# Topic 10.1: Interfaces

# Learning Goals:

- Differentiate between an interface and its implementation.
- Force a class to implement abstract methods by having it implement an interface.
- Use interfaces as variable types, parameter types, and return value types.

# 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.
- Let’s check out the StudentListInterface.zip folder to see this

# 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.
- Let’s check out the StudentListInterface.zip folder to see this

# The power of this?

- We can change a **single line of code** and use an **entirely different implementation**
  - The “user” has no idea (the output has not changed)

# Java Collections

- Java's collection classes use this strategy.
  - Both ArrayList and LinkedList (Java built-in LL, not ours) 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 practical Example: Comparable

- 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 practical Example: Comparable

- 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.
- 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 practical Example: Comparable

- 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.
- 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);
      }
}
```

- Now when we call students[i].compareTo(studentToAdd), it works
  - ...but wait...that worked before....what is the real ***secret power*** this gave us?

# A practical Example: Comparable

- **BEHOLD**
- Java has a standard (fast!) sorting function that works with a list of any objects implementing Comparable
- E.G. (Same Student class as previous slide, implementing Comparable)

```
public static void main(Stirng[] args) {
    ArrayList<Student> students = new ArrayList<Student>();

    // add some students in the list:
    Collections.sort(students); // .sort will use our compareTo
}
```

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