CIS 1200 Final Exam December 19, 2023

Steve Zdancewic and Swapneel Sheth, instructors

SOLUTIONS

1. **OCaml and Java Concepts** (21 points total)

   (a) (12 points) (1.5 points each) Indicate whether the following statements are true or false.

   (a) True □ False ☒
   In Java, if `s.equals(t)` returns **`true`**, then `s == t` is guaranteed to return **`true`**.

   (b) True □ False ☒
   In Java, if `s == t` returns **`true`**, then `s.equals(t)` is guaranteed to return **`true`**.

   (c) True ☒ False □
   In our GUI library, an `event_listener` is a first-class function stored in the hidden state of a notifier widget. When an event occurs in the widget, the notifier invokes all of the stored `event_listener`s.

   (d) True ☒ False □
   Variables in OCaml are immutable by default, whereas in Java, variables are mutable by default.

   (e) True □ False ☒
   In a Java **`try`**–**`catch`**–**`finally`** statement, the **`finally`** clause is executed only if the **`try`** clause does not throw an exception.

   (f) True ☒ False □
   In Java, a `final` instance variable can only be changed in a constructor.

   (g) True □ False ☒
   In Java, every method must declare every exception it might throw in its header.

   (h) True □ False ☒
   In Java, the default implementation of equality in the `Object` class uses reference equality for mutable objects and structural equality for immutable objects.

(b) For the following questions, select the most suitable data structure for each particular use case and justify why.

(Hint: An `ArrayList` in Java is similar to the "Resizable Array" example from class.)

i. (3 points) You are developing a navigation application (similar to Google Maps) where a route consists of a series of way-points. You will need to frequently insert and remove way-points as the route is optimized in real time. Which collection is best suited for this?

☐ `ArrayList` ☒ `LinkedList` ☐ `TreeMap`

Reason:
A `LinkedList` *is best suited since we can efficiently insert and remove using the head and tail of the list.*

ii. (3 points) You're designing a system for an online bookstore (similar to Amazon) to store books where they need to be sorted based on ISBN numbers and frequently retrieved (using the ISBN numbers). Which collection would you choose?

☐ `ArrayList` ☐ `LinkedList` ☒ `TreeMap`

Reason:
A `TreeMap` *is best suited since we can provide override the* `compareTo` *method for books and use the efficient lookup and insert functions for Binary Search Trees.*

iii. (3 points) You're designing a system for a music playing app (similar to Spotify) that stores curated playlists and once these playlists are created, they are never updated. Users can choose to play songs in any order they like (e.g., song 1 followed by song 3 followed by song 10). Which collection would you choose?

☒ `ArrayList` ☐ `LinkedList` ☐ `TreeMap`

Reason:
Since we want efficient access to the middle of the playlist and order in which the songs are inserted matters, an `ArrayList` *is the best option.*

2. **OCaml Higher Order Functions (again)** (16 points total)

Recall the higher-order list processing functions shown in Appendix 1.

For each of the following mystery functions, determine which implementation choice(s) will produce the correct output for the provided `input` list (Choose all that apply).

(a) (4 points)

```
mystery1 [1; 4; 7] = [2; 1; 5; 4; 8; 7]
```

☒ `fold (fun x acc -> (x+1)::x::acc) [] input`

☐ `transform (fun x-> x+1::[x]) input`

☐ `fold (fun x acc -> x+1@x@acc) [] input`

☐ `transform (fun x -> (x, x+1)) input`

(b) (4 points)

```
mystery2 [("I", "love"); ("dogs", "and"); ("cats", "!")] =
  ["I love"; "dogs and"; "cats !"]
```

☐ `fold (fun (a,b) acc -> a^" "^b) [] input`

☒ `transform (fun (a,b) -> a^" "^b) input`

☒ `fold (fun (a,b) acc -> (a^" "^b)::acc) [] input`

☐ `transform (fun (a,b) -> (a^" "^b)::[]) input`

(c) (4 points)

```
mystery3 [2; 3; 4] = 1
```

☒ `fold (fun x acc -> acc / x) 24 input`

☐ `transform (fun x -> 1) input`

☒
```
fold (fun x acc ->
  if ((x mod 3 = 0) || (acc = 1)) then 1 else 0) 0 input
```

