Using Collections

An introduction to the Java Programming Language

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

Dr. Siobhan Drohan (sdrohan@wit.ie)

Waterford Institute *of* Technology

INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Agenda

- Generic Collections
- Reviewing the Collection Interface
- Summary of Features & Performance
- Working with Collections

Generic Collections

- 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;
```

Generic Collections

- 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.

```java
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.

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.

Revised syntax

- The type of object to be stored is indicated on declaration:

```
private ArrayList<String> notes;
```

- ... and on creation:

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

- Collection types are parameterized.

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

Typed HashMaps

- HashMaps operate with (key,value) pairs.
- A typed HashMap required 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;
```

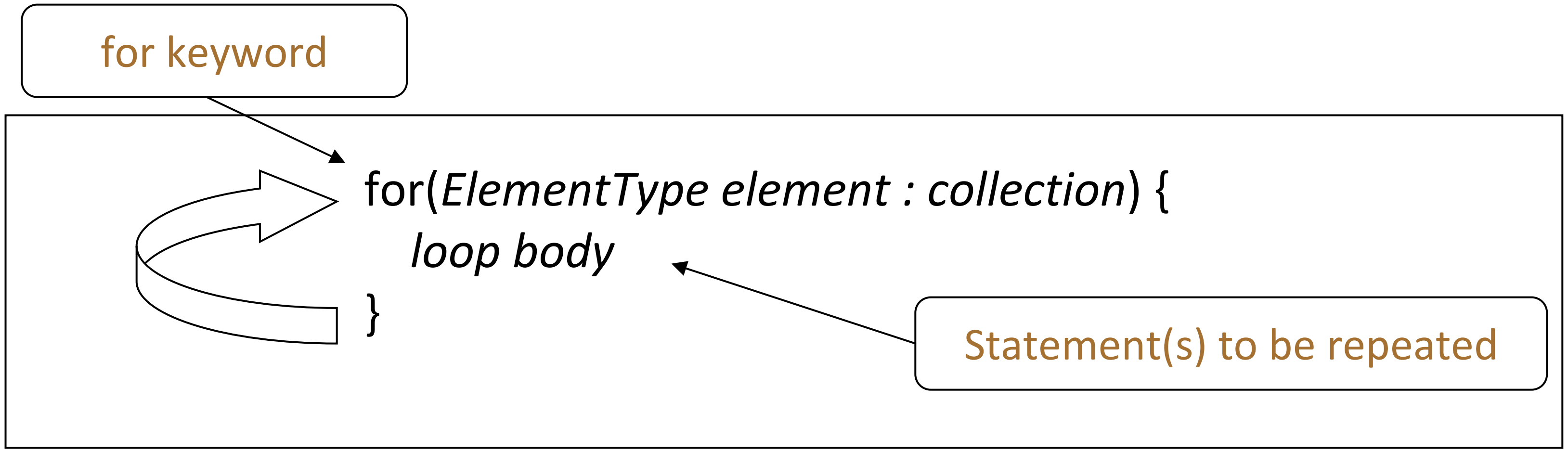
untyped / unsafe

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

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

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

typed / safe

for-each loop (pseudo code)

General form of the for-each loop



```
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*.

If a collections provides an Iterator, the enhanced for loop simplifies code.

