Chapter 3: Relationships between Classes

3.1. Inheritance

You may recall that we could inherit methods from another class in Python as follows:

```
class Child(Parent):
    ...
```

The syntax above defines class `Child` to be a subclass of class `Parent`.

In Java, inheritance works in a similar manner. However, instead of using brackets, we use the `extends` keyword:

```
class Child extends Parent {
    ...
}
```

This means that class `Child` inherits all the methods and variables defined in `Parent`, and that `Child` is an instance of `Parent`.

Throughout this chapter, we will be using the terms parent class/child class and superclass/subclass interchangeably.

3.1.1. Abstract classes

Abstract classes are classes that are not meant to be initialized directly. In Python, we signified a method was abstract by having a method that would raise a `NotImplementedError`. Any non-abstract child class would then have to implement this method.

In Java, we use the `abstract` keyword to signify that a class is abstract: this enforces the fact that no instance of the class should be created, even if there are no abstract methods in the class! We also use the `abstract` keyword to indicate any abstract methods.

```
abstract class AbstractClass{
    abstract void something();    // Abstract methods have no body!
}
```

Any non-abstract class that extends an abstract class then has to implement the body of *all* abstract methods.

```
class NonAbstract extends AbstractClass{
    void something(){
        ...   // Method body here!
    }
}
```

3.1.2. Overriding methods

In Python, we could override a parent class' methods by redefining it. In Java, we do the same thing, but we also include an `@Override` annotation. This informs the compiler that method is meant to override a method in a superclass. While the annotation is not required, including it helps us prevent errors (e.g., misspelling the name of a method, forgetting a parameter, etc.).

For example, if we have the following parent class:

```java
class Parent {
    void something(){
        ...    // Some method body here!
    }
}
```

We can override the method `something` as follows:

```java
class Child extends Parent {
    @Override
    void something(){
        ...    // Our new method body here!
    }

}
```

3.2. Interfaces

In Java, you can only extend a single class: you have one parent class, and that's it! However, sometimes we want to describe more behaviours for a class in a way that just one parent won't suffice.

For example, suppose we're writing a program to simulate plants. We would have a class called `Plant`: all `Plant` objects are able to `breathe` and `grow`. We could also have subclasses such as `Wheat` and `Flower` with their own subclasses. However, suppose we want to indicate that some plants are edible for humans: for instance `Corn` could have an `eat` method, and so could `Basil`. Not all plants are edible, so we can't add that method to `Plant`. We *could* define an `EdiblePlant` class, but then we would also need `EdibleWheat`, `EdibleFlower`, and so on: this isn't a very clean solution. Interfaces allow for an alternative design.

An **interface** in Java defines a contract for what a class can do, without specifying how it does it. It's used to define shared behavior across potentially unrelated classes. This is similar to an abstract class, but fundamentally differs in that a class can implement any number of interfaces.

In an interface:

- All methods are implicitly `public` and **abstract**, unless marked otherwise. You can try defining a method in your own interface and see what IntelliJ warns you about if you include various keywords.
- Variables in an interface are implicitly `public`, `static`, and `final`. You can not have instance variables for an interface — try it in IntelliJ and see what happens.
- Classes that implement an interface must provide implementations for its abstract methods.

Originally, interfaces in Java could only contain abstract methods. However, since **Java 8**, interfaces can also include:

- `default` methods — methods with a body, allowing backward-compatible enhancements without requiring updates to any classes already implementing an existing interface.
- `static` methods — useful for utility behavior related to the interface.

For our example, we could define an `Edible` interface such as:

```
interface Edible {
    // Method that must be implemented by any class implementing Edible
    void eat(); // note that we don't include the abstract or public
keywords!
}
```

And to define a `Washable` interface with a `default` method `wash`, we must use the keyword `default` when defining it:

```
interface Washable {
    // Default method that provides a basic implementation
    default void wash() {
        System.out.println("Washing the edible item...");
    }
}
```

And to use both interfaces, we would use the `implements` keyword:

```
class Corn extends Plant implements Edible, Washable {
    void eat() {
        ...    // Our implementation here!
    }

    // Overriding the default wash method
    @Override
    public void wash() {
        System.out.println("Thoroughly washing the corn...");
    }
}
```

If a class implements multiple interfaces, they are separated by commas after the `implements` keyword.

