Interfaces

CSE/IT 213

NMT Department of Computer Science and Engineering

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

— Martin Fowler

"A picture is worth a thousand words. An interface is worth a thousand pictures."

— Ben Shneiderman

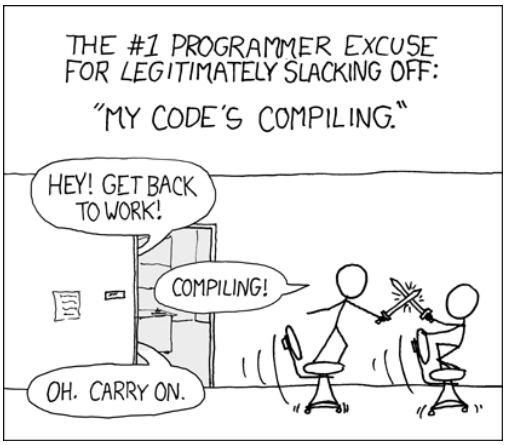


Figure 1: `https://xkcd.com/303/`

Introduction

In this assignment you will learn to use Interfaces in Java, and see some of the ways in which they differ from abstract classes. We also explain some of the logic behind Java's exception handling system, and introduce the Java 8 syntax for lambda expressions.

Exception Handling

In more low level programming languages, recovering from a serious error is almost impossible. In C for example, if your program accesses out of bounds memory, there's nothing you can do to stop it from segfaulting. Java takes a different approach. Any time a serious error occurs in Java, the program throws an *exception*. An exception is an object that represents a run time error, which the programmer can use to determine how the error should be handled.

For example, when you open a file for reading:

```
1 FileReader file = new FileReader("file.txt");
```

It is possible for the `FileReader` to throw an `IOException` – if the file does not exist, or you don't have permission to read it. When Java knows that an error like this might occur, it gives you exactly two options:

1. You can declare that your method `throws` the exception. This is what you've been doing up until now. This means that you are choosing *not* to handle the exception! So when the error occurs your method will crash, the exception will be passed down to the *caller*.

   The caller is then given the same ultimatum – either handle the error, or `throw` it down to *its* caller. If the exception is still not handled by the time it gets down to `main()`, then the program will finally crash.

2. Alternatively, you can wrap the offending code in a `try`/`catch` block. Catching the exception means that the error has been handled, and gives you the opportunity to stop the program from crashing.

The syntax for a `try`/`catch` block has two parts. The `try` block surrounds whichever parts of the program *might* fail. After that you have one or more `catch` blocks. Each one detects a specific `Exception` and surrounds code that telling the program what to do with it.

There is also an optional `finally` block, which *always* executes – whether there was an error or not. This is sometimes useful for ensuring resources are cleaned up when handling code that might fail.

```java
/**
 * @throws IOException: Any code that calls this method needs to either catch
 * this error, or also throw an IOException
 */
public static String readFirstLine(String fileName) throws IOException {
    FileReader file = new FileReader(fileName);
    Scanner reader = new Scanner(file);
    String first;

    // Scanner.nextLine() can throw a NoSuchElementException if the file is empty
    try {
        first = reader.nextLine();
    } catch (NoSuchElementException e) {
        first = "EOF";
    } finally {
        reader.close(); // in either case we want to close the file
    }

    return first;
}
```

Figure 2: An example of a method that catches one exception, but throws another

Java knows which methods in its standard library can and cannot throw exceptions, so it can determine whenever an error is possible. The system is designed so that you have to explicitly handle the error, one way or another. If you decide not to handle the exception in your method, then you are leaving it up to whoever *called* your method to pick up the slack.

