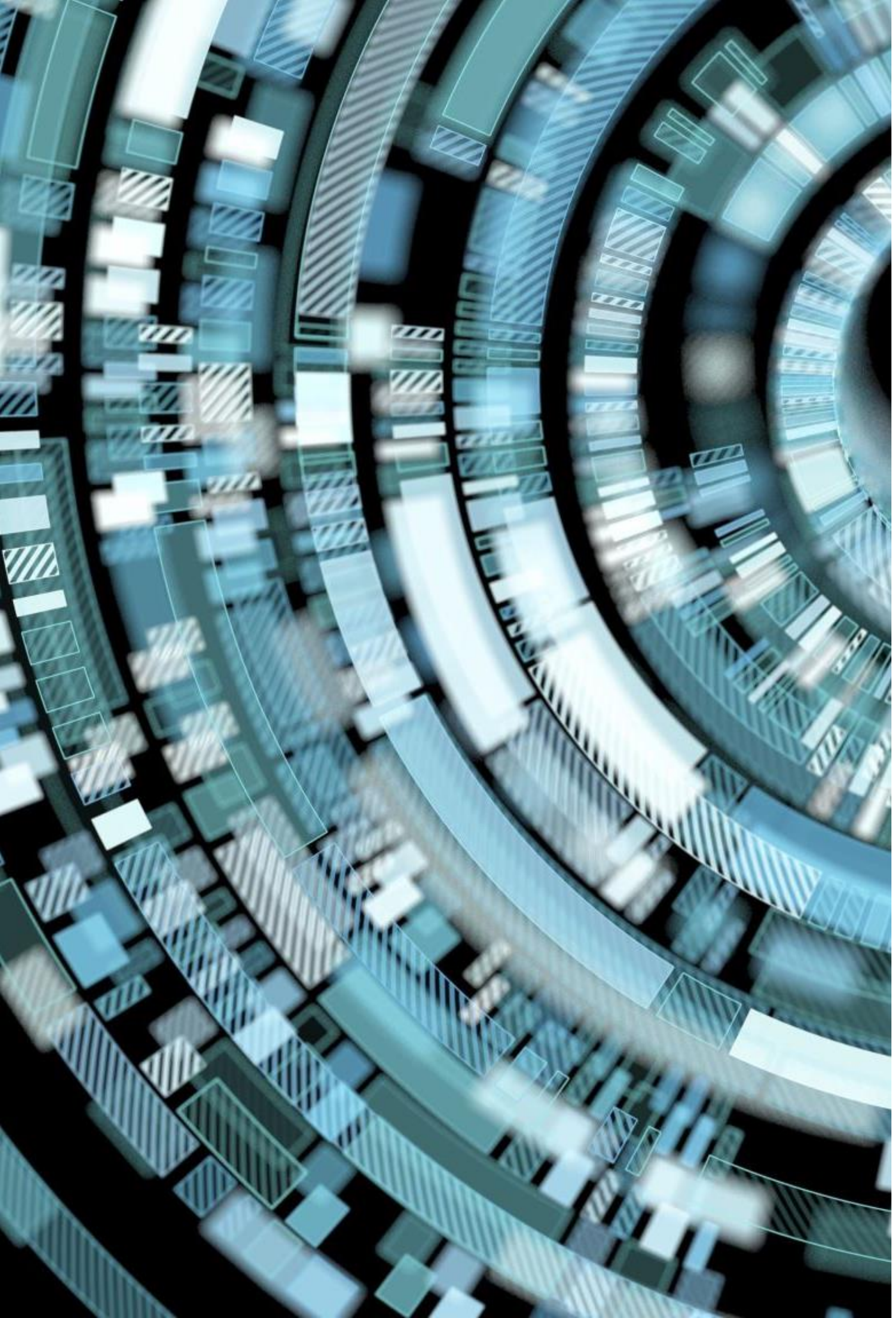
- Possible implementations
    - Using an array
    - Using a linked list (nodes with references to one another)

- Choosing an implementation
    - Array is faster for finding elements
    - Linked list is faster for inserting and deleting arbitrary elements

Operations of ADT

Java has ADT such as List, Stack, Queue, Set, Map, etc.

- Creators create new objects of the type
- Producers create new objects from old objects of the type
  - E.g., `String.concat()` concatenates two strings and produce a new one
- Observers takes an object of the abstract type and return an object of a different type
  - E.g., `List.size()` returns an integer
- Mutators change the object itself
  - E.g., `List.add()` changes the list



Lecture 3

- Generics
- Abstract Data Type (ADT)
- Collections

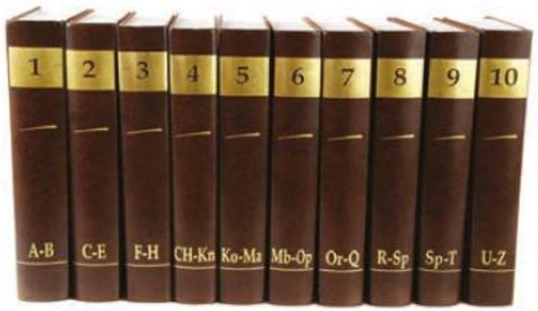
Concepts of Collections List, Stack, Map and Set?

A list is a collection that remembers the order of its elements.

A set is an unordered collection of unique elements.

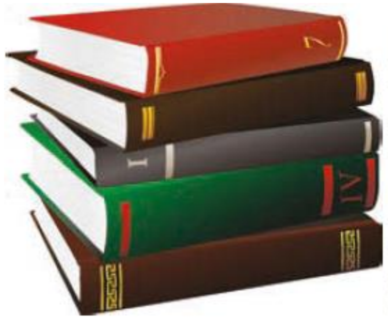
A stack is a collection of elements with “last-in, first-out” retrieval.

A map keeps associations between key and value objects.

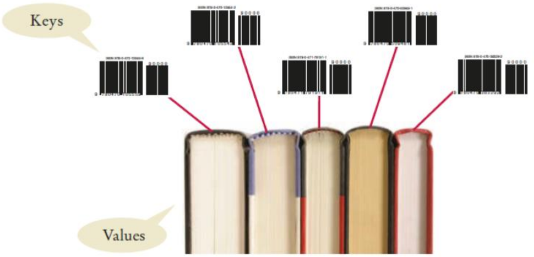


© Filip Fuxa/iStockphoto.

© parema/iStockphoto.



© Vladimir Trenin/iStockphoto.



(books) © david franklin/iStockphoto.

Materials from the slides of Dr. HE Mingxin

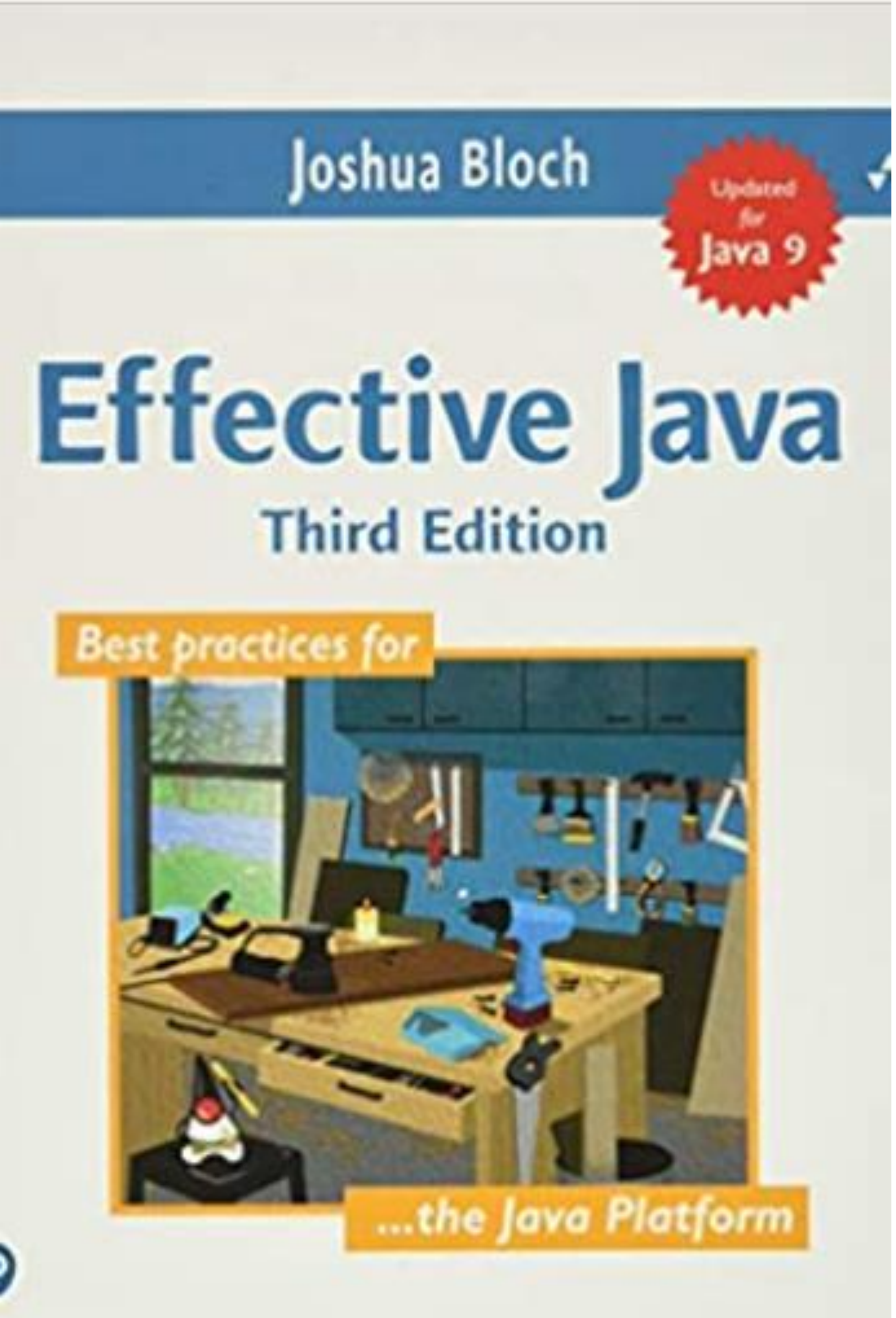
The Java Collections Framework

- Collection
  - A group of objects
  - Mainly used for data storage, data retrieval, and data manipulation
- Framework
  - A set of classes and interfaces which provide a ready-made architecture.
- Collections Framework
  - A unified architecture for representing and manipulating collections
  - Reusable data structures & functionalities
  - Collections can be manipulated independently of the details of their representation

History

- Before JDK 1.2 ('90s)
    - Java only has `Arrays`, `Vectors`, and `Hashtables` for grouping objects
    - They are defined independently with no common interface (although many concepts are the same)
    - Difficult to use, to remember, and to extend

- The Collections Framework was introduced in JDK 1.2 (1998)
    - Consistent APIs for common functionalities (e.g., add())
    - Reducing programming & design efforts
    - Increases program speed and quality



The collections framework was designed and developed primarily by Joshua Bloch

Joshua Bloch, is a former Distinguished Engineer at Sun Microsystems and Google's chief Java architect.

He led the design and implementation of numerous Java platform features, including JDK 5.0 language enhancements and the award-winning Java Collections Framework.

He holds a Ph.D. in computer science from Carnegie-Mellon University.

Collections

Parts of the following materials are adapted from the original slides from Josh Bloch

The Java^{TM} Platform Collections Framework

Joshua Bloch

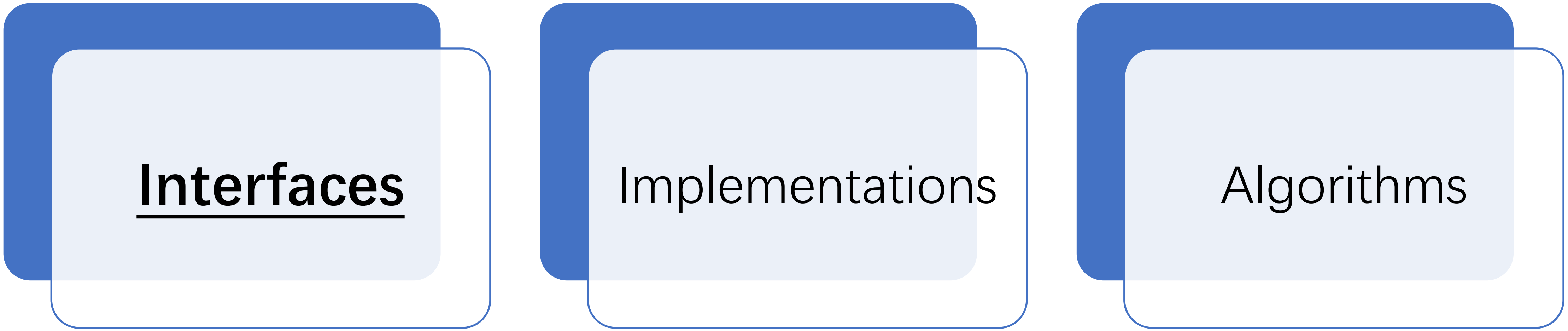
Sr. Staff Engineer, Collections Architect

Sun Microsystems, Inc.

