COMP 1020 - Interfaces

UNIT 9

Interfaces

- A Java class encapsulates the data that defines it, and the operations that we can perform on it.
  - The data (instance variables) define its properties; that is, what a particular object *is*.
  - The operations (instance methods) define *how* it processes that data.
- But even more important is a definition of what it *can do*.

Interfaces

- The public part of our class is the only way that other code can use our class.
  - That is, the headers of all our public methods
- These method headers define the **interface** to our class. They define what our class *can do*.
  - Without these, nobody could use our class.
  - They provide access to the rest of the program.

Interfaces are Critical

- In the real world, we use interfaces everywhere. They restrict access to things, often with good reason!

Interfaces are Critical

- In the real world, we use interfaces everywhere. They restrict access to things, often with good reason.

These interfaces make it difficult to do something hazardous!

Not User Interfaces

- To clarify, we are NOT talking about user interfaces.
  - Things like text boxes, buttons, and mouse pointers are part of a user interface.
- We are talking about the programming interfaces between different parts of our programs.

Interfaces vs Implementations

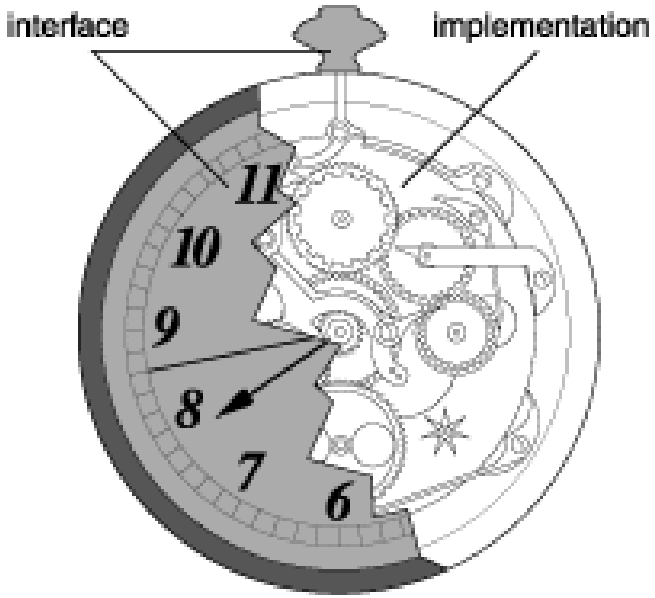
- Ideally the “public” parts of our classes are the only parts that the rest of our program gets to see.

- The other parts, like instance variables, and the code that makes up our methods, are hidden.

  - These other parts of our class are known as its **implementation**.

Hiding Implementations

- We separate (conceptually) the public interface from the private implementation.
- This enables us to use the class without having to know how it works.
- This separation is critical when our projects get larger and more complex!



Hiding Implementations

- This interface hides an entire electrical grid.

We don’t need to understand how it all works if we just want to plug in our toaster.

A simple example: CourseList

- Here's a method from a class that stores a list of university courses:

```
// Here's the public part:
public class CourseList {


    public void sortByCourseName() {


    }
}
```

A simple example: CourseList

- Here's a method from a class that stores a list of university courses:

```
public class CourseList {
    // How we store the list of courses is private
    // Array? ArrayList? LinkedList? Something else?
    public void sortByCourseName() {
        // The sorting algorithm we use is private too
        // Selection? Insertion? Bubble? Something else?
    }
}
```

A simple example: CourseList

- The interface forms a wall that protects the privacy of our implementation from the rest of the program. We can change the sort, or the list, without changing the part of the program that uses it.

courseList.sortByCourseName();

```
public void sortByCourseName() {
    // Our implementation goes here

}
```

A simple example: CourseList

- We can use the CourseList in a large program without knowing the details like how the courses are stored, or how we sort them.
  - This enables us to treat the entire class as an independent subproblem of a larger problem.
  - We can assume it works as advertised: it does what the interface says it does.
  - In the future, it will enable us to work with others on a team.

Big interfaces: APIs

- An API, or **Application Programming Interface**, defines an interface that is published for different programs to use.
  - You have been using the Java API since you first printed “Hello, World”.
  - You can call System.out.println from the Java API without knowing how it works!

https://docs.oracle.com/en/java/javase/11/docs/api/

Formal interfaces

- Java allows you to formally specify an interface, separately from the class that implements it.

- A Java interface (this is a key word in Java) lists public method headers but doesn't include private stuff.

  - Just the method headers followed by semicolons.

- We can write an interface and then one or more classes that implements (another Java key word) it.

