Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden
Christoph Daniel Schulze, Steven Smyth

Institut für Informatik

8. Übung zur Vorlesung „Programmierung“

After Eight: Heaps and Stacks for Christ-Chess
Abgabe am Sonntag, 08. Januar 2017 - 23:55

This is the Christmas set of tasks! The deadline for this task is 08.01.2017. Enjoy your holidays!

Even so task 3 is not mandatory, we highly encourage you to try to solve this task, because the pattern used in this task is a good preparation for the programming project at the end of the semester.

As the tests for this set are not completely finished at the moment, we have deactivated the tests for this weekend. We expect them to be operational at the beginning of the next week, which should still give you plenty of time to polish your solutions before the deadline. Make sure that you re-run the tests if you finished a task before that. Happy programming!

Aufgabe 1 - Mandatory Memory Trees

49 Punkte

This problem should help deepen your understanding of

- Recursion,
- Proper usage of local/instance/class variables,
- Memory usage of heap and stack,
- Hexadecimal numbers, and
- Character arithmetic.

In class, we illustrated recursion with a little program that drew trees. For your convenience, the code is repeated here:

```
import java.awt.*;
import acm.program.*;
import acm.graphics.*;

public class Tree extends GraphicsProgram {
  public void run() {
    setSize(500, 350);
    drawTree(250, 350, 100, 90);   // ADJUST "TREE PARAMETERS" HERE
  }

  public void drawTree(double x0, double y0,
      double len, double angle) {

    if (len > 2) {
      double x1 = x0 + len * GMath.cosDegrees(angle);
      double y1 = y0 - len * GMath.sinDegrees(angle);
      add(new GLine(x0, y0, x1, y1));
      drawTree(x1, y1, len * 0.75, angle + 30);
      drawTree(x1, y1, len * 0.66, angle - 50);
    }
  }
}
```

Part a)

Name your class `TreeHeapStackTrace` (instead of only naming it just `Tree`). Your task is to instrument this code such that it keeps track of heap and stack usage. For example, if the initial call to `drawTree` would be like this:

```
drawTree(250, 350, 3.5, 90);
```

Then the output should be exactly like this:

```
Create drawTree() stack frame at address 0xfffe4, depth 1
Create GLine object #1 at address 0x100000
Create drawTree() stack frame at address 0xffffc8, depth 2
Create GLine object #2 at address 0x100014
Create drawTree() stack frame at address 0xfffac, depth 3
Delete stack frame at address 0xfffac, depth 3
Create drawTree() stack frame at address 0xfffac, depth 3
Delete stack frame at address 0xfffac, depth 3
Delete stack frame at address 0xffffc8, depth 2
Create drawTree() stack frame at address 0xffffc8, depth 2
Create GLine object #3 at address 0x100028
Create drawTree() stack frame at address 0xfffac, depth 3
Delete stack frame at address 0xfffac, depth 3
Create drawTree() stack frame at address 0xfffac, depth 3
Delete stack frame at address 0xfffac, depth 3
Delete stack frame at address 0xffffc8, depth 2
Delete stack frame at address 0xfffe4, depth 1

HEAP:
Created 3 GLine objects,
requiring 60 bytes of heap space,
from address 0x100000 to 0x10003b.

STACK:
Created and discarded 7 drawTree() stack frames,
with maximal depth 3,
requiring 84 bytes of stack space,
from address 0xfffac to 0xffffff.
```

As you can tell, this is somewhat simplified, as it only reflects the calls to `drawTree` and not any other method calls. It also uses hypothetical addresses and object/stack frame sizes, which you should deduce from the trace given above. Again, please make sure your output matches exactly the above, also using the same numbers.

As another point of comparison, if the initial call to `drawTree` is as in the original code, the final statistics of the trace would look like this:

```
HEAP:
Created 3373 GLine objects,
requiring 67460 bytes of heap space,
from address 0x100000 to 0x110783.

STACK:
Created and discarded 6747 drawTree() stack frames,
with maximal depth 15,
requiring 420 bytes of stack space,
from address 0xfffe5c to 0xffffff.
```

To print the addresses in hex code, you should implement **your own `toHexString` method** that takes a non-negative integer n and returns a string that contains the hexadecimal representation of n, with leading "0x" and lower-case letters, as in the examples above. E.g., `toHexString(42)` will return the string `"0x2a"`.

