

# 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 2

---

- Generics
- Abstract Data Type (ADT)
- Collections



What is  
Generics?

Why do we  
need Generics?

# A World without Generics

```
public class ArrayList {  
    private Object[] elements;  
    .....  
    public Object get(int i){...}  
    public void add(Object o){...}  
}
```

# A World without Generics

Drawback 1: Explicit casting is required; inefficient and hard to read

Compiler error:

Type mismatch: cannot convert  
from Object to String



```
ArrayList list = new ArrayList();  
list.add("hi");  
String s = list.get(0);
```

Need to explicitly cast to String

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

# A World without Generics

Drawback 2: error-prone; may cause type-related runtime errors if a programmer makes a mistake with the explicit casting.

No compilation error here  
(no error checking)

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

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

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



# Solution?

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

- Introduced in JDK 5.0
- 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



# Solution: Generics

```
// Code is easier to read
// You can tell right away that this list contains String
ArrayList<String> list = new ArrayList<String>();
list.add("Hello");

// Compiler checks that you don't insert object
// of the wrong type
list.add(2022); ❌

// No explicit cast is required
// Compiler will add the correct type cast
String elem = list.get(0);
```

# 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; prone to runtime errors (crash)

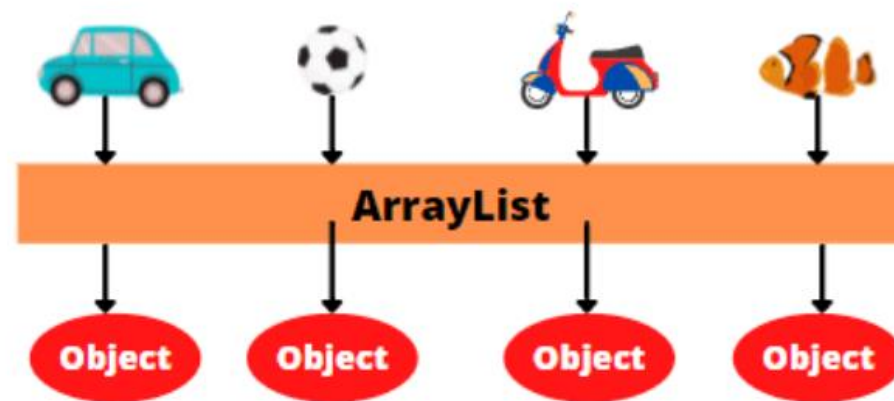
Could only put the specified element; otherwise compiler will complain

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) Type variable
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?

# Using Generics

- Generic classes
- Generic interfaces
- Generic methods

Classes in Java Collections (e.g., List, Queue, Set) are typically generic classes

```
/**
 * @version 1.00 2004-05-10
 * @author Cay Horstmann
 */
public class Pair<T>
{
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second) {
        this.first = first; this.second = second;
    }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

# Using Generics

- Generic classes
- **Generic interfaces**
- Generic methods

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

## 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

Image source: [https://www.cs.rit.edu/~rlaz/cs2/slides/CS2\\_Week5.pdf](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;
}
```

You can define generic methods both inside ordinary classes and inside generic classes

Example from “Effective Java”

# Bounds for Type Variables

`<T extends BoundingType>`

- T could be any subtype of the bounding type
- Both T and bounding type can be either a class or an interface
- Multiple bounds are allowed, separated by & (a class must be the first one in the bounds list)

`<T extends Animal & Comparable>`



# Bounds for Type Variables

```
public static <T extends Comparable> Pair<T> minmax(T[] a)
{
    if (a == null || a.length == 0) return null;
    T min = a[0];
    T max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.compareTo(a[i]) > 0) min = a[i];
        if (max.compareTo(a[i]) < 0) max = a[i];
    }
    return new Pair<>(min, max);
}
```

```
min = 2
max = is
min = 1815-12-10
max = 1910-06-22
```

```
String[] words = {"This", "is", "CS209a", "Java", "2"};
Pair<String> mm1 = minmax(words);
System.out.println("min = " + mm1.getFirst());
System.out.println("max = " + mm1.getSecond());

LocalDate[] birthdays =
{
    LocalDate.of(1906, 12, 9), // G. Hopper
    LocalDate.of(1815, 12, 10), // A. Lovelace
    LocalDate.of(1903, 12, 3), // J. von Neumann
    LocalDate.of(1910, 6, 22), // K. Zuse
};

Pair<LocalDate> mm2 = minmax(birthdays);
System.out.println("min = " + mm2.getFirst());
System.out.println("max = " + mm2.getSecond());
```

Example adapted from "Core Java Volume II"

# Inheritance Rules for Generic Types

```
public void process0(Object o){}
```

```
process0(0);  
process0("test");  
process0(100.00);
```

All code works since any thing is a Object

What if the method wants to take a List that can have any type of element?

# Inheritance Rules for Generic Types

```
public static void process1(List<Object> list){}
```

```
List<String> ln = new ArrayList<>();  
List<Integer> li = new ArrayList<>();  
process1(ln); // compilation error  
process1(li); // compilation error
```

Compiler will complain on type mismatch:

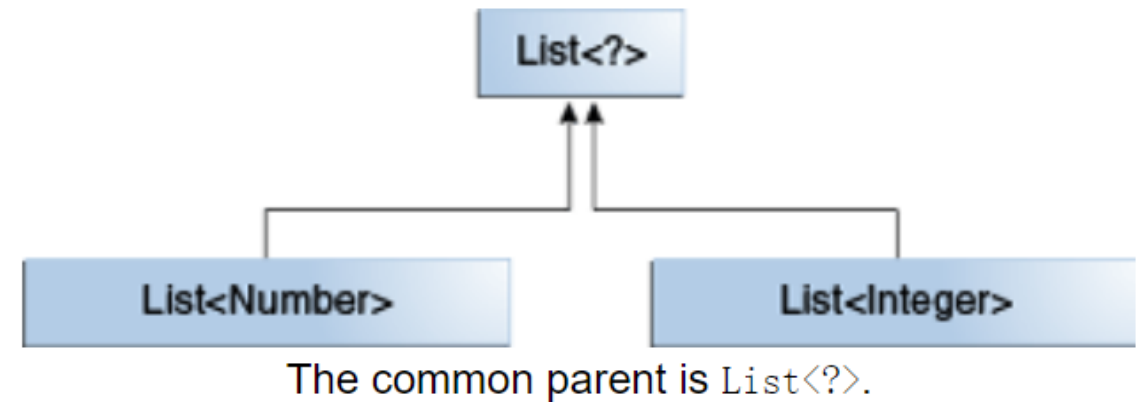
List<String> has no relationship to List<Object>, even though String is a subtype of Object

