Review of Why We Like Using Classes

Organizing code into classes provides ***encapsulation*** – the bundling together of closely related data, along with the methods that act on that data. For example a `Vector` class can keep together the coordinates associated with a vector along with operations that one might perform on vectors such as addition, cross product, etc.

Classes also allow for ***abstraction*** – hiding the implementation details while providing a simple interface. When working with the `String` class, we can simply get the upper case version of a string by calling the `toUpperCase` method, without concern about how to implement such a method.

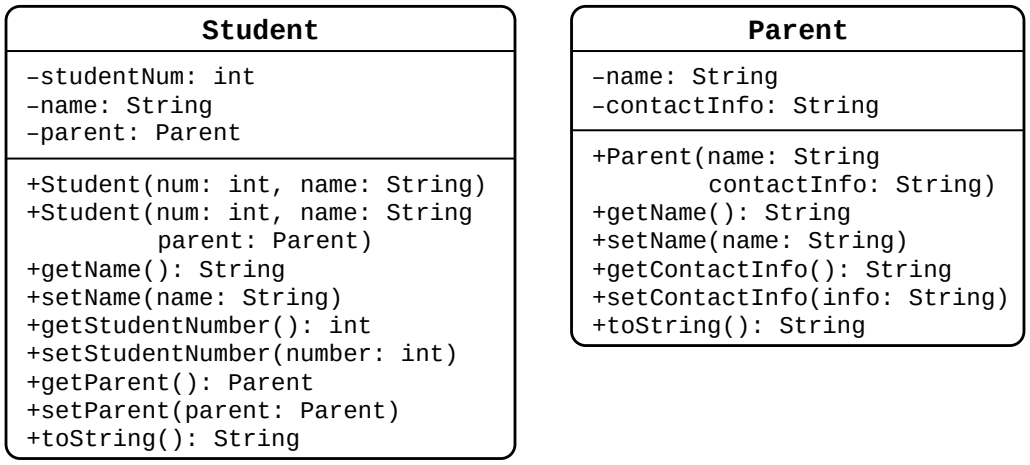
Inheritance for Code Re-Use

To understand what inheritance is and how it helps with code re-use, we will go through an example.

What do Students and Parents have in common?

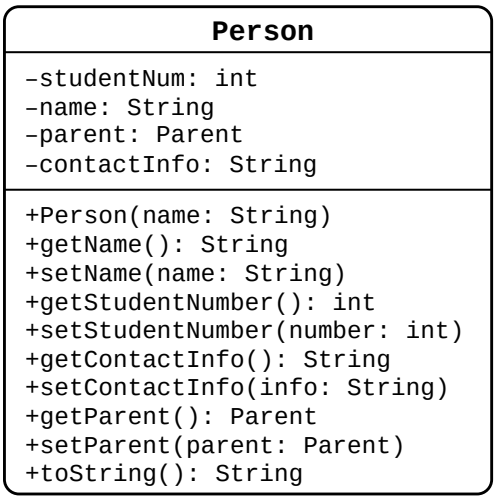
This section relies on the fact that both a student and a parent is a type of person.

We wish to write some software that will keep a registry of students for a school. To start, we wish to keep track of students and their parents. Each student will have a student number and name. Each parent will have a name and contact information. Since we will store different information for students and parents, we decide to make one object to represent students, and another to represent parents. Here is our initial UML diagram.