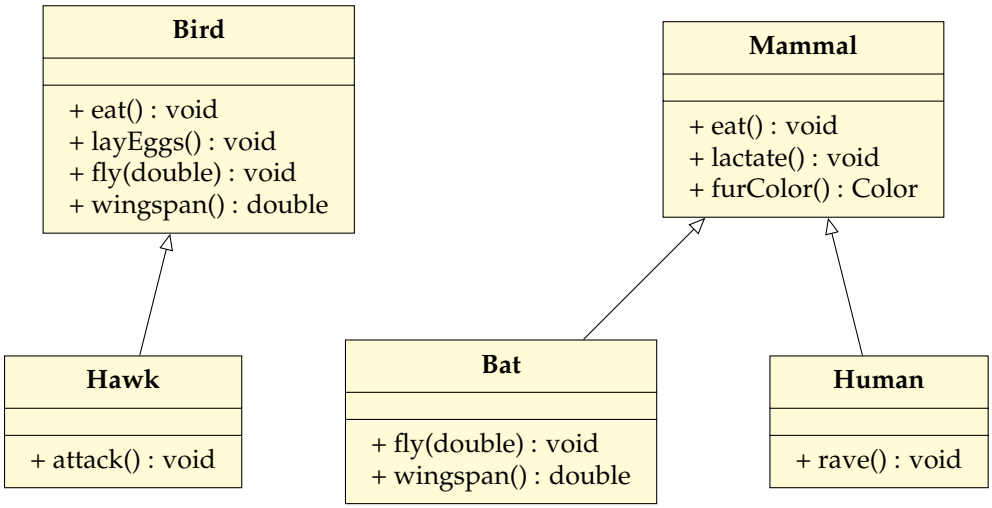
```java
/**
 * This method catches the exception, so Java considers it to be safe
 */
public static void yellFirstLine(String fileName) {
    try {
        String first = readFirstLine(fileName);
        System.out.println(first.toUpperCase() + "!!!");

    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("I HAVE NO TEXT, AND I MUST SCREAM!!!");
    }
}
```

Figure 3: This method catches the error thrown in the previous example

Interfaces

You've already worked with abstract classes, and have seen how they can help with the high level design of your program. If two different classes have a similar structure, then an abstract class can

join them together with a common type. This works great when the two classes you are joining logically belong to the same category of objects, but that is not always the case! What would you do if your program were set up like this:



Presumably there is code in `Bird.java` that you want to share with `Bat.java`. You want to say that a Bat is *like a* Bird, because it can `fly()` and has a `wingspan()`, but there are a few reasons why it would be wrong to say `class Bat extends Bird`:

- A Bat *is a* Mammal, not a Bird.
- A Bat cannot `layEggs()`, but if it inherited from Bird then it would get that method as well.
- Mammal and Bird both implement `eat()`, possibly in two very different ways! If you could inherit both, then it would be totally unclear which method `bat.eat()` should run.

At this point you may want to give up, and just rewrite `fly()` and `wingspan()` from scratch in `Bat.java`. But repeating yourself is never the answer [1]! In this case, we can use *interfaces* to achieve the flexibility that you can't get out of inheritance.

In Java, an interface is little more than a list of methods to implement. When you declare that a new class `implements` an interface, it sets up a contract guaranteeing that those methods will be implemented in that class. Interfaces are similar to abstract classes, except they are more flexible because they give the Java compiler fewer rules to enforce.

[1] Don't Repeat Yourself (DRY) is one of the main principles in software engineering. DRY is the opposite of WET, which stands for Write Everything Twice!

```java
public interface Flying {
    public void fly(double distance);
    public double wingspan();
}
```

```java
public class Bird implements Flying {
    public void fly(double distance) {
        ...
    }

    public double wingspan() {
        ...
    }
}
```

```java
public class Bat implements Flying {
    public void fly(double distance) {
        ...
    }

    public double wingspan() {
        ...
    }
}
```

Figure 4: Syntax examples for the keywords `interface` and `implements`

Since an interface is just a set of method names, there is no reason why a class can't implement *multiple* interfaces. You can declare a class that satisfies many interfaces like so:

```
public class Bat implements Flying, Eating, Hunting { ... }
```

This is only possible with interfaces because they do not contain any actual code. Even if the interfaces `Eating` and `Hunting` both declare a method called `eat()`, there is no confusion about what `Bat` needs to do. Both interfaces are telling it to do the exact same thing – implement its own `eat()` method!

Using Interfaces

An interface defines a *pseudo* type in Java. You can declare a variable with an interface instead of a type. However, since interfaces don't contain any code, an interface can't have it's own constructor. So even with an interface type, you still need an actual class in order to create a `new` object.

In short, this is okay:

```
Flying critter = new Bat();
critter = new Bird();
```

But this is illegal:

```
Flying critter = new Flying();
Bat bat = new Flying();
```

Common Interfaces

The Java standard library includes many built-in interfaces that are worth knowing about. You don't need to be an expert on any of these right now, they are just useful to be aware of.

Comparable and Comparator

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

public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

This interface is used by classes that can define an *ordering* on their objects. This is necessary if you want to be able to *sort* a collection of objects. To implement `Comparable`, you only need to write a `compareTo()` method that returns:

- A negative `int` if the current object is less than `other`
- A positive `int` if the current object is greater than `other`
- `0` if the two objects are equal

`Comparator` is almost exactly the same, except its `compare()` method takes two arguments instead of one. So with a `Comparator` you would use `comparator.compare(a, b)` instead of `a.compareTo(b)`. `Comparators` are generally small objects that exist only to sort two things for another type. This ends up being more useful in cases when data can be sorted in multiple different ways.