Note: the automatic test will not call the `run` method, as that method does not take any parameters, but it will call the `drawTree` method, with varying parameters. Therefore, to make your program pass the automatic test, **all output, including the final statistics, should be generated by the `drawTree` method or methods called thereby.** That is, the `run` method itself should not produce any output. Furthermore, the test will call `drawTree` multiple times with different parameters. Therefore, we recommend that you use class variables only for constants. We also recommend that you test your implementation by adding multiple calls `drawTree` to your `run` method. Remember to call your class `TreeHeapStackTrace`.

Part b)

As the traces indicate, we allocate about twice as many stack frames as we create `GLine` objects. How would you have to change the code such that we only allocate as many stack frames as there will be GLine objects? Explain your approach in a comment at the beginning of your uploaded source code.

Part c)

Explain the different parts of memory (stack, heap, static segment) and which data are stored in them. List examples from your solution in Part a).

Also, explain the following terms:

- Recursion
- Mutable class
- Wrapper class
- Boxing/unboxing
- Garbage collection

Aufgabe 2 - Mandatory Linked Items

49 Punkte

During the lecture, you saw how you can link objects together. In this assignment, we will use this concept to link objects together that hold an arbitrary value. If we save the first such object (the *head element*), we have effectively built what is usually called a *linked list*. Starting at the head element, we can assign each item in the linked list a 0-based index: the head element will have index 0, the next element will have index 1, and so on. The end of the list is reached once an element doesn't link to a next element anymore. Once you've finished, you should be able to use your class as follows:

```
LinkedElement<String> headElement = new LinkedElement<String>("Yo!");

headElement.add("some");
headElement.add("women");
headElement.add("just");
headElement.add("want");
headElement.add("to");
headElement.add("watch");
headElement.add("the");
headElement.add("world");
headElement.add("burn");

println(headElement.get(7));      // prints "the"
headElement = headElement.remove(0);
headElement.set(1, "men");
println(headElement.get(1));      // prints "men"
```

Note that each element can be understood as the beginning of a linked list.

Start by writing a class `LinkedElement<T>` in a package called `programming.set8.links`. `T` will be a placeholder for the data type of the value the element can hold. Add two private instance variables: one should hold the value the element holds, and the other should be a link to the next element, if any. Add a constructor that takes a parameter of type `T` that will become the value of the new linked element. Then, add the following methods:

```
/**
 * Returns the value of the i-th linked element.
 *
 * @param i
 *            0-based index of the element whose value to return.
 * @return the i-th element's value, or {@code null}if there is no element with that index.
 */
public T get(int i) {
    // Implement me!
}

/**
 * Sets the value of the i-th linked element to the given value. If there is no i-th linked
 * element, nothing happens.
 *
 * @param i
 *            0-based index of the element whose value to return.
 * @param newVal
 *            the new value to set.
 */
public void set(int i, T newVal) {
    // Implement me!
}

/**
 * Returns the index of the first occurrence of a linked element carrying the given value in
 * the list.
 *
 * @param val
 *            the value to search for.
 * @return index where the value was found, or -1 if it's not in any of the linked elements.
 */
public int firstIndexOf(T val) {
    // Implement me!
}

/**
 * Adds a new linked element holding the given value at the end of the linked elements.
 *
 * @param newVal
 *            the new value.
 */
public void add(T newVal) {
    // Implement me!
}

/**
```

```
 * Removes the i-th element from the linked elements. If {@code i == 0}, this will effectively
 * remove the head element. Thus, this method returns the linked element that is the new head
 * element.
 *
 * @param i
 *            index of the element to remove.
 * @return the new head element.
 */
public LinkedElement<T> remove(int i) {
    // Implement me!
}
```

Hints:

- It is probably easiest to implement all methods recursively. Think about what the trivial case is. Then think about what you can do if it is not.
- `remove(int)` is probably the most intricate method to implement. If an element is removed, you will have to think about what to do with the next element link of the previous element.

Aufgabe 3 - Christ-chess

1 Punkt

This task continues you endeavor to create a fully playable, self-programmed chess experience. You may continue with your solution or the example solution of task 2 set 4 or simply start from scratch.