15-214 5

(https://www.cs.cmu.edu/~charlie/courses/15-214/2016-fall/slides/15-collections%20design.pdf)

Core Elements in the Java Collections Framework



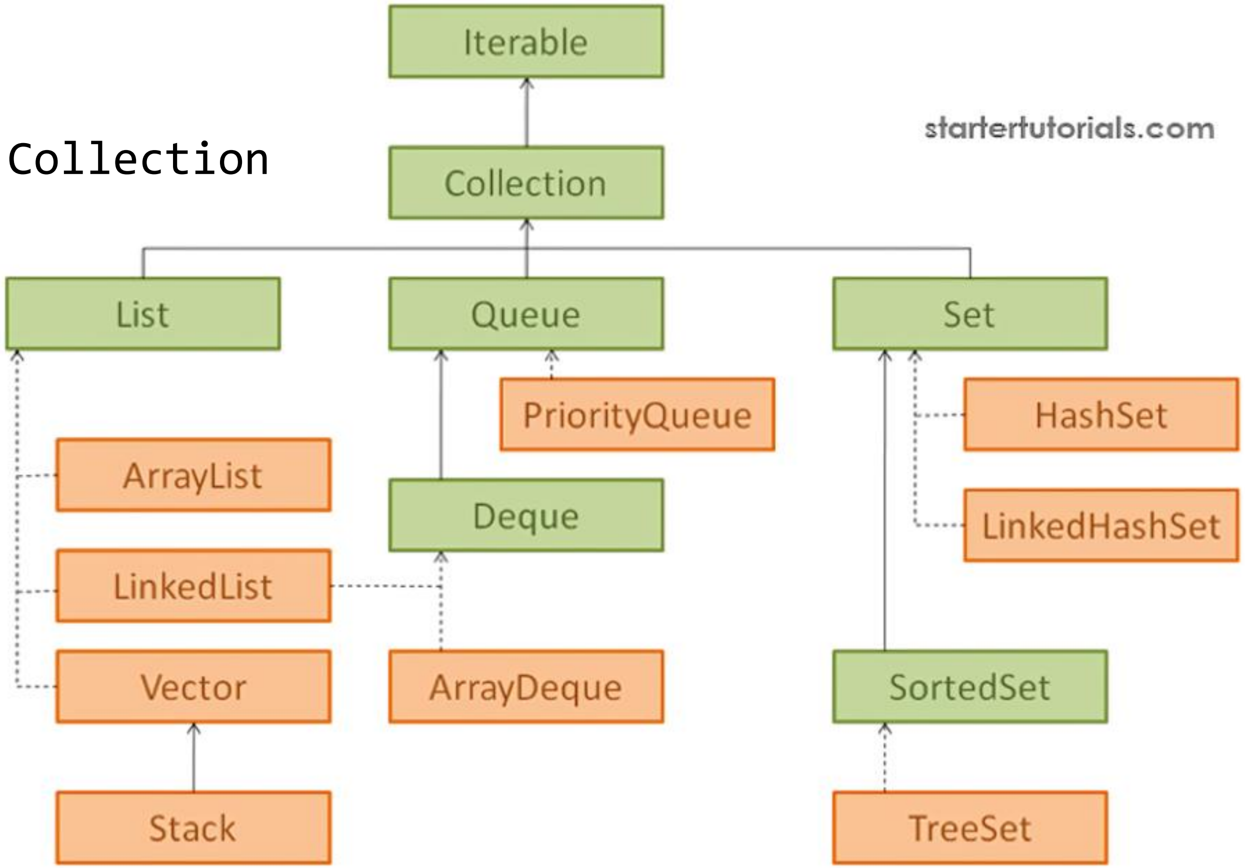
<u>Interfaces</u>

Implementations

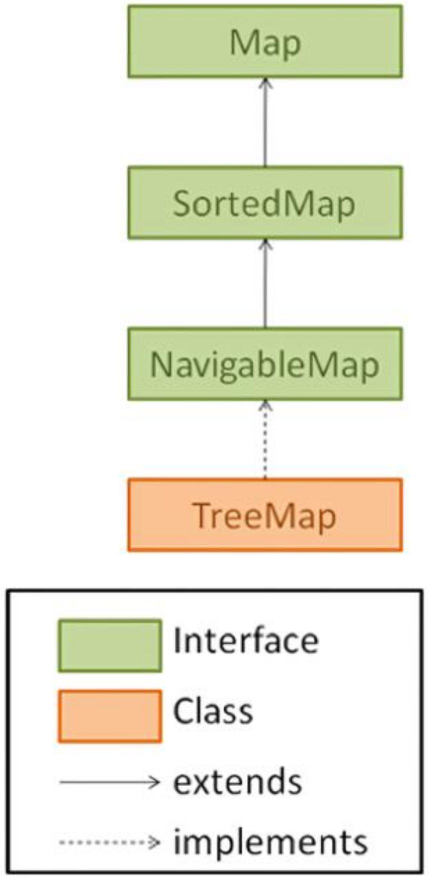
Algorithms

Collection Class Hierarchy

java.util.Collection

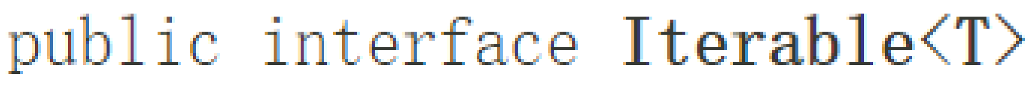


java.util.Map

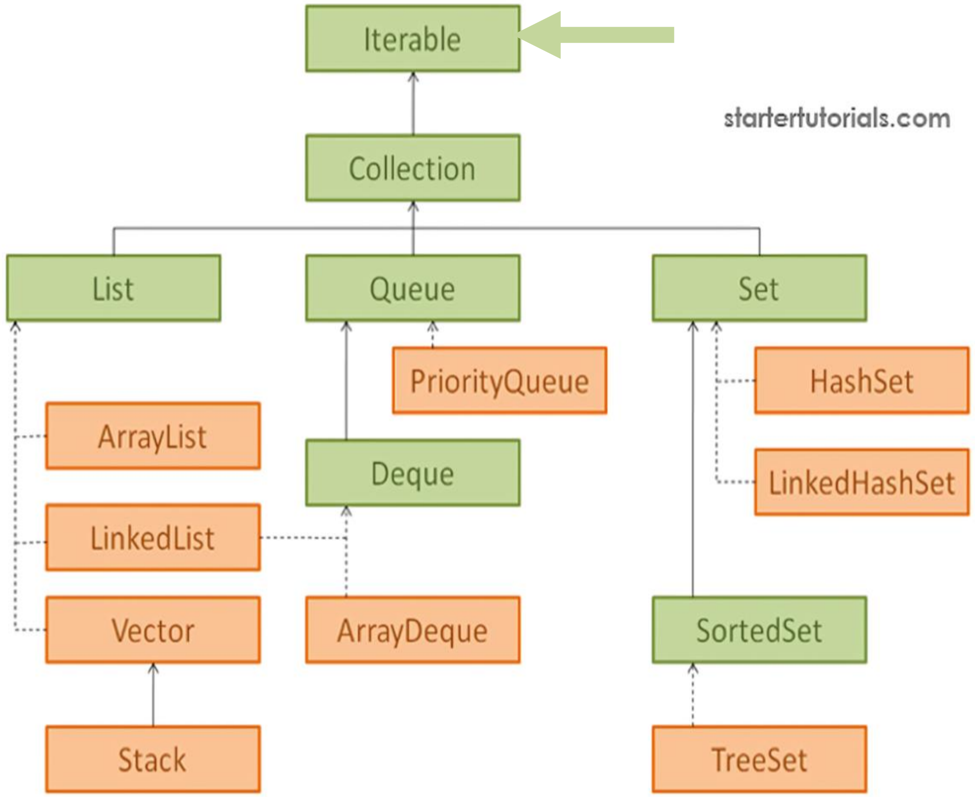


The `Iterable<T>` interface

- Iterable：可迭代的、可遍历的
- Implementing this interface allows an object to be the target of the "foreach" statement.

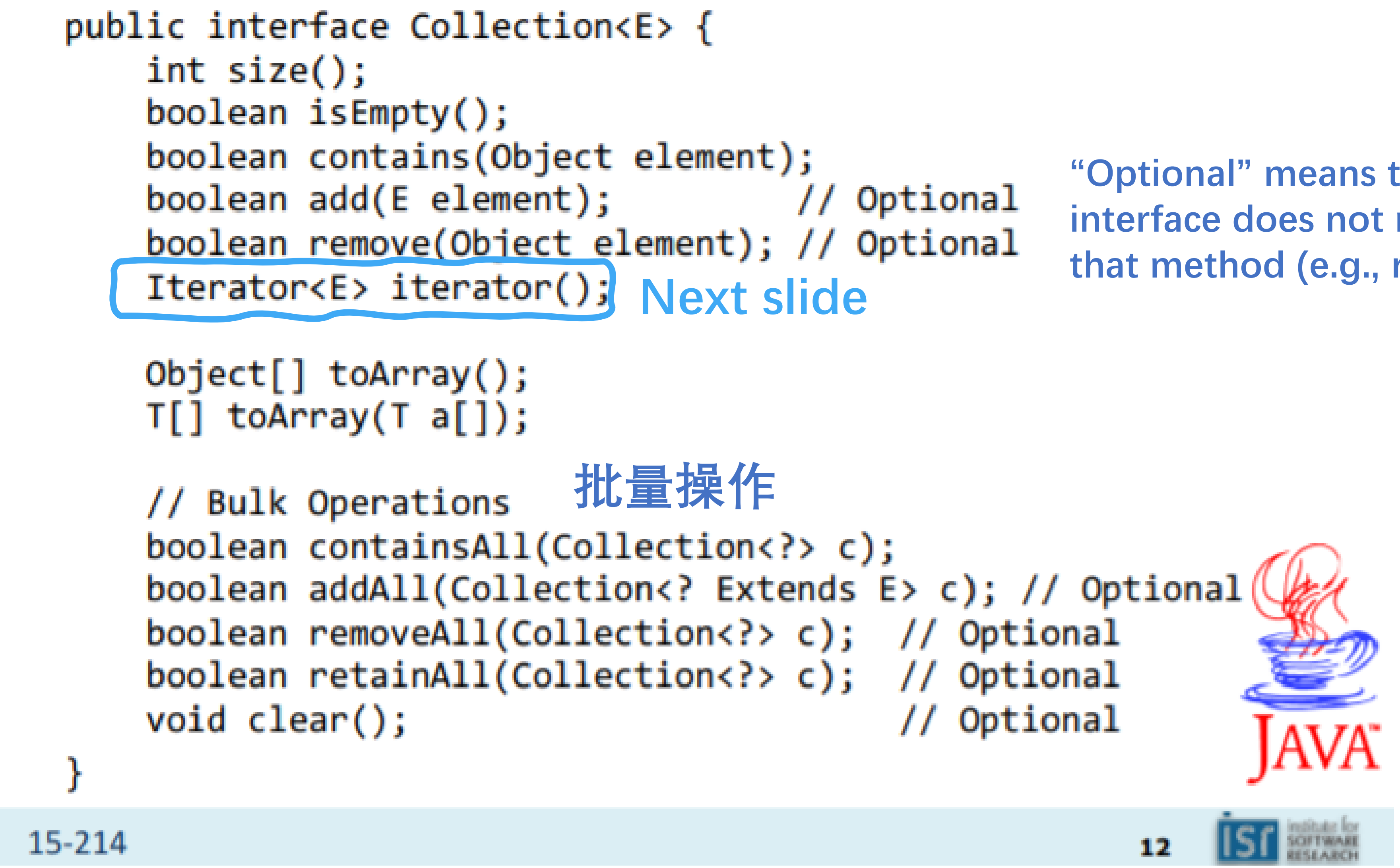


```
public interface Iterable<T>
```



Collection Interface

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



```
public interface Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();

    Object[] toArray();
    T[] toArray(T a[]);

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? Extends E> c); // Optional
    boolean removeAll(Collection<?> c);   // Optional
    boolean retainAll(Collection<?> c);   // Optional
    void clear();                         // Optional
}
```

15-214 12

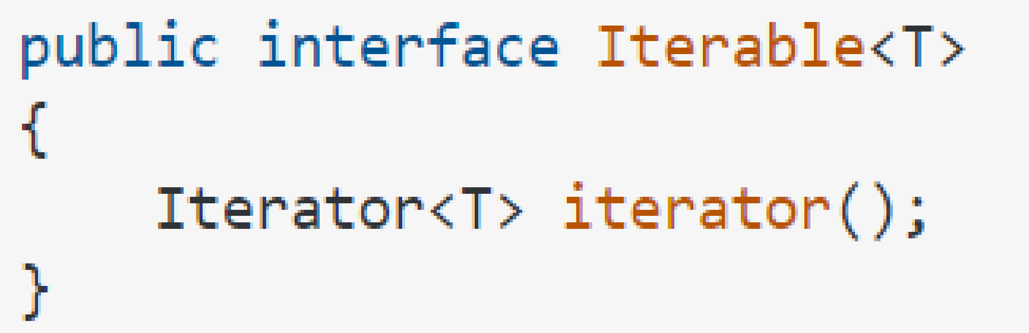
Next slide

批量操作

“Optional” means that classes implementing this interface does not necessarily have to implement that method (e.g., read-only collection)

