Programming technology

Generics

Collections

Dr. Rudolf Szendrei

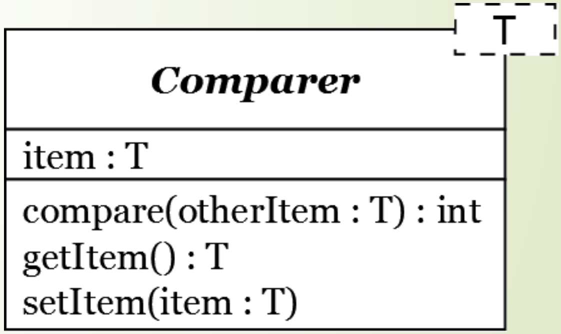
ELTE IK

2020.

Generics

- Java implements parametrized UML classes as generic classes, where the UML parameters are represented with generic parameters.
- In Java, only classes can be generic parameters.

```
public class Comparer<T>{
  private T   item;
  public  T   getItem(){}
  public void setItem(T item){...}
  public int  compare(T otherItem){...}
}
```



T

Comparer

item : T

compare(otherItem : T) : int
getItem() : T
setItem(item : T)

Generics

- We have to define the concrete values of the generic parameters (class names), if we want to use a generic class. Generic parameters are replaced in compilation time to concrete types.
  Since now, we can instantiate the concrete class.

```
public class Comparer<T>{
  private T   item;
  public  T   getItem(){}
  public void setItem(T item){...}
  public int  compare(T otherItem){...}
}
...
Comparer<String> comparer = new Comparer<>();
comparer.setItem("text");
int compared = comparer.compare("something");
```

Generics

- Using generics is a kind of abstraction, but unlike the abstract classes, we want to keep the types abstract, not the methods (at least partially).

```
public class Comparer<T>{
  private T item;
  ...
}

public abstract class GeometricShape{
  public abstract double getArea();
}
```

Generics

- We can use the two kind of abstraction side by side.

```
public abstract class ItemProcessor<T> {
  public abstract T getProccessedItem();
  ...
}
```

Collections

- A collection is an abstract data structure
    - It groups an arbitrary amount of data
    - Data are equally important at the perspective of the solution
    - Data can be processed in a standard way
- Stored data are usually coming from the same type, or at least they are derived from the same type
- Array is not considered as a collection, since its size is fixed. Although, arrays are often used to implement collections.

Collections – Example

```
public class SampleCollection<E> implements Collection<E> {
  @Override public int size(){...}
  @Override public boolean isEmpty(){...}
  @Override public boolean contains(Object o){...}
  @Override public Iterator<E> iterator(){...}
  @Override public boolean add(E e){...}
  @Override public boolean remove(Object o){...}
  @Override public void    clear(){...}
}
```

Collections – Example

```
public class SampleCollection<E> implements Collection<E> {
  ...
  @Override
  public boolean containsAll(Collection<?> c){...}

  @Override
  public boolean addAll(Collection<? extends E> c){...}

  @Override
  public boolean removeAll(Collection<?> c){...}

  @Override
  public boolean retainAll(Collection<?> c){...}

  @Override public Object[] toArray(){...}

  @Override public <T> T[] toArray(T[] a){...}
}
```

Iterating collections

- By iterating over a collection, we take each element of it one by one, and do something with that
- Collections are usually non indexable, but their items can be enumerated with an *iterator*
- Note: the `Double` is a wrapper for the `double` data type, which we need because a generic parameter must be a class.

```
Collection<Double> doubles = Arrays.asList(2.72, 3.14, 42.0);
double sum = 0.0;
for (Iterator<Double> it = doubles.iterator(); it.hasNext();){
  Double d = it.next();
  sum += d;
}
System.out.println(sum);
```

