INHERITANCE AND TYPE HIERARCHIES #2

Prof. Chris Jermaine
cmj4@cs.rice.edu

Review: What's Useful About This Stuff?

- Inheritance
  - — Useful because you can re-use code
  - — Cuts down SLOC, avoids copy/paste errors, makes maintenance a lot easier
- Polymorphism
  - — Much more subtle!
  - — Useful because you can separate "policy" from "mechanism"
  - — Often spend time building a skeleton (our nested loops)... "mechanism"
  - — Then spend even more time building in the control structures (edit ops)... "policy"
  - — Using polymorphism, easy to extract away specific policies, only worry about interface or how the policy controls the mechanism

Let’s Look At a More Complicated Example

- A Java code for playing checkers
- Around 650 SLOC, including comments
  - — Will attempt to demo a version without an AI
  - — But it’s really easy to add a brute-force AI into the mix (will do subsequently)
- What do I hope you take home from this?
  - — You’ll agree inheritance, polymorphism are very useful tools
  - — You’ll see some examples of these ideas in action
  - — And you’ll appreciate that there’s no one way to apply them

How Is This Code Organized?

- All starts with the abstract “AChecker” class
  - — A checker is aware of its position, type (king or regular), color (red or black)
  - — Note that color is more than just a boolean...
  - — It controls the meaning of things like forward, back, etc.
  - — Thus, “AChecker” will have two subclasses: “RedChecker”, “BlackChecker”

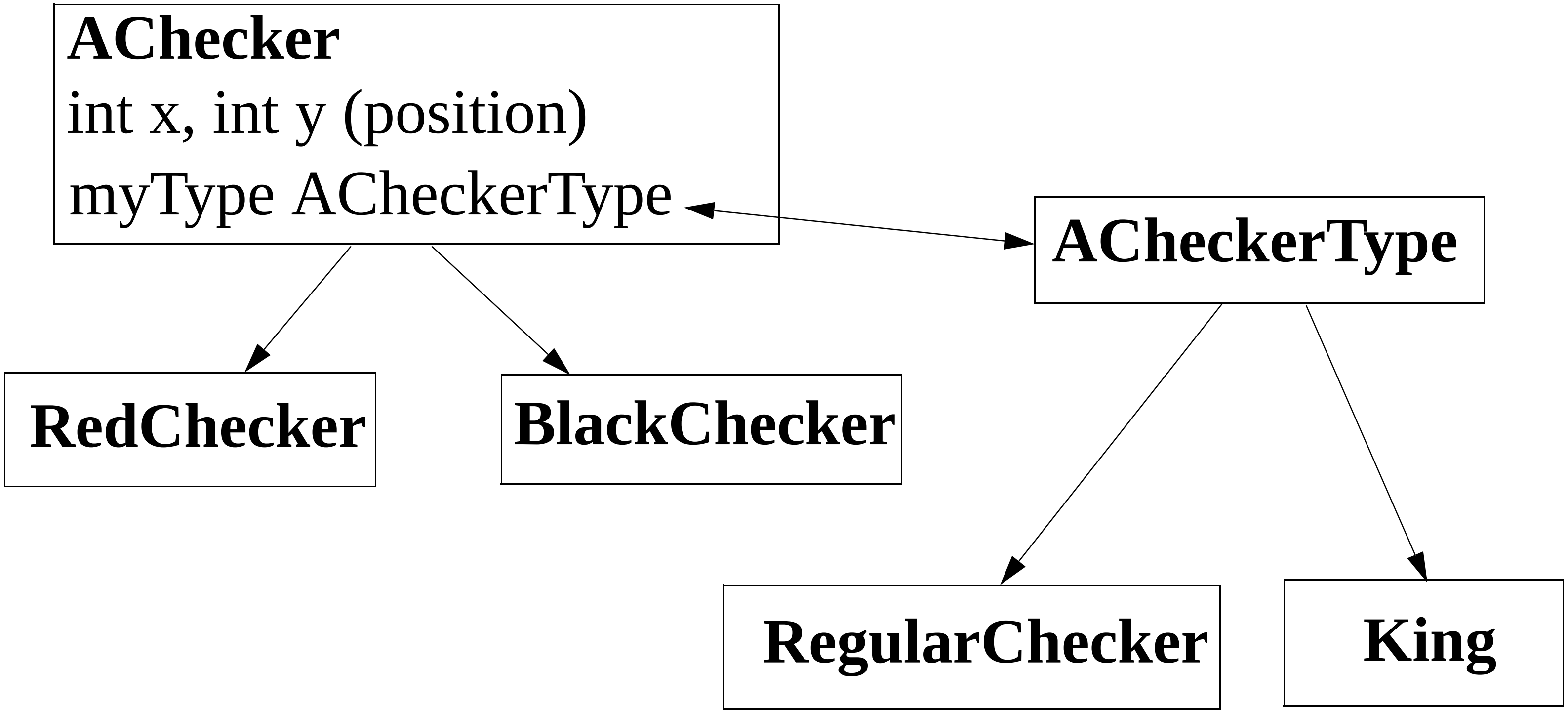
How Is This Code Organized?

