Comp 401: Factories

Instructor: Prasun Dewan
CONCEPTS USED

- Interfaces
- Abstract Methods
- Action Objects
- Exceptions
NEW CONCEPTS

- Factory Classes
- Static Factory Methods
- Indirection
- Binding Time
- Reading Files
- Static Blocks
- Reflection
- Multi-Exception Catch Block
- Abstract Factories
- Instance Factory Methods
- Singletons
public interface Counter {
    public void add (int amount);
    public int getValue();
}
IMPLEMENTATION 1: SHORT COUNTER

```java
public class AShortCounter implements Counter {
    short counter;
    public AShortCounter (short initValue) {
        counter = initValue;
    }
    public void add (int amount) {
        counter += amount;
    }
    public int getValue() {
        return counter;
    }
}
```
public class AnIntCounter implements Counter {
    int counter;
    public AnIntCounter (short initValue) {
        counter = initValue;
    }
    public void add (int amount) {
        counter += amount;
    }
    public int getValue() {
        return counter;
    }
}
public class ACourseVisits implements CourseVisits{
    Counter youTubeVisits = new AShortCounter((short) 0);
    Counter mixVisits = new AShortCounter((short) 0);

    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}

public class ACourseSubscriptions implements CourseSubscriptions {
    Counter youTubeSubscriptions = new AShortCounter((short) 0);
    Counter mixSubscriptions = new AShortCounter((short) 0);
    
    public void youTubeSubscribed(String aUser) {
        youTubeSubscriptions.add(1);
    }
    
    public void youTubeUnSubscribed(String aUser) {
        youTubeSubscriptions.add(-1);
    }
    
    public void mixSubscribed(String aUser) {
        mixSubscriptions.add(1);
    }
    
    public void mixUnSubscribed(String aUser) {
        mixSubscriptions.add(-1);
    }
}
CHANGING COURSE VISITS: USING INT COUNTER

```
public class ACourseVisits implements CourseVisits{
    Counter youTubeVisits = new AnIntCounter((short) 0);
    Counter mixVisits = new AnIntCounter((short) 0);

    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
```

- **Instantiating code not reused**
- **Method calls reused**
public class ACourseSubscriptions implements CourseSubscriptions{
    Counter youTubeSubscriptions = new AnIntCounter((short) 0);
    Counter mixSubscriptions = new AnIntCounter((short) 0);

    public void youTubeSubscribed(String aUser) {
        youTubeSubscriptions.add(1);
    }

    public void youTubeUnSubscribed(String aUser) {
        youTubeSubscriptions.add(-1);
    }

    public void mixSubscribed(String aUser) {
        mixSubscriptions.add(1);
    }

    public void mixUnSubscribed(String aUser) {
        mixSubscriptions.add(-1);
    }
}
Problem

How to allow easy switching between different alternative implementations

How to make main and other classes instantiating implementations not duplicate code?

Put the code in some method accessible to multiple classes
**Static Factory Method**

```java
public class StaticCounterFactory {
    public static Counter createCounter (short initValue) {
        return new AShortCounter(initValue);
    }

    public static Counter createCounter () {
        return createCounter((short) 0);
    }
}
```

- Class instantiated using a static method shareable by multiple accesses
- Method can provide actual instantiation arguments to constructors, saving class users from supplying default parameters
- Multiple static factory methods taking place of constructors and can be in one class, called static factory, associated with the interface
- Multiple related classes can be instantiate by factory methods in a class
Course Visits: Using Factory Methods

Direct instantiation

```java
public class ACourseVisits implements CourseVisits{
    private Counter YouTubeVisits = new AShortCounter((short) 0);
    private Counter mixVisits = new AShortCounter((short) 0);

    public void youTubeVisited(String aUser) {
        YouTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
```

Indirect instantiation

```java
public class AStaticFactoryMethodUsingCourseVisits implements CourseVisits{
    private Counter YouTubeVisits = StaticCounterFactory.createCounter();
    private Counter mixVisits = StaticCounterFactory.createCounter();

    public void youTubeVisited(String aUser) {
        YouTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
```

