Mise à niveau en Java

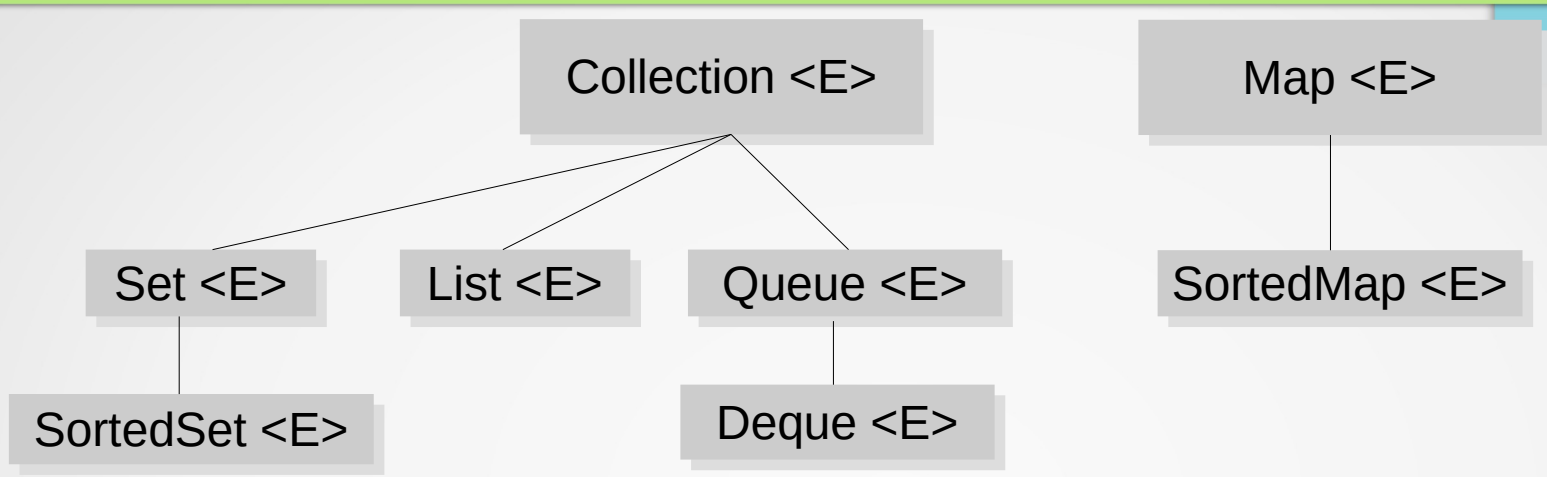
Generics and Collections
M1

Amin Farvardin
m.Farvardin@hotmail.com

Collections

- Lists, sets, stacks, queues are objects that group several elements into a single entity.
  - in common:
    - same questions: do they contain elements? how much?
    - same operations: we can add or remove an element to the structure, we can empty the structure. One can also browse the elements contained in the structure.
  - different implementations
- Q: How can we manipulate all of these structures?
- R: use a hierarchy of interfaces

Hierarchy of interfaces



Collection: Basic methods to browse, add, remove elements.

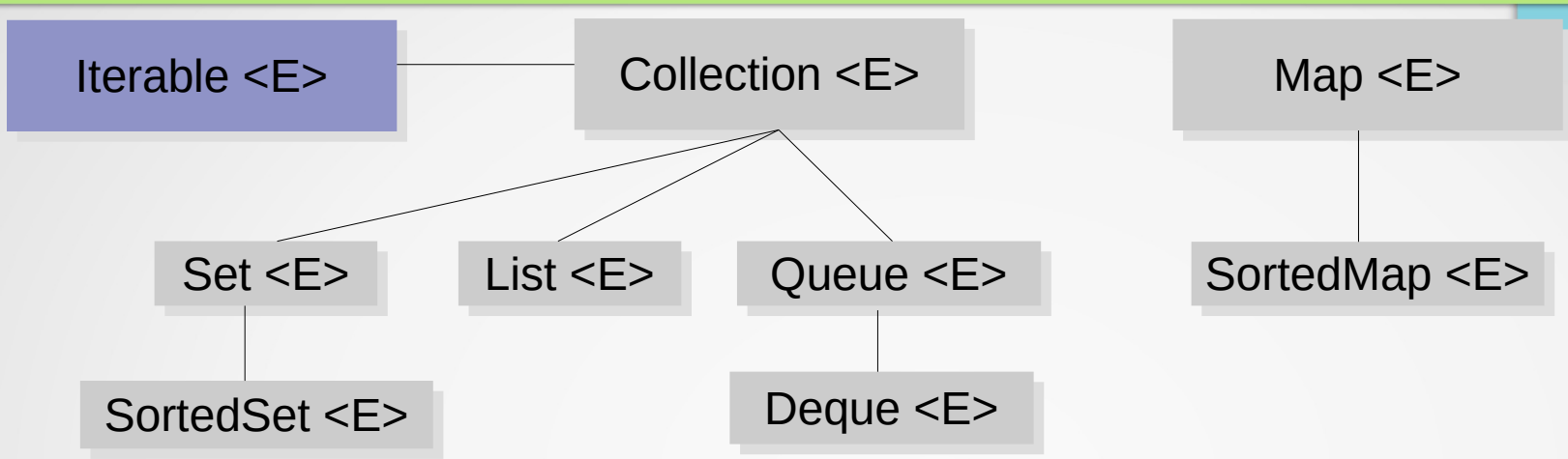
Set: This interface represents a set, and therefore, this type of collection does not admit any duplicate.