As the documentation for these interfaces [2] [3] points out, there are additional constraints to keep in

[2] `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`
[3] `https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html`

mind when implementing these interfaces. In particular, a correct implementation will make sure the ordering it defines has these three properties:

- **Reflexivity:** `a.compareTo(a)` should always return `0`.
- **Antisymmetry:** If `a.compareTo(b) < 0` then `b.compareTo(a) > 0`, and vice versa.
- **Transitivity:** If `a.compareTo(b) < 0` and `b.compareTo(c) < 0`, then `a.compareTo(c) < 0`.

This isn't required, in the sense that the compiler won't catch you if the method is written incorrectly. But for your code to make sense and work as expected with other Java libraries, `compare()` and `compareTo()` should follow these rules.

Lambda Expressions

Java 8 added support for a new feature called *lambda expressions* [4]. Lambdas are a popular construct borrowed from functional programming. They allow you to create small functions on the fly, without having to declare a method or give it a name. This is concept is a bit alien in object oriented programming, which usually treats all code as being *secondary* to the data it manipulates. Lambdas take the exact opposite approach, and allow you to treat the code itself as if it were an object in the program.

Let's say you wanted to sort a list of students by their names in alphabetical order. You could write a method to do this, but it could end up being very large. A much more elegant solution would be to use a *lamda expression* and the tools Java already has. The basic syntax for a lambda function is:

```
(Type1 arg1, Type2 arg2) -> { code(); return result; }
```

Essentially, you are creating a function on one line that is given an alias it can be called by. In the example of comparing students by their names in alphabetical order, we would use the following syntax:

```
1 Comparator<Student> compareByName = (Student x, Student y) ->
2     (x.getName().compareTo(y.getName));
```

Here, we're using a lamda expression to create a function with two parameters, x and y, which are both instances of some *Student* class defined in the program. It then gets the *name* attribute of x, and uses Java's *compareTo* method to compare it to the *name* attribute of y. This function can be referred to in the program with the name *compareByName*. In order to sort a whole list of students, we could use Java's Collection's *sort* method, and simply pass in the name of our lamda expression into that method as the function like so:

[4] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

```
Comparator<Student> compareByName = (Student x, Student y) -> {
    x.getName().compareTo(y.getName());
};

Collections.sort(listOfStudents, compareByName);
```

Figure 5: Creating a `Comparator` with a lambda expression

This means Collections will call the previously defined compareByName lamda expression, which as we defined before compares the *name* attributes of two Student objects, as its comparing function when sorting a list of Student objects.

Iterable and Iterator

```
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

```
public interface Iterator<T> {
    public T next();
    public boolean hasNext();
}
```

The Iterable interface is a bit more complicated. A class implements `Iterable` if it a method to return an `Iterator`. The `Iterator` then has two methods, `hasNext()` and `next()`, which allow you to read items in a collection one at a time.

The reason `Iterable` is interesting is that they are used behind the scenes in Java's extended `for` loops. So these two blocks of code have the exact same meaning in Java:

```
for (Item e : collection) {
    ...
}
```

```
for (Iterator<Item> iter = collection.iterator(); iter.hasNext();) {
    Item e = iter.next();
    ...
}
```

Cloneable

```
public interface Cloneable {
    public Object clone();
}
```

The `Cloneable` interface is interesting because, technically, *every* Java object implements this interface. The `clone()` method is implemented in `Object`, and since every class extends `Object` that means every class inherits that code.

The particular caveat here is that not every object can *use* `clone()`. You have to explicitly add `implements Cloneable` to the class declaration in order to use it. However, if you try to use `clone()` *without* explicitly implementing the interface, then your program will throw a `CloneNotSupportedException`!

For classes that implement `Cloneable`, running `instance.clone()` will create a *shallow* copy of the object in question. That means Java only copies a *reference* of each variable within the object, rather than also cloning the variables themselves. If you want to do a *deep* copy of an object, you'll need to override the `clone()` method. That also also comes with a few caveats:

```
public class Student implements Cloneable {
    private String name;
    private Date birthday;

    public Student clone() throws CloneNotSupportedException {
        Student copy = (Student) super.clone();
        copy.birthday = (Date) birthday.clone();

        return copy;
    }
}
```

Figure 6: How to override `Object.clone()`

Since we want the copied student's `birthday` to refer to a *different* `Date` object than the original, this implementation needs to `clone()` it separately. `clone()` returns value is of the type `Object`, which means it is necessary to explicitly cast the results to the correct type whenever you use it. Finally, either call could theoretically throw a `CloneNotSupportedException`, so we need to handle that by adding a `throws` declaration to the method signature.

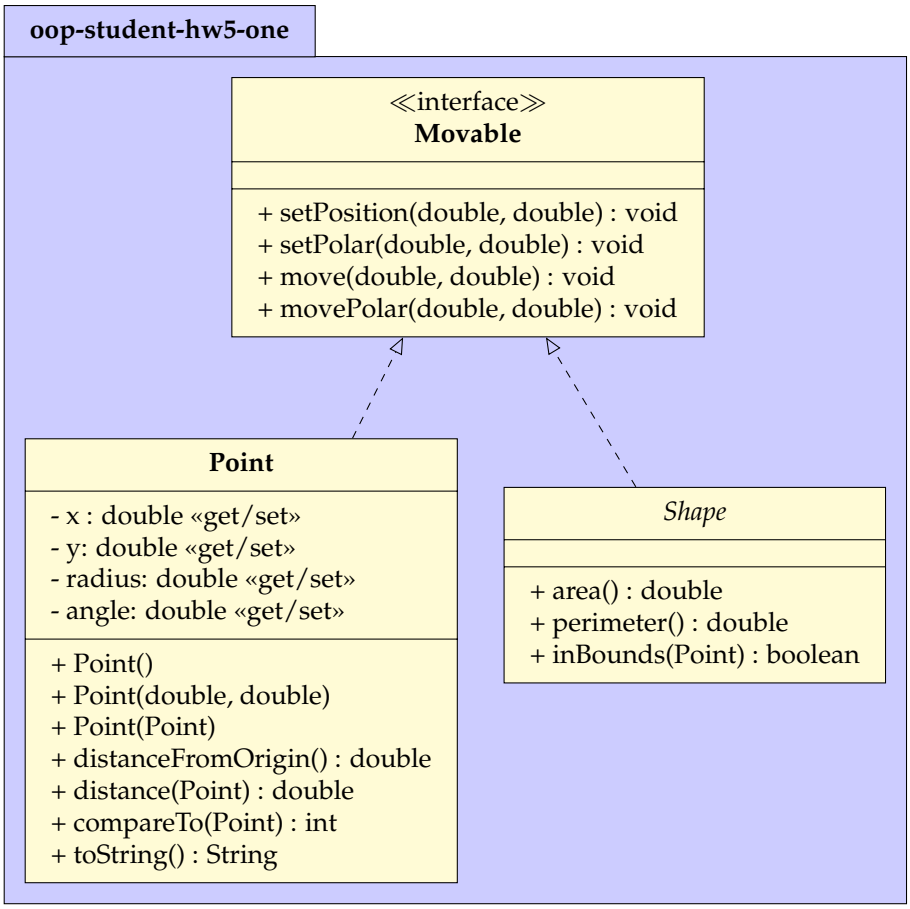
`Cloneable` is possibly the trickiest interface in Java, so it's worthwhile to learn how to use it correctly.

Problems

Problem 1: Even More Geometry!

Movable

Create a copy of the latest version of the geometry package, which you last visited in homework 3. Add a new interface called `Movable.java` with the following methods, and change `Shape.java` and `Point.java` so that they implement this interface:



For implementing `Movable` in `Point.java` is simple:

- `setPosition(x, y)` uses `Point`'s setter methods to move it to the new position, (x, y)
- `setPolar(radius, angle)` uses the setter methods to move the point to the new polar coordinate, (r, θ)
- `move(dx, dy)` uses the getters *and* setters to move the point to the position $(x + \delta x, y + \delta y)$
- `movePolar(radius, angle)` uses the getters and setters to move the point to the new polar coordinate position $(r + \delta r, \theta + \delta\theta)$

Since `Shape` is an abstract class, it doesn't need to implement these methods itself. However, this will require `Rectangle` and `Circle` to implement the interface.

`Rectangle.java` should implement `Moveable` by applying `setPosition()`, `setPolar()`, and `move()` to its lower left corner. You also need to move the upper right corner by the same amount, so that it remains in the same position relative to the lower left.

`Circle.java` is even more simple to implement – simply apply the methods to the center point.

Their subclasses `Parallelogram` and `Ellipse` have the exact same implementation of `Moveable`; which means they don't need to be modified at all! They will inherit the correct implementations from their parents.

Cloning

Also add the interface `Cloneable` to both `Point` and `Shape`. Since `Point` contains only primitive types, it does not need a custom implementation of `clone()`.

For both `Rectangle.java` and `Circle.java`, write the `clone()` method so that it does a *deep* copy. When you clone a shape object, it should also clone new copies of any `Points` within that object. This way moving a point in the new copy will not change the position of the original shape. You can see an example of how to do this in Figure 6.

Comparators

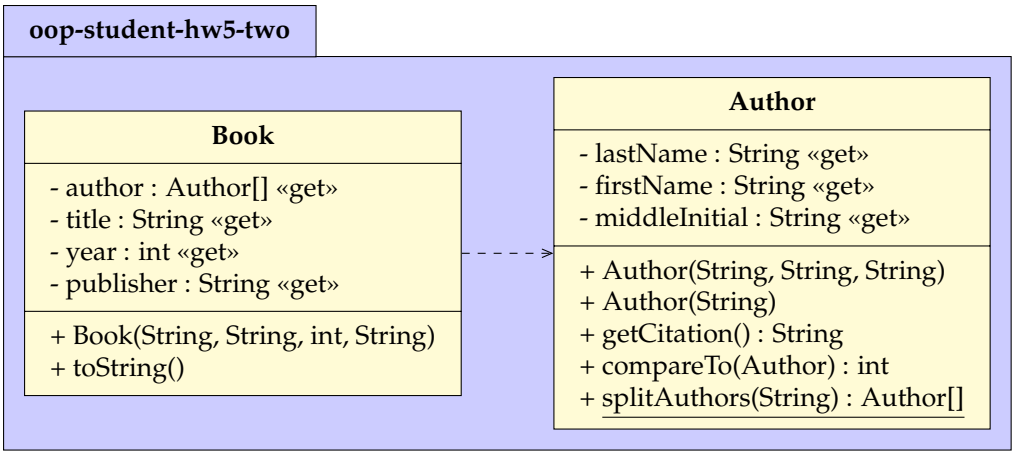
Add two new classes, `ShapeAreaComparator.java` and `ShapePerimeterComparator.java`, both of which implement the interface `Comparator<Shape>`.

The first class implements a `compare()` method that takes two Shapes, and returns `-1`, `0`, or `1` if the `area()` of the first shape is less than, equal to, or greater than that of the second shape, (respectively).

The second class does the same thing, but compares the shapes based on their `perimeter()` instead of their `area()`.

Problem 2: Sorting Books

In Homework 3 you wrote a program to manage different types of citations. To get started with this problem, you can use some of your code from `Book.java` and `Author.java` to create a simplified version of the Citations package:



Author

First modify `Author.java` so that it `implements` the interface `Comparable<Author>`. The `compareTo()` method has the following behavior:

- If `this.lastName` comes before `other.lastName` alphabetically, return `-1`; if the opposite is true, return `1`.
- If the last names are equal, fall back to doing the same comparison on the `firstName`.
- If both the last and first names are equal, return the result of comparing the middle initials.

Library

Write a program called `Library.java` to sort an `ArrayList` of Books. First, download the file `library.txt`, which contains a small list of classic novels in no particular order. The format of the file is:

