Table of contents

1 Preliminaries

Due: At the beginning of lecture on Tuesday, 3 November. You may turn it in before that date in the Carpenter consulting room. Do not turn it in at Carpenter on the due date.

Working with another person. You may do this assignment with one other person. However, you must share the work equally; you should write the code together, type it in together, and debug it together. It is not okay for one person to write and debug one method and another person to do another method; that is not togetherness. We want togetherness. Hand in only one assignment with both names on it.

Note. The first comment of your program must contain the name, Cornell ID, section day, section time, and section instructor of each person who is to receive a grade for the assignment you hand in. Don't write it in by hand; type it in as the first comment of the file that contains class Checkers.

Goals of assignment. (0) To have you work with a well-designed, well-documented program. As you study it, be conscious of the comments near declarations of variables that define the meaning of the variables, the specifications of methods, the statement-comments within method bodies, and the indentation. (1) To get you to work with a 2-dimensional array. (2) To introduce you to GUIs.

Grading. Section 9 describes the sections of code that you have to write. For each of them, you can get from 0 to 3 points, based on correctness (2) and style (1) of doing it (keep things simple and clear). None of these sections of code have to be long; the work will be in understanding the project as a whole and keeping the details straight.

What to hand in. The description of what you are to do for this assignment appears in Sec. 9, "The code that plays checkers". That section takes you through a series of steps, which end up with a complete program.

Hand in a copy of class *Checkers*. Also hand in a printout of one snapshot of a game played by your program, some point after the first move.

Finally, write (and hand in) a one-page essay on your experiences with this assignment. You may say whatever you want, but think about answering the following kinds of questions. Was it interesting? Worthwhile? What did you think about it before initially, halfway through, and after finishing it? What did you learn? Feel free to say negative as well as positive things; be honest. What you say will not affect your grade.

2 Synopsis

By a user interface we mean a method by which a user inputs data into a computer program or receives output. Thirty years ago, input was mostly by means of information stored on "punch cards" and information stored in some manner on a magnetic tape (usually produced by some other computer program). (A magnetic tape was, and still is, like an audio cassette tape.) Output would be printed on a piece of paper, stored in a file on a magnetic tape, or punched on cards. Relatively few monitors and interactive keyboards were in use.

Today, for input, one uses a keyboard and mouse (and perhaps other devices like a joystick) attached to a computer in connection with a set of "windows" on a

computer monitor, which allows one to type in information in many different places, to press "buttons", and so forth. Output, as you already know, can be in many forms —on the computer screen in the form of text, graphics, tables, and images, on paper, and on various storage devices like floppy disks, zip drives, and hard disks.

3 Files to retrieve

Browse the CS100A home page and find the material for assignment 7. You will find:

(a) A link to a version of our CodeWarrior program that you can execute from your browser. Execute this to become familiar with the game and with the GUI (Graphical User Interface) for the game. It is important to do this *before* trying to write code for this assignment. Look at the source for this html file to see something about how it works.

(b) Four files: TrivialApplication.java, Checkers.java, CheckersSquare.java, and IntLabel.java. These contain the classes that make up this implementation of the game of checkers, except that some sections of code have been removed from class Checkers; your task is to complete these sections. See Sec. 9.

(c) File Experiment.java, which you can use to learn about developing a Graphical User Interface, as explained in lecture on 27 October.

You should start a new CodeWarrior project using the Java stationery (not the CUCS Java stationery or the CUCS Java Graphics stationery). Then, replace file TrivialApplication.java in that project by the one mentioned in (b) above. Also, move copies of the other files mentioned in (b) into this project folder and then add them to your project. (The latter can be done by opening them and then using menu item "Add window").

You can compile and execute the program, but it won't perform suitably until you fill in the missing sections in class Checkers.

4 The game of Checkers

The game of Checkers is played by two people, say R and B, on an 8 x 8 board on which black and red pieces are placed. The initial configuration, which can be seen below or by starting up the program mentioned in (a) above, contains twelve red pieces and twelve black pieces, all on black squares. R plays first; the pieces move only on black squares since pieces can move only diagonally. A piece can move into a diagonally adjacent empty square or it can diagonally jump an enemy piece, taking it off the board. (In our present implementation, only one piece can be jumped on a move.) Normally, pieces can move only toward the side of the board opposite from which they started. When a piece reaches the last row, it becomes a king. Kings can move in any (diagonal) direction.