List: This interface represents a sequence of elements: the order of adding or removing elements is important (duplicates possible)

Queue: There is the leading element and there are the following elements.
The order of adding or removing elements is important (duplicates possible)

Deque (Double ended queue): This interface looks like queues, but the important elements are the header and queue elements.

Hierarchy of interfaces



Map: This interface represents a binary relation (surjective): each element is associated with a cell and each key is unique (but we can have duplicates for the elements).

SortedSet: is the ordered version of a set.

SortedMap: is the ordered version of a binary relation where the keys are ordered.

These interfaces are generic, i.e. we can give them a parameter to indicate that we have a collection of `Integer`, `String`, `objects(animals)`, etc.

Note: We can use a "`for each`" loop on any object implementing the iterable interface.

Browse in collection: First solution

- By using a generic, the compiler understands the type of elements in the collection.
- Solution: we have a collection which contains objects of type E (e.g., Integer, String, Object, etc.).
- We will access each element of the `Collection` using the `for` loop keyword,
- Each element will be stored in a variable X of type E.
- **For example:**

```
List<String> names = new ArrayList<String>();
names.add(“Bob”);
names.add(“Alice”);
for(String n: names)
    System.out.println(n);
```

Browse in collection: Second solution

- Use of an object dedicated to browsing elements in a collection: an object that implements the Iterator interface.
- Obtain: call to the iterator() method (Iterable interface)

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

  - `hasNext()`: returns a boolean indicating if there are any elements left for visitor,
  - `Next()`: gives access to the next element,
  - `remove()` removes the element from the collection.

Usage:

- Remove an element during iteration.
- Browses several collections in parallel.

Browse in collection: Second solution

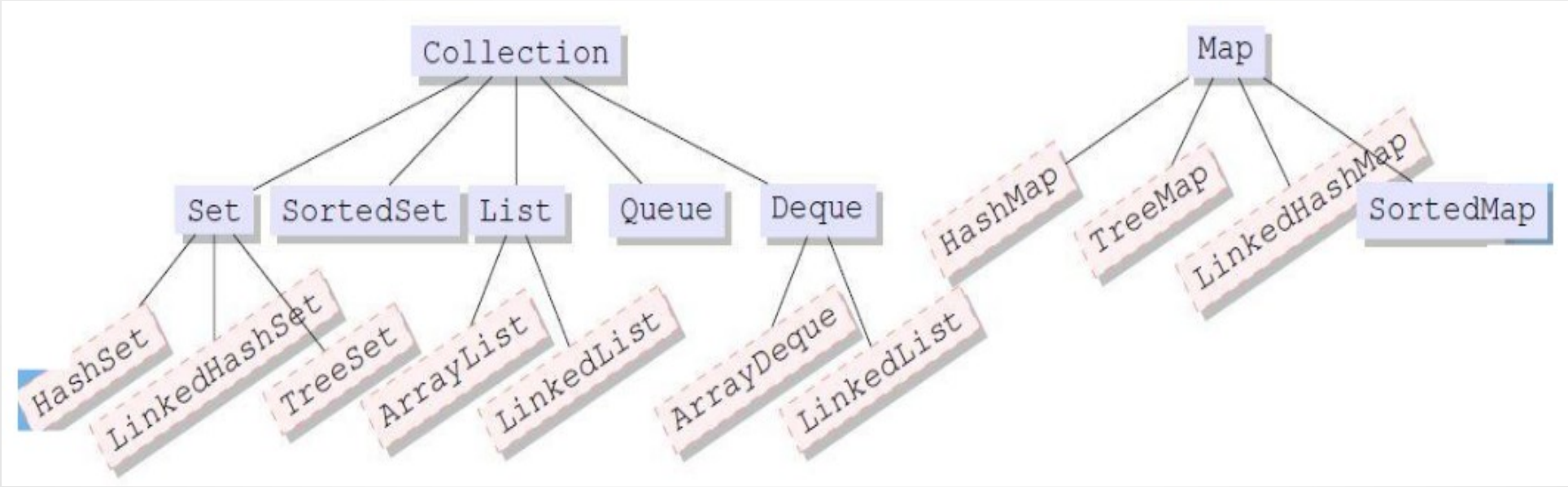
```
List<Gaulois> villager = new ArrayList<Gaulois>();
villager.add(new Gaulois("Asterix"));
villager.add(new Gaulois("Cetaumatix"));
villager.add(new Gaulois("Agecanonix"));
villager.add(new Gaulois("Ordralfabetix"));
villager.add(new Gaulois("Bonemine"));

Iterator<Gaulois> it = villager.iterator();
while(it.hasNext()) {
    Gaulois g = it.next();
    if(g.getName().equals("Asterix"))
        it.remove();
    else
        System.out.println(g.presentation());
}
```

