MET CS 622: Generics and Collections

Reza Rawassizadeh

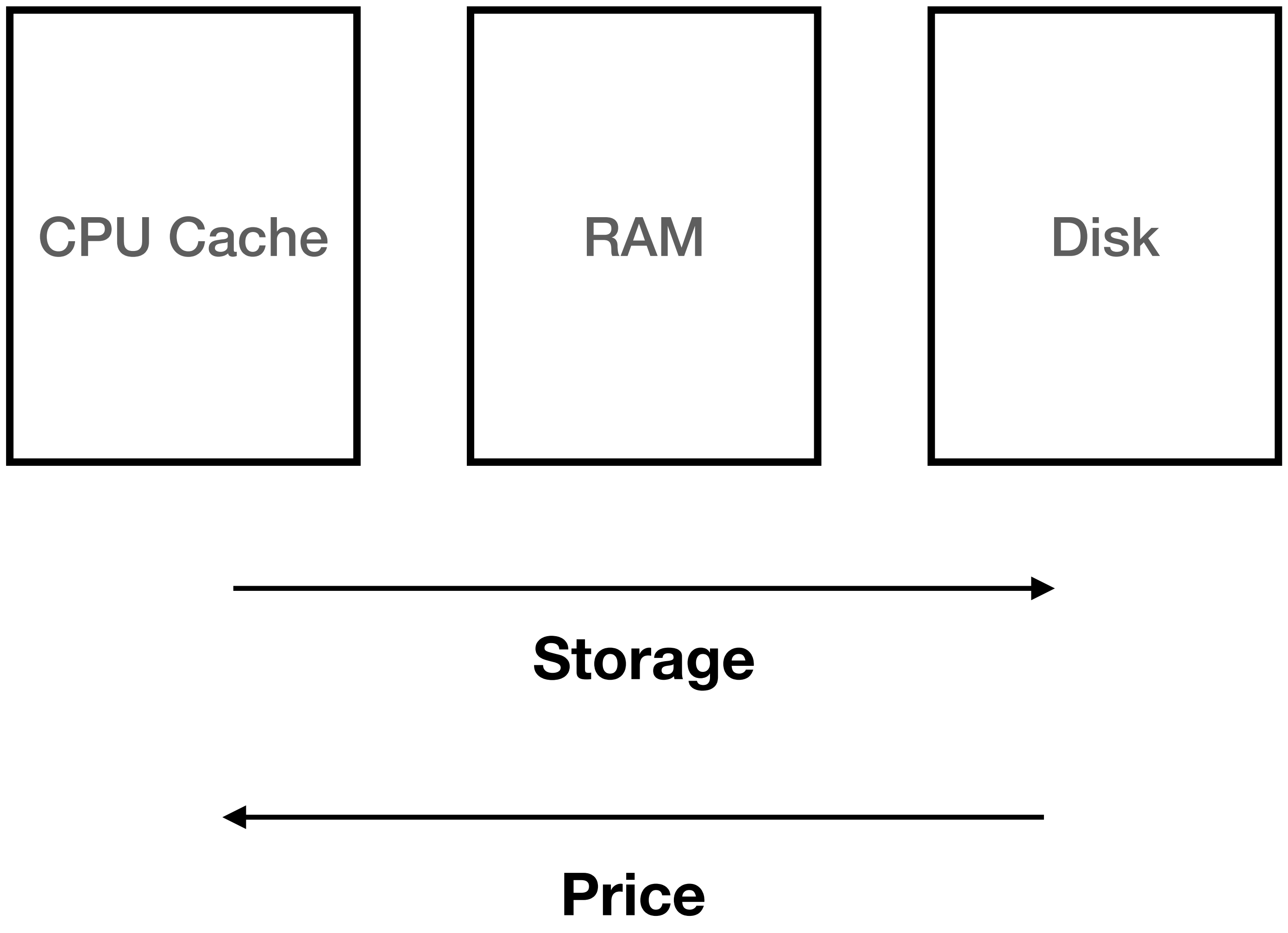
Outline

- Collections
- Generics
- Lambda

Outline

- **Collections**
- Generics
- Lambda

Why Using Collections?



Collections

- A collection is a group of similar objects, gathered together in a Java object, i.e. collections.

  `A = {obj`$_1$`, obj`$_2$`, obj`$_3$`,…}`

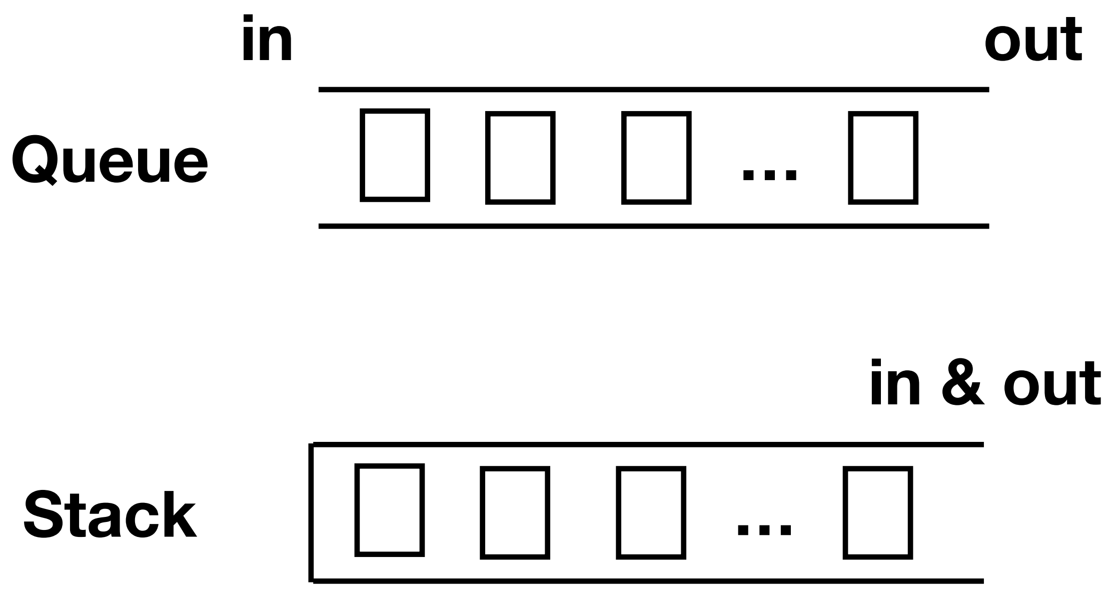
- *`A`* is a collection of *obj* objects.
- Most java collection objects are inside `java.utils` package.
- We can say collection is a database, which keeps data into the heap memory (and not disk)

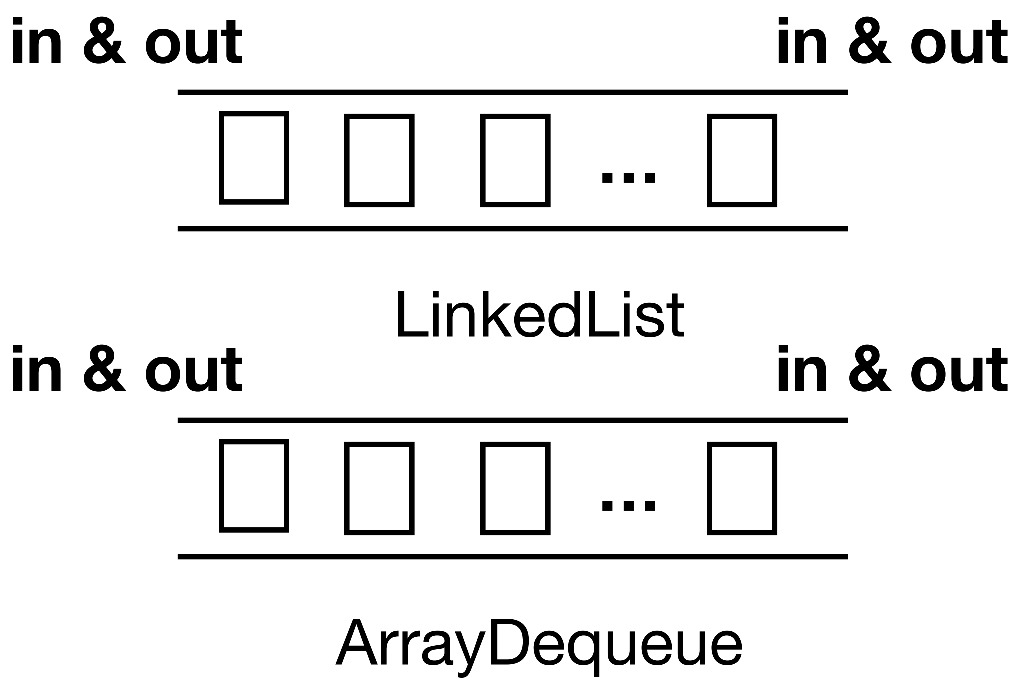
Collection Types

- **List:** It is an ordered collection of java objects and it allows duplicate entries as well. To access to its elements we need to use the index.
- **Set:** A collection, similar to list, but it does not allow duplicate.
- **Queue:** It is a collection of java objects, and its elements are ordered by first-in-first-out. However, we can change its ordering as well, e.g. creating stack (first-in-last-out).
- **Map:** A collection that stores every objects with a key. In other words, Map is a key-value collection.

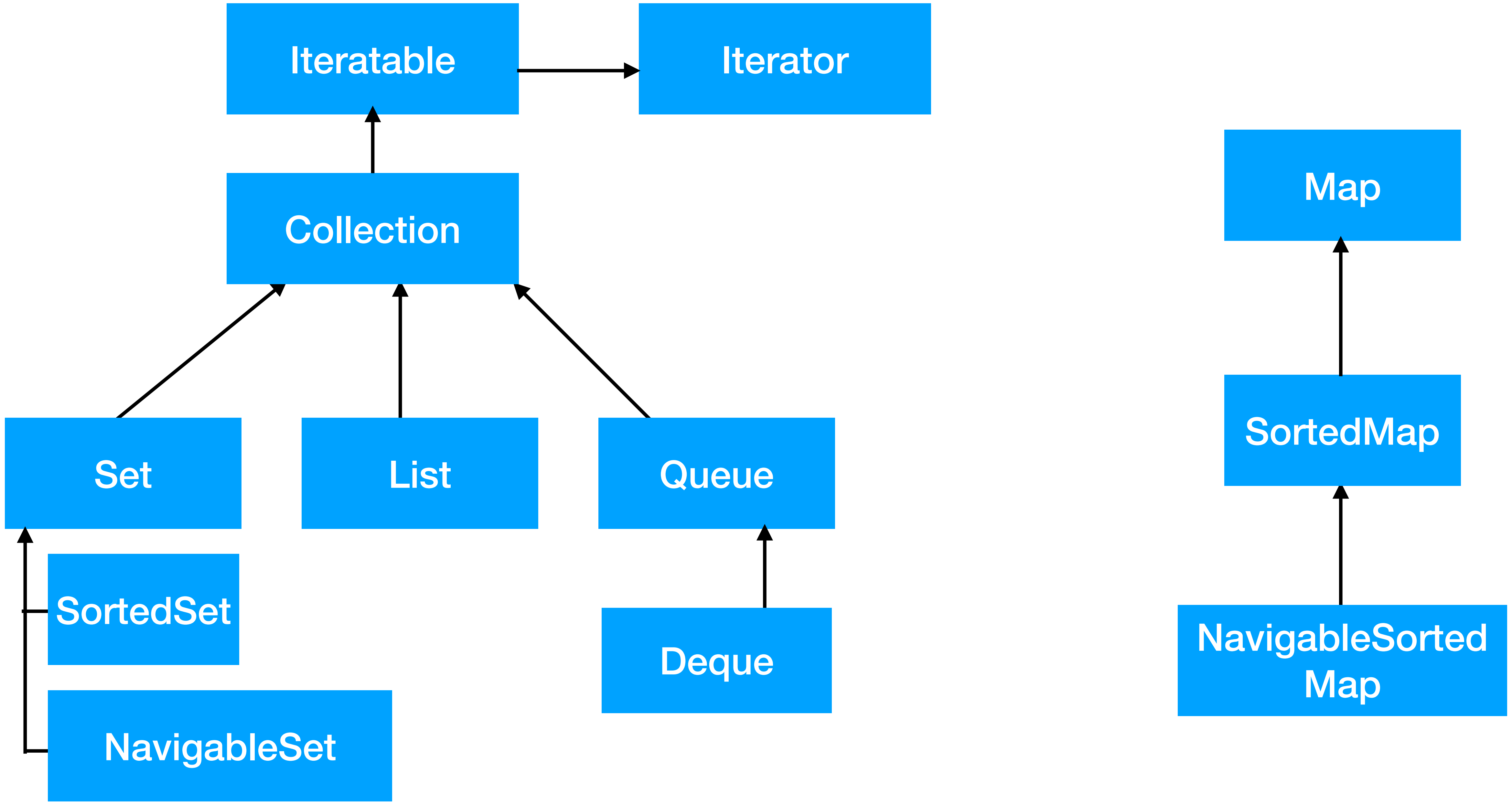
Queue/Stack Concept

- We use Queue when we intend to sort the data before processing, or we intend to add or remove elements in a specific order.
- The Queue is assumed to process data as First-In-First-Out (FIFO). There is another policy queue uses to process the data and it is First-In-Last_out (FILO), like stack.

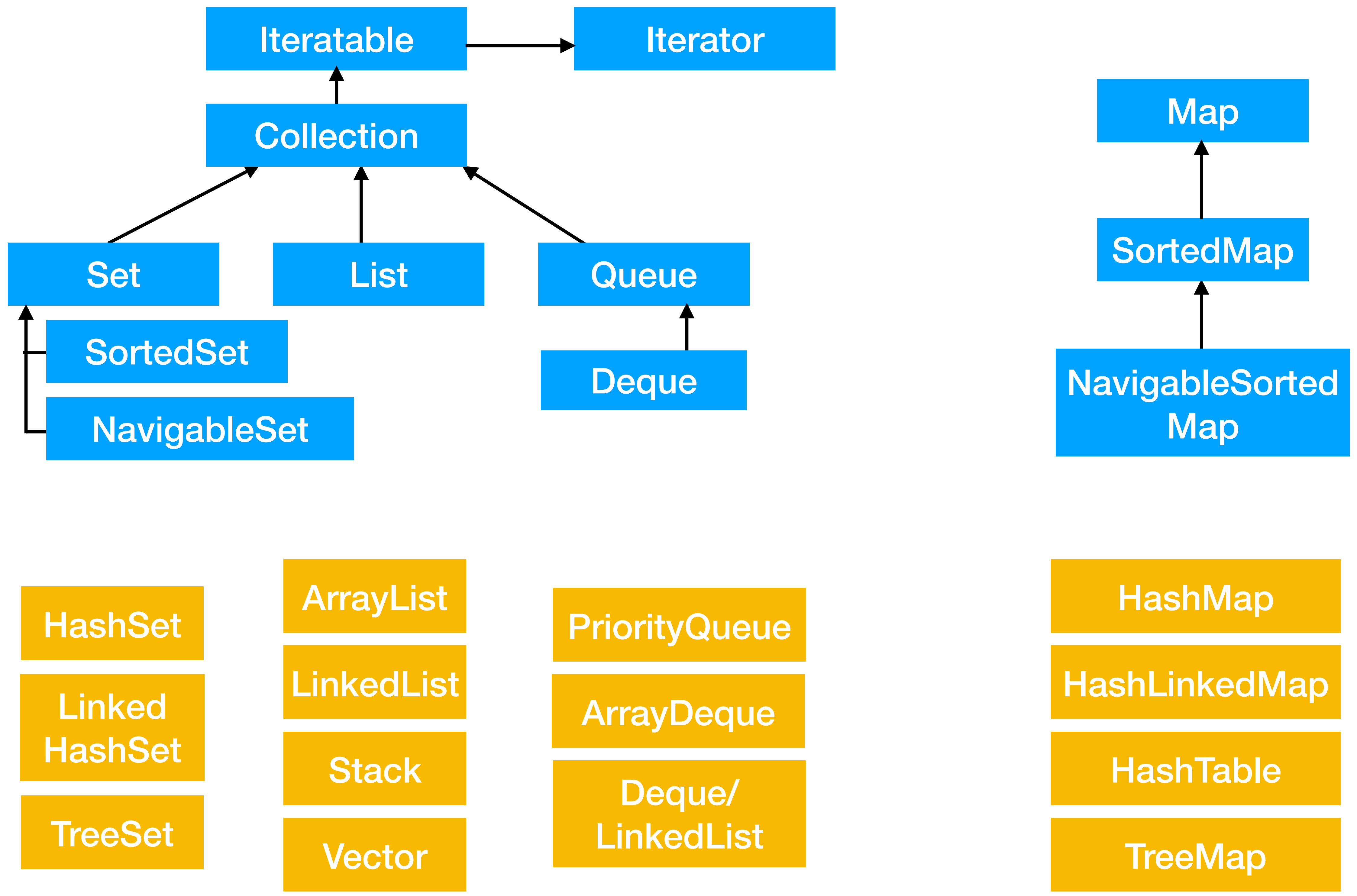




Collection Interfaces



Implementation Classes



Common Collection Methods

- `boolean add(E element)`
- `boolean remove(Object object)`
- `boolean isEmpty()`
- `int size()`
- `void clear()`
- `boolean contains(Object object)`

List Operations

List Examples

```
List objAL = new ArrayList();
List objLL = new LinkedList();
List objV = new Vector();
List objS = new Stack();
```

Outdated, new version is ArrayList

Outdated, new version is ArrayDeque

Positional Access

- `void add(int index,Object O)`
- `boolean addAll(int index, Collection c)`
- `Object remove(int index)`
- `Object get(int index)`
- `Object set(int index, Object new)`

List Operations

Search

- `int indexOf(Object o)`
- `int lastindexOf(Object o)`

Iteration

- Simply a traditional for loop, or `iterator` object.

Sub range

- `List subList(int fromIndex, int toIndex)`

Array List

- ArrayList is a java object which contains a set of other objects. These types of objects are called collections.

```
List<String> mylist = new ArrayList<>();

list.add(“data 1");
list.add(“data 2");
```

- ArrayList is one of the fastest and most efficient collection used in Java. <— preserve it here: 🧠

- ArrayList is a resizable object and it is very useful for situations we do not know the size of the list in advance.

Sorting with Array List

- Array list is a java object which contains a set of other objects. These types of objects are called collections.

```
List<Integer> list = Arrays.asList(1,7,5,8,3);
Collections.sort(list);
System.out.println(list);
System.out.println(Collections.binarySearch(list, 3)); // 1
System.out.println(Collections.binarySearch(list, 4)); // -3
```

LinkedList

- A Linkedlist is both ‘queue’ and ‘list’.
- The main advantages of the Linkedlist is that we can read/add/remove from <u>both beginning or end</u> of the Linkedlist.

```
void add(E element)
void add(int index, E element)
E get(int index)
int indexOf(Object o)
int lastIndexOf(Object o)
void remove(int index)
E set(int index, E e)
```

Iterator

```
Iterator iter = mylist.iterator();
while (iter.hasnext()) {
   System.out.println((String) iter.next())
}
```

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

Experiment in the Class

- Create a java array list with following content:

```
 {'foo', 'bar', 'test1', 'test2', 'METCS622', 'BU', 'Boston
University'}
```

1. Iterate through the list and print its content.

2. Search for the position of 'BU'.

3. Return the index of BU and our course (METCS622).

4. Check if the list contains 'test3'.

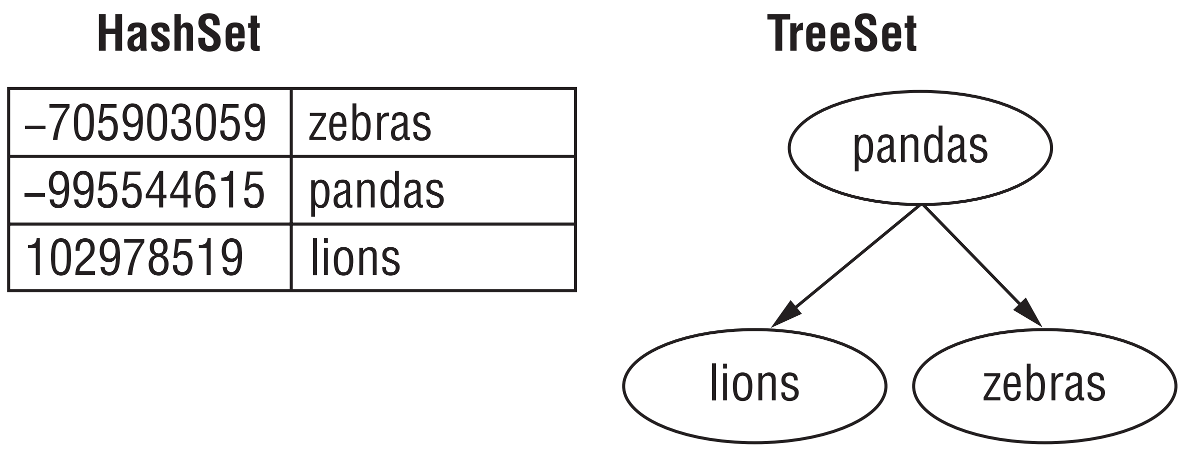
5. Remove 'foo' and 'bar' from the list.

Set

- Set is very useful collection to remove duplicate object entries. Adding elements and checking if an element in the set both have constant time, i.e. O(1).

- A `Hashset` stores its elements in a hash table. This means that it uses the `HashCode()` method to retrieve them more efficiently.

- A `Treeset` stores its elements in a sorted tree structure. Adding and checking computational complexity.



Set Example

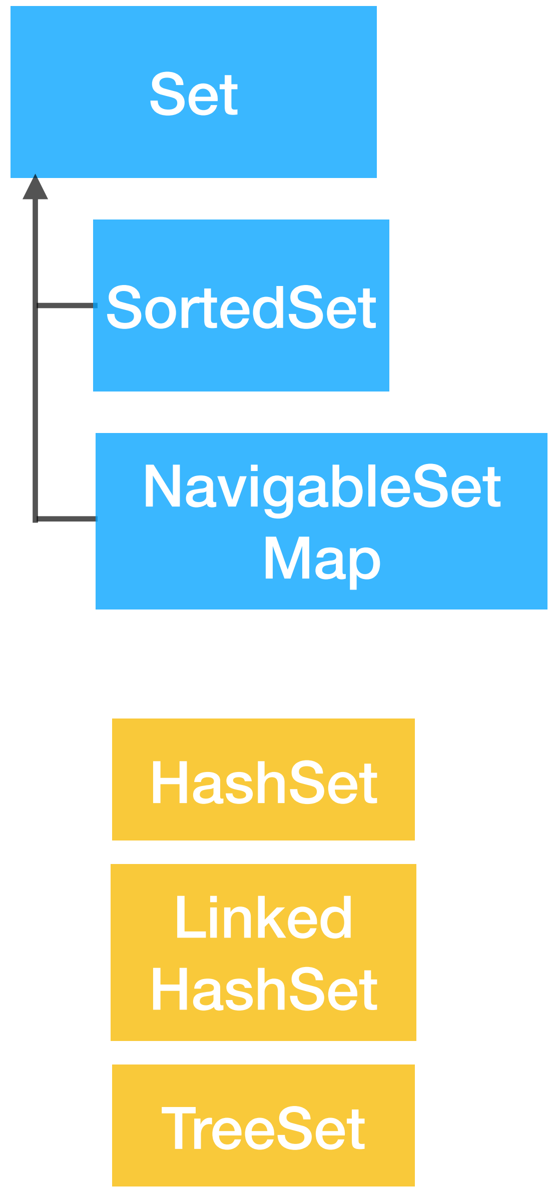
```
Set<Integer> set = new HashSet<>();
boolean b1 = set.add(3);
boolean b2 = set.add(1);
boolean b3 = set.add(3);
boolean b4 = set.add(2);
for (Integer integer: set)
    System.out.print(integer + “,"); //1,2,3
```

`NavigableSet` Interface

It is an interface that provides some useful functionalities.
`TreeSet` implements `NavigableSet`.

- `E lower(E e)`: returns the largest element that is < e
- `E floor(E e)`: returns the largest element that is <= e
- `E ceiling(E e)`: returns the smallest element that is >=e
- `E higher(E e)`: returns the smallest element that is >e

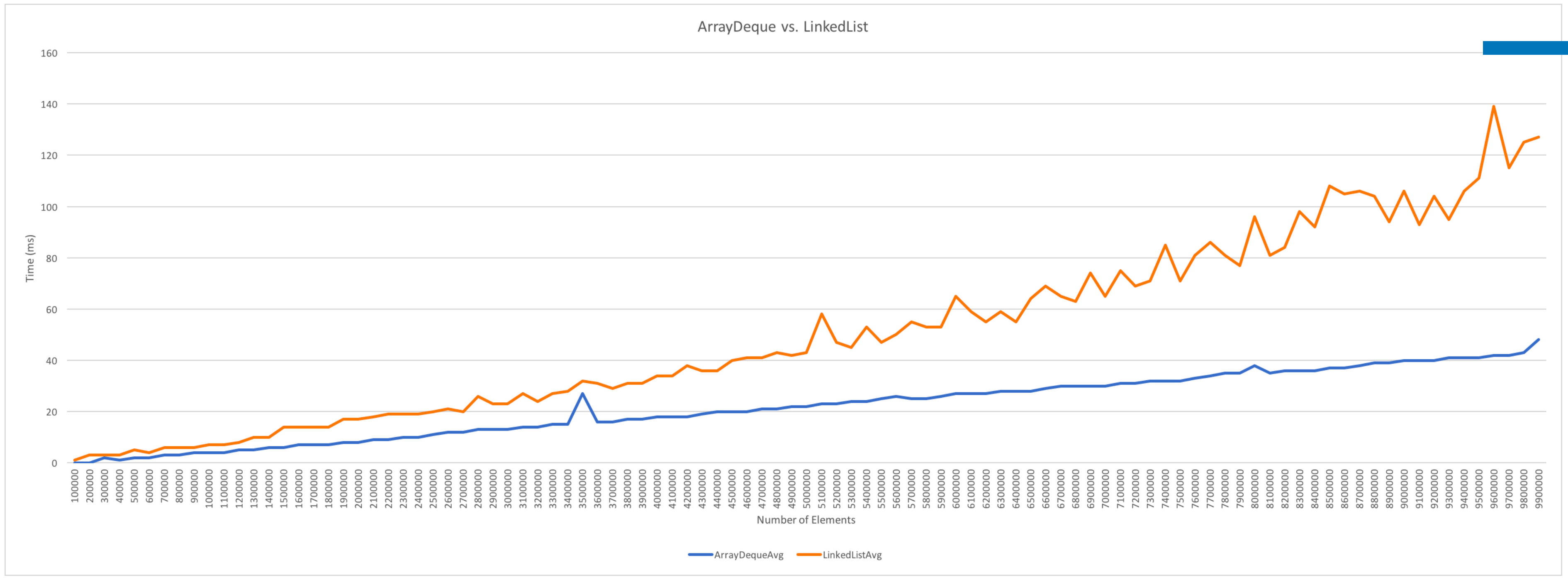
```
NavigableSet<Integer> set = new TreeSet<>();
```



`NavigableSet` Example

```
NavigableSet<Double> tset = new TreeSet<>();
    for (int i=0; i<= 30; i++) {
        tset.add(Double.valueOf(i));
    }
    System.out.println(tset.ceiling(12.5));
    System.out.println(tset.higher(30.0));
    System.out.println(tset.lower(15.1));
    System.out.println(tset.floor(15.1));
```

ArrayDequeue vs LinkedList



▬ LinkedList

▬ ArrayDequeue

- LinkedList is not a good structure for iteration.

- If we need to add or remove in both directions, ArrayDequeue is significantly better.

Source:

http://brianandstuff.com/2016/12/12/java-arraydeque-vs-linkedlist/

ArrayDeque Example

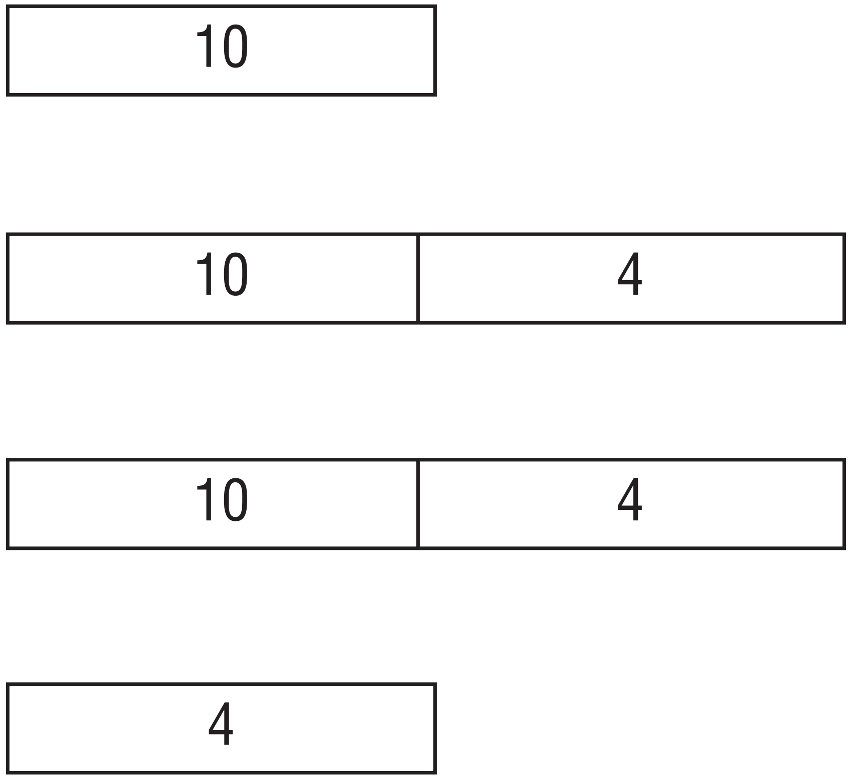
```java
package edu.bu.met622.collections;
import java.util.ArrayDeque;
import java.util.Deque;

public class ArrayDeqExample {
   public static void main(String[] args) {
       // Create an empty array deque with an initial size 4
      Deque<Integer> deque = new ArrayDeque<Integer>(4);

      // Use add() method to add elements into the deque
      deque.add(1);
      deque.add(2);
      deque.add(3);
      deque.add(4);
      deque.add(5);
      deque.add(6);

      // Print all the elements of deque
      for (Integer n : deque) {
         System.out.println(n);
      }
      deque.removeLast();
      System.err.println("-----");
      // Print all the elements of deque
      for (Integer n : deque) {
         System.out.println(n);
      }
   }
}
```

Queue Methods

- `boolean add(E e):` Add an element to the back of the queue, then returns true.
- `E element():` Returns the next element.
- `boolean offer(E e):` Add an element into the end of the queue.
- `E remove:` Remove and returns the next element from the queue.
- `void push(E e):` Adds an element to the beginning of the queue (*only works for double ended queues*).
- `E poll():` Returns and remove the head of the queue.
- `E peek():` Returns the next element in the queue.
- `E pop():` Removes and returns the next element (from stack).

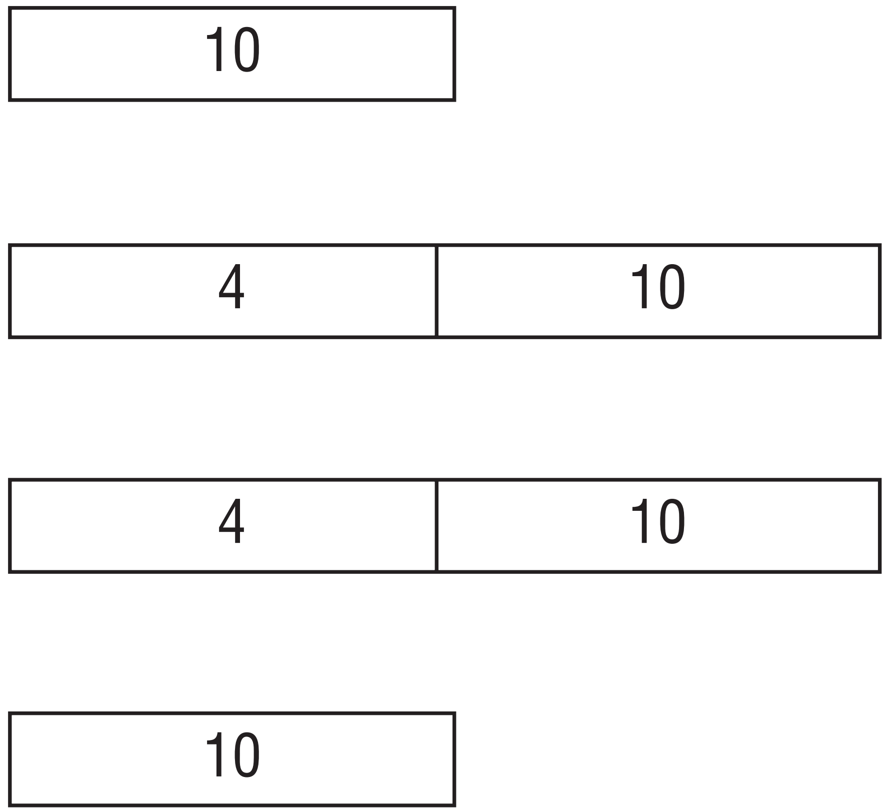
Queue Example

```
queue.offer(10); // true

queue.offer(4); // true

queue.peek(); // 10

queue.poll(); // 10

queue.poll(); // 4

queue.peek(); // null
```



Implementing Stack with Queue

```
queue.push(10);

queue.push(4);

queue.peek(); // 4

queue.poll(); // 4

queue.poll(); // 10

queue.peek(); // null
```



Experiment in the Class

- Create a Queue, add three elements to it, x, y, z and remove the first two elements (x and y will be removed and z will be remained inside the queue).
- Please use push, offer, poll and peek methods.

Map

- When we have a key-value style of data, we use Map.
- There are some databases called key-value database, such as MemecacheDB, BerkelyDB, Redis, Amazon DynamoDB,…

key value

id	data
employee123	Mr. X
employee100	Mrs. Y

Map Implementations

- **`HashMap()`** stores map keys in a hash table. Because of using the hash function, it is one of the fastest collection to store data.
- **`HashMap()`** is fast, because adding and retrieving element from the Hash table is fast and have constant time. Unlike other collections, which we might need to traverse them.
- The disadvantage of storing data with a hash function is that the data will be stored without order (stored based on the hash code)
- **`TreeMap`** is an implementation of Map function, which resolves the issue of storing data, ordered by using a tree style of data structure. In other words, its <u>keys are always sorted</u>. But it has *O(n log n)* complexity.

Map Methods

- `void clear():` removes all data (keys and values) from the map.
- `boolean isEmpty():` checks whether the map is empty or not.
- `int size():` returns the number of entries (key-value pairs) in the map.
- `V get(Object key):` returns the value mapped by key.
- `V put(K key, V value):` adds a new entry into the map.
- `V remove(Object key):` removes the given entry from the map.
- `boolean containsKey(Object key):` checks whether a key existed in the map.
- `boolean containsValue(Object):` checks whether a value existed in the map.
- `Set<K> keySet():` returns all keys in a set object.
- `Collection<V> values():` returns all values in a collection object.

Map Example

```java
package edu.bu.met622.collections;

import java.util.HashMap;
import java.util.Map;

public class MapExample {
 public static void main(String[] args) {
    Map<String, String> map = new HashMap<>();
    map.put("koala", "bamboo");
    map.put("lion", "meat");
    map.put("giraffe", "leaf");
    String food = map.get("koala"); // bamboo
    for (String key: map.keySet())
       System.out.print(key + ","); // koala,giraffe,lion,
 }
}
```

Map Example

```
package edu.bu.met622.collections;

import java.util.HashMap;
import java.util.Map;

public class Ma
 public static                                              {
    Map<String,                                         ap<>();
    map.put("koa
    map.put("lio
    map.put("gi
    String food = map.get("koala"); // bamboo
    for (String key: map.keySet())
       System.out.print(key + ","); // koala,giraffe,lion,
 }
}
```

There is no order here. because we have used a hash map.
If we use a Treemap, then they will be stored in order.

Experiment in the Class

- Create a java HashSet with following content:

```
 {'foo', 'bar', 'test1', 'test2', 'METCS422', 'BU', 'Boston University',
'BU'}
```

1. Iterate through the list and print its content.

2. Search for the position of ‘BU’

3. Check if the HashSet contains ‘test3’.

4. Remove ‘foo’ and ‘bar’ from the list

Comparison Among Collections

Type	Can contain duplicate elements?	Elements ordered?	Has keys and values?	Must add/remove in specific order?
`List`	Yes	Yes (by index)	No	No
`Map`	Yes (for values)	No	Yes	No
`Queue`	Yes	Yes (retrieved in defined order)	No	Yes
`Set`	No	No	No	No

Performance Benchmark

- Create a class call it TestPerformance then experiment ArrayList, and Also creates a Vector (outdated collection object),

- Add ~100,000 data into each collection.

- Search the content of collection and measure their differences.

- You can use this method to measure their execution differences

```java
import java.text.ParseException;
import java.util.concurrent.TimeUnit;

public class Main
{
  public static void main(String[] args) throws ParseException
  {
    Instant start = Instant.now();

    //do something here.

    Instant finish = Instant.now();
```

Outline

- Collections
- **Generics**
- Lambda

Wrapper Classes and Autoboxing

- Each primitive has a corresponding wrapper class and Auto boxing automatically converts the primitive into its wrapper classes (generic type) and vice versa.

- boolean —> Boolean
- byte —> Byte
- int —> Integer
- long —> Long
- double —> Double
- char —> Character
- float —> Float

Diamond Operator

- In earlier Java version, the developer should remember what will be the content of the collection or converts its content manually. Now with Auto boxing, we can convert them to the desired data type.

- After Java 5, a new operator introduced which handles this issue. This operator is used for **type casting**.

- Older version:

```
List sampleArray = new ArrayList();
```

- Later versions:

```
List<String> sampleArray = new ArrayList<String>();
List<String> sampleArray = new ArrayList<>();
```

Generic Types

- What is the output of following code?

```
List<String> names = new ArrayList<String>();
names.add(new StringBuilder("MET622"));
```

Generic Programming

- In the context of generic programming, codes are written based on a vague notion, i.e. **to-be-specified-later,** that are **then instantiated when needed** for **specific types of provided parameters**.

Why Generic Types?

- Assume we are developing a software for shipping company, and our developers write a class to handle shipping packages, boxes or checks.

- The “box” that is used to transfer a vehicle has very few things in common with a “box” (envelope) that is used to transfer (post) a bank check.

- Can we use inheritance?

```
Class Bank-check implements Box
Class Vehicle implements Box
```

Why Generic Types?

- It is not wrong doing so, but it doesn’t seem rational doing that. We have two classes that are different and has very few things in common, except our company will transfer them.

- Generics can help us resolve this issue.

Naming Conventions for Generics

- E for an **element**
- K for a **map key**
- V for a **map value**
- N for a **number**
- T for a **generic data type**
- S, U, V, and so forth, for multiple generic types

Generic Example

```
package edu.bu.met622.genericexample;

public class Sizelimitations <T,U> {

    private T package;
    private U sizeLimit;
    public Sizelimitations(T package, U sizeLimit) {
            this.package = package;
            this.weight = weight;
    }
}
```

```
package edu.bu.met622.genericexample;

public class Startup {
     public static void main(String[] args) {

          Vehicle myToyota = new Vehicle ();
          Integer numPounds = 15000;
          Sizelimitations<Vehicle, Integer> c1 = new Sizelimitations<>(myToyota, numPounds);

          Envelope mycheck = new Envelope ();
          Integer checkWeight = 10;
          Sizelimitations<Envelope, Integer> c2 = new Sizelimitations<>(mycheck, checkWeight);
     }

}
```

What <u>cannot</u> be done with Generics

- Calling a **constructor**, e.g. new `T( )`.
- Calling **instanceof**. This is not allowed because at runtime `List<Integer>` and `List<String>` look the same to Java (because of type erasure).
- Creating an array of static types, e.g.
  ```
  static ArrayList <E> = new ArrayList();
  ```
  or creating a static variable as a generic.
- Use primitive types as generic parameters.

What can not be done with Generics

- Calling a **constructor**, e.g. new T().

- Calling **instanceof**. This is not allowed because at runtime List<Integer> and List<String> look the same to Java (because of type erasure).

- Creating and array of static types, e.g.
static ArrayList <E> = **new** ArrayList();
or creating a static variable as a generic.

- Calling **instanceof**. This is not allowed because at runtime List<Integer> and List<String> look the same to Java (because of type erasure).

- Use primitive types as generic parameters.

What can not be done with Generics

- Calling a **constructor**, e.g. new T().

- Calling **instanceof**. This is not allowed because at runtime List<Integer> and List<String> look the same to Java (because of type erasure).

- Creating and array of static types, e.g.

```
static ArrayList <E> = new ArrayList();
```

or creating a static variable as a generic.

- Calling **instanceof**. This is not allowed because at runtime List<Integer> and List<S pe erasure).

- Use primiti

Since the static variable is shared between the objects, the compiler cannot determine which type to use.

Generic Methods

```
public static <T> Sizelimitation<T> ship(T t) {
   System.out.println("Preparing " + t);
   return new Sizelimitation<T>();
}
```

- Generic method are useful for static methods. Because they are not part of the instance of the class and thus <u>they use the a single slot of memory which has been assigned for the static class</u>.

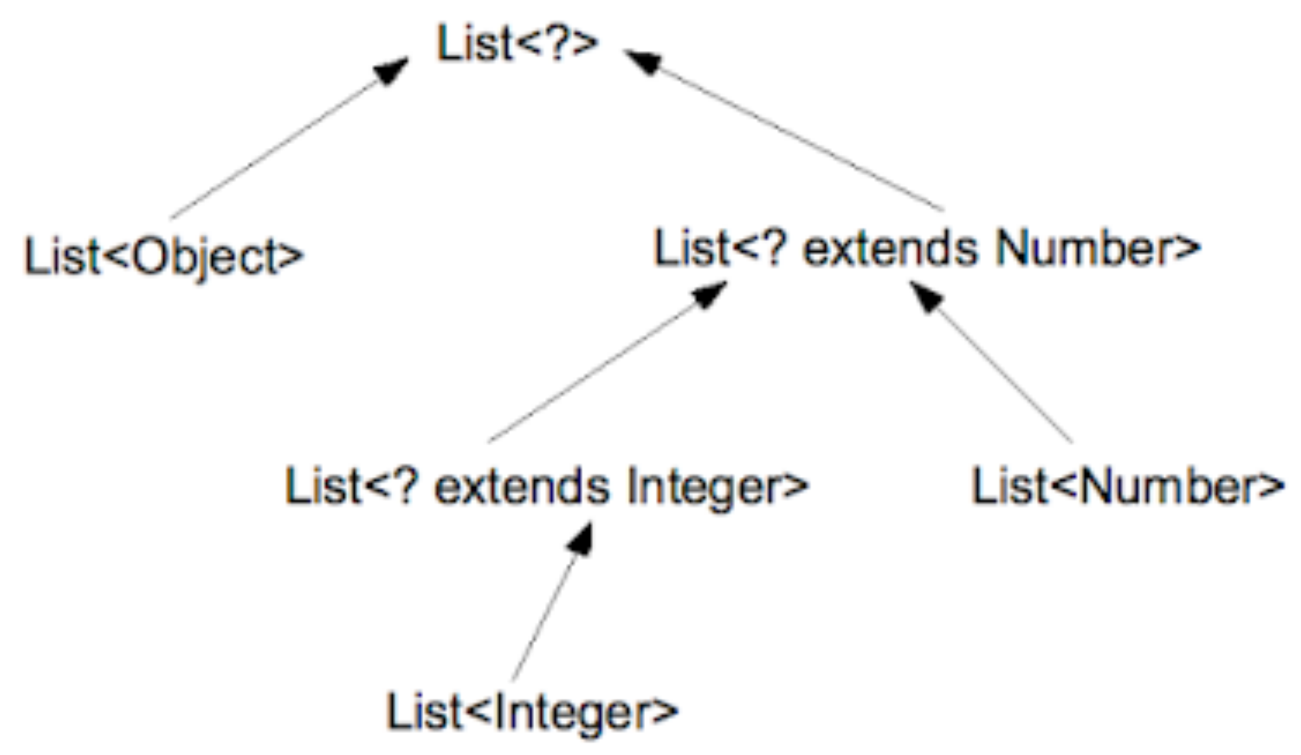
Bounds

- Generics were be treated as **`Objects`**, and thus do not have many methods available. This is a big limitations while using Generics.
- Bound type '?', which is a generic type can be used to specify bound for the generic.

```
List<?> l =new ArrayList<String>();

List<? extends Exception> l = new ArrayList<RuntimeException>();

List<? super Exception> l = new ArrayList<Object>();
```

'?' = **wildcard operator**

Java List Generic



Taxonomy of the generic *List* types

source: https://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html

Unbounded Wildcards

```
public static void printList(List<Object> list) {
    for (Object x: list)
    System.out.println(x);
}
public static void main(String[] args) {
    List<String> keywords = new ArrayList<>();
    keywords.add("java");
    printList(keywords);
}
```

doesn't compile

```
public static void printList(List<?> list) {
    for (Object x: list)
    System.out.println(x);
}
public static void main(String[] args) {
  List<String> keywords = new ArrayList<>();
  keywords.add("java");
  printList(keywords);
}
```

compile

More Examples

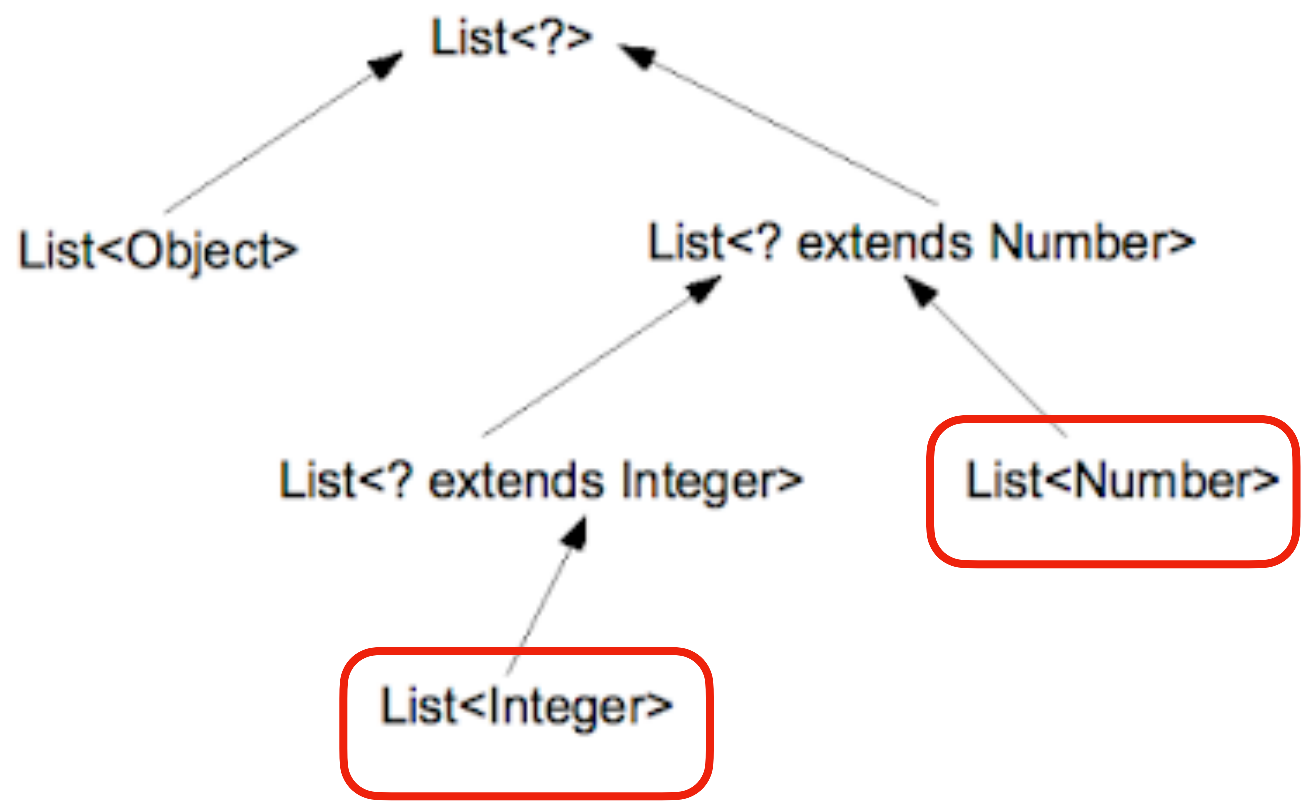
Does not compile

```
ArrayList<Number> list = new ArrayList<Integer>();
```

Compiles

```
List<? extends Number> list = new ArrayList<Integer>();
```

Java List Generic



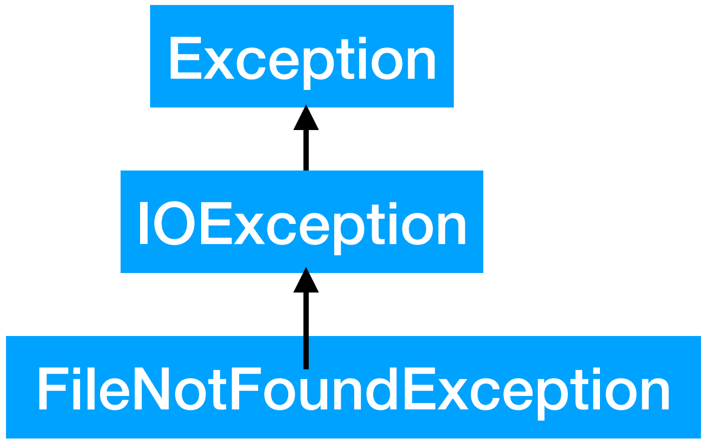
Taxonomy of the generic *List* types

source: https://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html

More Examples
(Superclass & subclass conversion)

- When you have subclasses and superclasses, lower bounds can get tricky:

```
List<? super IOException> exceptions = new ArrayList<Exception>();
exceptions.add(new Exception()); // DOES NOT COMPILE
exceptions.add(new IOException());
exceptions.add(new FileNotFoundException());
```



Let’s Experiment Together

Example

- `class A {}`
- `class B extends A { }`
- `class C extends B { }`

Which one does not compile?

```
1. List<?> list1 = new ArrayList<A>();

2. List<? extends A> list2 = new ArrayList<A>();

3. List<? super A> list3 = new ArrayList<A>();

4. List<? extends B> list4 = new ArrayList<A>();

5. List<? super B> list5 = new ArrayList<A>();

6. List<?> list6 = new ArrayList<? extends A>();
```

Which one does not compile?

```
1. List<?> list1 = new ArrayList<A>();

2. List<? extends A> list2 = new ArrayList<A>();

3. List<? super A> list3 = new ArrayList<A>();

4. List<? extends B> list4 = new ArrayList<A>();

5. List<? super B> list5 = new ArrayList<A>();

6. List<?> list6 = new ArrayList<? extends A>();
```

Which one does not compile?

```
1. 
2.                                                  ();
3.                                                  ;
4.                                                  ();
```

4. has an upper-bounded wildcard that allows ArrayList<B> or ArrayList<C> to be referenced. Since you have ArrayList<A> that is trying to be referenced, the code does not compile.