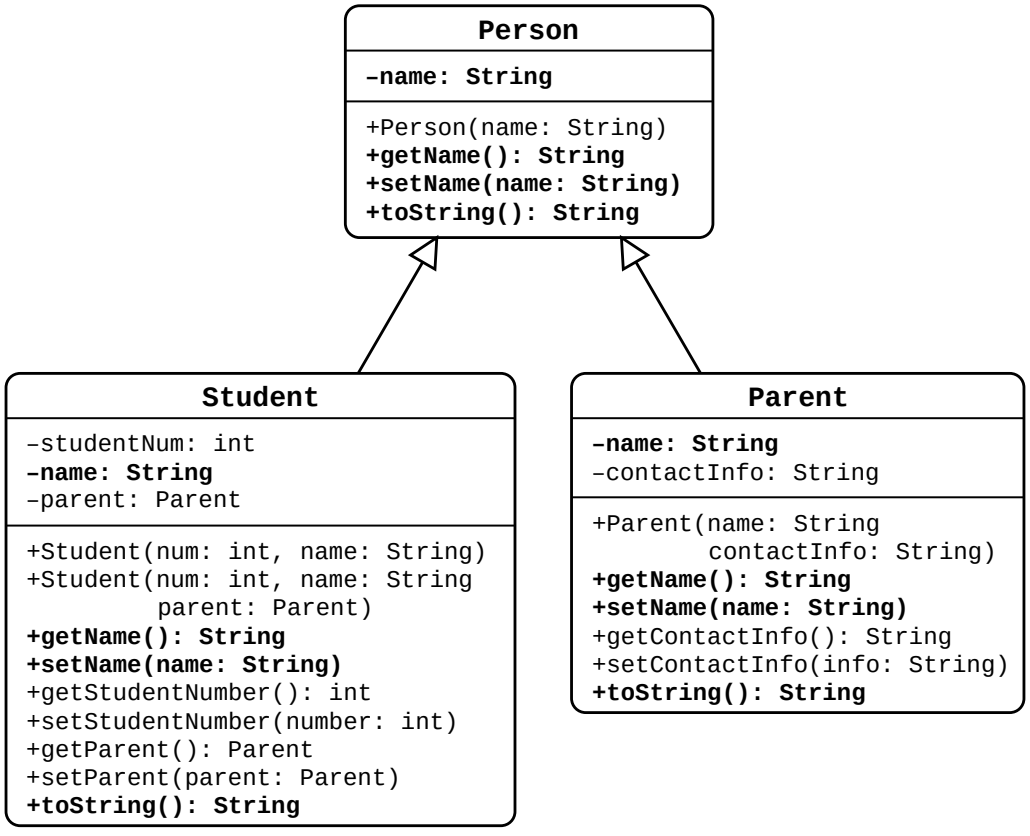


In our planning, we notice these two classes share some fields and methods. Both students and parents are people, so will share attributes that people have. In our simple example, we have only one shared attribute, `name`, and two associated methods, `getName` and `setName`. In more realistic cases, objects may share a number of attributes and methods. Hopefully you will have had it drilled into your brain that, if at all possible, you should avoid having multiple copies of the same code – in most cases that there's a better way.

Since students and parents are both people, perhaps we can try to make a single `Person` class and use it to store both students and parents. Merging the two classes above, we come up with the following UML diagram.



This new design is good in that we will not have duplicated code for saving and manipulating the `name` attribute. However, this new design is wasteful. We will not use the `studentNum` or `parent` fields nor their associated methods when we're storing the data for parents, and we will not use the `contactInfo` field nor its associated methods when we're storing the data for students.
So now we can see the problem that ***inheritance*** solves. Look at the UML diagram below.



We have created a `Person` class. In this class, we have a field for `name`, and code for the methods associated with it. The `Student` class and `Parent` class are ***subclasses*** of the `Person` class. Although the methods have been listed in the subclasses in the UML diagram, subclasses ***inherit*** all the fields and methods of the `Person` class, so that code does not need to be duplicated in the subclasses. We only need to implement the additional methods in the subclasses that have not been implemented in the parent class.

Document: Introduction to Inheritance

Implementing a ***superclass*** – a class that *subclasses* will inherit from – is no different from implementing any other class; nothing in particular needs to be done to a class to allow other classes to inherit from it. In fact, every class we write in Java is implicitly a subclass of the `Object` class, which defines a few basic methods such as `toString` and `equals`, and their default implementations, that we may want to *override*.

To implement a subclass, use the Java keyword `extends`, and we say the subclass `extends` the superclass. The class declaration for the `Parent` class is as follows:

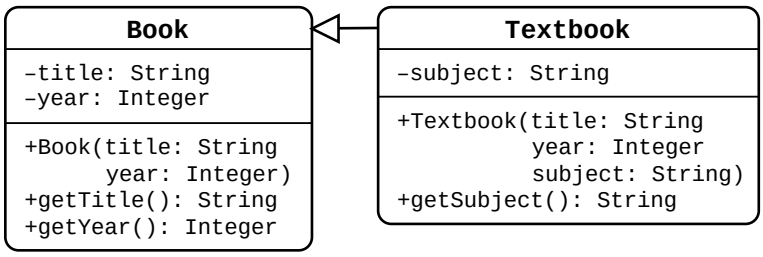
```
public class Parent extends Person
```

It should not be a surprise that the class declaration of the `Student` class also follows this pattern.

