Collections

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

Dr. Siobhán Drohan (sdrohan@wit.ie)

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Overview: Road Map

- The Collection Framework
- Interfaces
  - Collection
  - Iterator
- Interfaces and common implementations
  - Set
  - List
  - Map
- Generic Collections (Java 5)
  - Untyped vs Typed syntax
  - Defining collections for maintenance
  - For-each loop

What are Collections?

- Collections are Java objects that group multiple elements into a single unit.
  - Represent data items that form a natural group e.g. users, locations, activities.
- Collections store, retrieve, and manipulate other Java objects
  - Any Java object may be part of a collection, so collection can contain other collections.
- Collections do not store primitives.

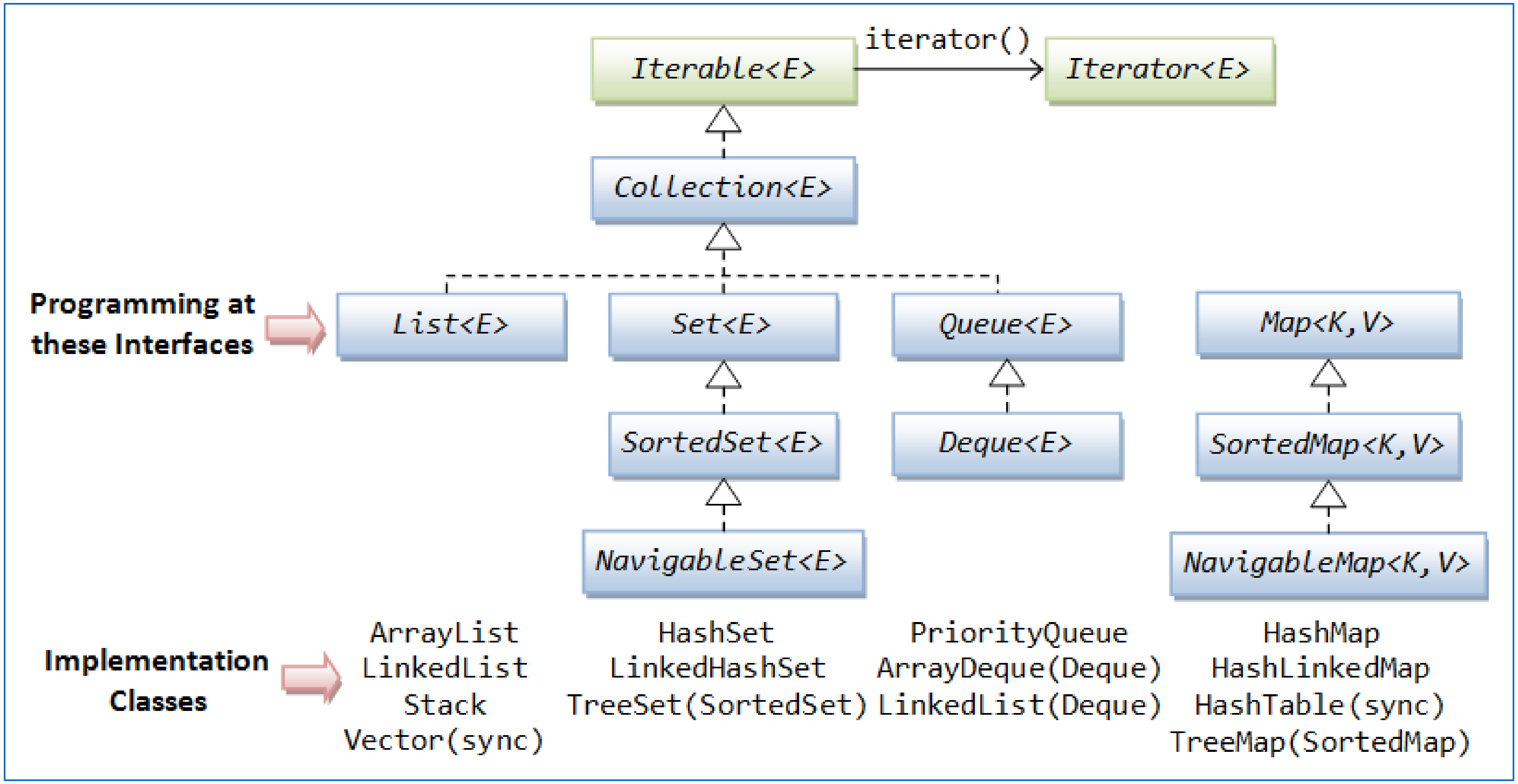
Based on: http://docs.oracle.com/javase/tutorial/collections/intro/index.html

Collection Architecture

- Java Collection architecture includes:
  - **Interfaces** - abstract data types representing collections
  - **Implementation** - concrete implementation of collection interfaces
  - **Algorithms** - methods for manipulating collection objects (e.g. sorting, searching, shuffling, etc).

Based on: http://docs.oracle.com/javase/tutorial/collections/intro/index.html

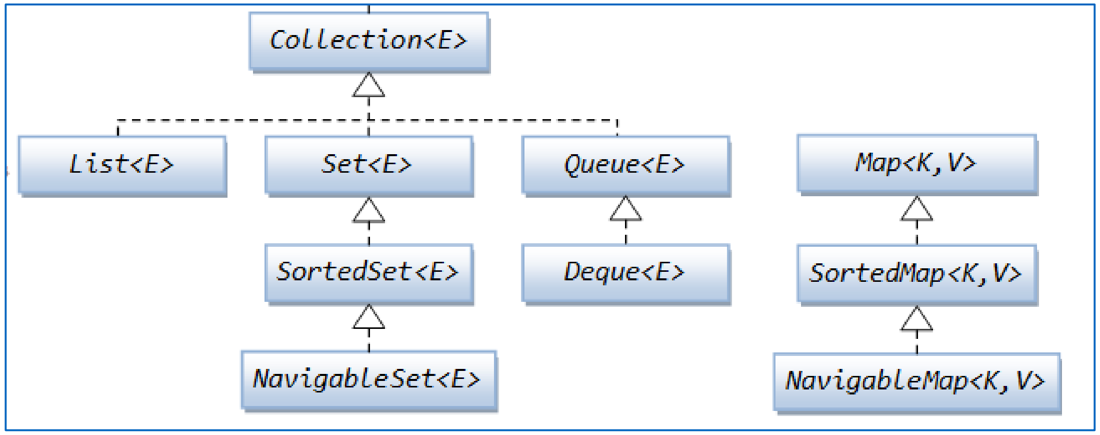
Collections Framework



Generic classes

- Collections are *parameterized* or *generic* types.