For-each Loop

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

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

Standard while loop

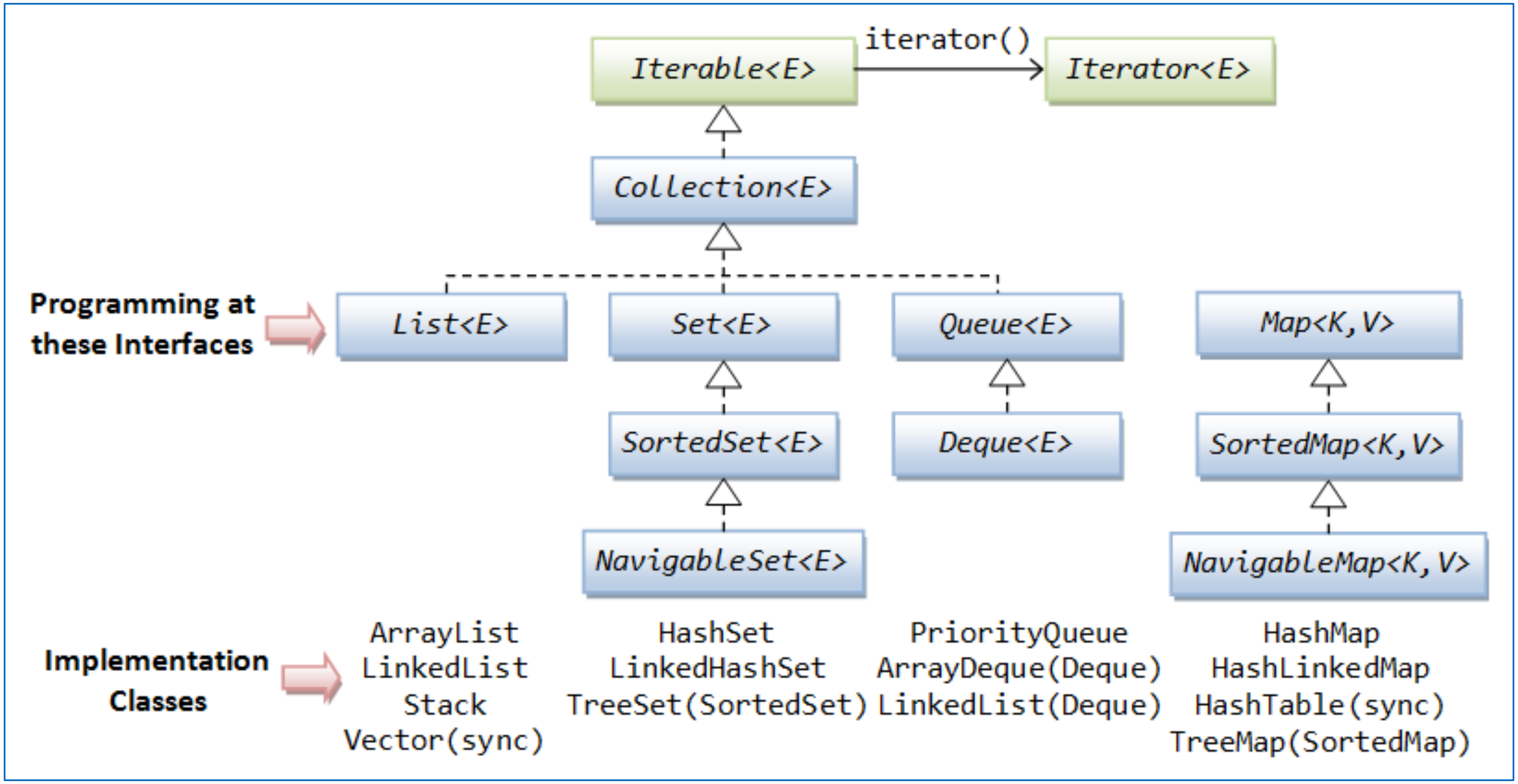
```
ArrayList<String> list = new ArrayList<String>();
//…
for (String element : list)
{
  System.out.println(element);
}
```

For-each loop

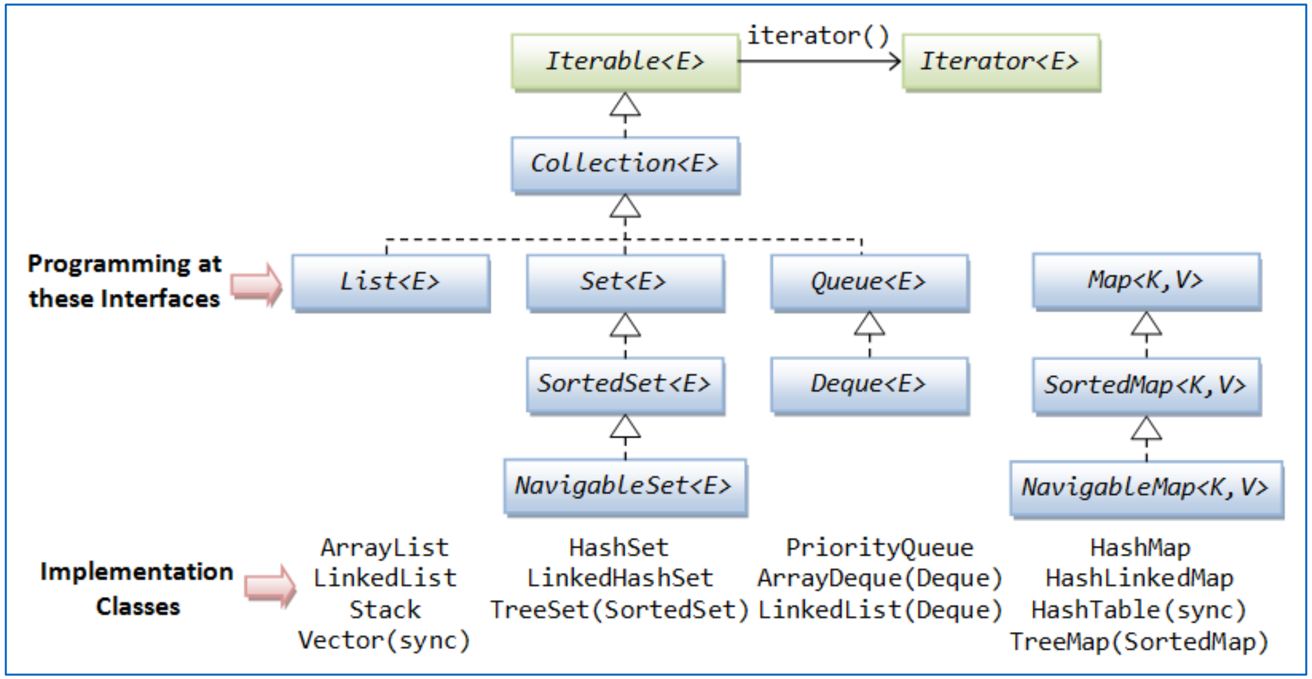
Agenda

- Generic Collections
- Reviewing the Collection Interface
- Summary of Features & Performance
- Working with Collections

