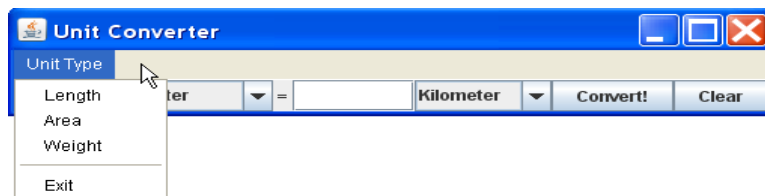


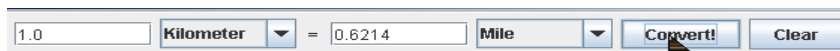
Assignment	Write a general unit converter that can convert several different types of units. You must have at least 4 kinds of units , including Length, Weight, Temperature, and at least one other kind of unit.
What to Submit	<ol style="list-style-type: none"> 1. Use the project on Github classroom to submit your work. Short URL: https://goo.gl/U5wbau Full URL: https://classroom.github.com/a/nROe74Bz The project is <code>unit-converter-githublogin</code>. 2. Create a runnable JAR file of your application and put it in the root folder of your project. Name the JAR <code>Converter-yourname.jar</code> (use your first name). 3. Use Git commands to commit work to Github. Please don't use file upload, students doing that put files in the wrong folder; as a result, their code does not compile. On this assignment, no credit if your code does not compile. 4. Create a UML class diagram of your application and submit it on paper.
Evaluation	<ol style="list-style-type: none"> 1. Implements requirements, performs correct conversions, and is usable. 2. Good OO design. Separate UI from application logic; use polymorphism instead of "if". No redundant code. 3. Code quality: follows <i>Java Coding Convention</i> for this course, code is well-documented and easy to read.

Requirements

1. Write a general unit converter that can convert values of different types of units, including Length, Weight, Temperature, and at least one other kind of unit.
2. Provide a **menu** to select the unit type: Length, Area, or Weight. Include an "Exit" option on the menu. See the Java tutorial for how to create a JMenuBar and JMenu.



3. When the user selects a type of unit, update the combo boxes to show **only that type of unit** in the combo-box. Don't mix unit types (e.g. meter and gram).
4. For each unit type include at least: 3 metric units (such as meter, cm, micron), 2 English units (such as foot, mile, acre, pound), and at least 1 Thai unit (wa, rai, thang). Exception: for Temperature there may not be enough standard unit types.
5. User should be able to convert in either direction: left-to-right or right-to-left. The converter should be smart enough to determine whether it should convert left-to-right or right-to-left, but give preference to left-to-right.



6. Program should **never crash** and **never print on the console**! Catch exceptions and handle them.
7. If user enters an invalid value you should catch it and change text color to **RED** (don't forget to change the color *back* the next time enter is pressed). Use try - catch.
8. Don't "hardcode" the unit information into the UI. The Unit Type menu must **not** contain the words "Length", "Weight", "Tempature", etc. Get the unit types from the `UnitFactory`!
9. Use polymorphism. The Controller should never refer to "Length", "Weight", "Temperature" -- everything is a unit. You should not need to *cast* unit values.

10. Use encapsulation: each unit type should have its own `convert()` method to convert values. This enables the app to handle non-linear conversion such as Temperature.

11. Create a UML class diagram for your application design.

12 Create a runnable JAR file. A runnable JAR file contains all your project classes in one JAR file and has a designated "main" class. You run a JAR by typing: `java -jar myjarfile.jar`

You can also run it by double clicking the JAR file's icon (on some operating systems).

In Eclipse, create a JAR file by right-clicking on the project and choose **Export...** Then choose "Jar" or "Runnable Jar" as export type.

Be careful that the JAR file includes your *.xml files. Otherwise, the JAR won't work.

In BlueJ use Project → Create Jar file...

Programming Hints

To enable polymorphism, you need an interface (or abstract superclass) for different kinds of units -- so that all units look alike. There are many ways to implement this. You **are not required to use enums** for the unit types or the units. You can devise another solution, such as reading unit data from a file, which may be more flexible.

Whatever solution you use, **don't "hardcode" the unit types or unit names into the UI!** The UI and Controller should accept any kind of units. The UI should get names of unit types and units from the UnitConverter.

Some suggestions are:

1. Use separate classes for the user interface (view), the controller, and the units.
2. You need to enable *polymorphism* so that the UI can accept any kind of unit. You need the different kinds of units to "look" and behave alike. Define an *interface* for **Unit** that specifies the behavior you require of all units.