Iterating collections

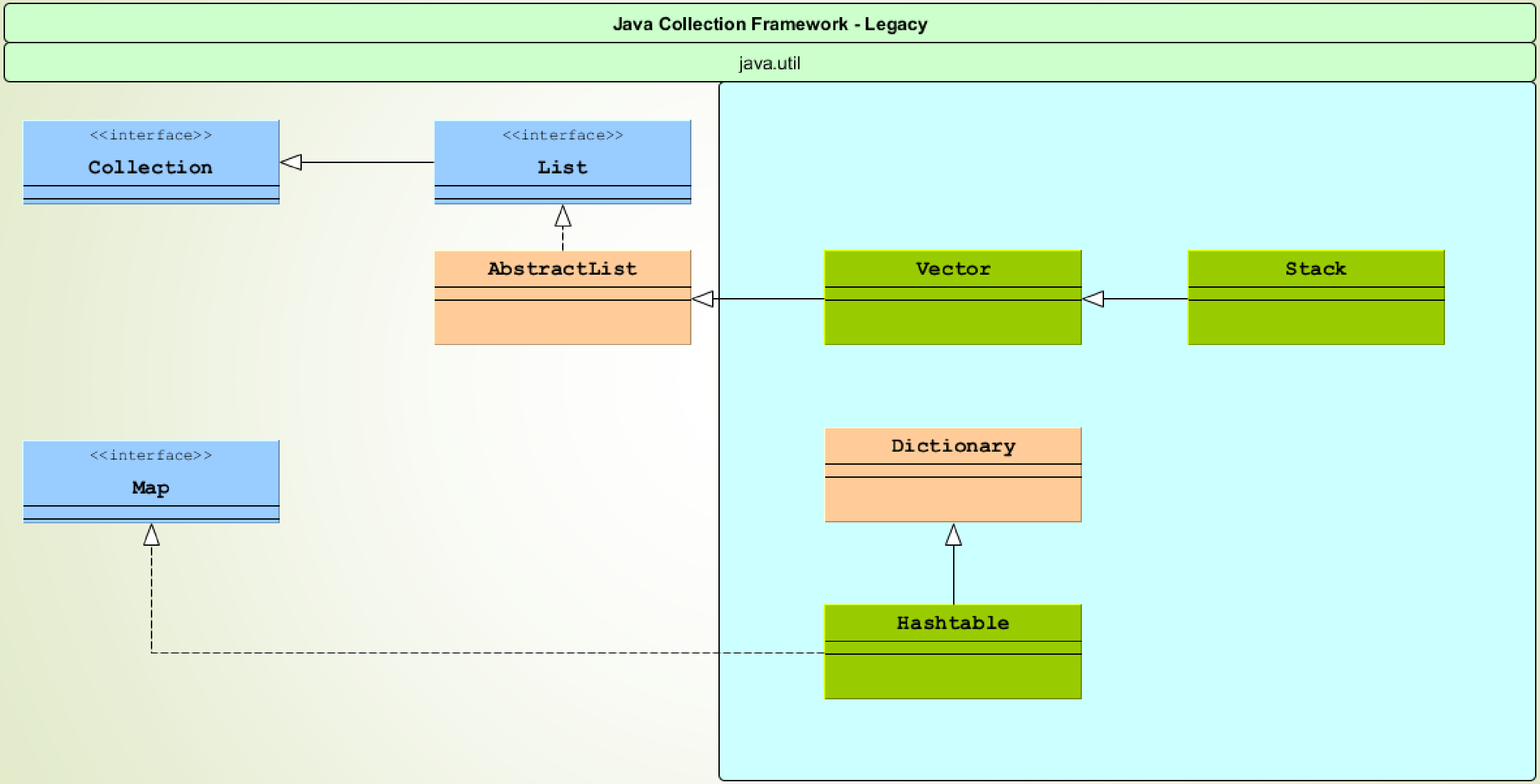
- Collections can be enumerated easier using a *foreach* loop

```
Collection<Double> doubles = Arrays.asList(2.72, 3.14, 42.0);
double sum = 0.0;
for (Double d : doubles) { sum += d; }
System.out.println(sum);
```