# Wildcards (通配符)

- Use “?” to create a relationship between generic types
- List<?> could be List<Number>, List<Integer>, List<String>, etc.

```
public static void process2(List<?> list){}
```

```
List<String> ln = new ArrayList<>();  
List<Integer> li = new ArrayList<>();  
process2(ln);  
process2(li);
```



# Wildcards

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

# Type Erasure

- To be compatible with previous versions, the implementation of Java generics adopts the strategy of pseudo generics
- Java supports generics in syntax, but the so-called “type erase” will be carried out in the compilation stage to replace all generic representations (contents in angle brackets) with specific types
- To JVM, there is no generics at all

<https://developpaper.com/detailed-explanation-of-type-erasure-examples-of-java-generics/>

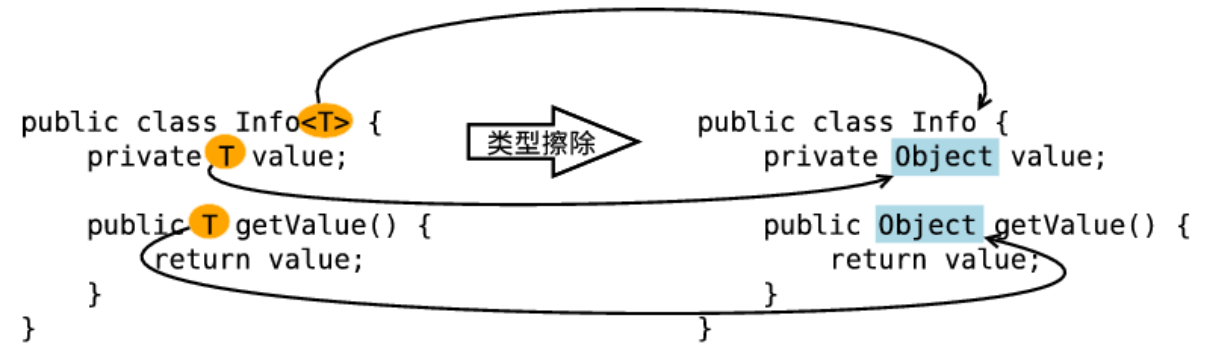


Figure 1: erasing type parameters in class definitions

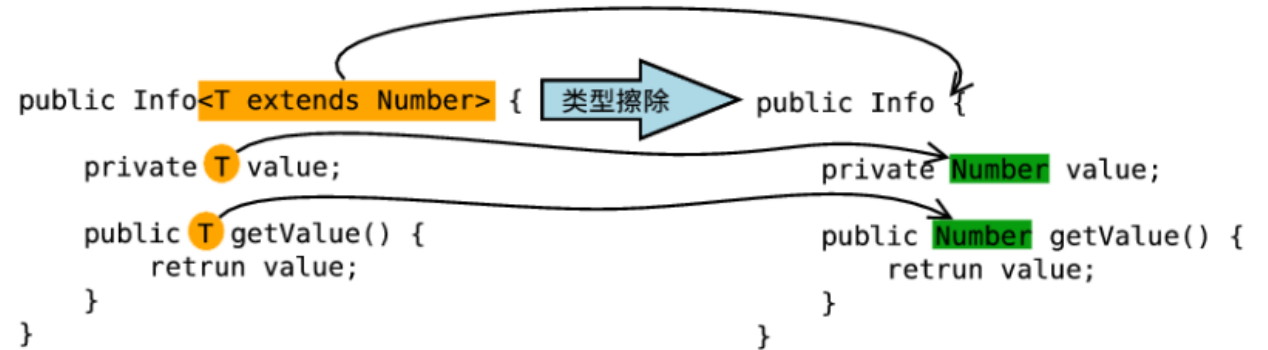


Figure 2: restricted type parameters in erase class definition

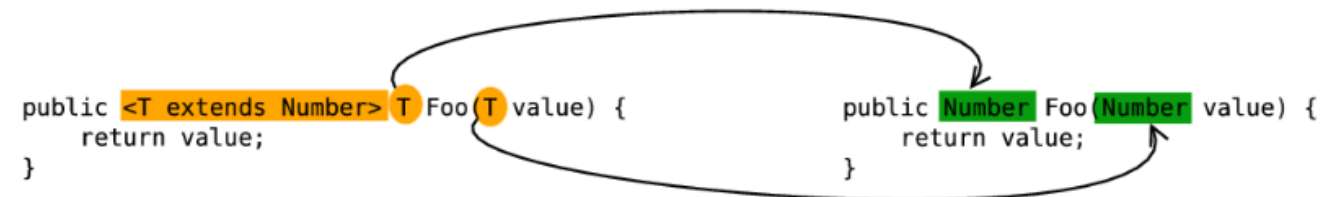


Figure 3: erasing type parameters in generic methods



# Lecture 2

---

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



# Abstract Data Type (ADT)

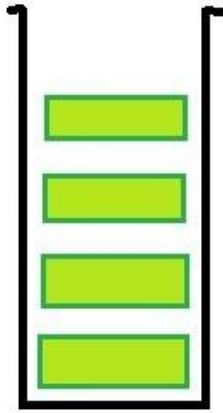
- A type (or class) for objects whose behavior is defined by a set of **values** and a set of **operations**.
  - How **values** are stored in memory is **hidden** from the client
  - How **operations** are implemented internally is **hidden** from client