☐ `transform (fun x -> if x = 2 then 1 else 0) input`

(d) (4 points)

```
mystery4 [[1; 2; 3]; [-1; -1; -1]; [2; 2; 2]] = [6; -1; 8])
```

☒ `transform (fun x -> (fold (fun y acc -> y * acc) 1 x)) input`

☐ `transform (fun x -> (transform (fun y -> y * y) x)) input`

☐ `fold (fun x1 acc -> (transform (fun x2 -> x1*x2::acc))) [] input`

☒
```
fold (fun x acc1 ->
  (fold (fun y acc2 -> y * acc2) 1 x)::acc1) [] input
```

3. **Java Subtyping and Dynamic Dispatch** (19 points total)

This problem refers to two interfaces and several classes that might appear in a program about the movie "Barbie". You can find them in Appendix 2.

(a) (2 points) Which of the following classes are an example of simple inheritance in Java (either explicitly or implicitly)? (Mark all that apply.)

☒ `Allan` ☒ `ComputerScienceBarbie` ☐ `JavaBarbie`
☐ `OcamlBarbie`

(b) (2.5 points)

```
__________________ barbie = new ComputerScienceBarbie("CIS", "Hoodie");
```

Which type can be correctly used for the declaration of `barbie` above?
(Mark all that apply.)

☒ `MattelToy`
☒ `Barbie`
☐ `Allan`
☒ `ComputerScienceBarbie`
☐ `JavaBarbie`
☐ `OcamlBarbie`
☒ `Object`

(c) (2.5 points)

```
MattelToy barbie = new _______??_________(...);
OCamlBarbie toy = (OCamlBarbie) barbie;
toy.debugOCamlCode("'a list");
```

What can be used on the first line (instead of the `???`) so that the code successfully compiles *but throws an exception when run*? (Choose all that apply.)

☐ `MattelToy`
☐ `Barbie`
☐ `Allan`
☒ `ComputerScienceBarbie`
☒ `JavaBarbie`
☐ `OcamlBarbie`
☐ `Object`

Which of the following lines is legal Java code that will not cause any compile-time (i.e., type checking) or run-time errors?

If it is legal code, check the "Legal Code" box and answer the questions that follow it. If it is not legal, check one of the "Not Legal" options and explain why.

You can assume each option below is independent.

(d) (3 points)

```
Barbie barbie = new ComputerScienceBarbie("Natalie", "Blazer");
String result = barbie.code();
```

☐ Legal Code

A. The static type of `barbie` is _____.

B. The dynamic class of `barbie` is _____.

☐ Not Legal — Will compile, but will throw an `Exception` when run

☒ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

The `code` *method is not defined for the static type* `Barbie`.

(e) (3 points)

```
MattelToy toy = new OCamlBarbie("OCaml", "Caml");
toy.manufacture();
```

☒ Legal Code

A. The static type of `toy` is `MattelToy`.

B. The dynamic class of `toy` is `OcamlBarbie`.

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(f) (3 points)

```
ComputerScienceBarbie barbie = new JavaBarbie("Java", "Coffee cup");
System.out.println(barbie.speak());
```

The code above is Legal and doesn't throw an exception when run. Now, consider changing the second line to the following:

```
System.out.println(((JavaBarbie) barbie).speak());
```

The new version is:

☒ Legal Code

The updated code above will print (Choose all that apply.)

- ☐ "Hi Computer Science Barbie"
- ☒ "Hi Java Barbie"
- ☐ "Hi OCaml Barbie"

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(g) (3 points)

```
List<ComputerScienceBarbie> list = new ArrayList<JavaBarbie>();
```

☐ Legal Code

The code above is legal because: (Choose all that apply.)

- ☐ `JavaBarbie` is a subtype of `ComputerScienceBarbie`
- ☐ This example illustrates Parametric Polymorphism.