- The type parameter says what we want e.g.:
  - **`List<Account>`**
  - **`Map<String, Person>`**
  - etc.



Collection Architecture - Benefits

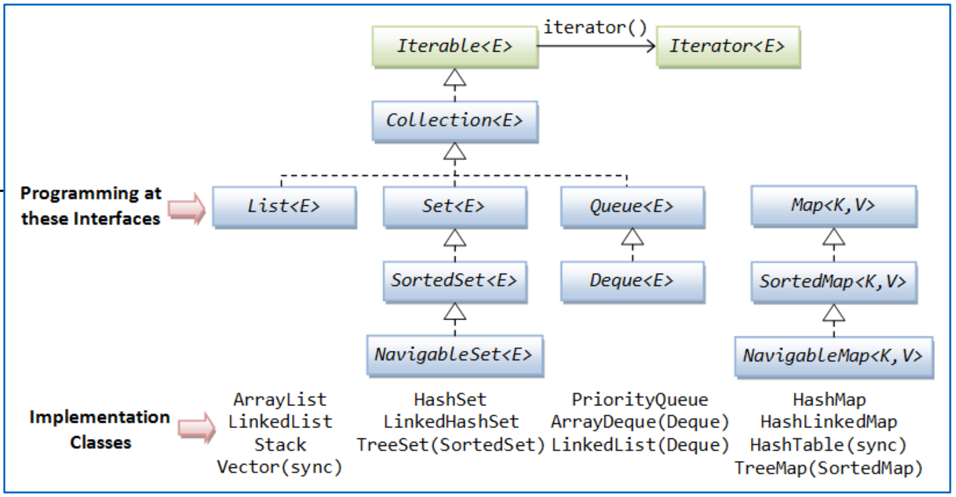
Reusability	Promotes software reuse
Uniformity	Similar, consistent design adopted across the collections.
Faster development	As programmers don’t have to write their own data structures, they have more time to concentrate on writing programs.
Higher quality	High performance and quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations.
Interoperability	Collections generated by other (unrelated) APIs can be taken as input to or sent as output from our Java programs.
Less programming	Programmers don’t have to write classes to manage data structures.

Based on: http://docs.oracle.com/javase/tutorial/collections/intro/index.html

Overview: Road Map

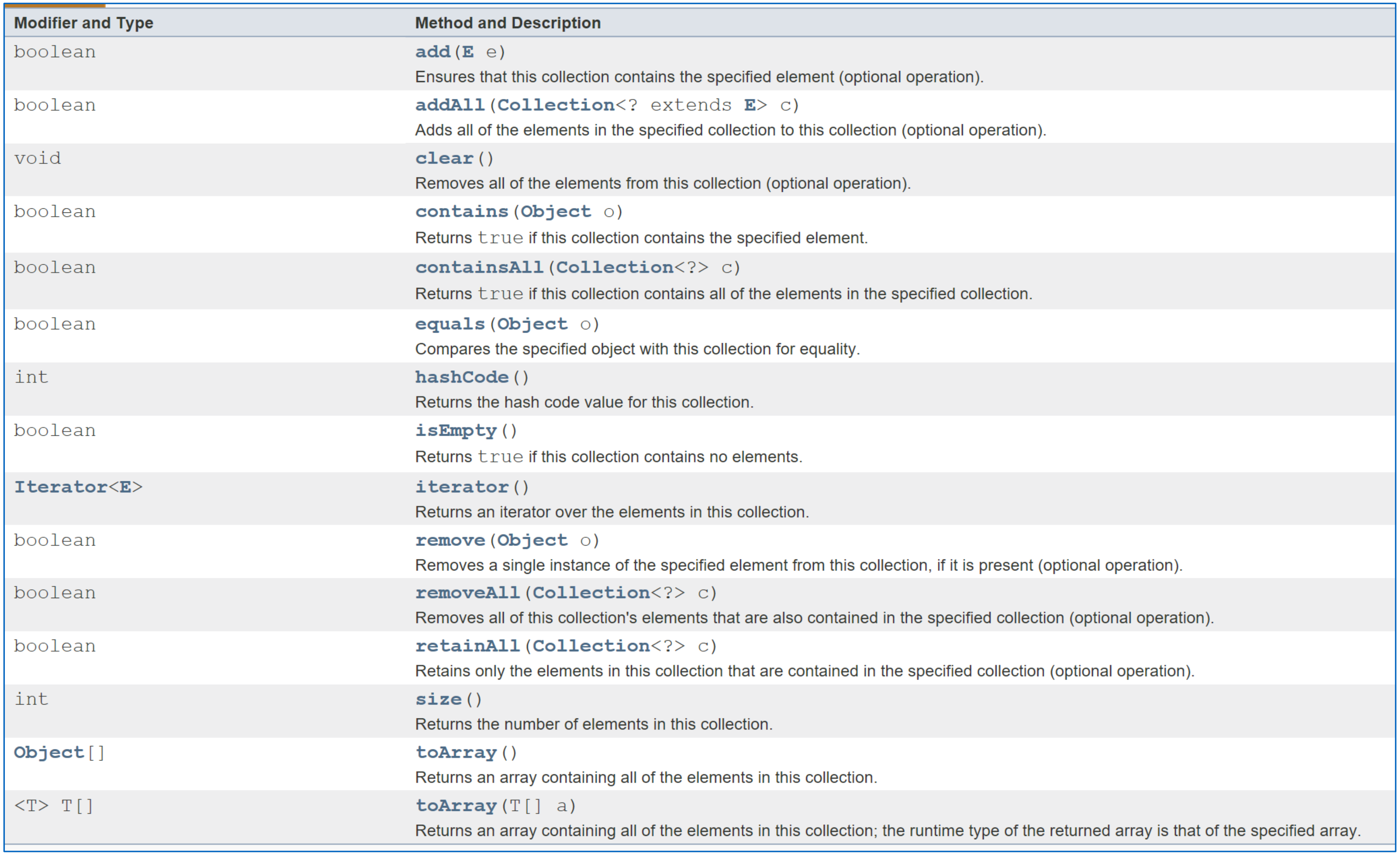
- The Collection Framework
- Interfaces
  - Collection
  - Iterator
- Interfaces and common implementations
  - Set
  - List
  - Map
- Generic Collections (Java 5)
  - Untyped vs Typed syntax
  - Defining collections for maintenance
  - For-each loop

Collection Interface



- Collection represents a group of objects and is the root of the collection hierarchy
  - These collection objects are known as collection elements
- There is no direct implementation of this interface in JDK
  - Concrete implementations are provided for its subtypes
- You choose the implementation of a collection based on your requirements e.g.
  - Are you storing duplicate elements?
  - Do you need a sorted collection?
  - Are you maintaining a First-In-First-Out order?