Subclass Constructors

Recall that if no constructor is declared for a class, the Java compiler will automatically provide a default constructor that has no parameters and initializes the fields defined in the class. Although we did not mention it previously, it also calls its parent class zero-parameter constructor. Thus if the parent class does not have a zero-parameter constructor, the default constructor cannot be used, and a constructor must be explicitly declared.

To make this more clear, let us look at an example. Consider the following class diagram. In these diagrams, the inherited methods are not included in the child (`Textbook`) class.



If the `Textbook` class were defined without a constructor, the default constructor that would be supplied by the compiler would be equivalent to the code for the *Default Zero-Parameter Textbook Constructor* to the right. In line 2 of this constructor, the keyword `super` followed immediately by parentheses is a call to the parent class constructor.

Default Zero-Parameter Textbook Constructor

```
1 public Textbook() {
2    super();
3    this.subject = null;
4 }
```

However, as we can see in the UML class diagram above, the `Book` class (the parent class of `Textbook`) does not have a zero-parameter constructor. Thus, this default constructor will fail compilation. To resolve this, we must declare an explicit constructor for the `Textbook` class that explicitly calls the `Book` class constructor, again using the `super` keyword. This constructor might look like the one given in the code box labeled *Parameterized Textbook Constructor*.

Parameterized Textbook Constructor

```
1 public Textbook(String title,
2                 int year,
3                 String subject) {
4    super(title, year);
5    this.subject = subject;
6 }
```

Important note: the call to the super constructor must be the first statement of the constructor. If no call is provided, the default zero-parameter constructor will be automatically called, and no call to a superclass constructor can subsequently be made.

Notice that, according tot he UML diagram, the fields in the `Book` class have been declared as `private`. As you have learned, `private` fields cannot be accessed outside of the defining class. If there is a case where a subclass will need to modify a superclass field directly, but the field does not have a good reason to be accessible to other classes, the field can be declared as `protected`. A `protected` field cannot be accessed outside the class by any class that is not a subclass.

Calling Instance Methods From the Superclass

Calling a `static` method that exists in a superclass is no different than calling a `static` method in any other class. We simply use the class name, a period character (`.`), then the method name. For example, if a subclass were to extended the `Math` class and needed to call the `random` method, `Math.random()` would still be used in the subclass.

In order to call an instance method in a subclass, the `super` keyword is used. We will re-use the example of a `Textbook` class and its parent class `Book` from the previous section. Examine the code for a `toString` method for the `Book` class (above right). It will print the book title on one line, and the publication year on the following line.

Book Class toString Method

```
1  @Override
2  public String toString() {
3     return "Title: " + title + "\n" +
4            "Year: " + year + "\n";
   }
```

Textbook Class toString Method

```
1  @Override
2  public String toString() {
3     return super.toString() +
4          "Subject: " + subject + "\n";
5  }
6
```

For the `Textbook` class, we may wish to output the exact same information, but also add a line for the textbook `subject`. Examine the code in the box labeled *Textbook Class toString Method*. The toString method of the superclass (the Book class) is invoked with the code "`super.toString()`".

Given this code, consider a textbook variable instantiated with the following line of code:

```
Textbook myMathBook = new Textbook("Basic Algebra", 2025, "Mathematics");
```

A call to `myMathBook.toString()` would return the text:

```
Title: Basic Algebra
Year: 2025
Subject: Mathematics
```

Considering these past two sections, you should notice that the usage of the super keyword is very similar to the usage of the this keyword. If the `this` keyword is followed immediately by parentheses, it is a call to the constructor in the current class. If the `super` keyword is followed immediately by parentheses, it is a call to the constructor in the superclass. If the `this` keyword is followed by a method name, it is a call to the method from the current class. If the `super` keyword is followed by a method name, it is a call to the method from the superclass.