The Iterator<T> interface

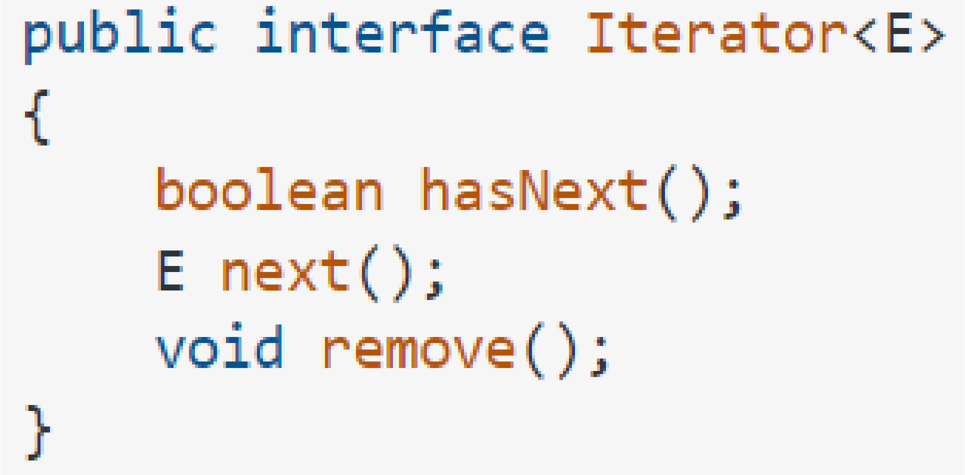
可迭代的



```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

A representation of a series of elements that can be iterated over

迭代器

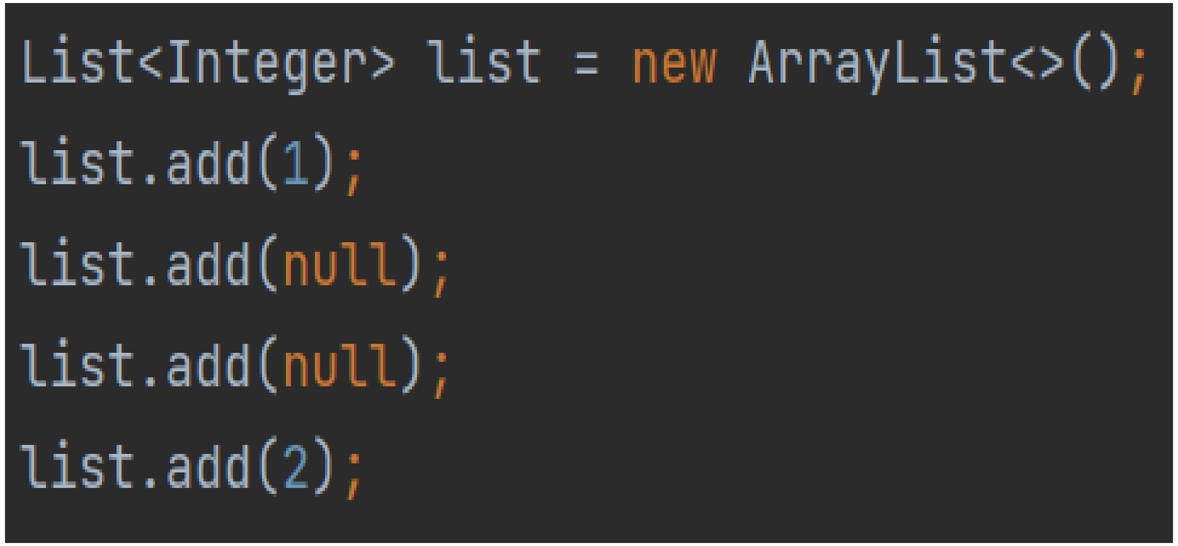


```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

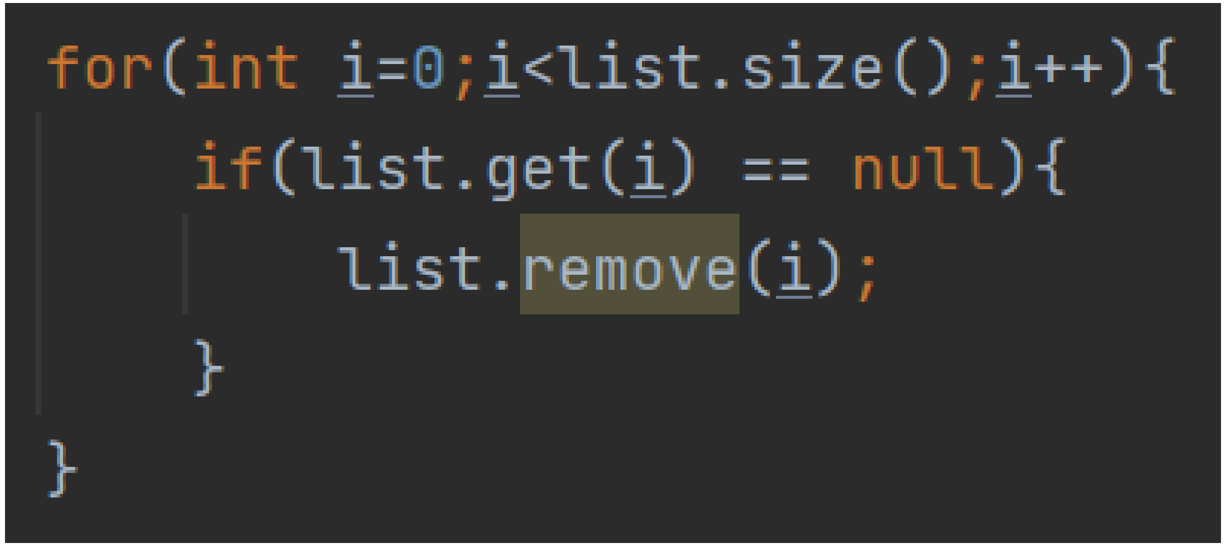
An iterator supports specific operations for performing iteration

An Iterable class could be iterated over using an Iterator

Example: remove all the nulls from a list



```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(null);
list.add(null);
list.add(2);
```



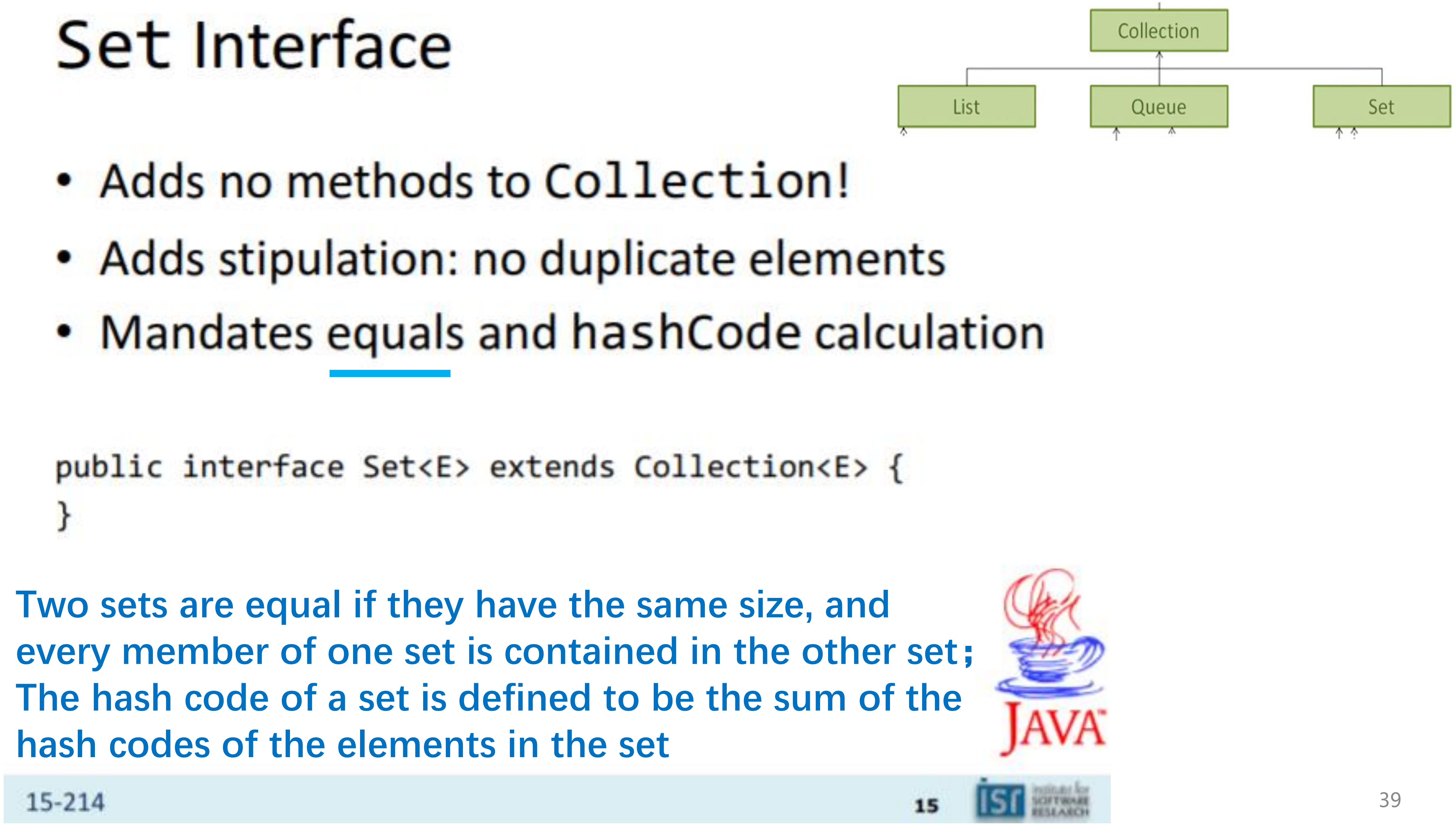
```
for(int i=0;i<list.size();i++){
    if(list.get(i) == null){
        list.remove(i);
    }
}
```

Content of list: [1, null, 2]

Example: remove all the nulls from a list

Iterators allow the caller to remove elements from the underlying collection during the iteration

```
public static void removeNulls(Collection<?> c) {
    for (Iterator<?> i = c.iterator(); i.hasNext(); ) {
        if (i.next() == null){
                i.remove();
        }
}

List<Integer> list = new ArrayList<Integer>();
list.add(1);
list.add(null);
list.add(null);
list.add(2);

removeNulls(list);
for(Integer i: list) {
    System.out.println(i);
}
```

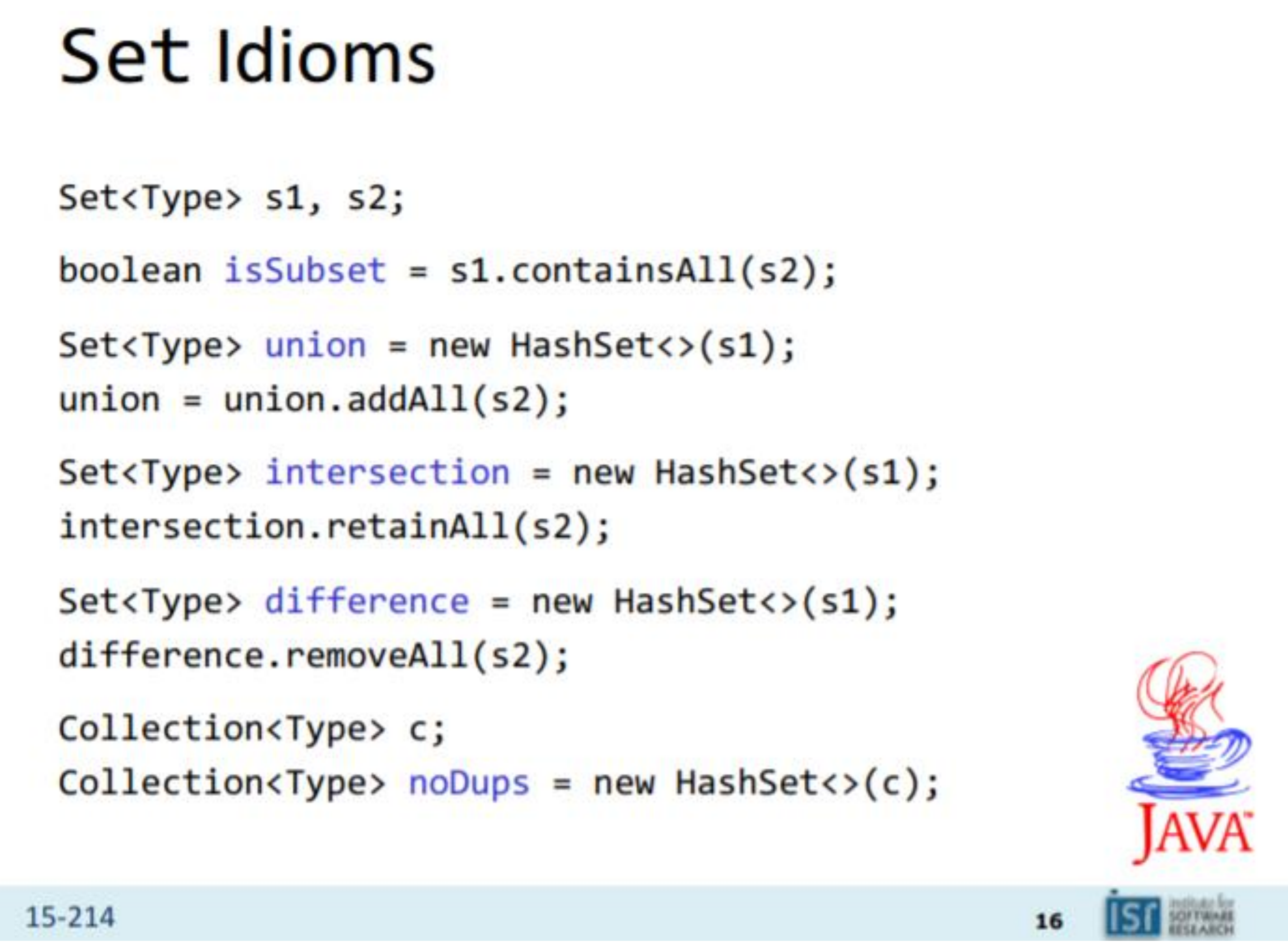


Set Interface

- Adds no methods to Collection!
- Adds stipulation: no duplicate elements
- Mandates equals and hashCode calculation