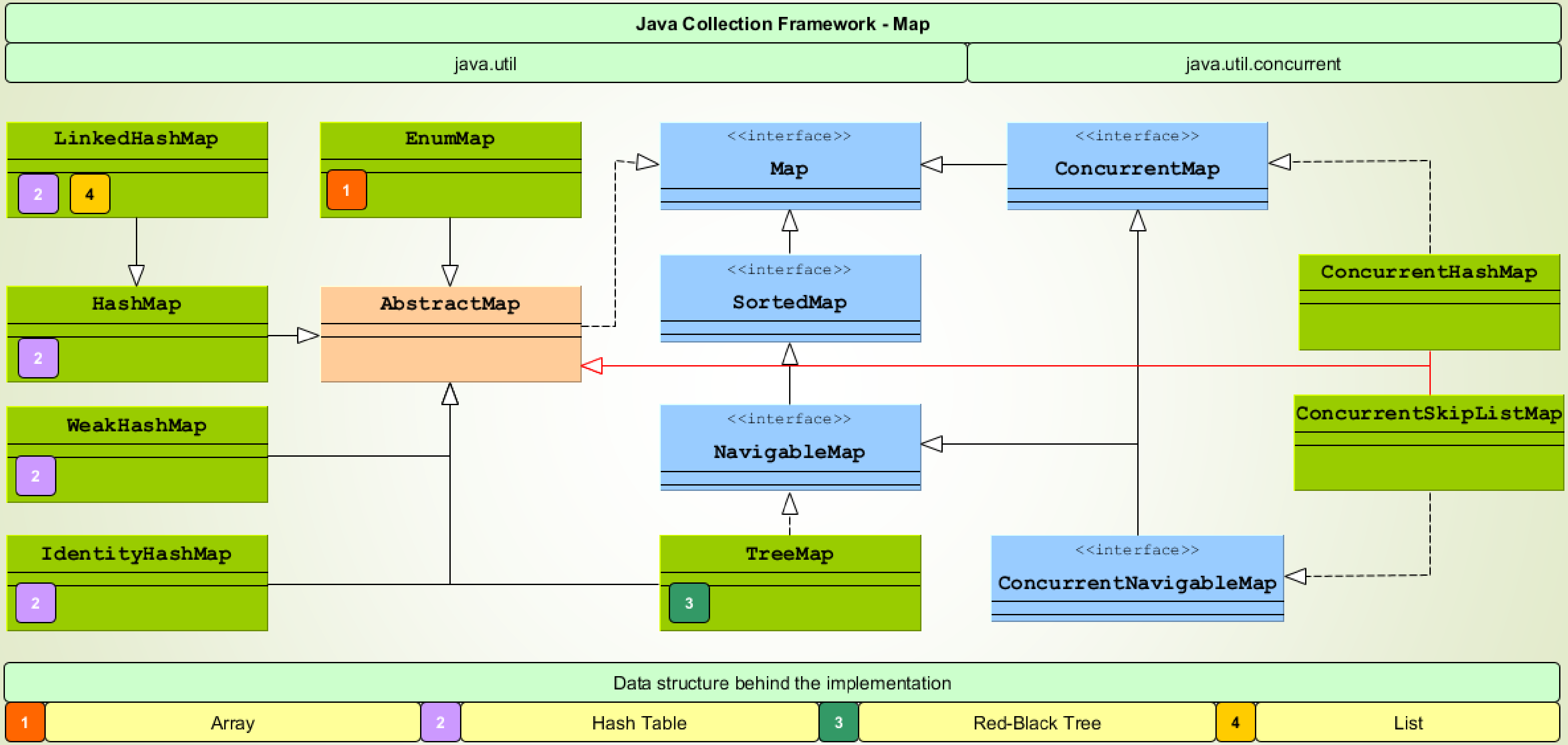
Implementation of Collections

- We have to implement the `Collection<E>` interface, or any of its specialized interfaces
- It worths to implement a collection as a generic, so it will be possible to store several types of data in it
- The abstract methods of the collection are already given in the interface, so we only have to consider the representation of the data and the implementation of the methods
- The `AbstractCollection<E>` class already has the regular behavior of a collection, so it worths to derive our collection from it, and focus only to the important things

