Building Java Programs

Interfaces, Comparable

reading: 9.5 - 9.6, 16.4, 10.2
HOW TO WRITE GOOD CODE:

START PROJECT.

DO THINGS RIGHT OR DO THEM FAST?

FAST

CODE FAST

NO

DOES IT WORK YET?

ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.

RIGHT

CODE WELL

NO

ARE YOU DONE YET?

NO

NO, AND THE REQUIREMENTS HAVE CHANGED

THROW IT ALL OUT AND START OVER.

GOOD CODE.
Related classes

Consider classes for shapes with common features:

- **Circle (defined by radius \( r \))**:  
  \[
  \text{area} = \pi r^2, \quad \text{perimeter} = 2\pi r
  \]

- **Rectangle (defined by width \( w \) and height \( h \))**:  
  \[
  \text{area} = w h, \quad \text{perimeter} = 2w + 2h
  \]

- **Triangle (defined by side lengths \( a \), \( b \), and \( c \))**:  
  \[
  \text{area} = \sqrt{s(s-a)(s-b)(s-c)}
  \]
  where \( s = \frac{1}{2}(a+b+c) \),  
  \[
  \text{perimeter} = a + b + c
  \]

- Every shape has these, but each computes them differently.
Interfaces (9.5)

• **interface**: A list of methods that a class can promise to implement.
  
  • Inheritance gives you an is-a relationship *and* code sharing.
    • A **Lawyer** can be treated as an **Employee** and inherits its code.
  
  • Interfaces give you an is-a relationship *without* code sharing.
    • A **Rectangle** object can be treated as a **Shape** but inherits no code.
  
  • Analogous to non-programming idea of roles or certifications:
    • "I'm certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
    • "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know how to compute my area and perimeter."
Interface syntax

public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}

Example:
public interface Vehicle {
    public int getSpeed();
    public void setDirection(int direction);
}
Shape interface

// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}

- Saved as Shape.java

- **abstract method**: A header without an implementation.
  - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.
Implementing an interface

public class name implements interface {
    ...
}

- A class can declare that it "implements" an interface.
- The class must contain each method in that interface.

public class Bicycle implements Vehicle {
    ...
}

(Otherwise it will fail to compile.)
Banana.java:1: Banana is not abstract and does not override abstract method area() in Shape
public class Banana implements Shape {
    ^
Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
- They allow **polymorphism**.
  (the same code can work with different types of objects)

```java
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
    System.out.println();
}
...
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```
Linked vs. array lists

- We have implemented two collection classes:
  - ArrayIntList
    - Index: 0 1 2 3
    - Value: 42 -3 17 9
  - LinkedIntList
    - Front: 42
    - Next: -3
    - Next: 17
    - Next: 9

- They have similar behavior, implemented in different ways. We should be able to treat them the same way in client code.
An IntList interface

// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}

public class ArrayIntList implements IntList { ...
public class LinkedIntList implements IntList { ...
public class ListClient {
    public static void main(String[] args) {
        IntList list1 = new ArrayIntList();
        process(list1);

        IntList list2 = new LinkedIntList();
        process(list2);
    }

    public static void process(IntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1);
        System.out.println(list);
    }
}
**ADTs as interfaces (11.1)**

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - Collection, Deque, List, Map, Queue, Set

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`

  - They messed up on `Stack`; there's no `Stack` interface, just a class.
Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:
  
  ```java
  List<String> list = new ArrayList<String>();
  ```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:
  
  ```java
  public void stutter(List<String> list) {
      ...
  }
  ```
The Comparable Interface

reading: 10.2
# Collections class

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(list, value)</td>
<td>returns the index of the given value in a sorted list (&lt; 0 if not found)</td>
</tr>
<tr>
<td>copy(listTo, listFrom)</td>
<td>copies listFrom's elements to listTo</td>
</tr>
<tr>
<td>emptyList(), emptyMap(), emptySet()</td>
<td>returns a read-only collection of the given type that has no elements</td>
</tr>
<tr>
<td>fill(list, value)</td>
<td>sets every element in the list to have the given value</td>
</tr>
<tr>
<td>max(collection)</td>
<td>returns largest/smallest element</td>
</tr>
<tr>
<td>min(collection)</td>
<td></td>
</tr>
<tr>
<td>replaceAll(list, old, new)</td>
<td>replaces an element value with another</td>
</tr>
<tr>
<td>reverse(list)</td>
<td>reverses the order of a list's elements</td>
</tr>
<tr>
<td>shuffle(list)</td>
<td>arranges elements into a random order</td>
</tr>
<tr>
<td>sort(list)</td>
<td>arranges elements into ascending order</td>
</tr>
</tbody>
</table>
Ordering and objects

- Can we sort an array of Strings?
  - Operators like `<` and `>` do not work with `String` objects.
  - But we do think of strings as having an alphabetical ordering.

- **natural ordering**: Rules governing the relative placement of all values of a given type.

- **comparison function**: Code that, when given two values $A$ and $B$ of a given type, decides their relative ordering:
  - $A < B$, $A == B$, $A > B$
The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:
    ```java
    public int compareTo(String other)
    ```
  - A call of `A.compareTo(B)` will return:
    - a value $< 0$ if $A$ comes "before" $B$ in the ordering,
    - a value $> 0$ if $A$ comes "after" $B$ in the ordering,
    - or $0$ if $A$ and $B$ are considered "equal" in the ordering.
Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```java
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) { // true
    ...
}
```