6. allows a reference to any generic type. But, while instantiating the ArrayList, the JVM needs to know what is that type.
It is not useful anyway, because we can’t add any elements to that ArrayList.

```
5. List<? super B> list5 = new ArrayList<A>();

6. List<?> list6 = new ArrayList<? extends A>();
```

Outline

- Collections
- Generics
- **Lambda**

Lambda

- A lambda expression is a block of code that <u>does not need to belong to any class</u>. It is provided by Java to support functional programming (not object oriented programming)

- It implements only **one abstract class.**

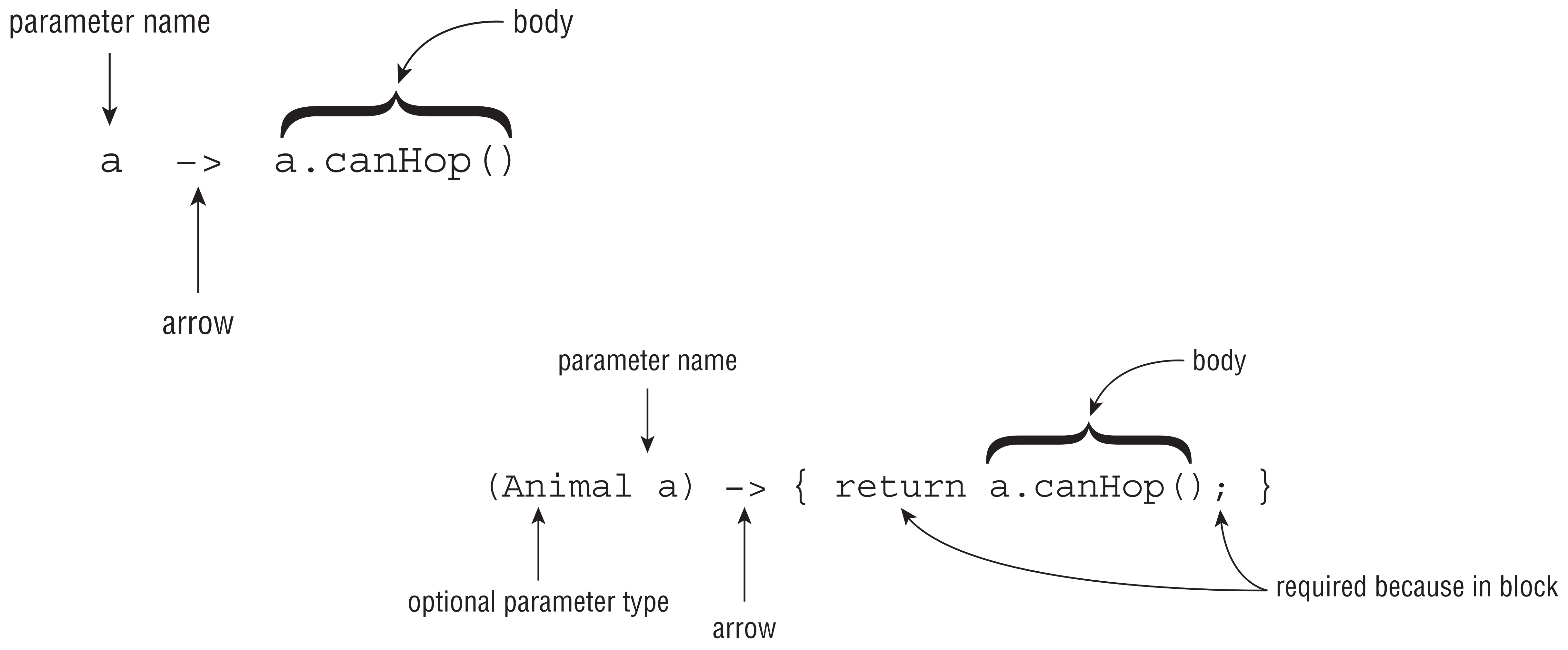
- It has been introduced in Java 8.

- Lambda expression divides the lambda expression into two parts:

```
(n) -> System.out.println(n);
```

*The left-side, specify **required parameter** of the implemented method, it can be empty if no parameter is required.*
*On the right-side of the lambda expression, we need to specify **the action of the lambda expression** (body of the implemented method).*

Lambda Syntax



```
a  ->  a.canHop()
```

```
(Animal a) -> { return a.canHop(); }
```

Source: Oracle Certified Professional Java SE 8 Programmer II

Functional Interface

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

The @symbol denotes Annotation.
Java annotation adds a special attribute to the variable, method, class or interface.

```
@FunctionalInterface

public interface Sprint {

    public void sprint(Animal animal);

}
```

It is highly recommended to mark the Functional Interface with annotation.
It is not mandatory, but using this annotation enables other developers to know exactly on which interface they can apply Lambda.

Functional Interface

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

The @symbol
Java annotatio   d, class or
interface.

```
    @Functiona

    public int

        public

    }
```

From **Java** 8 onwards, lambda expressions can be used to represent the instance of a **functional interface. ...** Runnable, ActionListener, Comparable are some of the examples of **functional** interfaces.

It is highly recommended to mark the FunctionalInterface with annotation. We can give up, but using this annotation enables other developers to know on which interface they can apply Lambda.

Functional Interface and Lambda

- The lambda expression is an **anonymous method** (a method without a name) that is used to implement the abstract method of the functional interface.

- The lambda expression introduces a new operator in Java called the `ARROW ( -> )` operator.
  `arguments -> body of lambda expression`

- A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

Examples

```
@FunctionalInterface
public interface Sprint {
     public void sprint(Animal animal);
}
```

Which one of the followings are FunctionalInterface?

```
//1
public interface Run extends Sprint {}

//2
public interface SprintFaster extends Sprint {
     public void sprint(Animal animal);
}

//3
public interface Skip extends Sprint {
   public default int getHopCount(Kangaroo kangaroo) {
      return 10;
   }
   public static void skip(int speed) {}
}
```

Examples

```
@FunctionalInterface
public interface Sprint {
```

All three are valid functional interfaces!
-The first interface, Run, defines no new methods, but since it extends Sprint, which defines a single abstract method, it is also a functional interface.

-The second interface, SprintFaster, extends Sprint and defines an abstract method, but this is an override of the parent sprint( ) method; therefore, the resulting interface has only one abstract method, and it is considered a functional interface.

-The third interface, extends Sprint and defines a static method and a default method, each with an implementation. Since **neither of these methods are abstract**, the **resulting interface has only one abstract method** and is a functional interface.
read this about default method: https://www.geeksforgeeks.org/default-methods-java/

```
    public static void skip(int speed) {}
}
```

Simple Lambda Example

```java
package edu.bu.met622.lambdaexample;
import java.util.ArrayList;

/**
 * adapted from here: https://www.geeksforgeeks.org/lambda-expressions-java-8/
 * http://tutorials.jenkov.com/java/lambda-expressions.html
 *
 */
public class SecondLambdaExample {
    // A Java program to demonstrate simple lambda expressions
        public static void main(String args[])
        {
            // Creating an ArrayList with elements
            // {1, 2, 3, 4}
            ArrayList<Integer> arrL = new ArrayList<Integer>();
            arrL.add(1);
            arrL.add(2);
            arrL.add(3);
            arrL.add(4);
            arrL.add(5);
            arrL.add(6);
            arrL.add(7);

            // empty lambda is not working in recent version of Java. It should be an interface
            //() -> System.out.println("Zero parameter lambda");

            // Using lambda expression to print all elements  of arrL
            arrL.forEach( n -> System.out.println("just print them:"+n) );

            // Using lambda expression to print even elements of arrL
            arrL.forEach( n -> { if (n%2 == 0) System.out.println("n is:"+n);} );
        }

}
```

Simple Lambda Example

```java
package edu.bu.met622.lambdaexample;
import java.util.ArrayList;

/**
 * adapted from here: https://www.geeksforgeeks.org/lambda-expressions-java-8/
 * http://tutorials.jenkov.com/java/lambda-expressions.html
 *
 */
public class SecondLambdaExample {
    // A Java program to demonstrate simple lambda expressions
        public static void main(String args[])
        {
            // Creating an ArrayList with elements
            // {1, 2, 3, 4}
            ArrayList<Integer> arrL = new ArrayList<Integer>();
            arrL.add(1);
            arrL.add(2);
            arrL
            arrL
            arrL
            arrL
            arrL
```

we must always have a functional interface on the left side of lambda operator

```java
            // empty lambda is not working in recent version of Java. It should be an interface
            //() -> System.out.println("Zero parameter lambda");

            // Using lambda expression to print all elements  of arrL
            arrL.forEach(n -> System.out.println("just print them:"+n));

            // Using lambda expression to print even elements of arrL
            arrL.forEach(n -> { if (n%2 == 0) System.out.println("n is:"+n); });
        }

}
```