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. The user interface uses MoneyFactory to create money. No "new Coin", "new BankNote". 4. A way to change which concrete MoneyFactory will be used (Thai or Malay). This can be done in the Main class, or (much cooler) via a properties file. 5. Design & code your own solution to these 2 problems: a) Banknotes should not choose their own serial number! The country's money factory should do that. b) For Malay money, a 0.05 Ringgit coin should display "5 *Sen*" not "0.05 *Ringgit*". But, all Malay money has currency equal "Ringgit" (otherwise, withdraw won't work). There are many currencies like this. In Thailand, 0.25 Baht is "25 *Satang*".

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. Thailand used to have a 10 Baht banknote but switched to a coin.

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 2 things:

1) provides access to a concrete factory instance that is a subclass of MoneyFactory

2) defines the methods used 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.
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. If value is not a valid number, then throw IllegalArgumentException.

Example: get a MoneyFactory and create money (in the local currency) with a face value of "10".:

```
// get the actual money factory.
MoneyFactory factory = MoneyFactory.getInstance( );
// get a "10" Baht, Ringgit, or whatever
Valuable m = factory.createMoney(10.0);
```

1.1 Define MoneyFactory as an Abstract Class

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

```
public abstract class MoneyFactory {
    public abstract Valuable createMoney(double value);

    public Valuable createMoney(String value) {
       // parse the String as a double and call the other createMoney method
   }
```

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

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 Factories

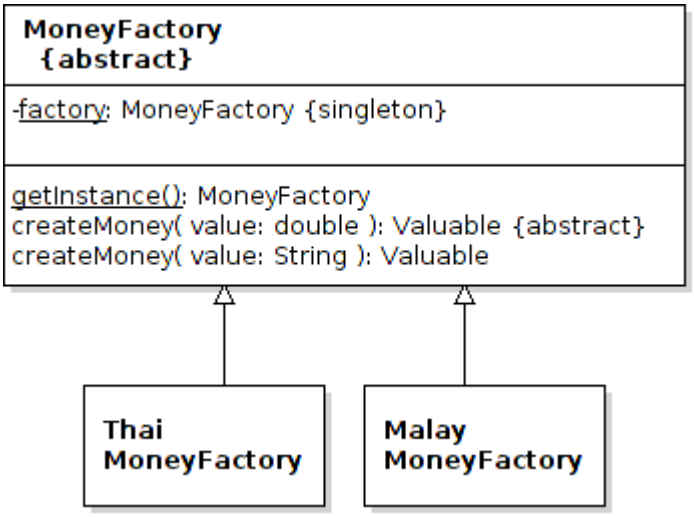
Write the classes ThaiMoneyFactory and MalayMoneyFactory as subclasses of MoneyFactory.

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

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.

2.3 For Malaysian money (but not Thai or US money) a 0.05 Ringgit coin we *should* have value = 0.05, currency="*Ringgit*" so money will be sorted and withdrawn correctly. But the Coin's toString() method should return "5-*Sen* coin", not "*0.05-Ringgit coin*".

Try to write a clean solution to this. A *really* good solution would work for many currencies, but its OK if your solution only works for Malay money. Don't modify the existing API of Coin (if you do, it will fail JUnit tests), but you can define a subclass or make internal changes.



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" not "Ringgit")
> 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: Modify ConsoleDialog to get a MoneyFactory and use it

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 4: Enable Factory Selection

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

Here are **two solutions**. You can implement **either one.** The first solution is simpler.

4.1 Use Dependency Injection

Define a *static* setMoneyFactory(MoneyFactory mf) method in MoneyFactory. This enables us to "inject" a particular MoneyFactory. This is useful for testing.

In the Main (application) class create the concrete factory object using code and call setMoneyFactory(factory). The Main needs to do this before starting the rest of the application.

4.2 Use a Property File (as ResourceBundle)

This is a more general solution but requires more code. Many applications let you configure the program using a *properties file* that the application reads at start-up. For this application, the properties file will contain just one property: the fully qualified name of the concrete MoneyFactory class.

A *properties file* contains lines of the "key = value". For the purse app 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 = purse.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).

4.2.1 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?

4.2.2 Create an object from a String containing the class name

You can load and instantiate a class at runtime using 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( );
```

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 instance = null;
try {
    instance = (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 (instance == null) System.exit(1);
```

Where to put this code?

To avoid making MoneyFactory too complicated, put this code in the Main class. The Main class creates the MoneyFactory object and then calls MoneyFactory.setMoneyFactory(...) just like in case 4.1.

Problem 5. Test Your MoneyFactory

Write some test code to verify:

a) it is a singleton. Prove that everytime you call getMoneyFactory is returns the same object.

b) it correctly creates Coins and Banknotes

Appendix: Runtime Application Configuration using Properties

Many applications let you define their configuration by editing a text file, called a *configuration file* or *properties file*. Eclipse uses a file name eclipse.ini, BlueJ uses file bluej.properties (in your home directory), and Log4J (open source library) uses log4j.config (in your application directory). A properties file looks like this:

```
# this is a comment
root.loglevel = WARN
```

Lines beginning with # are comment lines. Comments and blank lines are ignored. Other lines are of the form: `propertyname=value`. No quotation marks are used around the values.

Java can read and write a properties file for you, using either of these classes:

java.util.Properties - a Map of key-value pairs with extra methods for reading and writing properties files.
java.util.ResourceBundle - similar to Properties, but allows multiple files each with a different *locale* suffix.

The code for using a ResourceBundle is easier to write.

First, create a file named **purse.properties** *inside your top source code directory* ("src" for Eclipse).

```
CoinPurse/
        src/
           purse.properties
```

Put some properties in this file. Usually property names are *lowercase*, like Java package names.

```
# purse.properties
# Lines beginning with # are comments
purse.author = Bill Gates
# Name of the class for creating money
purse.moneyfactory = coinpurse.factory.ThaiMoneyFactory
```

This file contains 2 properties in the form *key=value*. Spaces around the key name and = sign are ignored.

A property name may contain periods, such as "`purse.author`". This avoids name conflicts (called *naming collision*) in property names used by different components.

Load the properties in your app as a ResourceBundle:

```
ResourceBundle rb = ResourceBundle.getBundle("purse");
```

Print some properties to verify it works:

```
System.out.println("Purse by " + rb.getString("purse.author") );
// print all properties
Enumeration<String> keys = rb.getKeys();
while( keys.hasMoreElements() ) {
     String key = keys.nextElement();
     System.out.println( key + " = " + rb.getString(key) );
}
```

Your application only has *one* ResourceBundle and you only need to load it *once*, so consider creating a separate class to access property values from the ResourceBundle. For example, a PropertyManager class. Then you can write:

```
String author = PropertyManager.getProperty( "purse.author" );
```