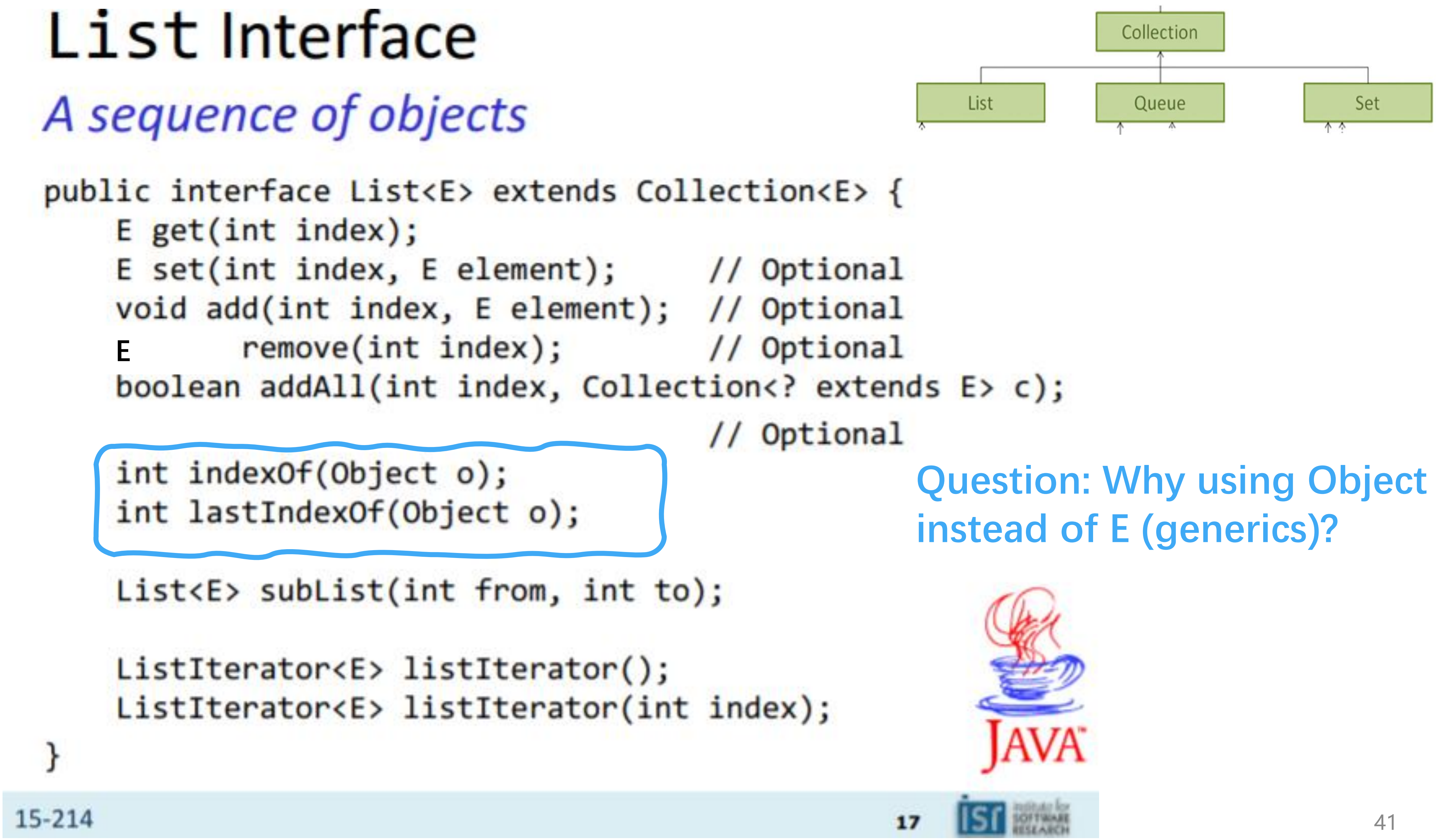
```
public interface Set<E> extends Collection<E> {
}
```

Two sets are equal if they have the same size, and every member of one set is contained in the other set; The hash code of a set is defined to be the sum of the hash codes of the elements in the set



Set Idioms

```
Set<Type> s1, s2;

boolean isSubset = s1.containsAll(s2);

Set<Type> union = new HashSet<>(s1);
union = union.addAll(s2);

Set<Type> intersection = new HashSet<>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<>(s1);
difference.removeAll(s2);

Collection<Type> c;
Collection<Type> noDups = new HashSet<>(c);
```

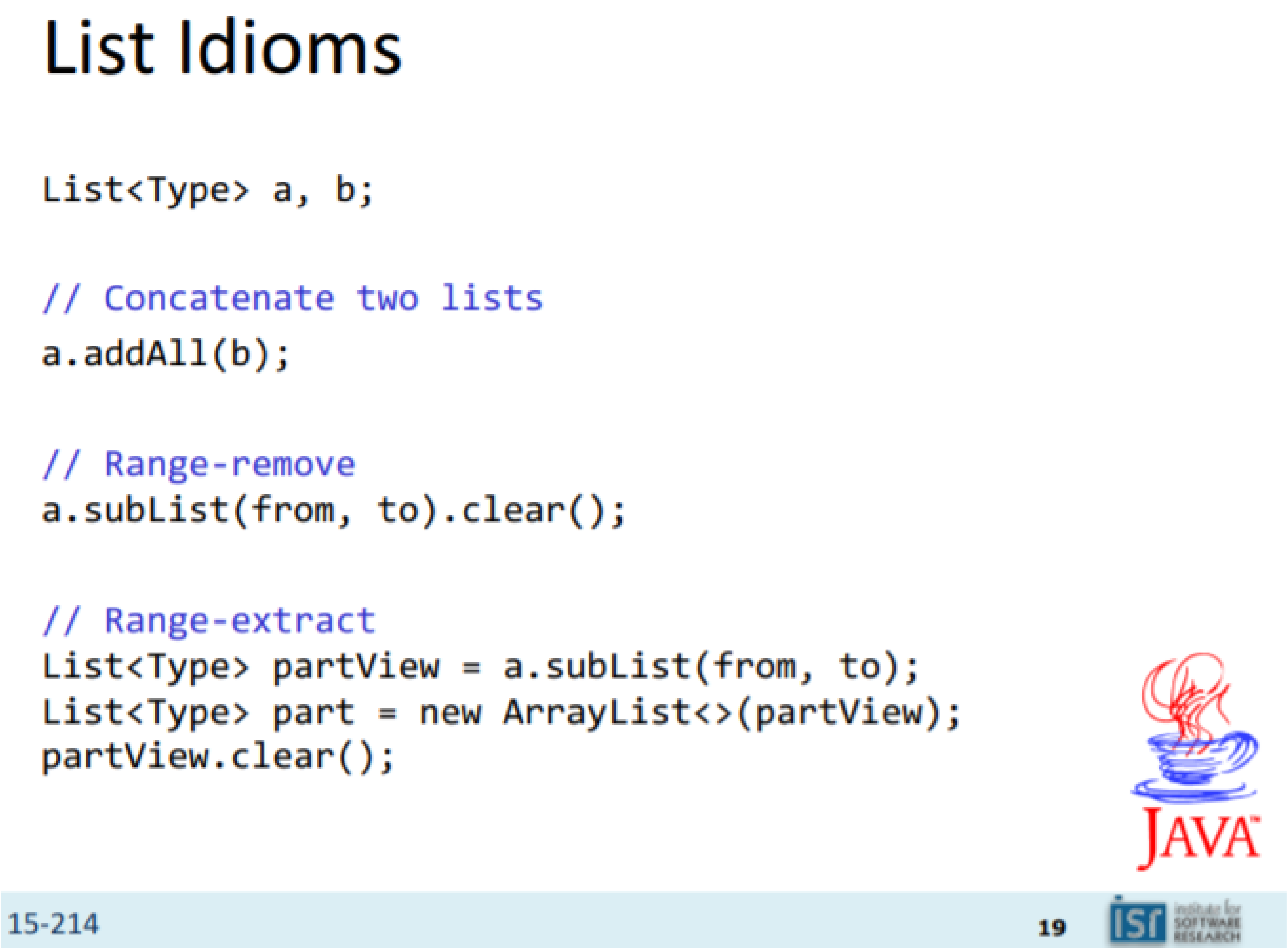


List Interface

A sequence of objects

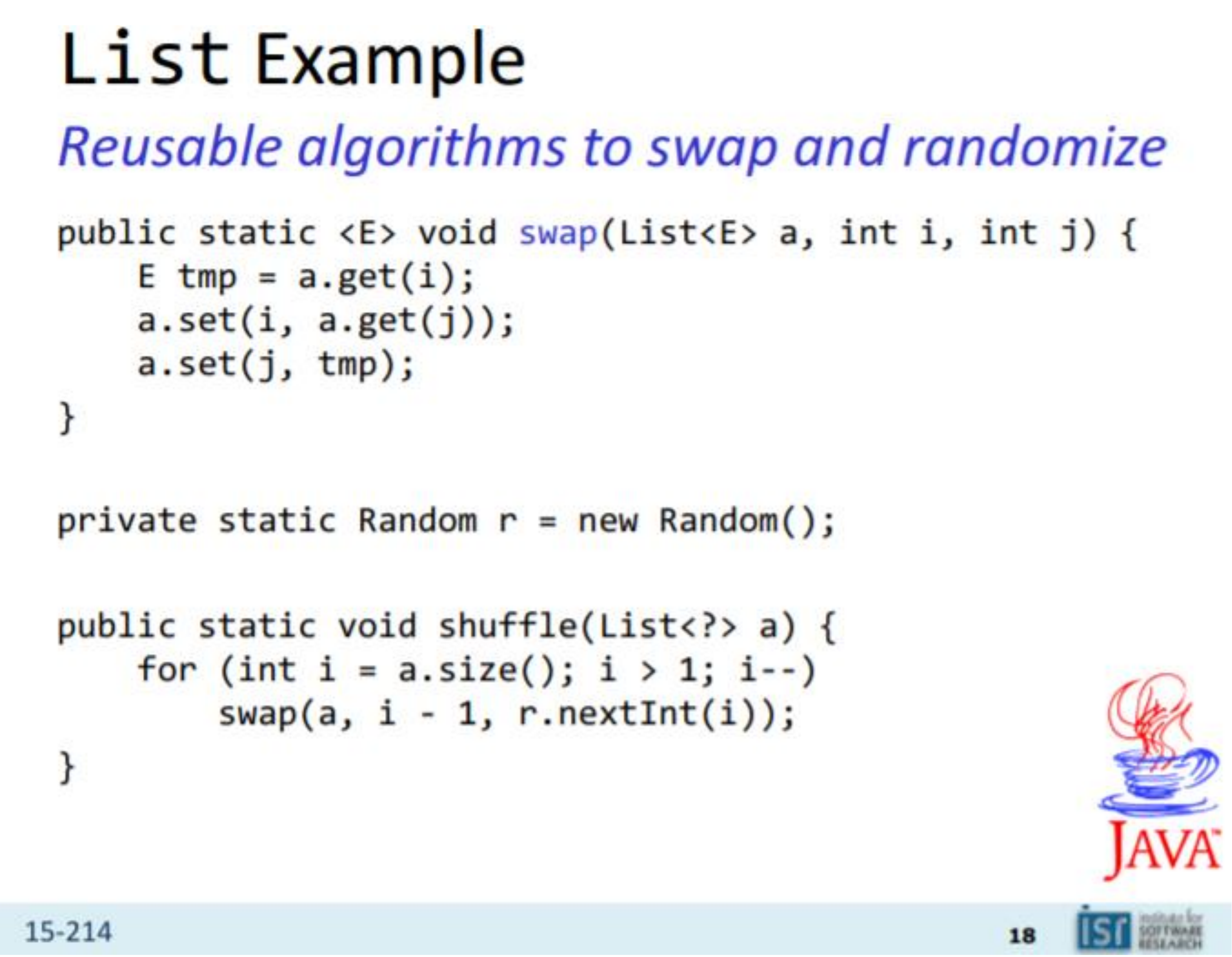
```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);       // Optional
    void add(int index, E element);    // Optional
    E       remove(int index);         // Optional
    boolean addAll(int index, Collection<? extends E> c);
                                       // Optional
    int indexOf(Object o);
    int lastIndexOf(Object o);

    List<E> subList(int from, int to);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
}
```

Question: Why using Object instead of E (generics)?



List Idioms

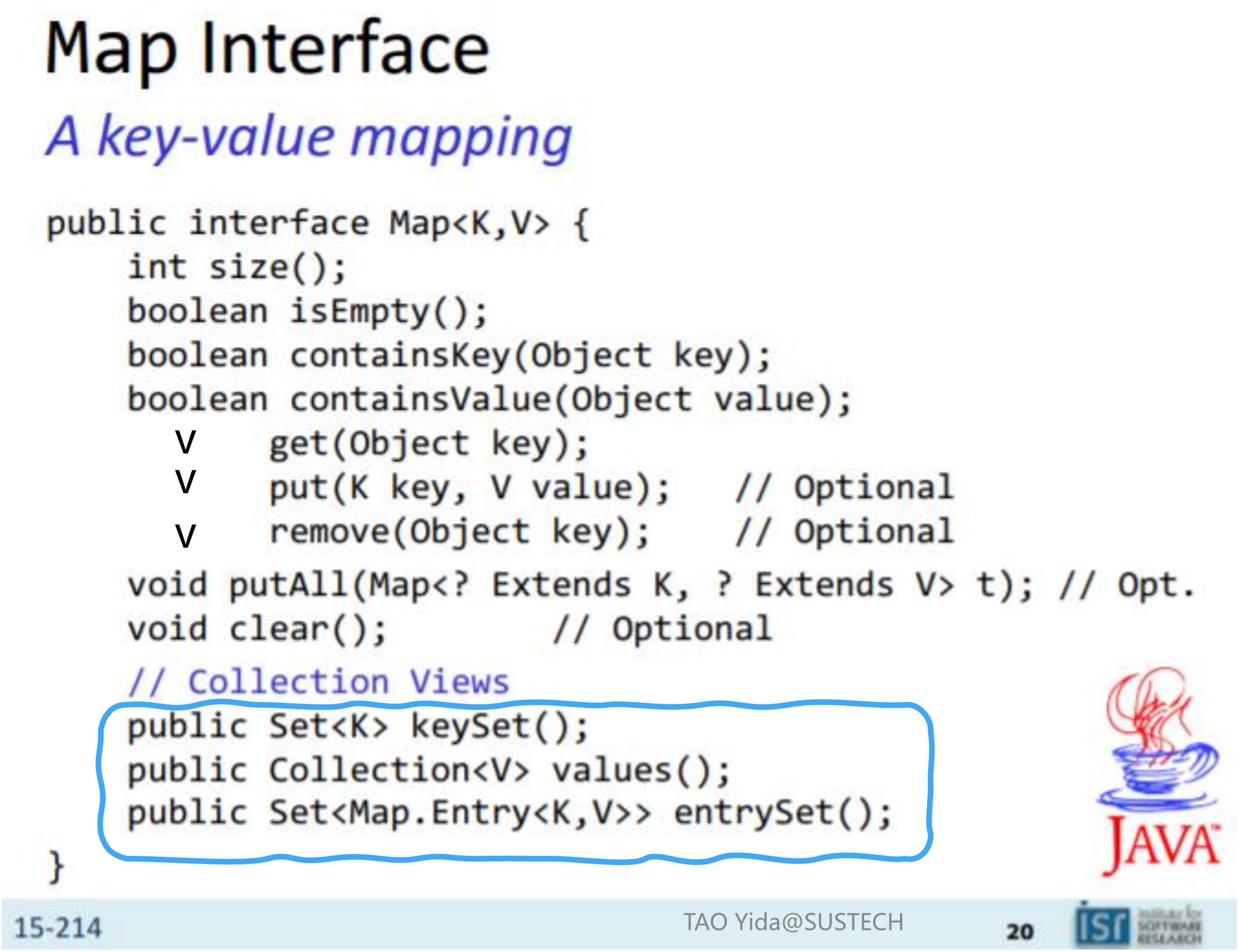
```
List<Type> a, b;

// Concatenate two lists
a.addAll(b);

// Range-remove
a.subList(from, to).clear();

// Range-extract
List<Type> partView = a.subList(from, to);
List<Type> part = new ArrayList<>(partView);
partView.clear();
```