Collection Interface



Modifier and Type	Method and Description
`boolean`	`add(E e)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()` Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this collection contains all of the elements in the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this collection for equality.
`int`	`hashCode()` Returns the hash code value for this collection.
`boolean`	`isEmpty()` Returns `true` if this collection contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this collection.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)` Removes all of this collection's elements that are also contained in the specified collection (optional operation).
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this collection that are contained in the specified collection (optional operation).
`int`	`size()` Returns the number of elements in this collection.
`Object[]`	`toArray()` Returns an array containing all of the elements in this collection.
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Adding Elements

- In general two methods are defined for adding elements to the collection:

```
interface Collection
{
  //…
  /**
   * Adds element to the receiver.
   * Returns true if operation is successful, otherwise return s false.
   */
  boolean add(Object element);

  /**
   * Adds each element from collection c to the receiver.
   * Returns true if operation is successful, otherwise returns false.
   */
  boolean addAll(Collection c);
}
```

Removing Elements

- Similarly to adding protocol, there are two methods are defined for removing elements from the collection:

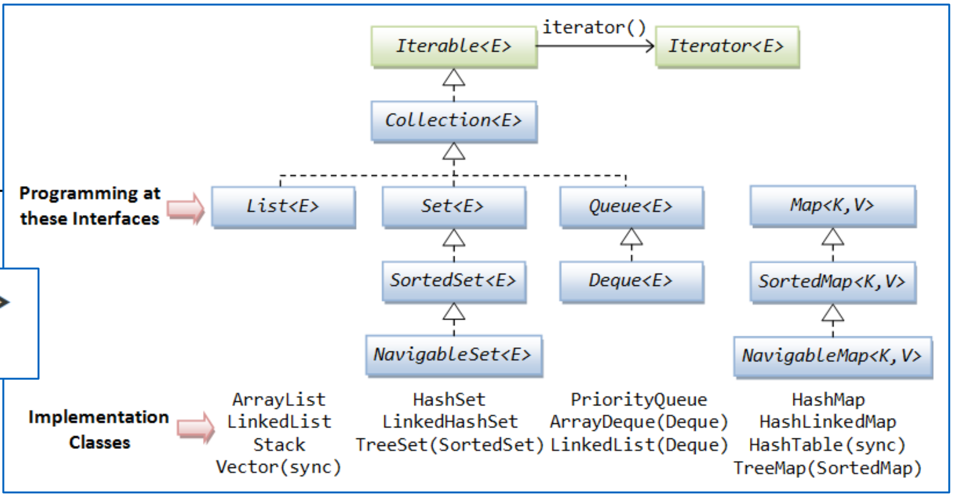
```
interface Collection
{
  //…
  /**
   * Removes element from the receiver.
   * Returns true if operation is successful, otherwise returns false.
   */
  boolean remove(Object element);

  /**
   * Removes each element contained in collection c from the receiver.
   * Returns true if operation is successful, otherwise returns false.
   */
  boolean removeAll(Collection c);
}
```

Other Collection Methods

- Includes methods for:
  - Checking how many elements are in the collection
  - Checking if an element is in the collection
  - Iterating through collection

```
boolean contains(Object element);
boolean containsAll(Collection c);
int size();
boolean isEmpty();
void clear();
boolean retainAll(Collection c);
Iterator<E> iterator();
```

Iterator Interface



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

```
public interface Iterator
{
  /**
   * Returns whether or not the underlying collection has next
   * element for iterating.
   */
  boolean hasNext();

  /**
   * Returns next element from the underlying collection.
   */
  Object next();

  /**
   * Removes from the underlying collection the last element returned by next.
   */
  void remove();
}
```

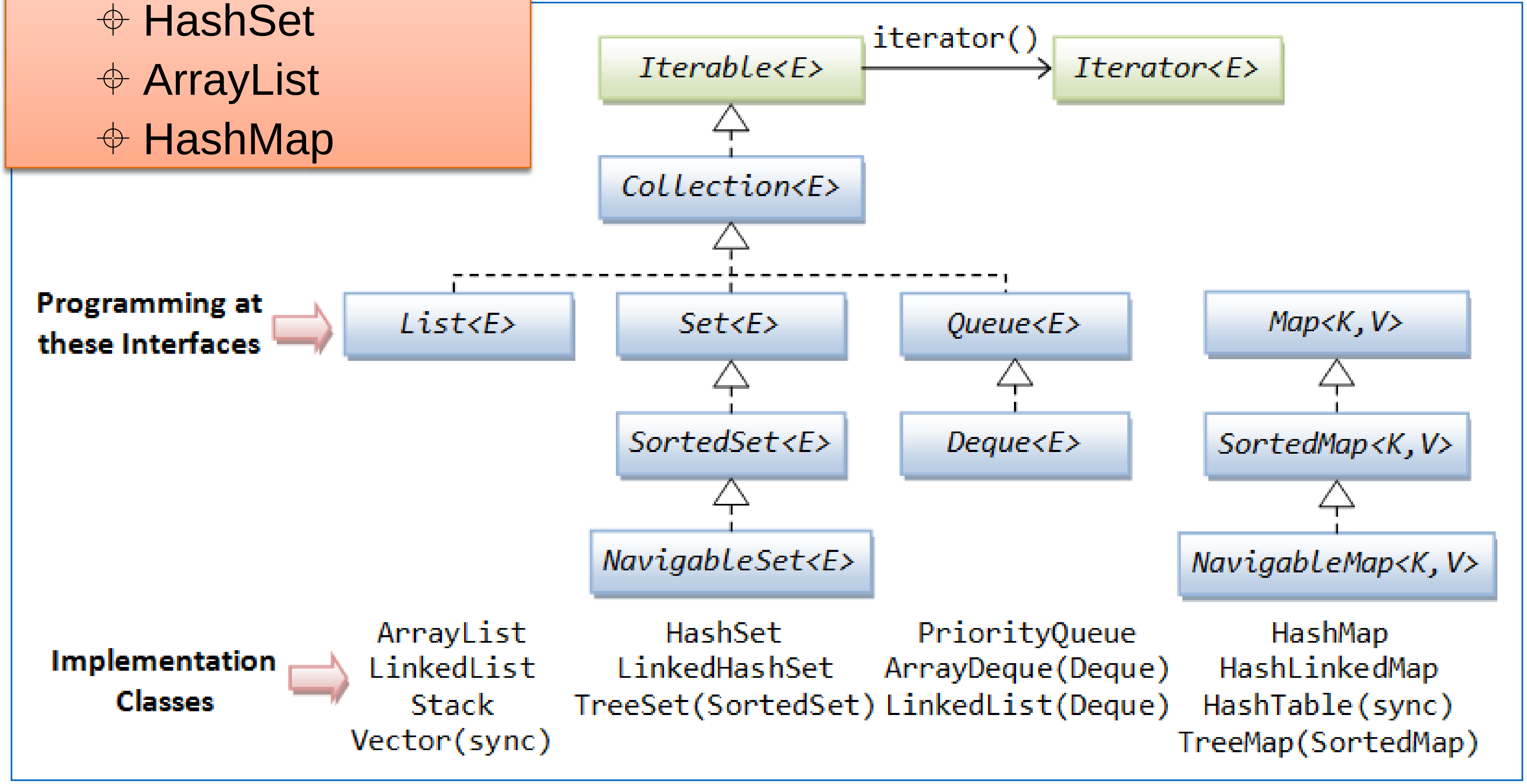
Defines a protocol for iterating through a collection.

