CSE 143, Autumn 2013
Programming Assignment #1: Tiles (40 points)
Due Thursday, October 3, 2013, 11:30 PM

thanks to Mike Clancy of UC Berkeley for the original idea of this nifty assignment!

This project will give you insights into how operating systems manage multiple programs' windows. Its implementation focuses on using the `ArrayList` collection class and basic object-oriented programming. Turn in a file named `TileManager.java` online using the turnin link on the Homework section of the course web site. You will need the support files `DrawingPanel.java`, `Tile.java`, and `TileMain.java` from the Homework section of the course web site; place these in the same folder as your program or project. (If you use Eclipse, you may need to put these files in the `src/` subdirectory of your project if such a subdirectory exists.)
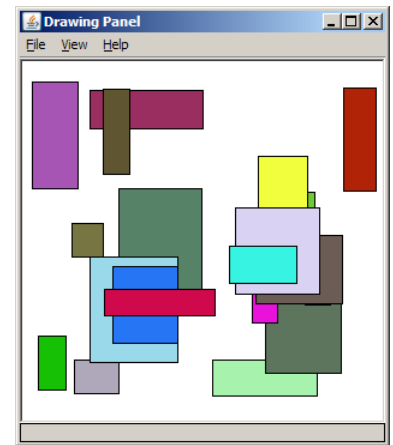
You should not modify the provided files. The code you submit must work properly with their unmodified versions.
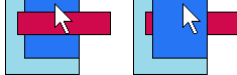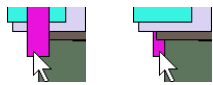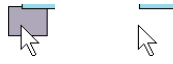
Program Description:

In this assignment you will write the logic for a graphical program that allows the user to click on rectangular tiles. You will not write any graphical code since it is provided for you in `DrawingPanel.java`.

The main client program you should run is the provided `TileMain` class. When it runs, it will create a graphical window on the screen; this is an object of type `DrawingPanel`. The panel displays a list of tiles. (Initially the panel shows 20 tiles, but you can add more by typing N.) Each tile is represented as an object of the provided class `Tile`. Each tile's position, size, and color are randomly generated by `TileMain`. You will write the `TileManager` to store and maintain a list of `Tile` objects.

The order of `Tile` objects stored in the `TileManager`'s list determines tiles' drawing order. For example, consider the tall tile that overlaps the wide tile in the upper left corner of the screenshot at right. The two tiles occupy some of the same (x, y) pixels in the window but the wide one was drawn first because it occurred before the tall one in the list. When the tall tile was drawn later, it covered part of the wide tile. The list's ordering is called the 3-dimensional ordering or *z-ordering*.

The graphical user interface ("GUI") displays the tiles and allows the user to manipulate them. Depending on the kind of input, different actions occurs:

- If the user clicks the **left mouse button** while the mouse cursor points at a tile, that tile is moved to the very *top* of the z-ordering (the end of the tile list).

- If the user clicks the **left mouse button** and holds down the **Shift key** while the mouse cursor points at a tile, that tile is moved to the very *bottom* of the z-ordering (the start of the tile list).

- If the user clicks the **right mouse button** while the mouse cursor is pointing at a tile, that tile is removed from the tile list and disappears from the screen.

- If the user clicks the **right mouse button** and holds the **Shift key**, *all tiles that occupy that pixel* are removed from the tile list and disappear from the screen.

- If the user types the **N key** on the keyboard, a new randomly positioned tile is created and added to the screen.

- If the user types the **S key** on the keyboard, the tiles' order and location are randomly rearranged (*shuffled*).

If you use a Mac with a 1-button mouse, you can simulate a right-click with a Ctrl-click (or a two-finger tap on the touch pad on a Mac laptop). If there is no tile where the user clicks, nothing happens.

If the user clicks a pixel that is occupied by more than one tile, the top-most of these tiles is used. (Except if the user did a Shift-right-click, in which case it deletes all tiles touching that pixel, not just the top one.)

Note that **your code does not need to directly detect mouse clicks or key presses**. Code in `TileMain.java` detects any user input. When user input is detected, methods that you define in `TileManager.java` are called, as described on the next page.

Implementation Details:

Your `TileManager` class should store a list of tiles as a **field of type `ArrayList`**. The various methods listed below will cause changes to the contents of that list. Remember to `import java.util.*;` to use `ArrayList`.

The following sections describe in detail each method you must implement in your `TileManager` class. For any methods that accept objects as parameters, you may assume that the value passed is not `null` or otherwise invalid.

---

public **TileManager**()

This constructor is called every time a new tile manager object is created. Initially your manager is not storing any tiles.

---

public void **addTile**(Tile rect)

In this method you should add the given tile to the end of your tile manager's list of tiles.

---

public void **drawAll**(Graphics g)

This method should cause all of the tiles in the tile manager to draw themselves on the screen using the given graphical pen. You do not need to do this yourself directly by calling methods on the `Graphics` object; each `Tile` object already has a `draw` method that it can use to draw itself. Draw the tiles from bottom (start) to top (end) of your manager's list.

Recall that in order to refer to type `Graphics`, you must `import java.awt.*;` in your code.

The next four methods are called by the graphical user interface ("GUI") in response to various mouse clicks, passing you the x/y coordinates where the user clicked. If the coordinates passed do not touch any tiles, no action or error should occur. After any click, the GUI will clear the screen for you and call `drawAll` to re-draw all of the tiles in your list.