Indirection: Not doing a task (e.g. instantiation) directly

A la clues in treasure hunt
public class AStaticFactoryMethodUsingCourseSubscriptions implements CourseSubscriptions {

  public void youtubeSubscribed(String aUser) {
    youTubeSubscriptions.add(1);
  }

  public void youtubeUnSubscribed(String aUser) {
    youTubeSubscriptions.add(-1);
  }

  public void mixSubscribed(String aUser) {
    mixSubscriptions.add(1);
  }

  public void mixUnSubscribed(String aUser) {
    mixSubscriptions.add(-1);
  }
}
public class AStaticFactoryMethodUsingCourseSubscriptions implements CourseSubscriptions {

    public void youTubeSubscribed(String aUser) {
        youTubeSubscriptions.add(1);
    }

    public void youTubeUnSubscribed(String aUser) {
        youTubeSubscriptions.add(-1);
    }

    public void mixSubscribed(String aUser) {
        mixSubscriptions.add(1);
    }

    public void mixUnSubscribed(String aUser) {
        mixSubscriptions.add(-1);
    }
}

public class AStaticFactoryMethodUsingCourseVisits implements CourseVisits {

    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}

public class AStaticFactoryMethodUsingCourseVisits implements CourseVisits {

    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
public class StaticCounterFactory {
    public static Counter createCounter (short initValue) {
    }

    public static Counter createCounter () {
        return createCounter((short) 0);
    }
}

ORIGINAL STATIC FACTORY METHOD
public class StaticCounterFactory {
    public static Counter createCounter (short initValue) {
    }

    public static Counter createCounter () {
        return createCounter(((short) 0));
    }
}

Change not duplicated!

Must have access to source code

Decision made at program writing time
**Binding Time**

Time when some property of a program (e.g. which counter class, type or value of a variable) bound to a value (a particular counter class, a particular type or value)

- Program writing time
- Program compile time
- Program load time
- Program start time
- Program runtime

- Late binding is (usually) more flexible
- Late binding is (usually) less efficient
**CHANGED STATIC FACTORY METHOD**

```java
public class StaticCounterFactory {
    public static Counter createCounter (short initValue) {
    }

    public static Counter createCounter () {
        return createCounter((short) 0);
    }
}
```

How to make decision at program start time?

Must have access to source code

Decision made at program writing time
Configuration File

Configurable Static Factory Class

Configuration file

Class reads name of class from configuration file before factory methods are called

Converts name into class object using reflection

Finds constructor object taking short value

Invokes constructor of the class
public class StaticConfigurableCounterFactory {
    public static final String CONFIGURATION_FILE_NAME = "counter_config.txt";
    static Class counterClass = AShortCounter.class;
    static Constructor counterConstructor;
    public static Counter createCounter (short initValue) { ... }  
    public static Counter createCounter () { ... }  
    static {  // executed once for each class before it is used
        try {
            Scanner aScanner =
                new Scanner (new File(CONFIGURATION_FILE_NAME));
            counterClass = Class.forName(aScanner.nextLine());
        } catch (FileNotFoundException | NoSuchElementException |
            ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
public class StaticConfigurableCounterFactory {
    public static final String CONFIGURATION_FILE_NAME = "counter_config.txt";
    static Class counterClass = AShortCounter.class;
    static Constructor counterConstructor;
    public static Counter createCounter (short initValue) {
        try {
            counterConstructor =
                counterClass.getConstructor(short.class);
            return (Counter)
                counterConstructor.newInstance(initValue);
        }
        catch (NoSuchMethodException |
                InstantiationException |
                IllegalAccessException |
                IllegalArgumentException |
                InvocationTargetException e) {
            e.printStackTrace();
            return new AShortCounter((short) 0);
        }
    }
}
public class StaticConfigurableCounterFactory {

    ...

    public static Counter createCounter () {
        return createCounter((short) 0);
    }

}
**Binding Time**

Time when some property of a program (e.g. which counter class, type or value of a variable) bound to a value (a particular counter class, a particular type or value)

- Program writing time
- Program compile time
- Program load time
- Program start time
- Program runtime

What is we want an API to change the counter at runtime
public class StaticCounterFactory {
    public static Counter createCounter (short initValue) {
        return new AShortCounter(initValue);
    }
    public static Counter createCounter () {
        return createCounter((short) 0);
    }
}

