Creational Design Patterns

CSC 335: Object-Oriented Programming and Design

Outline

- Three Creational Design Patterns
  - Singleton
  - Factory
  - Prototype

To use new or to not use new? That is the question.

- Since most object-oriented languages provide object instantiation with new and initialization with constructors
- There may be a tendency to simply use these facilities directly without forethought to future consequences
- The overuse of this functionality often introduces inflexibility in the system

Creational Patterns

- Creational patterns describe object-creation mechanisms that enable greater levels of reuse in evolving systems: Builder, Singleton, Prototype
- The most widely used is Factory
- This pattern calls for the use of a specialized object solely to create other objects

OO Design Pattern
Singleton

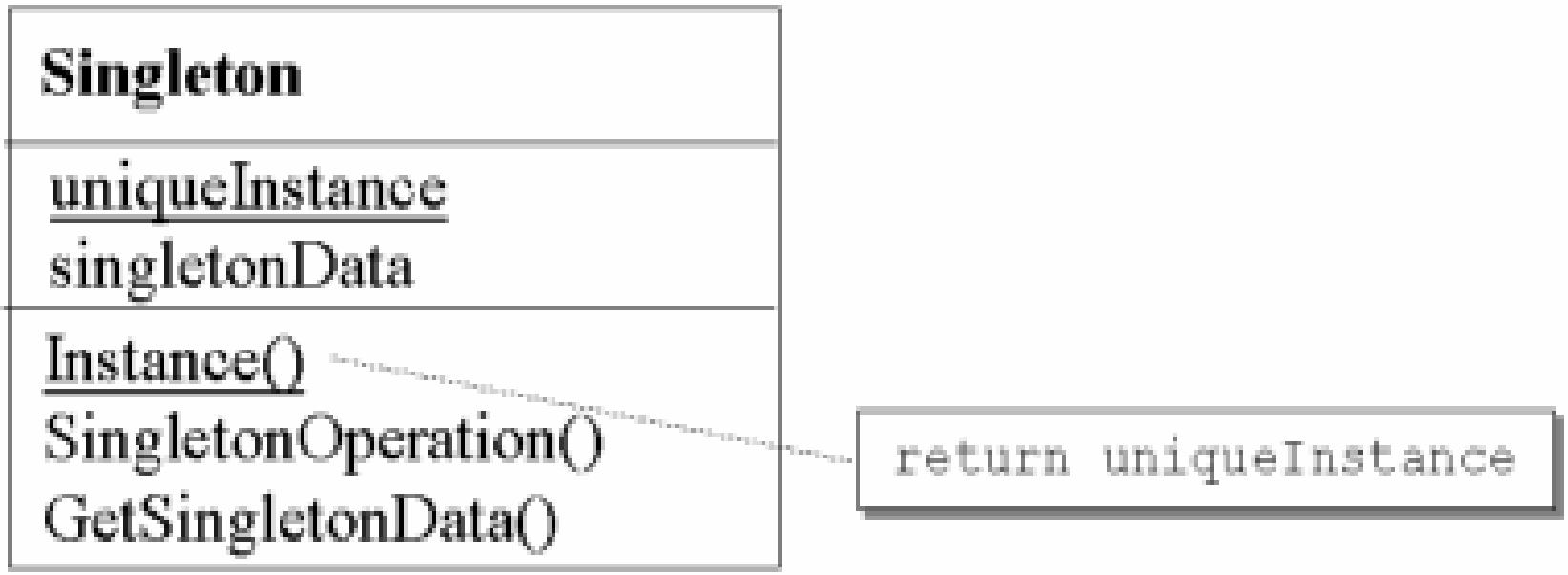
Recurring Problem

- Some classes have only one instance. For example, there may be many printers in a system, but there should be only one printer spooler
- How do we ensure that a class has only one instance and that instance is easily accessible?

