Purpose	1. Practice implementing the Factory Method and Singleton patterns. 2. Practice using Properties for run-time configuration of objects.
What to Submit	Commit the revised coin purse to your Bitbucket project. The revised purse project must have these: 1. AbstractValuable has a currency. Update the constructors to AbstractValue, Coin, and BankNote to accept a currency parameter. 2. Singleton MoneyFactory class that creates a single concrete factory. 3. ThaiMoneyFactory and MalaiMoneyFactory as subclasses of MoneyFactory. These factories create money using the local currency. 4. No "new Coin" or "new BankNote" statements <u>anywhere</u> in your code <u>*except*</u> in (a) Factory classes and (b) JUnit tests. 5. MoneyFactory uses a properties file to get the name of the MoneyFactory class it is supposed to use.

The Problem

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

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")
BankNotes	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 currency.

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

Problem 1. Create a Singleton MoneyFactory

MoneyFactory is an abstract class that creates a concrete factory of a subclass. It has these methods:

static MoneyFactory getInstance()	get an instance of MoneyFactory. Use *lazy instantiation*. This means you create the singleton object the first time getInstance() is called.
abstract Valuable createMoney(double value)	create a new money object in the local currency. Only accept valid values. If an invalid value of the parameter then throw IllegalArgumentException.
Valuable createMoney(String value)	convenience method converts String to double, e.g. createMoney("10") and invokes createMoney(10.0).

Reference: student reports on Singleton Pattern and Factory Method pattern

Problem 2. Write Concrete Factories

Implement ThaiMoneyFactory and MalaiMoneyFactory as subclasses of MoneyFactory. They create coins and banknotes according to the table of currency values above.

The only method in the subclasses is createMoney(double). *Don't write* getInstance() *in subclasses!*

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 Banknote"
> Valuable m3 = factory.createMoney(0.05);
 throws IllegalArgumentException
```

In Malaysia we would observe this:

```
> MoneyFactory factory = MoneyFactory.getInstance();
> Valuable m = factory.createMoney( 5 );
> m.toString()
 "5 Ringgit Banknote"
> Valuable m2 = factory.createMoney("1000.0");
 throws IllegalArgumentException
> Valuable m3 = factory.createMoney( 0.05 );
> m3.toString()
 "5 Sen coin"
```

2.1 Modify Coin, BankNote, and super class(es) to support currency.

Modify Coin, Banknote, and their superclasses to support currency.

You should be able to design and implement this yourself.

Problem 3: Enable Factory Selection at Runtime

MoneyFactory has to create a single concrete MoneyFactory object, but we must be able to *change* the type of factory (ThaiMoneyFactory or MalayMoneyFactory) *without* modifying the code.

How can we do this?

Here are **two solutions**. You will implement both.

3.1 *Dependency Injection*

Define a static setMoneyFactory( ) method to set the factory via code. This is useful for testing. This method should be invoked *before* getInstance( ).

3.2 *Use a Property File*

You can load and instantiate a class at runtime using Class.forName("class name").newInstance();

for example, to create a new Date object you could write:

```
      Object date = Class.forName("java.util.Date").newInstance( );
```

we can use this ability to change the MoneyFactory at runtime by doing this:

I) read the name of the MoneyFactory class from a file

2) use the above code to create an instance of this class

java.util.Properties and java.util.ResourceBundle are key-value maps for properties, that also provide methods to make it easy to load values from a file. ResourceBundle is easier to create, so that's the one we'll use here.

How Properties and ResourceBundle works.

A *properties file* contains keys and values such as:

```
# this is a comment line
moneyfactory = purse.factory.ThaiMoneyFactory
logging.level=INFO
version=1.0
```

There are constructors to create a Properties or ResourceBundle from an InputStream or Reader. ResourceBundle also has a static method to create a bundle from a classpath resource:

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

Then you can get individual properties like in a map:

```
// get value of "currency" property (if there is one)
String value = bundle.getString( "currency" );
```

You can use this to decide which concrete MoneyFactory class to create.

Let's assume that

(1) the properties file is named "`purse.properties`" and located in the root of our project classpath. In Eclipse that means you put it in `src/purse.properties` (Eclipse will copy it your `bin` directory and export it to a Jar file). To load the properties from the properties file, you would write:

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

(2) the properties file uses the key `moneyfactory` for the name of the concrete MoneyFactory class to create.

```
String factoryclass = bundle.getString("moneyfactory");
//TODO if factoryclass is null then use a default class
//this is for testing...
System.out.println("Factory class is " + factoryclass);  //  testing
try {
    instance = (MoneyFactory)Class.forName(factoryclass).newInstance();
} catch (_________________ ex) {
//TODO what exceptions might be thrown?
    System.out.println("Error creating MoneyFactory "+ex.getMessage() );
    return defaultMoneyFactory();
}
```

You can put this code right in MoneyFactory.getInstance(), but it will make the method rather complex. A cleaner solution is to define a private static method for this and invoke it from getInstance().

Example:

```
MoneyFactory factory = MoneyFactory.getInstance();
System.out.println( factory.createMoney("10") );
System.out.println( factory.createMoney(50.0) );
```

If purse.properties contains `factoryclass=purse.ThaiMoneyFactory` then you should see:

```
10 Baht Coin
50 Baht BankNote
```

If purse.properties contains `factoryclass=purse.MalaiMoneyFactory` then you should see:

```
10 Ringgit Banknote
50 Ringgit BankNote
```

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.

A ResourceBundle is slightly easier to read from a file.

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.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" );
```