The actual unit types (Length, Area, Weight) implement this interface. You can use a class or enum for the actual units.

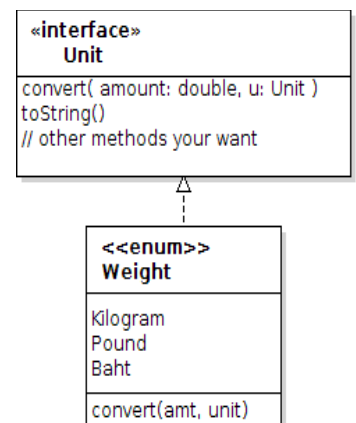
An enum can implement an interface, so Length can be an enum. But enum is not required, you can use a class for actual units. For example, if you want to convert *currencies* (in real time), you probably need a class.

In this example, the Unit interface has a `convert` method. But the parameter is just Unit (not a particular type of unit), which means `convert` will need to *cast* the parameter back to its own specific type (e.g. Length, Temperature) to access attributes of that Unit. For example:

```
// in the Length enum
public double convert(double amount, Unit toUnit) {
    Length unit = (Length) toUnit;
    // now apply formule for converting from this unit to other Length unit
}
```

If you don't like using a cast, you can add a type parameter to the Unit interface (like on the *Comparable* interface).

```
interface Unit<T> {
    public double convert(double amount, T toUnit);
}
enum Length implements Unit<Length> {
    ...
}
```



This eliminates the need to apply a *cast* in **convert**, but in the ConverterController you will get warnings about using Unit without a type parameter. If your code works correctly, you can ignore those warnings.

3. You need a way to add different kinds of units to the application.

This is similar to the problem of creating different kinds of money for the Coin Purse. Define a UnitFactory with 2 methods: a) method to get the types of units, b) method to get all the units of one type.

```
public class UnitFactory {
    /** Get the names of the types of units that the factory knows about. */
    public String[] getUnitTypes() { ... }

    /** Get all the units of one type, such as Volume. */
    public Unit[] getUnits(String unitType) { ... }
}
```

You can use the getUnitTypes() method to create MenuItems in the ConverterController class, and add them to the menu. When the user selects an item from the menu, invoke a handler method that calls getUnits() to get all the units of that type, and put them in the ComboBoxes.

This design is OK, but using Strings for unit types isn't type safe or easy to extend. You could define a UnitType enum that contains names of the units:

```
public enum UnitType {
    Length,
    Volume,
    Temperature,
    Weight;
}
```

and modify the methods of UnitFactory to accept and return UnitType instead of String.

4. The application only needs one UnitFactory, so make it a Singleton.

5. A MenuItem object can have its own "On Action" event handler. You can have all the MenuItems call the same Event Handler method. The(ActionEvent) parameter can be used to determine which MenuItem caused the event. For example:

```
/** Event handler for selecting a unittype using a menu item. */
public void handleUnitSelected(ActionEvent event) {
    MenuItem mitem = (MenuItem) event.getSource();
    String unitname = mitem.getText();
    UnitType unittype = UnitType.valueOf(unitname); // kind of cludgy
    // Now get the units of unittype from the UnitFactory
    // and populate the comboboxes.
    Unit<?>[] units = UnitFactory.getInstance().getUnits(unittype);
    // Don't forget to clear the old units from combobox first.
}
```

Its *bad coding* and *not extensible* to write a separate handler method for each unit type. Don't write separate "handleLength", "handleVolume", etc. methods.

6. The numbers may be very large or very small. So avoid displaying unformatted numbers! For example:

$$1 \text{ light-year} = 9.4605284 \times 10^{15} \text{ meters}$$

Try the "%g" format, which automatically chooses between fixed point and scientific notation.

Try this in BlueJ: **String.format("%.5g", x)** with large and small values of x.

Java 8 Lambdas:

We haven't covered Lambda expressions yet, but by using a Lambda expression you could avoid the need to "decode" the event parameter in the MenuItem event handler to get the unit name. If you are creating the MenuItems in a loop, this is an easy way to define an event handler that just calls another method.

```
MenuItem mitem = new MenuItem( unittype.toString() );  
// a Lambda expression for an event handler that invokes  
// a method named loadUnits(unittype). The "event" parameter is not used.  
mitem.setOnAction( (event) -> loadUnits(unittype) );
```

Now you can write a loadUnits(UnitType utype) method that gets units and populates the comboboxes.

Testing

Write and test each unit type first.

Then write and test the UnitFactory. Using a simple command line application may help.