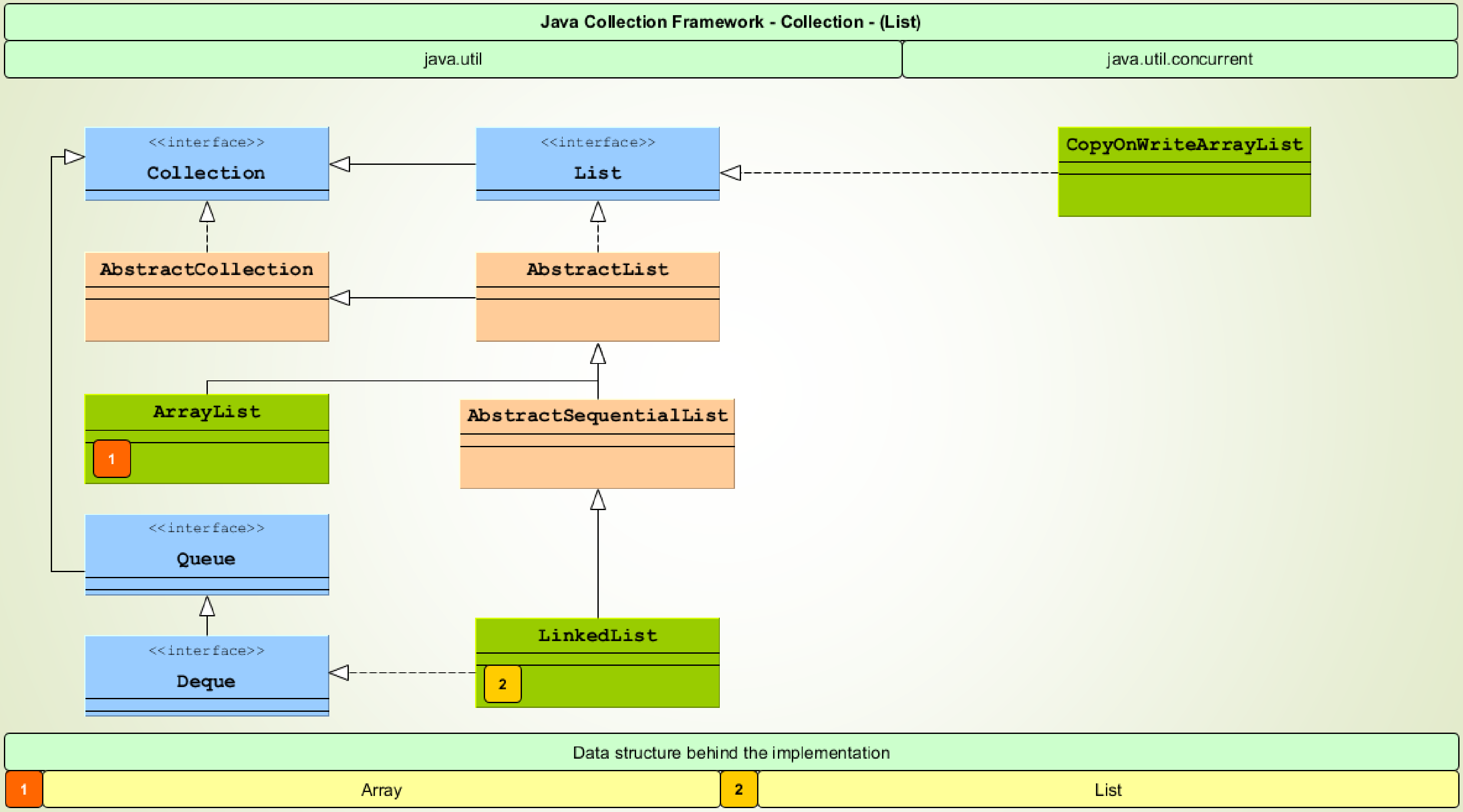
Java Collections



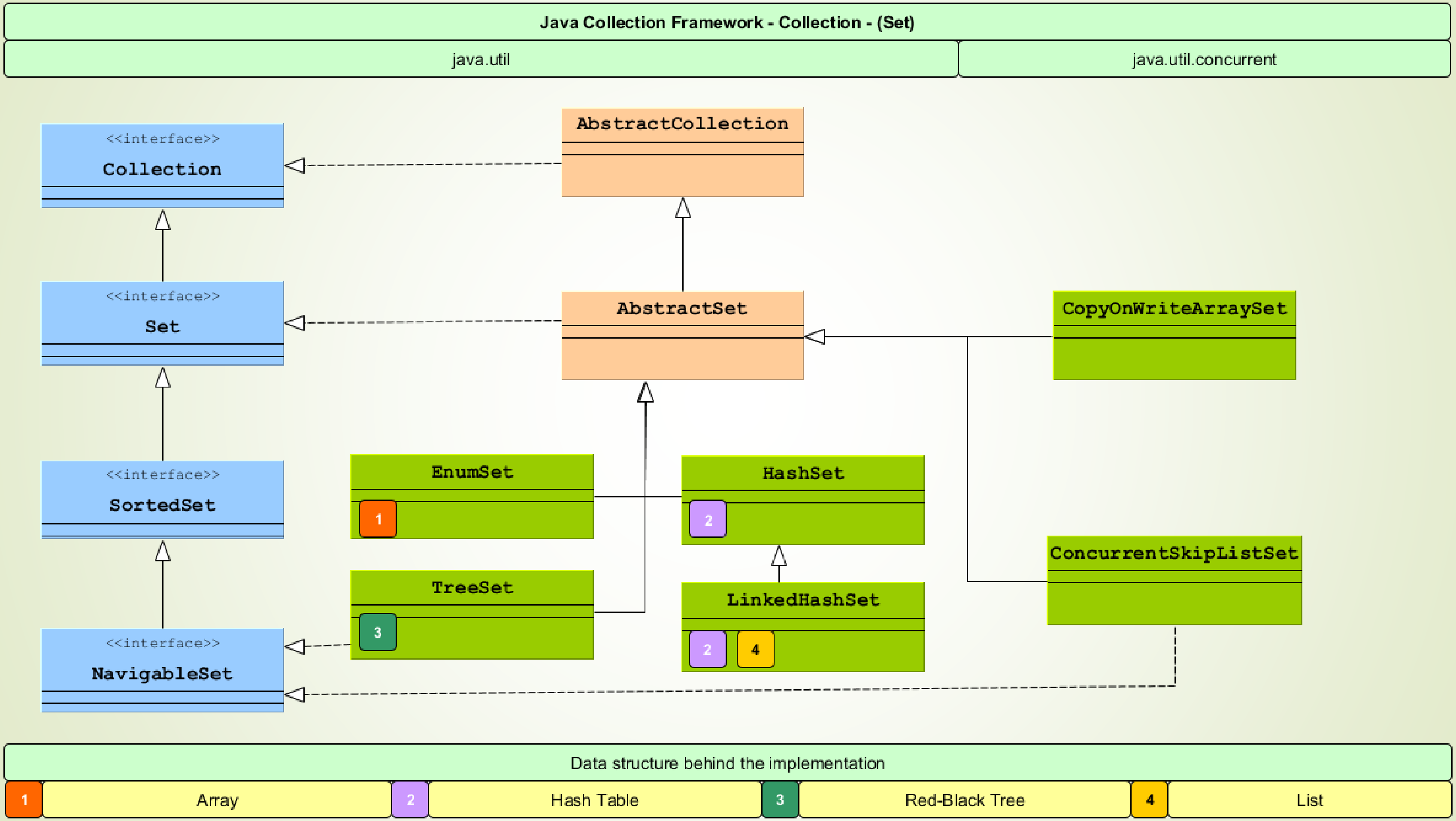
Java Collections



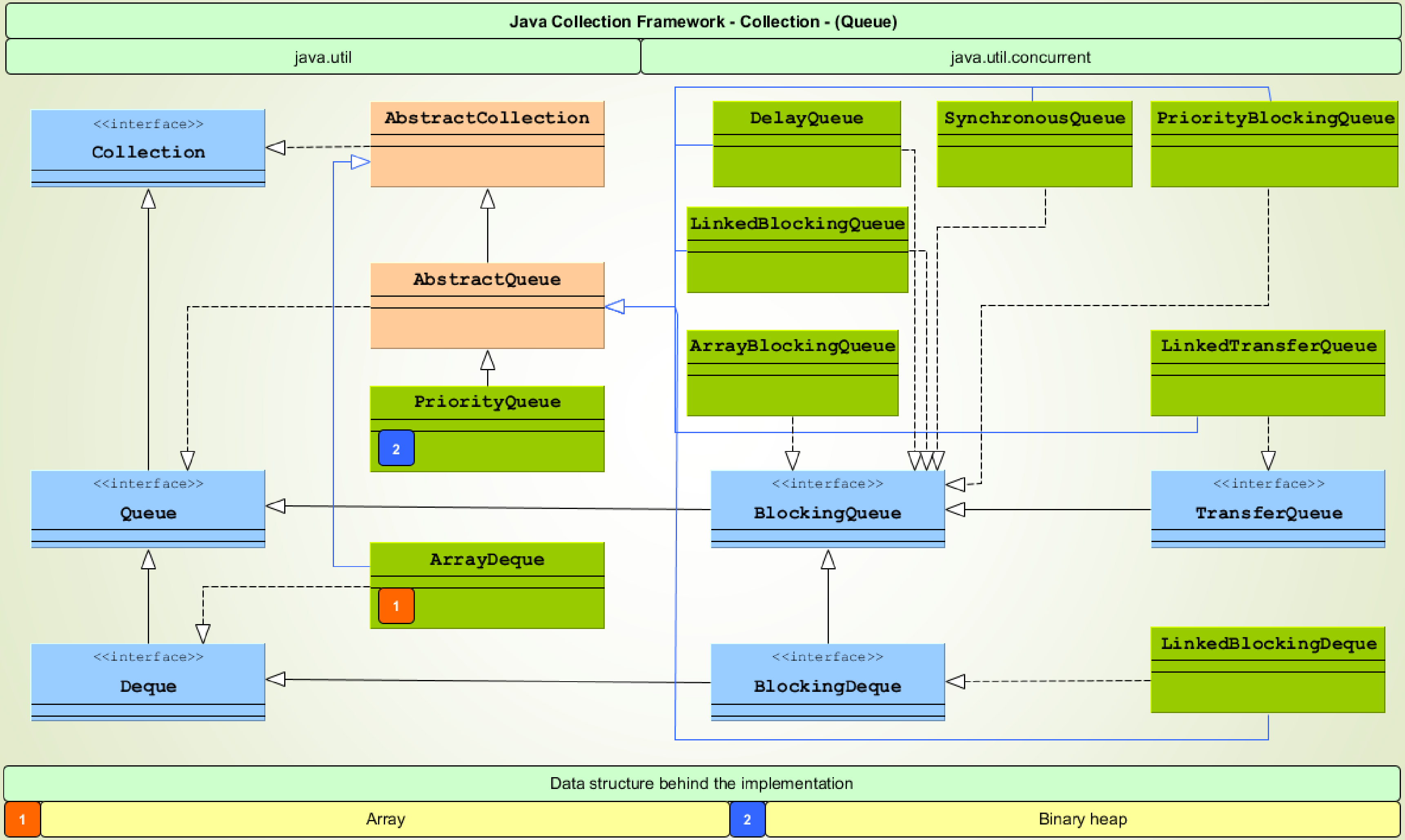
Java Collections



Java Collections



Java Collections



Java Collections

Collections as data structures

- The time of item insertion, modification and removal highly depends on the choosen data structure (see subject Algorithms and data structures)
- Some rules of thumb for beginners:
  - Data structure is choosen considering the critical run time (e.g.: searching and insertion);
  - Data structure is choosen considering the processing of the items (e.g.: sorting);
  - We want to process small amount of data easily;
- In case of a special task, we may create our own data structure (e.g.: in case of geospatial tasks).

Java Collections

Collections as data structures

- Java collections can be grouped based on the used data structure:
  - Direct accessible, indexable;
  - Linked list;
  - Tree;
  - Hash function based.