Solution

- Have constructor return the same instance when called multiple times
- Takes responsibility of managing that instance away from the programmer
  - It is simply not possible to construct more instances

UML General form as UML



Java Code General Form

```
// NOTE: This is not thread safe!
public class Singleton {

  private static Singleton uniqueInstance;
  // other useful instance variables here

  private Singleton() {}

  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }

  // other useful methods here
}
```

Example Used in a final project names changed to protect identity

```
/** This class is a DECORATOR of ArrayList. Its purpose is to make
  * sure there are no duplicate names anywhere in the universe.
  * That's why it's SINGLETON; because many classes use it but
  * there should be only one.  */
public class NamesList implements Serializable {
  private ArrayList<String> npcNames;
  private static NamesList self;

  private NamesList() {
    npcNames = new ArrayList<String>();
  }

  public static syncronized NamesList getInstance() {
    if (self == null) {
      self = new NamesList();
    }
    return self;
  }
```

OO Design Pattern
Factory Method

- **Name**: Factory Method
- **Problem**: A Client needs an object and it doesn't know which of several objects to instantiate
- **Solution**: Let an object instantiate the correct object from several choices. The return type is an abstract class or an interface type.

Characteristics

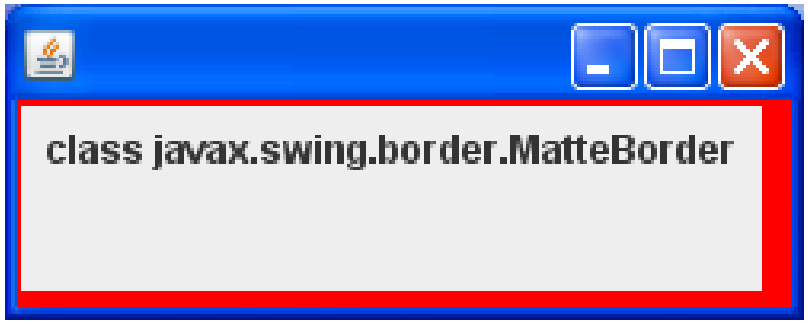
- A method returns an object
- The return type is an abstract class or interface
- The interface is implemented by two or more classes or the class is extended by two or more classes

Example from Java

- Border is an interface
- AbstractBorder is an abstract class that implements Border
- BorderFactory has a series of static methods returning different types that implement Border
  - This hides the implementation details of the subclasses
- Factory methods such as `createMatteBorder createEthedBorder createTitleBorder` directly call constructors of the subclasses of AbstractBorder

One type

```
JFrame f = new JFrame();
f.setSize(250, 100);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel toBeBordered = new JPanel();
Border border = BorderFactory.createMatteBorder(2,1,5,9,Color.RED);
toBeBordered.add(new JLabel("" + border.getClass()));
toBeBordered.setBorder(border);

f.getContentPane().add(toBeBordered);
f.setVisible(true);
```

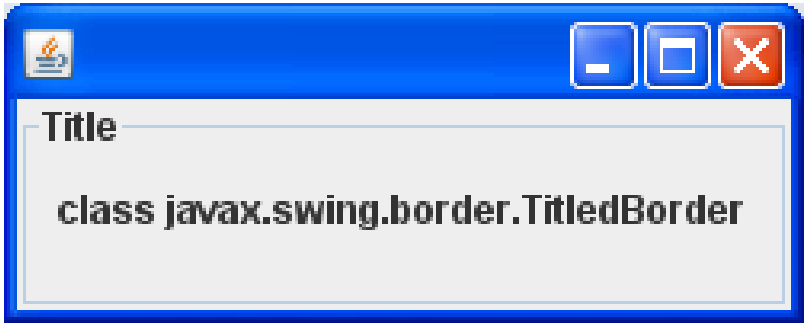


Another type

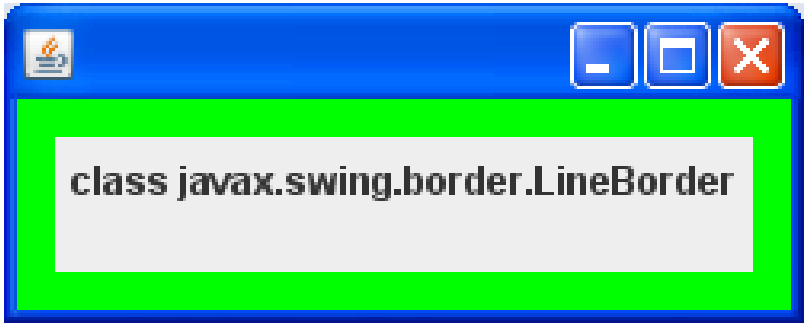
```
JFrame f = new JFrame();
f.setSize(250, 100);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel toBeBordered = new JPanel();
Border border = BorderFactory.createEtchedBorder();
toBeBordered.add(new JLabel("" + border.getClass()));
toBeBordered.setBorder(border);

f.getContentPane().add(toBeBordered);
f.setVisible(true);
```

