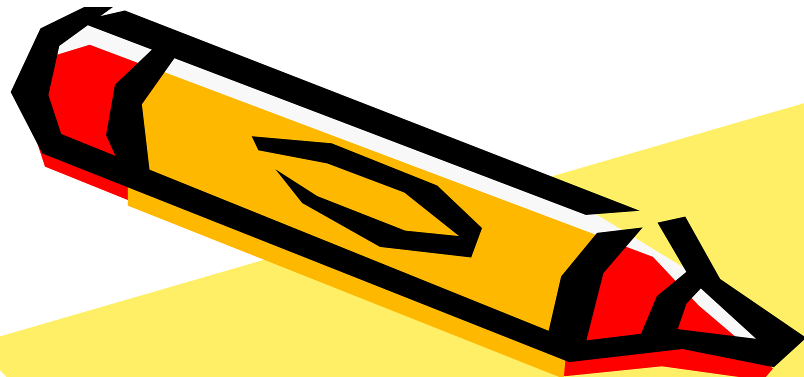
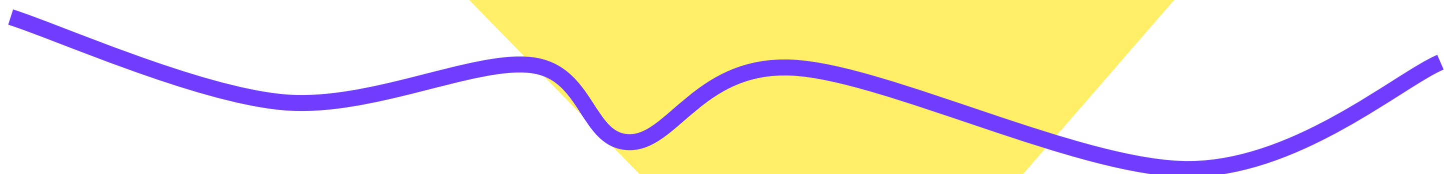
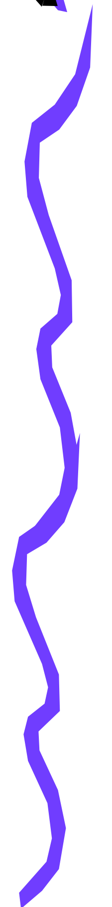