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

# Stack ADT

a) Conceptual



b) Physical Structure

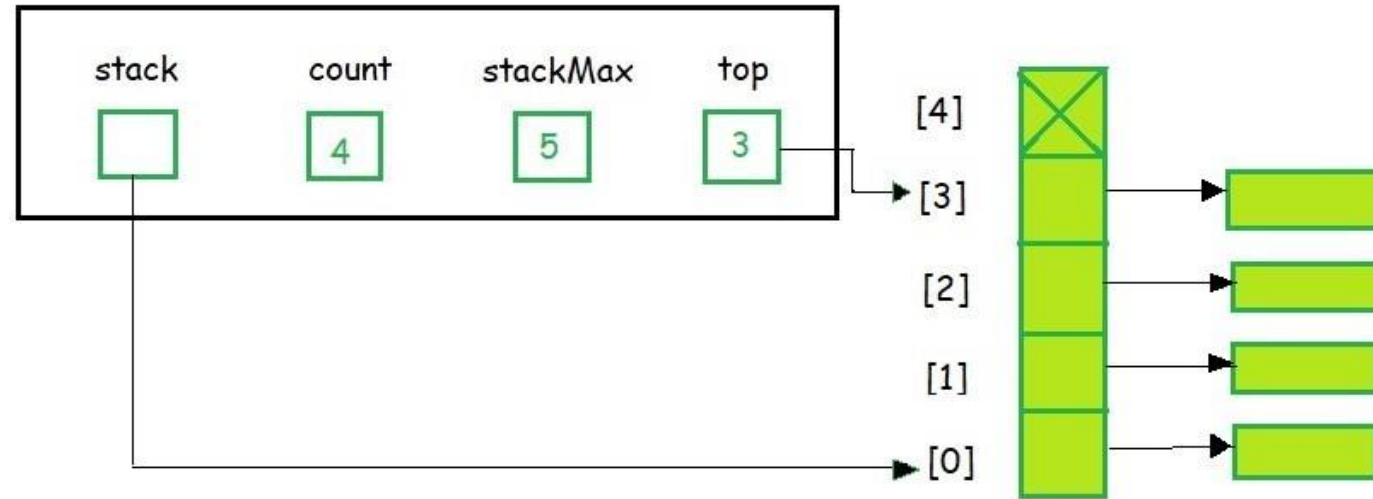


Image source: <https://www.geeksforgeeks.org/abstract-data-types/>

## 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

- Conceptual: a series of elements with insertion and deletion operations
- Possible implementations
  - Using an array
  - Using a linked list (nodes with references to one another)

**Which implementation is better? Why?**

# Operations of ADT

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

- **Creators** create new objects of the type (e.g., constructors)
- **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
  - For immutable types, there is no mutator operation



# Lecture 2

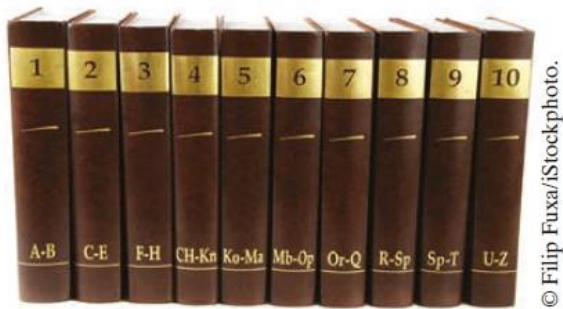
---

- Generics
- Abstract Data Type (ADT)
- Collections



# Concepts of Collections List, Stack, Map and Set?

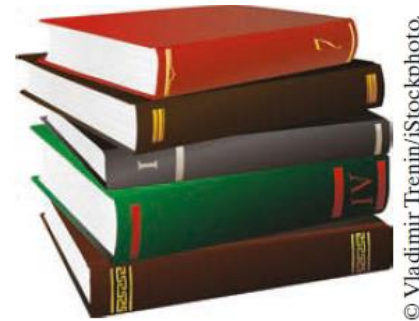
A list is a collection that remembers the order of its elements.



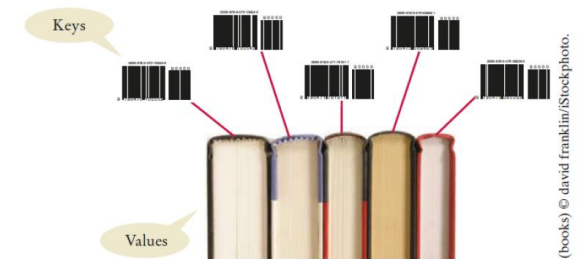
A set is an unordered collection of unique elements.



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



A map keeps associations between key and value objects.



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



Joshua Bloch

Updated  
for  
Java 9

# Effective Java

Third Edition

Best practices for



...the Java Platform

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

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

# Collections

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

## The Java™ Platform Collections Framework

Joshua Bloch  
Sr. Staff Engineer, Collections Architect  
Sun Microsystems, Inc.