Overview: Road Map

- The Collection Framework
- Interfaces
  - Collection
  - Iterator
- Interfaces and common implementations
  - Set
  - List
  - Map
- Generic Collections (Java 5)
  - Untyped vs Typed syntax
  - Defining collections for maintenance
  - For-each loop

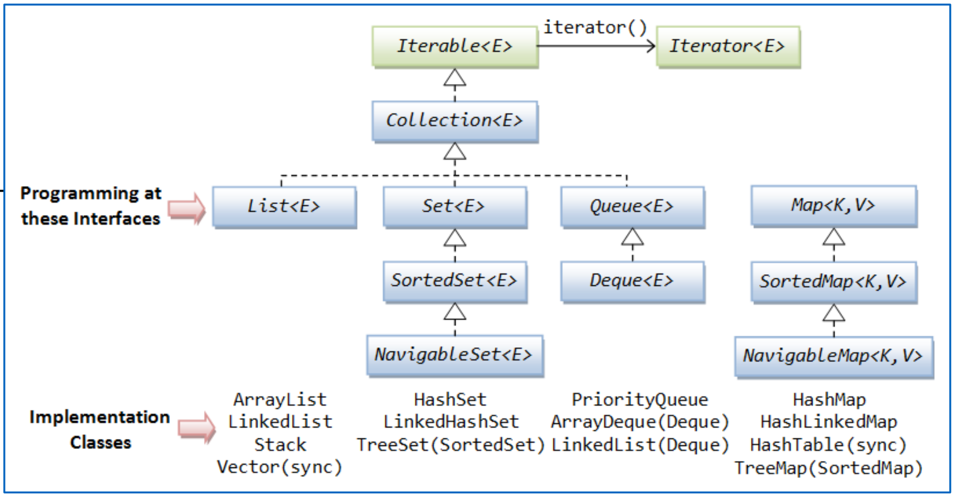
Most Commonly Used Collections

- Three of the most commonly used collections:
  - HashSet
  - ArrayList
  - HashMap



Set and HashSet

Set Interface



- Set is a collection contains no duplicate elements
  - All constructors in a set must create a set that does not contain duplicate elements.
  - Contains at most one null element.

- The set does not maintain its elements in any particular order.

HashSet

- Concrete implementation of the Set interface
  - Cannot contain duplicates; the add is ignored if a duplicate element is already stored.
  - is the best-performing implementation
  - makes no guarantees concerning the order of iteration and order of elements cannot be guaranteed over time.

- Performance of the set is affected by size of the set and capacity of the map
  - It is important not to set the initial capacity too high, or the load factor too low if performance of iteration is important

Note: The load factor is a measure of how full the set is allowed to get before its capacity is automatically increased.

HashSet Example

```java
//create new set
Set<String> set = new HashSet<>();

//add elements to the set
set.add("One");
set.add("Two");
set.add("Three");

//elements cannot be duplicated in the set
set.add("One");

//print the set
System.out.println(set);
```

Console

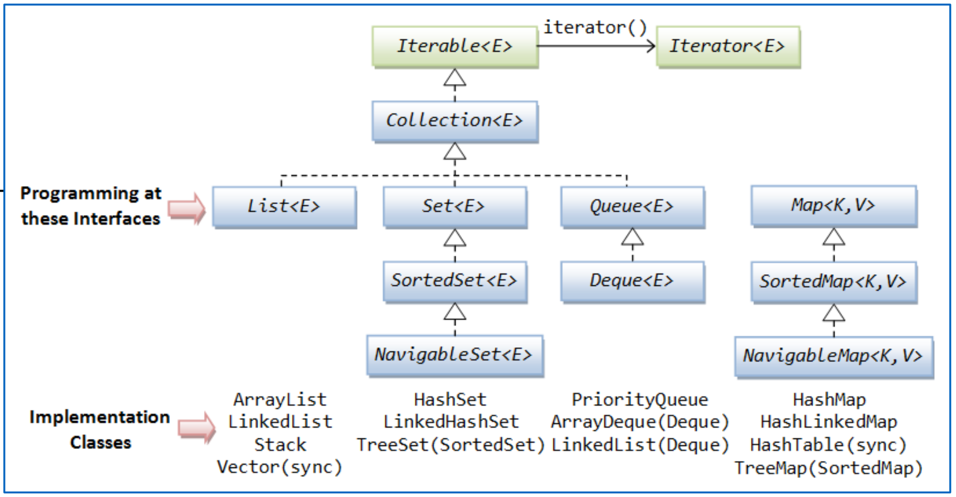
```
[One, Three, Two]
```

Collection Summary (so far)

Class	Map	Set	List	Ordered	Sorted	Allow Duplicates
HashSet		X		No	No	No
TreeSet		X		Sorted	By natural order or custom comparison rules	No
LinkedHashSet		X		By insertion order	No	No

