

## Factory Methods

A *Factory Method* is a method that produces new objects. Factory methods are often used by frameworks and toolkits to create objects that provide access to the framework.

Factory methods are also used to control the *type* of actual object created.

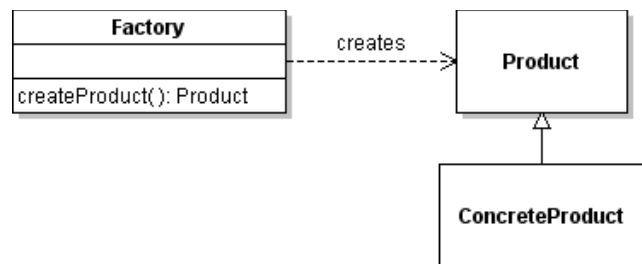
The Java Calendar class provides a simple factory method. The class designers want to allow for different types of Calendars depending on locale. To create a Calendar use:

```
Calendar cal = Calendar.getInstance();
```

In fact, the object it creates is *not* an instance of Calendar. It is a **GregorianCalendar** (a subclass).

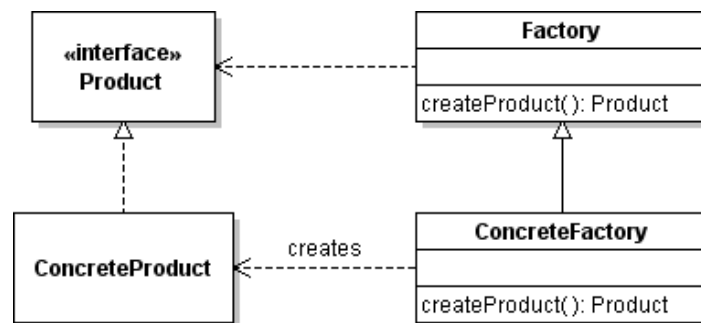
## UML for Simple Factory Method

A simple factory method, such as `Calendar.getInstance()`, has a structure like:



## Factory Method Pattern

The *Factory Method Pattern* is a little more than a simple factory method. In the pattern, the factory method is defined in an *interface* (or abstract class) and the Product is also an interface.



## Why Use Factory Methods?

There are several situations where factory methods are useful:

1. creating objects is complex. A factory method can handle the complexity so the user's code does not have to. Calendar is an example: the Calendar properties need to match the user's locale and time zone.
2. you want to dynamically change the type of object created. A Factory Method can create objects of different classes. It can use information from the environment to decide at run-time what kind of object to create.
3. you can limit how many objects are created. A *Singleton* looks like a factory method, but always returns the same instance.

## Java Media Framework and MP3 Player

The Java Media Framework (JMF) is a framework for playing multimedia, including MP3. To play an MP3 file you must create a Player object. JMF creates different *kinds* of Player depending on what you want to play (MP3, MPEG, AVI) and the *codecs* on your system.

To make it easy to create the correct kind of Player, the framework has a factory class named **Manager** to create Player objects.

```

import java.net.URL;
import javax.media.*;
...
URL url = new URL("file:///d:/music/somefile.mp3");
Player player = Manager.createRealizedPlayer( url );
player.start();
// plays the mp3 file
player.stop();

```

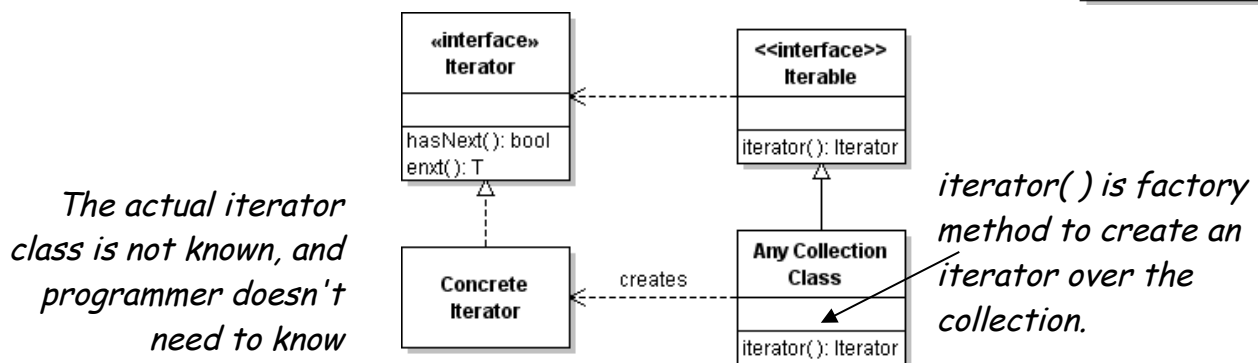
A Player *may* have a visual control panel (that extends java.awt.Component) for use in GUI interface:

```
Component controlpanel = player.getControlPanelComponent();
```

This is another factory method. It creates a ControlPanel based on the type of player. May return null.