☐ Not Legal — Will compile, but will throw an `Exception` when run

☒ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

Even though `JavaBarbie` *is a subtype of* `ComputerScienceBarbie`*, Java Generics are invariant.*

Grading scheme: For (a), +0.5 for each option correctly answered

Grading scheme: For (b, c), , +0.5 if 6-7 incorrectly answered OR +1 if 4-5 incorrectly answered OR +1.5 if 2-3 incorrectly answered OR +2 if 1 incorrectly answered OR +2.5 for all correctly answered

Grading scheme: For Illegal Code (d, g), +1.5 Selected correct option AND (+0.5 for partially correct reason OR +1.5 for completely correct reason)

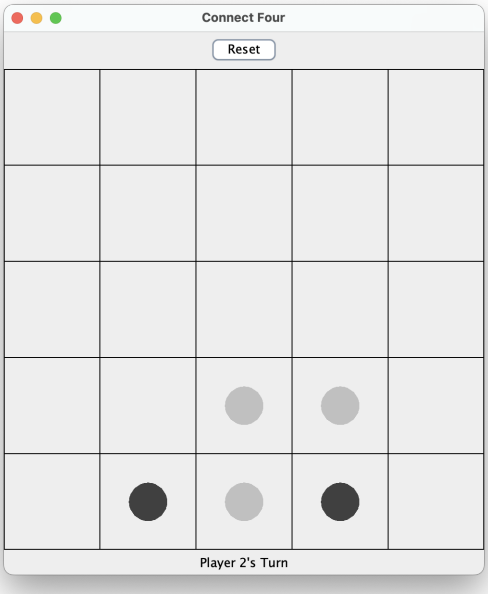
Grading scheme: For (e), +1.5 Selected correct option AND +0.5 for static type correctly answered and +1 for dynamic class correctly answered

Grading scheme: For (f), +1.5 Selected correct option AND +0.5 for each print option correctly answered

4. **Java Swing Programming** (16 points total)

Appendix 3 shows code for a partial implementation of a "Connect Four" game in Java.

The image below shows the GUI after a few turns. The exact rules of the game are not relevant to the questions below; the questions test your understanding of both Java and Swing programming idioms instead.



(a) (2 points)

True ☒ False ☐

On line 44, calling `super.paintComponent(g)` invokes the `paintComponent` method of the `JPanel` class.

(b) (2 points)

There are several occurrences of the **`new`** keyword in the code provided. List all occurrences (i.e., line numbers) that correspond to anonymous inner classes.

Just one instance – lines 15–25.

_____

_____

(c) (4 points)

How will the program's behavior change if we delete the call to `repaint()` on line 31? (Select one.)

- ☐ Nothing at all will ever be displayed – just a blank window.
- ☐ The initial GUI will be displayed, but no shapes will ever be drawn.
- ☐ The initial GUI will be displayed and tiles will show up when the grid is selected, but clicking the reset button will have no effect.
- ☐ The initial GUI will be displayed and tiles will show up when the grid spots are selected. Clicking the reset button will not clear the existing tiles and no further GUI updates will happen (regardless of what the user does).
- ☒ The initial GUI will be displayed and tiles will show up when the grid spots are selected. Clicking the reset button will not clear the existing tiles, but clicking on a grid spot will update the UI to the correct state.
- ☐ No change in behavior.

(d) (4 points) For the game overall, which of the following are true? (Mark all that apply.)

- ☐ `GameBoard` is a supertype of `JPanel`.
- ☒ The code would still compile successfully and behave identically if we replace the Lambda function on lines 74–97 with an analogous anonymous inner class.
- ☒ The `@Override` annotations (such as on lines 16, 42, and 65) are optional and the code will successfully compile if we delete them.
- ☐ Java requires the use of the `MouseAdapter` class whenever we want to add a `MouseListener` to an object.

(e) (4 points) Consider the argument to `reset.addActionListener(...)` on line 90. Which of the following are true?

The Javadocs for `addActionListener` are shown below:

```
public void addActionListener(ActionListener l)
Adds an ActionListener to the button.

Parameters:
l - the ActionListener to be added
```

(Select one.)

- ☐ The static type of the argument is **`null`**.
- ☐ The static type of the argument is `Object`.
- ☒ The static type of the argument is `ActionListener`.
- ☐ The argument has a static type, but it's not one of the options listed above.
- ☐ The argument does not have a static type.

(Select one.)