We can implement as many interfaces as we want! In addition, interfaces can also `extend` other interfaces.

As an example, some food can be steamed so we might want a `Steamable` interface. These are also edible, so we could do the following:

```
interface Steamable extends Edible {
    void steam();
}
```

Any class that `implements Steamable` must then have both a `steam` and `eat` method!

3.3. super

In Python, we could use `super()` to refer to methods in the parent class. For instance, we could use `super().__init__()` to call the parent constructor or `super().method()` to call the parent's method.

In Java, we have the `super` keyword that functions in a similar way. If we want to call a parent's constructor, we use `super()`, or `super(a, b, c)` if we needed to pass in some parameters. If we wanted to call a parent's method, we would use `super.method()`!

Note the difference between Python and Java: `super()` is used in Python and has brackets while `super` is used in Java with no brackets!

3.3.1. Constructors with super

When extending another class, Java *requires* a call to a constructor of its superclass to be made in the constructor of the subclass. Furthermore, this call *must* be the very first thing done. If no constructor call is explicitly made in the subclass constructor, then an *implicit* call to `super()` will be made.

For instance, this code:

```
class Child extends Parent {
    int attribute1;
    int attribute2;

    public Child(int a, int b) {
        // There's no super() call here, but it's implicit!
        this.attribute1 = a;
        this.attribute2 = b;
    }
}
```

Would be equivalent to:

```java
class Child extends Parent {
    int attribute1;
    int attribute2;

    public Child(int a, int b) {
        super();  // What happens if Parent has no empty constructor?
        this.attribute1 = a;
        this.attribute2 = b;
    }
}
```

If `Parent` didn't have a constructor that takes no arguments, then an error would be raised during compilation. The only time Java will implicitly call the parent constructor is when the only constructor that exists in the parent class takes no arguments. That is, when there is no ambiguity about our intention. When developing code in IntelliJ, you will see helpful error messages when developing a constructor for your own subclasses.

If the parent class has a constructor taking no arguments, then we can even omit a constructor in the subclass if we don't need to perform any additional information. Java will implicitly call the parent constructor when creating a new instance of our subclass.

3.4. Polymorphism

Polymorphism is the ability of an object to take on many forms. In Java, this means that an object can be treated as an instance of its own class, any superclass, or any interface it implements.

For example, if we had the following code:

```java
class Dog extends Canine implements Domesticatable { ... }
```

A Dog object is:

- a `Dog`
- a `Canine`
- possibly an `Animal` (if `Canine` extends `Animal`)
- an `Object` (recall `Object` will be at the top of the hierarchy)
- a `Domesticatable` (since `Dog` implements this interface)

This means a Dog instance will pass `instanceof` checks for *all* of these types:

```java
Dog d = new Dog();
System.out.println(d instanceof Dog);             // true
System.out.println(d instanceof Canine);          // true
System.out.println(d instanceof Animal);          // true
System.out.println(d instanceof Object);          // true
System.out.println(d instanceof Domesticatable);  // true
```

> **Note:** `instanceof` in Java is not strictly limited to parent/child class relationships.
> An object is considered an instance of a class or interface if it can be **safely cast** to that type.
> This includes interfaces — any object that implements an interface will pass `instanceof` checks for that interface.

Polymorphism enables flexible and reusable code by allowing objects to be treated according to their capabilities rather than their exact types.

As an example, the following would exhibit polymorphism:

```
Animal[] animals = {new Cat(), new Dog(), new Axolotl()};

for (Animal a : animals){
    a.eat();    // 'a' in this line of code can have various types!
}
```

Even though we have multiple types of objects, we can deal with them in a uniform manner. If we didn't have polymorphism here, we wouldn't be able to have such a simple and clean loop! Much of design comes down to defining the right abstractions to allow code to be written in a way that is flexible, extensible, and easy to reason about.

By programming to an interface or superclass, rather than a specific implementation, we enable our code to work with a wide variety of objects that share common behavior. This not only reduces duplication, but also makes it easier to introduce new types without changing existing logic — a key principle of maintainable software design.

3.5. Casting

As we have learned, unlike in Python, all variables in Java have a declared type. Often, we'll want to convert between types: sometimes a more general superclass is better for one situation, while a subclass is better in another.

