COP 3330 – Object Oriented Programming in Java
Abstract Classes

***** NOTE: THESE NOTES WILL LIKELY BE UPDATED IN THE FUTURE *****

We've now learned about classes, inheritance and interfaces. A class contains a full specification for an object, including what it's made out of, how to build it, and definitions of methods that operate on the object. An interface seems to almost be at the other end of the spectrum:

It doesn't specify how any of the data is stored.
It doesn't specify how to build any portion of the object.
It doesn't define how any methods in the class work.

Rather, it only indicates that any class implementing the interface must include a method with a particular signature. An interface makes minimal specifications for a class.

One might wonder if there's a middle ground. would it be possible to:

1. Specify how SOME of the data is stored.
2. Can have constructors to partially build the object.
3. Can define some methods concretely leaving others required but undefined.

In the crude PacMan example previously shown, there was a Piece class and PacMan could either eat Fruit or Pieces. That might have seemed a bit weird as the actual intention of the design was for Piece to just store information common to all the types of things that might appear in the Pac Man playing arena. Since we hadn't learned Abstract Classes at that point in time, I made the Point class a real class and generated actual Point objects.

But, in reality, it might have been better if I created maybe Fruit and Dots and both of them inherited from Piece, but where Piece itself was not an actual real object. This is precisely what an abstract class is. It allows for a middle ground between an interface (little specification) and a class (complete specification). It allows for us to put some common elements in one place without having to fully develop those common elements into a complete description of a class and how to build/use objects of that class.

Thus, the sole purpose of creating an abstract class is to store common elements in a place that can be inherited from by multiple classes (if you were only ever going to build one class it wouldn't make sense to aggregate the common elements among different classes in one place!) without being forced to fully define the class.

For the rest of this lecture, we'll look at an adjusted version of the Pac Man example. This one will make Piece abstract, add a class Dots that inherits from Piece, adds a class Ghost that inherits from Piece, and remakes the Game object to be ever so slightly closer to the real game. (We'll get rid of the crazy movement in the old example and also provide a better toString() method for the board so that it looks a tiny bit like a Pac Man playing area.

Pac Man Version 2 Example

Quite a few changes have been made (so far), to the Pac Man game, but it's not near where it should be in terms of polish. (It has lots of bugs.) But it's in a currently semi-working state utilizing an abstract class. Unfortunately, the class is "barely" abstract. It has mostly complete instance variables specified as well as many defined methods. In fact, the only thing that makes it abstract is the following abstract method added:

```
public abstract char getCode();
```

In our effort to print out the PacMan board in a more realistic way, for each class that inherits from the Point class, we'll display its representation on the board in a single character.

In addition, once the game was being modified, in order to keep pieces on a well-defined playing board, two methods were added to check if a piece was in bounds or not and also to see if moving the piece by its specified direction would move it out of bounds. Here's the current version of this class:

```
abstract class Piece {

    protected Position loc;
    protected Position dir;

    public Piece(int x, int y, int dx, int dy) {
        loc = new Position(x, y);
        dir = new Position(dx, dy);
    }

    public Piece(Position cur, Position change) {
        loc = cur;
        dir = change;
    }

    public boolean wouldStayInBounds(int minX, int maxX, int minY, int maxY) {
        Position tmp = loc.wouldMoveBy(dir);
        return tmp.inbounds(minX, maxX, minY, maxY);
    }

    public boolean inbounds(int minX, int maxX, int minY, int maxY) {
        return loc.inbounds(minX, maxX, minY, maxY);
    }

    public String toString() {
        return loc.toString();
    }

    public void move() {
        loc.moveBy(dir);
    }

    public void setDir(Position mydir) {
        dir = mydir;
    }
```

```
    public void negateDir() {
        dir.negate();
    }

    public void incX() {
        setDir(new Position(0,1));
    }

    public void decX() {
        setDir(new Position(0,-1));
    }

    public void decY() {
        setDir(new Position(-1,0));
    }

    public void incY() {
        setDir(new Position(1,0));
    }

    public Position getLoc() {
        return loc;
    }

    /*** This is what makes this class abstract. All classes
         that inherit from this one must have a single character code
         that is returned by a getCode() method.
    ***/
    public abstract char getCode();

}
```

In addition to inbounds checking, several methods were added to make moving in the cardinal directions easier. Finally, the abstract method is listed at the end.