What is we want an API to change the counter at runtime

More indirection

Make factory methods instance methods

Factory methods be set by the programmer

API to set Factories with these methods

Abstract factories used to access the factories
Instantiable Factory

Factory

void create\(^1\)(...)

void create\(^2\)(...)

implements

AFactory

AnotherFactory

Provides instance methods for creating one or more related classes

Each method takes instantiation parameters

Different implementations can instantiate different classes

Typically instantiation parameters become constructor parameters

How to choose among different factories?
Abstract Factories or Factory Selectors

- Factory Selector/Abstract Factory
  - static Factory getFactory()
  - static setFactory(Factory f)

- Has-A

- Factory
  - create¹(...)
  - create²(....)

- Helps choose between factories
  - Has a link to a Factory instance
  - Static methods to change and get reference
  - Can get reference to factory and invoke factory methods
  - Not an abstract class
public class StaticCounterFactory {
    public static Counter createCounter (short initValue) {
        return new AnIntCounter(initValue);
    }

    public static Counter createCounter () {
        return createCounter((short) 0);
    }
}

**Static Factory Methods**
Instantiatable Multiple Factory Classes

```java
public interface CounterFactory {
    public Counter createCounter (short initValue) ;
    public Counter createCounter () ;
}
```

```java
public class AnIntCounterFactory implements CounterFactory {
    public Counter createCounter(short initValue) {
        return new AnIntCounter(initValue);
    }
    public Counter createCounter() {
        return createCounter((short) 0);
    }
}
```

```java
public class AShortCounterFactory implements CounterFactory {
    public Counter createCounter(short initValue) {
        return new AShortCounter(initValue);
    }
    public Counter createCounter() {
        return createCounter((short) 0);
    }
}
```
public class StaticCounterFactorySelector {
    static CounterFactory counterFactory =
        new AShortCounterFactory();
    public static CounterFactory getCounterFactory() {
        return counterFactory;
    }
    public static void setCounterFactory (CounterFactory aCounterFactory) {
        counterFactory = aCounterFactory;
    }
}
public class AFactorySelectorUsingCourseVisits implements CourseVisits{

    Counter youTubeVisits = StaticCounterFactorySelector.getCounterFactory().createCounter();
    Counter mixVisits = StaticCounterFactorySelector.getCounterFactory().createCounter();

    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
public static void main (String[] args) {
    StaticCounterFactorySelector.setCounterFactory(
        new AShortCounterFactory());
    CourseVisits aCourseVisits =
        new AFactorySelectorUsingCourseVisits();
    aCourseVisits.mixVisited("anonymous");
    StaticCounterFactorySelector.setCounterFactory(
        new AnIntCounterFactory());
    aCourseVisits =
        new AFactorySelectorUsingCourseVisits();
    aCourseVisits.mixVisited("anonymous2");
}
**Binding Time**

Time when some property of a program (e.g. which counter class, type or value of a variable) bound to a value (a particular counter class, a particular type or value)

- Program writing time
- Program compile time
- Program load time
- Program start time
- Program runtime
**FACTORY ALTERNATIVES**

- Static factory classes (with static factory methods)
- Instantiatable factory classes and abstract factories
- Both can be configurable through a file
```
public class ACourseSubscriptions implements CourseSubscriptions {
    Counter youTubeSubscriptions = new AnIntCounter((short) 0);
    Counter mixSubscriptions = new AnIntCounter((short) 0);

    public void youTubeSubscribed(String aUser) {
        youTubeSubscriptions.add(1);
    }

    public void youTubeUnSubscribed(String aUser) {
        youTubeSubscriptions.add(-1);
    }

    public void mixSubscribed(String aUser) {
        mixSubscriptions.add(1);
    }

    public void mixUnSubscribed(String aUser) {
        mixSubscriptions.add(-1);
    }
}
```

```
public class ACourseVisits implements CourseVisits {
    Counter youTubeVisits = new AnIntCounter((short) 0);
    Counter mixVisits = new AnIntCounter((short) 0);

    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
```

**Problem**

How to make main and other classes instantiating implementations not duplicate code?