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (a &lt; b) { ...</code></td>
<td><code>if (a.compareTo(b) &lt; 0) { ...</code></td>
</tr>
<tr>
<td><code>if (a &lt;= b) { ...</code></td>
<td><code>if (a.compareTo(b) &lt;= 0) { ...</code></td>
</tr>
<tr>
<td><code>if (a == b) { ...</code></td>
<td><code>if (a.compareTo(b) == 0) { ...</code></td>
</tr>
<tr>
<td><code>if (a != b) { ...</code></td>
<td><code>if (a.compareTo(b) != 0) { ...</code></td>
</tr>
<tr>
<td><code>if (a &gt;= b) { ...</code></td>
<td><code>if (a.compareTo(b) &gt;= 0) { ...</code></td>
</tr>
<tr>
<td><code>if (a &gt; b) { ...</code></td>
<td><code>if (a.compareTo(b) &gt; 0) { ...</code></td>
</tr>
</tbody>
</table>
**compareTo and collections**

- You can use an array or list of strings with Java's included `binarySearch` method because it calls `compareTo` internally.

  ```java
  String[] a = {"al", "bob", "cari", "dan", "mike"};
  int index = Arrays.binarySearch(a, "dan"); // 3
  ```

- Java's TreeSet/Map use `compareTo` internally for ordering.

- A call to your `compareTo` method should return:
  - a value `< 0` if this object is "before" the other object,
  - a value `> 0` if this object is "after" the other object,
  - or `0` if this object is "equal" to the other.
Comparable (10.2)

```java
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the Comparable interface to define a natural ordering function for its objects.

- A call to your `compareTo` method should return:
  - a value < 0 if this object is "before" the other object,
  - a value > 0 if this object is "after" the other object,
  - or 0 if this object is "equal" to the other.

- If you want multiple orderings, use a Comparator instead (see Ch. 13.1)
Comparable template

public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
compareTo tricks

- **delegation trick** - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```java
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- **toString trick** - If your object's `toString` representation is related to the ordering, use that to help you:

```java
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```
compareTo tricks

- *subtraction trick* - Subtracting related values produces the right result for what you want `compareTo` to return:

```java
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;  // different x
    } else {
        return y - other.y;  // same x; compare y
    }
}
```

- The idea:
  - if \( x > \text{other}.x \), then \( x - \text{other}.x > 0 \)
  - if \( x < \text{other}.x \), then \( x - \text{other}.x < 0 \)
  - if \( x == \text{other}.x \), then \( x - \text{other}.x == 0 \)

- NOTE: This trick doesn't work for doubles (but see `Math.signum`)