The initial configuration of the board is:

	R		R		R		R
R		R		R		R	
	R		R		R		R
B		B		B		B	
	B		B		B		B
B		B		B		B	

All pieces lie on black squares of the checkerboard.

The game ends when one player has no pieces left.

When you look at the program, consider how could you change it to permit a jump of two or more pieces.

5 The GUI

A set of Java classes, called the *abstract window toolkit* (awt), comes with each Java programming environment. The toolkit provides classes for the windows, menus, text fields, buttons, etc., that one sees on a computer screen these days. The classes are defined in an abstract, machine-independent fashion and are then implemented in a machine-dependent (or operating system-dependent) fashion in each operating system. Thus, if you run a Java program

that uses this toolkit on a Macintosh, you will get Macintosh-like buttons, textfields, etc., and if you run the same program on a PC running Windows 95, you will get Windows-95-like buttons, textfields, etc. The machine-independent nature of the Java definition and its faithful implementation on different platforms helps make Java a success today, in the context of the world-wide web.

This discussion of the awt is only an overview. We encourage you to spend some time looking at some of the awt classes, like Button, after you read this section. Study some of the methods in these classes, get a feel for the overall structure. This may enable you later to develop your own GUI for a small project.

Also, while reading this section, refer often to the appropriate parts of classes Checkers and CheckersSquare.

One obtains the ability to reference the classes of the abstract window toolkit by placing the statement

```
import java.awt.*;
```

at the beginning of a Java program file.

In this assignment, you will not have to write any code that deals directly with the awt —the code you write will deal with arrays. The awt is too large and the idea is too new to you. And, you don't have enough programming experience to be expected to program using the awt in such a short time. The awt is discussed in this assignment only to acquaint you with building GUI's in Java. Studying this assignment and the code should prove enlightening in this regard.

Below is a list of the classes that the Checkers program uses. (There are more classes in the awt, which we don't show you.) Indentation denotes subclassing. For example, Frame is a subclass of Window, which is a subclass of Container, which is a subclass of Component. Below this list, we describe the classes briefly. At the end of this document, we provide a picture of Checkers being played, with the various components indicated.

```
Component
    Container
        Window
            Frame
        Button
        Canvas
        Label
        TextComponent
            TextArea
            TextField
Color
Event
Graphics
GridBagConstraints
GridBagLayout
```

Component is the superclass of many classes that appear on the screen; a Component (i.e. an instance of class Component) has a position, has a size, can be painted on the screen, and can receive input "events" (keyboard strokes, mouse clicks, etc.).

Container represents all components that can hold other components.

Button: a labeled button.

Canvas: a rectangular area with graphics capability.

Color: the colors that can be used.

Event: The representation of an event, like a mouse down, or a keystroke, or a mouse drag.

Frame: a Frame (i.e. an instance of class Frame) is a top-level Window with a title and a border. By "top-level", we mean that it does not appear within some other component on the screen, but appears by itself. A Frame can have a menu. The awt sends the Frame all mouse and keyboard events that occur over it.

Graphics: the class that enables drawing graphical figures in (some) components, like a Frame or a Canvas.

GridBagConstraints contains constraints that are used for laying out components using class GridBagLayout.

GridBagLayout: a "layout manager" that places components in a window (or frame, etc.). The idea of a layout manager is discussed below.

Label: a component for placing text in a container. The text can be changed by the application, but a user cannot edit it directly.

TextComponent: a component that allows editing of text in it by the user.

TextArea: a component that is a two-dimensional text area with scroll bars. The user can edit it (unless the code makes it uneditable).

TextField: a component that is a single line of text. The user can edit it (unless the code makes it uneditable).

Window: a top-level window (one that does not appear in some other component), with no title or menu bar.

6 Dealing with the Frame

6.1 The fields (variables) of a Frame

Look at method *TrivialApplication.main*. It prints a message and declares and initializes variable *game* using the statement

```
Checkers game= new Checkers();
```

That is all it does! Obviously, a lot must go on in constructor *Checkers*.