---

public void **raise**(int x, int y)

Called when the user left-clicks. It passes you the x/y coordinates the user clicked. If these coordinates touch any tiles, you should move the topmost of these tiles to the very top (end) of the list.

---

public void **lower**(int x, int y)

Called when the user Shift-left-clicks. If these coordinates touch any tiles, you should move the topmost of these tiles to the very bottom (beginning) of the list.

---

public void **delete**(int x, int y)

Called when the user right-clicks. If these coordinates touch any tiles, you should delete the topmost of these tiles from the list.

---

public void **deleteAll**(int x, int y)

Called when the user Shift-right-clicks. If these coordinates touch any tiles, you should delete *all* such tiles from the list.

---

public void **shuffle**(int width, int height)

Called when the user types S. This method should perform *two actions*: (1) reordering the tiles in the list into a random order, and (2) moving every tile on the screen to a new random x/y pixel position. The random position should be such that the square's top-left x/y position is non-negative and also such that every pixel of the tile is within the passed width and height. For example, if the width passed is 300 and the height is 200, a tile of size 20x20 must be moved to a random position such that its top-left x/y position is between (0, 0) and (280, 180).

You can use the built-in Java method `Collections.shuffle` to randomly rearrange the elements of your list (1).

Your manager's list stores `Tile` objects. Each `Tile` object has the following public methods:

---

public int getX(), public int getY(), public void setX(int x), public void setY(int y)

These methods return or modify the tile's top-left x/y pixel position.

---

public int getWidth(), public int getHeight()

These methods return the tile's width and height in pixels.

---

public void draw(Graphics g)

Draws the tile on the screen using the given graphical pen.

---

public String toString()

Returns a text representation of the tile, such as `"(x=57,y=148,w=26,h=53)"`.
It can be useful to print a tile (or a collection of tiles) with `println` to examine their state.

Development Strategy and Hints:

Developing your code in stages and knowing how to test your solutions will be critical to your success on 143 assignments. One of the most important techniques for professional developers is to write code in stages ("**iterative enhancement**" or "**stepwise refinement**") rather than trying to do it all at once. This includes testing for correctness at each stage before moving to the next one. The next few paragraphs contain a detailed development plan. Study it carefully and think about why we suggest the plan we do.

Start by writing empty "**stub**" versions of all the required methods so that the `TileManager` class will compile.

We suggest that you write your constructor and `addTile` first, then `drawAll`. You can then run the program to make sure that you can see the tiles appear on the screen. Add click-related methods one at a time and test each one individually to be sure it works before moving on to the next.

One part of this program involves figuring out which tile(s), if any, touch a given x/y pixel. You can figure this out by comparing the x/y position of the click to the x/y area covered by the tile. For example, if a tile has a top-left corner of (x=20, y=10), a width of 50, and a height of 15, it touches all of the pixels from (20, 10) through (69, 24) inclusive. Such a tile contains the point (32, 17) because 32 is between 20 and 69 and 17 is between 10 and 24.

If you see compiler errors about class `Tile`, you may not have downloaded `Tile.java` or added it to your project.

If you have bugs or exceptions in your code, there are several things you can try. You can print out the state of your list and of each `Tile` object with temporary **`println` statements**. (Though this is a graphical program, `println` output does still appear on the console.) You should remove any such `println` statements before you turn in the assignment. You can also use a **debugger** as found in jGRASP or Eclipse to pause the code at a breakpoint in the middle of an operation. Stepping through line-by-line can help you to see what has gone wrong.

A **sample solution** is available on the course web site. You can use it to verify your program's behavior.

Style Guidelines and Grading:

A major focus of our style grading is **redundancy**. As much as possible, avoid redundancy and repeated logic in your code. One powerful way to avoid redundancy is to create "helper" method(s) to capture repeated code. It is legal to have additional methods in your `TileManager` class beyond those specified here. For example, you may find that multiple methods in your class do similar things. If so, you should create helper method(s) to capture the common code. (You should declare such methods to be `private` rather than `public`, so that outside code cannot call them.)

Your tile manager should maintain its list of tiles internally in a field of type `ArrayList` as stated previously. You should not use any other data structures. Your code should use "generic" type parameters appropriately when creating any collection objects. You must declare such collections with a suitable element type within <> brackets. You should use the array list and its methods appropriately, and take advantage of its ability to "shift" elements as they are added or removed. Your list should not store any invalid or `null` elements as a result of any mouse click activity.

Properly **encapsulate** your objects by making any data fields in your class `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

You should follow good general Java style guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as methods, loops, and factoring common code out of `if/else` statements; properly using indentation, good variable names, and types; and not having any lines of code longer than 100 characters in length. (If you have any such lines, split them into two or more lines using a line break.)

You should **comment** your code with a heading at the top of your class with your name, section, and a description of the overall program. Also place a comment heading on top of each method, and a comment on any complex sections of your code. Comment headings should use descriptive complete sentences and should be written *in your own words*, explaining each method's behavior, parameters, return values, and assumptions made by your code, as appropriate.

Unless otherwise specified, your solution should use only material taught in class and in the book chapters covered so far.

For reference, our solution to this program is around **90 lines long** including comments (and has **42 "substantive" lines** according to our Indenter tool on the course web site). But you do not have to match this; it's just listed as a sanity check.