Week 2

Genericity

A generic class is one whose definition has multiple type parameters. As an example `List<String>` corresponds to the parametrized type `List<E>`

Here's an example of a generic class:

```
public final class Cell<E> {
  private final E e;

  public Cell(E e) { this.e = e; }
  public E get() { return e; }
}
```

And here's how we instantiate a generic class:

```
Cell<String> message = new Cell<>("Bonne année ");
Cell<Date> date = new Cell<>(Date.today());
System.out.println(message.get() + date.get().year());
```

And here's how we define a generic pair class:

```
public final class Pair<F, S> {
  private final F fst;
  private final S snd;

  public Pair(F fst, S snd) {
    this.fst = fst;
    this.snd = snd;
  }
  public F fst() { return fst; }
  public S snd() { return snd; }
}
```

Now a special generic method syntax is as follows:

```
public final class Cell<E> {
  private final E e;
  // … comme avant
  public <S> Pair<E, S> pairWith(S s) {
    return new Pair<>(e, s);
  }
}
```

This allows our Cell to be paired with any different type. Here's an example:

```
Cell<String> message = new Cell<>("Bonne année ");
Pair<String, Date> pair =
  message.pairWith(Date.today());
System.out.println(pair.fst() + pair.snd().year());
```

Primitive types and genericity

If a generic class accepts a primitive class as argument, we must make sure to pass in the wrapper class instead done as follows:

```
Cell<Integer> c = new Cell<>(new Integer(1));
int succ = c.get().intValue() + 1;
```

Bounds

If we want to limit the type passed into a parameter we set a bound as follows:

```
public final class Cell<E extends Number> {
  // … comme avant
}
```

Week 2

Collection

Generally, for each collection, Java contains an interface and multiple implementations of the interface.

A *view* is a type exposing part of another type. For instance `subList` is a view of `List`.

Now some implementations of collection offer methods to modify the object. However, the object may be easily made immutable by setting all these methods to throw `UnsupportedOperationException`

To achieve a nonmodifiable implementation of collection, we must do one of the following:

- Use a method returning an immutable structure such as `List.of`
- Obtain a non-modifiable view of collection by using methods such as `unmodifiableList`

Lists

A list is ordered and dynamic in Java.

```
public interface List<E> extends Collection<E> {
  // … méthodes
}
```

And here is the list view method:

```
List<E> subList(int b, int e)
// returns subList between given index lats index being exclusive
```

And as mentioned before, we use the `of` method to construct an immutable List. Similarly, to obtain an immutable copy, use `copyof`

Two popular implementation of lists are ArrayList and LinkedList. While LinkedList performs adding in constant time, it does search in linear time and vice versa for ArrayList.

And some common List types:

- stack(insert and delete at same extremite)
- queue(insertion and deletion at two extrema)

And here's an easy way to loop through a list:

```
List<String> l = …;
for (String s: l)
  System.out.println(s);
```

Alternatively, we can use the iterator object:

```
List<String> l = …;
Iterator<String> i = l.iterator();
while (i.hasNext()) {
  String s = i.next();
  System.out.println(s);
}
```

Week 3

Set extends collection

The set interface extends Collection with `HashSet` and `TreeSet` as implementations. The `set` interface is similar to its mathematics counterpart offering the following methods:

Opération Méthode union () `addAll` test d'appartenance (?) `contains` test d'inclusion (?) `containsAll` différence (\) `removeAll` intersection () `retainAll`

TreeSet owes its name to the fact that it stores the elements of the set in a binary search tree, while HashSet owes its name to the fact that it stores them in a hash table. These concepts will be discussed later, but it is important to know that thanks to them, TreeSet can implement the main operations on sets (adding, membership test, etc.) in O(log n), while HashSet does even better and implements them in O(1), which is remarkable!

HashSet / TreeSet rule: Use HashSet as an implementation of sets in Java, except when it is useful to browse items in ascending order, in which case you may prefer TreeSet.

Equality of sets

Now how do we consider if two set objects are equal. We can resort to `equality by reference` or `equality by structure`.

Comparable interface

This interface offers `compareTo` which returns an int, -1,0,1. The below shows how equals is equivalent to compareTo.

`o1.equals(o2) ⇔ o1.compareTo(o2) == 0`

Comparator interface

This interface although similar to comparable has the main difference that it is implemented by a foreign class. That is it is capable of comparing two objects of different type.

TreeSet

Stores objects in sorted, ascending order.

```
Set<Integer> s = new TreeSet<>();
s.addAll(Arrays.asList(1,3,5,7,2,4,6,8));
for (int i: s)
  System.out.println(i); // 1, 2, …, 8
```

It is possible to make TreeSet in descending order as below:

```
Set<Integer> s = new TreeSet<>(new IntInvComparator());
s.addAll(Arrays.asList(1,3,5,7,2,4,6,8));
for (int i: s)
  System.out.println(i);
```

Hashing

All hashing functions satisfy:

$$\forall x, y : x = y \Rightarrow h(x) = h(y)$$

A hashing function is *perfect* if:

$$\forall x, y : x \neq y \Rightarrow h(x) \neq h(y)$$

Taking the contrapositive of the above it is obvious that a hashing function is perfect if it is injective.

Hashing in JAVA

The `equals` and `hashCode` methods are said to be *compatible* if the following holds:

x.equals(y) ⇒ x.hashCode() == y.hashCode()

Quick summary of Collectible objects

Collection Methods required `List<E>` *none* `Set<E>` `equals` `HashSet<E>` `equals` and `hashCode` `TreeSet<E>` `equals` and `compareTo` `TreeSet<E>` `equals` and `compare`

Dictionary aka. Associative map