List Example

Reusable algorithms to swap and randomize

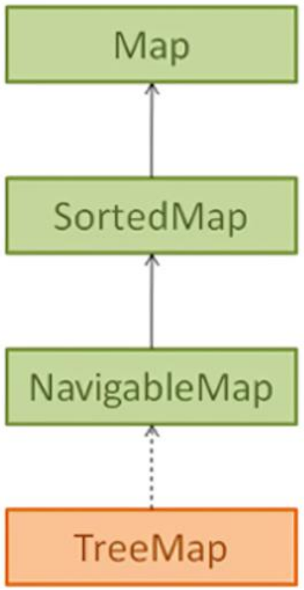
```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}

private static Random r = new Random();

public static void shuffle(List<?> a) {
    for (int i = a.size(); i > 1; i--)
        swap(a, i - 1, r.nextInt(i));
}
```



Map Interface

A key-value mapping

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
      V    get(Object key);
      V    put(K key, V value);    // Optional
      V    remove(Object key);     // Optional
    void putAll(Map<? Extends K, ? Extends V> t); // Opt.
    void clear();          // Optional
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
}
```

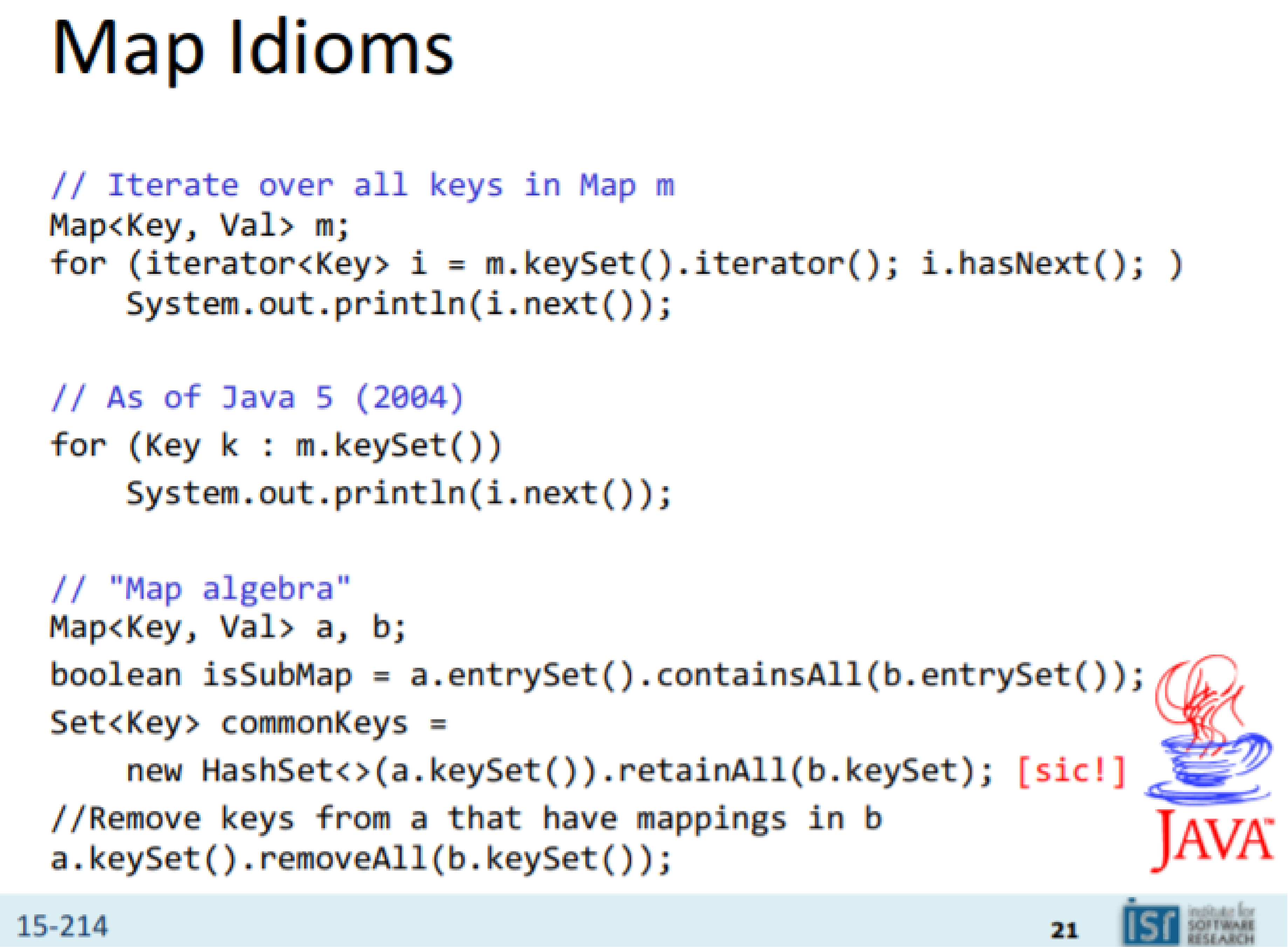


Map

SortedMap

NavigableMap

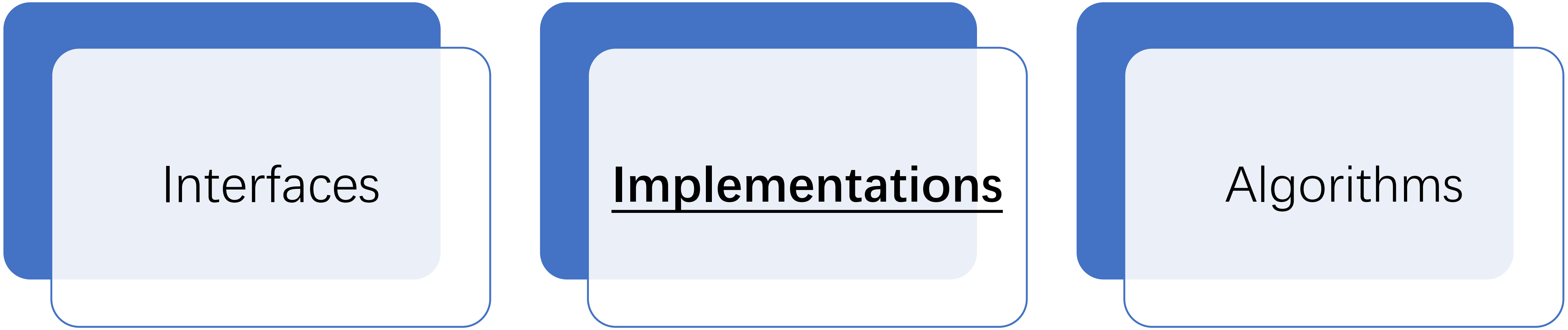
TreeMap



Map Idioms

```
// Iterate over all keys in Map m
Map<Key, Val> m;
for (iterator<Key> i = m.keySet().iterator(); i.hasNext(); )
    System.out.println(i.next());

// As of Java 5 (2004)
for (Key k : m.keySet())
    System.out.println(i.next());

// "Map algebra"
Map<Key, Val> a, b;
boolean isSubMap = a.entrySet().containsAll(b.entrySet());
Set<Key> commonKeys =
    new HashSet<>(a.keySet()).retainAll(b.keySet); [sic!]
//Remove keys from a that have mappings in b
a.keySet().removeAll(b.keySet());
```

Core Elements in the Java Collections Framework



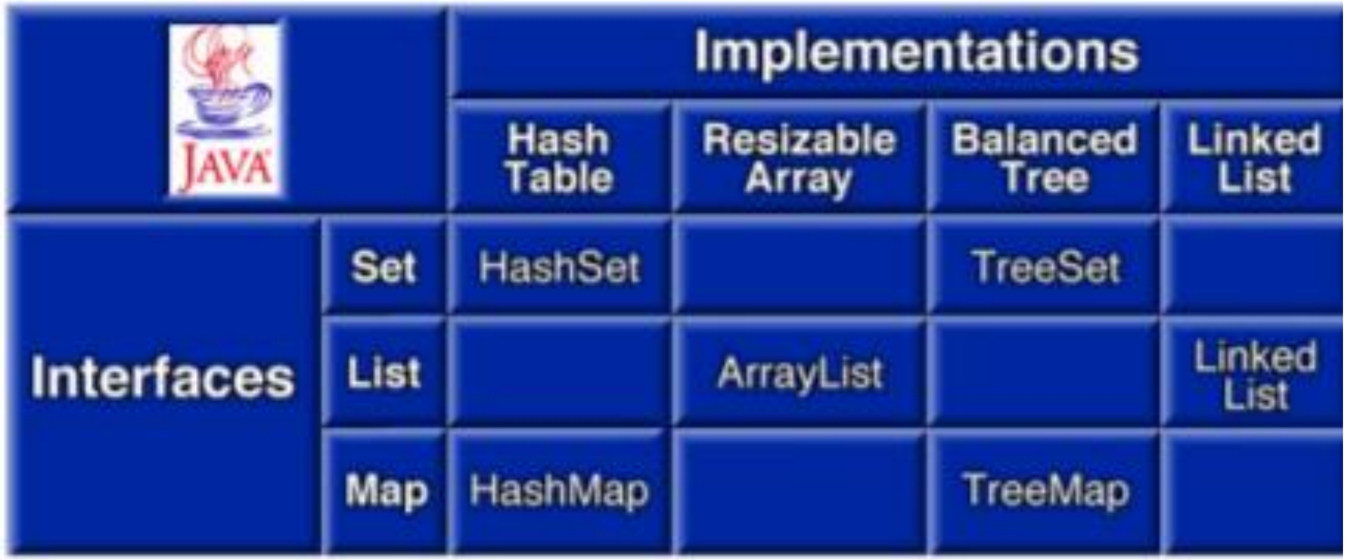
Interfaces

Implementations

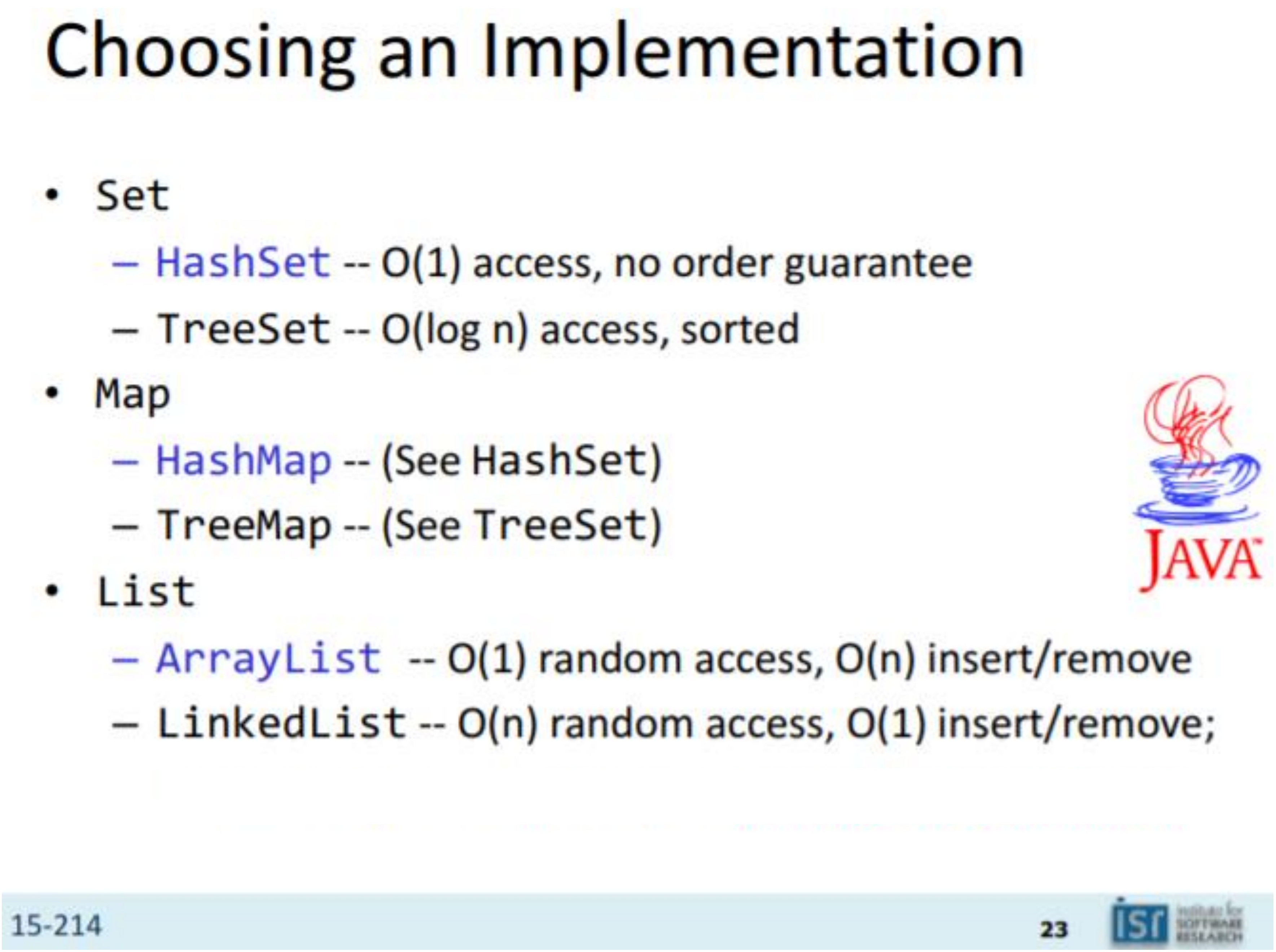
Algorithms

General-purpose Implementations

- The Collection framework provides several general-purpose implementations of the `Set`, `List` , and `Map` interfaces
- `HashSet`, `ArrayList`, and `HashMap` are most often used



		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		Linked List
	Map	HashMap		TreeMap	



Choosing an Implementation

- Set
  - HashSet -- O(1) access, no order guarantee
  - TreeSet -- O(log n) access, sorted
- Map
  - HashMap -- (See HashSet)
  - TreeMap -- (See TreeSet)
- List
  - ArrayList -- O(1) random access, O(n) insert/remove
  - LinkedList -- O(n) random access, O(1) insert/remove;

Implementation Behaviors

- All implementations permit `null` elements, keys, and values
- All are Serializable

- None are synchronized (i.e., not thread-safe by default)
  - Multiple threads could change the same collection, leading to inconsistent data
- All have `fail-fast` iterators
  - Detecting illegal concurrent modification during iteration and fail quickly and cleanly

Wrapper Implementations

- Add extra functionality on top of a collection (sound familiar?)
- `Synchronization Wrappers` add automatic synchronization (thread-safety) to an arbitrary collection (obsolete now and replace by concurrent collections)
- `Unmodifiable Wrappers` forbid the modification of the collection by intercepting all the operations that would modify the collection and throwing an `UnsupportedOperationException`

Convenience Implementations I

- `Arrays.asList(E[] a)` returns a List view of its array argument (allowing array to be “viewed” as list)
- Used as a bridge between array-based and collection-based APIs