In general it is often a good idea to separate your programm data from its representation (also known as view). Therefore, you should divide the state of the game from the graphical representation. A control class managing the whole scene could then look like this (but is not required to):

```
package programming.set8.christchess;

import acm.program.GraphicsProgram;

/**
 * This class controls a simplified chess game.
 */
@SuppressWarnings("serial")
public class Christchess extends GraphicsProgram {

    @Override
    public void run() {
        // Create the view using the programs canvas and
        // set the size appropriately.
        ChessView chessView = new ChessView(getGCanvas());
        setSize(getGCanvas().getWidth(), getGCanvas().getHeight());

        // Create the game data object and
        // initialize a game.
        ChessData chessData = new ChessData();
        chessData.initNewGame();
        chessView.init(chessData);

        // Main game loop
        do {
            ChessPiece piece = null;
            String moveStr;
```

```
            int x = -1;
            int y = -1;
            do {
                // Ask the user to select a piece.
                do {
                    String selectStr = readLine((chessData.getActivePlayer() == ChessPiece.PLAYER1 ? "
                    piece = chessData.getPieceAt(selectStr);
                } while (!chessData.isValidSelection(piece));

                // Display which squares are valid moves.
                chessView.updateValidMoves(chessData, piece);

                // Ask the user where to move the selected piece.
                do {
                    moveStr = readLine("Move " + piece + " to: ");
                    if (moveStr.equals("c")) break;
                    x = ChessData.stringToX(moveStr);
                    y = ChessData.stringToY(moveStr);
                } while (!chessData.isValidMove(piece, x, y));

                // Reset the visual for the valid moves.
                chessView.updateValidMoves(chessData, null);

            } while (moveStr.equals("c"));

            // Move the piece and check for captured pieces.
            ChessPiece capturedPiece = chessData.movePieceTo(piece, x, y);
            if (capturedPiece != null) {
                println("Capture: " + piece + " captures " + capturedPiece);
            }

            // Change the player and update the board.
            chessData.togglePlayer();
            chessView.update(chessData);

            // Repeat until a player is checkmate.
        } while (chessData.isCheckmate() == ChessPiece.NO_PLAYER);
        println("Player " +
                (chessData.isCheckmate() == ChessPiece.PLAYER1 ? "White" : "Black") +
                " is checkmate.");
    }

}
```

Using the classes you have to implement, the program will manage a complete (simplified) chess game. After selecting a chess piece, the view should display all valid moves. The user then may select the target square for the selected piece.

A console trace for this scenario could be as follows:

```
White, select a piece: e4
Move white rook to: c4
Capture: white rook captures black pawn
Black, select a piece:
```

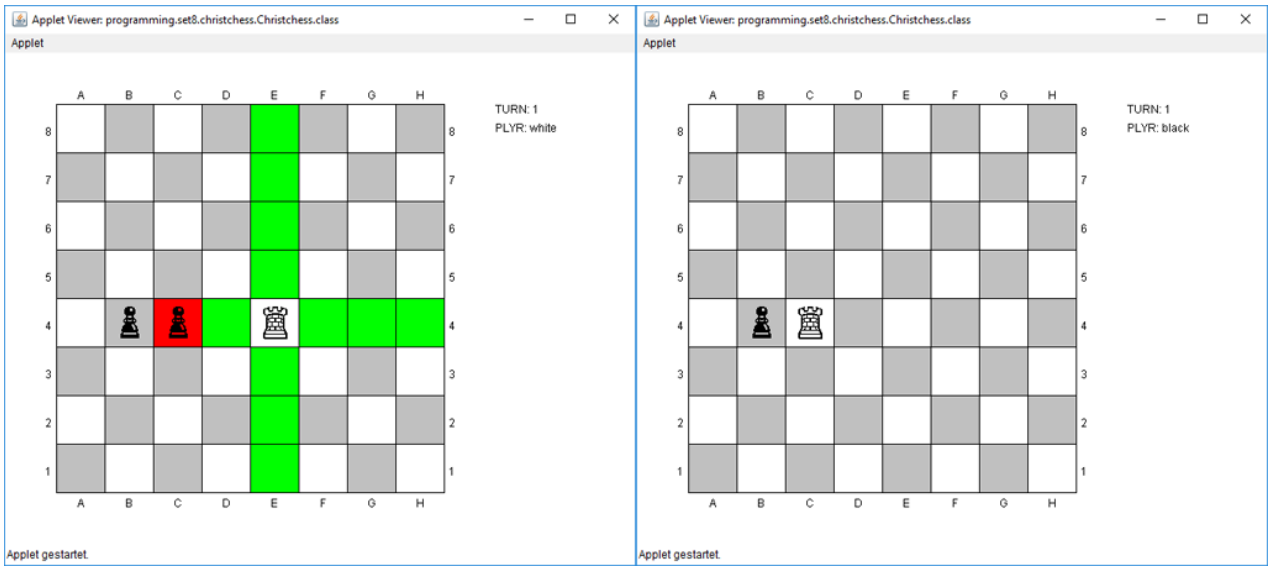

Abbildung 1:

Your task is to write at least three classes `ChessPiece`, `ChessData`, and `ChessView` that together with a main controlling class (like the one above) form a complete (simplified) chess game.