List and ArrayList

List Interface



- List represents an ordered collection
  - Precise control over where in the list each element is inserted
  - Zero based indexed access
- Lists may contain duplicate elements
- Lists extend behavior of collections with operations for:
  - Positional Access
  - Search
  - List Iteration
  - Range-view (the *sublist* method performs arbitrary *range operations* on the list)

ArrayList

- Represents resizable-array implementation of the List interface
  - Permits all elements including null
  - Zero-based indexing
  - Permits duplicates
  - Unsorted
- It is generally the best performing List interface implementation
- Instances of this class have a capacity
  - It is size of the array used to store the elements in the list, and it’s always at least as large as the list size
  - It grows as elements are added to the list

ArrayList Examples

```
//declare list
List<String> list = new ArrayList<>();

//add elements to the list
list.add("First element");
list.add("Second element");

//get the list size
int listSize = list.size();

//print the list size and the first element
System.out.println(listSize);
System.out.println(list.get(0));

//add first element in the list
list.add(0,"Added element");

//get the list iterator
Iterator iterator = list.iterator();
while (iterator.hasNext())
{
   String element = (String)iterator.next();
   System.out.println(element);
}
```

Console

```
2
First element
```

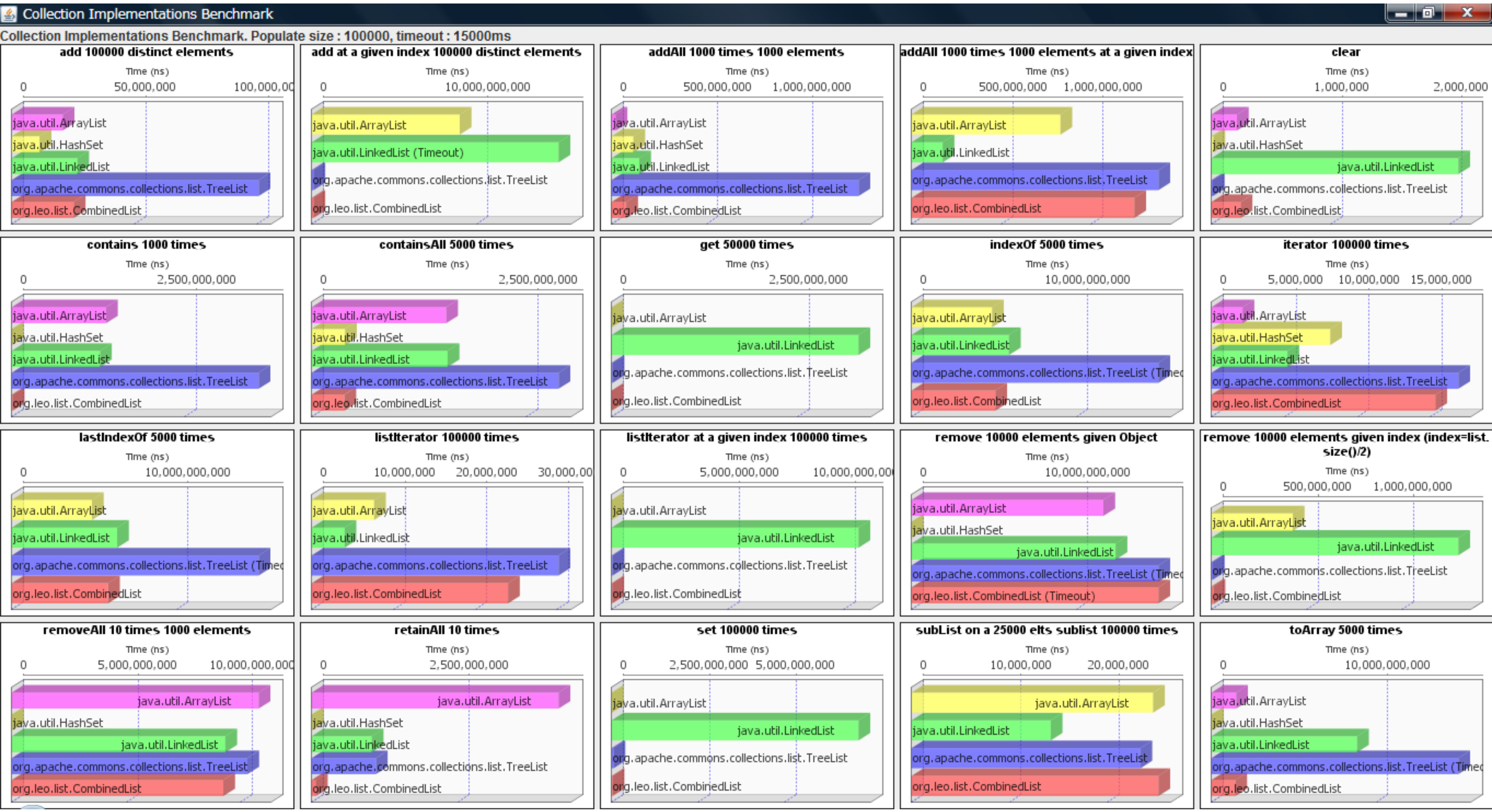
Console

```
Added element
First element
Second element
```

Collection Summary (so far)

Class	Map	Set	List	Ordered	Sorted	Allow Duplicates
HashSet		X		No	No	No
TreeSet		X		Sorted	By natural order or custom comparison rules	No
LinkedHashSet		X		By insertion order	No	No
ArrayList			X	By index	No	Yes
Vector			X	By index	No	Yes
LinkedList			X	By index	No	Yes