Casting is when we explicitly change the type of a reference to another, usually to access more specific functionality that isn't available in the more general type. For example, if we have a reference of type `Animal`, but we know it's actually referencing a `Dog` object, we can cast it to `Dog` to call methods that only exist in the `Dog` class.

```
Animal a = new Dog(); // a dog is an animal so we can make this assignment
((Dog) a).bark(); // casting to Dog to access bark()
```

Casting is safe only when the actual object is an instance of the target type. If not, it will throw a `ClassCastException` at runtime.

In the example above, we actually have two examples of casting. The first line is performing **upcasting**. Upcasting is when we assign a subclass object to a superclass reference. This is **implicit** and always safe, because every `Dog` is an `Animal`. The second line is performing **downcasting**. Downcasting is converting from a superclass to a subclass. It is only safe if the actual object is an instance of the subclass, and should be checked with `instanceof` to avoid runtime errors.

We can use `instanceof` to check before casting as follows:

```
if (a instanceof Dog) {
    ((Dog) a).bark();
}
```

This ensures that the cast is valid and avoids runtime errors.

3.5.1. Primitive conversions

We can also cast some primitives, such as converting an `int` into a `double` and vice versa:

```
int x = 1;
double y = 1.1;
double double_x = (double) x;
int int_y = (int) y;
```

When we cast **objects**, we're essentially "re-labeling" the reference type — we're telling the compiler to treat the object as a different type, without changing the object itself. The actual data remains the same.

However, when we cast **primitives**, we're converting the value itself, which may result in **loss of precision** or **truncation**. For example, casting a `double` to an `int` removes the decimal part:

```
double y = 3.7;
int truncated = (int) y; // truncated to 3
```

This kind of change is **irreversible** in the sense that the original precision is lost — you can't recover the `.7` from the `int` without referring back to the original `y`. Of course, the variable `y` still refers to 3.7.

Primitive casting can be **implicit** (e.g., `int` to `double`) or **explicit** (e.g., `double` to `int`), depending on whether there's a risk of data loss.

3.6. Comparable

3.6.1. Being comparable enables sorting

In Python, we had the `list.sort()` method and `sorted()` function to allow us to sort lists. Similarly, Java provides a `sort` method capable of sorting an array of `int` values or of any other primitive type. In fact, `sort` is overloaded: there is a series of `sort` methods, each one capable of handling one of the primitive types. They are all defined as static methods in the `Arrays` class.

Here we use the `sort` method that takes an array of `int` values:

```
int[] ages = {10, 24, 3, 45, 83, 9};
Arrays.sort(ages);
```

Great! But what if we want to sort objects of some other type? For instance, Java provides a class called `MonthDay` that can keep track of the month and day parts of a date. (It, and many other classes related to time, are defined in `java.time`.) What if we wanted to sort an array of `MonthDay` objects? And what if we wanted to also sort an array of `File` objects and an array of `Double` objects? The designers of Java could have made a general sort method that accepts an array of `Object`, like:

```
public static void sort(Object[] a)
```

But in order for `sort` to do its job, it must be able to compare the elements of the array to decide on their order. It can't simply use operators like `<` to compare two instances of `MonthDay`. These operators will not accept instances of `MonthDay` as operands.

Instead, the `sort` method requires that all elements in the array must implement the `Comparable` interface. This interface in turn requires any class that implements it to define this method:

```
int compareTo(T o)
    Compares this object with the specified object for order.
    Returns a negative integer, zero, or a positive integer
    as this object is less than, equal to, or greater than
    the specified object.
```

In Python, we had a similar concept: if we implemented the `__lt__` method, we could compare and sort our objects! Java is just slightly different, where we return a negative integer, zero, or positive integer instead of `True` or `False`.

Returning to our `MonthDay` example: the authors of the `MonthDay` class define this method and declare that the class **implements** the `Comparable` interface. This means that if we pass an array of `MonthDay` objects to sort, it is *guaranteed* to be able to use `compareTo` to put them in order. Here is an example of code that takes advantage of this capability:

```
public static void main(String[] args) {
    // Create some MonthDay objects and put them in an array.
    // This class does not provide any constructors. Instead, we
    // call static method "of" to make an instance.
    MonthDay md1 = MonthDay.of(1, 5);
    MonthDay md2 = MonthDay.of(7, 24);
    MonthDay md3 = MonthDay.of(7, 24);
    MonthDay md4 = MonthDay.of(1, 28);
    MonthDay md5 = MonthDay.of(2, 14);
    MonthDay[] dates = {md1, md2, md3, md4, md5};

    // Because MonthDay implements Comparable, we can call sort,
    // which depends on that:
    Arrays.sort(dates);
}
```

The sort method has another requirement: all elements in the array must be **mutually** comparable. This prevents us from trying to sort an array with a mixture of `DateTime` objects and `File` objects, for instance. These objects are comparable *within* each class, but not *across* classes (unless the classes which the objects are instances of share a parent class implementing `Comparable`).

