Chap 2: Data collections (containers)

ILO: Collect data with the consideration of efficiency.

Acknowledgement: https://docs.oracle.com/javase/tutorial/collections/intro/index.html

By: Dr, Jananie Jarachanthan, Senior Lecturer, Department of Computer Engineering, Faculty of Engineering, University of Jaffna.

Collections

- An object that represents a group of objects.
- Typical data belongs to a natural group (e.g. list of books in library).
- Different objects in a similar collection.
- So,
  - Store/retrieve and manipulate.
  - Transmit data from one methods to another.

Java Collections Framework

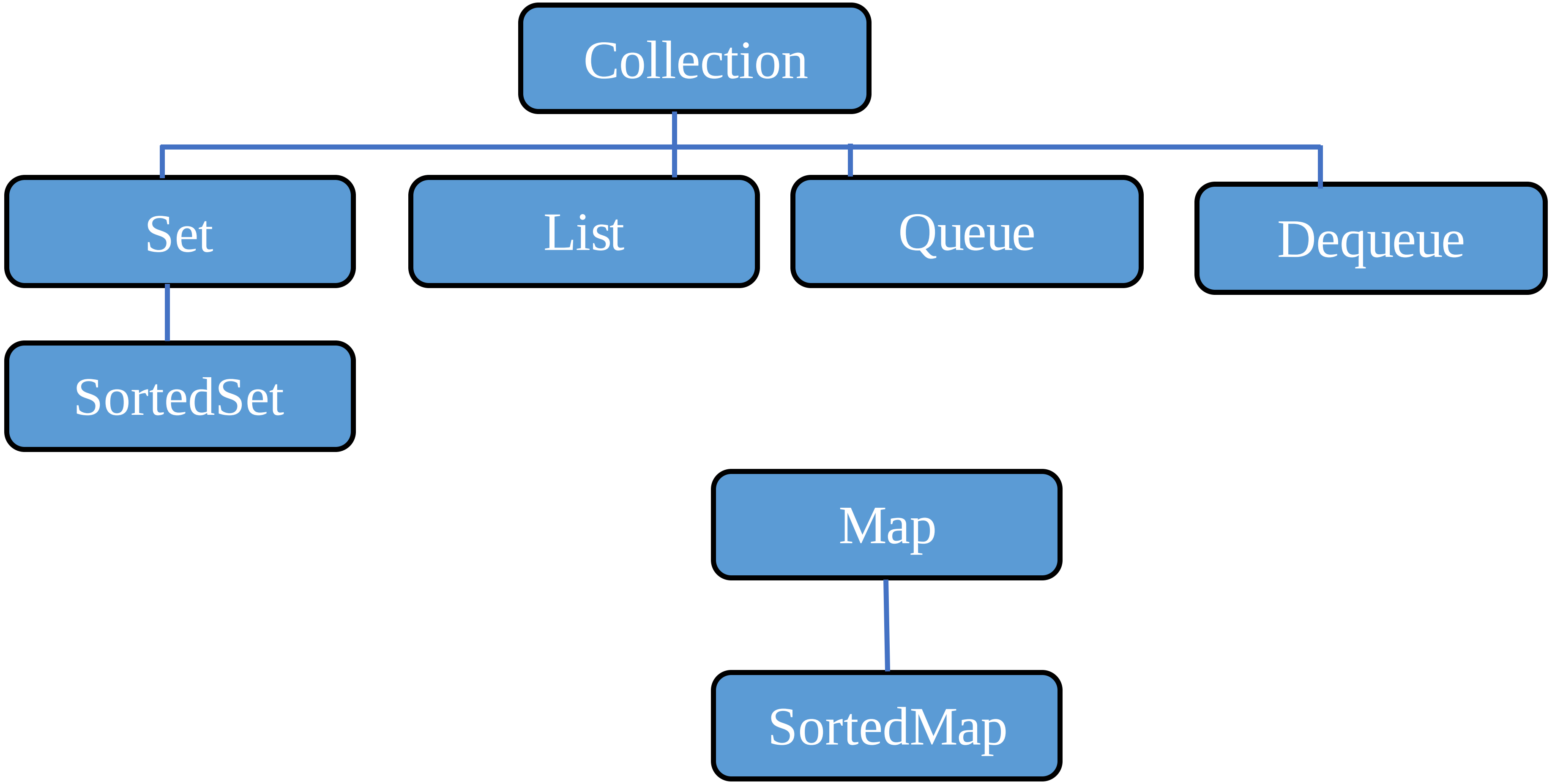
- A collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.

  Interface

  Implementation

  Algorithm (example: sorting, searching etc.)

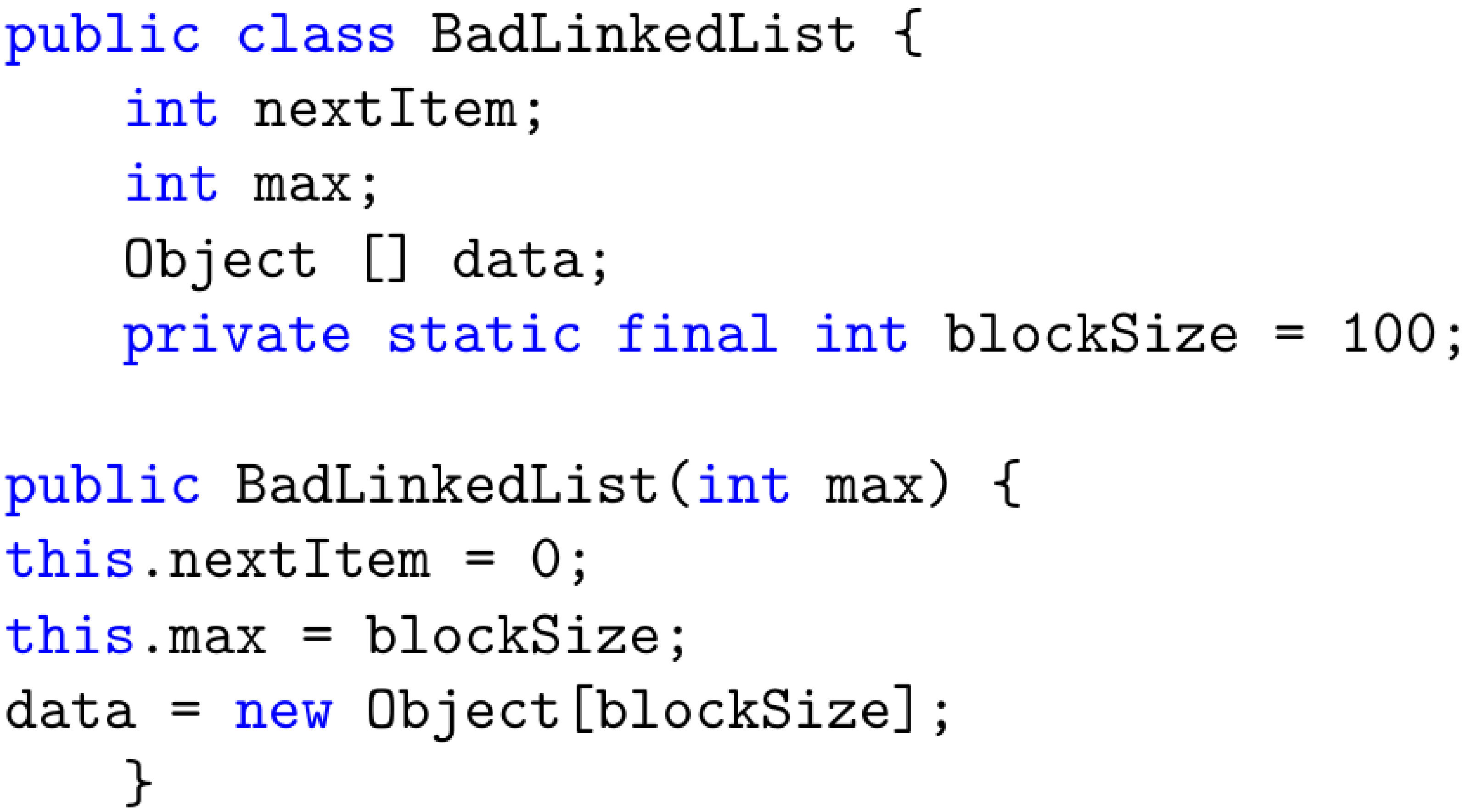
The core collection interfaces



The core collection interfaces

- Set — A collection that cannot contain duplicate elements, Order is not important.
- SortedSet — A Set that maintains its elements in ascending order.
- List — An ordered collection (sometimes called a sequence). Lists can contain duplicate elements.
- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. Order is not important.
- SortedMap — a Map that maintains its mappings in ascending key order.

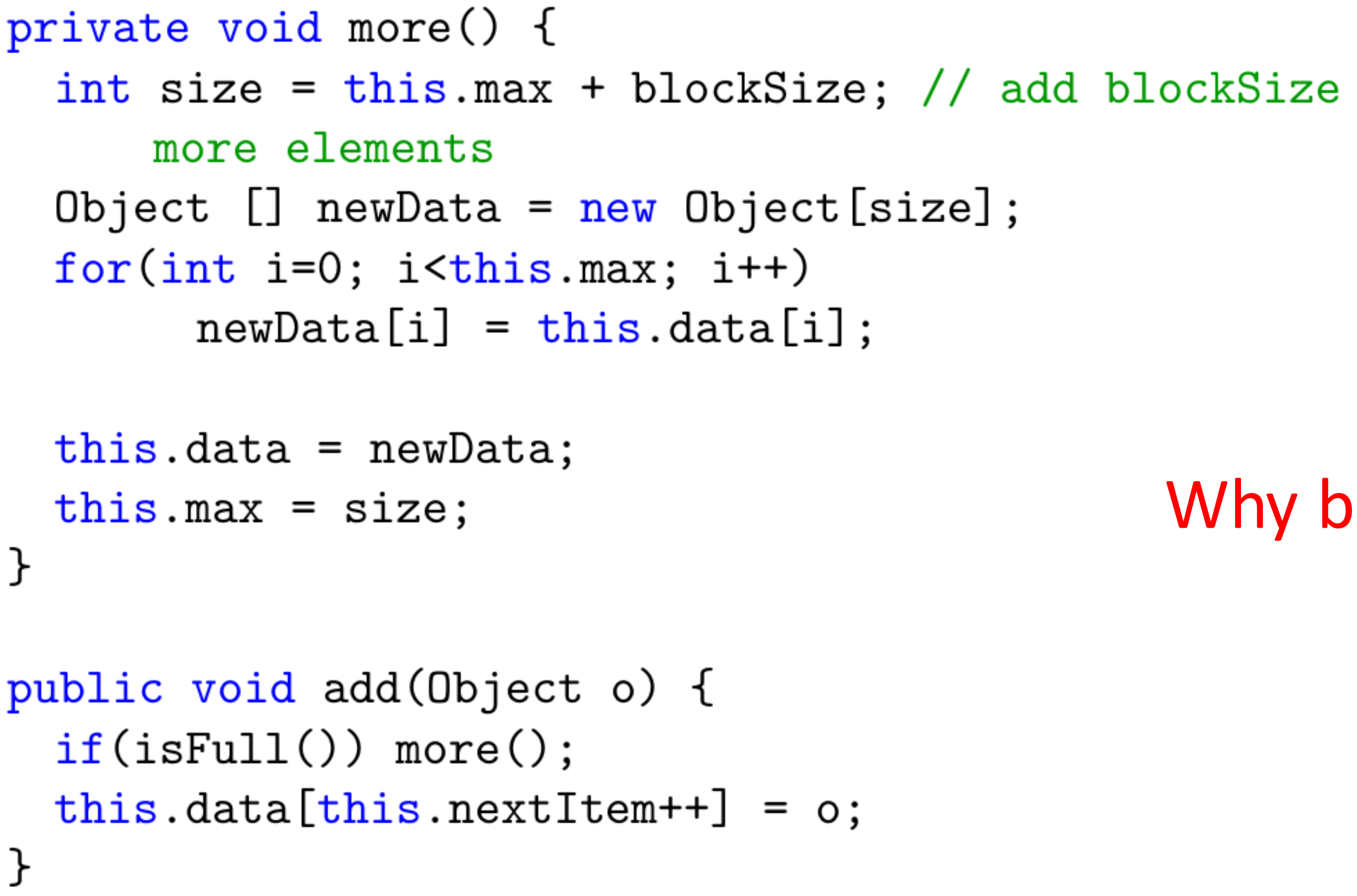
Generics: Motivation



```
public class BadLinkedList {
    int nextItem;
    int max;
    Object [] data;
    private static final int blockSize = 100;

public BadLinkedList(int max) {
this.nextItem = 0;
this.max = blockSize;
data = new Object[blockSize];
    }
```

Generics: Motivation



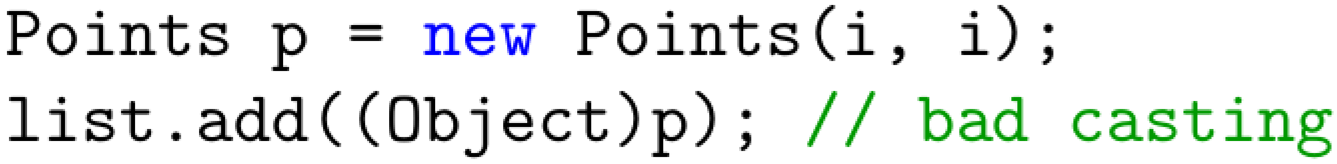
```
private void more() {
  int size = this.max + blockSize; // add blockSize
      more elements
  Object [] newData = new Object[size];
  for(int i=0; i<this.max; i++)
        newData[i] = this.data[i];

  this.data = newData;
  this.max = size;
}

public void add(Object o) {
  if(isFull()) more();
  this.data[this.nextItem++] = o;
}
```

Why bad?

Casting



```
Points p = new Points(i, i);
list.add((Object)p); // bad casting
```

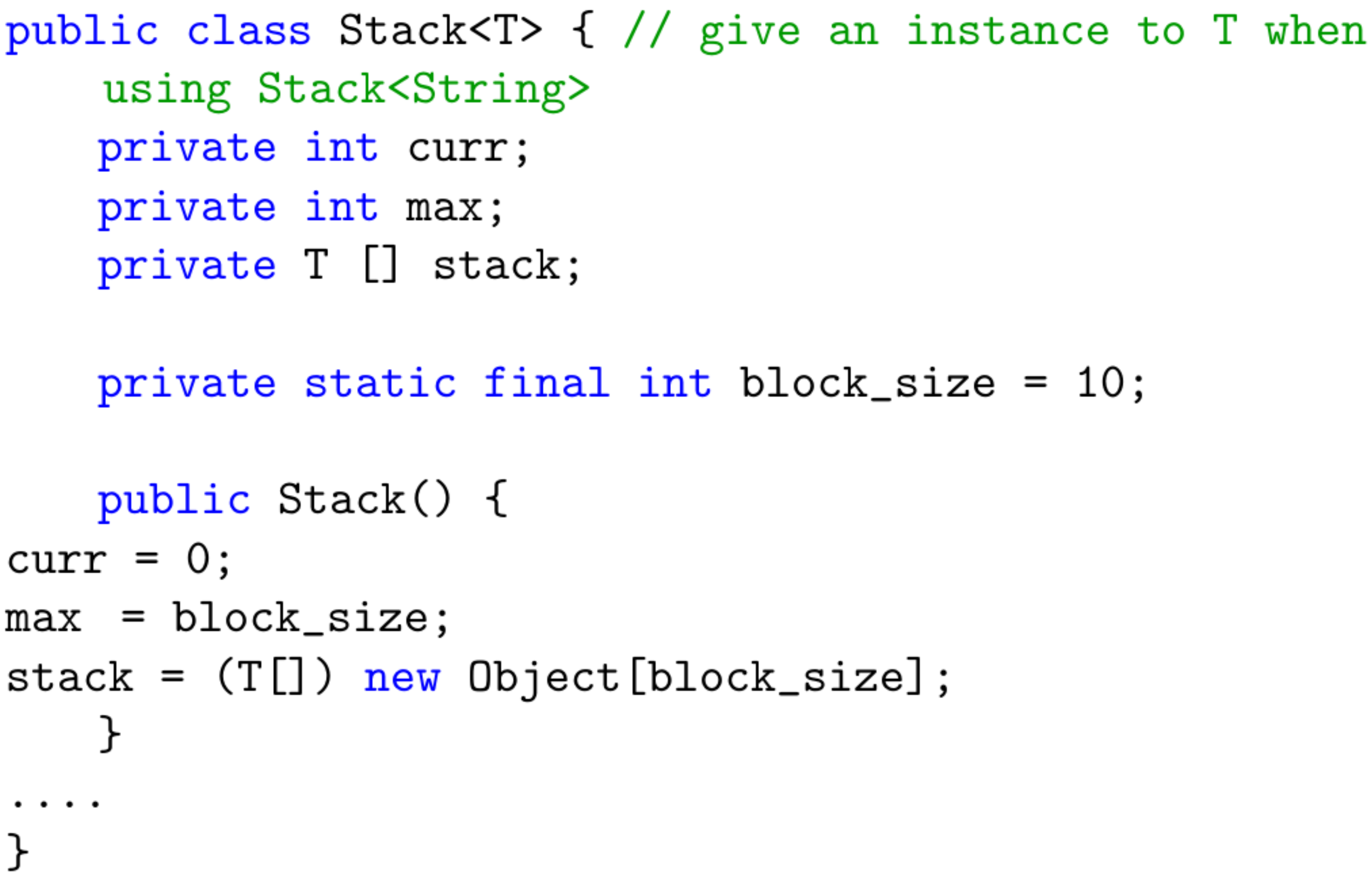
Generics

- Types to-be specified-later.
- Instantiated when needed for specific types.

Why?

- No casting needed.
- Compile time errors.

Example

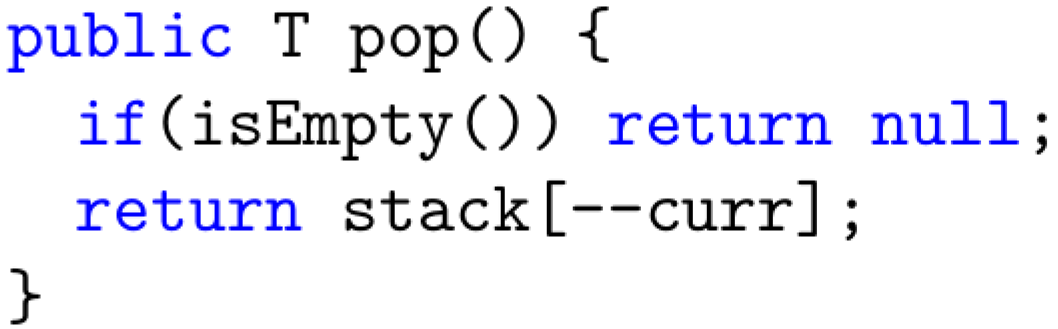


```
public class Stack<T> { // give an instance to T when
    using Stack<String>
    private int curr;
    private int max;
    private T [] stack;

    private static final int block_size = 10;

    public Stack() {
curr = 0;
max  = block_size;
stack = (T[]) new Object[block_size];
    }
....
}
```

Example



```
public T pop() {
  if(isEmpty()) return null;
  return stack[--curr];
}
```

Parameterized type

- class Stack<T> { /* stack of T can take many */
- class Stack<T1, T2, ... // example of taking many
- Stack <Points> pntStack = new Stack <Points> (); // create a stack of points
- Stack <String> strStack = new Stack <String> (); //create a stack of strings

Parameterized type

- E - Element (used extensively by the Java Collections Framework)
- T - Type
- N - Number
- K - Key
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Restrictions with Generics

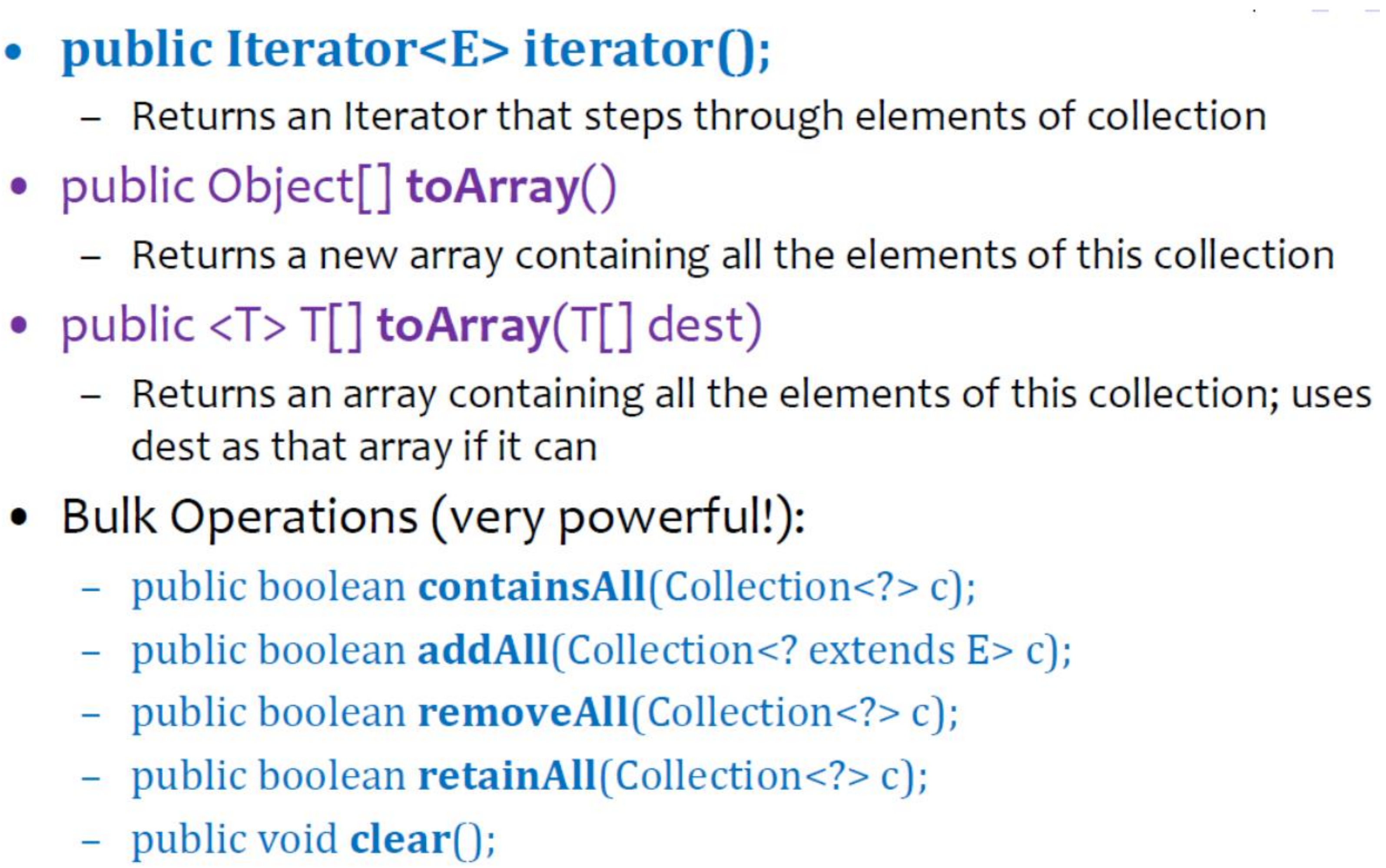
- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

- Collections continues ...

Interface

- public int **size**( )
- public boolean **add**(Object obj)
- public boolean **contains**(Object obj)
- public boolean **isEmpty**( )

Interface

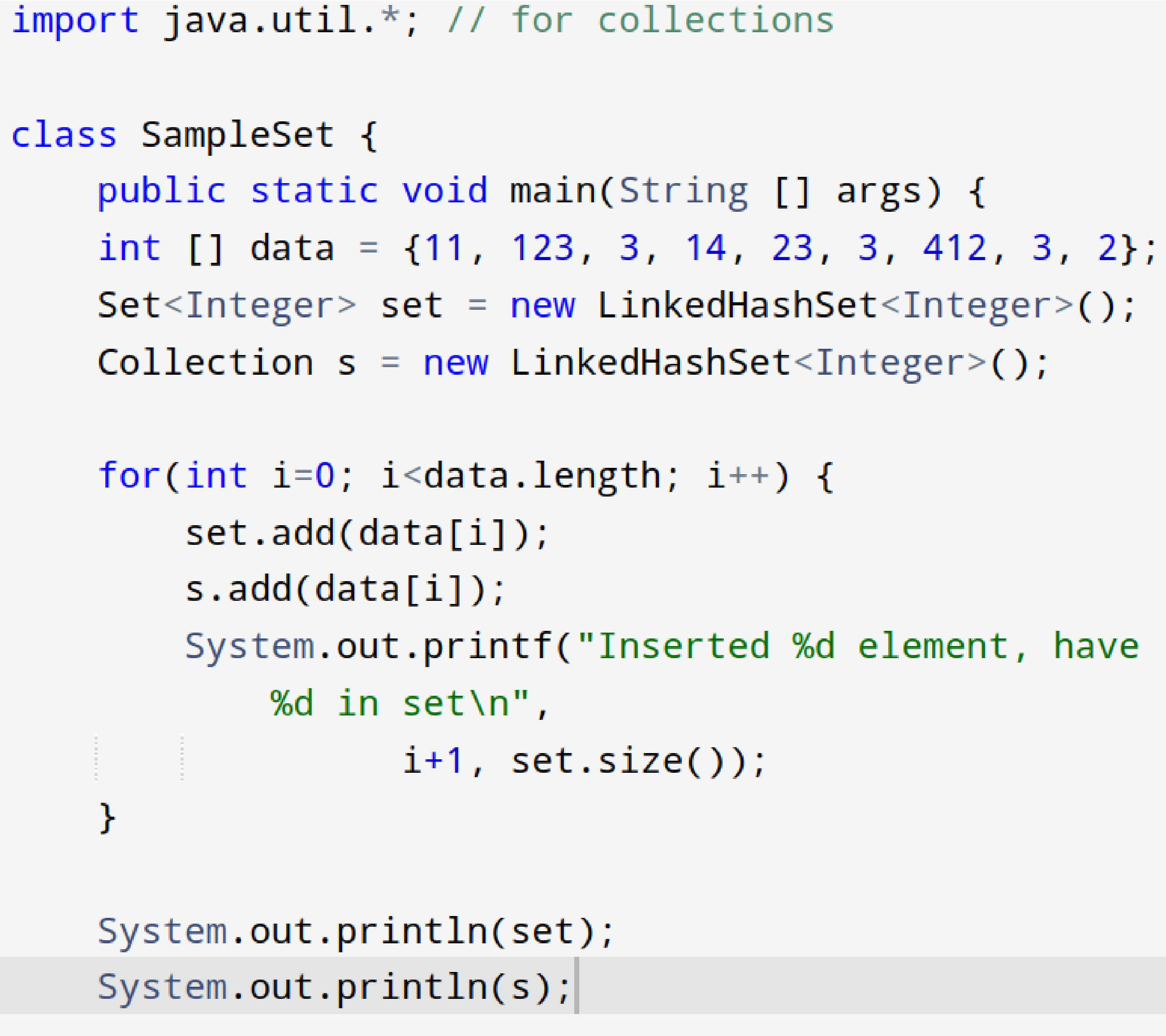


- **public Iterator<E> iterator();**
  - Returns an Iterator that steps through elements of collection
- public Object[] **toArray**()
  - Returns a new array containing all the elements of this collection
- public <T> T[] **toArray**(T[] dest)
  - Returns an array containing all the elements of this collection; uses dest as that array if it can
- Bulk Operations (very powerful!):
  - public boolean **containsAll**(Collection<?> c);
  - public boolean **addAll**(Collection<? extends E> c);
  - public boolean **removeAll**(Collection<?> c);
  - public boolean **retainAll**(Collection<?> c);
  - public void **clear**();

Set

- HashSet
- TreeSet
- LinkedHashSet

Set Example



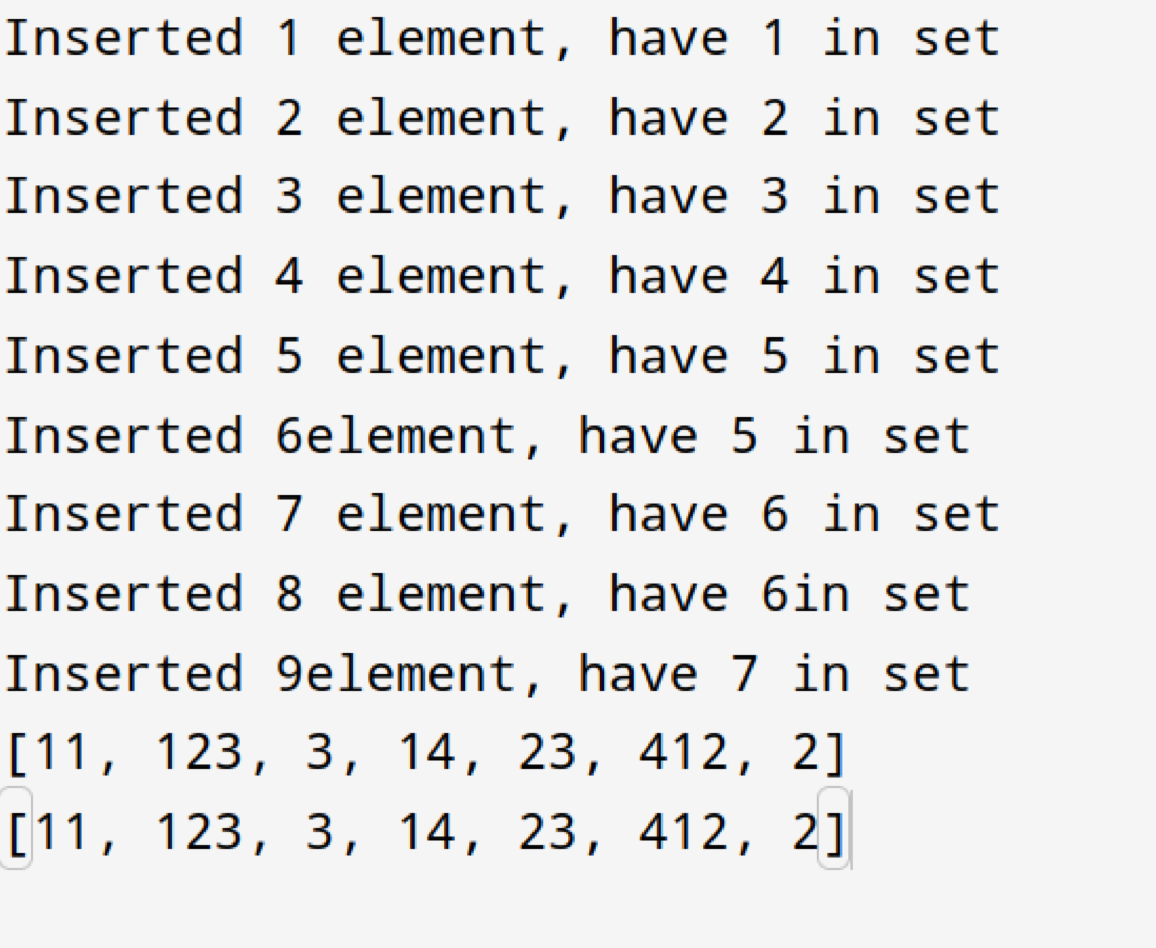
```java
import java.util.*; // for collections

class SampleSet {
    public static void main(String [] args) {
    int [] data = {11, 123, 3, 14, 23, 3, 412, 3, 2};
    Set<Integer> set = new LinkedHashSet<Integer>();
    Collection s = new LinkedHashSet<Integer>();

    for(int i=0; i<data.length; i++) {
        set.add(data[i]);
        s.add(data[i]);
        System.out.printf("Inserted %d element, have
            %d in set\n",
                  i+1, set.size());
    }

    System.out.println(set);
    System.out.println(s);
```

Set Example

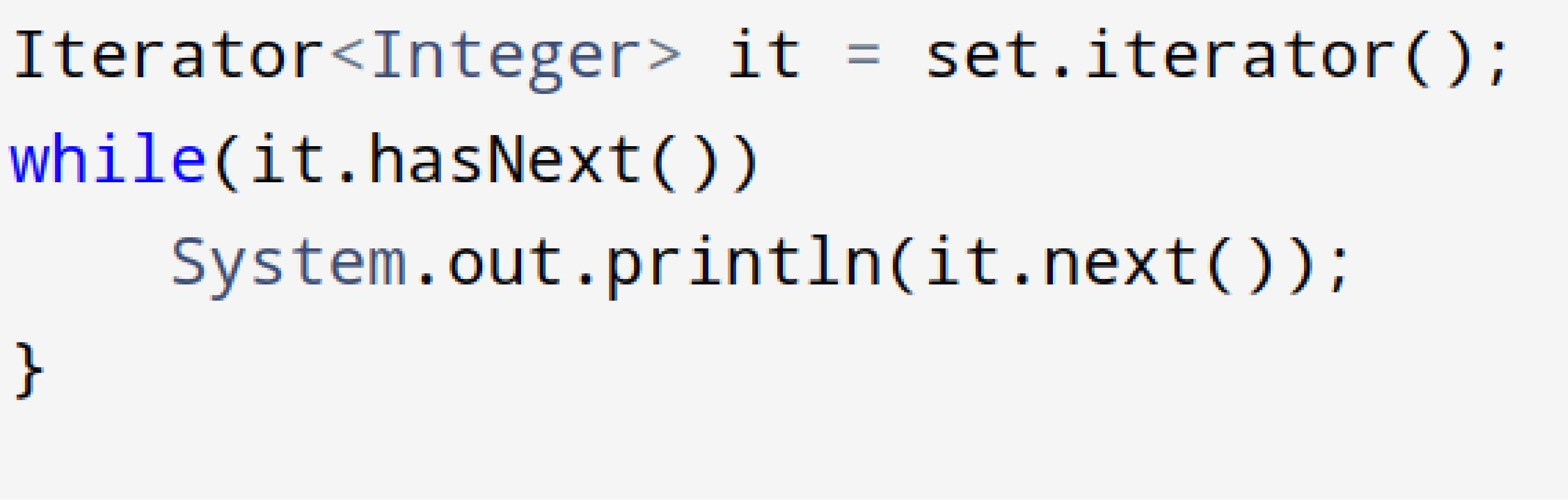


```
Inserted 1 element, have 1 in set
Inserted 2 element, have 2 in set
Inserted 3 element, have 3 in set
Inserted 4 element, have 4 in set
Inserted 5 element, have 5 in set
Inserted 6element, have 5 in set
Inserted 7 element, have 6 in set
Inserted 8 element, have 6in set
Inserted 9element, have 7 in set
[11, 123, 3, 14, 23, 412, 2]
[11, 123, 3, 14, 23, 412, 2]
```

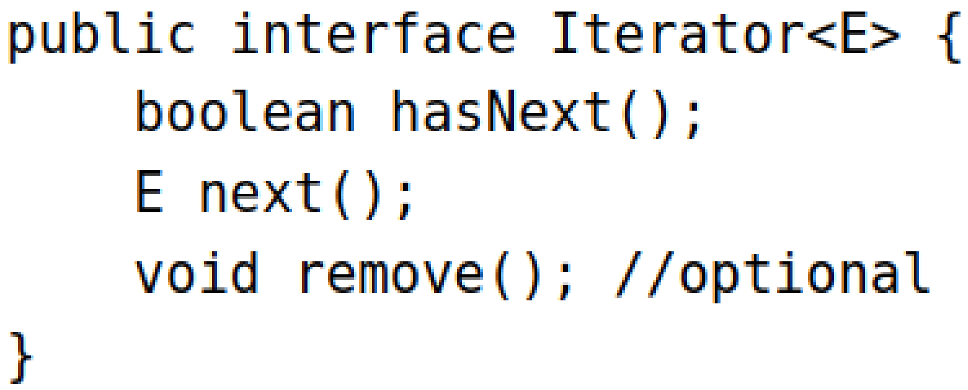
How to traverse collections?

- Iterators
- for-each Construct
- Aggregate Operations

How to traverse collections?

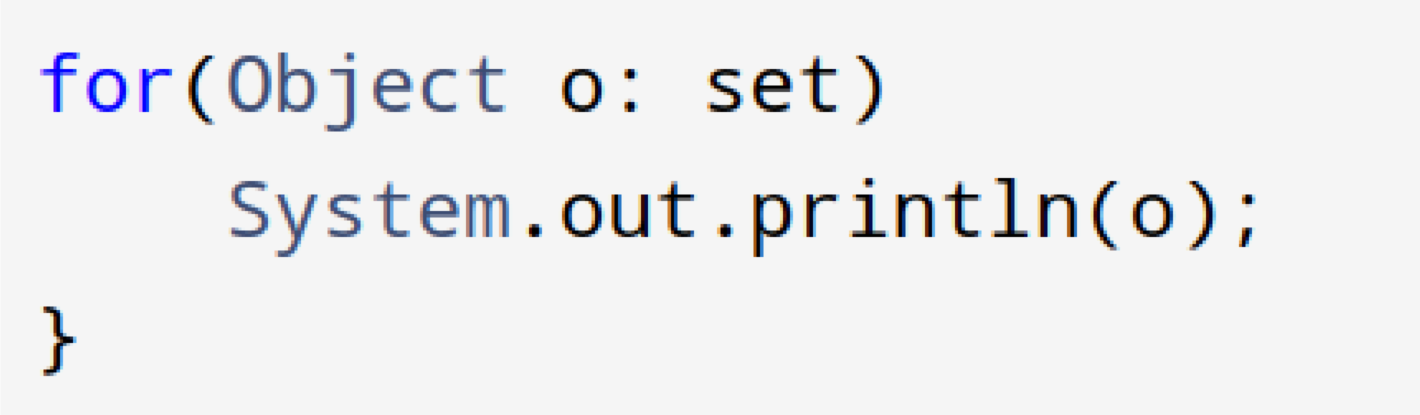


```
Iterator<Integer> it = set.iterator();
while(it.hasNext())
    System.out.println(it.next());
}
```



```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

How to traverse collections?



```
for(Object o: set)
    System.out.println(o);
}
```

Use Iterator instead of the for-each construct when you need to remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.

List

- ArrayList
- LinkedList

List

- Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
- Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
- Range-view — The sublist method performs arbitrary range operations on the list.

java.util.List<E>

- Access to elements via indexes, like arrays
  - public E **get**(int index)
  - Public E **set**(int index, E x)
  - Public void **add**(int index, E x)
  - Public E **remove** (int index)
- Search for elements
  - public int **indexOf**(object x)
  - public int **lastIndexOf**(object x)
- Specialized Iterator, call ListIterator
- Extraction of sublist
  - **subList**(int fromIndex, int toIndex)

List Algorithms

- sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)
- shuffle — randomly permutes the elements in a List.
- reverse — reverses the order of the elements in a List.
- rotate — rotates all the elements in a List by a specified distance.
- swap — swaps the elements at specified positions in a List.
- replaceAll — replaces all occurrences of one specified value with another.
- fill — overwrites every element in a List with the specified value.
- copy — copies the source List into the destination List.
- binarySearch — searches for an element in an ordered List using the binary search algorithm.
- indexOfSubList — returns the index of the first sublist of one List that is equal to another.
- lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.

Map

- HashMap
- TreeMap
- LinkedHashMap

Map

- V **put**(K key, V value)

Adds a key-value pair. Return previous or null.

- V **get**(Object Key)

Looks up the associated value for the given key.

- boolean **containsKey**(Object Key)
- boolean **containsValue**(Object value)