Now turn to class *Checkers* and notice that it extends class *Frame*, so that *Checkers* is a subclass of *Frame*. We now investigate constructor *Checkers*.

A *Frame* is a stand-alone window that can appear on your computer monitor. The purpose of constructor *Checkers* is to add the components that make up the game Checkers (as you see it on the monitor) and then place the *Frame* on the monitor. Look at the beginning of class *Checkers*. The first declarations define the variables that will contain the components that will be in the *Frame*. Variable *board* will be an 8 x 8 array of elements of class *CheckersSquare*, which is a subclass of class *Canvas*. Each array element *board*[*i*][*j*], then, is a (red or black) square of the checkerboard, and it can be drawn on using graphics methods like *drawOval*.

Following variable *board* are variables of class *Label* (they contain the strings of characters that appear on the right side of the *Frame*) and a variable *helpText* (which contains the help information that appears at the bottom of the *Frame* when the *Help* button is pressed).

Finally, array *buttons* contains the *String*s that appear on the three buttons in the upper righthand corner of the *Frame*.

The components just mentioned have to be placed in the *Frame* in appropriate positions. For example, in our *Frame*, the game board of *Canvas* components appears in the upper left of the *Frame* and the buttons in the upper right. In the Java *awt*, the placement of components is done using a *layout manager*. There are several layout managers, each of which has a different approach to describing how the components are to be placed.

In this project, we use layout manager *GridBagLayout*, which uses an instance of class *GridBagConstraints* to do its job. Variable *gb* contains the layout manager and variable *gbc* contains the associated instance of *GridBagConstraints*. The declarations of *gb* and *gbc* come next in class *Checkers*.

Following the declarations of *gb* and *gbc* come declarations of variables that are used to simulate a game of checkers. These have comments that describe their meaning, so we don't discuss them here. Note that only six variables are needed here! Information about what pieces are actually on the board is contained in the individual elements of array board.

6.2 Overview of constructor Checkers

Now look at the body of constructor *Checkers* in order to get an overview of what it does and how it does it. Some of the methods that it calls are in superclass *Frame*, so you would have to look at that class to get a rigorous definition of those methods.

The initial call *super*("*Checkers*") provides the *Frame* with its title. Next come assignments that generate instances for variables *gb* and *gbc*, set the font suitably, and tell the *Frame* which layout manager to use (*setLayout*(*gb*)).

Then comes a sequence of sections of code to (0) create the array of *CheckersSquare* elements (which make up the game board) and add them to the *Frame*; (1) create the buttons and add them to the *Frame*, (2) add the informational *Labels* to the *Frame*.

Finally, we have the following statements: (0) *pack*() causes the components in the window to be laid out using their preferred size; (1) *move*(150, 150) indicates where the upper left corner of the *Frame* is to be placed on the screen; (2) *newGame*(), which we wrote, places the initial pieces on the board and sets the informational labels accordingly, and (3) *show*() causes the *Frame* to be drawn on the monitor.

Note that the *TextArea* containing the help information has not been added to the *Frame*.

At this point, note how the constructor is presented. There is a comment for each part, which describes at a high level, but precisely, what that part does. Thus, you can get a good sense of the constructor without understanding all the details. Later, you can investigate each of the parts in detail. Thus, you can understand the constructor at several "levels of abstraction".

6.3 Adding a component to the Frame

Layout manager *GridBagLayout* views the *Frame* as a two-dimensional table of columns and rows. A component can occupy any rectangular piece of this table. So, component *A* could occupy the element at column 0 row 0, while component *B* could occupy the 2 x 2 set of elements whose left upper element is in column 1 row 1, as shown below:

	0	1	2	3	4	5	6	
0	A							···
1		B	B					···
2		B	B					···
⋮								

In the *Frame* of game *Checkers*, each square of the board takes up a 2 x 2 array of elements of the *Frame*, as shown below. This is so that the labels to the right don't take up so much space. In the diagram, a number "ij" in a square indicates that the square is used as part of the checkerboard square in column i, row j —remember, there are two different two-dimensional tables used here, the checkerboard and the *Frame* on which it is drawn. To the right, "quit" in two squares indicates that the menu button titled "quit" occupies those squares. And so forth.