### Example: Iterable and Iterator

Many kinds of objects *create* Iterators. The *Iterable* interface defines a factory method for creating an *Iterator*. All Collection classes are *Iterable*. This enables us to write software that uses an iterator without knowing how the Iterator is created or *what* the actual Iterator class is.



```

List<String> list = new ArrayList<String>( );
list.add( "dog" );
...
// create an iterator
Iterator<String> iter = list.iterator( );
while ( iter.hasNext() ) System.out.println( iter.next() );

```

A for-each loop requires an *Iterable* object as argument. It implicitly creates an *Iterator* to loop over the elements:

```
for( String item : list ) System.out.println( item );
```

### Example: MIDI System

The `javax.sound.midi` package contains classes for controlling the computer's sound system. The sound system provides **Synthesizers**, **Sequencers**, and **Receivers**. To play notes you can use a Synthesizer. But Synthesizer is just an *interface*. How do you create a concrete Synthesizer *object* for your hardware?

The **MidiSystem** class contains several factory methods, including `getSynthesizer`:

```
Synthesizer synthesizer = MidiSystem.getSynthesizer();
```

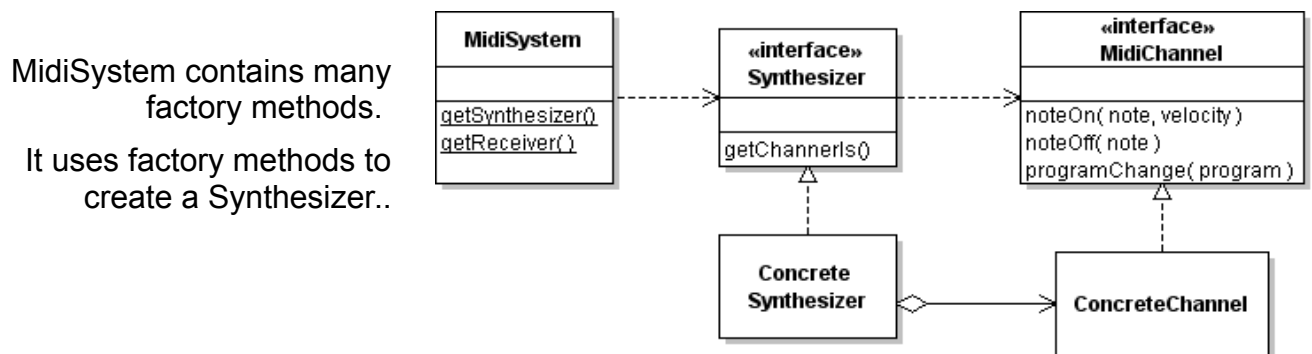
The **Synthesizer** interface has its own factory methods for getting **Soundbanks** and **Synthesizer Channels**.

```
synthesizer.open();
Soundbank soundbank = synthesizer.getDefaultSoundbank();
synthesizer.loadAllInstruments( soundbank );
// get a channel so we can play notes
MidiChannel channel = synthesizer.getChannels()[0];
```

Play a note using numbers 0 - 127. Middle C is note 60.

```
channel.noteOn( 60, 200 ); // 200 is "velocity" of the note.
```

We can use the **MidiSystem** without knowing *any* concrete classes that the objects belong to. This is possible because of factory methods and interfaces describe a general Midi system.



### Example: Logging and slf4j

slf4j is a Logging framework that *adapts* other Logging frameworks. To create a Logger object in a particular class, you write:

```
Logger logger = LoggerFactory.getLogger( MyClass.class );
logger.warn( "this is a warning message" );
logger.info(
    "this Logger is really a " + logger.getClass().getName() );
```

slf4j can use the JDK `java.util.logging` classes, the well-known Log4J logger, "simple" logging that prints to `System.out`, or "No-op" logging (does nothing) as the underlying logging program. The choice depends on which JAR file you include in your project: if you include `slf4j-simple.jar` it uses simple logging, if you include `slf4j-log4j12.jar` it uses Log4J, etc. The `LoggerFactory` makes the decision at run-time based on what it finds on the classpath.

### How to Dynamically "Program" a Factory?

To write a factory that can change the kind of objects it creates at runtime *without changing the Java code*, there are several common techniques:

1. *Register* a concrete factory with the abstract factory class. The abstract factory chooses among available concrete factories when an object is requested.
2. *Dynamically load* a factory class using configuration information from a properties file.

3. Use the **ServiceLoader** class (JDK 6 and above) to locate available *service provider classes* (classes that implement a "service" interface). The **ServiceLoader** class uses information from JAR files (in the META-INF/services directory) to locate available service provider classes. The JDBC drivers use this mechanism.

**Example:**

Suppose we have an *interface* named **Factory**. We also have an **AbstractFactory** class with a static method named `getInstance( )` that returns a concrete **Factory** object:

```
Factory myfactory = AbstractFactory.getInstance( );
```

We can change the actual type of the object returned by `getInstance( )` by having **AbstractFactory** read the name of the actual factory class (at runtime) and create a new object of this factory class.

How does **AbstractFactory** get the name of the concrete factory to create? One way is to use a property, either a system property or your own properties file. You can choose any property name (being careful to avoid names of existing system properties). Let's use the property name `factory.name`.

We can create a load the new **Factory** class at run-time and create an object by using code like this:

```
public abstract class AbstractFactory {
    public static Factory getInstance() {
        String factoryclass =
            System.getProperty( "factory.name" );
        //TODO this may throw many exceptions. Catch them.
        Factory factory =
            (Factory) Class.forName(factoryclass).newInstance( );
        return factory;
    }
}
```

What is not shown above is code to catch exceptions, and there should be a "default" factory class to use in case the `factory.name` property is not set or can't be used.