- All starts with the abstract “AChecker” class
  - — A checker is aware of its position, type (king or regular), color (red or black)
  - — Note that color is more than just a boolean...
  - — It controls the meaning of things like forward, back, etc.
  - — Thus, “AChecker” will have two subclasses: “RedChecker”, “BlackChecker”
- How to deal with kings vs. regular pieces?
  - — Tough for two reasons...
  - — One: don’t want to have 2 times 2 = 4 different classes defined
  - — Two: the type of the checker can actually change! But types are immutable in Java

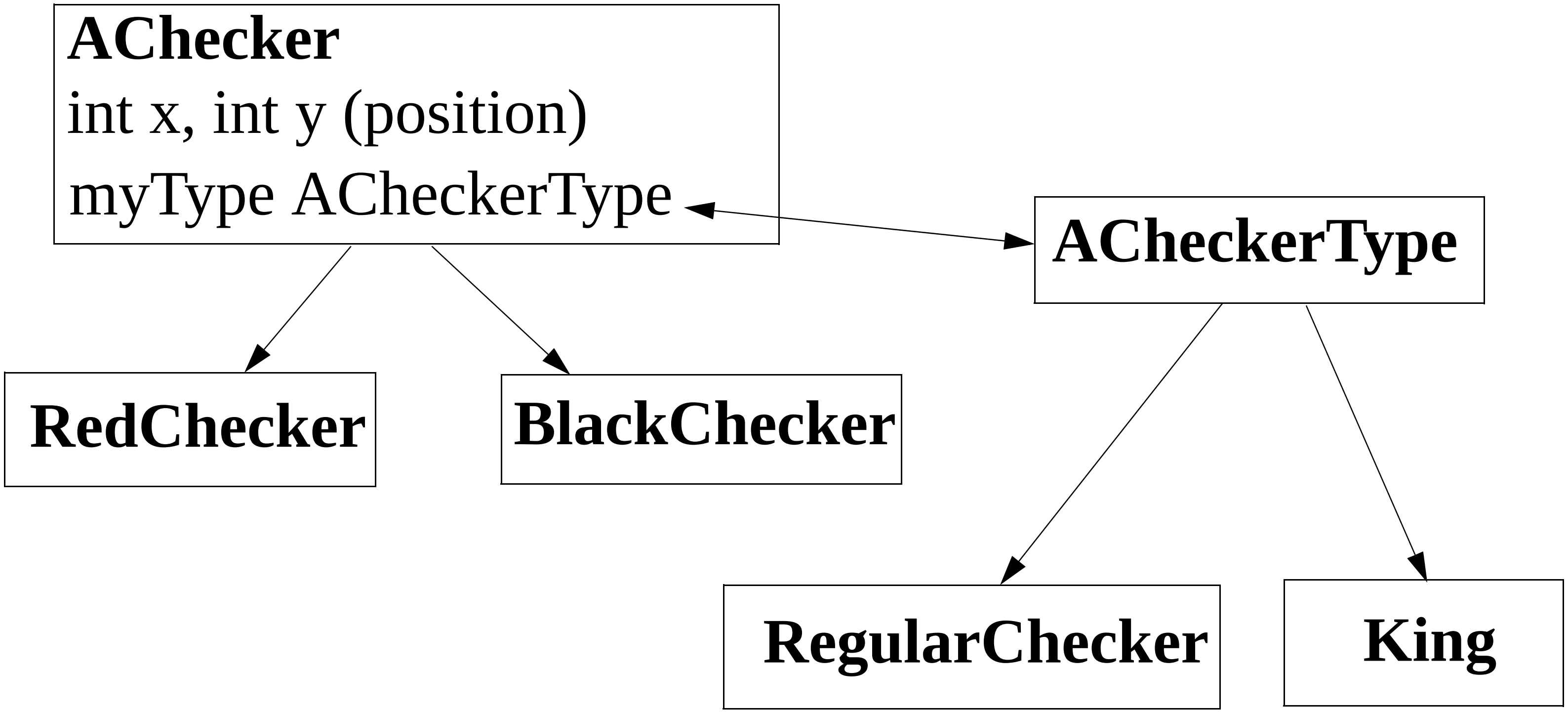
How Is This Code Organized?

