Data Structures and Algorithms

XMUT-COMP 103 - 2020 T1

Lecture by : Dr Karsten Lundqvist

School of Engineering and Computer Science

Victoria University of Wellington

Java Interfaces and ADT's - - Lecture 03

- A Java ***Interface*** corresponds to an Abstract Data Type
  - Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value)
  - Behaviour of methods is only given in comments (but cannot be enforced)

× No constructors - can't make an instance: ~~new~~ Set() ~~new~~ List()

× No fields - doesn't say how to store the data

Type variable – stands for whatever it is a set of..

```
public interface Set <E> {
    public boolean add(E item);          /*…description…*/
    public boolean remove(E item);       /*…description…*/
    public boolean contains(E item);     /*…description…*/
    …
        // (plus lots more methods in the Java Set interface)
```

Collections Class and Arrays Class

- Two Classes of useful methods to operate on
  - Collection objects
    - some methods work on any kind of collection
    - some only work on lists
    - some on work on collections of certain kinds of objects
  - Array objects

- eg, sort(…), reverse(…), max(…), fill(… …)

- Most methods are static, (like the Math class)
  - Collections.reverse(mylist);

Using Java Collection library

- Your program can
  - Declare a variable, parameter, or field of the interface type

```
private  List <String>  images;                    // defined using the ADT – interface
Set <Student>  students;
```

  - Create a collection object

```
images= new ArrayList <String> ();        // constructed using a class
students= new HashSet <Student> ();
```

  - Call methods on that variable, parameter, or field

```
images.add(UIFileChooser.open("Choose an image file");
images.set(i, images.remove(j));
Collections.fill(images, “sunset.jpg");
if (students.contains(st)){ ….
```

Why?

- Why use the Interface type to declare the field/variable then use a class to make the object?

- Can't make an object of the interface type:

```
List <Double> myNumbers = new List <Double>();
```

- Why not just use the class?

```
ArrayList <Double> myNumbers = new ArrayList <Double>();
```

- More flexible design to use the Interface type:

```
List <Double> myNumbers = new ArrayList <Double>();
```

Could change the class later; rest of program works on any kind of List.

Comments on code style for 103

- I will drop “this.” except when needed.
  - instead of `this.loadFromFile(fname)`
    just `loadFromFile(fname)`
  - instead of `this.shapes.addShape(shape)`
    just `shapes.addShape(shape)`

- I may leave out { } when surrounding just one statement
  - instead of

```
while (i < name.length) {
   name[i] = null;
}
```

    just

```
while (i < name.length)
   name[i] = null;
```

Undo

How does it work?

- Have to keep a record of all the actions as they are done
  - Have to keep enough information to be able to undo them later

- Undo button steps backwards through the record of actions, undoing the next one.

What kind of collection do we need for the record of actions?

What kind of object do we need for each action?

Brick Constructor

- What are the actions to remember?
- What info do we need to remember for each action?
- To incorporate undo:
  - make a stack of undo record
  - at each action in the program, add a new record to the stack
  - add an undo button and method which pops the top record from the stack and undoes the action
  - create a new class to store all the information for each undo record
    - fields
    - constructors
    - getters