3.6.2. Being comparable enables comparisons

If we ever wish simply to compare a MonthDay to any other MonthDay, we can do this as well:

```
        System.out.println(md1.compareTo(md2)); // -6
        System.out.println(md2.compareTo(md1)); //  6
        System.out.println(md2.compareTo(md3)); //  0
```

The same holds for any class that implements `Comparable`, which includes many built-in classes, such as `String`, `File`, `Integer`, and `Double`.

3.6.3. Making our own classes Comparable

Consider this class:

```
class Review {
    /**
     * A review, for example of a book or movie.
     */

    // === Class Variables ===

    // The name of the item that this Review is about.
    String item;
    // The numeric rating, between 0 and 100, associated with this Review.
    private int rating;
    // The written component of this review.
    private String text;
    // The number of likes that this review has received.
    private int likes;

    public Review(String item, int rating, String text) {
        this.item = item;
        this.rating = rating;
        this.text = text;
        this.likes = 0;
    }

    public String toString() {
        return this.item + " (" + this.rating + "): " +
            this.text + "; likes = " + this.likes;
    }

    /**
     * Records a like for this Review.
     */
    public void like() {
        this.likes += 1;
    }
}
```

To make instances of this class comparable, we need to do two things. First, we must implement the `compareTo` method. It's up to us to decide how two reviews should be compared. Here's one possible implementation, based on ratings:

```
    /**
     * Compares this object with the specified object for order.
     *
     * Returns a negative integer, zero, or a positive integer as this
     * object is less than, equal to, or greater than the specified
object.
     *
     * @param other the object to be compared.
     * @return a negative integer, zero, or a positive integer as this
     * object is less than, equal to, or greater than the specified
object.
     */
    @Override
    public int compareTo(Review other) {
        if (this.rating < other.rating) {
            return -1;
        } else if (this.rating > other.rating) {
            return 1;
        } else {
            return 0;
        }
    }
```

Second, we must change the class declaration to indicate that we have fulfilled the requirements of implementing the `Comparable` interface:

```
class Review implements Comparable<Review> {
```

Notice that we wrote `Comparable<Review>` rather than just `Comparable`. The `Comparable` interface is specified using generics, which we will discuss in more detail later. For now, we just need to know that we need to specify what kinds of objects we are declaring that we can compare with. We want to be able to compare instances of our `Review` class with other `Review` objects, so we use this syntax.

Now that we have implemented `Comparable`, we can do the same sorts of things we did with `MonthDays`:

```
public static void main(String[] args) {
    Review r1 = new Review("Emoji Movie", 10,
        "Cinematic malware");
    Review r2 = new Review("Dunkirk", 95,
        "Gifted ensemble cast and masterful direction");
    Review r3 = new Review("Spider Man: Homecoming", 95,
        "A fun adventure");
    Review r4 = new Review("My Neighbour Totoro", 99,
        "A work of art");
    Review r5 = new Review("Despicable Me 3", 60,
        "Zany but scattershot humour");

    System.out.println(r1.compareTo(r2)); // -1
    System.out.println(r2.compareTo(r1)); // 1
    System.out.println(r2.compareTo(r3)); // 0

    Review[] badFruit = {r1, r2, r3, r4, r5};
    Arrays.sort(badFruit);
    for (int i = 0; i < badFruit.length; i++) {
        System.out.println(badFruit[i]);
    }

}
```

The reviews come out sorted by rating:

```
Emoji Movie (10): Cinematic malware; likes = 0
Despicable Me 3 (60): Zany but scattershot humour; likes = 0
Dunkirk (95): Gifted ensemble cast and masterful direction; likes = 0
Spider Man: Homecoming (95): A fun adventure; likes = 0
My Neighbour Totoro (99): A work of art; likes = 0
```

3.6.4. Being comparable enables more

In addition to enabling sorting, a class that implements `Comparable` can be used in certain "Collections" that care about order, such as `SortedSet`. You will learn about Collections in later readings.