- All starts with the abstract “AChecker” class
  - — A checker is aware of its position, type (king or regular), color (red or black)
  - — Note that color is more than just a boolean...
  - — It controls the meaning of things like forward, back, etc.
  - — Thus, “AChecker” will have two subclasses: “RedChecker”, “BlackChecker”
- How to deal with kings vs. regular pieces?
  - — Tough for two reasons...
  - — One: don’t want to have 2 times 2 = 4 different classes defined
  - — Two: the type of the checker can actually change! But types are immutable in Java
- Solution: the “ACheckerType” class
  - — Has two sublcasses: “RegularChecker” and “King”
  - — The “AChecker” class has a member variable of type “ACheckerType”

Here's a Picture



Here's a Picture



Note: can get away with this because we can factor (almost) all checker functionality into three groups: functions depending upon color, those depending upon king or not, and those that are the same for all checkers. 2 function versions, not 4!

Moving Checkers on the Board

- The reason we have both kings & regular checkers:
  - — They define possible movements of the piece
  - — Kings can go backward, regular pieces can't
- Thus, an "ACheckerType" object is basically a factory
  - — It spits out all of the moves associated with the type of checker...
  - — The possible moves are all of type "AMove"
  - — Are eight different subclasses of "AMove"
  - — "ForwardLeft", "ForwardRight", "JumpBackwardLeft", etc.