Collections Framework

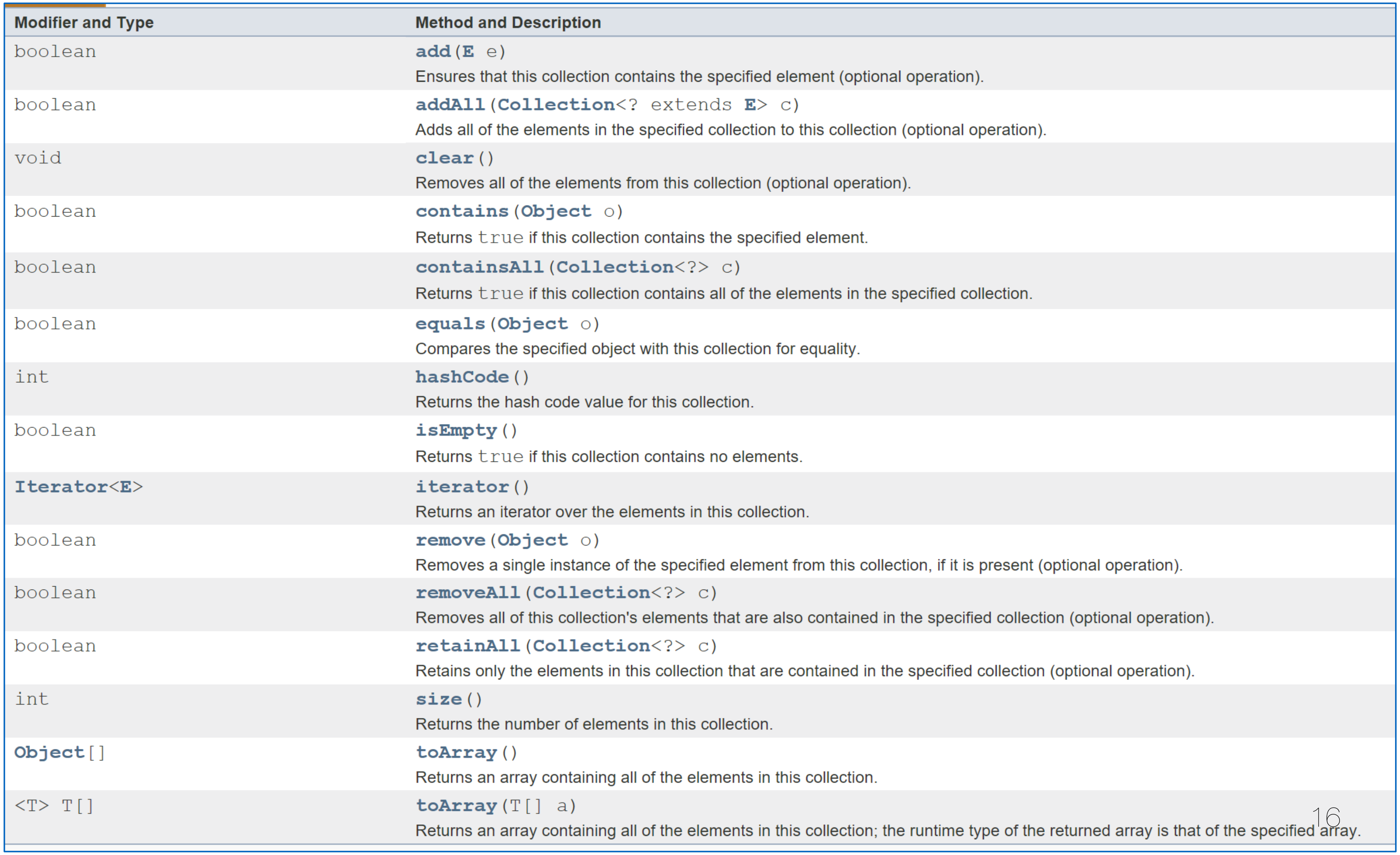


Collection Interface



- Collection is the root of the collection hierarchy
- There is no direct implementation of this interface in JDK
- Concrete implementations are provided for its subtypes

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.