```
List<String> list = Arrays.asList(new String[size]);
```

Convenience Implementations II

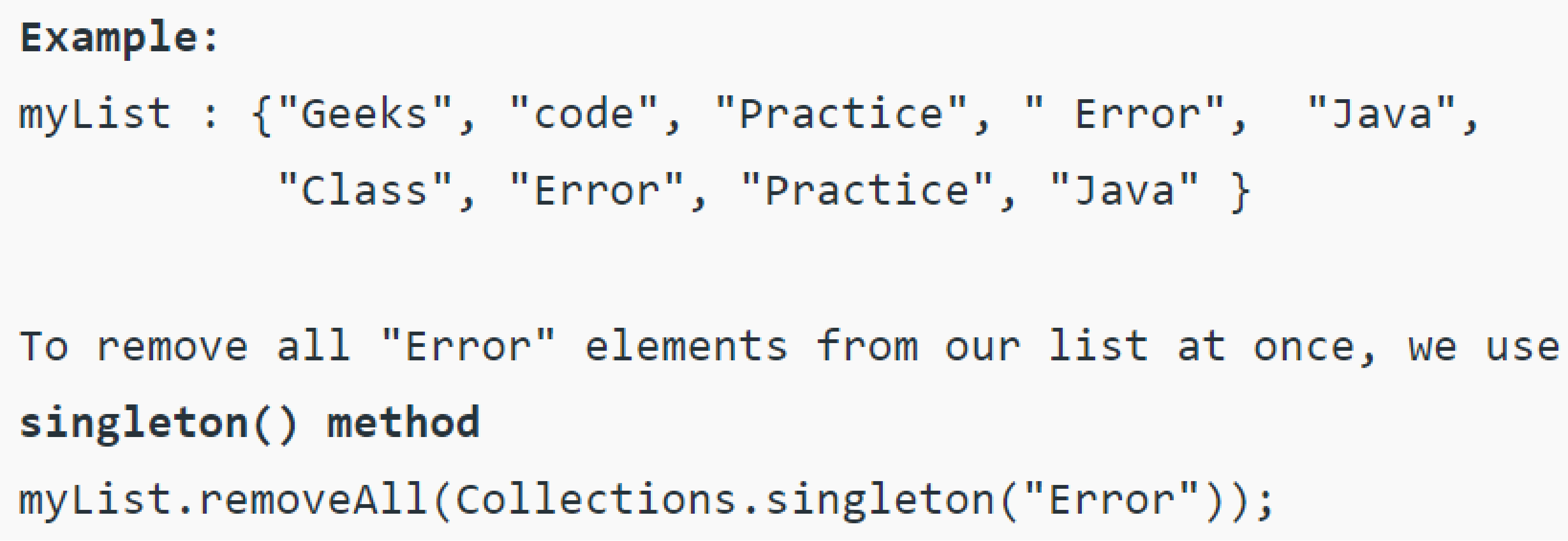
- `Collections.nCopies(int n, T o)` returns an immutable list consisting of n copies of the object o
- Useful in combination with the `List.addAll()` method to grow lists

```
List<Type> list = new ArrayList<Type>(Collections.nCopies(1000, (Type)null));
```

```
pets.addAll(Collections.nCopies(3, “cat”));
```

Convenience Implementations III

- `Collections.singleton(T o)` returns an immutable set containing only the specified object o
- Useful in combination with the `removeAll()` method to remove all occurrences of a specified element from a Collection



```
Example:
myList : {"Geeks", "code", "Practice", " Error",  "Java",
          "Class", "Error", "Practice", "Java" }


To remove all "Error" elements from our list at once, we use
singleton() method
myList.removeAll(Collections.singleton("Error"));
```

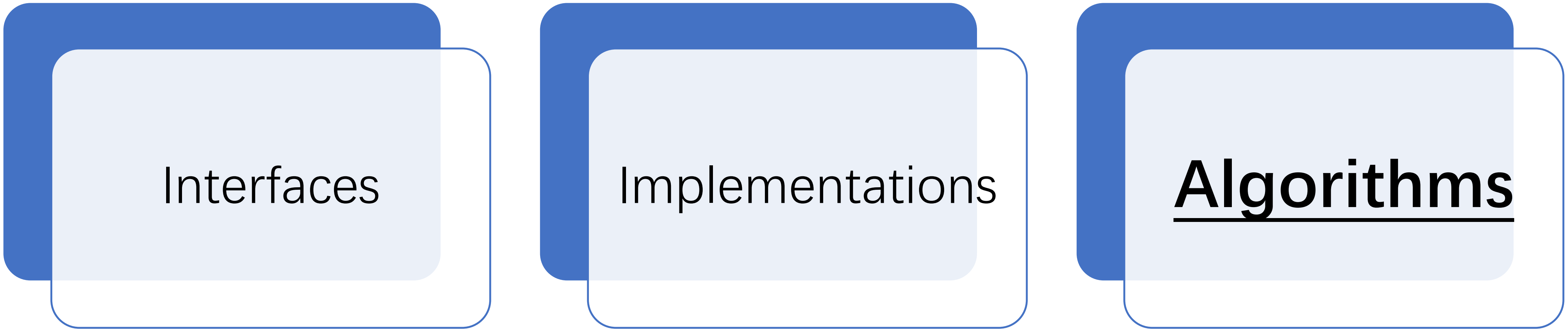
https://www.geeksforgeeks.org/collections-singleton-method-java/

Convenience Implementations IV

- The Collections class provides methods to return the empty Set, List, and Map — `emptySet()`, `emptyList()`, and `emptyMap()`
- Used as input to methods that take a Collection of values but you don't want to provide any values

```
tourist.declarePurchases(Collections.emptySet());
```

Core Elements in the Java Collections Framework

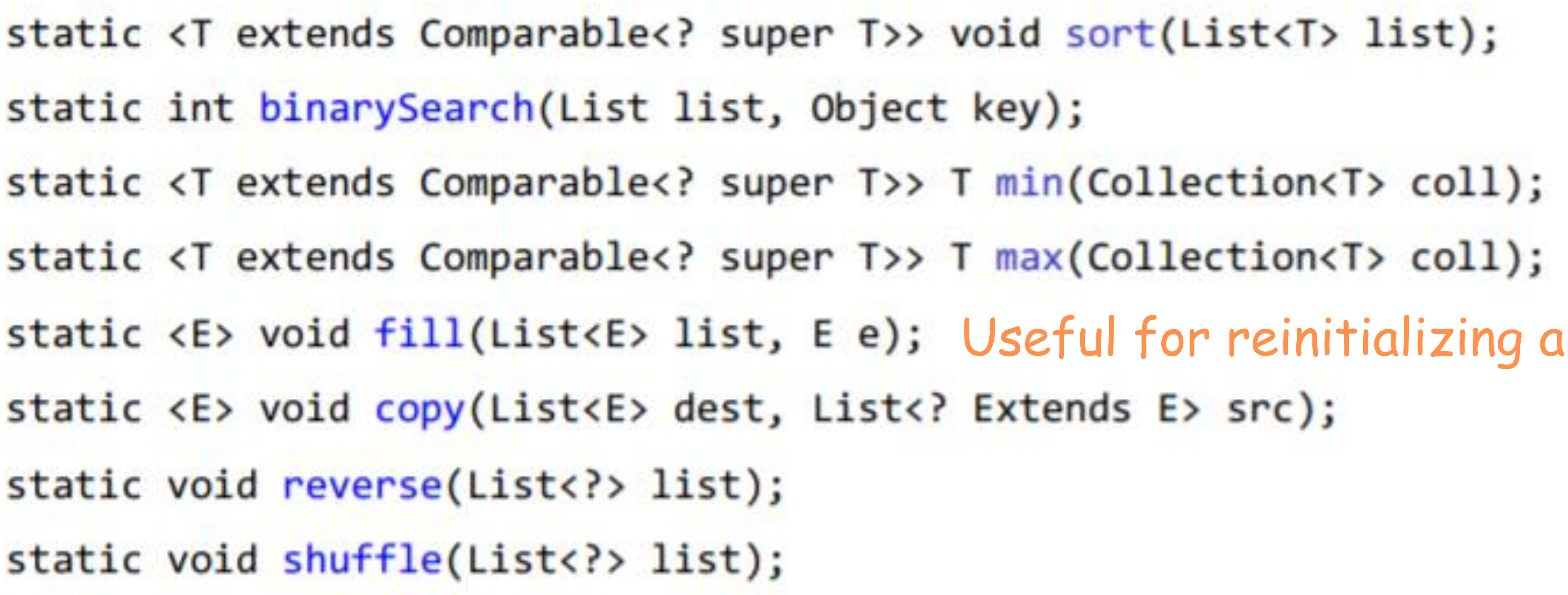


Interfaces

Implementations

Algorithms

Reusable Algorithms



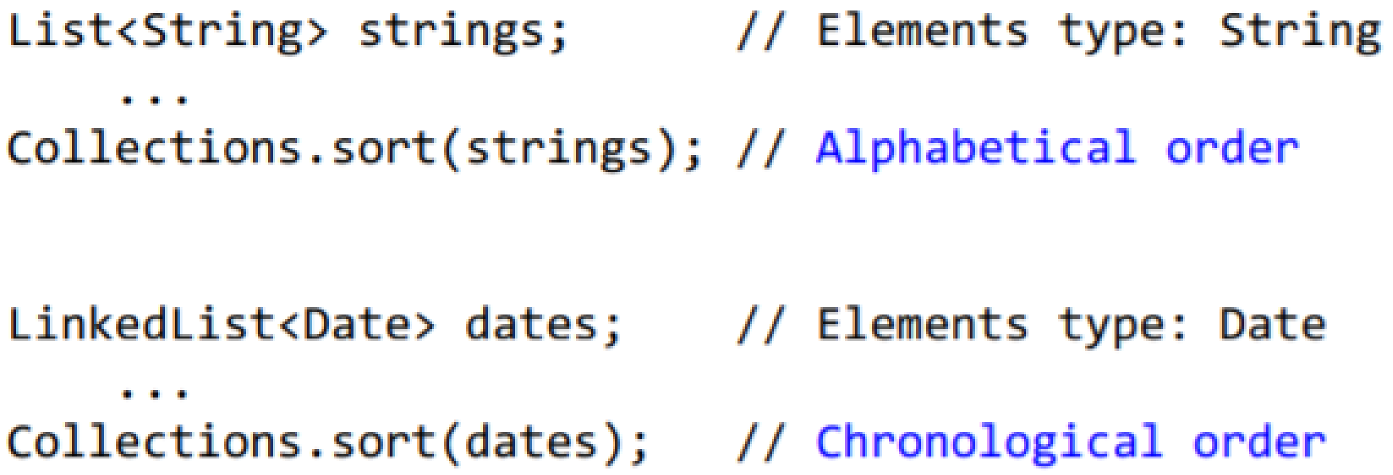
```
static <T extends Comparable<? super T>> void sort(List<T> list);
static int binarySearch(List list, Object key);
static <T extends Comparable<? super T>> T min(Collection<T> coll);
static <T extends Comparable<? super T>> T max(Collection<T> coll);
static <E> void fill(List<E> list, E e);
static <E> void copy(List<E> dest, List<? Extends E> src);
static void reverse(List<?> list);
static void shuffle(List<?> list);
```

Finding extreme values

Useful for reinitializing a list

Algorithm Example I

- `sort()` reorders a List according to an ordering relationship

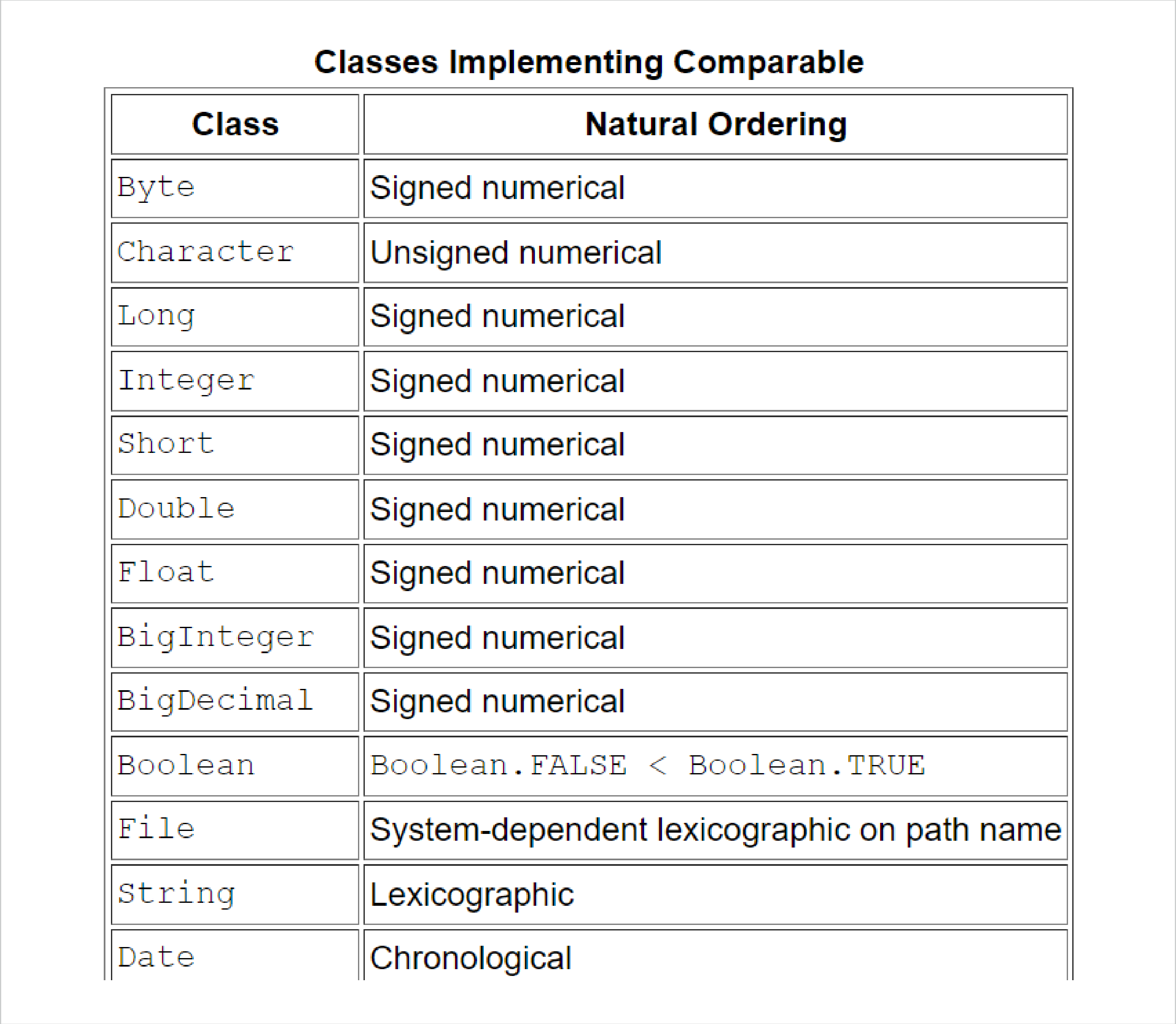