Output:

```
My name is Cetaumatix. I am a Gaulois.
My name is Agecanonix. I am a Gaulois.
My name is Ordralfabetix. I am a Gaulois.
My name is Bonemine. I am a Gaulois.
```

Implementations

For each of the interfaces, there are several implementations.



Map

- a `Map` represents a binary relation: each element of a `Map` is a peer between a key and a value.
- In a `Map`, each key is unique, but we can have duplicates for the values.
- Attention, `Map` is not a sub-interface of Iterable, so we cannot browse a `Map` with a `For each` loop!
- We can obtain the set of keys, the set of values, and the set of pairs (key, value) using the following methods:
  - Set<K>: keySet()
  - Set<Map.Entry<K,V>>: entrySet()
  - Collection<K>: values()
- `Map.Entry` designates an `Entry` class which is internal to the `Map` class.

You can create classes inside classes, but I won't talk more about that today.

Example route of a Map

```
Map<Integer, String> nums = new HashMap<Integer, String>();
// adding elements to the map called nums
nums.put(1, "one");
nums.put(2, "two");
nums.put(3, "three");
nums.put(7, "seven");

for(Map.Entry<Integer, String> pair: nums.entrySet()) {
    // Converting to Map.Entry
    // so that we can get key and value separately
    Integer i = pair.getKey();
    String s = pair.getValue();
    System.out.println(i + " -> " + s);
}
```

This example starts with a *Map* so that its key is a number and its value is a string.

Collection → List → ***ArrayList***

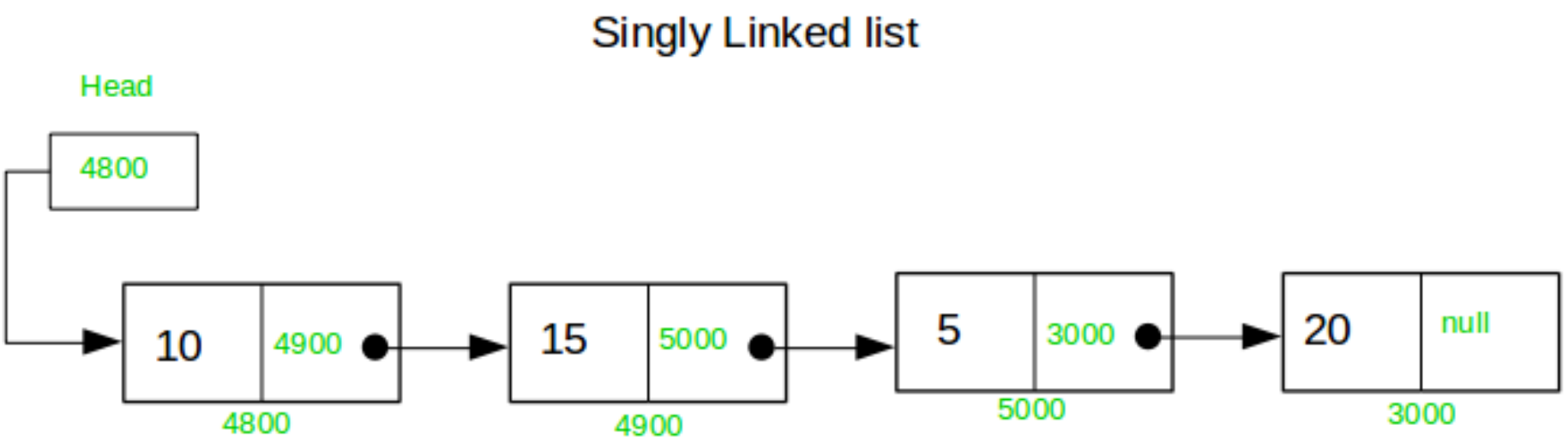
- *Array* (recap):
  - Ex: String[5] cars = {"Volvo", "BMW", "Ford", "Mazda"};
  - The size of an array cannot be modified. The above example reserved five continuous memory spaces for keeping car names in string type.
- ***ArrayList*** :
  - The **`ArrayList`** class is a resizable array, which can be found in the **`java.util`** package.
  - Elements can be added and removed from an **`ArrayList`** whenever you want.