15-214



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

# Core Elements in the Java Collections Framework



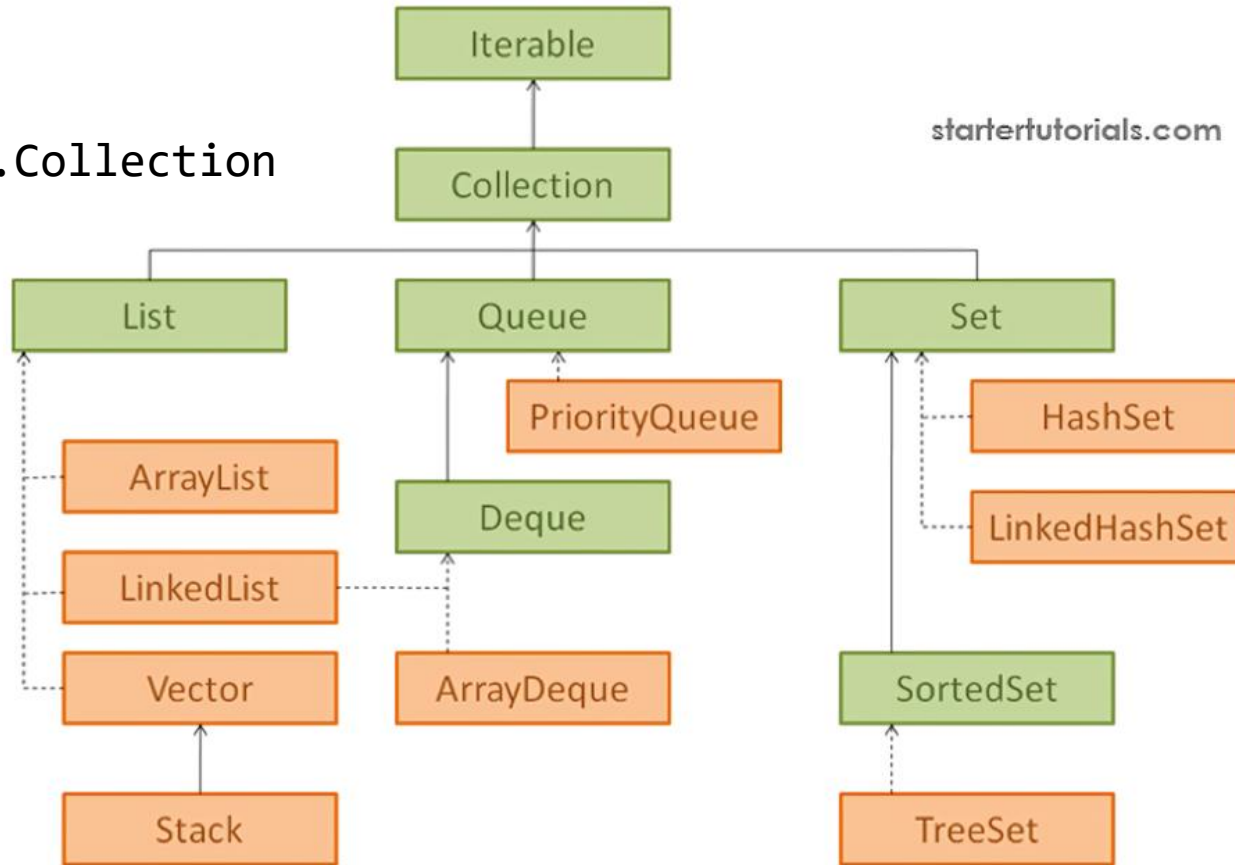
**Interfaces**

Implementations

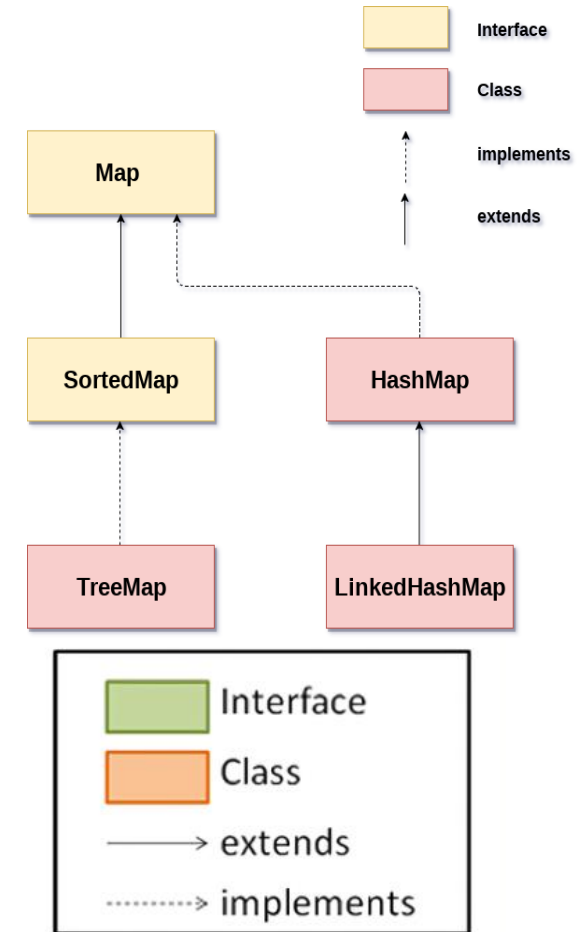
Algorithms

# Collection Class Hierarchy

java.util.Collection



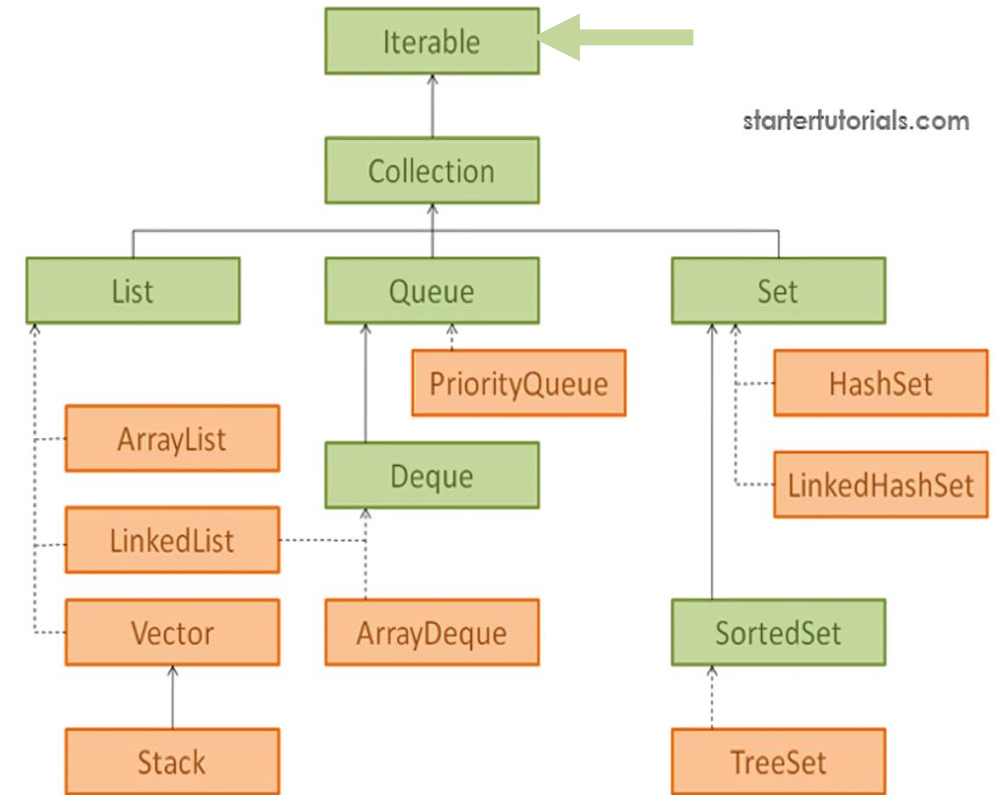
java.util.Map



# 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(); Next slide
```

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

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

Generic utility methods that operate on any kind of collection

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

```
}
```



# 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

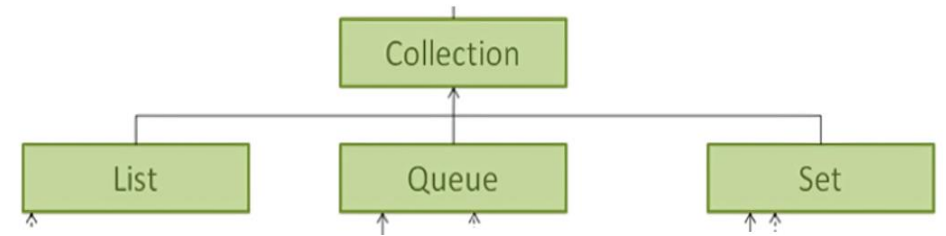
Iterators allow the caller to remove elements from the underlying collection during the iteration