```
List<String> strings;      // Elements type: String
    ...
Collections.sort(strings); // Alphabetical order


LinkedList<Date> dates;    // Elements type: Date
    ...
Collections.sort(dates);   // Chronological order
```

How does this "smart sorting" happen?

Algorithm Example I

- `String` and `Date` both implement the `Comparable` interface (`compareTo(T o)`), allowing their objects to be sorted automatically
- `Collections.sort(list)` will throw a `ClassCastException` if elements do not implement `Comparable`



Classes Implementing Comparable

Class	Natural Ordering
`Byte`	Signed numerical
`Character`	Unsigned numerical
`Long`	Signed numerical
`Integer`	Signed numerical
`Short`	Signed numerical
`Double`	Signed numerical
`Float`	Signed numerical
`BigInteger`	Signed numerical
`BigDecimal`	Signed numerical
`Boolean`	`Boolean.FALSE < Boolean.TRUE`
`File`	System-dependent lexicographic on path name
`String`	Lexicographic
`Date`	Chronological

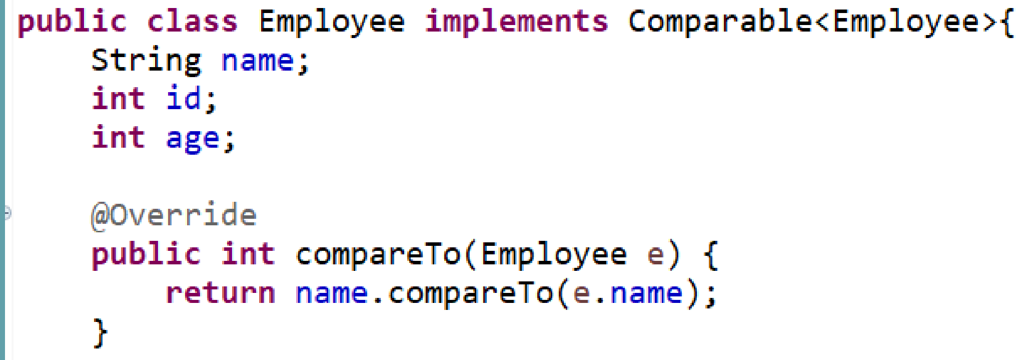
The `Comparator<T>` Interface

```
public interface Comparator<T>
```

- The `Comparable` interface is used to compare objects using one of their property as the default sorting order.
  - Provide `compareTo(T o)`
  - A comparable object can compare itself with another object

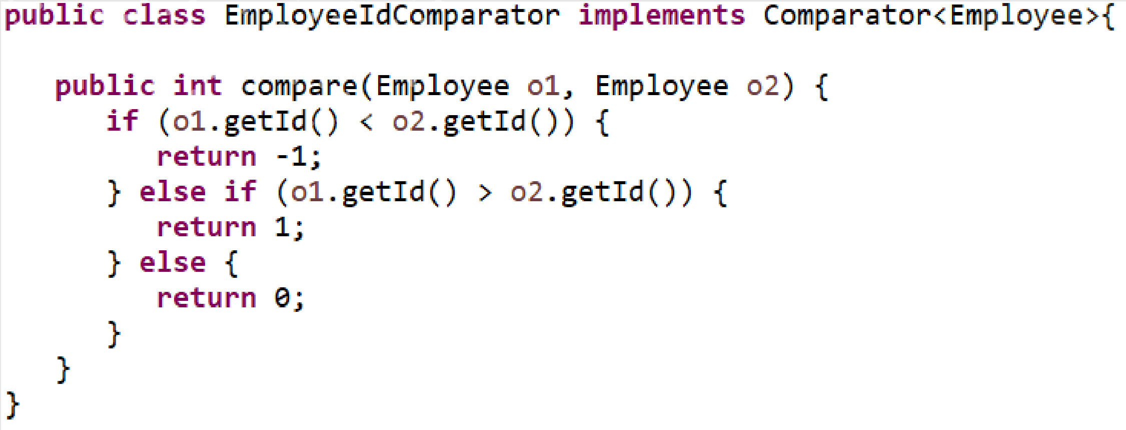
- The `Comparator` interface is used to compare two objects of the same class by different properties
  - Provide `compare(T o1, T o2)`
  - Comparator is a separate class and external to the element type being compared

Algorithm Example II

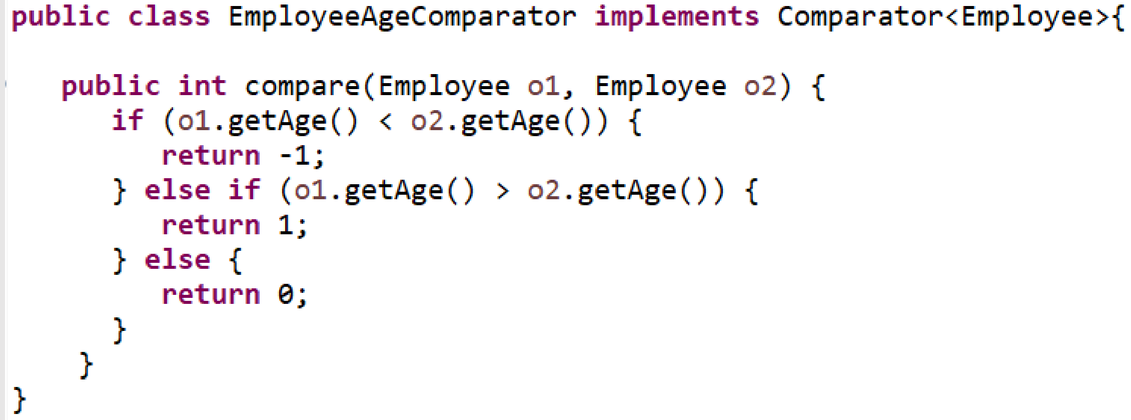


```
public class Employee implements Comparable<Employee>{
    String name;
    int id;
    int age;

    @Override
    public int compareTo(Employee e) {
        return name.compareTo(e.name);
    }
```

Default ordering is by name

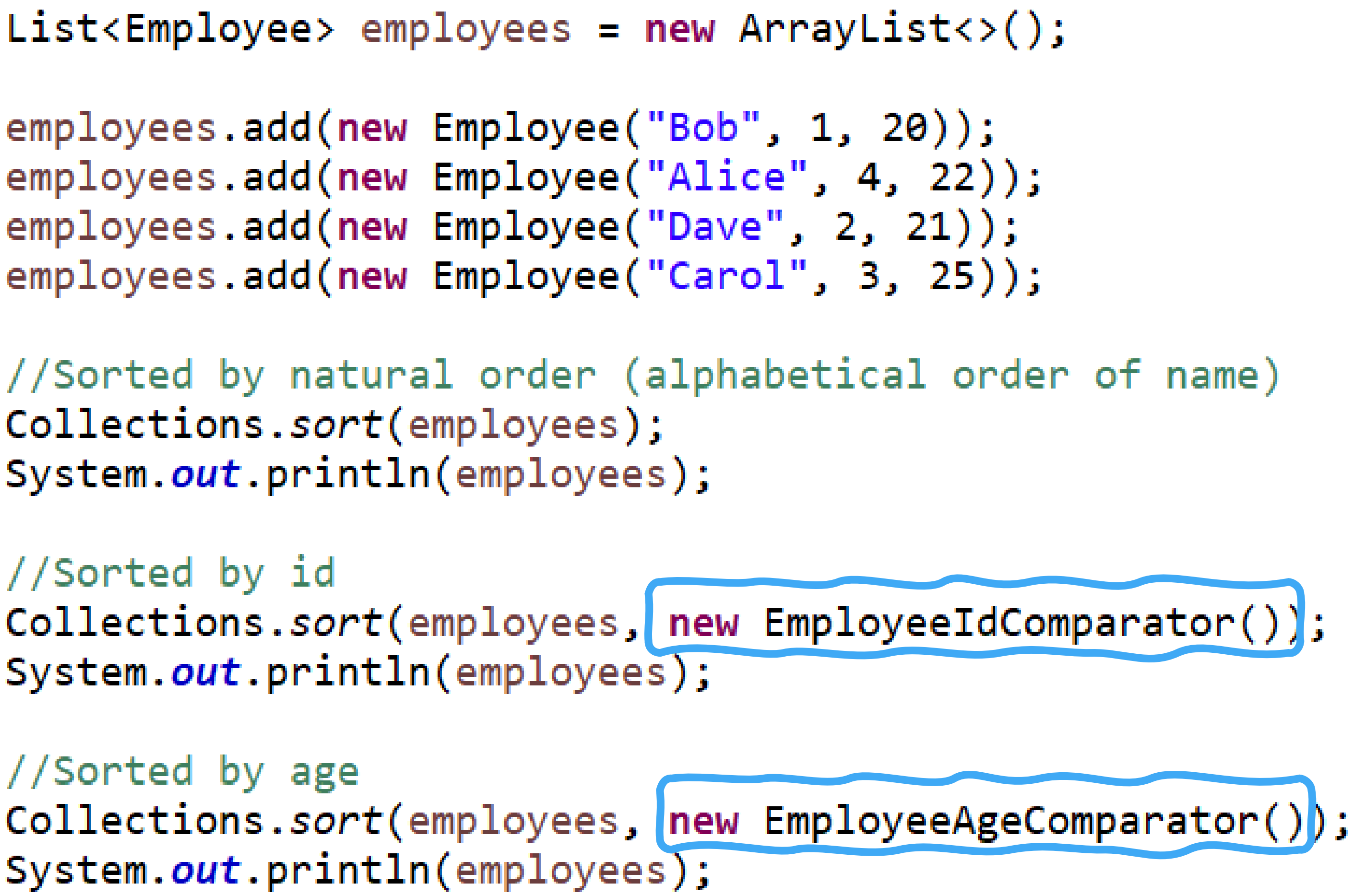


```
public class EmployeeIdComparator implements Comparator<Employee>{

   public int compare(Employee o1, Employee o2) {
      if (o1.getId() < o2.getId()) {
         return -1;
      } else if (o1.getId() > o2.getId()) {
         return 1;
      } else {
         return 0;
      }
   }
}
```

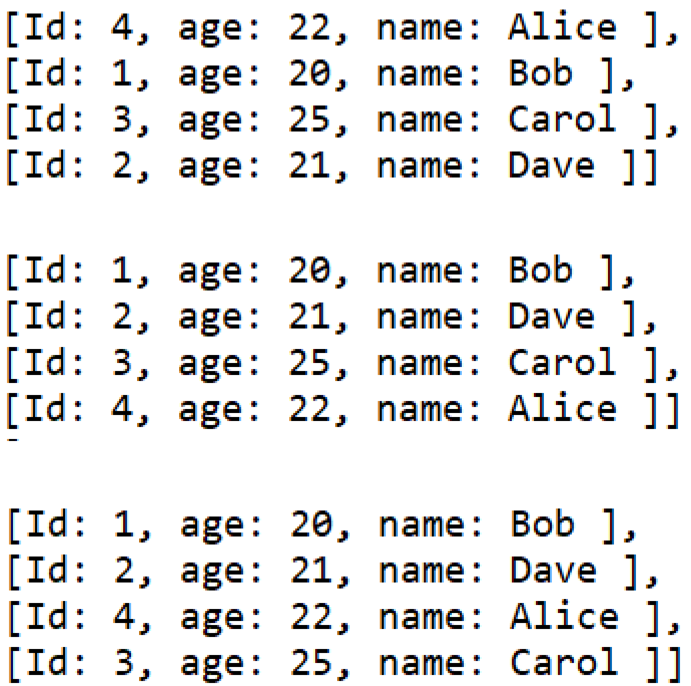


