Purpose	1. Practice implementing a factory method and singleton class. 2. Enable the Purse application to handle different kinds of money. 3. Practice using runtime configuration, either via dependency injection or Properties file.
What to Submit	Commit the revised coin purse to your Github project. Your revised coinpurse project must have these: 1. Singleton MoneyFactory class. 2. ThaiMoneyFactory and MalaiMoneyFactory as subclasses of MoneyFactory. These factories create money using the local currency. 3. Test classes for MoneyFactory methods: MoneyFactoryDemo and MoneyFactoryTest 4. The user interface uses MoneyFactory to create money. No "new Coin", "new BankNote". 5. A properties file for specifying a MoneyFactory class name. The Main class reads the class name from this file and creates a MoneyFactory object. 6. Banknotes do not choose their own serial number! The country's money factory does that. Each currency may have different rules for serial numbers. 7. (Optional) For Malaysian coins, a 0.05 Ringgit coin should display "5 *Sen*" not "0.05 *Ringgit*". But, the currency is still "Ringgit" (otherwise, sorting and withdraw won't work). A Money object (which can have any value, not join currency values) would display "0.05 Ringgit".

The Problem

KU wants to use your Coin Purse at its new campus in Malaysia. In Malaysia the currency is *Ringgit* and the *denominations* are different from Thailand:

Currency Values	Thailand	Malaysia
Coins	1, 2, 5, 10 Baht	0.05 (called "5 Sen"), 0.10 ("10 Sen"), 0.20 ("20 Sen"), 0.50 ("50 Sen")
Bank Notes	20, 50, 100, 500, 1000 Baht	1, 2, 5, 10, 20, 50, 100 Ringgit

Note: 500 Ringgit and 1000 Ringgit notes were *canceled* in 1999 during the financial crisis to prevent export of large amounts of cash.

Modify your coin purse *and* the user interface so that it can create and use either kind of currency. In Thailand, "1" means a 1-Baht coin; in Malaysia "1" means a 1-Ringgit banknote, etc.

Problem 1. Create a MoneyFactory that is a Singleton

Define a MoneyFactory class for creating money. This way our application won't have to use "new Coin(...)" or "new Banknote(...)".

MoneyFactory is an *abstract class* that does only:

1) a static getInstance() method to get a concrete factory instance. Returns a subclass of MoneyFactory.

2) a way to "set" the MoneyFactory instance. This is mainly for testing.

3) methods to create coins and banknotes.

static MoneyFactory getInstance()	get an instance of MoneyFactory. This method returns an object of a subclass (such as ThaiMoneyFactor). The instance is a *Singleton* -- it always returns the same object.
abstract Valuable createMoney(double value)	create new money object in the local currency. If the `value` is not a valid currency amount, then throw IllegalArgumentException. *You must include a String message in the exception that describes the problem, e.g. "100,000 is not a valid currency value".*

Valuable createMoney(String value)	accepts money value as a String, e.g. createMoney("10"). This method is for *convenience* of the U.I. The default implementation of this method converts parameter to a double and calls createMoney(double), but a subclass may override it to permit other parameter values. Throws: IllegalArgumentException if value of string is not a number.
static void setFactory(MoneyFactory f)	Static method to a "set" the MoneyFactory object that is used. This is mostly for testing of MoneyFactory.

Example: To get a MoneyFactory the application calls the static getInstance() method. Then the app can use the MoneyFactory instance to create Money.

This uses *polymorphism* twice: 1) the app doesn't need to know which concrete factory it is using, and 2) it doesn't know (or care) whether createMoney returns a Coin, BankNote, or something else.

```
// get the actual money factory.
MoneyFactory factory = MoneyFactory.getInstance( );
// get a "10" unit money object, in the local currency
Valuable m = factory.createMoney("10.0");
```

1.1 Define MoneyFactory as an Abstract Class

1. Use the package **`coinpurse`**. Write good Javadoc comments for all methods.

```
public abstract class MoneyFactory {
    /** singleton instance of MoneyFactory. */
    private static MoneyFactory instance;   // use any descriptive name

    public static MoneyFactory getInstance() { /* TODO */ }

    public abstract Valuable createMoney(double value);

    public Valuable createMoney(String value) {
       // parse the String as a double and call the other createMoney method
       // if String value isn't a number then throw IllegalArgumentException
    }

    public static void setFactory(MoneyFactory factory) { /* TODO */ }
```

2. createMoney(String) is a concrete method. It parses the String parameter as a double and calls the abstract createMoney(double) method. If the String does not contain a number, throw IllegalArgumentException. The exception must contain a *String message* that describes the problem! Only throw exception if string param cannot be parsed as a double.