```
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);
```

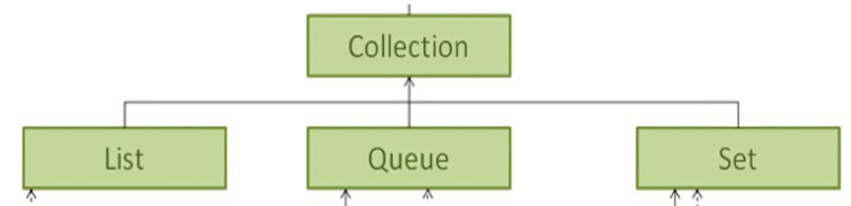
```
Collection<Type> c;
```

```
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();
```

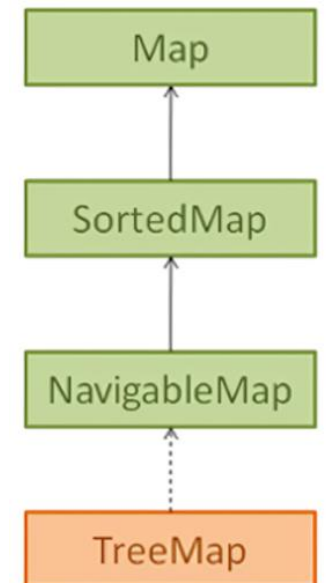




# Map Interface

*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 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



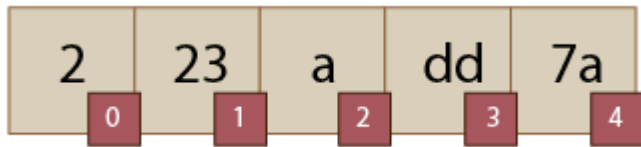
# General-purpose Implementations

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

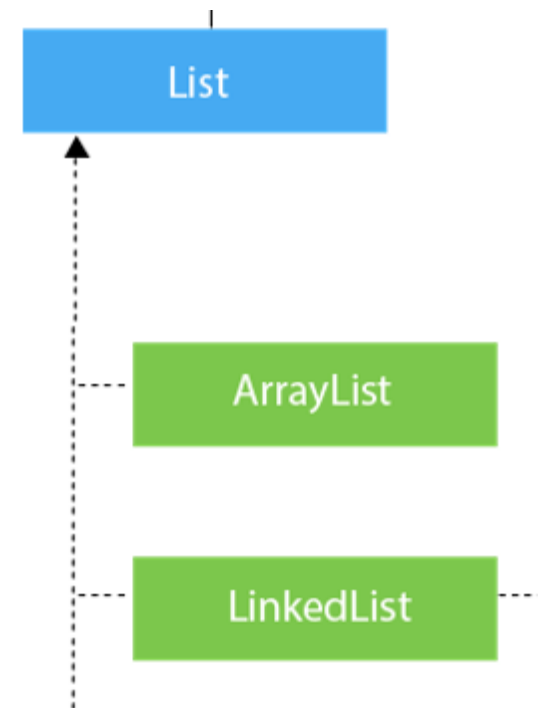
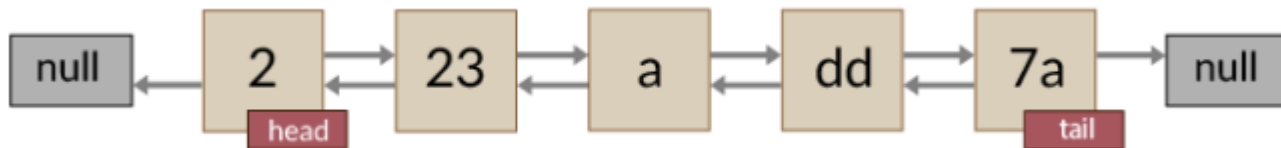
 JAVA		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		Linked List
	Map	HashMap		TreeMap	

# List Implementation

- ArrayList: internally uses an array to store the elements



- LinkedList: internally uses a doubly linked list to store the elements.



# Choosing an Implementation - List

- **ArrayList**: Accessing an element takes constant time ( $O(1)$ ) and adding an element takes  $O(n)$  time in worst case.