An example: TablePrinter

- This interface describes methods that will neatly print the contents of a class in a table:

```
public interface TablePrinter {
    void printHeader();
    void printData();
}
```

An example: TablePrinter

- This interface describes methods that will neatly print the contents of a class in a table:

```
public interface TablePrinter {
    void printHeader();
    void printData();
}
```

- Note that method headers omit the word “public”: all the methods in our interface are implicitly public.

- There are no instance variables or method code.

An example: TablePrinter

- A class can implement the interface:

```
public class Course implements TablePrinter {
    // Instance variables and other methods not shown
    public void printHeader() {
        System.out.println("Faculty: Code: Name:");
    }
    public void printData() {
        System.out.printf("%8s %5s %s\n",
            faculty, code, name);
    }
}
```

An example: TablePrinter

- More classes can implement the interface

```
public class Student implements TablePrinter {
    // it has printHeader and printData
    // implementations (not shown)
    ...
}
```

Some notes about interfaces

- The methods listed in the interface must have identical headers in every class that implements them.
  - Except they must be explicitly public.
  - A class must implement all of them.
- A class can implement more than one interface; e.g.:

```
class Course implements TablePrinter, Registration { ...
```

Using interfaces

- You can declare variables of an interface type, and have them refer to any object whose class implements that interface:

```
TablePrinter printableObject;
printableObject = course;

...

// later:
printableObject = student;
```

Using interfaces

- Then you can call any method from the interface on that object:

```
TablePrinter printableObject;
printableObject = course;

printableObject.printHeader();
printableObject.printData();
```

- But you can't call any method that's not listed in the interface through the variable of the interface type (at least, not directly; more later).

Using interfaces

- More interestingly, you can use them as parameters:

```
public static void printInTable(TablePrinter object) {
    // more fancy printing code not shown
    object.printHeader();
    object.printData();
}

// call it with any TablePrinter implementation:
printInTable(course);
printInTable(student);
```

Using interfaces

- We can use an interface type to describe the kind of object we store in a collection.

```
// works just as well with arrays
ArrayList<TablePrinter> printable = new ArrayList<>();

printable.add(course);
printable.add(student);

for (TablePrinter p : printable) {
    p.printData();
}
```

Another example: collections

- We can use an interface type to describe the kind of object we store in a collection.

- And collections themselves are frequently described using interfaces.

Another example: collections

- And collections themselves are frequently described using interfaces:

```
interface StudentList {
    void add(Student s);
   Student find(int studentNumber);
    void remove(int studentNumber);
    void sortByName();
    void printInTable();
}
```

Another example: collections

- The collection can be implemented in different ways:

```
class ArrayListOfStudents implements StudentList {
    // sortByName uses a selection sort
    …
}
class LinkedListOfStudents implements StudentList {
    // sortByName uses an insertion sort
    …
}
// or: ArrayOfStudents, maybe others
```

Another example: collections

- In our program, we use the StudentList interface any time we want to refer to a list:

```
public static void main(String[] args) {
    StudentList students;
    // the only place we name the implementing class:
    students = new ArrayListOfStudents();

    students.add(…);
    students.sortByName();
    students.printInTable();
}
```

Another example: collections

- And now we can change the implementation by only changing one line in the program:

```
public static void main(String[] args) {
    StudentList students;
    // the only place we name the implementing class:
    students = new LinkedListOfStudents();

    students.add(…);
    students.sortByName();
    students.printInTable();
}
```

Java's collections

- Java's collection classes use this strategy.

- Both ArrayList and LinkedList implement the standard List interface. This means we can write:

```
List<Student> list;
list = new ArrayList<>(); // or replace with LinkedList

// the rest of the program works the same with any list:
list.add(new Student(...));
```

- It's preferred style to use the interface when we can.

A very practical example

- We know that we can compare strings alphabetically with the compareTo method.
- We can extend this idea to compare any two objects.
- Java's standard Comparable interface defines a single compareTo method.

A very practical example

- For example, a Comparable student that can be ordered by student names:

```
// a student can be compared to any other student
class Student implements Comparable<Student> {
    // again, other parts not shown
    public int compareTo(Student other) {
        return name.compareTo(other.name);
    }
}
```

A very practical example

- Now, inside a sorting method, we can compare a student using compareTo:

```
// selection sort:
...
if (student[i].compareTo(student[minPos]) < 0) {
    minPos = i;
}
...
```

A very practical example

- Java also has a standard (fast!) sorting function that works with a list of any objects implementing Comparable:

```
ArrayList<Student> students = ...;

// after we put some students in the list:
Collections.sort(students);
```

- This only works if the objects in the ArrayList implement the Comparable interface.