Use try - catch for this:

```
    double value = 0;
    try {
       value = Double.parseDouble( arg );
    } catch (_________?________ ex) {
        // throw a new IllegalArgumentException with a message
    }
    return createMoney( value );
```

1.2 Make MoneyFactory be a Singleton

We only need one instance of the actual MoneyFactory in the application.

Write the getInstance() method so that it always returns the same object. If the factory has already been created or *injected* by calling setFactory(), then getIntance() just returns it. If no factory instance yet, then create it.

To get started, the default is to create a ThaiMoneyFactory. Later in the lab you will use a properties file to specify which factory class to create.

How to write a Singleton class is in the `patterns` folder on the course web pages:

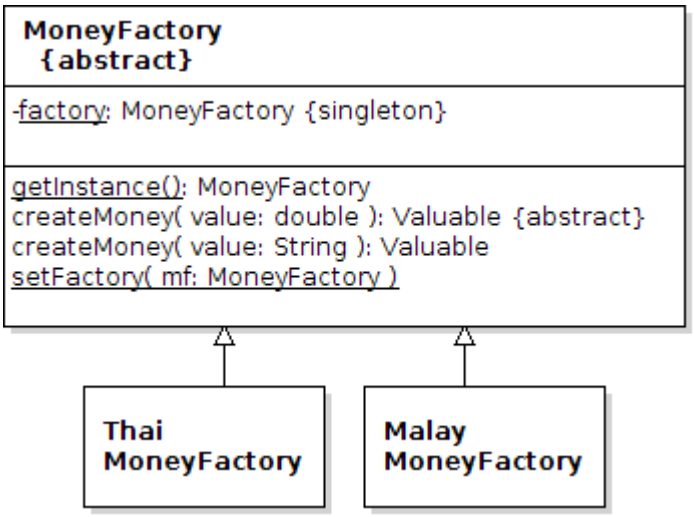
https://skeoop.github.io/patterns/Singleton-Pattern.pdf

Problem 2. Write Concrete Money Factories

Write classes ThaiMoneyFactory and MalayMoneyFactory as subclasses of MoneyFactory.

2.1 Implement createMoney(double) for each MoneyFactory subclass. Create coins and banknotes according to the table of currency values above. *Don't write* getInstance() *or setFactory() in subclasses! Only* MoneyFactory (superclass) *has those methods*

2.2 Redesign BankNote so the country's money factory assigns the serial numbers! Banknotes should not assign their own serial numbers. Design your own solution to this. However, the BankNote should not "call" the MoneyFactory to ask for a serial number. That creates an extra dependency (coupling) from BankNote to MoneyFactory. It is better to *minimize coupling* when you can.



Example: In Thailand, we would observe this:

```
> MoneyFactory factory = MoneyFactory.getInstance();
> Valuable m = factory.createMoney( 5 );
> m.toString()
 "5-Baht coin"
> Valuable m2 = factory.createMoney("1000.0");
> m2.toString()
 "1000 Baht note [1000001]"
```

In Malaysia we would observe this:

```
> MoneyFactory factory = MoneyFactory.getInstance();
> Valuable m = factory.createMoney( 5 );
> m.toString()
 "5 Ringgit note [1000004]"
> Valuable m3 = factory.createMoney( 0.05 );
> m3.toString()
 "5 Sen coin"    (notice the display value is "Sen")
```

```
> m3.getCurrency()
 "Ringgit"        (the actual currency is still Ringgit)
> Valuable m2 = factory.createMoney("1000.0");
 IllegalArgumentException      (Malaysia doesn't have 1,000 note)
```

Problem 3: Write Console Demo and JUnit Tests for MoneyFactory

3.1 Write a **MoneyFactoryDemo** class to create a MoneyFactory and call its methods. Print results on the console. Your code should show that a) MoneyFactory is a singleton, b) all the methods work as specified.

3.2 Write a **MoneyFactoryTest** class containing JUnit tests of MoneyFactory methods. We haven't studied JUnit yet but you have used it in sevaral assignments. Use the previous JUnit test files as example.

(Some examples given in lab.)

Problem 4: Modify ConsoleDialog to use MoneyFactory. No "new Coin()" or "new BankNote()"!