- In some cases it is possible that we need to implement a hybrid data structure
  (e.g.: we also want to store the leaves of a tree in a double linked list.).

Java Collections

Direct accessible, indexable collections

- Main properties:
  - Constant access time,
  - Slow insertion,
  - Easy sorting

- Frequently user Java classes:
  - ArrayList,
  - Vector,
  - Stack...

Java Collections

Linked-list data structure based collections

- Main properties :
  - First/last list item can be accessed directly,
  - Inner items of the list may be accessed slowly,
  - Easily expandable

- Frequently user Java classes :
  - Queue,
  - Deque,
  - PriorityQueue,
  - LinkedList...

Java Collections

Tree data structure based collections

- Main properties:
  - Time of access, modification and removal is logarithmic,
  - Easily expandable,
  - Indexing can be implemented (choosing the *n* th item)
  - Good choice for associative containers

- Frequently user Java classes :
  - TreeSet,
  - TreeMap...

Java Collections

Hash function based collections

- Main properties:
  - Fast access, modification and removal time,
  - Can be super fast considering a small amount of items and a proper hash function,
  - Non-indexable
  - Items cannot be sorted
  - Good choice for associative containers

- Frequently user Java classes :
  - Hashtable
  - HashSet, LinkedHashSet,
  - HashMap, LinkedHashMap...

Usage of Java Collections

Important!

- In case of collections, where items put into a container based on the comparison of the items, the `equals` method must be implemented! (e.g.: `Set`, `Map` etc.)

- If the chosen collection uses a hash function, then both the `equals` and `hashCode` methods must be implemented! (e.g.: `HashSet`, `HashMap` etc.)

Built-in Java algorithms

- Algorithms on collections - java.util.Collections

  http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

- Algoritms on arrays - java.util.Arrays

  http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html