<u>COP 3330 – Object Oriented Programming in Java</u>
<u>Inheritance (IS-A relationship) – Extended Example</u>

PacMan Example

This lecture will feature all of the tools from the previous lecture, but in service of building a more "practical" example which also illustrates the advantages of utilizing inheritance as it relates to code reuse.

The example we will use is an adaptation of ideas from the classic video game Pac Man. Since we haven't looked at any graphics yet, we'll just store relevant backend information. For the purposes of our "game", the game grid is Cartesian cells labeled with integer x and y coordinates. Each "piece" in the game can move in various ways. Our pieces for this particular example will be just the Pac Man player itself and Fruit. As you might imagine, both of these are classes in the game.

A second observation is that while Pac Man and Fruits operate differently, they share some behaviors in common, particularly with having a position on the board and movement. Utilizing this observation, we'll go ahead and design the four following classes to store the objects for our game:

1. Position (Just keeps information about a Cartesian Coordinate)
2. Piece (All pieces have both a current position and a current direction of movement)
3. Fruit
4. Pacman

Let's look at the IS-A and HAS-A relationships here:

Piece HAS-A Position. (As we'll soon see, it has two of them.)
Fruit IS-A Piece
Pacman IS-A Piece

Finally, we'll create a PacmanGame object, which manages a PacmanGame. This object will have both a single Pacman object and several Piece objects. The objects that Pacman can eat are of the following types:

1. Piece
2. Fruit

While Pieces won't be worth any points, Fruit will be. What inheritance allows us to do then is to just have an array of Piece instead of two separate arrays of Piece and Fruit. This makes the code much cleaner.

Also, since we haven't learned the built in ArrayList class, I am not using that class, but most people would design this example with an ArrayList of Piece instead of an array of Piece. Also, the game itself is terribly unrefined. A great student exercise would simply be to improve it!!!

This is all the set up we need, let's dig in!

Position class

We will maintain the x and y coordinates in this class, but also allow some functionality related to movement. In particular, we'll provide two separate methods that directly mutate a Position object (essentially move it to somewhere else.) The class is fairly small. Here is all of the code:

```
public class Position {

    private int x;
    private int y;

    // Creates a Position at (myx, myy).
    public Position(int myx, int myy) {
        x = myx;
        y = myy;
    }

    // Returns true iff this and other are the same Position.
    public boolean equals(Position other) {
        return x == other.x && y == other.y;
    }

    // Moves this Position by an offset of other.
    public void moveBy(Position other) {
        x += other.x;
        y += other.y;
    }

    // Changes this Position by (dx, dy).
    public void moveBy(int dx, int dy) {
        x += dx;
        y += dy;
    }

    // String representation of the object.
    public String toString() {
        return "("+x+", "+y+")";
    }
}
```

The constructor, equals and toString methods should be self-explanatory. For our game, we'll have directions of movement which will be stored in another Position object. Thus, it will be useful if we could "shift" one Position by another Position. For example if object A is at (4, 9) and the direction we want to move is stored in the object B which is (2, -1), then if we move object A by the direction indicated by object B, object A will change to (6, 8), since 4 +2 = 6 and 9 + (-1) = 8. Also, if someone doesn't have a Position object but individually knows much much they want to change x and y by, that alternate functionality is provided. This is fairly common in Java, where the Java API provides multiple methods that roughly do the same thing, in an effort to make the object easy and flexible to use.

Note that the x and y components are private. I do NOT let any other class directly modify these. To do so, all other classes must call one of the moveBy methods. This is a design decision that only allows the user to change Position objects in specified ways.

Piece class
A piece will have BOTH a current position, as well as a direction of movement. A piece will have a move method, which will essentially move it for one time frame. Here's the full class:

```
public class Piece {

    protected Position loc;
    protected Position dir;

    // Creates a piece at location start, with a movement direction of delta.
    public Piece(int x, int y, int dx, int dy) {
        loc = new Position(x, y);
        dir = new Position(dx, dy);
    }

    // Executes a single move of this Piece.
    public void move() {
        loc.moveBy(dir);
    }

    // Returns where this Piece is.
    public Position getLoc() {
        return loc;
    }

    // String representation of a generic piece.
    public String toString() {
        return "Piece at "+loc.toString();
    }
}
```

Take a look at the move method. We are only allowing pieces currently to move in a very fixed way. Also, currently, we aren't allowing for a simple way for the direction of movement to change. In a typical game, we would want to add functionality that allows someone outside of this class to easily change the direction of movement. (This will be a practice problem!)

Fruit class
Now that we've written these two classes, we see that the Piece class houses most of the functionality of what we might imagine for either a Pacman or Fruit class. Let's look at the more simple of these two classes first, the Fruit class. This is our use of inheritance in this program. For this implementation, each Fruit will have a name, a color and number of points, in addition to the attributes of a Position object. Here is the full class (note the use of the key word extends, indicating inheritance):

```
public class Fruit extends Piece {

    private String name;
    private String color;
    private int points;

    // Build my fruit!!!
    public Fruit(int x, int y, int dx, int dy, String myName, String myColor,
                 int myPoints) {

        super(x, y, dx, dy);
        name = myName;
        color = myColor;
        points = myPoints;
      }

    // String representation of this Fruit.
    public String toString() {
        return name+" color "+color+" at "+loc+" worth "+points;
    }

    // Returns the number of points this Fruit is worth.
    public int getPoints() {
        return points;
    }
}
```

Our constructor takes in all seven pieces of information that generate a Fruit object, uses the first four to generate the super call to build the Position portion of the object and then assigns the rest of the instance variables. We'll want other classes to be able to find out how many points a Fruit object is, so we create a method to allow others to access that data.

Pacman Class
Our Pacman object will also have a name. It will also have a score, which is the sum of points of the fruit it eats, as well as keeping track of the total number of captures of either Fruit or Piece. Also, we'll limit PacMan's movement to UP, DOWN, LEFT and RIGHT by 1 unit.

We need to simply provide functionality in this class that allows Pacman to move and eat other Pieces.

The class is on the next page:

```java
public class Pacman extends Piece {

    private String name;
    private int score;
    private int numCaptures;

    // Creates a new Pacman object at (x,y) with name myName.
    public Pacman(int x, int y, String myName) {
        super(x, y, 0, 0);
        name = myName;
        score = 0;
        numCaptures = 0;
    }

    // Returns a String representation of this Pacman.
    public String toString() {
        return "Pacman "+name+" at "+loc+" with score "+score+" and "
                        +numCaptures+" captures";
    }

    // This Pacman eats Piece p.
    public void eat(Piece p) {

        // One more capture.
        numCaptures++;

        // We get points if it's a fruit!
        if (p instanceof Fruit)
            score += ((Fruit)p).getPoints();
    }

    public void incDX() {
        dir.moveBy(1,0);
    }

    public void decDX() {
        dir.moveBy(-1,0);
    }

    public void incDY() {
        dir.moveBy(0,1);
    }

    public void decDY() {
        dir.moveBy(0,-1);
    }
}
```

We initialize Pacman to not be moving with the super call in the constructor:

```java
super(x, y, 0, 0);
```

Then, we limit Pacman's movement by only providing the methods incDX(), decDX(), incDY() and decDY() for the user outside of this class.

Now, let's take a careful look at the eat method, which is set up to naturally handle both Piece and Fruit objects:

```
public void eat(Piece p) {

    // One more capture.
    numCaptures++;

    // We get points if it's a fruit!
    if (p instanceof Fruit)
        score += ((Fruit)p).getPoints();
}
```

It's safe for us to always increment the number of captures, so we do this right away. But now, if our piece happens to be a fruit, we need to update our score. We must use the instanceof operator to check if p is actually referencing a Fruit obect, which is done in the if statement.

One question is why must we cast p to type Fruit?

The answer is that without the cast, the code won't compile. At compilation time, all we know is that p is a Piece. But, the getPoints() method is NOT written in the Piece class, since this isn't functionality needed by the Piece class. Thus, p.getPoints() would not compile, because the compiler will look in the class of the reference, in this case, Piece, to find a matching method.

BUT, if we cast p to a Fruit, then we are treating this expression (the reference) for the purposes of this statement, as a Fruit reference. If we treat p as a Fruit reference, then the compiler will look in the Fruit class for the method getPoints() and find it. This allows us to update our score only when needed.

PacmanGame class
Now, let's put it all together. This game class will allow the user to run a PacmanGame. Our Pacman object will consist of the four following parts:

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

We keep a separate variable storing the number of Food objects so that we have easy access to the size of the food array.

In addition, we have several constants and one class variable:

```
final public static String[] FRUITNAMES = {"strawberry", "banana", "orange",
                                           "kiwi"};
final public static String[] COLORS = {"red", "yellow", "orange", "green"};
final public static int[] POINTS = {100, 50, 150, 300};

public static Random rndObj = new Random();
```

Now, let's take a look at the constructor, which takes in the name of the player and the number of Piece objects to create for food:

```
public PacmanGame(String playerName, int numFood) {

    // I start at the origin. Yes, the universe centers around PacMan!
    player = new Pacman(0, 0, playerName);

    // Allocate space for the food.
    numFoodObj = numFood;
    food = new Piece[numFoodObj];

    // Keeps track of how many moves were made.
    timeStep = 0;

    // Create random food objects.
    for (int i=0; i<numFoodObj; i++) {

        int x = 1+rndObj.nextInt(10);
        int y = 1+rndObj.nextInt(10);
        int dx = -2 + rndObj.nextInt(5);
        int dy = -2 + rndObj.nextInt(5);

        // One out of every 3 things (roughly) will be fruit.
        if (rndObj.nextInt(3) == 0) {
            int fruitIdx = rndObj.nextInt(FRUITNAMES.length);
             food[i] = new Fruit(x, y, dx, dy, FRUITNAMES[fruitIdx],
                                 COLORS[fruitIdx], POINTS[fruitIdx]);
        }

        // Boring pieces!
        else
            food[i] = new Piece(x, y, dx, dy);
    }
}
```

As this example shows, not all constructors are trivial. It's certainly much harder to build some objects than other objects. In this case, it's easy for us to build Pacman. We just start him at (0, 0). But, building the food is harder. We first allocate the array of references to Pieces. But then, we have to create each one individually in the loop. Our design decision here is to make roughly one out of every three pieces a Fruit. Each object will start with x and y coordinates in between 1 and 10 inclusive, and both their dx and dy will range from -2 to 2. If we create a fruit, we randomly choose one of the four fruits (strawberry, banana, orange or kiwi) to generate via rndObj. Notice the use of the constant arrays here in making this task easier and the code relatively succinct. Index i in each of the arrays stores information about fruit i.

The other key method in this class is the move method, which coordinates one full move (a single timestep) for the whole game. This will involve moving Pacman based on the user input, moving all of the Piece objects, seeing which Piece object(s) Pacman ate, and then updating both Pacman's score and the Piece array accordingly.

Here is the beginning of the function which takes care of moving Pacman:

```
public void move(char myMove) {

    // Update movement vector of player.
    if (myMove == 'U') player.incDY();
    if (myMove == 'D') player.decDY();
    if (myMove == 'L') player.decDX();
    if (myMove == 'R') player.incDX();
    player.move()

    // Rest of the code here.
}
```

The method takes in a character of movement and based on that character, as long as it's one of 'U', 'D', 'L' or 'R', the appropriate method from the player class is called. Then we call the move method for the player (which really resides in the Piece class.)

Next, we move the food. Notice how easy inheritance makes this!

```
for (int i=0; i<numFoodObj; i++)
    food[i].move();
```

At this point, since we are managing our own array, we first need to figure out how many pieces we are going to eat on this step. We have a private method called numEaten() which figures this out as follows:

```
private int numEaten() {
    int numEaten = 0;
    for (int i=0; i<numFoodObj; i++)
        if (player.getLoc().equals(food[i].getLoc()))
            numEaten++;
    return numEaten;
}
```

Basically, we just loop through the food, and for each food object, check to see if it's location is equal to the location of the player. If so, that's one more piece Pacman gets to eat!

This could also be written as:

```
private int numEaten() {
    int numEaten = 0;
    for (Piece p: food)
        if (player.getLoc().equals(p.getLoc()))
            numEaten++;
    return numEaten;
}
```

This is the way to loop through an array without indexing it directly. It's known as an iterator loop. It's guaranteed to go through each item in the array food.

Next, in the move method we figure out how many pieces we're going to eat and store the new size of our Piece array:

```
int lose = numEaten();
int newSize = numFoodObj - lose;
```

Finally, we'll copy over the uneaten pieces to a new temporary array. This will require two indexes: one into the original array (i) and one into our new array of surviving Piece objects (j):

```
Piece[] survived = new Piece[newSize];
for (int i=0,j=0; i<numFoodObj; i++) {

    // Eat me!
    if (player.getLoc().equals(food[i].getLoc())) {
        player.eat(food[i]);
    }

    // Copy into survived food.
    else {
        survived[j++] = food[i];
    }
}
```

For each Piece object, one of two things must be true:

1. You were eaten
2. You weren't eaten.

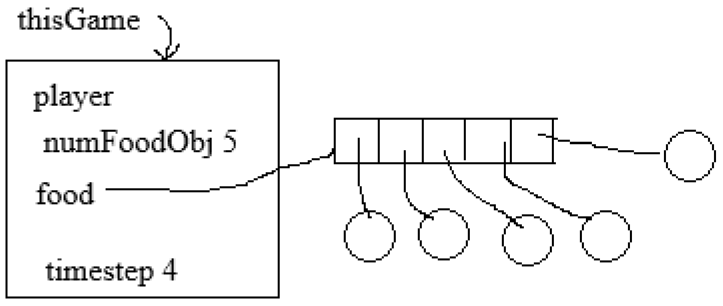
If the first is true, we must execute Pacman eating the Piece object (via the eat method in the Pacman class).

if the second is true, we must copy you into the new temporary array. We use the post increment in the index of survived to make the code more compact. The else could have been written as:

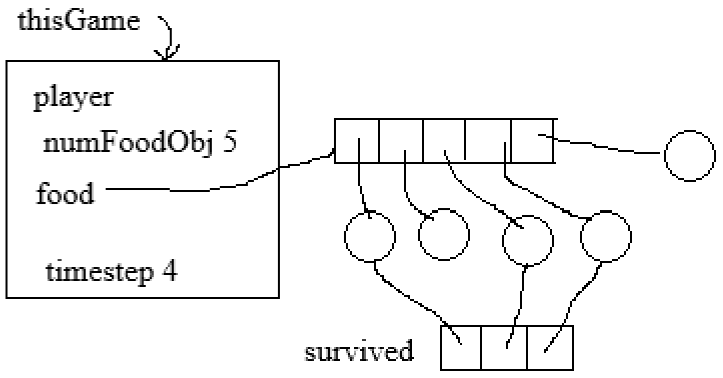
```
else {
    survived[j] = food[i];
    j++;
}
```

This is the most complicated part of the code. Let's look at a drawing of what's going on in detail here. Consider a situation where numFoodObj is 5 and there are 2 eaten objects, in indexes 1 and 4, respectively. It this drawing, the name of our PacmanGame object is thisGame.

Here is the before picture:



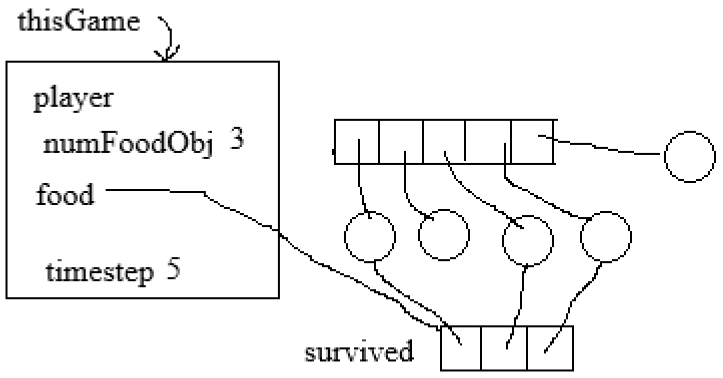
Now, let's look at what the picture looks like after the for loop completes:



Now, we just need to update our object with these lines of code:

```
numFoodObj = newSize;
food = survived;
timeStep++;
```

Here's the change in the picture after these three lines execute:



This is the end of the method. Soon after, the floating array of size 5 will be garbage collected and the objects that don't have active references pointing to them (the ones that used to be pointed to by index 1 and index 4 of the old array) will also be garbage collected.

This takes us to the main method in the class that runs the game:

```
public static void main(String[] args) {

     // Get player name.
     Scanner stdin = new Scanner(System.in);
     System.out.println("What is your name?");
     String name = stdin.next();

     // Create a game with 10 objects to eat.
     PacmanGame thisGame = new PacmanGame(name, 10);

     for (int i=0; i<20; i++) {

          // Print the game.
          System.out.println(thisGame);

          // Get move and execute it.
          System.out.println("What move?(U,D,L,R,S)");
          char myMove = stdin.next().charAt(0);
          thisGame.move(myMove);
     }
}
```

Notice how short and clean this is. By building up a good design in the objects we created, we greatly simplify the work done at the top level and the readability of the code. In this example, we play the game for 20 time steps. But, we could easily write our main to do some pretty different things! (We could wait until we had eaten some number of pieces, or until Pacman dies, etc.) This version doesn't have the facility for Pacman to die, but that could easily be built into the code.