FUNDAMENTALS

CS62

DATA STRUCTURES AND ADVANCED PROGRAMMING

6: Interfaces and Generics

Alexandra Papoutsaki
she/her/hers

Lecture 6: Interfaces and Generics

- Interfaces
- Background
- Generics

Interfaces

- Contracts of what a class must do, not how to do it, abstracting from implementation.
- In Java, an interface is a reference type (like a class), that contains abstract methods and default methods.
- A class that implements an interface is obliged to implement its abstract methods.
- Interfaces cannot be instantiated (no new keyword). They can only be *implemented* by classes or *extended* by other interfaces.

Example

```
public interface Enrollable{
    void enrollInCourse(String course);
    void withdrawFromCourse(String course);
    void viewCourseSchedule();

    default int getMaxCredits(){
        return 4;
    }
}
```

Example

```
class PomonaStudent implements Enrollable{

…
    public void enrollInCourse(String course) {
         // implementation
     }

public void withdrawFromCourse(String course) {
     // implementation
 }

public void viewCourseSchedule() {
     // implementation
 }
```

Example

```
class FourthYearPomonaStudent extends PomonaStudent{

…
    public int getMaxCredits(){
          return 6;
     }
}
```

Interfaces

- A class can implement multiple interfaces.
  - `class A implements Interface1, Interface2{…}`
- An interface can extend multiple interfaces.
  - `public interface GroupedInterface extends Interface1,Interface2{…}`

PRACTICE TIME - Worksheet

- Create an interface called `Adoptable` that contains four abstract methods: a `void requestAdoption()`, `boolean isAdopted()`, `void completeAdoption()`, and `String makeHappyNoise()`.
- Have the class `Animal` implement the interface. You can provide some very minimal implementation of the methods so that you don't receive a compile-time error.
- Override the `makeHappyNoise()` in the `Cat` and `Dog` subclasses.

ANSWER

```
public interface Adoptable {
    void requestAdoption();  // Method to initiate the adoption process
    boolean isAdopted();     // Method to check if the animal has been adopted
    void completeAdoption(); // Method to finalize the adoption
    String makeHappyNoise();   // Method that returns a happy noise the adopted animal makes
}
public class Animal implements Adoptable {
    ...
    public void requestAdoption() {
        // Implementation for an animal's adoption request
    }
    public boolean isAdopted() {
        return adopted;
    }
    public void completeAdoption() {
        // Implementation to finalize the adoption for an animal
        adopted = true;
    }
    public String makeHappyNoise(){
        return "I was adopted hooray!";
    }
```

ANSWER

```
public class Cat extends Animal{

    …

     public String makeHappyNoise(){

          return "I am a happy cat!";

     }

}

public class Dog extends Animal{

    …

     public String makeHappyNoise(){

          return "I am a happy dog!";

     }

}
```

Lecture 6: Interfaces and Generics

- Interfaces
- Background
- Generics

Why do we need data structures?

- To organize and store data so that we can perform efficient operations on them based on our needs.
  - Imagine walking to an unorganized library and trying to find your favorite title or books from your favorite author.
- We can define efficiency in different ways.
  - Time: How fast can we perform certain operations on a data structure?
  - Space: How much memory do we need to organize our data in a data structure?
- There is no data structure that fits all needs.
  - That's why we're spending a semester looking at different data structures.
  - So far, the only data structure we have encountered is arrays.
    - And ArrayList, but informally.

Types of operations on data structures

- Insertion: adding a new element in a data structure.
- Deletion: Removing (and possibly returning) an element.
- Searching: Searching for a specific data element.
- Replacement: Replacing an existing element with a new one (and possibly returning old).
- Traversal: Going through all the elements.
- Sorting: Sorting all elements in a specific way.
- Check if empty: Check if data structure contains any elements.

- Not a single data structure does all these things efficiently.
- You need to know both the kind of data you have, the different operations you will need to perform on them, and any technical limitations to pick an appropriate data structure.

Linear vs non-linear data structures

- Linear: elements arranged in a linear sequence based on a specific order.
  - E.g., Arrays, ArrayLists, linked lists, stacks, queues.
  - Linear memory allocation: all elements are placed in a contiguous block of memory. E.g., arrays and ArrayLists.
  - Use of pointers/links: elements don't need to be placed in contiguous blocks. The linear relationship is formed through pointers. E.g., singly and doubly linked lists.
- Non-linear: elements arranged in non-linear, mostly hierarchical relationship.
  - E.g., trees and graphs.

Lecture 6: Interfaces and Generics

- Interfaces
- Background
- Generics

List interface

- We want any list-based data structure to support adding elements, removing them, indexing the data structure to support adding, replacing, and removing an element at a specific index.
- We will build an interface List that forces any data structure that implements it to implement these operations.

Lists should support any type of element

- We want our data structure to support any type of elements, as long as they are of the same type. We could use the class Object but this requires casting to the desired type:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- Instead, we will use generics.

Generics

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
public class ArrayList<E> implements List<E>{
```

Formal type parameters

- In the invocation, all occurrences of the formal type parameters are replaced by the actual type argument

- ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```
  ```

Generics

- Generics enable types (that is classes and interfaces) to be used as parameters when defining classes, interfaces, and methods.

- E: element (common in data structures), T: type, K: key, V: value, N: number.

- The additional advantage is that bugs are now caught at compile time instead of runtime (much easier to fix!)

Readings:

- Interfaces: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- Generics: https://docs.oracle.com/javase/tutorial/java/generics/index.html
  https://docs.oracle.com/javase/tutorial/extra/generics/intro.html

Code

- Lecture 6 code

Worksheet

- Lecture 6 worksheet