Now, let's take a quick look at the new Dot class (sort of what the Piece class was before):

```
public class Dot extends Piece {

    public Dot(int x, int y) {
        super(x, y, 0, 0);
    }

    public String toString() {
        return "dot at "+loc;
    }

    public char getCode() {
        return '.';
    }
}
```

In fact, the toString() method isn't really needed, so this class is very bare bones. But it illustrates the idea that there's no such thing as a Point but that a Dot isn't much more than all the functionality in Point. We just have the character code of '.' for objects from this class.

Now, let's take a quick look at the Ghost class:

```
public class Ghost extends Piece {

    public Ghost(Position p, Position dir) {
        super(p, dir);
    }

    public String toString() {
        return "Ghost! at "+loc;
    }

    public char getCode() {
        return 'G';
    }
}
```

It's basically identical to the Dot class except that the constructor takes in a movement direction and its code is different.

To change our functionality, the biggest changes were made in the PacMan class and the PacmanGame class.

Here is the new eat method in Pacman:

```
public boolean eat(Piece p) {

    if (p instanceof Ghost)
        return false;

    if (p instanceof Dot)
        numCaptures++;

    if (p instanceof Fruit)
        score += ((Fruit)p).getPoints();

    return true;
}
```

The big difference here is that the eat method returns something: true if we are still alive after the move, and false otherwise. To check if we're alive, we check to see if the piece we're trying to eat is a Ghost. If so, this method returns false and ends. Alternatively, if we ate a Dot, it adds to our captures, and if we ate a Fruit, it adds to our score.

Here is the implementation of the abstract method:

```
public char getCode() {
    return 'C';
}
```

Instead of showing the whole PacmanGame class in code in these notes, let's just illustrate how getCode() gets used.

First, let's take a look at the instance variables in the PacmanGame class:

```
private int numX;
private int numY;
private Pacman player;
private int numFoodObj;
private Piece[] food;
private int timeStep;
private int dotsLeft;
```

The first two instance variables define the size of the playing area, which extends from -numX to numX in x, and -numY to +numY in y. These were added to the previous version. Also, the instance variable dotsLeft was added to detect when the player wins the game.

Now, let's take a look at the toString method, which uses getCode():

```
public String toString() {

    char[][] grid = new char[2*numX+1][2*numY+1];
    for (int i=0; i<grid.length; i++)
        Arrays.fill(grid[i], '_');

    for (Piece p: food)
        if (p.inbounds(-numX, numX, -numY, numY))
            grid[p.loc.getX()+numX][p.loc.getY()+numY] = p.getCode();

    if (player.inbounds(-numX, numX, -numY, numY))
        if (player.inbounds(-numX, numX, -numY, numY))
            grid[player.loc.getX()+numX][player.loc.getY()+numY] =
                                                    player.getCode();

    String res = "";
    for (int i=0; i<grid.length; i++)
        res = res + new String(grid[i])+"\n";
    res = res + player + "\n";
    res = res + "Dots Left = "+ dotsLeft + "\n";
    return res;
}
```

The grid is a 2D character array indexed from 0 to 2*numX in x, and 0 to 2*numY in y. Since we store our coordinates from -numX to numX and -numY to numY, when displaying pieces, we have to offset our indexes into grid by adding numX and numY, respectively to the first and second indexes into the grid array. Once we know where we are, we place the piece's code there. If two pieces occupy the same square, whichever piece loops through last will show up on the board. We don't want the program to crash, so inbounds checks have been added before we place a piece on the board. Then, we build our String from the character grid we created, finally adding information about the player and the number of dots left. We'll scan through the rest of the code for this class in class.