```
<author>, <title>, <year>
<publisher>
<author>, <title>, <year>
<publisher>
...
```

Each book has exactly one author, and no title has a `","` in its name, so parsing should be fairly straightforward.

In the `main()` method, start by prompting the user for the name of a file to read from. If the file cannot be opened for reading, print a warning and ask the user to enter a valid file name. Repeat this until `new FileReader()` succeeds without throwing an exception (you can use a `try/catch` block inside a `while` loop).

Read the list line by line, and create a new `Book` for every two lines in the file. Store each book in an `ArrayList` as you go.

Next, prompt the user to decide whether the library should be sorted by author, title, or year. Use `Collections.sort()` to sort the array, using a different `Comparator` depending on the user's response. You can use lambda expressions more easily create the different Comparators.

Finally, iterate over the sorted array list and print out the citation for each book, using the same `toString()` method you wrote in homework 3.

Example output:

```
Enter file name for reading list: library.docx
Error! Cannot open file: library.docx

Enter file name for reading list: library.txt
Sort collection by [author/title/year]: year

Cervantes, M. D., Don Quixote., 1605
Francisco de Robles

Shakespeare, W., Macbeth., 1606
William and Isaac Jaggard

Swift, J., Gulliver's Travels., 1726
Benjamin Motte

Austen, J., Sense and Sensibility., 1811
Thomas Egerton

...
```

Submission

Make sure that you have Javadoc style comments for every class and method in your source code, as described in Homework 0. Document any unresolved bugs in the Javadoc comments for each of your classes. When you are satisfied that your code is complete, create a TAR file containing all of the source code for this assignment called:

`cse213_<firstname>_<lastname>_hw5.tar.gz`

Upload your submission to Canvas before the due date.