CSE 331

Design Patterns 2: Prototype, Factory

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer
http://www.cs.washington.edu/331/

Pattern: Prototype

An object that serves as a basis for creation of others

Objects as prototypes

- **prototype**: An object that serves as a template or model to assist in the creation of other similar/equal objects.

- *Problem:* Client wants another object similar to an existing one, but doesn't care about the details of the state of that object.
  - Sometimes client doesn't even care about the object's exact type.

- *Solution:* Clone or copy the object's state into a new object, modify as needed, then use it.
  - Often closely related to Java's `clone` method.
  - Sometimes done with producer methods that return new objects.

  (Prototype is a creational pattern.)

Scenario: Store products

- Suppose a store has a hierarchy of classes representing products.

```
public class Product {...}
public class Book extends Product {...}
public class DVD extends Product {...}
```

  - The store keeps a large catalog of all products by ID.

- Customers want to buy products from the catalog and put them into their shopping carts.
  - The add-to-cart code doesn't want to worry about what kind of product is being bought, its state, etc.
  - We don't want to add the original product to the customer's cart, because it is mutable and they will modify its price/status/etc.

Prototype store products

- The following code gives each customer his own product copy:

```
// maps from product IDs to the products themselves
private Map<Integer, Product> catalog;
...
public void addToCart(ShoppingCart cart,
                      int id, double price) {
    Product p = catalog.get(id);
    p = p.clone();        // make a copy for this user
    p.setPrice(price);
    cart.add(p);
}
```

Prototype producer method

- Sometimes the object serves as a prototype by supplying producer method(s) that return new objects similar to itself:

```
public class Product implements Cloneable {
    ...
    public Product clone() { ... }

    // a new product like this one, but half price
    public Product halfPrice() {
        Product copy = this.clone();
        copy.setPrice(this.getPrice() / 2);
        return copy;
    }
}
```

Drawing fonts/colors

- Suppose we want to draw fonts/colors on a graphical window.
  - We will make use of a CSE 142/143 class, `DrawingPanel`.

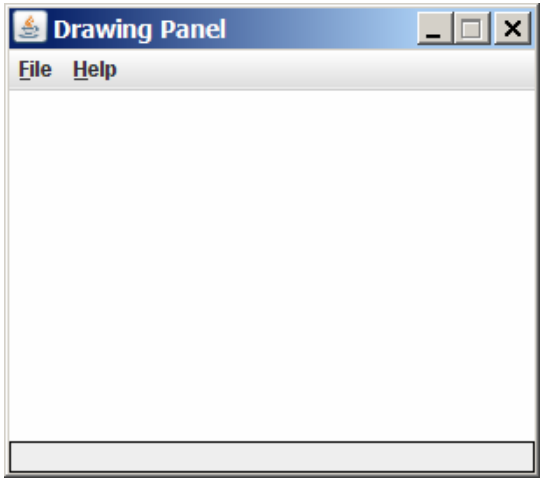
- To create a window:

```
DrawingPanel name = new DrawingPanel(width, height);
```

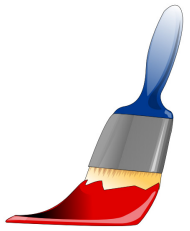
Example:

```
DrawingPanel panel = new DrawingPanel(300, 200);
```

- The window has nothing on it.
  - We draw shapes / lines on it with another object of type `Graphics`.



Graphics



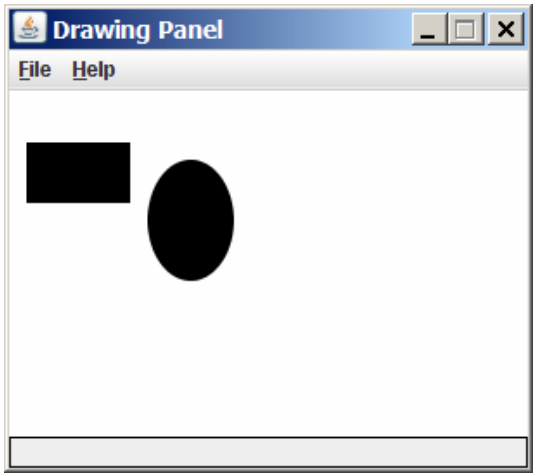
"Pen" or "paint brush" object to draw lines and shapes

- `import java.awt.*;  // needed to use Graphics`
- Access it by calling `getGraphics` on your `DrawingPanel`.

```
Graphics g = panel.getGraphics();
```

- Draw shapes by calling methods on the `Graphics` object.

```
g.fillRect(10, 30, 60, 35);
g.fillOval(80, 40, 50, 70);
```



Graphics methods

Method name	Description
`g.drawImage(Image, x, y, [w, h], panel);`	an image at the given x/y position and size
`g.drawLine(x1, y1, x2, y2);`	line between points (*x1*, *y1*), (*x2*, *y2*)
`g.drawOval(x, y, width, height);`	outline largest oval that fits in a box of size *width* * *height* with top-left at (*x*, *y*)
`g.drawRect(x, y, width, height);`	outline of rectangle of size *width* * *height* with top-left at (*x*, *y*)
`g.drawString(text, x, y);`	text with bottom-left at *(x, y)*
`g.fillOval(x, y, width, height);`	fill largest oval that fits in a box of size *width* * *height* with top-left at (*x*, *y*)
`g.fillRect(x, y, width, height);`	fill rectangle of size *width* * *height* with top-left at (*x*, *y*)
`g.setColor(color);`	paint any following shapes in the given color
`g.setFont(font);`	draw any following text with the given font

Specifying colors

- Pass a `Color` to `Graphics` object's `setColor` method.
  - Specified by constructor, using Red-Green-Blue (RGB) values 0-255:

```
Color brown = new Color(192, 128, 64);
```

  - Or use predefined `Color` class constants:

`Color`**.CONSTANT_NAME** where **CONSTANT_NAME** is one of:

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY,
MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW
```

  - Or create a new color, using an existing color as a *prototype*:

```
public Color brighter()
public Color darker()
```

Specifying fonts

- Pass a `Font` to `Graphics` object's `setFont` method.
  - Specified by the `Font` constructor:

```
public Font(String name, int style, int size)
```

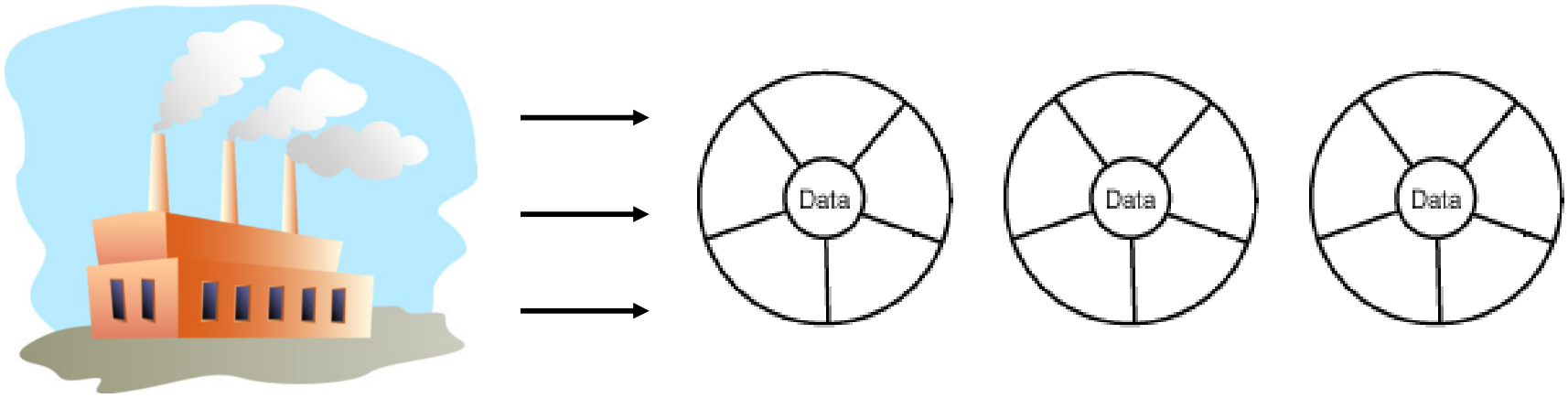
    - Styles are represented as integer constants in the `Font` class:

```
public static final int PLAIN
public static final int BOLD
public static final int ITALIC
```

  - Or create a new font, using an existing font as a *prototype*:
    - `public Font deriveFont(float size)`
    - `public Font deriveFont(int style, float size)`

Pattern: Factory

A method or object that creates other objects



Factory pattern

- **factory**: A method or object whose primary purpose is to manage the creation of other objects (usually of a different type).

- *Problem*: Object creation is cumbersome or heavily coupled for a given client. Client needs to create but doesn't want the details.

- **Factory Method** pattern
  - A helper method that creates and returns the object(s).
  - Can return subclass objects if so desired (hidden from client).

- **Abstract Factory** pattern
  - A hierarchy of classes/objects, each of which is a factory for a type.
  - Allows hot-swappable factory to be used by a given client.

  (Factory is a creational pattern.)

DateFormat as a factory

- `DateFormat` class knows how to format dates/times as text
  - Options: Just date? Just time? Date+time? Where in the world?
  - Instead of passing all options to constructor, use factories.
  - The subtype created doesn't need to be specified.

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(
                     DateFormat.FULL, Locale.FRANCE);
Date today = new Date();
System.out.println(df1.format(today));  // "Apr 20, 2011"
System.out.println(df2.format(today));  // "10:48:00 AM"
System.out.println(df3.format(today));
                                  // "mecredi 20 avril 2011"
```

Border factory

- Java graphical components like DrawingPanel can have borders:

```
public void setBorder(Border border)
```

- But `Border` is an interface; cannot construct a `new Border`.
  - There are many different kinds of borders (classes).

- Instead, use the provided `BorderFactory` class to create them:

```
public static Border createBevelBorder(...)
public static Border createEtchedBorder(...)
public static Border createLineBorder(...)
public static Border createMatteBorder(...)
public static Border createTitledBorder(...)
```

  - Avoids a constructor that takes too many "option / flag" arguments.

Scenario: Drawing images

- Suppose we want to display images on a graphical window.
- The `Graphics` object has a `drawImage` method:
  - `public void drawImage(Image img, int x, int y, panel)`
  - `public void drawImage(Image img, int x, int y, int w, int h, panel)`

- Images are hard drive files in a given format:
  - GIF, JPEG, PNG, BMP, TIFF, ...

- So how do we get an `Image` object to draw?
- Can't simply say `new Image` :
  - `Image img = new Image("bobafett.gif");  // error`

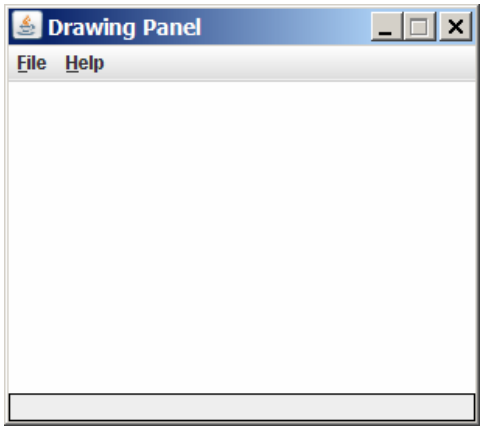
Toolkits

- `Toolkit` is a class for GUI system info and resource loading.
- Java handles loading of images through `Toolkit`s:
  - `public Image getImage(String filename)`
  - `public Image getImage(URL url)`
- Can't simply say `new Toolkit`:
  - `Toolkit tk = new Toolkit();  // error`
- Have to call a static method to get a toolkit (Why? What is this?):
  - `public static Toolkit getDefaultToolkit()`
  - `Toolkit tk = Toolkit.getDefaultToolkit();  // ok`

Buggy image client

- The following well-intentioned client does not show the images:

```
public static void main(String[] args) {
    Toolkit tk = Toolkit.getDefaultToolkit();
    Image img1 = tk.getImage("calvin.gif");
    Image img2 = tk.getImage("cuteicecream.jpg");
    Image img3 = tk.getImage("tinman.png");

    DrawingPanel panel = new DrawingPanel(600, 500);
    Graphics g = panel.getGraphics();
    g.drawImage(img1, 0, 0, panel);
    g.drawImage(img2, 200, 50, panel);
    g.drawImage(img3, 400, 200, panel);
}
```



Media trackers

- When you tell a toolkit to load an image, it *doesn't actually do so.*
  - It simply buffers a request to eventually load the image.
  - If you try to draw the image too quickly, it won't be loaded yet.

- Java uses *media tracker* objects to wait for an image to load:
  - `public MediaTracker(`**`panel`**`)`
  - `public void addImage(Image img, int id)`
  - `public void removeImage(Image img)`
  - `public void removeImage(Image img, int id)`
  - `public void waitForAll() **`
  - `public void waitForAll(long ms) **`
  - `public void waitForID(int id) **`
  - `public void waitForID(int id, long ms) **`

```
** throws InterruptedException
```

Media tracker example

```
public static void main(String[] args) {
    Toolkit tk = Toolkit.getDefaultToolkit();
    Image img1 = tk.getImage("calvin.gif");
    Image img2 = tk.getImage("cuteicecream.jpg");
    Image img3 = tk.getImage("tinman.png");

    MediaTracker mt = new MediaTracker(panel);
    mt.addImage(img1, 1);
    mt.addImage(img2, 2);
    mt.addImage(img3, 3);
    try {
        mt.waitForAll();
    } catch (InterruptedException e) {}

    DrawingPanel panel = new DrawingPanel(600, 500);
    Graphics g = panel.getGraphics();
    g.drawImage(img1, 0, 0, panel);
    g.drawImage(img2, 200, 50, panel);
    g.drawImage(img3, 400, 200, panel);
}
```

Image loading factory

- The preceding code is too cumbersome to write every time we want to load an image.
  - Let's make a factory method to load images more easily:

```
public static Image loadImage(
        String filename, DrawingPanel panel) {
    Toolkit tk = Toolkit.getDefaultToolkit();
    Image img = tk.getImage(filename);

    MediaTracker mt = new MediaTracker(panel);
    mt.addImage(img, 0);
    try {
        mt.waitForAll();
    } catch (InterruptedException e) {}

    return img;
}
```

A factory class

- Factory methods are often put into their own class for reusability:

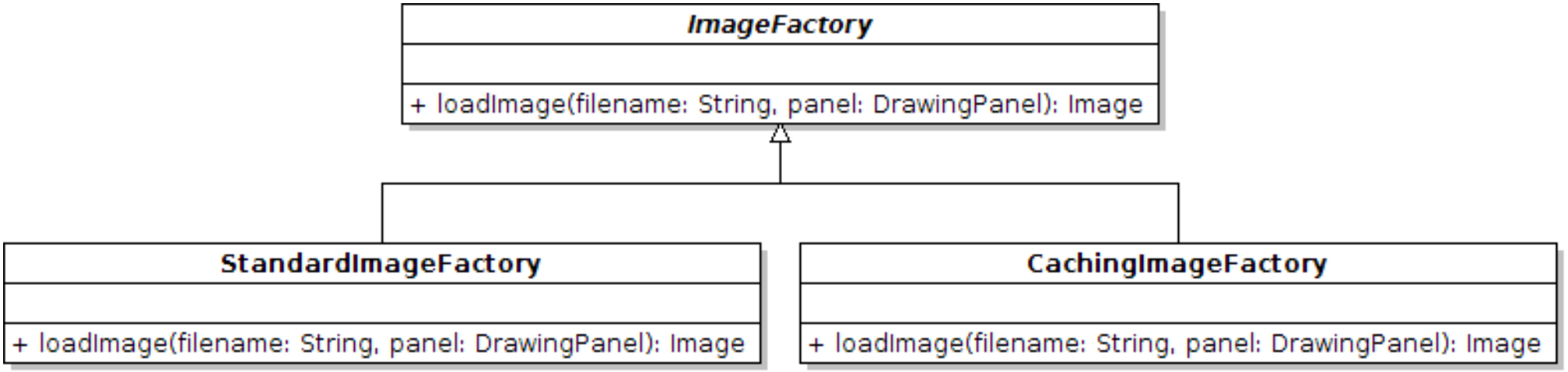
```
public class ImageFactory {
    public static Image loadImage(
            String filename, DrawingPanel panel) {
        Toolkit tk = Toolkit.getDefaultToolkit();
        Image img = tk.getImage(filename);
        MediaTracker mt = new MediaTracker(panel);
        mt.addImage(img, 0);
        try {
            mt.waitForAll();
        } catch (InterruptedException e) {}
        return img;
    }

    public static Image loadImage(
            File file, DrawingPanel panel) {
        return loadImage(file.toString(), panel);
    }
}
```

Exercise: Caching factory

- Loading large images from the disk repeatedly can be slow.
- Write a modified version of the image factory that caches images so that it does not ever have to re-load the same image file twice.
- Things to think about:
  - Can you think of any cases where caching would not be desired?
  - How could we provide the client the ability to turn caching on/off?
  - Can we make this decision transparent to most client code, settable in a single place?
- Another possibility: A factory that loads images from URLs.

Abstract Factory pattern

- **abstract factory**: A superclass factory that can be extended to provide different sub-factories, each with different features.
  - Often implemented with an *abstract* superclass.
  - Idea: Client is given an instance of `ImageFactory`, which will actually be a `Standard` or `Caching ImageFactory`.
  - Client just uses it and doesn't worry about which one it was given.



Abstract factory code

```
public abstract class ImageFactory {
    public abstract Image loadImage(
            String filename, DrawingPanel panel);
}

public class StandardImageFactory extends ImageFactory {
    public Image loadImage(String filename,
            DrawingPanel panel) { ... }
}

public class CachingImageFactory extends ImageFactory {
    public Image loadImage(String filename,
            DrawingPanel panel) { ... }
}

public class WebImageFactory extends ImageFactory {
    public Image loadImage(String filename,
            DrawingPanel panel) { ... }
}
```