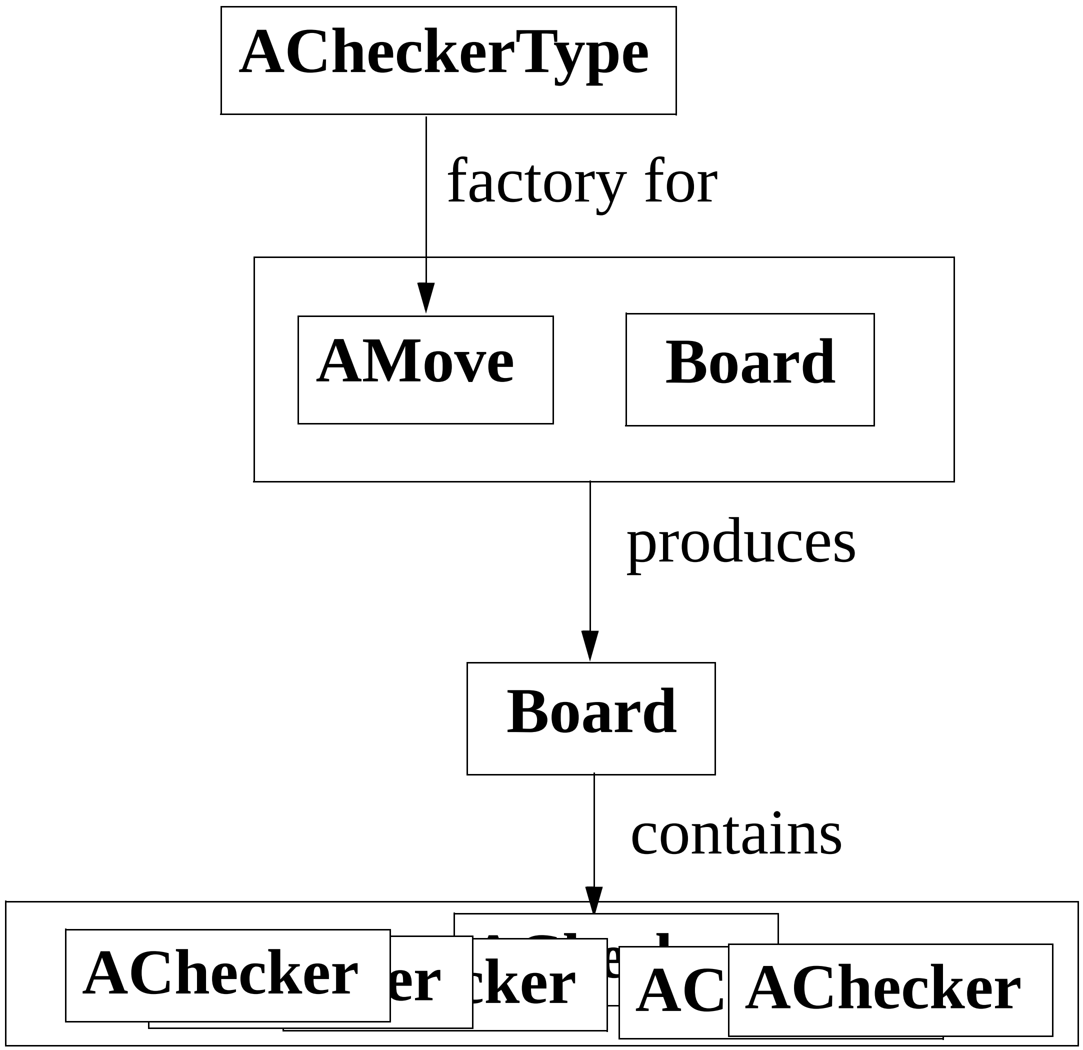
The Checkerboard

- Not surprisingly, we also have a “Board” class
- A “Board” object is a container for “AChecker” objects
  - It is designed sort of like a LIFO stack
  - LIFO is nice ‘cause when you remove a checker, move it, put it back on, it’s the next one to come off the stack (convenient when doing chains of jumps)

An AMove Is a Board-to-Board Function

- An “AMove” object basically maps a board to a board
- That’s what a move is, right?
- Given all of this, here’s what the checkers code does, repeatedly:
  - The user selects a checker
  - We ask the checker for all of its possible moves (via its “ACheckerType” object)
  - Those moves are checked for applicability
  - Moves that survive are presented to the user
  - The user selects a move
  - Then the selected move is applied to obtain a new board
  - And the cycle repeats again!

Here’s a Picture



One More Bit of Complication…

- “AMove” objects move checkers using four simple motions:
  - “forward”, “left”, “right”, “backward”
- But as alluded to before, these are color-dependent
  - So “AMove” objects don’t move checkers directly
  - Instead, they call abstract “forward”, “left”, “right”, “backward” ops
  - Then the particular “AChecker” subclass figures out how to actually run them

Questions?

- Now let's look at some code!

The AChecker Class

```
abstract class AChecker {

  // the position of the checker (x is horizontal)
  private int x, y;

  private ACheckerType myType = new RegularChecker ();

  // check to see if the same color
  public boolean sameColor (AChecker asMe) {}

  // get all possible moves for this piece...
  public ArrayList <AMove> getAllMoves (
    Board curBoard, boolean onlyChainable) {}

  // print the checker
  public void print () {
    PrettyCheckerPrinter.print (myType, this);
  }
```

The AChecker Class (cont)

```
  // these allow us to change the position of the checker...
  abstract public void moveForward ();
  abstract public void moveBackward ();
  abstract public void moveLeft ();
  abstract public void moveRight ();

  // get and set the x and y position of the checker...
  // used only by the implementations of the “move” methods
  protected int getX () {}
  protected int getY () {}
  protected void setX (int newX) {}
  protected void setY (int newY) {}

  // xform into a king
  protected void makeKing () {}

  // some other stuff...
}
```