00	00	10	10	...	70	70	new game
00	00	10	10	...	70	70	new game
01	01	11	11	...	71	71	quit
01	01	11	11	...	71	71	quit
02	02	12	12	...	72	72	help
02	02	12	12	...	72	72	help
03	03	13	13	...	73	73	red to play
03	03	13	13	...	73	73	(crosshair)
⋮							⋮

Before adding a component to the *Frame* using the *GridBagLayout* manager, one must set various "constraints" in the associated instance *gbc* of *GridBagConstraints*. As you read this description, look also at method *Checkers.add*. Here are some of the properties, or constraints, that you can set:

- The column number *gbc.gridx* and row number *gbc.gridy* of the upper left element of the grid that a component is to occupy.
- The numbers *gbc.gridwidth* and *gbc.gridheight* of columns and rows of the grid that this component is to occupy.
- The relative horizontal weight *gbc.weightx* and relative vertical weight *gbc.weighty* that is to be used when the *Frame* is resized (e.g. made bigger or smaller by the user). Usually, these numbers are between 0 and 100. We illustrate by example how these are used. If *gbc.weightx* is 0 for a component, then the width of the component will not change, no matter how big or small the *Frame* becomes. If *gbc.weightx* for this component is 100 and *gbc.weightx* for another component is 50, then when the *Frame* is made bigger, this component receives twice as much space than the other. Thus, these are relative weights.
- *gbc.fill* indicates how much of the space allocated to a component it should actually occupy. If *gbc.fill* = *gbc.NONE*, then it occupies the minimal amount of space. If *gbc.fill* = *gbc.HORIZONTAL*, then it occupies the minimal amount of vertical space and maximum amount of horizontal space. If *gbc.fill* = *gbc.VERTICAL*, then it occupies the maximum amount of

vertical space and the minimum amount of horizontal space. If *gbc.fill* = *gbc.BOTH*, then it occupies the maximum amount of vertical space and the maximum amount of horizontal space.

You can get an idea about the use of *gbc.fill* by changing the assignment to gbc.*fill* within the code that adds the buttons to the *Frame* to

gbc.fill = *GridBagConstraints.NONE*;

and executing the program. Be sure to change *NONE* back to *BOTH* after noticing this difference.

To add component *c* (say) to the *Frame*, one first sets the constraints in *gbc* as desired, lets *GridBagLayout* manager *gb* know about these constraints by executing *gb.setConstraints*(*c*, *gbc*), and finally adds the component to the *Frame* by calling method *add* of the *Frame*. You can see this sequence of statements in the body of method *Checkers.add*.

Now, this is a rather long and torturous sequence of statements. To simplify and shorten all this, we wrote another method *Checkers.add*, which is used throughout the constructor to add the components to the *Frame*. For example, study the code that adds the squares of array board to the *Frame*.

6.4 Handling events over the Frame

Pushing down a mouse button when the cursor is in some square of the game board, letting the mouse button up, pushing a button, and clicking the mouse when the cursor is in the "destroy window" box (upper right hand corner on the PC, upper left corner on the Mac) are called *events*. When one of these events take place, the program must recognize it and take appropriate action.

Such events are modeled as instances of class *Event*. Class *Event* contains constants to represent events. Thus, integer constant *Event.MOUSE_DOWN* represents pressing the mouse button down. Below are some event constants and their values —you can take a look at class *Event* to see what events your program could recognize and process:

```
MOUSE_DOWN  = 501
MOUSE_UP    = 502
MOUSE_MOVE  = 503
MOUSE_ENTER = 504
MOUSE_EXIT  = 505
MOUSE_DRAG  = 506
```

There are two ways for the system to tell the program that an event has occurred within the *Frame*:

(0) call method *action* (if a button has been pressed)

(1) to call method *handleEvent* if some other event has occurred.

Look carefully at these two methods, and you will see that determining and processing the events is rather straightforward.

7 Class CheckersSquare

As mentioned earlier, each element of 8 x 8 array *board* is an instance of class *CheckersSquare*, which extends class *Canvas*. Thus, an instance of *CheckersSquare* can be a component of the *Frame* for the game, and one can draw circles, etc. on it.