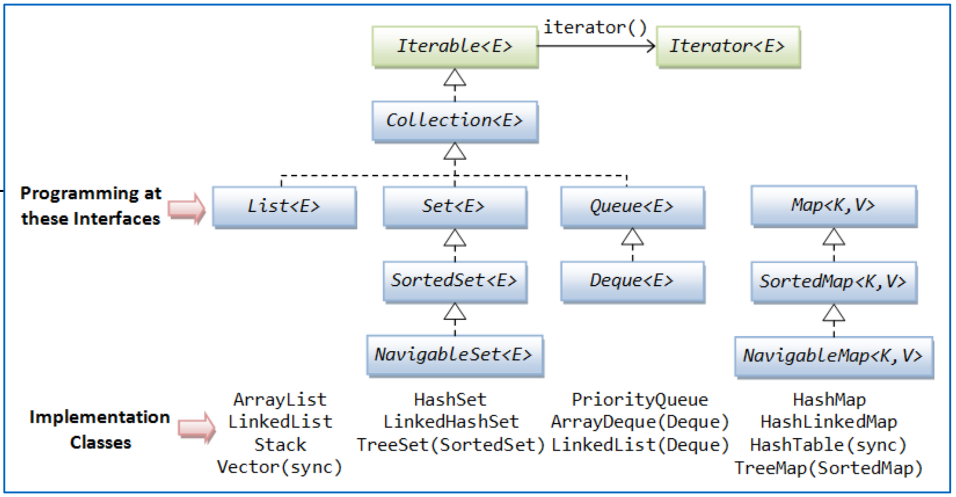
Java Collection Performance


Collection Implementations Benchmark
Collection Implementations Benchmark. Populate size : 100000, timeout : 15000ms
add 100000 distinct elements
add at a given index 100000 distinct elements
addAll 1000 times 1000 elements
addAll 1000 times 1000 elements at a given index
clear
contains 1000 times
containsAll 5000 times
get 50000 times
indexOf 5000 times
iterator 100000 times
lastIndexOf 5000 times
listIterator 100000 times
listIterator at a given index 100000 times
remove 10000 elements given Object
remove 10000 elements given index (index=list.size()/2)
removeAll 10 times 1000 elements
retainAll 10 times
set 100000 times
subList on a 25000 elts sublist 100000 times
toArray 5000 times
Time (ns)
java.util.ArrayList
java.util.HashSet
java.util.LinkedList
java.util.LinkedList (Timeout)
org.apache.commons.collections.list.TreeList
org.apache.commons.collections.list.TreeList (Timed
org.leo.list.CombinedList
org.leo.list.CombinedList (Timeout)

https://dzone.com/articles/java-collection-performance

Map and HashMap

Map Interface



- Map is an object that maps keys to values
  - Keys must be unique, i.e. map cannot contain duplicate keys
  - Each key in the map can map to most one value, i.e. one key cannot have multiple values

HashMap

- Collection that contains pair of objects
  - A value is stored at a key
  - No duplicate keys allowed; when adding an identical key, the existing value is replaced with the new value.
  - Permits null values and a null key
  - The order of the map is not guaranteed
- Two parameters affect performance of a hash map:
  - Initial capacity, indicates capacity at the map creation time
  - Load factor, indicates how full the map should be before increasing its size
    - 0.75 is the default

HashMap Example

```
//create a number dictionary
Map<String, String> numberDictionary = new HashMap<>();
numberDictionary.put("1", "One");
numberDictionary.put("2", "Two");
numberDictionary.put("3", "Three");
numberDictionary.put("4", "Four");
numberDictionary.put("5", "Five");

//get an iterator of all the keys
Iterator keys = numberDictionary.keySet().iterator();
while (keys.hasNext())
{
  String key = (String)keys.next();
  String value = (String)numberDictionary.get(key);
  System.out.println("Number: " + key + ", word: " + value);
}
```

Console

```
Number: 1, word: One
Number: 2, word: Two
Number: 3, word: Three
Number: 4, word: Four
Number: 5, word: Five
```

Collection Summary

Class	Map	Set	List	Ordered	Sorted	Allow Duplicates
HashSet		X		No	No	No
TreeSet		X		Sorted	By natural order or custom comparison rules	No
LinkedHashSet		X		By insertion order	No	No
ArrayList			X	By index	No	Yes
Vector			X	By index	No	Yes
LinkedList			X	By index	No	Yes
HashMap	X			No	No	No duplicate key allowed
Hashtable	X			No	No	No duplicate key allowed
TreeMap	X			Sorted	By natural order or custom comparison rules	No duplicate key allowed
LinkedHashMap	X			By insertion order or last access order	No	No duplicate key allowed

Overview: Road Map

- The Collection Framework
- Interfaces
  - Collection
  - Iterator
- Interfaces and common implementations
  - Set
  - List
  - Map
- Generic Collections (Java 5)
  - Untyped vs Typed syntax
  - Defining collections for maintenance
  - For-each loop

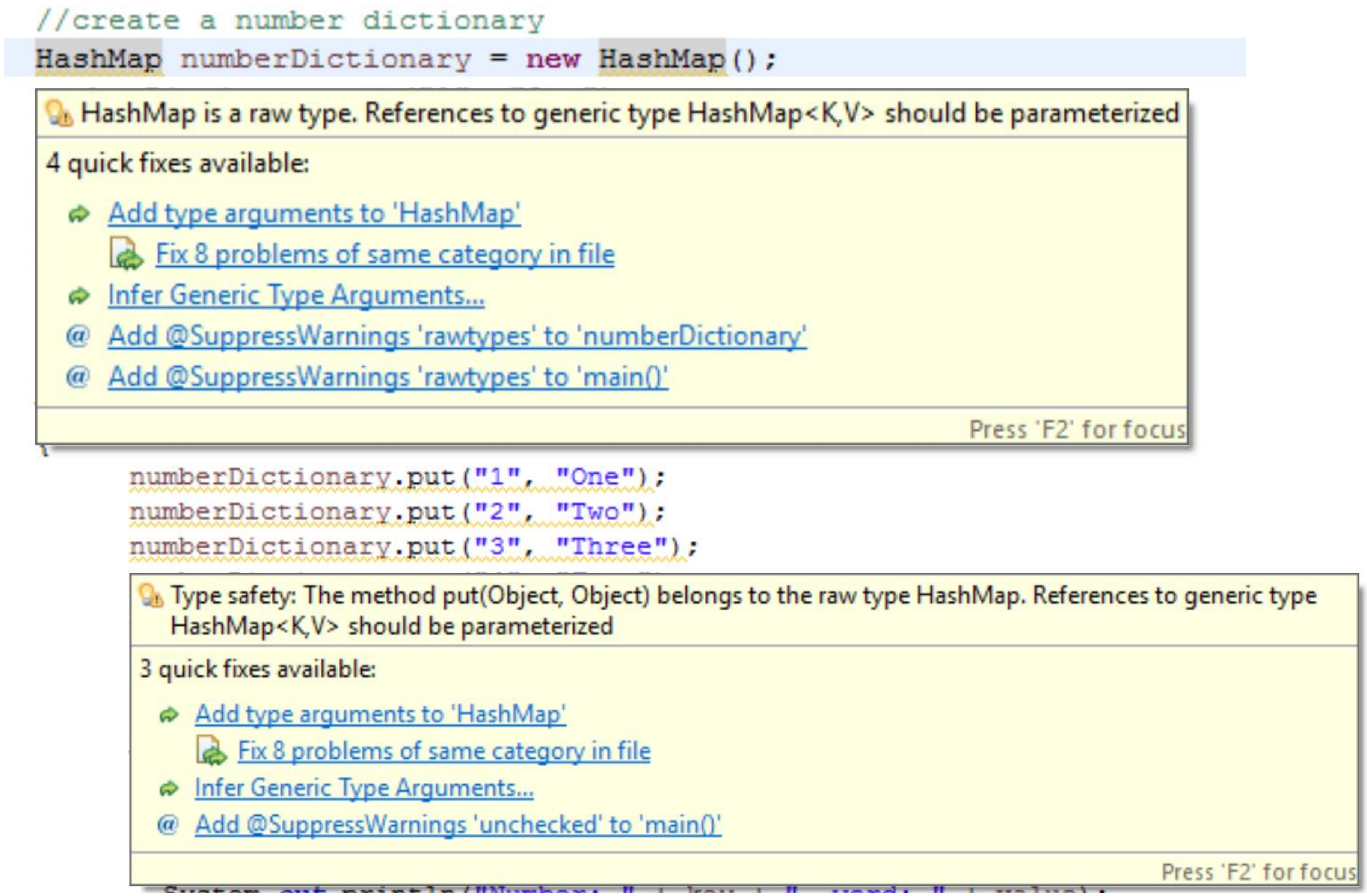
Generic Collection (Java 5)