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
cars.add("Volvo");    // Add an Item
cars.add("BMW");
cars.get(0);          // Access an Item
cars.set(0, "Mazda"); // Change an Item
cars.remove(0);       // Remove an Item
cars.size();          // Size of the ArrayList
cars.clean();         // Clean the ArrayList
```

Collection → List → ***LinkedList***

- The `LinkedList` class is almost identical to the `ArrayList`.
- The `LinkedList` class has all of the same methods as the `ArrayList` class because they both implement the `List` interface.
- **But** `LinkedList` is built very differently.
  - The `ArrayList` class has a regular array inside it.
  - The `LinkedList` stores its items in "*containers*." The list has a link to the first container and each container has a link to the next container in the list.

LinkedList representation




```
class Node {
  // node variables
  int data;
  Node next;

  public Node(int data) {
    this.data = data;
    this.next = null;
  }
}
```

LinkedList implementation

```
class SinglyLList {
  Node head; // create reference Node

  void InsertAtStart(int data) {
    // create a node
    Node new_node = new Node(data);

    new_node.next = head;
    head = new_node;
  }

  void InsertAtLast(int data) {
    Node new_node = new Node(data);
    if (head == null) {
       head = new_node;
       return;
    }

    new_node.next = null;
    Node last = head;
    while (last.next != null) {
      last = last.next;
    }
    last.next = new_node;
 }
}
```

```
class Node {
  // node variables
  int data;
  Node next;