ChessPiece Write a class `ChessPiece` that can be used to represent a single chess piece. You can use an appropriate class hierarchy to model all your pieces with `ChessPiece` being the superclass. The minimal signature of `ChessPiece` looks like this:

- Constructor `public ChessPiece(int type, int player, int x, int y)` that creates a `ChessPiece` object.
- A method `public List<Point> getValidTargetSquares(ChessData data)` that retrieve a list of points that are valid target squares for this piece. This includes opponent's squares if that would be a valid figure capture. It is sufficient to include the standard moves for all the pieces. You don't have to include special moves that involve two pieces to change position at the same time. However, consider the right movement rules for pawns depending on their actual position. Use `java.awt.Point`.
- The getter methods `public int getX()`, `public int getY()`, `public int getPlayer()`, `public int getType()`.
- A method to move the piece to a specific square `public void moveTo(int x, int y)`, and
- override `public String toString()` to provide a user-friedly string output.

Use the following constants for the piece types and the players:

```
public static final int PAWN = 0;
public static final int KNIGHT = 1;
public static final int BISHOP = 2;
public static final int ROOK = 3;
public static final int QUEEN = 4;
public static final int KING = 5;

public static final int NO_PLAYER = 0;
public static final int PLAYER1 = 1;
public static final int PLAYER2 = 2;
```

ChessData Create a class `ChessData` that stores all information about the game state, meaning information on **what** pieces are **where** on the board. Additionally, the class must keep track of the game turns and which player is the active one. The `ChessData` class must provide the follwing methods:

- A `public void initNewGame()` method that initializes the game with the usual chess game layout.
- A method `public ChessPiece addNewPiece(int type, int player, int x, int y)` that adds a new specific piece to a specific position on the chess board and returns the newly created `ChessPiece`. You can use this method to create specific scenarios on the board. Particularily, you should not have to call `initNewGame()` if you just want to place specific chess pieces.
- A method `public void removeAllPieces()` that removes all pieces from the chessboard.
- The getter methods `public List<ChessPiece> getPieces()` for all pieces, `public int getActivePlayer()` for the active player, and `public int getTurn()` to retrieve the actual turn count.
- `public ChessPiece getPieceAt(int x, int y)` retrieves the `ChessPiece` at x and y. If there is no ChessPiece or the location is invalid, return **null**.
- `public ChessPiece getPieceAt(String str)` retrieves the `ChessPiece` at the string representation of the chess square. For example, **a1** would be (0,7) in x and y.
- To convert conveniently, write two static methods `public static int stringToY(String str)` and `public static int stringToX(String str)`.
- Method `public boolean isValidSelection(ChessPiece piece)` returns true if the `ChessPiece` is valid and the active player is allowed to move this piece.
- Similarily, `public boolean isValidMove(ChessPiece piece, int x, int y)` returns true if the target location is a valid location for that piece.
- Eventually, `public ChessPiece movePieceTo(ChessPiece piece, int x, int y)` actually moves that piece to the specified location. If there is an opponent's piece on that square, remove it from the board and return the captured `ChessPiece` object.
- `public void setActivePlayer(int player)` sets a new active player. Also add a convenient method `public void togglePlayer();` that switches from one player to the other.
- `public int isCheckmate()` returns the ID of the player whos king is checkmate. If there is no King on the board, the method returns `NO_PLAYER`.

`ChessData` is not allowed to draw on the canvas! In fact, the data class does not even know which kind of graphical representation is used.

ChessView Also, create a class `ChessView` that is responsible for drawing the chessboard and all the pieces according to the data stored in your data object. Additionally, you should display the actual number of turns and the player who's turn it is.

- Constructor `public ChessView(GCanvas gc)` expects a canvas to draw on. Check the controler above for an usage example.
- A method `public void init(ChessData data)` that is called to initialize a game. This could also happen when a game restarts. Hence, if there is already a game running, the method should discard any obsolete visual data.
- Method `public void update(ChessData data)` takes the actual game data and updates the graphical representation.
- Method `public void updateValidMoves(ChessData data, ChessPiece piece)` takes the actual game data and a `ChessPiece` to update the squares color to display valid moves as in the figure above. Use `Color.GREEN` for valid free squares and `Color.RED` if that square is occupied by an opoonent's piece. If the `piece` is **null** the default chessboard should be displayed.

