Agile Software Development

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://

http://

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Collections

Overview

- Collections Architecture
  - Definition
  - Architecture
- Interfaces
  - Collection
  - List
  - Set
  - Map
  - Iterator
- Implementations
  - ArrayList
  - HashMap
  - HashSet
- Java 5 Generic Collections
  - Untyped vs Typed syntax
  - For-each loop

Overview

- Collections Architecture
  - Definition
  - Architecture
- Interfaces
  - Collection
  - List
  - Set
  - Map
  - Iterator
- Implementations
  - ArrayList
  - HashMap
  - HashSet
- Java 5 Generic Collections
  - Untyped vs Typed syntax
  - For-each loop

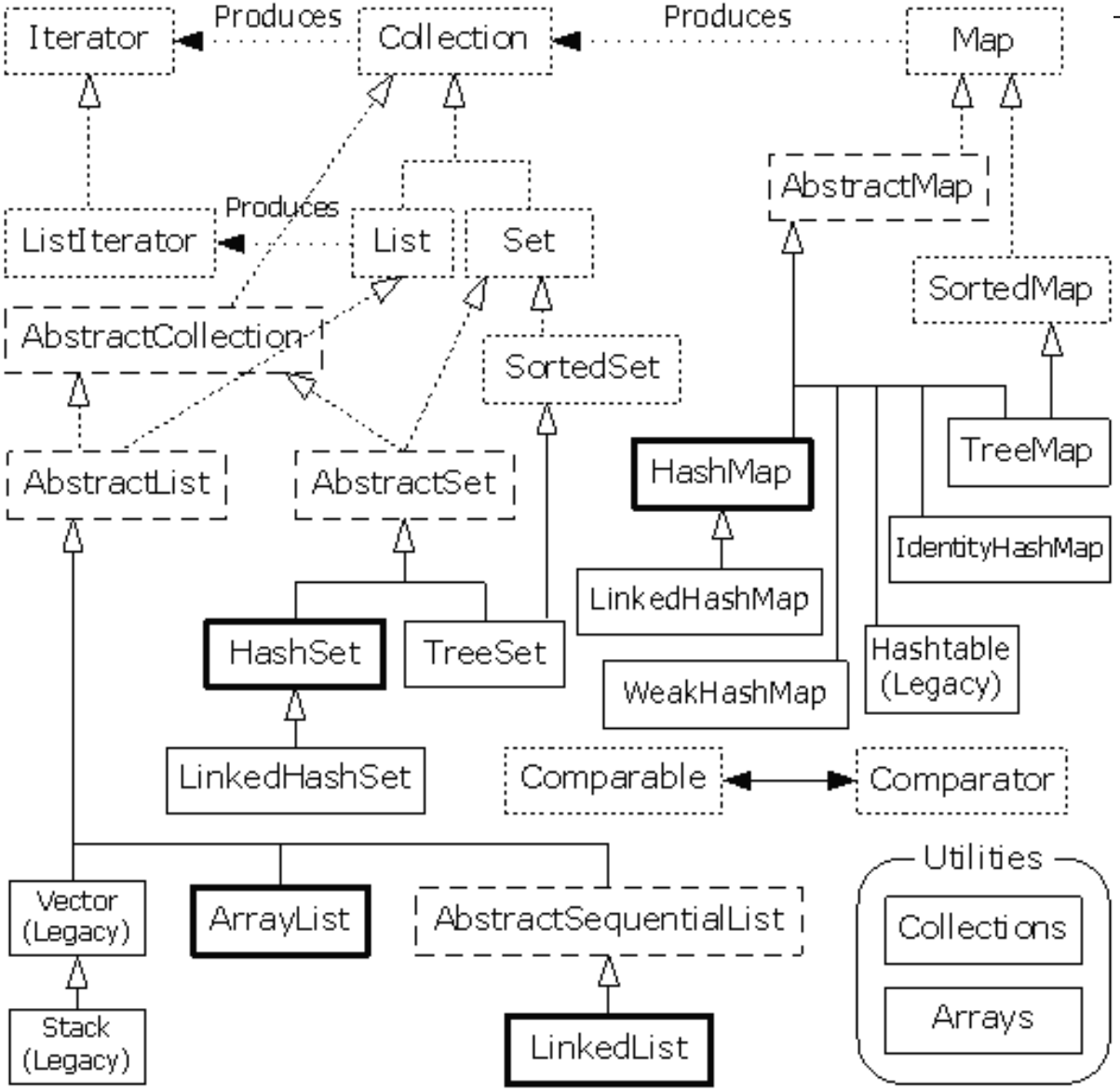
What are Collections?