```
public class EmployeeAgeComparator implements Comparator<Employee>{

   public int compare(Employee o1, Employee o2) {
      if (o1.getAge() < o2.getAge()) {
         return -1;
      } else if (o1.getAge() > o2.getAge()) {
         return 1;
      } else {
         return 0;
      }
    }
}
```

Algorithm Example II

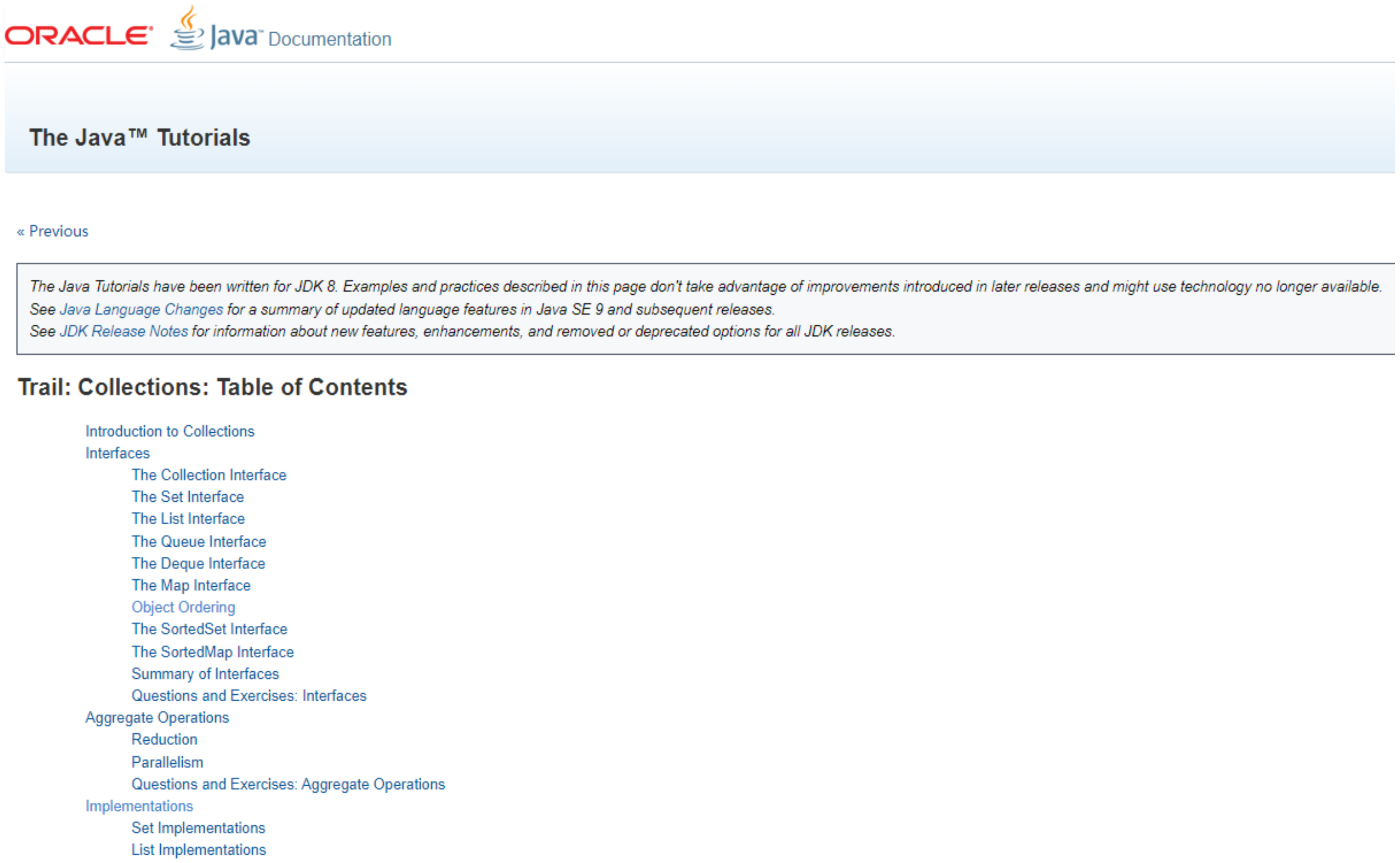


```java
List<Employee> employees = new ArrayList<>();

employees.add(new Employee("Bob", 1, 20));
employees.add(new Employee("Alice", 4, 22));
employees.add(new Employee("Dave", 2, 21));
employees.add(new Employee("Carol", 3, 25));

//Sorted by natural order (alphabetical order of name)
Collections.sort(employees);
System.out.println(employees);

//Sorted by id
Collections.sort(employees, new EmployeeIdComparator());
System.out.println(employees);

//Sorted by age
Collections.sort(employees, new EmployeeAgeComparator());
System.out.println(employees);
```



```
[Id: 4, age: 22, name: Alice ],
[Id: 1, age: 20, name: Bob ],
[Id: 3, age: 25, name: Carol ],
[Id: 2, age: 21, name: Dave ]]

[Id: 1, age: 20, name: Bob ],
[Id: 2, age: 21, name: Dave ],
[Id: 3, age: 25, name: Carol ],
[Id: 4, age: 22, name: Alice ]]

[Id: 1, age: 20, name: Bob ],
[Id: 2, age: 21, name: Dave ],
[Id: 4, age: 22, name: Alice ],
[Id: 3, age: 25, name: Carol ]]
```

Further Reading



The Java™ Tutorials

« Previous

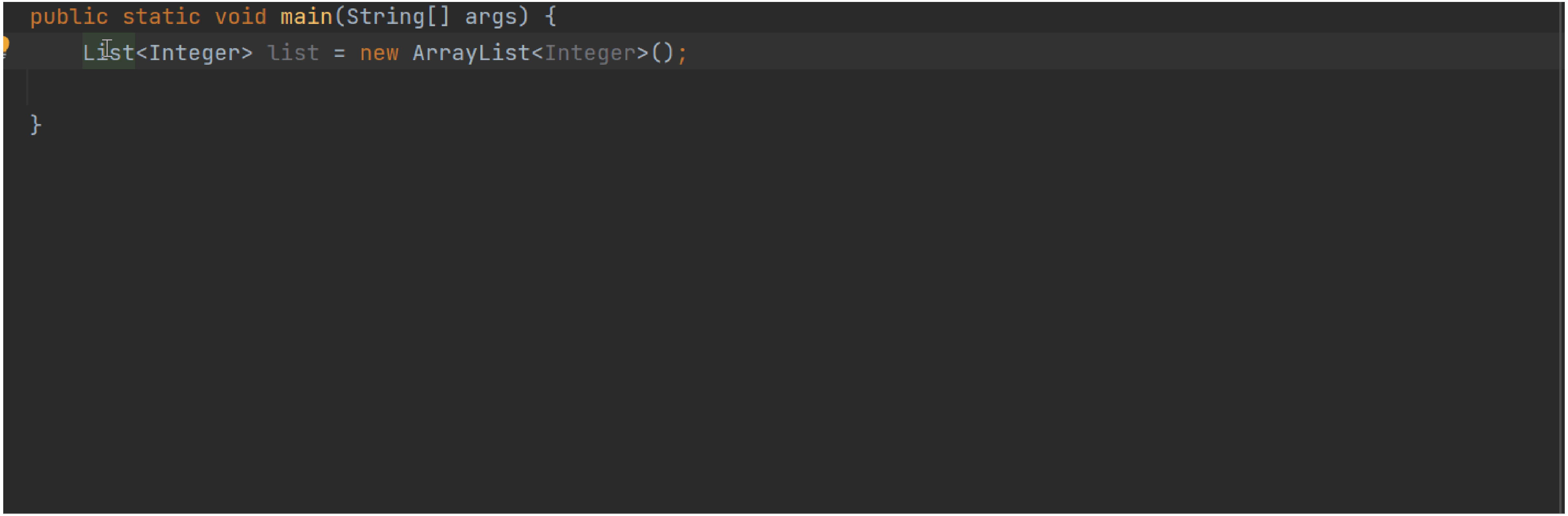
The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Trail: Collections: Table of Contents

- Introduction to Collections
- Interfaces
    - The Collection Interface
    - The Set Interface
    - The List Interface
    - The Queue Interface
    - The Deque Interface
    - The Map Interface
    - Object Ordering
    - The SortedSet Interface
    - The SortedMap Interface
    - Summary of Interfaces
    - Questions and Exercises: Interfaces
- Aggregate Operations
    - Reduction
    - Parallelism
    - Questions and Exercises: Aggregate Operations
- Implementations
    - Set Implementations
    - List Implementations

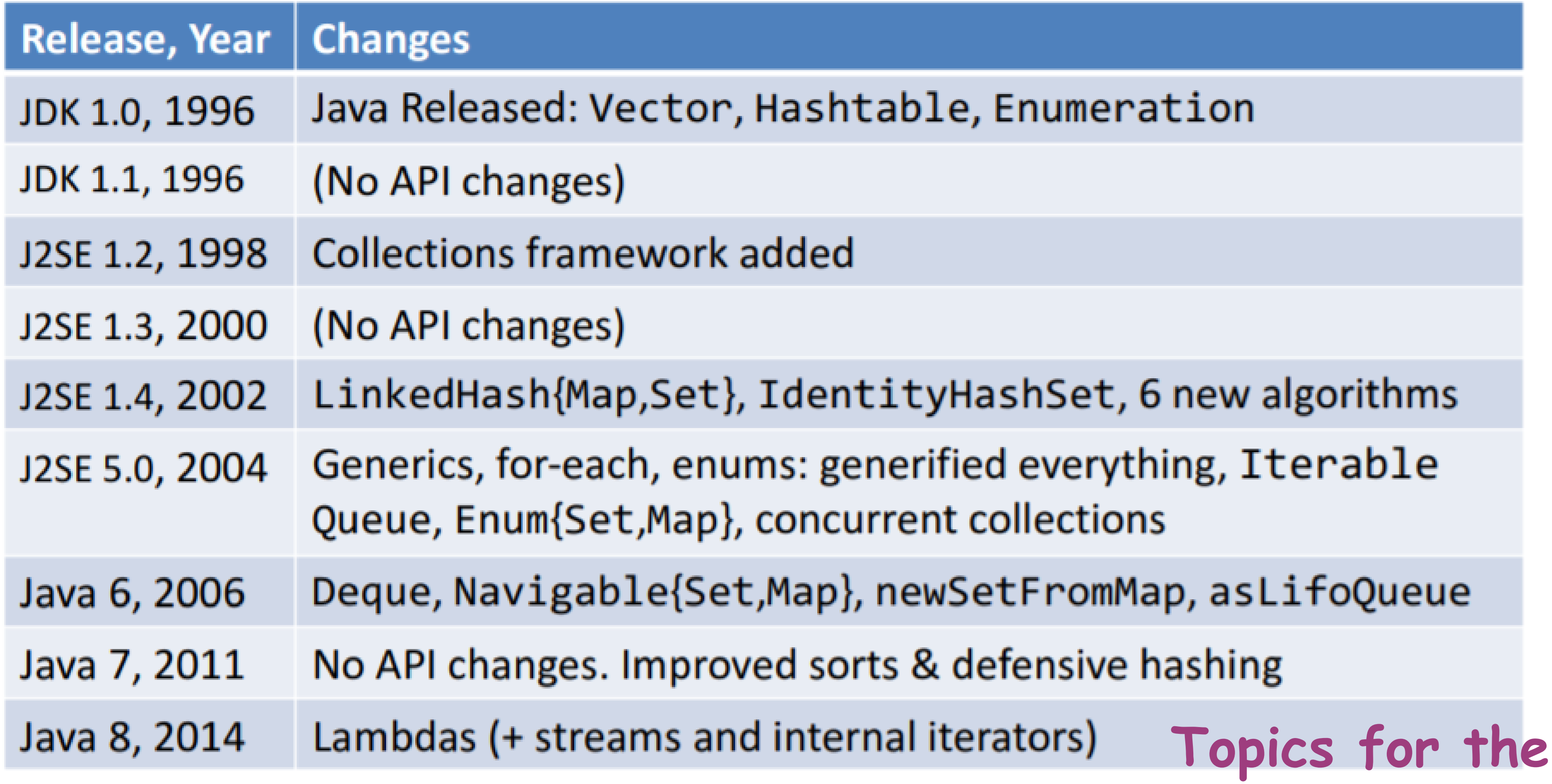
https://docs.oracle.com/javase/tutorial/collections/TOC.html

Get Documentation from IDE



```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();

}
```

Evolution of Java Collections



Release, Year	Changes
JDK 1.0, 1996	Java Released: Vector, Hashtable, Enumeration
JDK 1.1, 1996	(No API changes)
J2SE 1.2, 1998	Collections framework added
J2SE 1.3, 2000	(No API changes)
J2SE 1.4, 2002	LinkedHash{Map,Set}, IdentityHashSet, 6 new algorithms
J2SE 5.0, 2004	Generics, for-each, enums: generified everything, Iterable Queue, Enum{Set,Map}, concurrent collections
Java 6, 2006	Deque, Navigable{Set,Map}, newSetFromMap, asLifoQueue
Java 7, 2011	No API changes. Improved sorts & defensive hashing
Java 8, 2014	Lambdas (+ streams and internal iterators)

Topics for the next lecture

https://www.cs.cmu.edu/~charlie/courses/15-214/2016-fall/slides/15-collections%20design.pdf

Next Lecture

- Functional Programming
- Lambda Expressions
- Streams API