Remove all the "new Coin()" and "new Banknote()" code from ConsoleDialog. Use MoneyFactory to create money.
The moneyFactory.createMoney( ) methods throw an exception if the amount is invalid. In the user interface, *catch* the exception and print a friendly message. For example:

```
String amount = console.next();
try {
    valuable = moneyFactory.createMoney( amount );
} catch (IllegalArgumentException ex) {
    System.out.println("Sorry, "+amount+" is not a valid amount.");
    continue;  // if inside a loop then go back to top
}
```

Problem 5: Enable Factory Selection via Configuration File

We want to be able to easily *change* which factory is used in our application (ThaiMoneyFactory, MalayMoneyFactory, or other). And we want to change it *without* modifying the MoneyFactory code!

You have already implemented a partial solution -- a *setFactory* method. This lets us specify which factory to use, but it still requires writing Java code to create the factory object.

5.1 Use a Property File (as ResourceBundle)

This is a more general solution. We want to write the *name* of the Factory class in a configuration file, and have our application read the class name, then create an object of that class.

Many applications use a *properties file* or *configuration file* to let the user specify values that he wants the program to use. Eclipse has a configuration file named `eclipse.ini`. BlueJ has a properties file named `bluej.properties` (on Linux its in the .bluej directory). You can edit these files (with a text editor) to set values that you want.

A *properties file* contains lines of the "key = value". For the Coin Purse it might look like this:

File: purse.properties

```
# lines beginning with "#" are comment lines
# the name and package of the concrete money factory to use
moneyfactory = coinpurse.ThaiMoneyFactory
# example of another property
version=1.0
```

This file (**purse.properties**) needs to be on the application *classpath*, so put in your project **src/** directory. Put it directly in src/ not in a subdirectory (i.e. a package).

5.2 How to read a Properties File

The Java API has classes that can read a properties file and create a *map* of key-value pairs from the file. The easiest one to use is **`java.util.ResourceBundle`**. (Another class is java.util.Properties.).

ResourceBundle has a static method to create a bundle from a classpath resource:

```
// create a ResourceBundle from file "purse.properties" on the classpath
// the ".properties" extension is automatically appended to the name
ResourceBundle bundle = ResourceBundle.getBundle( "purse" );
```

Then you can get individual properties like in a map:

```
// get value of "moneyfactory" property
String factoryclass = bundle.getString( "moneyfactory" );
// for testing, try this:
System.out.println("Factory class name is "+factoryclass);
```

You can also use bundle.keySet() to get all the keys in the properties file, just like a Map.

Now you have the *name* of the factory class as a String. How to create an instance of a class using only a String?

5.3 Create an object from a String containing the Class Name

You can load and instantiate a class at runtime using just a String containing the class name!

This is very useful -- it gives you a way to add new classes to a program *after* you have written and compiled that program.

The syntax is:

```
Object obj = Class.forName("class_name").newInstance();
```

The "class_name" must include the package name, such as "`java.util.Date`".

```
Object date = Class.forName("java.util.Date").newInstance( );
System.out.println( date );  // prints the date and time
```

Usually you need to *cast* the result to a type that you want.

This code may throw several exceptions, so use try - catch to catch and handle them.

```
String factoryclass = bundle.getString("moneyfactory");
//TODO if factoryclass is null then use a default class name
MoneyFactory factory = null;
try {
    factory = (MoneyFactory)Class.forName(factoryclass).newInstance();
}
catch (ClassCastException cce) {
    //the object could not be cast to type MoneyFactory
    System.out.println(classname+" is not type MoneyFactory");
}
catch (Excepion ex) {
    // any other exception means we could not create an object
    System.out.println("Error creating MoneyFactory "+ex.getMessage() );
}
// if no factory then quit
if (factory == null) System.exit(1);
```

Where to put this code?

To avoid making MoneyFactory too complicated, put this code in a separate method of the Main class, such as `Main.init( )`. The Main class creates the MoneyFactory object and then calls MoneyFactory.setFactory(factory) to inject it, before starting the user interface.

Problem 6. Add Country-Specific Formatting to Money (Optional)

For Malaysian coins (but not other kinds of coins) a 0.05 Ringgit coin has value = 0.05, currency="*Ringgit*". But the Coin's toString() method should return "5-*Sen* coin", not "*0.05-Ringgit coin*". Design your own solution.

Try to write a clean solution to this. A *really* good solution is something that can be applied to other currencies as well. For example, 0.00000001 Bitcoin is "*1 Satoshi*". But it is OK if your solution only works for Malay money.

Don't modify the existing Coin methods API (if you do, it will fail JUnit tests), but you can define a new method or subclass.