- ☐ The dynamic class of the argument is **`null`**.
- ☐ The dynamic class of the argument is `Object`.
- ☐ The dynamic class of the argument is `ActionListener`.
- ☒ The argument has a dynamic class, but it's not one of the options listed above.
- ☐ The argument does not have a dynamic class.

5. **Java Exceptions** (12 points total)

This problem makes use of the class definitions for a game called "Hot Potato," which is provided for you in Appendix 4.

(a) (5 points)

The `getsIt` method has a **`throws`** declaration in the method signature (line 14). Which of the following are true? (Choose all that apply.)

- ☐ It is possible to omit the **`throws`** declaration (*without* making any other changes) and the code would still compile successfully.
- ☒ It is possible to omit the **`throws`** declaration (*with* making other changes) and the code would still compile successfully.
- ☐ Any code that calls the `getsIt` method must also have a **`throws`** declaration in the method signature.
- ☐ Any code that calls the `getsIt` method doesn't need to have a **`throws`** declaration in the method signature since `HotPotato` is an *unchecked* exception.
- ☒ Any code that calls the `getsIt` method either needs a **`throws`** declaration in the method signature or handles the exception via a **`try`**–**`catch`** block since `HotPotato` is a *checked* exception.

(b) (7 points)

What happens when the following code is run? (Select one.)

```
String n1 = "Player 1";
String n2 = "Player 2";
String n3 = "Player 3";
HotPotato potato = new HotPotato();
Player p1 = new Player(n1, null);

Player p2 = new Player(n2, null);
Player p3 = new Player(n3, p2);
p1.nextInLine = p3;
p3.nextInLine.nextInLine = p1;

Player.thrower = p3;
try {
    p2.getsIt(potato);
} catch (HotPotato p) {
    System.out.println("Dropped it!");
}
```

(Answer choices on the next page.)

☐ Nothing is printed to the console and the program immediately terminates.

☐ The program throws a `NullPointerException` (possibly after printing some output).

☐ The console prints the following output and execution terminates normally.

```
Player 2 gets the potato.
Player 2 throws the potato.
Player 2 catches the potato and throws it.
Dropped it!
```

☒ The console prints the following output and execution terminates normally.

```
Player 2 gets the potato.
Player 2 passes the potato.
Player 1 gets the potato.
Player 1 passes the potato.
Player 3 gets the potato.
Player 3 throws the potato.
Player 1 catches the potato and throws it.
Player 2 catches the potato and throws it.
Dropped it!
```

☐ The console prints the following output and execution terminates normally.

```
Player 2 gets the potato.
Player 2 passes the potato.
Player 3 gets the potato.
Player 3 passes the potato.
Player 1 gets the potato.
Player 1 throws the potato.
Player 2 catches the potato and throws it.
Dropped it!
```

☐ The console prints

```
Player 2 gets the potato.
Player 2 passes the potato.
Player 1 gets the potato.
Player 1 passes the potato.
Player 3 gets the potato.
Player 3 passes the potato.
Player 2 gets the potato.
Player 2 passes the potato.
Player 1 gets the potato.
Player 1 passes the potato.
```

and continues looping until a `StackOverflow` occurs.

6. **Java Iterators and Testing** (36 points total)

In this problem, you will create a class, `NumberGenerator`, which is an `Iterator` that produces a sequence of numbers for which a provided validator returns **`true`**.

The numbers are non-negative and they are produced in ascending order.

For example, to print out the first 10 prime numbers, we could use the following code:

```
Iterator<Integer> primes = new NumberGenerator(new PrimeNumberValidator());
for(int i = 0; i < 10; i++) {
    System.out.print(primes.next() + " ");
}
```

When executed, the output will be:

```
2 3 5 7 11 13 17 19 23 29
```

In the code above, the `PrimeNumberValidator` class (see Appendix 5) is an instance of the `NumberValidator` interface, which is defined as:

```
public interface NumberValidator {
    public boolean isValidNumber(int x);
}
```

In the case of `PrimeNumberValidator`, the `isValidNumber(x)` method returns **`true`** if `x` is prime, and returns **`false`** otherwise.

(a) (3 points) Fill in the blanks below so that correct implementations of `NumberGenerator` and `PrimeNumberValidator` will pass, *or*, if that is not possible, mark the box indicating why.

```
@Test
public void testNumberGeneratorTestA() {
    int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

    Iterator<Integer> iter = new NumberGenerator(new PrimeNumberValidator());
    for(int i = 0; i < 5; i++ ) {
        assertEquals(primes[i], iter.next());
    }
    assertEquals(13, iter.next());
    assertTrue(iter.hasNext());
}
```

OR (choose one)

- ☐ It is not possible because of syntax errors in the provided code.
- ☐ It is not possible because of type errors in the provided code.

(b) (3 points) Fill in the blanks in the test case below so that it will pass, *or*, if that is not possible, mark the box indicating why.

```
@Test
public void testNumberGeneratorTestB() {
    Iterator<Integer> iter = new NumberGenerator(x -> x==0);
    assertEquals(0, iter.next());
    assertFalse(iter.hasNext());
    assertThrows(NoSuchElementException.class, () -> iter.next());
}
```

OR (choose one)

- □ It is not possible because of syntax errors in the provided code.
- □ It is not possible because of type errors in the provided code.

Constructors The `NumberGenerator` will support two constructors. The first one accepts an **`int`** *upper bound* with the meaning that all yielded results be *strictly less than* that bound, along with a `NumberValidator`. If the bound provided is negative or the `NumberValidator` is **`null`**, the `NumberGenerator` constructor should throw an `IllegalArgumentException`.

The second constructor (that we provide) accepts just a `NumberValidator` and invokes the first constructor with a bound of `Integer.MAX_VALUE`, which is the largest **`int`** value representable in Java.

(c) (2 points)

True □ False ☒

The presence of two constructors for `NumberGenerator` that have the same name but differ in their types is an example of *overriding* in Java.

(d) (3 points) Fill in the blanks in the test case below so that it will pass, *or*, if that is not possible, mark the box indicating why.

```
@Test
public void testC() {
    assertThrows(IllegalArgumentException.class,
            () -> new NumberGenerator(-3, x -> x==0));
    assertThrows(IllegalArgumentException.class,
            () -> new NumberGenerator(3, null));

    Iterator<Integer> iter = new NumberGenerator(2, x -> true);
    assertEquals(0, iter.next());
    assertEquals(1,  iter.next());

    assertFalse(iter.hasNext());
}
```

OR (choose one)

- ☐ It is not possible because of syntax errors in the provided code.
- ☐ It is not possible because of type errors in the provided code.

Implementation

The following page contains a partial implementation of the `NumberGenerator` class itself. Complete the code so that it correctly implements the `Iterator<Integer>` interface (see the JavaDocs in Section 6).

After a `NumberGenerator` is constructed from a given `NumberValidator` and `bound`, each call to `next()` should yield the smallest non-negative (i.e. `>= 0`) integer that has not already been yielded and such that `v.isValidNumber(n) == true` and the number is `< bound`. If there is no such number, `next()` should throw a `NoSuchElementException`. The yielded values should be generated on demand, that is: your code should generate at most one number at a time.

You may *not* use any other Java libraries but you can assume that `Iterator` and appropriate exception types are imported.

(e) (4 points) You will have to add at least one **`private`** field and may add private helper method(s). In the space below, briefly state the representation *invariant(s)* used by your code for the private field(s). Hint: the representation invariant should be *exploited* by `hasNext()`.

INVARIANT:

The **`int`** field `next` is maintained so that, either:

- `0 <= next < bound` and `next` is the next number to return from `next()`, or
- `next == bound` and there are no more valid numbers

(f) (21 points) Complete this implementation:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Generates an ascending sequence of numbers all of which are
 * >= 0 and < the given bound and that also satisfy the
 * isValidNumber predicate provided by the NumberValidator.
 */
public class NumberGenerator implements Iterator<Integer> {

    private final NumberValidator v;
    private final int bound;    // all results are < bound

    // invariant:
    //   - 0 <= next < bound and next is the next number to return, or
    //   - next == bound and there are no more valid numbers
    private int next;

    public NumberGenerator(int bound, NumberValidator v) {
        if (bound < 0 || v == null) throw new IllegalArgumentException();
        this.v = v;
        this.bound = bound;
        this.next = 0;
        findNext();
    }