  public Node(int data) {
    this.data = data;
    this.next = null;
  }
}
```

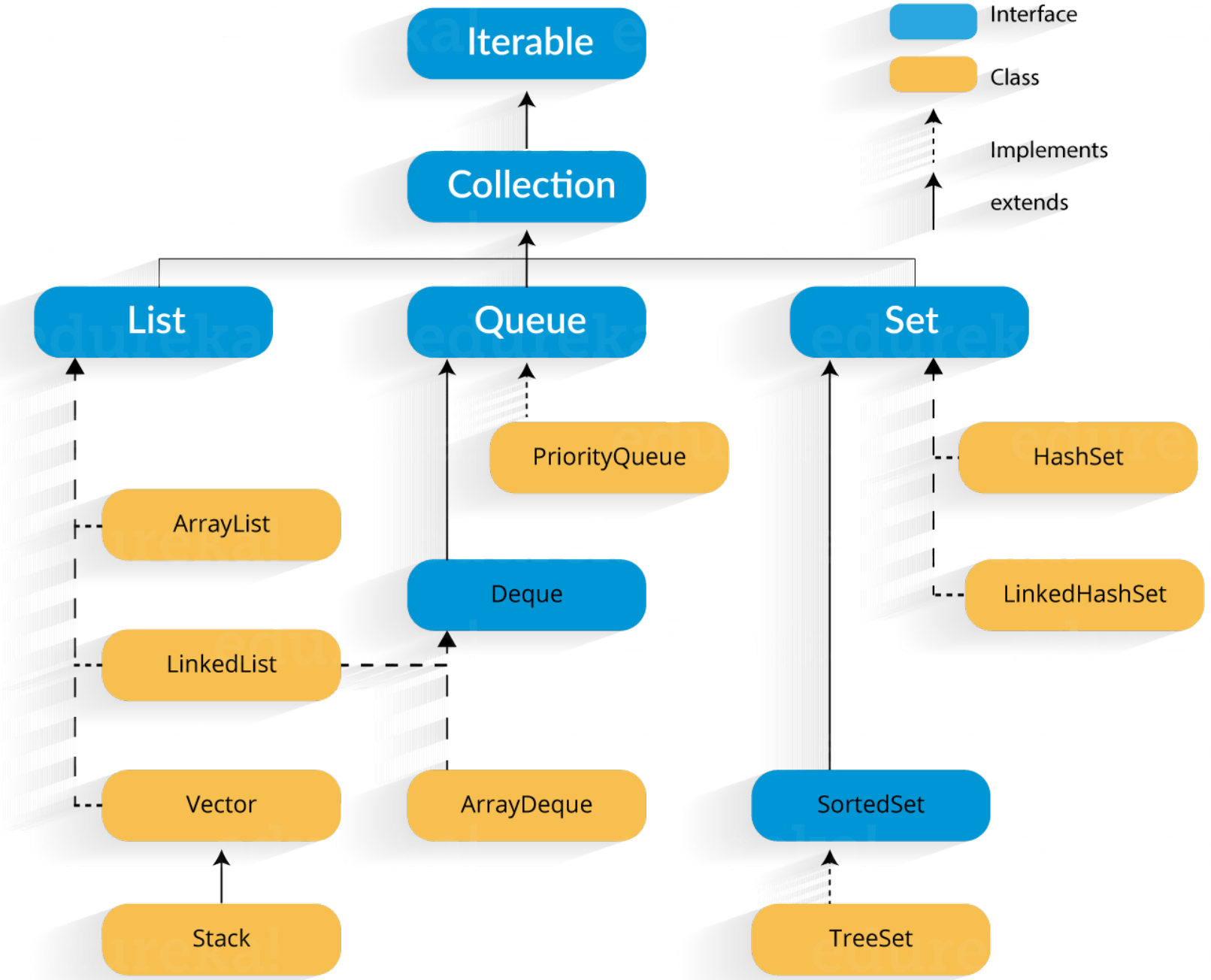
```
SinglyLList list = new SinglyLList();

list.InsertAtLast(3);
list.InsertAtLast(97);
System.out.println(list.head.data);
```

Output:

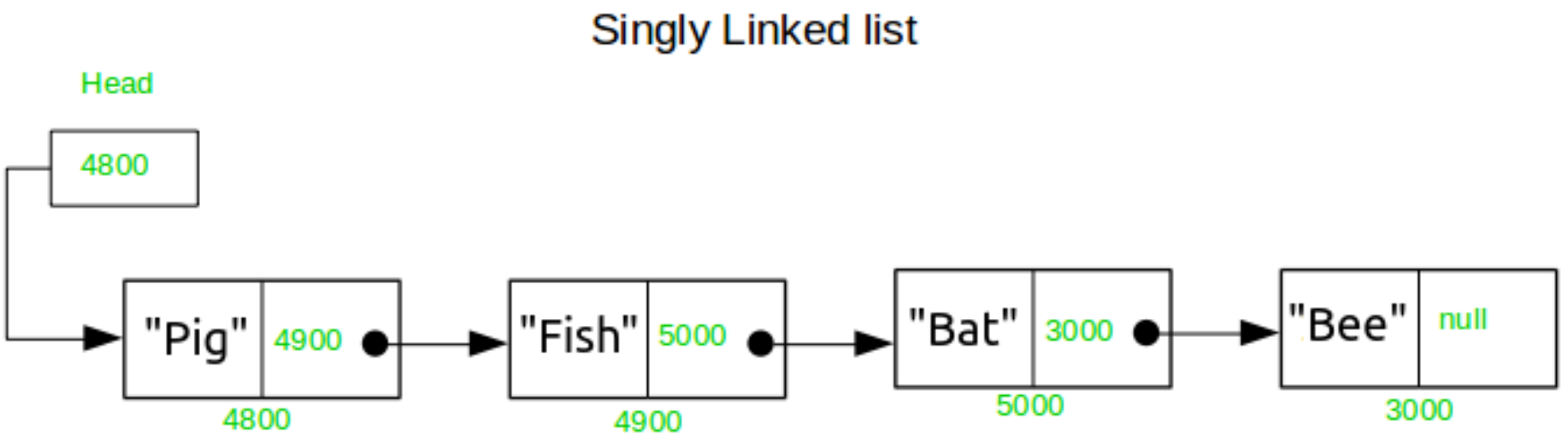
97

September 7, 2020
4th session

Recap (Hierarchy of interfaces)



LinkedList of `string`




```
class Node {
  // node variables
  String data;
  Node next;

  public Node(String data) {
    this.data = data;
    this.next = null;
  }
}
```

What if we would like to make a list of *People*?

Generics concept in Java

- **Idea 1**: Change the primitive type of the `Node` and `SinglyList` class from string to *Person* object. **Code duplication!**
- **Idea 2**: Put Object in place of String and make a string list of Object.

  Possible but will require explicit casts

- Java offers **Generic** parameter to the `Node` and `SinglyList` class.

```
class Node<E> {
  // node variables
  E data;
  Node next;

  public Node(E data) {
    this.data = data;
    this.next = null;
  }
}
```

```
class Node {
  // node variables
  People data;
  Node next;

  public Node(People data) {
    this.data = data;
    this.next = null;
  }
}
```

Generics concept in Java

- Attention, the generic method, we cannot use primitive data type (ex: `int`, `char`, `double`, etc), instead we need to use their wrapper class (ex: `Integer`, `Character`, `Double`, etc.).
- Generics also provide **compile-time** type safety that allows programmers to catch invalid types at compile time.

LinkedList implementation

```
class SinglyLList <E> {
  Node<E> head;

  void InsertAtStart(E data) {
   // create a node
   Node<E> new_node = new Node<>(data);

   new_node.next = head;
   head = new_node;
  }

  void InsertAtLast(E data) {
   Node<E> new_node = new Node<>(data);
   if (head == null) {
     head = new_node;
     return;
   }

   new_node.next = null;
   Node<E> last = head;
   while (last.next != null) {
     last = last.next;
   }
   last.next = new_node;
 }
}
```

```
class Node<E> {
  // node variables
  E data;
  Node<E> next;

  public Node(E data) {
    this.data = data;
    this.next = null;
  }
}
```

```
SinglyLList<Integer> list = new SinglyLList<>();
list.InsertAtLast(3);
list.InsertAtLast(97);

SinglyLList<Person> list = new SinglyLList<>();
list.InsertAtLast(new Person("Bob"));
list.InsertAtLast(new Person("Alice"));
```

Introduce Diamond operation

- Raw type (There is no way for type arguments to be parameterized when constructing a collection)

```
List cars = new ArrayList();
cars.add(new Object());
cars.add("car");
cars.add(new Integer(1));
```

Led to potential casting exceptions at runtime

- Generic type (which allowed us to parameterize the type arguments for classes)

```
List<String> cars = new ArrayList<String>();
cars.add("BMW");
```

  - Specifying the parameterized type in the constructor, which can be somewhat unreadable:

```
Map<String, List<Map<String, Map<String, Integer>>>> cars
= new HashMap<String, List<Map<String, Map<String, Integer>>>>();
```

  - Raw types still exist for the sake of backward compatibility, **But** it will prompt us with a warning message (`ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized`):

```
List<String> generics = new ArrayList<String>();
List<String> raws = new ArrayList();
```

Introduce Diamond operation

- The diamond operator – introduced in Java 1.7 – adds type inference and reduces the verbosity in the assignments – when using generics:

```
List<String> cars = new ArrayList<>();
cars.add("BMW");
```

- The Java compiler's detect the suitable constructor declaration that matches the invocation. For example:

```
1  public interface Engine { }
2  public class Diesel implements Engine { }
3  public interface Vehicle<T extends Engine> { }
4  public class Car<T extends Engine> implements Vehicle<T> { }
```

```
1  Car<Diesel> myCar = new Car<>();
```

- Internally, the compiler knows that Diesel implements the Engine interface and then is able to determine a suitable constructor by inferring the type.

Autoboxing

- Now that Java can know the type of objects contained in a structure, Java offers possibilities to simplify the code: for example the automatic transformation in primitive types.

```
LinkedList<Integer> mylst = new LinkedList<>();
// old style
mylst.add(new Integer(7));
Integer i = mylst.get(0);
System.out.println(i.intValue());
// new style
mylst.add(6);
int six = mylst.get(1);
System.out.println(six);
```

Generics and Parameterized type

- A **generic type** is a reference type that has one or more type parameters. These type parameters are later replaced by type arguments when the generic type is instantiated. For example:

```
interface Collection<E>  {
  public void add (E x);
  public Iterator<E> iterator();
}
```

- The instantiation of a generic type with actual type arguments is called a **parameterized type**.

```
Collection<String> coll = new LinkedList<String>();
```

Generic Type Instantiated

- By providing a type argument per type parameter.
- The type argument list is a comma separated list that is delimited by angle brackets and follows the type name. The result is a so-called parameterized type.

```
class Pair<X,Y> {
    private X first;
    private Y second;

    public Pair(X a1, Y a2) {
        first  = a1;
        second = a2;
    }
    public X getFirst() { return first; }
    public Y getSecond() { return second; }
    public void setFirst(X arg) { first = arg; }
    public void setSecond(Y arg) { second = arg; }
}
```

```
public void printPair(Pair<String, Long> pair) {
    System.out.println("(" + pair.getFirst() + "," + pair.getSecond() + ")");
}
```

OR

```
public void printPair(Pair<?, ?> pair) {
    System.out.println("(" + pair.getFirst() + "," + pair.getSecond() + ")");
}
```

```
Pair<String, Long> limit = new Pair<String, Long>("maximum", 1024L);
printPair(limit);
```

Generic `static` methods

- Goal: write a `swap` method which permutes two elements of an array.
  - Regardless of the type of array, the method for swapping two elements is the same.
  - write a **`static`** method which takes an array as a parameter
  - ➔ write a generic **`static`** method
- When declaring a generic method, the type parameter is declared before the return type and after the scope (`public, private`) and the indication of a class method (**`static`**).

```
public static class ArrayUtil {
    public static <T> void swap(T[] array, int i, int j) { ... }
```

- Note that it is not useful to specify a parameter for the `ArrayUtil` class.
- When we are going to use `swap`, we will not instantiate an object, we will just call the **`static`** method, so it is important that this method which uses a parameter is used.

Calling a generic `static` method

- When calling a generic method, we don't need to specify the type parameter, it is inferred by Java.
  - ex: `ArrayUtil.swap(villager, 2, 6);`

- If we want to, we can still give the type (this will give a better error message if something goes wrong).
  - ex: `ArrayUtil.<Gaulois> swap(villagers, 2, 6);`

Generic types are invariant

- A subtlety that is important to understand

```
LinkedList<Gaulois> lg = new LinkedList<Gaulois>();
LinkedList<Person> lp = lg;
```

Error message: Type mismatch: cannot convert from LinkedList<Gaulois> to LinkedList<Person>

- In the second line, we want to say that a list of **Gaulois** is a list of **Person**.

```
lp.add(new Person("Jules"));
Gaulois g = lg.get(0);
```

- When we get an element via the **lg** list, we don't necessarily get a **Gaulois**!
- The Java compiler will not allow the second line.
  - **Note:** If F is a class of the descendants of class M, and if G is a generic class, G<F> is not in the descendants of G<M>.
  - In other words, there is no relation between G<F> and G<M>

September 9, 2020
5th session

Wildcard (Jockers)

- Java offers the possibility of using "Wildcard" which will be used to express an unknown type.

```
LinkedList<?> cars = new LinkedList<String>();
LinkedList<?> list = new LinkedList<Gaulois>();
```

- The question mark (?) is known as the wildcard in generic programming .
- It represents an unknown type.
- The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.
- Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly.
- This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.
- Types of wildcards in Java:
  - **Unknown** Wildcard Boundary
  - **Extends** Wildcard Boundary
  - **Super** Wildcard Boundary

Unknown **Wildcards Boundary**

A list of unknown type is used in the following cases:

- When writing a method which can be employed using functionality provided in Object class.
- When the code is using methods in the generic class that don't depend on the type parameter

```
class unboundedwildcardemo
{
    public static void main(String[] args) {
        //Integer List
        List<Integer> list1= Arrays.asList(1,2,3);

        //Double list
        List<Double> list2=Arrays.asList(1.1,2.2,3.3);

        printlist(list1);
        printlist(list2);
    }

    private static void printlist(List<?> list) {
        System.out.println(list);
    }
}
```

extends Wildcards Boundary

```
public static void add(List<? extends Number> list)
```

- It can be used when you want to relax the restrictions on a variable.
- For example, to write a method that works on List <Integer>, List<Double>, and List<Number> , you can using an **upper bounded wildcard.**
- Here, Integer (i.e., list1) and Double (i.e., list2) are subclasses of class **Number**.

```
public static void main(String[] args)
{
    // Integer list
    List<Integer> list1 = Arrays.asList(4,5,6,7);
    System.out.println("Total sum is:"+sum(list1));

    // Double list
    List<Double> list2 = Arrays.asList(4.1,5.1,6.1);
    System.out.print("Total sum is:"+sum(list2));
}

private static double sum(List<? extends Number> list)
{
    double sum = 0.0;
    for (Number i: list){
        sum += i.doubleValue();
    }
    return sum;
}
```

super Wildcards Boundary

```
Syntax: Collectiontype <? super A>
```

- However if we pass list of type Double then we will get compilation error. It is because only the Integer field or its superclass can be passed . Double is not the superclass of Integer.
- Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don't use wildcard when you get and put values in a structure.
- **Note:** You can specify an upper bound or a lower bound for a wildcard, but you cannot specify both.

```
public static void main(String[] args) {
    // Lower Bounded Integer List
    List<Integer> list1 = Arrays.asList(4, 5, 6, 7);
    myprint(list1); // Integer list object is being passed

    // Number list
    List<Number> list2 = Arrays.asList(4, 5, 6, 7);
    myprint(list2); // Integer list object is being passed
}

// print Only Integer Class or SuperClass
public static void myprint(List<? super Integer> list) {
    System.out.println(list);
}
```

Order

- `Comparable` Interface contains only one method:

```
Public interface Comparable<T> {
  int compareTo(T o);
}
```

- This method returns:
  - A negative integer if the object is smaller than the object passed as a parameter.
  - zero if they are equal.
  - A positive integer if the object is larger than the object passed as a parameter.
- `String, Integer, Double, Date, GregorianCalendar` and many others all implement the `Comparable` interface.

Example

```
public class CompareToExample {
    public static void main(String args[]) {
      Integer x = 5;

      System.out.println(x.compareTo(3));
      System.out.println(x.compareTo(5));
      System.out.println(x.compareTo(8));
    }
}
```

Output

```
 1
 0
-1
```

Example

```
public class CompareToExample {
   public static void main(String args[]) {
        String str1 = "String method tutorial";
        String str2 = "compareTo method example";
        String str3 = "String method tutorial";

        int var1 = str1.compareTo( str2 );
        System.out.println("str1 & str2 comparison: "+var1);

        int var2 = str1.compareTo( str3 );
        System.out.println("str1 & str3 comparison: "+var2);

        int var3 = str2.compareTo("compareTo method example");
        System.out.println("str2 & string argument comparison: "+var3);
   }
}
```

```
str1 & str2 : -16
str1 & str3 : 0
str2 & string argument comparison : 0
```

Example

```
public class Gaulois2 extends Person
                  implements Comparable<Gaulois2> {
    public Gaulois2 (String name, int q) {
        super(name);
        this.quantitySingler = q;
    }

    String name;
    int quantitySingler;

    public int compareTo(Gaulois2 ixis) {
        return this.quantitySingler - ixis.quantitySingler;
    }
}
```

Interface Comparator

- Ex: sorting the elements of a collection: using the interface Collections

```
// interface Collections
public static <T extends Comparable<? super T>> void sort(List<T> list) {}
public static <T> void sort (List<T> list, Comparator<? super T> c) {}

public interface Comparator<T> {
   int compare(T o1, T o2);
   boolean equals(Object obj);
}
```

- To compare Gaulois, and even all Person according to their size, we can write the following class:

```
public class SortingHeight implements Comparator<Person> {
    @Override
    public int compare(Person left, Person right) {
        // TODO Auto-generated method stub
        return left.heigth < right.heigth ? -1 :
            (left.heigth == right.heigth ? 0 : 1);
    }}
```

Example

- Then, we can use this new class to sort Person according to their height.

```
public class SortingHeight implements Comparator<Person> {
    @Override
    public int compare(Person left, Person right) {
        // TODO Auto-generated method stub
        return left.height < right.height ? -1 :
            (left.height == right.height ? 0 : 1);
    }

    public static void main(String args[]) {
        Person obelix = new IrreducibleGaulois("Obelix", 1.81);
        Gaulois astrix = new IrreducibleGaulois("Astrix", 1.60);
        Person cesar = new Person("Cesar", 1.75);

        ArrayList<Person> persons = new ArrayList<>();
        persons.add(obelix);
        persons.add(astrix);
        persons.add(cesar);

        for(Person p : persons)
            System.out.println(p.presentation());

        Comparator<Person> heigth = new SortingHeight();
        Collections.sort(persons, heigth);

        for(Person p : persons)
            System.out.println(p.presentation());
    }
```

Output:

```
My name is Obelix. I am a Gaulois.
My name is Astrix. I am a Gaulois.
My name is Cesar.

My name is Astrix. I am a Gaulois.
My name is Cesar.
My name is Obelix. I am a Gaulois.
```