- **LinkedList**: Adding an element takes  $O(n)$  time and accessing also takes  $O(n)$  time. LinkedList uses more memory than ArrayList.
- Summary: ArrayList is preferable in many more use-cases than LinkedList. If you're not sure — just start with **ArrayList**.



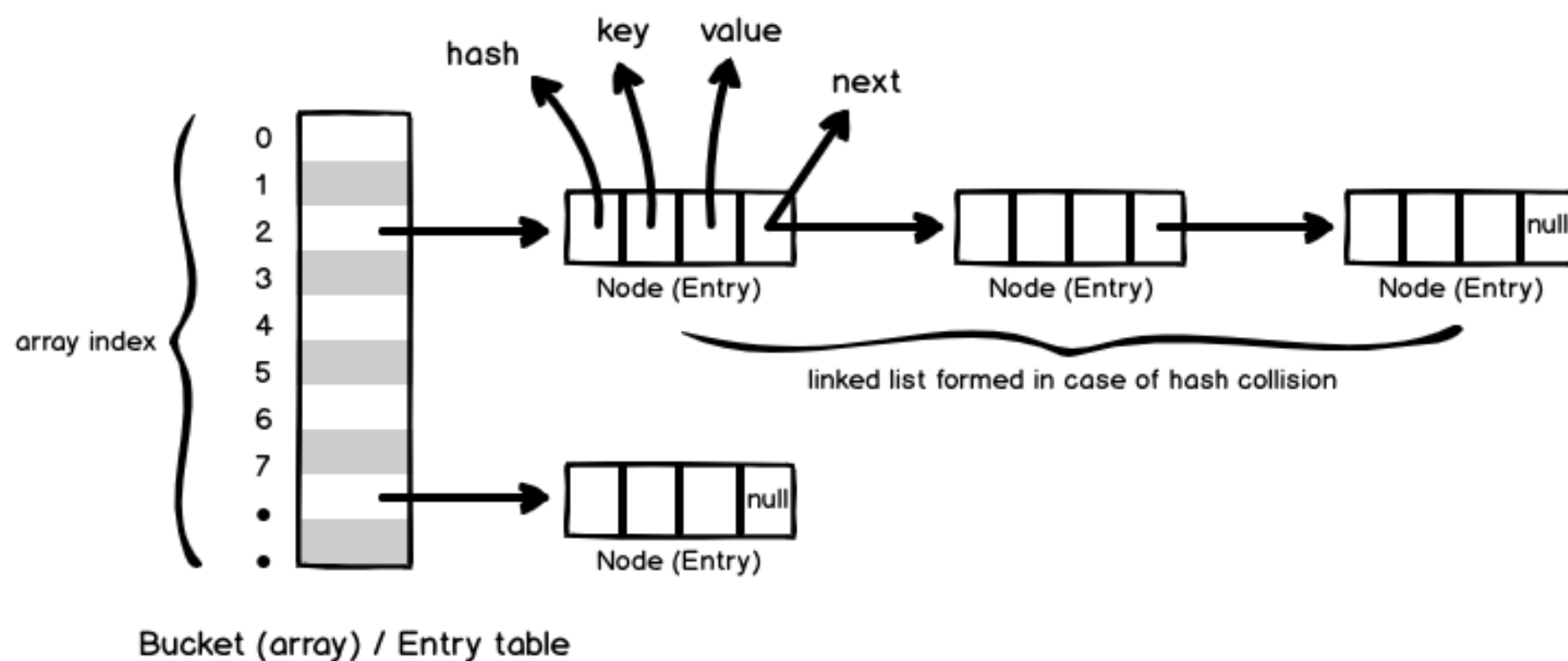
Joshua Bloch ✓  
@joshbloch

回复 @jerrykuch

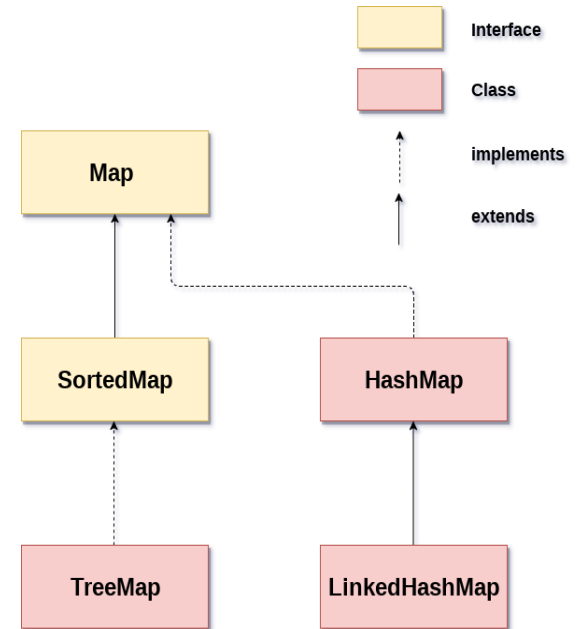
@jerrykuch @shipilev @AmbientLion Does anyone actually use LinkedList? I wrote it, and I never use it.

上午10:10 · 2015年4月3日 · Twitter Web Client

# Map Implementation - HashMap

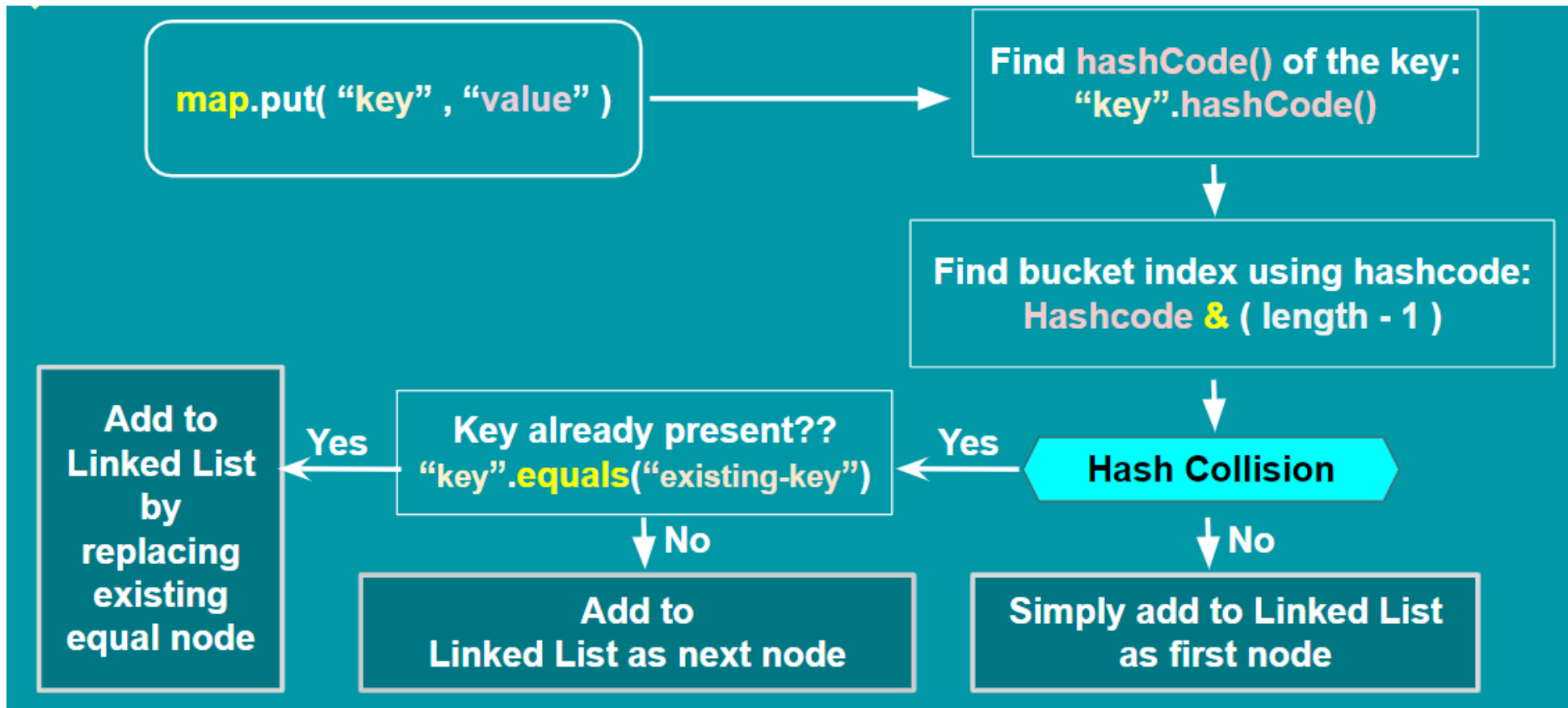


HashMap



Source: <https://www.javaquery.com/2019/11/how-hashmap-works-internally-in-java.html>

# Map Implementation - HashMap



Source: [https://docs.google.com/presentation/d/1jElOUz-FTG3Ea9FqxDQEiyTTCZGj7zhRMCOgYx2g9dM/edit#slide=id.g94208dd8e0\\_0\\_46](https://docs.google.com/presentation/d/1jElOUz-FTG3Ea9FqxDQEiyTTCZGj7zhRMCOgYx2g9dM/edit#slide=id.g94208dd8e0_0_46)

# Map Implementation - HashMap