3.7. Comparator

What if we want to be able to choose between ordering reviews according to their rating, the length of their text or the number of likes they have? There can only be one `compareTo` method in the class. Or what if we want to make instances of `MonthDay` comparable according to just the month, so February 14th and

February 20th would be considered tied? We didn't write this class, so we can't change what its `compareTo` does.

We can accomplish these goals by defining a "comparator" class for each kind of comparison we want. A "comparator" class implements the `Comparator` interface, which requires this method:

```
int compare(T o1, T o2)
    Compares its two arguments for order.
    Returns a negative integer, zero, or a positive integer
    as the first argument is less than, equal to, or greater
    than the second.
```

Here's an example of a `Comparator` that orders `Reviews` according to likes:

```
import java.util.Comparator;

class LikesComparator implements Comparator<Review> {
     /**
      * Compares its two arguments for order.
      *
      * Returns a negative integer, zero, or a positive integer
      * as r1 is less than, equal to, or greater than r2 in terms
      * of number of likes.
      *
      * @param r1 the first Review to compare
      * @param r2 the second Review to compare
      * @return a negative integer, zero, or a positive integer
      *      as r1 is less than, equal to, or greater than r2
      */
    @Override
    public int compare (Review r1, Review r2) {
        return r1.getLikes() - r2.getLikes();
    }
}
```

(Sidenote: The `compare` method needs to know the number of likes a review has received. Since this is stored in a private instance variable, we have added a getter method to provide access to it.)

Now we can use a version of `sort` that accepts a `Comparator` as a second argument, and uses it to determine how things are sorted. Here, we call it with our `LikesComparator`:

```
public static void main(String[] args) {
    Review[] freshVeg = {r1, r2, r3, r4, r5};
    // Let's add some likes so the sorting will be interesting.
    r1.like();
    r1.like();
    r1.like();
    r4.like();
```

```
        r3.like();
        Arrays.sort(freshVeg, new LikesComparator());
        for (int i = 0; i < freshVeg.length; i++) {
            System.out.println(freshVeg[i]);
        }
    }
```

The output produced is indeed in order according to likes:

```
Dunkirk (95): Gifted ensemble cast and masterful direction; likes = 0
Despicable Me 3 (60): Zany but scattershot humour; likes = 0
Spider Man: Homecoming (95): A fun adventure; likes = 1
My Neighbour Totoro (99): A work of art; likes = 1
Emoji Movie (10): Cinematic malware; likes = 3
```

We can define another `Comparator` that orders `Review` objects differently. This one does it by length:

```
import java.util.Comparator;


class TextComparator implements Comparator<Review> {

    /**
     * Compares its two arguments for order.
     *
     * Returns a negative integer, zero, or a positive integer
     * as r1 is less than, equal to, or greater than r2 in terms
     * of length of text.
     *
     * @param r1 the first Review to compare
     * @param r2 the second Review to compare
     * @return a negative integer, zero, or a positive integer
     *      as r1 is less than, equal to, or greater than r2
     */
    @Override
    public int compare (Review r1, Review r2) {
        return r1.getText().length() - r2.getText().length();
    }
}
```

(Here, we needed to add a getter for the `text` instance variable.)

Now we can write code that sorts `Review` objects using this `TextComparator` instead of the `LikesComparator`:

```
    public static void main(String[] args) {
        Review[] rottenVeg = {r1, r2, r3, r4, r5};
        r1.like();
        r1.like();
        r1.like();
        r4.like();
        r3.like();
        Arrays.sort(rottenVeg, new TextComparator());
        for (int i = 0; i < rottenVeg.length; i++) {
            System.out.println(rottenVeg[i]);
        }
    }
```

This output is in order according to the length of the review text:

```
My Neighbour Totoro (99): A work of art; likes = 2
Spider Man: Homecoming (95): A fun adventure; likes = 2
Emoji Movie (10): Cinematic malware; likes = 6
Despicable Me 3 (60): Zany but scattershot humour; likes = 0
Dunkirk (95): Gifted ensemble cast and masterful direction; likes = 0
```

3.7.1. When to use Comparable vs. Comparator?

If you are not the author of the class, you cannot make it `Comparable`. Your only option is to define one or more comparators.

If you are the author of the class, you have both options available to you!