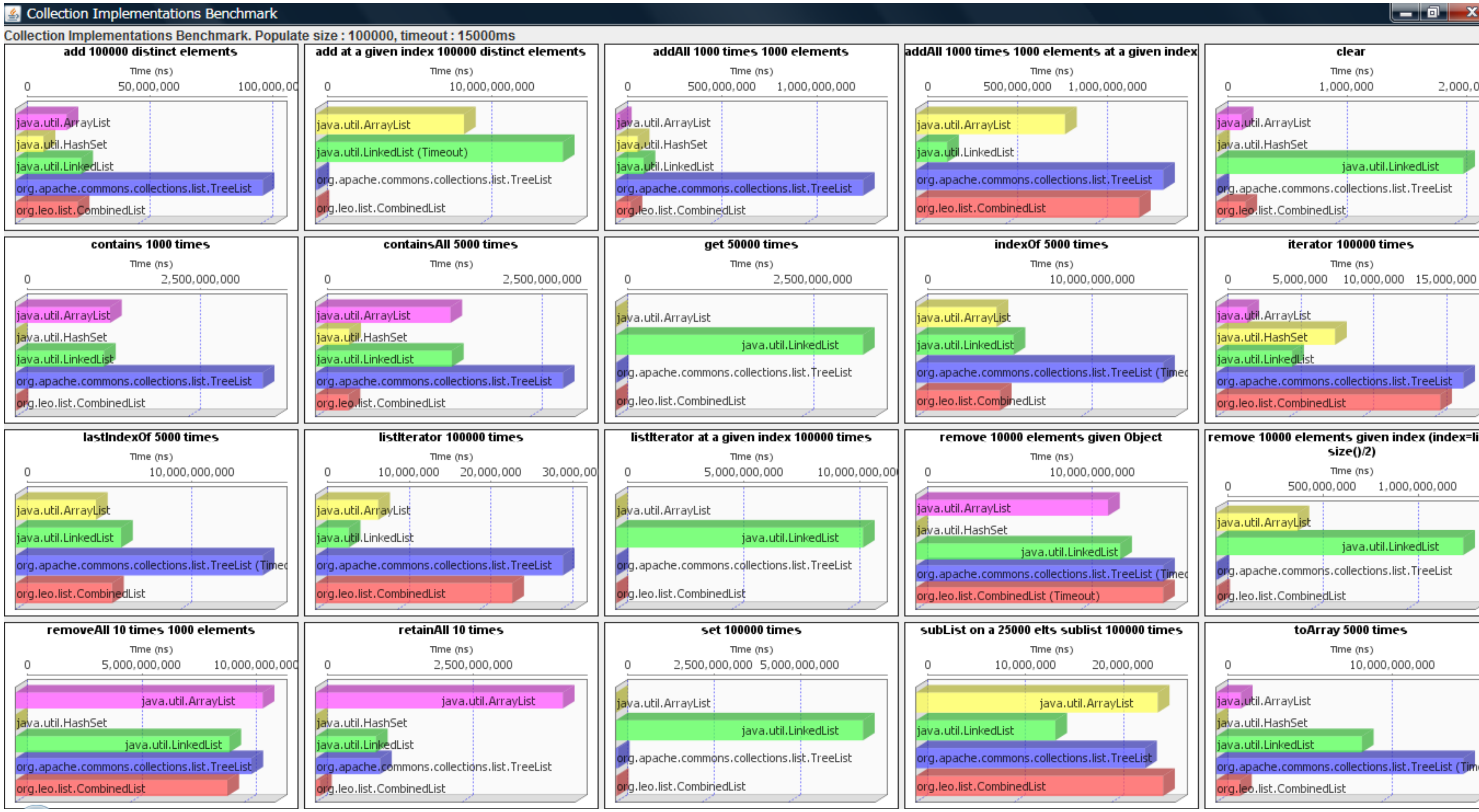
Agenda

- Generic Collections
- Reviewing the Collection Interface
- Summary of Features & Performance
- Working with Collections

Collection Summary

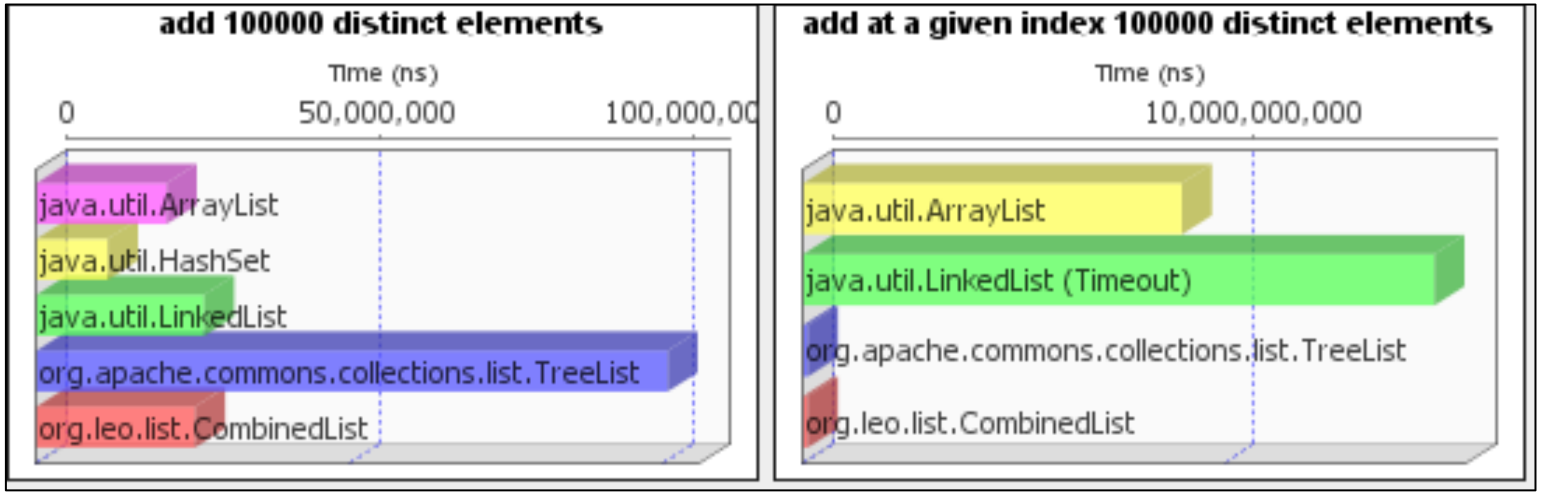
Class	Map	Set	List	Ordered	Sorted	Allow Duplicates
HashSet		X		No	No	No
TreeSet		X		Sorted	By natural order or custom comparison rules	No
LinkedHashSet		X		By insertion order	No	No
ArrayList			X	By index	No	Yes
Vector			X	By index	No	Yes
LinkedList			X	By index	No	Yes
HashMap	X			No	No	No duplicate key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