Two others

```
Border border = BorderFactory.createTitledBorder("Title");
```




```
Border border = BorderFactory.createLineBorder(Color.GREEN, 12);
```



Lots of Subclasses

javax.swing.border.AbstractBorder

java.lang.Object

javax.swing.border.AbstractBorder

All Implemented Interfaces:

Serializable, Border

Direct Known Subclasses:

BasicBorders.ButtonBorder, BasicBorders.FieldBorder, BasicBorders.MarginBorder, BasicBorders.MenuBarBorder, BevelBorder, CompoundBorder, EmptyBorder, EtchedBorder, LineBorder, MetalBorders.ButtonBorder, MetalBorders.Flush3DBorder, MetalBorders.InternalFrameBorder, MetalBorders.MenuBarBorder, MetalBorders.MenuItemBorder, MetalBorders.OptionDialogBorder, MetalBorders.PaletteBorder, MetalBorders.PopupMenuBorder, MetalBorders.ScrollPaneBorder, MetalBorders.TableHeaderBorder, MetalBorders.ToolBarBorder, TitledBorder

Iterators

- The iterator methods isolate the client from knowing the class to instantiate

```
List<String> list = new ArrayList<String>();
Iterator<String> itr = list.iterator();
System.out.println(itr.getClass().toString());
```

- What type is `itr`?

```
class java.util.AbstractList$Itr
```

- What type is `itr` with this change?

```
List<String> list = new LinkedList<String>();
```

Do we need new?

- Objects can be returned without directly using new

```
double amount = 12345.1234656789457;
NumberFormat formatter =
          NumberFormat.getCurrencyInstance();
System.out.println(formatter.format(amount));
```

Output if the computer is set to US Locale

```
$12,345.12
```

Change the computer setting to Germany Locale and we get this:

```
12.345,12 €
```

What Happened?

- `getCurrencyInstance` returns an instance of `DecimalFormat` where methods like `setCurrency` help build the appropriate object
  - It encapsulates the creation of objects
- Can be useful if the creation process is complex, for example if it depends on settings in configuration files or the jre or the OS

Behind the scenes

- Client: `main` method
- Factory Method: `getCurrencyInstance`
- Product: a properly configured instance of `DecimalFormat`
- This is another example of Factory in use