# More design patterns

The ADAPTER Pattern
Actions and the COMMAND Pattern

The FACTORY METHOD Pattern
The PROXY Pattern
The SINGLETON Pattern



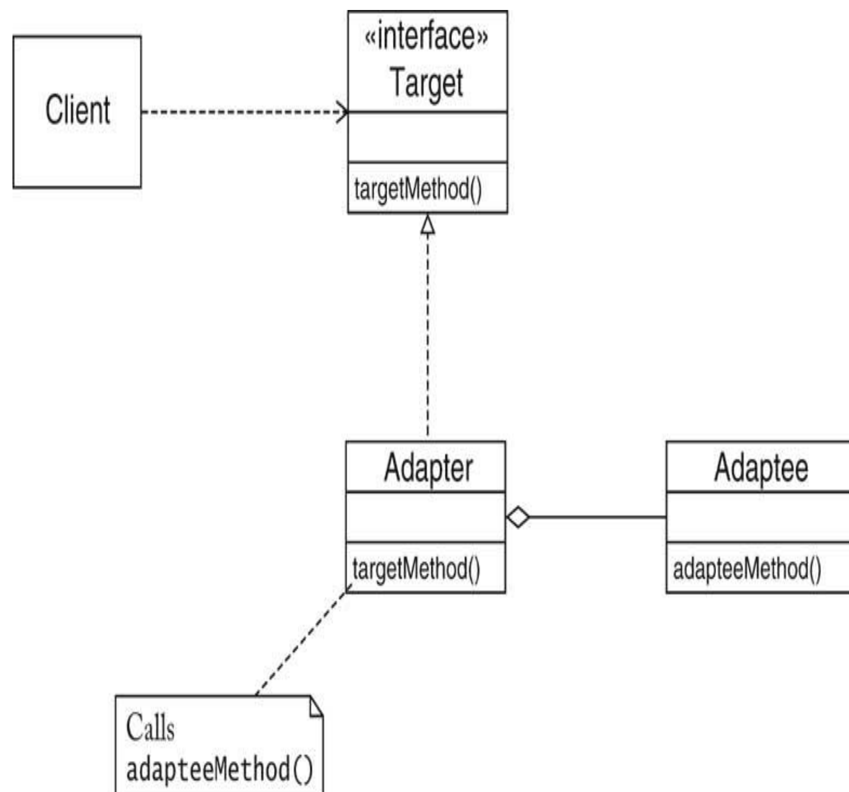


# Adapter pattern

- Want to adapt class to foreign interface type?
- Example: Add CarIcon to container
- Problem: Containers take components, not icons
- Solution: Create an adapter that adapts Icon to Component
- Ch10/adapter/IconAdapter.java
- Ch10/adapter/IconAdapterTester.java

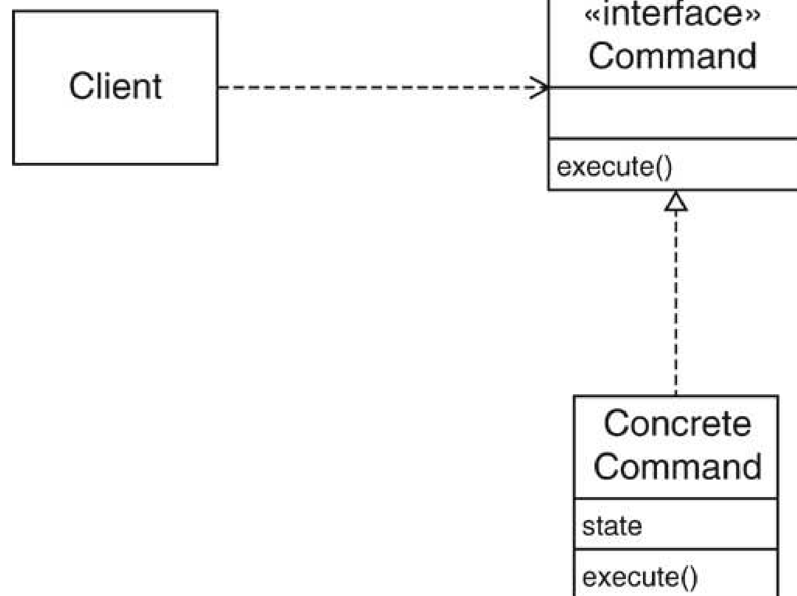
# The ADAPTER Pattern Solution

- Define an adapter class that implements the target interface.
- The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods.
- The client wraps the adaptee into an adapter class object.



# The COMMAND Pattern

- When you wuant to implement commands that behave like objects
  - because you need to store additional information with commands
  - because you want to collect commands.
  - **Solution**
  - Define a command interface type with a method to execute the command.
  - Supply methods in the command interface type to manipulate the state of command objects.
  - Each concrete command class implements the command interface type.
  - To invoke the command, call the execute method



# Factory method

Name Divider

Enter name:

Smith, Sandy

First name: Sandy

Last name: Smith

Compute | Clear | Close

```
NameFactory nfactory = new NameFactory();
private void computeName() {

//send the text to the factory and get a class
back
namer = nfactory.
getNamer(entryField.getText());

//compute the first and last names
//using the returned class
txFirstName.setText(namer.getFirst());
txLastName.setText(namer.getLast());
}
```

```
class NameFactory {
//returns an instance of LastFirst or FirstFirst
//depending on whether a comma is found
```

```
public Namer getNamer(String entry) {

int i = entry.indexOf(","); //comma determines name
order
if (i>0)
return new LastFirst(entry); //return one class
else
return new FirstFirst(entry); //or the other
}
}
```

```
class Namer {
//a simple class to take a string apart into two names

protected String last; //store last name here
protected String first; //store first name here

public String getFirst() {
return first; //return first name
}
public String getLast() {
return last; //return last name
}
}
```

```
class FirstFirst extends Namer {
    //split first last
public FirstFirst(String s) {
int i = s.lastIndexOf(" ");

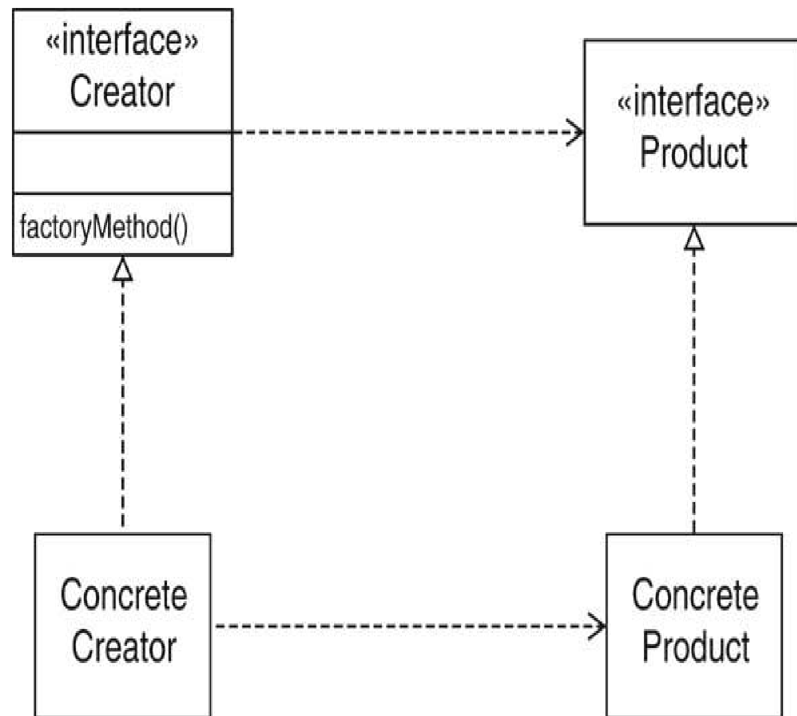
if (i > 0) {
//left is first name
first = s.substring(0, i).trim();
//right is last name
last =s.substring(i+1).trim();
}
else {
first = ""; // put all in last name
last = s; // if no space
}
}
```

```
class LastFirst extends Namer {
//split last, first
public LastFirst(String s) {
int i = s.indexOf(",");

if (i > 0) {
//left is last name
last = s.substring(0, i).trim();
//right is first name
first = s.substring(i + 1).trim();
}
else {
last = s; // put all in last name
first = ""; // if no comma
}
}
}
```

# The FACTORY METHOD Pattern

- A Factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it.
- Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data.
- **Context**
- A type (the creator) creates objects of another type (the product).
- Subclasses of the creator type need to create different kinds of product objects.
- Clients do not need to know the exact type of product objects.

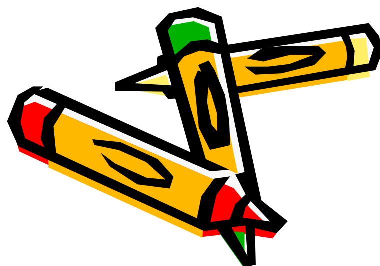


# Factory Pattern

- **When to Use a Factory Pattern**
- You should consider using a Factory pattern when
- · A class can't anticipate which kind of class of objects it must create.
- · A class uses its subclasses to specify which objects it creates.
- · You want to localize the knowledge of which class gets created.

- **There are several similar variations on the factory pattern to recognize.**
- 1. The base class is abstract and the pattern must return a complete working class.

- 2. Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.



# PROXIES

Example: Delay instantiation of object

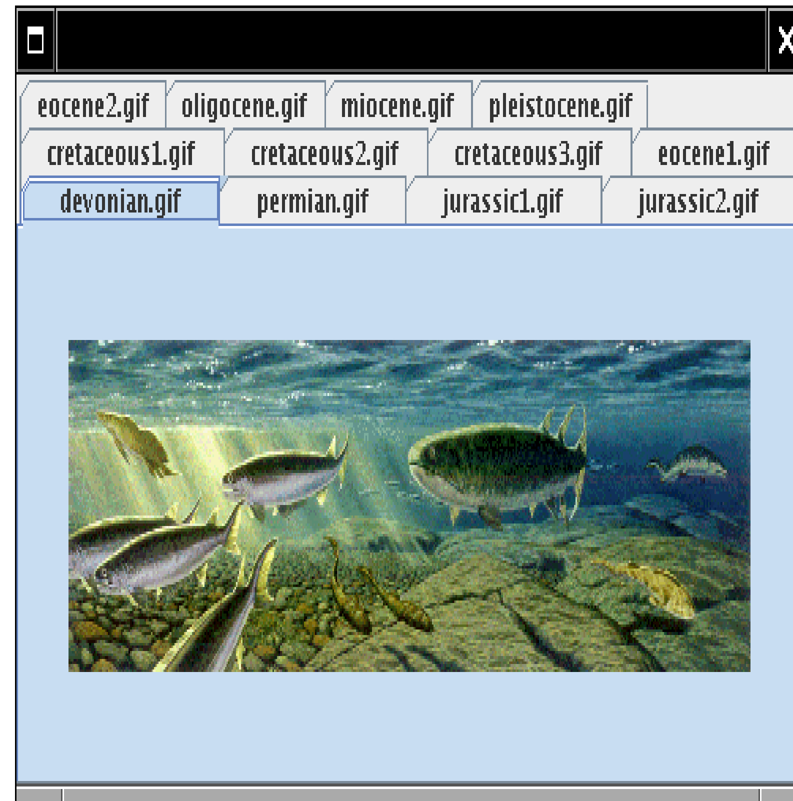
- Expensive to load image
- Not necessary to load image that user doesn't look at
- Proxy defers loading until user clicks on tab