```
map.put("FB", 1);  
map.put("LD", 2);  
map.put("Ea", 3);  
map.put("FB", 4);
```

```
hashcode of FB = 2236 | index 12  
hashcode of LD = 2424 | index 8  
hashcode of Ea = 2236 | index 12
```

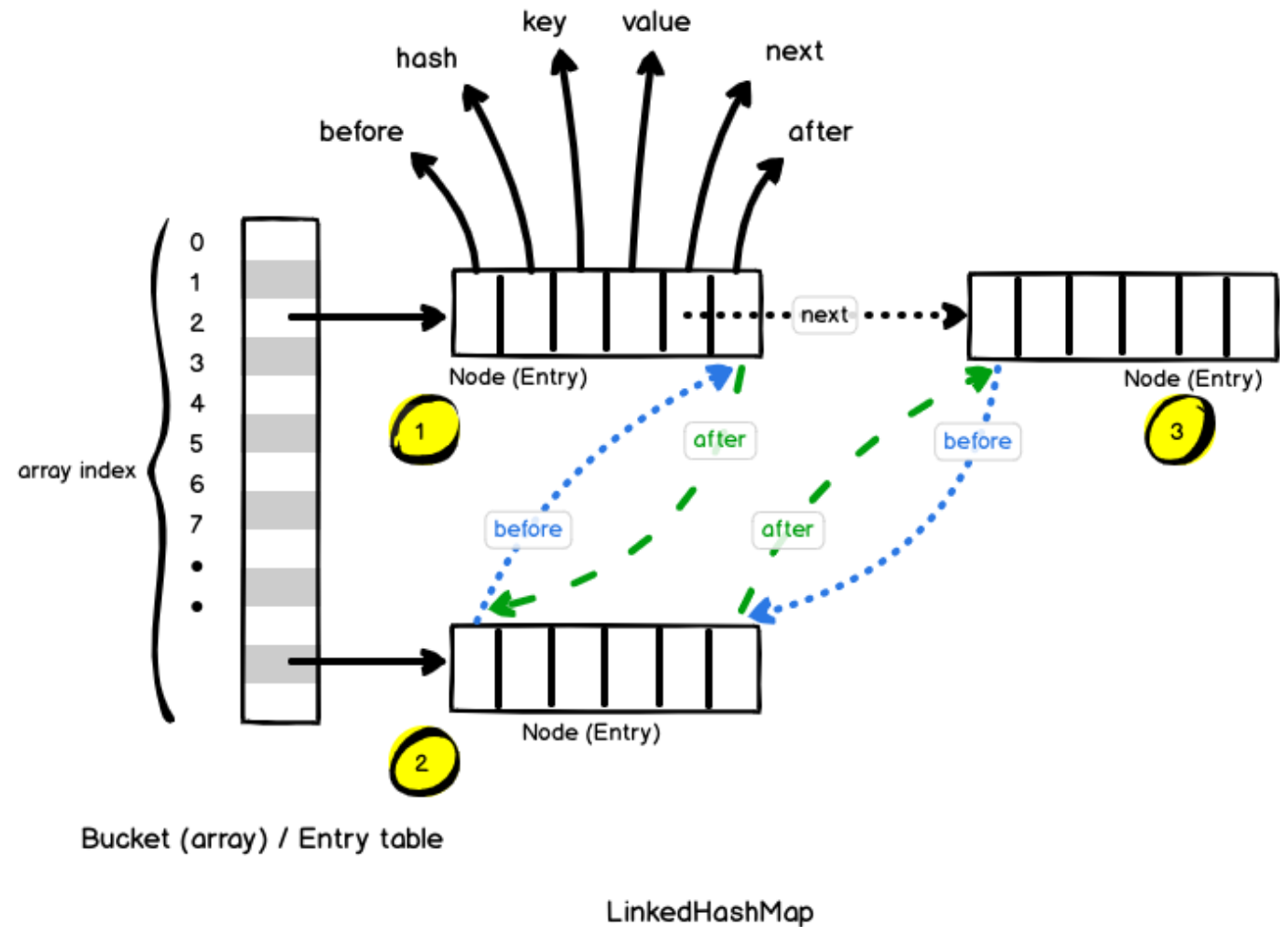
What does the internal HashMap look like?

# Overriding hashCode and equals

- `hashCode()` returns an integer value. By default, it converts the internal address of the object into an integer
- `equals()` checks if objects are equal. By default,  
`Object.equals(Object obj) { return (this == obj); }`
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result (if you override `equals`, you must override `hashCode`).

# Map Implementation – LinkedHashMap

LinkedHashMap uses before and after to preserve the insertion order of the keys



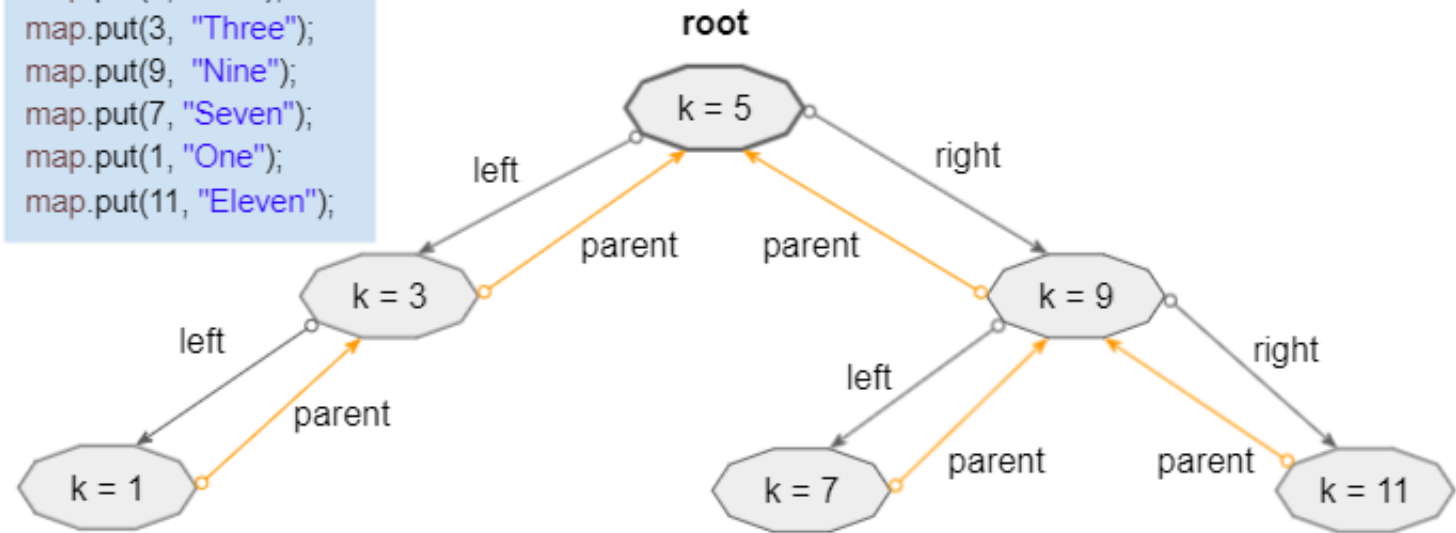
Source: <https://www.javaquery.com/2019/12/how-linkedhashmap-works-internally-in.html>



# Map Implementation – TreeMap

- Use TreeMap when keys **need to be ordered** using their natural ordering or by a Comparator.

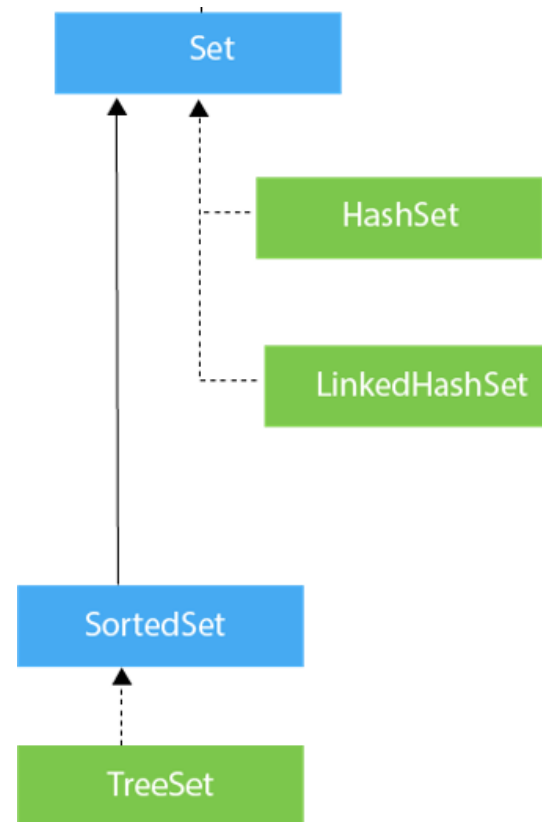
```
map.put(5, "Five");  
map.put(3, "Three");  
map.put(9, "Nine");  
map.put(7, "Seven");  
map.put(1, "One");  
map.put(11, "Eleven");
```



Source: <https://o7planning.org/13597/java-treemap>

# Choosing an Implementation – Set

HashSet	LinkedHashSet	TreeSet
HashSet internally uses HashMap to store its elements.	LinkedHashSet internally uses LinkedHashMap to store its elements.	TreeSet internally uses TreeMap to store its elements.
HashSet doesn't maintain any order of elements.	LinkedHashSet maintain insertion order of elements.	TreeSet maintains default natural sorting order.
HashSet gives better performance than LinkedHashSet and TreeSet.	The performance of LinkedHashSet is between HashSet and TreeSet.	The TreeSet gives less performance than HashSet and LinkedHashSet.
HashSet allow maximum one null element.	LinkedHashSet also allow maximum one null element.	The TreeSet doesn't allow even single element.



# Common Implementation Behaviors

- All implementations permit `null` elements, keys, and values (except for `TreeSet` and `TreeMap`)
- 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

# Core Elements in the Java Collections Framework



Interfaces

Implementations

**Algorithms**

# Reusable Algorithms

Collections class in java represents an utility class in java.util package. It contains exclusively static methods that operate on or return collections