Put the code in some method accessible to multiple classes.
public class ACourseVisits implements CourseVisits{
    Counter youtubeVisits = new AnIntCounter((short) 0);
    Counter mixVisits = new AnIntCounter((short) 0);

    public void youTubeVisited(String aUser) {
        youtubeVisits.add(1);
    }

    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}

How to remove code duplication in a single class
public class ACourseVisitsWithFactoryMethods implements CourseVisits{
    Counter youtubeVisits = createCounter();
    Counter mixVisits = createCounter();
    public Counter createCounter() {
        return new AnIntCounter ((short) 0);
    }
    public void youTubeVisited(String aUser) {
        youtubeVisits.add(1);
    }
    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}

A subclass can override factory method used in superclass
public abstract class AnAbstractCourseVisitsWithFactoryMethods implements CourseVisits{
    public abstract Counter createCounter();
    Counter youTubeVisits = createCounter();
    Counter mixVisits = createCounter();
    public void youTubeVisited(String aUser) {
        youTubeVisits.add(1);
    }
    public void mixVisited(String aUser) {
        mixVisits.add(1);
    }
}
### Concrete Classes

```java
public class AnIntCourseVisits extends AnAbstractCourseVisitsWithFactoryMethods {
    @Override
    public Counter createCounter() {
        return new AnIntCounter ((short) 0);
    }
}
```

```java
public class AShortCourseVisits extends AnAbstractCourseVisitsWithFactoryMethods {
    @Override
    public Counter createCounter() {
        return new AShortCounter ((short) 0);
    }
}
```

Classes can differ only in the factory methods

Different implementation of an interface used by different classes
FACTORY ALTERNATIVES

Static factory classes (with static factory methods)

Instantiatable factory classes and abstract factories with (overridable) instance factory methods

Instance (overridable), possibly not public factory methods called by the same class or its superclasses or subclasses

Factory class approach

Factory method approach
(Special) Factory Classes vs. (Mixed) Factory Methods

- Used by multiple classes that do not have to be related by an IS-A relationship
- Creates a global configuration
- Creates local configurations.
- If class C implements factory method defined in super class S, then configuration applies to C, and all subclasses of C and superclasses between C and S that do not override/implement it
**Factory Principle**

Keep code that creates and uses an instance in separate methods

Instantiate a class in a special method that does nothing other than instantiating the class and possibly calling methods that initialize the state of the object

The method can be in a special factory class that provides only factory methods or an arbitrary class

Makes it easier to instantiate and substitute classes
FACTORY USES

Makes it easier to instantiate and substitute classes
COUNTING COUNTERS

public interface Counter {
    public void add (int amount);
}

public class AShortCounter implements Counter {
    short counter;
    public AShortCounter (short initValue) {
        counter = initValue;
    }
}

public class AnIntCounter implements Counter {
    int counter;
    public AnIntCounter (short initValue) {
        counter = initValue;
    }
}

How do we count the number of instances of counters that are created?

Create a special counter (that is not counted) to count the other counters

The constructor of classes of other counters increment the special counter

The counter can be used for anyone interested in the count
public class AnInstanceCountingShortCounter implements Counter {
    short counter;
    public AnInstanceCountingShortCounter (short initValue) {
        counter = initValue;
    }
    public void add (int amount) {
        counter += amount;
    }
    public int getValue() {
        return counter;
    }
}
public class AnInstanceCountingIntCounter implements Counter {
    int counter;

    public AnInstanceCountingIntCounter (short initValue, Counter anInstanceCounter) {
        counter = initValue;
        anInstanceCounter.add(1);
    }

    public void add (int amount) {
        counter += amount;
    }

    public int getValue() {
        return counter;
    }
}
public class AnInstanceCountingShortCounter implements Counter {
    int counter;

    public AnInstanceCountingShortCounter(short initValue, Counter anInstanceCounter) {
        counter = initValue;
        anInstanceCounter.add(1);
    }

    public void add(int amount) {
        counter += amount;
    }

    public int getValue() {
        return counter;
    }
}
public class AnInstanceCountingIntCounterFactory implements InstanceCountingCounterFactory {

    public Counter createCounter(short initValue, Counter anInstanceCounter) {
        return new AnInstanceCountingIntCounter(initValue, anInstanceCounter);
    }

    public Counter createCounter(Counter anInstanceCounter) {
        return createCounter((short) 0, anInstanceCounter);
    }
}