Design Pattern
Prototype

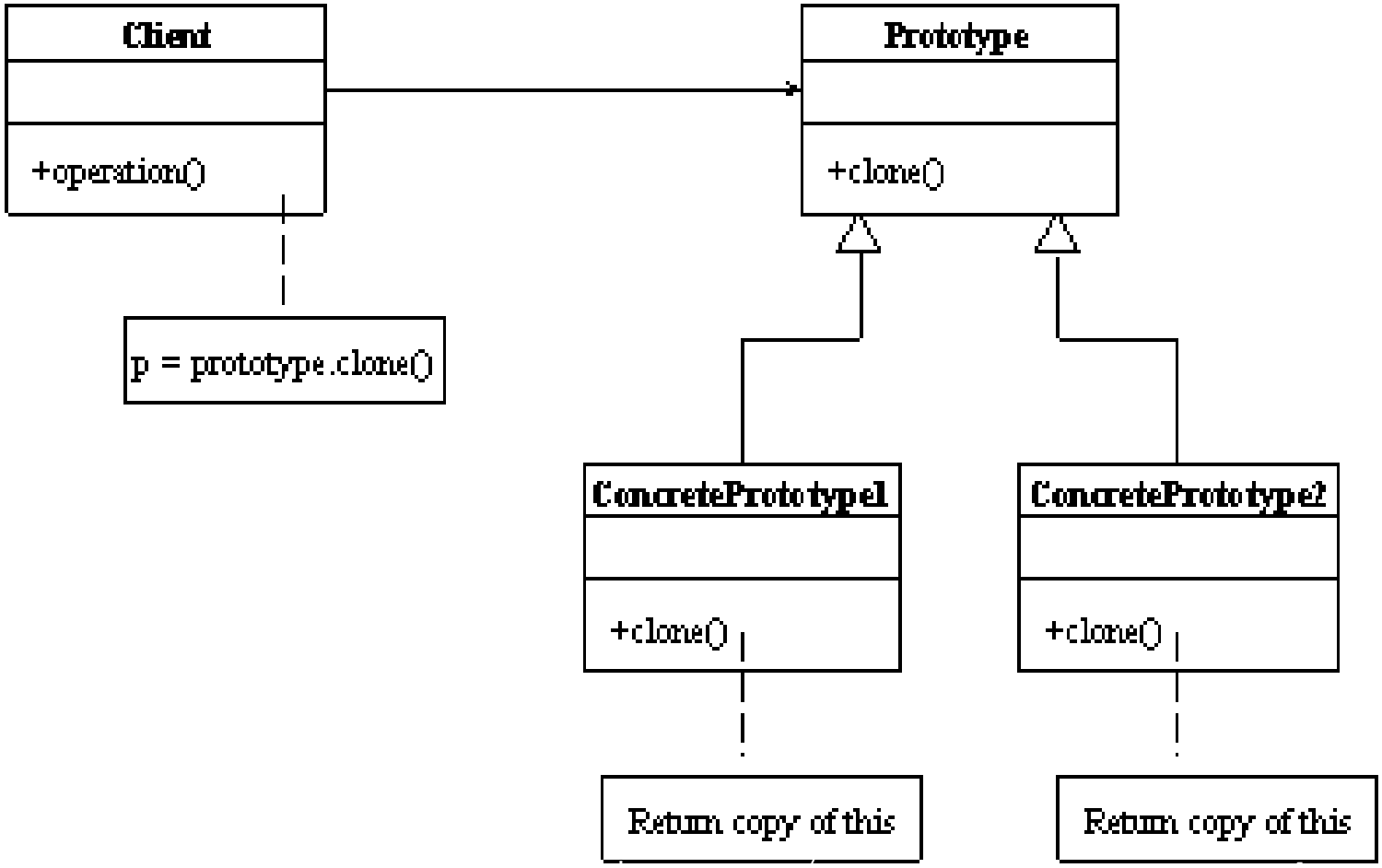
Recurring Problem

- Each product in family in Factory patterns is a subclass. How to avoid implementing so many subclasses?

Solution

- Use a master instance object as a prototype to construct other instances (clone) instead of using `new`.

UML for the Prototype Pattern



Participants: It is all about cloning

Prototype

Defines interfaces of objects to be created

Defines a clone( ) method

ConcretePrototype

Implements the Prototype interface

Implements clone( )

Client

Creates new instances by cloning the prototype

Cloning in Java

- Classes that are associated with instances that can be cloned must implement the java.lang.Cloneable interface
- This is a marker interface: there are no methods in Cloneable
- It is a contract that states that all fields are cloneable. This will be checked by the compiler
- `Object clone()` is a method of Object
- If the object is not cloneable, jvm throws a new `CloneNotSupportedException()`

Use Example: Text Messages

```
public class TextMsg implements Cloneable {
  private String myText;
  private int priority;

  public TextMsg(String textIn, int priorityIn) {
    myText = textIn;
    priority = priorityIn;
  }

  public void setText(String textIn) {
    myText = textIn;
  }

  public String toString() {
   return priority + ":" + myText;
  }

  public TextMsg clone(int newPriority) {
    return new TextMsg(myText, newPriority);
  }
}
```

Use Example: Text Messages

```
public class TextMessageRunner {

  public static void main(String[] args) {
    TextMsg toBoss = new TextMsg("Hello Sir", 1);
    TextMsg toWife = toBoss;
    TextMsg toFriend = toBoss.clone(2);

    toWife.setText("Bye Honey");
    toFriend.setText("Hey, Pal");

    System.out.println(toBoss);   // 1:Bye, Honey
    System.out.println(toWife);   // 1:Bye, Honey
    System.out.println(toFriend); // 2:Hey, Pal
  }
}
```

Shallow versus Deep Copies

`Object a = Object b`

- Makes a *shallow copy* (copies the *reference to* b into a)
- Changes to a or b will effect a and b.

`Object clone()`

- Makes a *deep copy* (byte-for-byte)
- Primitives within cloned object are copied by value
- Objects within cloned object are copied by reference
- Can define own clone( ) method with a different semantics, as done in TextMsg example. If you don't do this, you will have to cast from Object

Summary

- Used when classes only differ in properties, not behavior
- The cloned object copies the state of the original object