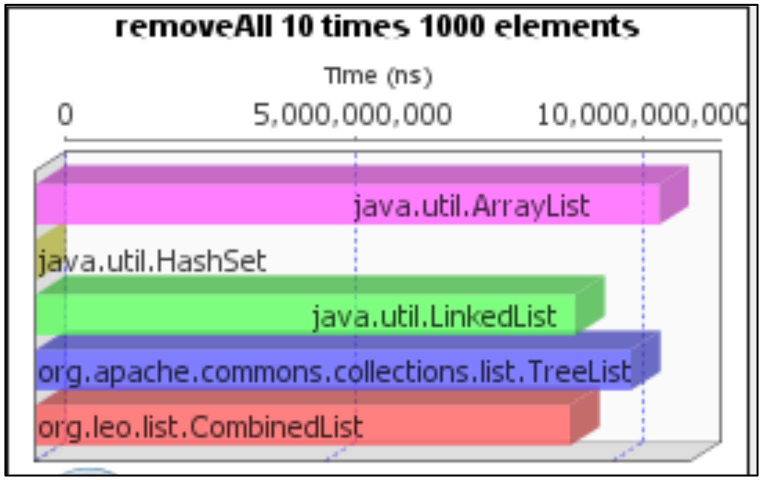
Java Collection Performance

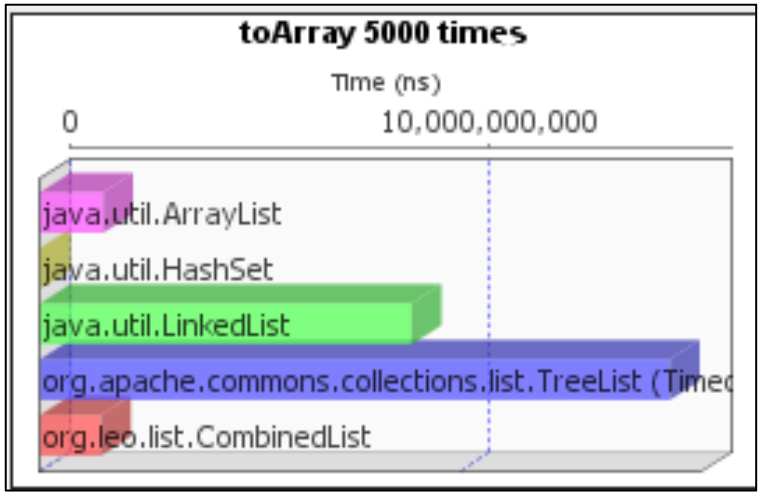


https://dzone.com/articles/java-collection-performance

Java Collection Performance

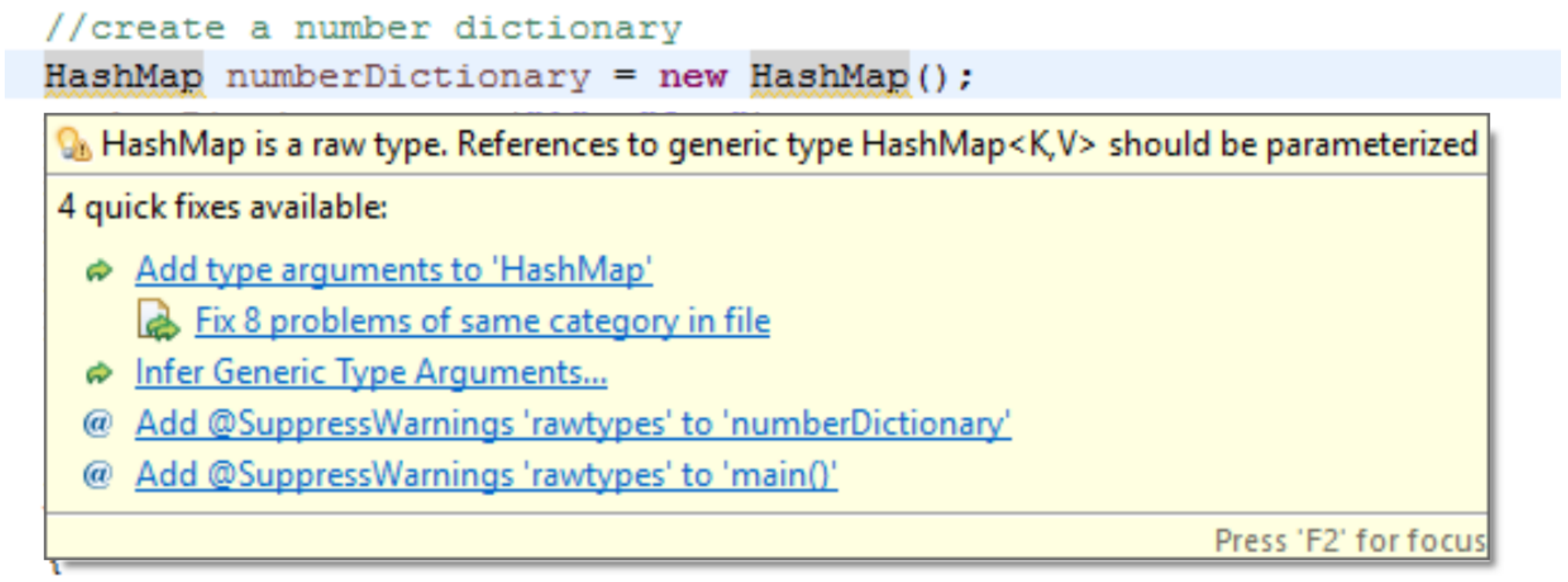






https://dzone.com/articles/java-collection-performance

Compiler warnings for untyped collections



