

Revision

- We want to make a figure drawing program.
- Problem: we want our program to be "extensible", i.e., it should require minimal changes to add support to new figure types.
- We have made an abstract class called Figure to define what any figure should be able to do. It also manages depth and fill character of Figure objects.
- Figures are Comparable (comparing depths). A figure object can read itself (read(StdIn)), knows its fill character (getFill()), and knows whether a given point is in it (contains(int,int)).
- We have made one non-abstract (concrete) class called CircleFigure that extends the Figure class.

0911B L26-1

First try: Getting Objects

Now we can write a program to do figure drawing, supporting only circles. Our program does the following:

Get an array of figures from the user
Draw the array of figures

To get an array of figures is easy:

Ask for the number of figures
Create an array of that many Figure references
For each figure reference
 Create a circle as the figure in the reference
 Read the figure

0911B L26-2

Implementation: Getting Objects

```
import chapman.io.*;
public class DrawCircles {
    public static void main(String[] args) {
        Figure[] figure_list = getFigList();
        drawFigures(figure_list);
    }
    static Figure[] getFigList() {
        System.out.print("Number of figures? ");
        int num_figures = in.readInt();
        Figure[] list = new Figure[num_figures];
        for (int i = 0; i < num_figures; ++i) {
            list[i] = new CircleFigure();
            list[i].read(in);
        }
        return list;
    }
    ...
    static StdIn in = new StdIn();
}
```

0911B L26-3

First try: Draw the Array of Figures

Drawing the screen is also easy:

Sort the array of figures in depths
For each y from -11 to 11
 For each x from -39 to 39
 Draw the point x, y.
 End the line

Drawing a point (x,y):

For each figure f in the array
 If f contains (x,y)
 Draw the character for f
 Return
// No figure contains the point
Draw a space

0911B L26-4

Implementation: Draw the Array of Figures

```
public class DrawCircles {
    ...
    static void drawFigures(Figure[] figure_list) {
        java.util.Arrays.sort(figure_list);
        for (int y = -11; y <= 11; y++) {
            for (int x = -39; x <= 39; x++)
                drawPoint(figure_list, x, y);
            System.out.println();
        }
    }
    ...
}
```

Note that java.util.Arrays.sort(figure_list) makes sense because the Figure's are Comparable (i.e., implements Comparable interface).

0911B L26-5

Implementation: Draw the Array of Figures (cont'd)

```
public class DrawCircles {
    ...
    static void drawPoint(Figure[] figure_list, int x, int y) {
        for (int i = 0; i < figure_list.length; ++i) {
            Figure fig = (Figure) figure_list[i];
            if (fig.contains(x, y)) {
                System.out.print(fig.getFill());
                return;
            }
        }
        System.out.print(' ');
    }
}
```

Note that fig.contains(x, y) calls the correct "contains", depending on what the object fig really refers to.

This is usually called "late binding": the name "contains" is binded to a particular method only at run-time (hence "late").

0911B L26-6

Evaluation of the first try

What we need to do if we need to support, e.g., rectangles?

- We need to create a new class `RectangleFigure` extending `Figure`. It defines the data fields of a rectangle, how to read a `Rectangle`, and how to determine whether a point is within a rectangle.
- **Figure drawing need no change at all.**
 - For figure drawing, we never call methods of `CircleFigure`.
 - Instead, we have a `Figure` reference, call the `contains()` method using that `Figure` reference, and the Java interpreter chooses to call `CircleFigure.contains()`.
 - If we have a `Rectangle` object, Java will choose `RectangleFigure.contains()` instead.

Clean?

0911B L26-7

Defining Rectangle

```
public class RectangleFigure extends Figure {
    public boolean contains(int loc_x, int loc_y) {
        if (loc_x < x || loc_x > x + width)
            return false;
        if (loc_y < y || loc_y > y + height)
            return false;
        return true;
    }
    public void read(chapman.io.StdIn in) {
        super.read(in);
        System.out.print(" width? ");
        width = in.readInt();
        System.out.print(" height? ");
        height = in.readInt();
    }
    private int width, height;
}
```

0911B L26-8

Figure reading is not done

- But there is quite something that we need to do to support reading rectangles.
- The current code to read `Circles` always read a circle, and never read anything else. To support reading `Rectangles`, we must make it choose to make `Rectangles` instead.
- Looking at the specification, we want to read a string before choosing what object to create.
- That gives rise to a series of `if/else` statements to choose a class to create. This will be the only place we want to change if we extend the program.

0911B L26-9

Second try: drawing circles

Here, instead of saying "new `CircleFigure()`", we call `makeFigure()`, to be defined.

```
import chapman.io.*;
public class DrawFigures {
    ... // Everything else is the same as DrawCircles
    static Figure[] getFigList() {
        System.out.print("Number of figures? ");
        int num_figures = in.readInt();
        Figure[] list = new Figure[num_figures];
        for (int i = 0; i < num_figures; ++i) {
            list[i] = getFigure();
            list[i].read(in);
        }
        return list;
    }
    ...
}
```

0911B L26-10

Getting a figure

To make a figure, we use a loop asking for a figure type. Then we call a method to make the figure. If it can make one, return it. Otherwise, we ask again.

```
public class DrawFigures {
    ...
    static Figure getFigure() {
        for (;;) {
            System.out.print("Figure type? ");
            String type = in.readString();
            Figure fig = makeFigure(type);
            if (fig != null)
                return fig;
            System.out.println("Sorry: " + type + " not supported.");
        }
    }
    ...
}
```

0911B L26-11

Making a figure

Depending on the type, we need to make a figure. We do it like this:

```
public class DrawFigures {
    ...
    static Figure makeFigure(String type) {
        if (type.equalsIgnoreCase("circle"))
            return new CircleFigure();
        if (type.equalsIgnoreCase("rectangle"))
            return new RectangleFigure();
        return null;
    }
    ...
}
```

So we just check for each possible type before making a figure. Notice that this method requires us to know what are the possible figures.

0911B L26-12

An even more ambitious target

- Now we have a program that can be modified easily to support additional objects. But we still have to modify it.
- In fact, the only place where modifications is really needed is when we create an object, i.e., in `makeFigure()`.
- Is it possible that we can instead sub-class our main program in order to support new objects? (At times, modifying the program is not possible, e.g. the file is in the server.)
- That is, is it possible to write `DrawFigure` in such a way that we can subclass it to, say, `DrawFigureV2` to support some more Figures?

0911B L26-13

Static methods cannot be overridden

There is a big problem that we must resolve for this to work.

- We want this: "Make a subclass of `DrawingFigure` that has all the methods of `DrawingFigure`, except that `makeFigure()` should allow, e.g., "parabola", which makes a `ParabolaFigure`.
- This requires late binding: when `getFigure()` calls `makeFigure()`, Java should choose to run `DrawingFigure.makeFigure()` or `DrawingFigureV2.makeFigure()` depending on whether we are running `DrawingFigure` or `DrawingFigureV2`.
- But late binding happens only when we have an instance of `DrawingFigure` or `DrawingFigureV2`, and the `makeFigure()` is an instance method!

0911B L26-14

Using an object to represent the currently running program

Now we need an object of class `DrawingFigure` and `DrawingFigureV2`, representing the current program, for these to work.

The simplest solution: to make everything in `DrawFigure`, except `main(String[])`, non-static.

Main will just create an instance and start running it.

```
public class DrawFigures {
    public static void main(String[] args) {
        DrawFigures app = new DrawFigures();
        app.run();
    }
    void run() {
        Figure[] figure_list = getFigList();
        java.util.Arrays.sort(figure_list);
        drawFigures(figure_list);
    }
    ... // Just remove static from everything else
}
```

0911B L26-15

Sub-classing DrawFigure

Now comes the interesting thing: to sub-class it.

Let's make it support parabola. See how easy it is now.

```
public class DrawFiguresV2 extends DrawFigures {
    public static void main(String[] args) {
        DrawFiguresV2 app = new DrawFiguresV2();
        app.run();
    }
    Figure makeFigure(String type) {
        if (type.equalsIgnoreCase("parabola"))
            return new ParabolaFigure();
        return super.makeFigure(type);
    }
}
```

0911B L26-16

Parabola class

Of course we still need to make `Parabola`.

```
// Parabola
class ParabolaFigure extends Figure {
    public boolean contains(int loc_x, int loc_y) {
        int dx = loc_x - x;
        int dy = loc_y - y;
        return dx * dx <= a * dy;
    }
    public void read(chapman.io.StdIn in) {
        super.read(in);
        System.out.print(" focal length? ");
        a = in.readInt();
    }
    private int a;
}
```

0911B L26-17

Summary

- In Java, **late binding** happens when we call an instance method through an object reference.
- Java determine the method to call at run-time, depending on the type of the object, not the type of the object reference.
- This is very useful when making extendable programs. We can call a method of a class without knowing what type of object it really is, so the code is usable even after we add more classes.
- Sometimes, it is beneficial to treat the whole program as an object just because we want to be able to subclass it with late binding.

0911B L26-18