- Collections are Java objects used to 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
- Java collection architecture includes:
  - Interfaces - abstract data types representing collections
  - Implementation - concrete implementation of collection interfaces
  - Algorithms - methods for manipulating collection objects

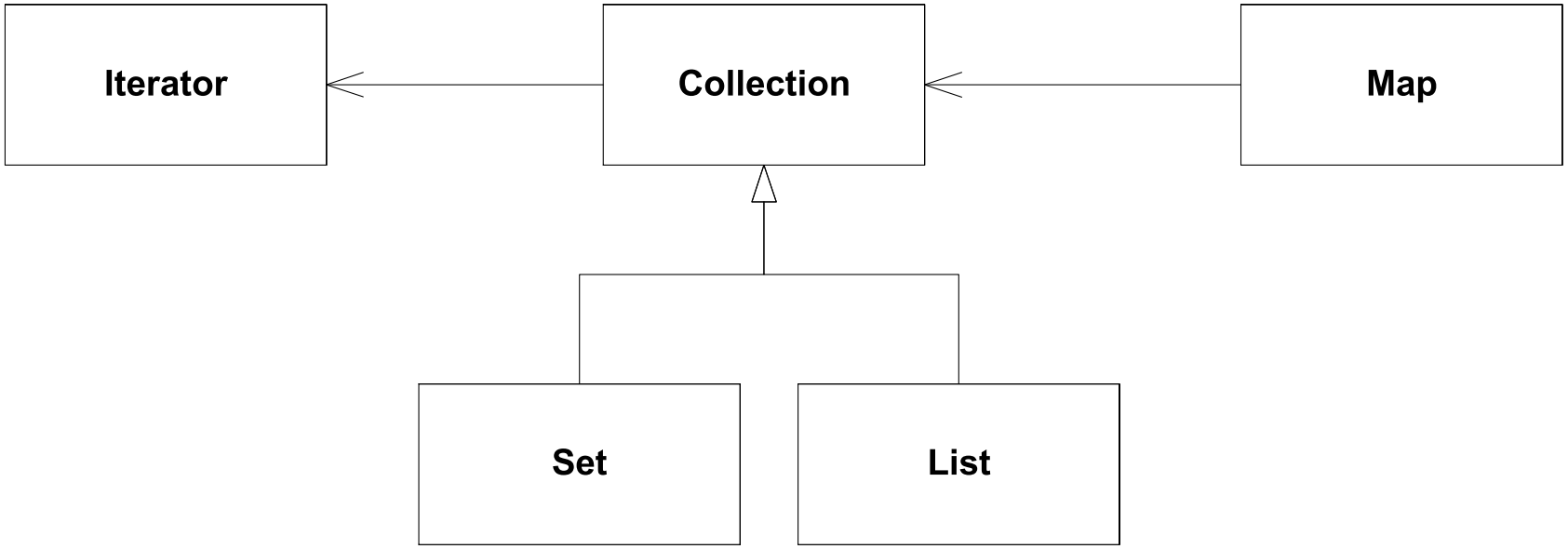
Collection Architecture

- Collection framework benefits include:
  - Reusability
  - Uniformity
  - Faster development
  - Higher quality
  - Interoperability
  - Less programming

Collection Architecture



Interfaces



- Collection “uses” Iterator
- Map “uses” Collection
- Set extends Collection (subtyping)
- List extends Collection (subtyping)

Road Map

- Collections Architecture
  - Definition
  - Architecture
- Interfaces
  - Collection
  - List
  - Set
  - Map
  - Iterator
- Implementations
  - ArrayList
  - HashMap
  - HashSet
- Java 5 Generic Collections
  - Untyped vs Typed syntax
  - For-each loop

Collection Interface

- Collection represents a group of objects
  - These collection objects are known as collection elements
- There is no direct implementation of this interface in JDK
  - Concrete implementations are provided for subtypes
- Collections in general can allow duplicate elements, and can be ordered
  - Unordered collections that allow duplicate elements should implement directly Collection interface

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 iterator;
```

Iterator Interface

- Defines a protocol for iterating through a collection:

```
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();
}
```

Set Interface

- Set is a collection that does not contain duplicate elements
  - This is supported by additional behavior in constructors and add(), hashCode(), and equals() methods
  - All constructors in a set must create a set that does not contain duplicate elements
- It is not permitted for a set to contain itself as an element
- If set element changes, and that affects equals comparisons, the behavior of a set is not specified

List Interface

- List represents an ordered collection
  - Also known as sequence
- Lists may contain duplicate elements
- Lists extend behavior of collections with operations for:
  - Positional Access
  - Search
  - List Iteration
  - Range-view

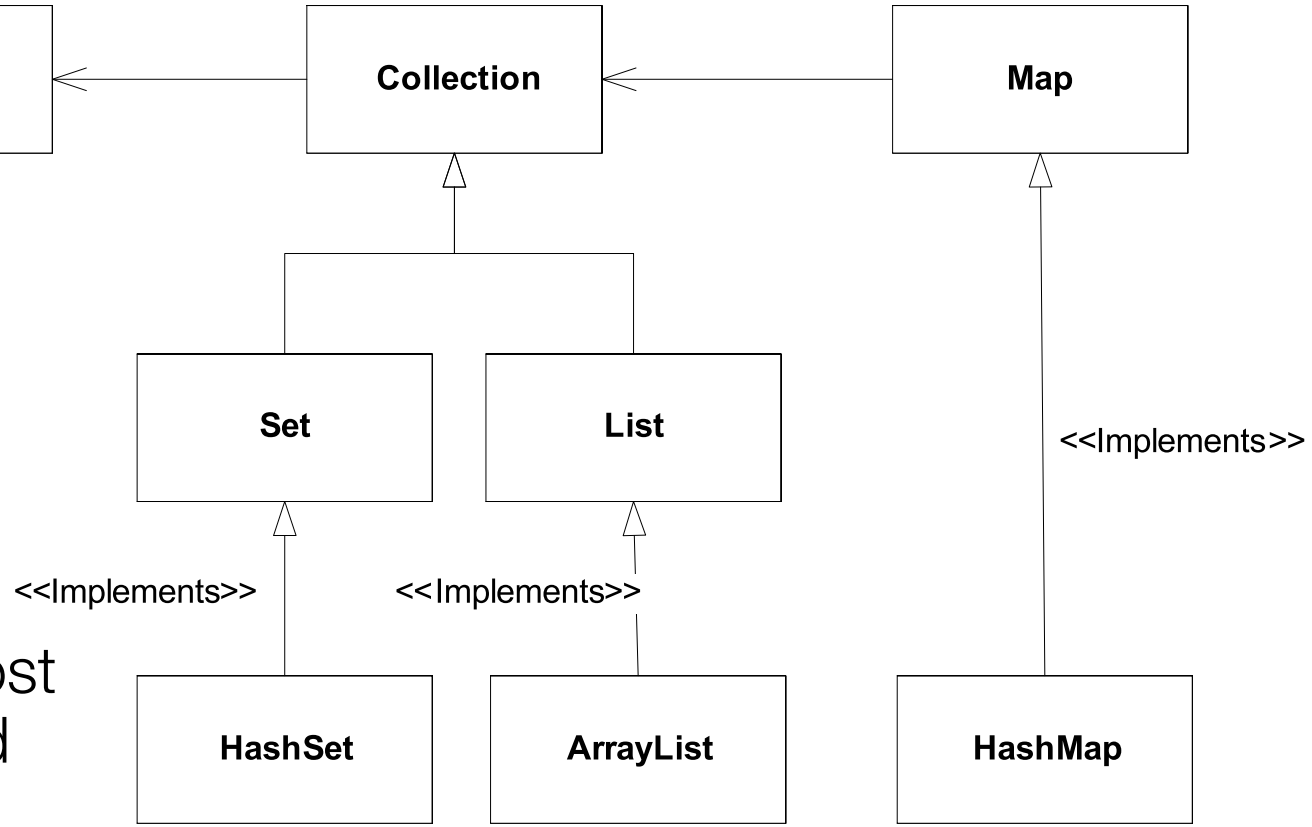
Map Interface

- 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
- Map interface defines protocols for manipulating keys and values

Collections

- Collections Architecture
  - Definition
  - Architecture
- Interfaces
  - Collection
  - List
  - Set
  - Map
  - Iterator
- Implementations
  - ArrayList
  - HashMap
  - HashSet
- Java 5 Generic Collections
  - Untyped vs Typed syntax
  - For-each loop

Most Commonly Used Collections



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

ArrayList

- Represents resizable-array implementation of the List interface
  - Permits all elements including null
- 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
ArrayList 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
```

HashMap

- Collection that contains pair of objects
  - Values are stored at keys
- It is a hash table based implementation of the Map interface
  - Permits null values and null keys
  - 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
HashMap 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);
}
```

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

Console

HashSet

- Concrete implementation of the Set interface
  - Backed up by an instance of HashMap
  - Order is not guaranteed
- 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
- Elements in the set cannot be duplicated

HashSet Example

```
//create new set
HashSet 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]
```

Collections

- Collections Architecture
  - Definition
  - Architecture
- Interfaces
  - Collection
  - List
  - Set
  - Map
  - Iterator
- Implementations
  - ArrayList
  - HashMap
  - HashSet
- Java 5 Generic Collections
  - Untyped vs Typed syntax
  - For-each loop

Java 5 Generic Collection

- 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.

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

```
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

```
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

- 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

Review

- Collections Architecture
  - Definition
  - Architecture
- Interfaces
  - Collection
  - List
  - Set
  - Map
  - Iterator
- Implementations
  - ArrayList
  - HashMap
  - HashSet
- Java 5 Generic Collections
  - Untyped vs Typed syntax
  - For-each loop