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

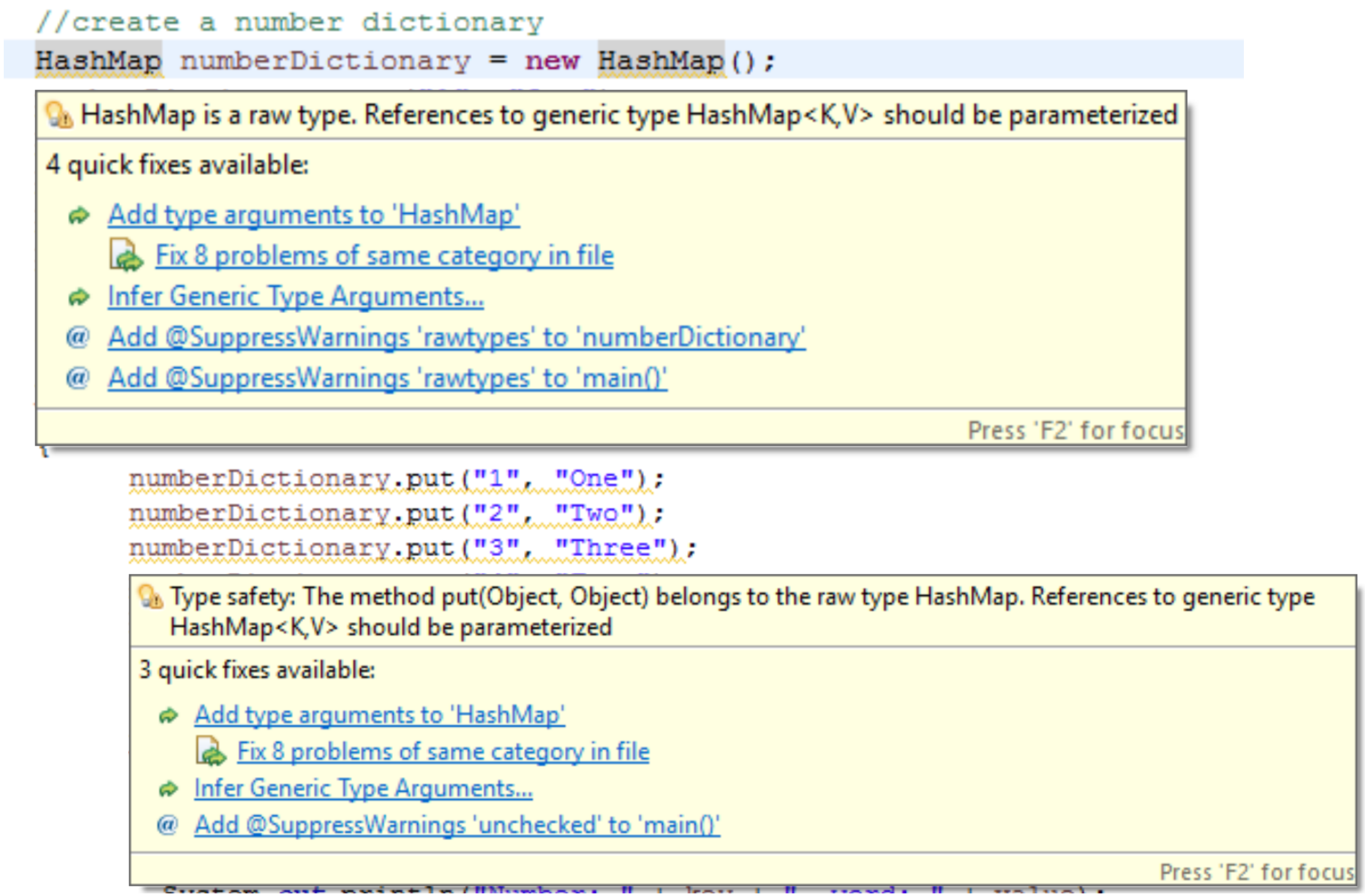
HashMap is a raw type. References to generic type HashMap<K,V> should be parameterized

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

Compiler warnings for untyped (= unsafe) 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:

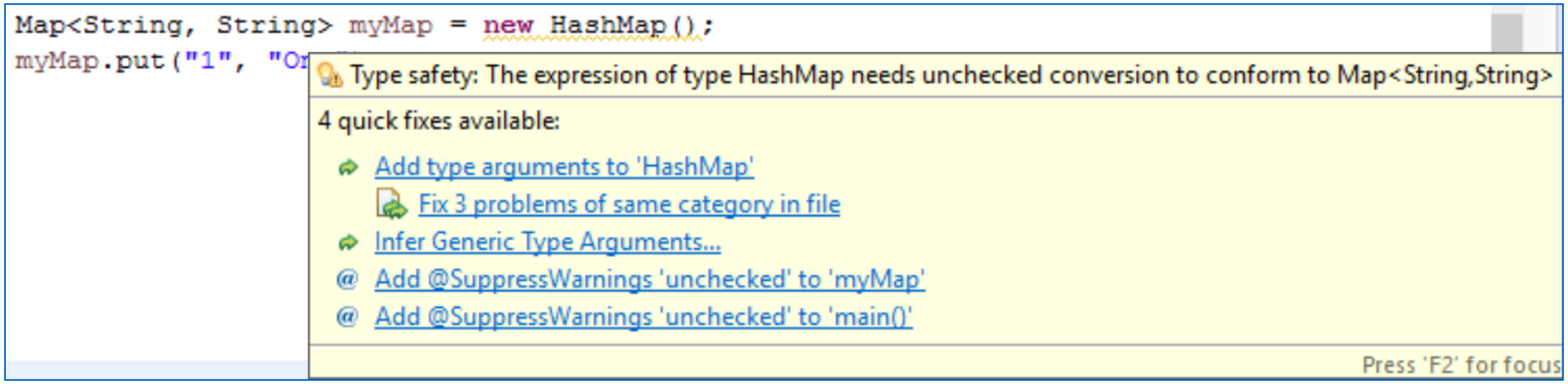
- Add type arguments to 'HashMap'
  - Fix 8 problems of same category in file
- Infer Generic Type Arguments...
- Add @SuppressWarnings 'unchecked' to 'main()'

Press 'F2' for focus

Type Inference

- Since Java 7, type inference applies to collections (<>) :
  - Map<String, String> myMap = new HashMap**<>**();

<> is required.



```
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()'

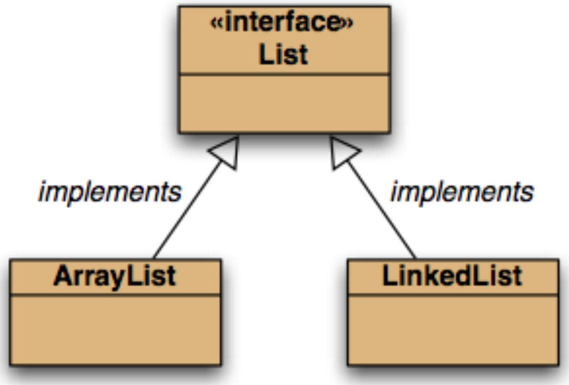
Press 'F2' for focus

Defining Collections

For more **maintainable** code, 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?



Defining Collections

For more **maintainable** code, 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?

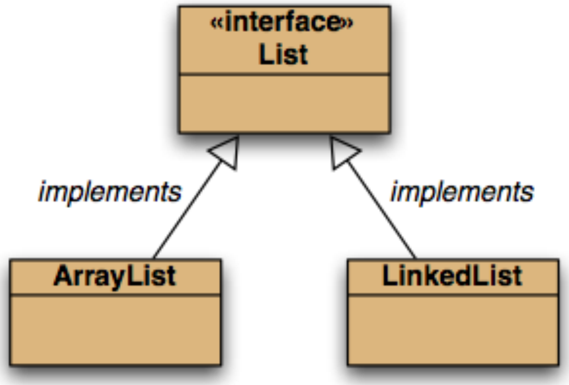
If we want to use a LinkedList instead of an ArrayList → minor changes in the class i.e.

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

becomes

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

and import java.util.LinkedList;



while vs for-each

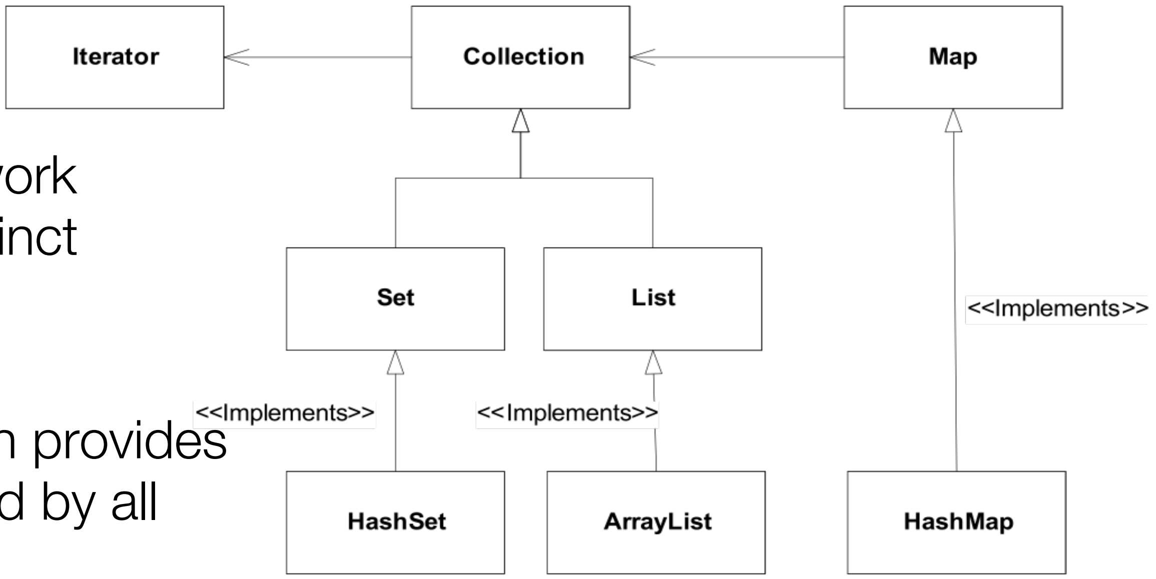
```
List<String> list = new ArrayList<String>();
//…
Iterator <String> iterator = list.iterator();
while (iterator.hasNext())
{
  String element = iterator.next();
  System.out.println(element);
}
```

Standard while loop

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

for-each loop

Summary



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

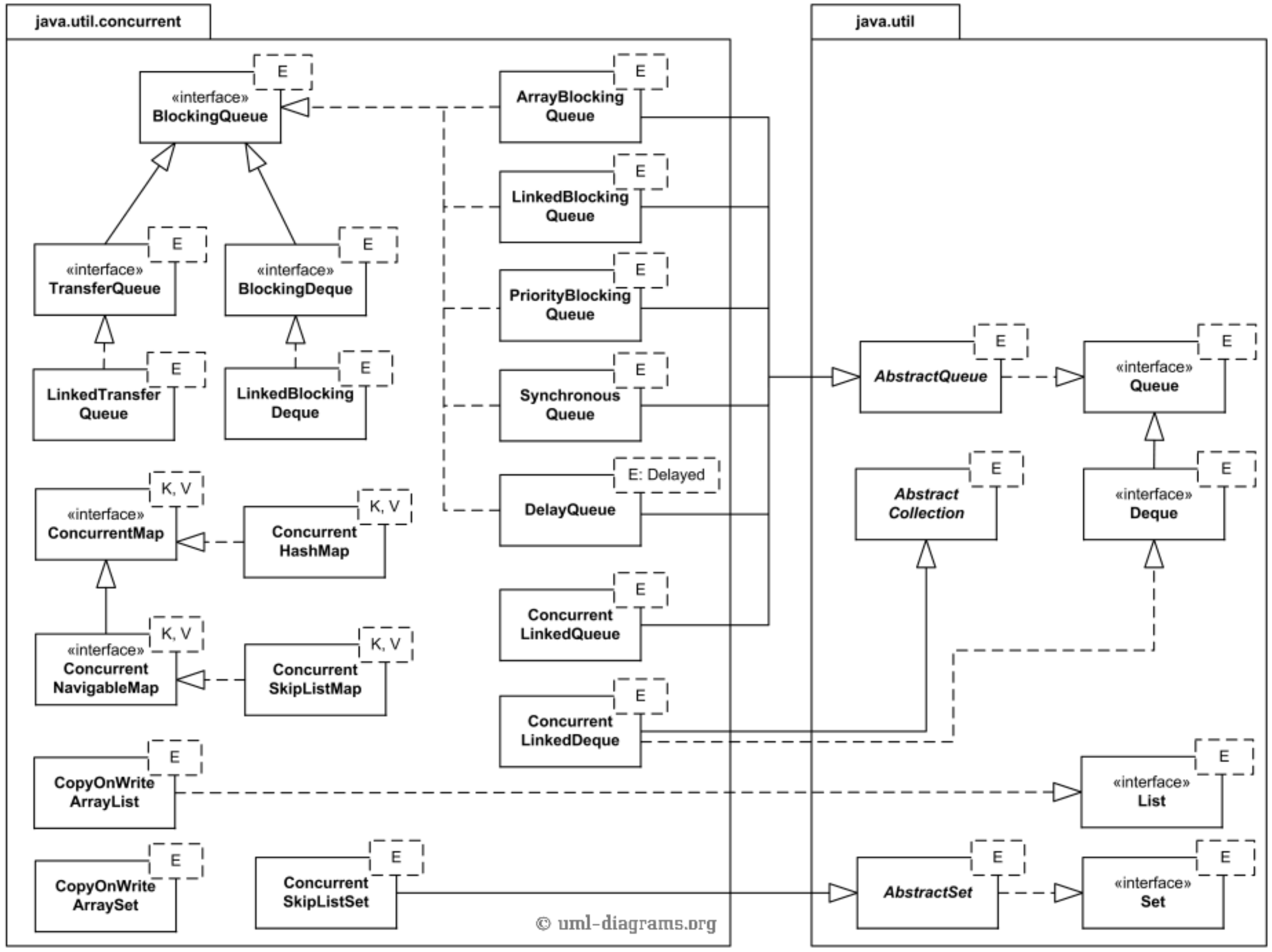
The first tree starts with the Collection interface, which provides for the basic functionality used by all collections

- 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.

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

Concurrent Collections



Used in the context of multi-threaded applications (beyond scope of this course)

Useful References