eocene2.gif oligocene.gif miocene.gif pleistocene.gif
cretaceous1.gif cretaceous2.gif cretaceous3.gif eocene1.gif
devonian.gif permian.gif jurassic1.gif jurassic2.gif

# Proxies

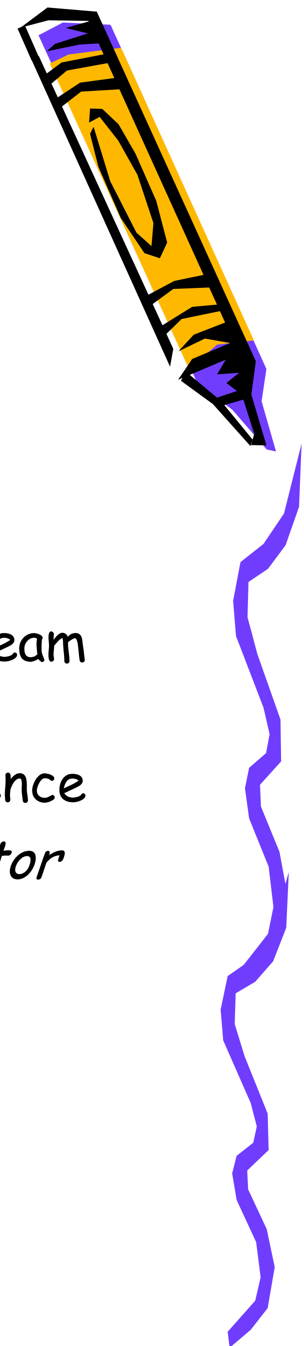
- Normally, programmer uses image for label:
- JLabel label = new JLabel(new ImageIcon(imageName));
- Use proxy instead:
  JLabel label = new JLabel(new ImageProxy(imageName));
- paintIcon loads image if not previously loaded
- public void paintIcon(Component c, Graphics g, int x, int y)

```
{
   if (image == null) image = new ImageIcon(name);
   image.paintIcon(c, g, x, y);
}
```

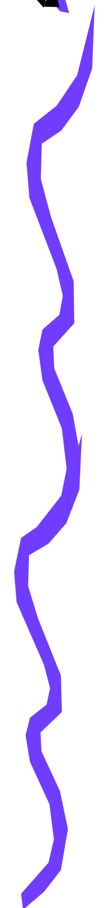


# Singletons

- Singleton class = class with one instance
- "Random" number *generator* generates predictable stream of numbers by using seed
- Convenient for debugging: can reproduce number sequence
- Only if all clients use *the same random number generator*



# Random Number Generator Singleton



- ```
  public class SingleRandom
  {
     private SingleRandom() { generator = new Random(); }
     public void setSeed(int seed) { generator.setSeed(seed); }
     public int nextInt() { return generator.nextInt(); }
     public static SingleRandom getInstance()
     { return instance; }

     private Random generator;
     private static SingleRandom instance = new
     SingleRandom();
  }
  ```

  ```

# The SINGLETON Pattern

- **Context**
- All clients need to access a single shared instance of a class.
- You want to ensure that no additional instances can be created accidentally.
- **Solution**
- Define a class with a private constructor.
- The class constructs a single instance of itself.
- Supply a static method that returns a reference to the single instance.