```
java.lang.Object  
java.util.Collections
```

---

```
public class Collections  
extends Object
```

```
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);
```

Useful for reinitializing a list

```
static <E> void copy(List<E> dest, List<? Extends E> src);
```

```
static void reverse(List<?> list);
```

```
static void shuffle(List<?> list);
```

Finding  
extreme  
values

```
public class Shuffle {  
    public static void main(String[] args) {  
        List<String> list = Arrays.asList(args);  
        Collections.shuffle(list);  
        System.out.println(list);  
    }  
}
```

# Why Generic (Reusable) Algorithms?

max() is a common algorithm for collections. But we do not want to write, test, debug this max for different types of collections

The max() algorithm is implemented to take any object that implements the Collection interface

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

# Why Generic (Reusable) Algorithms?

`max()` is a common algorithm for collections. But we do not want to write, test, debug this `max` for different types of collections

The `max()` algorithm is implemented to take any object that implements the `Collection` interface

```
String[] wordlist = {"This", "is", "CS209A", "Java2"};
List<String> slist = Arrays.asList(wordlist);
Set<Integer> iset = new HashSet<>();
iset.add(3);
iset.add(5);
iset.add(2);
Queue<String> squeue = new LinkedList<>();
squeue.add("Hello");
squeue.add("World");
squeue.add("Java2");

System.out.format("List max: %s\n", Collections.max(slist));
System.out.format("Set max: %s\n", Collections.max(iset));
System.out.format("Queue max: %s\n", Collections.max(squeue));
```

# Sorting Algorithm

- `sort()` reorders a collection 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?



# Sorting Algorithm

- 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	<code>Boolean.FALSE &lt; Boolean.TRUE</code>
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

# Sorting Algorithm

```
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;
        }
    }
}
```

# Sorting Algorithm

```
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 ]]
```

# Convenience Operation 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 Operation II

- `java.util.Collections` is a class consists exclusively of static methods that operate on or return collections
- `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 Operation 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 Operation 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());
```



# 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