- Collections use polymorphism to store objects of any type.
- A drawback is type loss on retrieval.
  - HashMap stores key/value pairs as java Objects.
  - get() method returns a matching Object for the given key.

```
HashMap numberDictionary = new HashMap();

numberDictionary.put("1", "One");
numberDictionary.put("2", "Two");

Object value = numberDictionary.get("1”);
String strValue = (String) value;
```

- The key/values in this code are actually Strings
- The return value must be type cast back to a String in order to accurately recover the stored object.

Compiler warnings for untyped collections



```
//create a number dictionary
HashMap numberDictionary = new HashMap();
```

HashMap is a raw type. References to generic type HashMap<K,V> should be parameterized

4 quick fixes available:

- Add type arguments to 'HashMap'
  - Fix 8 problems of same category in file
- Infer Generic Type Arguments...
- Add @SuppressWarnings 'rawtypes' to 'numberDictionary'
- Add @SuppressWarnings 'rawtypes' to 'main()'

Press 'F2' for focus

```
    numberDictionary.put("1", "One");
    numberDictionary.put("2", "Two");
    numberDictionary.put("3", "Three");
```

Type safety: The method put(Object, Object) belongs to the raw type HashMap. References to generic type HashMap<K,V> should be parameterized

3 quick fixes available:

- Add type arguments to 'HashMap'
  - Fix 8 problems of same category in file
- Infer Generic Type Arguments...
- Add @SuppressWarnings 'unchecked' to 'main()'

Press 'F2' for focus

Untyped = Unsafe

- Type casting is undesirable (due to possibility of run time errors).

- Therefore, use of untyped (pre-Java 5) collections is considered ‘unsafe’.

- Typed collections avoid type loss.

- Runtime checks are simplified because the type is known.

Typed HashMaps

- HashMaps operate with (key,value) pairs.

- A typed HashMap requires two type parameters:

```
private HashMap<String, String> responses;
...
responses = new HashMap<String, String> ();
```

HashMaps

```
HashMap numberDictionary = new HashMap();

numberDictionary.put("1", "One");
numberDictionary.put("2", "Two");

Object value = numberDictionary.get(“1”);
String strValue = (String) value;
```

untyped / unsafe

```
HashMap<String,String> numberDictionary =
 new HashMap<String,String>();

numberDictionary.put("1", "One");
numberDictionary.put("2", "Two");

String value = numberDictionary.get(“1”);
```

typed / safe

Using a typed collection

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

list.add("First element");
list.add("Second element");

String first = (String)list.get(0);
String second = (String)list.get(1);
```

untyped / unsafe

```java
ArrayList<String> list = new ArrayList<String>();

list.add("First element");
list.add("Second element");

String first = list.get(0);
String second = list.get(1);
```

typed / safe

Using a Typed Iteration

```
ArrayList list = new ArrayList();

Iterator iterator = list.iterator();
while (iterator.hasNext()
{
   String element = (String)iterator.next();
   System.out.println(element);
}
```

untyped / unsafe

```
ArrayList<String> list = new ArrayList<String>();

Iterator<String> iterator = list.iterator();
while (iterator.hasNext())
{
   String element = iterator.next();
   System.out.println(element);
}
```

typed / safe

Defining Collections

Your code is more **maintainable** when you define collections like this:

```
List<Product> products = new ArrayList<Product>();
Map<String, String> addresses = new HashMap<String, String>();
Set<String> words = new HashSet<String>();
```

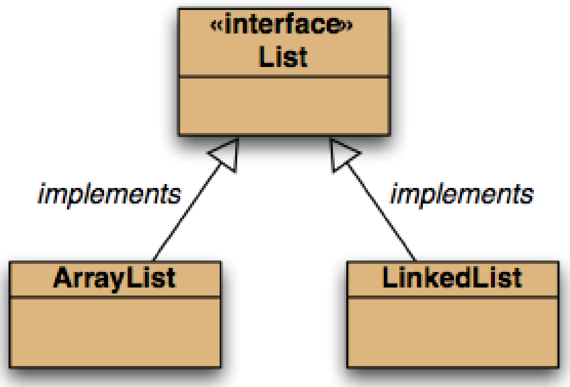
Why: if we want to use a LinkedList instead of an ArrayList, we only have to make minor changes in the class i.e.