The RedChecker Class

```
class RedChecker extends AChecker {

  public RedChecker (int x, int y) {
    setX (x);
    setY (y);
  }

  public void moveForward () {
    setY (getY () + 1);
    if (getY () == 7) // end of the board, we're a king!
      makeKing ();
  }
  public void moveBackward () {
    setY (getY () - 1);
  }
  public void moveLeft () {
    setX (getX () - 1);
  }
  public void moveRight () {
    setX (getX () + 1);
  }
```

The RedChecker Class

```
class RedChecker extends AChecker {

  public RedChecker (int x, int y) {
    setX (x);
    setY (y);
  }

  public void moveForward () {
    setY (getY () + 1);
    if (getY () == 7) // end of the board, we're a king!
      makeKing ();
  }
  public void moveBackward () {
    setY (getY () - 1);
  }
  public void moveLeft () {
    setX (getX () - 1);
  }
  public void moveRight () {
    setX (getX () + 1);
  }
```

Polymorphism! If ask a checker to move itself, will depend upon whether it is red or black

getAllMoves

- What does “getAllMoves” in the AChecker class do?

```
// get all possible moves for this piece... onlyChainable
// indicates we are restricted to running chainable moves
public ArrayList <AMove> getAllMoves (
  Board curBoard, boolean onlyChainable) {

  ArrayList <AMove> returnVal =
    myType.getAllMoves (curBoard, this);

  // kill all of the moves that do not work
  for (int j = returnVal.size () - 1; j >= 0; j--) {
    if (!returnVal.get (j).isApplicable (curBoard, this) ||
     (onlyChainable && !returnVal.get (j).isChainable ()))
      returnVal.remove (j);
  }
  return returnVal;
}
```

Uses the type to generate all moves

Then Implementing a Type is Easy…

```
class RegularChecker extends ACheckerType {

  public ArrayList <AMove> getAllMoves (
    Board currentBoard, Checker oneToMove) {

    ArrayList <AMove> returnVal = new ArrayList <AMove> ();
    returnVal.add (new JumpForwardLeft (currentBoard, oneToMove));
    returnVal.add (new JumpForwardRight (currentBoard, oneToMove));
    returnVal.add (new ForwardLeft (currentBoard, oneToMove));
    returnVal.add (new ForwardRight (currentBoard, oneToMove));
    return returnVal;
  }
}
```

— Polymorphism! Different checker types generate different types of moves, by re-defining the “getAllMoves” method.

— So a given checker asks its type to generate its moves

So What Does This Design Give Us?

- Want to know how you can move a checker? Just ask it for its possible moves via “getAllMoves”
- Want to actually move a checker? Ask it to move via “moveForward”, “moveBackward”, etc.
  - And it will correctly take into account its color when moving!
- Now let’s look at the AMove class.

The AMove Class

```
abstract class AMove {

  // the checker we are trying to move, and the move’s name
  private Board result;
  private String myName;

  // get the result of applying the move...
  public Board apply () {
    return result;
  }

  // see if the move is actaully applicable
  public boolean isApplicable () {
    return (result.isValid ());
  }
```

```
  // allows sublcass to set the board resulting from the move
  protected void setBoard (Board inVal) {
    result = inVal;
  }

  // allows a subclass to set the name
  protected void setName (String inName) {
    myName = inName;
  }

  // see if the move can be chained with others
  public abstract boolean isChainable ();

  // print out a description of the move to the screen
  public void print () {
    System.out.println (myName);
  }
}
```

What Does a Specific Move Look Like?

- Just sets up the resulting board in its constructor

```
class JumpForwardRight extends ChainableMove {
  public JumpForwardRight (Board applyToMe,
    Checker pieceToMove) {
    Board returnVal = applyToMe.copy ();
    Checker myGuy = pieceToMove.copy ();
    myGuy.moveForward ();
    myGuy.moveRight ();
    returnVal.jumpChecker (myGuy);
    myGuy.moveForward ();
    myGuy.moveRight ();
    returnVal.push (myGuy);
    setBoard (returnVal);
    setName ("Jump forward right");
  }
}
```

What Are We Missing?

- The "Board" class... but that's easy; just a container for checkers
  - — With a few other ops for doing things like jumping a checker
- Also missing a "main" method...
  - — Can we get some pseudo-code for this?

Questions?