The class contains five constants (variables with property **final**): *EMPTY*, *RED*, *BLACK*, *REDKING*, and *BLACKKING*, which are used to indicate what type of piece sits on the square. These constants are **static**, which means that they belong to the class rather than to an instance of the class.

Each instance of class *CheckersSquare* has variables that contain the column and row of the instance on the game board as well as the contents of the square (*EMPTY*, *RED*, *BLACK*, *REDKING*, or *BLACKKING*). There are also variables that give the background color of the square since it needs to be different from red or black; in addition there is a boolean variable *toBeMoved*.

Now look at constructor *CheckersSquare*. First, it calls the constructor of the superclass (*Canvas*); then it saves the desired column and row number for this square. Finally, it uses three methods of class Canvas to set the background color, to set the preferred size of the component, and to paint the component (that is the purpose of method *repaint*). Function *bounds* yields the perimeter of the canvas as an instance of

class *Rectangle*, which is stored in *b*.

Since this component can reference *Graphics* methods, it has a method *paint* to draw on it. You have seen such paint procedures before, so this one should be relatively easy to understand. Note that it saves the current color for the component and then restores it at the end. If the square has a piece on it, the piece is drawn with red or black ovals for the top of the piece and with underlying magenta or gray ovals, which provide the apparent side of the piece. A yellow K is drawn on a king.

CheckersSquare has ten other methods, which are simple and short. Note that three of them repaint the square if they make changes to it. To see why this is necessary, comment out a repaint call, execute the program, and notice what happens when pieces are placed on the board. Be sure to remove the comment symbol after this experiment.

8 Class IntLabel

The numbers of black and red pieces on the board are printed on the GUI. This requires that the numbers be kept in two forms: the integers themselves and the *Label* that contains the *String* representation of the integer. Requiring the program to maintain these two forms is frought with danger —if one form is changed without changing the other, which is quite likely, an error results.

Object-oriented programming helps us guard against such mistakes. We encapsulate in class *IntLabel* both of these representations. Take a look at this class – note its methods. Then notice the two fields *redCount* and *blackCount* that are declared in class *Checkers*.

9 The code that plays Checkers

In this section, we discuss how the code plays checkers and describe what you are to do in this assignment. We have removed sections of the code, which you have to fill in. We take you through a series of steps to complete the program.

First, familiarize yourself with the variables that are used in the game —the fields in class *Checkers*. Also become familiar with classes *CheckersSquare* and *IntLabel*, because you will be reading and writing calls on their methods. Then study method *Checkers.processSquareClick*, which, as it says, processes a mouse click in a square. Read the high-level comments first, as well as the specifications of the methods that it calls, to get an idea how it works.

9.1 Method newGame

This method sets everything to play a new game. At the moment, it puts only the 12 red pieces on the board. Change it at the places marked with a comment so that it also places the black pieces on the board.

Run the program and make sure it works correctly before proceeding to the next step.

9.2 Method isValidNonJump

At the moment, the body of this method simply returns false, so no move is considered valid. Statement-comments have been written, indicating the sequence of actions to be done in the body. Write the method body as indicated. You can check your code out by placing the statement System.out.println("move to " + newSq + "valid"); just before returning true and a similar statement just before returning false. Make sure this method is correct before proceeding.

9.3 Part I of Method validMove

Fill in the missing code that is marked with comments as Part I in this method. The comment specifies exactly what is to be done; most of what you write will be calls on methods in class *CheckersSquare*. When this is done correctly, your program should make non-jump moves correctly. Make sure this method is correct before proceeding.

9.4 Method isValidJump

At the moment, the body of this method simply returns false, so no move is considered valid. Statement-comments have been written, indicating the sequence of actions to be done in the body. Write the method body as indicated. You can check your code out by placing the statement System.out.println("move to " + newSq + "valid"); just before returning true and a similar statement just before returning false. Make sure this method is correct before proceeding.

9.5 Part II of Method validMove

Fill in the missing code that is marked with comments as Part II in this method. This is similar to what you did in Part I for this method. When this is done, your program is finished!

Once this code is written, the program should run correctly.