```
new ArrayList<Product>();
```

becomes

```
new LinkedList<Product>();
```

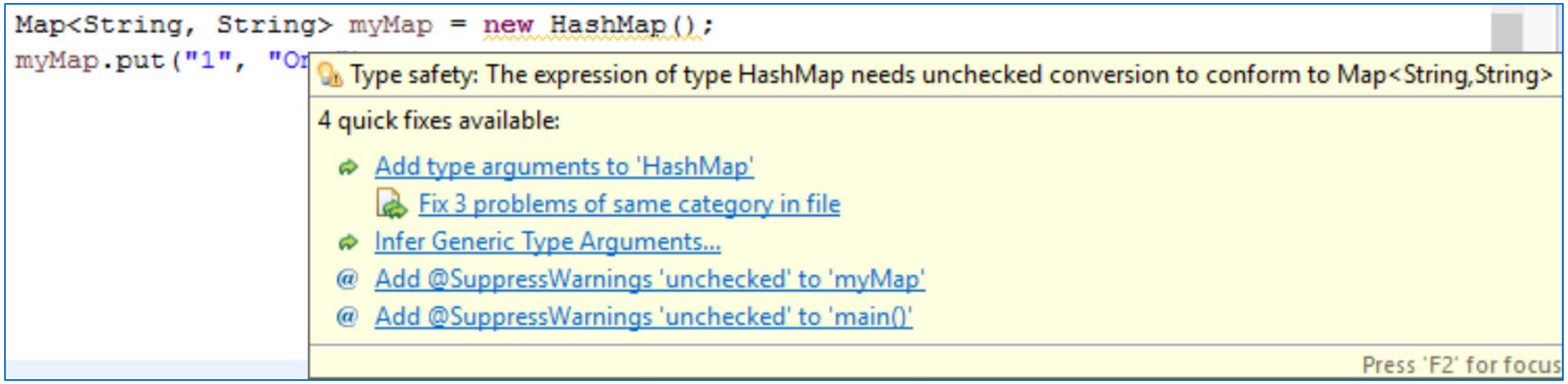
and `import java.util.LinkedList;`



Type Inference

- Since Java 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context:
  - Map<String, String> myMap = new HashMap<>();

- However, you must specify the diamond operator (<>):



```
Map<String, String> myMap = new HashMap();
myMap.put("1", "On
```

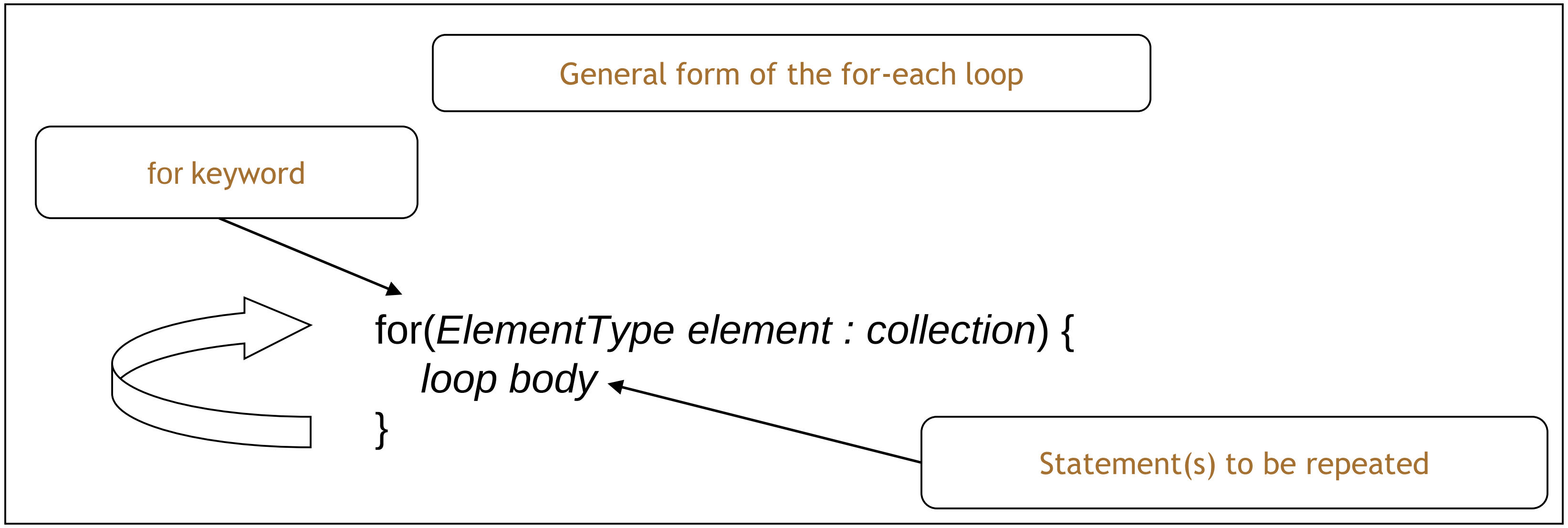
Type safety: The expression of type HashMap needs unchecked conversion to conform to Map<String,String>

4 quick fixes available:

- Add type arguments to 'HashMap'
  - Fix 3 problems of same category in file
- Infer Generic Type Arguments...
- Add @SuppressWarnings 'unchecked' to 'myMap'
- Add @SuppressWarnings 'unchecked' to 'main()'

Press 'F2' for focus

for-each loop pseudo code


General form of the for-each loop
for keyword
Statement(s) to be repeated

```
for(ElementType element : collection) {
    loop body
}
```

Pseudo-code expression of the actions of a for-each loop

For each *element* in *collection*, do the things in the *loop body*.

for-each Loop

- Iteration over collections is a common operation.
- If a collections provides an Iterator, the enhanced for loop simplifies code.

```
List<String> list = new ArrayList<String>();
//…
Iterator <String> iterator = list.iterator();
while (iterator.hasNext())
{
  String element = iterator.next();
  System.out.println(element);
}
```

Standard while loop

```
List<String> list = new ArrayList<String>();
//…
for (String element : list)
{
  System.out.println(element);
}
```

For-each loop

for-each Loop

- The for-each loop is used solely for iteration.
- It cannot be used for any other kind of operation, such as removing or editing an item in the collection or array.
- To safely remove from a collection while iterating over it you should use an Iterator.

Summary of Collection Interfaces

The Java Collections Framework hierarchy consists of two distinct interface trees:

- The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. We looked at the subinterfaces, Set and List.
  - Set: does not allow duplicate elements. Useful for storing collections such as a deck of cards or student records.
  - List: provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- The second tree starts with the Map interface, which maps keys and values.

Summary outline of some collections

ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows efficient insertions and removal at any location
ArrayDeque	A double-ended queue that is implemented as a circular array
HashSet	An unordered collection that rejects duplicates
TreeSet	A sorted set
LinkedHashSet	A set that remembers the order in which elements were inserted
PriorityQueue	A collection that allows efficient removal of the smallest element
HashMap	A data structure that stores key/value associations

https://dzone.com/refcardz/core-java

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit