1 Introduction and Motivation

In lecture today, we went over a few scenarios where basic Java classes weren’t ”enough” to properly express what we wanted out of an implementation of Chess. The classes I proposed, Knight, Rook and Board are at the back of this paper. Remember that these first iterations of our code had a lot of repetition! It didn’t follow DRY (Don’t Repeat Yourself) principles and generally would be a hassle to change (since changes would need to be made in each class). Furthermore, the Board class would need to keep track of all these different class and would need to know about these details. What if we wanted to introduce new types of chess pieces in the future? We’d need to alter every single class since they are all dependent on each other.

After this, I introduced the Piece class, which aimed to solve the repeating code issue with our previous implementation by having the type of the chess piece be moved into a property of the Piece class rather than as a separate class. This prevented us from duplicating code, and it cleaned up the Board class by allowing for a single collection of Piece objects, but increased complexity within our methods! This wasn’t suitable either.

2 Inheritance Basics

What if we could have a way of sharing properties among the different chess piece classes while being able to modify them to fit individual specifications? This would relate them to a common entity (the idea of being a chess piece) while decoupling them from relying on each other. Luckily, we can create a superclass that these new subclasses can inherit from. Take a look at the new classes in the second section of code. Here we can see that the new Rook and Knight classes now inherit from Piece through the extends keyword. This means that all of the current methods and instance variables of Piece are accessible from the other two classes, since they ARE Pieces! Remember the IS-A relationship between subclasses and superclasses. You can visualize instances of subclasses as having a whole instance of the superclass inside them, with those accessible variables and methods. So, even though Rook and Knight have no classes or methods right now, we can do the following

```java
Knight k = new Knight();

// We can access superclass variables and methods!
k.type = "Knight";
k.move()

// k.move() => should print "Moving."
```

Furthermore, because of the IS-A relationship and the fact we can think about Rooks and Knights AS Pieces, we can carry this over to the reference variable types!

```java
//Knight IS-A Piece
Piece p = new Knight();

//Not all Pieces are Knights though!
//WON’T WORK
Knight k = new Piece();
```
This is AWESOME. The idea here is that we can have a particular variable with a certain static type `(Piece p)` that can take on MANY DIFFERENT STATIC TYPES. This is polymorphism. We'll go over the power of this later.

### 3 Constructors and super

I stated above that you can think about instances of subclasses as having mini-instances of the superclass contained within them. This is because constructors in the subclass implicitly call the default constructor of the superclass (unless another superclass constructor is specified). Consider the following:

```java
class Piece {
    String type;
    public Piece() {
        this.type = "chess piece";
    }
}

class Knight {
    public Knight() {
        //Blank! No calls to the super constructor!
    }
}

//Below code would be in a main method
Knight k = new Knight();
System.out.println(k.type);
//We'd think it might return 'null' since we don't touch k.type in the Knight constructor,
//but it actually return 'chess piece'!
//The Piece superclass constructor is implicitly called.
```

There are some more rules about superclass methods, variables, and constructors, but those are covered in reading and will be examined in lab. The key things here are that constructors behave in interesting ways with inheritance, and that to access specific superclass behavior in a subclass instance, you can use the `super` keyword.

### 4 Dynamic Method Lookup

Let's take a look at the set of examples I walked through in class and what their output was. The first few should be pretty straightforward, but the last two might make you shake your head a bit. First, a few background things:

#### 4.1 Method signatures

Methods in Java are defined by more than just the name; they are defined uniquely by their method SIGNATURE. The SIGNATURE of a method consists of the name AND the parameter types! Remember, method parameters are the same as method arguments, just a different name. Thus, the following are the same methods:
4.2 Resolving method arguments by static type

Now that we have an understanding of how methods are defined, we should talk about how to decide what method to call for a class! Consider overloaded methods in the **Knight** class:

```java
public void capture(Knight k){ System.out.println("Capturing Knight!"); }
public void capture(Piece piece){ System.out.println("Capturing Piece!"));
```

We always look at the static type of the reference variable (the declared type of the variable) being passed in to a method call to resolve it to the correct method based on signature. It will be more clear when we walk through the following:

```java
Knight k1 = new Knight();
Knight k2 = new Knight();
Piece k3 = new Knight();
k1.capture(k2);
// We resolve k1.capture by looking at the static type of k2
// Since k2 is of type Knight, we have a match for the capture(Knight k) method!
// This means the result is 'Capturing Knight!' being printed

k1.capture(k3)
//k3's static type is Piece. Therefore we resolve this to the capture(Piece p) method!
// This means the result is 'Capturing Piece' being printed
```

Easy right? Not so fast! What about inheritance and the idea that Knights ARE Pieces? What if in the above, there wasn’t a `capture(Knight k)` method in the **Knight** class and we only had `capture(Piece p)`. Would the method call `k1.capture(k2)` resolve? YES, because Knights ARE Pieces! This brings up a very important issue with inheritance: When we resolve methods using static types, we start with the most specific type of the reference variable being passed in as the argument to see if a signature matches,
Okay! Now that we have a better understanding of methods, signatures, and how methods are called, let’s go back to the examples. These are going to be referencing the last set of Piece, Knight and Rook classes in the back of this paper.

```java
Piece p1 = new Piece();
Piece p2 = new Knight();
Knight p3 = new Knight();

//Piece->Piece captures piece
p1.capture(p1) // --> "I’m bored."

//Knight->Knight captures Knight
p3.capture(p3); // --> "TOO SLOW BUDDY."

//Piece->Piece captures Knight
p1.capture(p3) // --> "I’M BORED."

//Piece->Knight captures Piece
p2.capture(p2) // --> "Knight is bored."

//Piece-Knight captures Knight
p2.capture(p3) // --> "Knight is bored."
```

The first few are pretty straightforward and resolve easily. But things get interesting starting with `p1.capture(p3)`. What’s going on here? We’ve got a Piece variable referencing a Piece object, and we call its capture method with a Knight variable as a parameter. Remember, we resolve method arguments by static type, so we check the Piece class for a method with this signature. None exists, but it does have a `capture(Piece p)` method, and since `p3` is of type Knight which is a subclass of Piece, this matches! Thus we invoke this method and get "I’M BORED", even though there’s no `capture(Knight)` method. Inheritance and polymorphism to the rescue!

The next two are much more interesting. These are instances where dynamic method lookup takes place. Let’s first revisit the TV Analogy from lecture.

### 4.3 The TV Analogy

Suppose you have an old TV with an old remote control. This old TV has the ability to change channels and change the volume. The old remote control has buttons which correspond to those TV functions, so everything works properly! Now suppose we upgrade the TV to a new TV with the ability to split the
screen. The old remote control does not have a split button, so no matter how hard we try, we can’t get the new TV to split screens with this old remote.

So what do we do? We upgrade the remote to a new remote! This new remote has more specific functionality than the general old one, and this can invoke the new TV’s ability to split the screen. Thus by upgrading the remote, we’ve gained access to the TV split screen capability. What if we have two new TVs (TV A, and TV B) that can both split screens, except that ‘A’ will split the screen horizontally, and ‘B’ will split the screen vertically? How will the new remote work? It only has one split button. This button invokes the same operation on each TV, but it’s the TV that ends up resolving how to perform the split operation! Calling the same split function from the same remote on two different TVs lead to different behavior because the split function is defined differently on each TV and is actually invoked there.

This is a nearly perfect example of what dynamic method lookup is. It’s the idea that we need to check the reference variable type that’s referencing an object (the remote) to make sure the method we want to invoke is actually there (the presence of the split button on the remote) before we can call that method. However, once that method has been shown to exist and we invoke it, we resolve the method based on the actual class of the object the variable is referencing (the TVs with different splits). Note that the actual class of the object can be different than the reference variable because of inheritance and the IS-A relationship between superclass and subclass. Piece p = new Knight() is an example of this. Thus we look to the static reference variable type to find the methods we can call, but the behavior is dictated by the actual dynamic class of the object that the variable references. This is why we call this process dynamic method lookup.

This should clear up things with p2.capture(p2). We call the capture(Piece) method of the Piece class since the static type of p2 is Piece, but the actual capture(Piece) method that gets invoked is of the dynamic type of p2, which is a Knight! We press the remote button, but let the TV that it controls resolve the behavior we ask for. That’s why we get “Knight is bored.”

Great! Now let’s look again at p2.capture(p3). What is the static type of p2? It’s a Piece! Just like the remote, we need to see what methods it has and whether what we are calling matches the signature of any of them. p3 is of static type Knight, but there’s no such method capture with parameter Knight. However, there IS a method capture(Piece). This matches, and remember that methods are defined uniquely by their signature! Thus we invoke the method capture(Piece p) on the object p2 references, even though we know p3 is of type Knight! Even though p2’s dynamic class has a method capture(Knight k), the static type class does not, so we CANNOT ACCESS IT. This is why we get ”Knight is bored.”

I hope this clears some things up about inheritance. There’s a lot more to learn though! Please continue asking questions in lab and on Piazza. Also, DO experiments with subclassing! That’s the best way to justify to yourself the behavior that inheritance brings to classes.
Appendix 1: Initial Classes

```java
public class Knight {
    String color;
    int[] position;

    move() {
        //update position
    }

    captureKnight(Knight k) {
        // Do stuff
    }

    captureRook(Rook k) {
        // Do stuff
    }
}

public class Rook {
    String color;
    int[] position;

    move() {
        //update position
    }

    captureKnight(Knight k) {
        // Do stuff
    }

    captureRook(Rook k) {
        // Do stuff
    }
}

class Board {
    Knight[] knights;
    Rook[] rooks;

    //COMPLICATED CODE
}
```
4.3 The TV Analogy

Appendix 2: New Piece Method

```java
class Piece{
    String color; // White or Black
    int[] position;
    String type; // Rook, Knight, etc.
    public void move()
        if (type.equals(Rook)) { ... }
        else if(type.equals(Knight)) { ... }
        ....
    }
    public void capture(Piece p){
        if (p.type.equals(Rook)) { ... }
        else if(p.type.equals(Knight)) { ... }
        ....
    }
}

class Board{
    Piece[] pieces;
    public Board()
        pieces = new Pieces[16];
        pieces[0] = new Piece(Rook);
        pieces[1] = new Piece(Knight);
        ....
    ....
    // All future methods can iterate through pieces as needed, no need for separate things
}
```
4.3 The TV Analogy

Appendix 3: Inheritance Implementation

```java
public class Piece{
    String color;
    int[] position;

    public void move(){
        System.out.println("Moving.");
    }

    public Piece(){
        color = "";
    }

    public void capture(){
        System.out.println("Capturing.");
    }

    public void capture(Piece p){
        System.out.println("I'm bored.");
    }

    public void capture(Knight k){
        System.out.println("TOO SLOW BUDDY.");
    }
}

public class Knight extends Piece{

    public void move(){
        System.out.println("I am Batman.");
    }

    public void capture(){
        System.out.println("YO.");
    }

    public void capture(Piece p){
        System.out.println("Knight is bored.");
    }

    public void capture(Knight k){
        System.out.println("TOO SLOW BUDDY.");
    }
}
```