COUNTING INT FACTORY
public interface InstanceCountingCounterFactory {
    public Counter createCounter (short initValue, Counter anInstanceCounter);
    public Counter createCounter (Counter anInstanceCounter);
}
public class InstanceCountingCounterFactorySelector {
    static InstanceCountingCounterFactory counterFactory;

    public static InstanceCountingCounterFactory getCounterFactory() {
        return counterFactory;
    }

    public static void setCounterFactory(
            InstanceCountingCounterFactory aCounterFactory) {
        counterFactory = aCounterFactory;
    }
}
public static void main (String[] args) {
    Counter instanceCounter = new AnInstanceCountingCounter((short)0);
    InstanceCountingCounterFactorySelector.setCounterFactory(new AnInstanceCountingShortCounterFactory());
    CourseVisits aCourseVisits = new AnInstanceCountingCourseVisits(instanceCounter);
    aCourseVisits.mixVisited("anonymous");
    InstanceCountingCounterFactorySelector.setCounterFactory(new AnInstanceCountingShortCounterFactory());
    aCourseVisits = new AnInstanceCountingCourseVisits(instanceCounter);
    aCourseVisits.mixVisited("anonymous2");
    System.out.println("Num instances:" + instanceCounter.getValue());
}
How do we count the number of instances of counters that are created?

Create a special counter (that is not counted) to count the other counters.

The constructor of classes of other counters increment the special counter.

The counter can be used for anyone interested in the count.

Must change the body of counter constructors to increment instance counter.

Must change code that accesses instance counter values.

Also had to change counter constructor parameters, factory interface, factory implementations, factory selector, factory selector setter caller.

Make minimal changes?
## Global Counter

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make the instance counter a global object like System.in or System.out</td>
</tr>
<tr>
<td>Accesses through a getter rather than public variable</td>
</tr>
<tr>
<td>Create it on demand, only if accessed</td>
</tr>
<tr>
<td>Factory method creates the counter and returns it</td>
</tr>
</tbody>
</table>
public class InstanceCountingCounterSingletonFactory {
    static Counter instanceCounter;
    public static Counter getCounter() {
        if (instanceCounter == null) {
            instanceCounter = new AnInstanceCountingCounter((short)0);
        }
        return instanceCounter;
    }
}
public class AShortCounter implements Counter {
    short counter;
    public AShortCounter (short initValue) {
        counter = initValue;
        InstanceCountingCounterSingletonFactory.getCounter().add(1);
    }
    public void add (int amount) {
        counter += amount;
    }
    public int getValue() {
        return counter;
    }
}
public class AnIntCounter implements Counter {
    int counter;

    public AnIntCounter (short initValue) {
        counter = initValue;
        InstanceCountingCounterSingletonFactory.
        getCountCounter().add(1); }

    public void add (int amount) {
        counter += amount;
    }

    public int getValue() {
        return counter;
    }
}
public static void main (String[] args) {
    StaticCounterFactorySelector.setCounterFactory(new AShortCounterFactory());
    CourseVisits aCourseVisits =
        new AFactorySelectorUsingCourseVisits();
    aCourseVisits.mixVisited("anonymous");
    StaticCounterFactorySelector.setCounterFactory(new AnIntCounterFactory());
    aCourseVisits =
        new AFactorySelectorUsingCourseVisits();
    aCourseVisits.mixVisited("anonymous2");
    System.out.println("Num instances: "+
        InstanceCountingCounterSingletonFactory.getCounter().getValue());
}
**Singleton?**

```java
public class InstanceCountingCounterSingletonFactory {
    static Counter instanceCounter;
    public static Counter getCounter() {
        if (instanceCounter == null) {
            instanceCounter = new AnInstanceCountingCounter((short)0);
        }
        return instanceCounter;
    }
}
```

Only one instance of a Singleton class (expected to be) instantiated in an application