`ChessView` is not allowed to change the game data! In fact, the `ChessView` is only responsible for displaying the actual game state.

Tips

- `List` is the superclass of `ArrayList`. If a method returns `List` you can create an `ArrayList` object as shown in the lecture.
- Implementing all classes at once could be hard to debug. It might be a good idea, to implement and test reasonable parts of this task one after another. For example, one approach could be to create a preliminary `ChessPiece` class and then test the link between your `ChessData` and `ChessView` classes. Only after this part works, implement the valid movement handling and display, and so on...
- Make sure that you implemented all methods before testing in your classes in the iLearn. The test will most likely fail if it is not able to find all methods as described above.

Submitting Your Solution

- The package of your chess classes must be `programming.set8.christchess`.

Most likely you structure your classes in different files. You can upload all your classes as an archive file. To turn your project into an archive file that you can upload, perform the following steps in Eclipse:

1. Right-click your project in the *Package Explorer* and click *Export*.
2. From the *Java* category, select *JAR file* and clickt *Next*.
3. Make sure your project's `src` folder is exported. Choose not to export generated class files and resources, but only **Java source files** and resources.
4. Choose where to save the file and click *Finish*.

Zusatzaufgabe 4 - Imagine Re-Imaged Images 1 Punkt

According to wikipedia *a digital watermark is called* ***perceptible*** *if its presence in the marked signal is noticeable (e.g. Digital On-screen Graphics like a Network Logo, Content Bug, Codes, Opaque images). On videos and images, some are made transparent/translucent for convience for people due to the fact that they block portion of the view.*

Write a class `Watermark` in a package `programming.set8.watermark`, which is a subclass of `GraphicsProgram`, that adds a watermark to a given `GImage`. We will give you some materials to work with:

- You may use the images of the press services: Use nina101w.jpg to satisfy the automatic test, but of course you may play with other images as well.
- Use the trollface as watermark. It is located here: trollface.jpg
- The watermark should be a semi-transparent overlay meaning that the original pixels should be (partly) visible through the watermark.
- The watermarked image should be scaled to the size of the application. Set the size of your application window to `WIDTH` and `HEIGHT`.
- The resulting image should not be transparent (which would allow other images/elements behind it to shine through). Therefore, it **must not** contain any pixels with an alpha value $\neq 255$.
- Blend the marked pixels with a **bitwise and** and `0x80ffffff` to satisfy the test. However, you are free to play with these values, too.

Background:

- In image processing, *image masks* play an important role. An image mask is an image that may be colored/structured in different ways to tell image processors what to do with specific pixels in another image. A simple mask is a black/white image mask. The areas in the *source image* that correspond to

the black and white areas in the mask image are treated differently. The black pixels of image mask mark the pixels in the source image that should represent the watermark, whereas the other pixels in the source image should stay unchanged.

The following constants may be useful... of course you may play around with the values...

```
/** Width of display area */
public static final int WIDTH = 640;
/** Height of the displaz area */
public static final int HEIGHT = 480;
/** Name of the image */
public static final String IMAGE_NAME = "nina101w.jpg";
/** Name of the watermark image */
public static final String WATERMARK_IMAGE_NAME = "trollface.jpg";
/** Pixel blend mask value */
public static final int PIXEL_BLEND_MASK = 0x80ffffff;
```

Implement the methods `createScaledGImage`, `createWatermarkGImage`, and `convertARGBtoRGB`.

```
/**
 * Creates a copy of a given GImage with new proportions.
 *
 * @param image
 *              the source image
 * @param width
 *              the width of the new image
 * @param height
 *              the height of the new image
 * @return the new GImage object.
 */
public GImage createScaledGImage(GImage image, double width, double height) {
  // Implement me!
}

/**
 * Creates a copy of a given GImage and adds a watermark to it.
 *
 * @param image
 *              the source image
 * @param watermark
 *              the image that's used as watermark
 * @return the new GImage including the watermark.
 */
public GImage createWatermarkGImage(GImage image, GImage watermark) {
  // Implement me!
}

/**
 * Converts a color value of the ARGB domain to a color value of the RGB domain.
 * The alpha channel is removed via normal blending mode (assuming a white background).
 * ( new = old  alpha + 255.0  (1.0  alpha ) )
 *
 * @param argbValue
 *          the source color value
 * @return the color in the RGB domain.
```

```
 */
public static int convertARGBtoRGB(int argbValue) {
  // Implement me!
}
```

Tips:

- Use `getPixelArray()` to retrieve a 2D-Array of the image.
- `pixelArray.length` will give you the height of the pixel field and `pixelArray[0].length` the width. Why is that?
- In *normal blend mode* a color channel is blent according to the formula $new = old * alpha + 255.0 * (1.0 - alpha)$ with *alpha* between 0.0 and 1.0. E.g. `double newRed = oldRed * alpha + 255.0 * (1.0 - alpha);`

Reference Image:

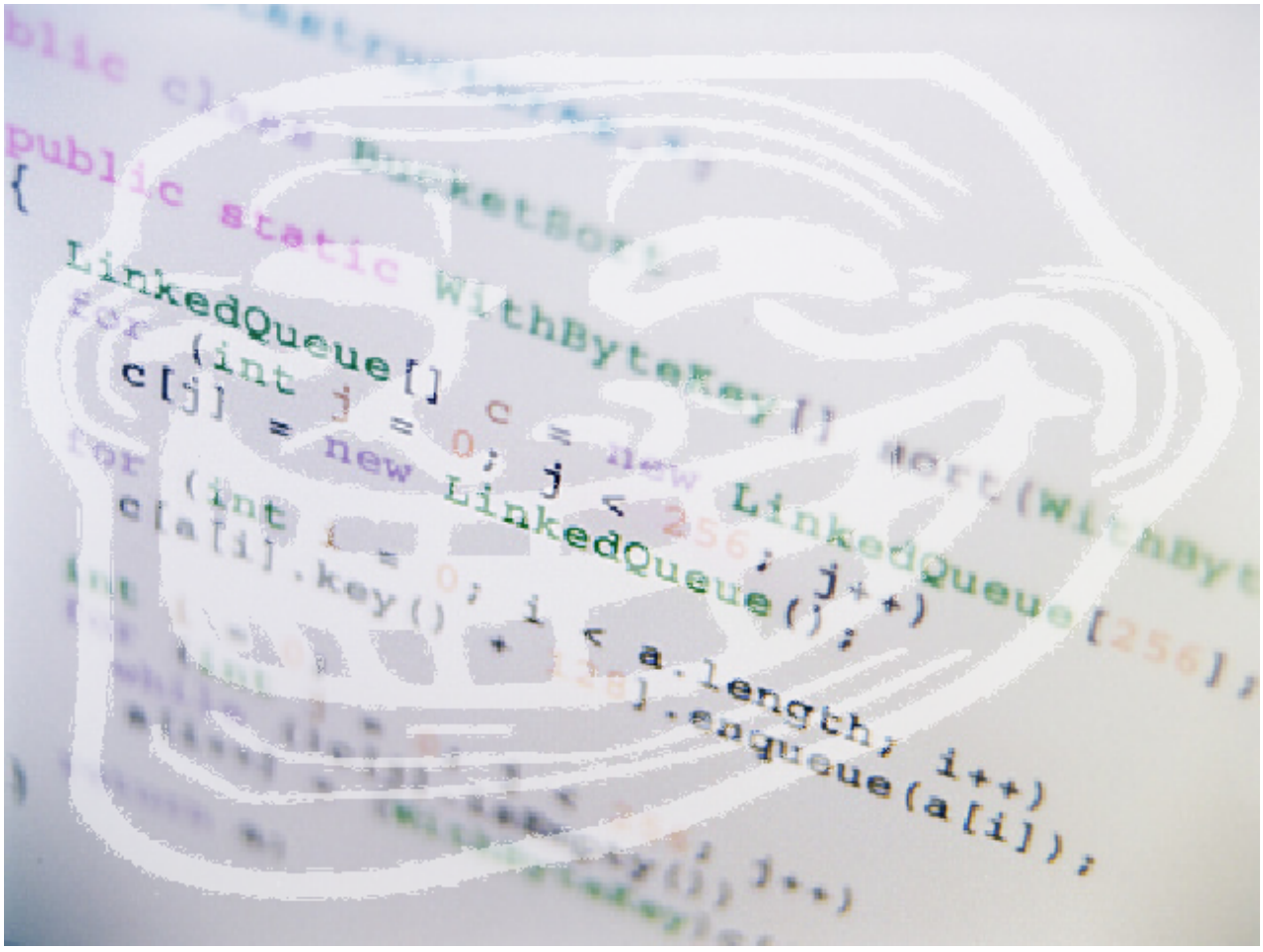

Abbildung 2: Target: watermarked nina101w