Can make constructor of any class non public to ensure only a factory in the same package can instantiate it
public class AnInstanceCountingShortCounter implements Counter {
    short counter;

    public AnInstanceCountingShortCounter (short initValue) {
        counter = initValue;
    }

    public void add (int amount) {
        counter += amount;
    }

    public int getValue() {
        return counter;
    }
}

Can make constructor of any class non public to ensure only a factory (method) in the same package can instantiate it
**COMMON APPROACH**

```java
public class ASingletonCounter implements Counter {
    short counter;

    private ASingletonCounter (short initValue) {
        counter = initValue;
    }

    public void add (int amount) {
        counter += amount;
    }

    public int getValue() {
        return counter;
    }

    static Counter instance;

    public static Counter getInstance() {
        if (instance != null) {
            instance = new ASingletonCounter ((short) 0);
        }
        return null;
    }
}
```

No other class can create multiple instances

No separation of concerns and assumes no alternative class exists
FACTORY USES

- Makes it easier to instantiate and substitute classes
- Makes it possible to create global objects on demand
- Can be used to force singletons
JAVA EXAMPLE

LineBorder blackline = BorderFactory.createLineBorder(Color.black);

LineBorder blackline = new LineBorder(Color.black);

Factory can return a single instance of LineBorder for all black line borders
Swing/AWT Substitution

Swing Widgets: JFrame, JPanel, JTextField

AWT Widgets: Frame, Panel, TextField
Multiple toolkits provide same kind of widgets with different look and feel/implementations.

- **Package java.awt**
  - TextField, Button, Panel

- **Package javax.swing**
  - JTextField, JButton, JPanel

Could define a common factory interface

- getTextField(), getButton(), getPanel()

Java does not define common interfaces
ObjectEditor provides a layer that unites SwingFactory and AWTFactory classes implement interface
FactorySelector switches between two sets of classes to change implementation
public static void main (String[] anArgs) {
    BMISpreadsheet aBMISpreadsheet = new ABMISpreadsheet();
    VirtualToolkit.setDefaultToolkit(new SwingToolkit());
    ObjectEditor.edit(aBMISpreadsheet);
    VirtualToolkit.setDefaultToolkit(new AWTToolkit());
    ObjectEditor.edit(aBMISpreadsheet);
}
Swing Toolkit

TextFieldSelector.setTextFieldFactory(new SwingTextFieldFactory());
PanelSelector.setPanelFactory(new SwingPanelFactory());
FrameSelector setFrameFactory(new SwingFrameFactory());

Single class ensures matching objects created
AWT Toolkit

TextFieldSelector.setTextFieldFactory(new AWTTextFieldFactory());
PanelSelector.setPanelFactory(new AWTPanelFactory());
FrameSelector setFrameFactory(new AWTFrameFactory());

Single class ensures matching objects created
**Defining our own Factory**

```java
public class MySwingFrameFactory extends SwingFrameFactory implements FrameFactory {
    @Override
    protected JFrame createJFrame() {
        JFrame aJFrame = new JFrame();
        aJFrame.setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
        return aJFrame;
    }
}
```

Factory class with overriddable factory method

Factory method
public static void main (String[] anArgs) {
    BMISpreadsheet aBMISpreadsheet = new ABMISpreadsheet();
    ObjectEditor.edit(aBMISpreadsheet);
    FrameSelector setFrameFactory(new MySwingFrameFactory());
    ObjectEditor.edit(aBMISpreadsheet);
}
VIDEO
FACTORY USES

- Makes it easier to instantiate and substitute classes
- Makes it possible to create global objects on demand
- Can be used to force singletons
- Can be used to ensure compatible classes instantiated
FACTORY USES

- Should we always instantiate via factories?
- Factory classes add overhead
  - Factory interfaces, classes
  - Factory selector interfaces, classes
- If not using Factory classes, at least use factory methods
**Classes vs. Factory**

- We also called a class a factory
  - It defines blueprints for its instances
- Factory methods and classes are broker that orders objects for you.
- Factory selector decides between different kinds of brokers
- Analogy
  - I ask my IT department to get me a 4lb laptop
  - They decide to go to the CCI “factory”
  - CCI factory specifies matching computer and accessories
  - These are then ordered from the real factory
- Car Analogy
  - Dealership selling you cars and accessories that go with them.
FACTORIES AND INTERFACES

- Factories allow us to switch between alternative objects providing same methods
  - AShortCounter and AnIntCounter
  - JTextField and TextField

- Alternative objects must be united by a common interface

- Otherwise common factory interface cannot be defined.

- Moral: define interfaces!