    // provides a constructor with a large default maximum value
    public NumberGenerator(NumberValidator v) {
        this(Integer.MAX_VALUE, v);
    }

    private void findNext() {
        while (next < bound && !v.isValidNumber(next)) {
            next = next + 1;
        }
    }

    public boolean hasNext() {
        return (next < bound);
    }

    public Integer next() {
        if (!hasNext()) throw new NoSuchElementException();
        int curr = next;
        next = next + 1;
        findNext();
        return curr;
    }
}
```

Appendices

Section 1 - Higher-Order List Processing Functions

Section 2 - Java Code for Subtyping

Section 3 - Java Code for “Connect Four”

Section 4 - Java Code for “Hot Potato”

Section 5 - Java Code for `NumberValidator` and `PrimeNumberValidator`

Section 6 - JavaDocs for `Integer` and `Iterator`

1 Higher-Order List Processing Functions

Here are the higher-order list processing functions:

```
let rec transform (f: 'a -> 'b) (xs: 'a list): 'b list =
  begin match xs with
  | [] -> []
  | h::tl -> f h :: transform f tl
  end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list) : 'b =
  begin match l with
  | [] -> base
  | h::tl -> combine h (fold combine base tl)
  end
```

2 Java Code for Subtyping

```
interface MattelToy {
    public void manufacture();
}

interface Barbie extends MattelToy {
    public void changeOutfit(String newOutfit);
    public void dance();

}

abstract class Allan implements MattelToy {
    private String name;

    public Allan(String name) {
        // Constructor body
    }

    public void manufacture() {
        // Implementation of manufacture method
    }

    public abstract void waveToAllan();
    public abstract void beach();

}

class ComputerScienceBarbie implements Barbie {
    private String name;
    private String outfit;

    public ComputerScienceBarbie(String name, String outfit) {
        // Constructor body
    }

    public void manufacture() {
        // Body of manufacture method
    }

    public void changeOutfit(String newOutfit) {
        // Body of changeOutfit method
    }

    public void dance() {
        // Body of dance method
    }

    public String code() {
        return "Generic code";
    }

    public String speak() {
```

```
        return "Hi Computer Science Barbie";
    }

}

class JavaBarbie extends ComputerScienceBarbie {

    public JavaBarbie(String name, String outfit) {
        super(name, outfit);
        // Additional constructor body for JavaBarbie
    }

    public String code() {
        return "Java code";
    }

    public void debugJavaCode(String code) {
        // Body of debugJavaCode method
    }

    public String speak() {
        return "Hi Java Barbie";
    }
}

class OCamlBarbie extends ComputerScienceBarbie {

    public OCamlBarbie(String name, String outfit) {
        super(name, outfit);
        // Additional constructor body for OCamlBarbie
    }

    public void debugOCamlCode(String code) {
        // Body of debugOCamlCode method
    }

    public String speak() {
        return "Hi OCaml Barbie";
    }
}
```

3 Java Code for “Connect Four”

```
1  class GameBoard extends JPanel {
2
3      private ConnectFour c4; // model for the game -- the code is hidden
4      private JLabel status; // current status text
5      public static final int SIZE = 5;
6      public static final int BOARD_WIDTH = 100 * SIZE;
7      public static final int BOARD_HEIGHT = 100 * SIZE;
8
9      public GameBoard(JLabel statusInit) {
10         setBorder(BorderFactory.createLineBorder(Color.BLACK));
11
12         c4 = new ConnectFour();
13         status = statusInit;
14
15         addMouseListener(new MouseAdapter() {
16             @Override
17             public void mouseReleased(MouseEvent e) {
18                 Point p = e.getPoint();
19
20                 c4.playTurn(p.x / 100, p.y / 100);
21
22                 updateStatus();
23                 repaint();
24             }
25         });
26     }
27
28     public void reset() {
29         c4.reset();
30         status.setText("Player 1's Turn");
31         repaint();
32     }
33
34     private void updateStatus() {
35         if (c4.getCurrentPlayer()) {
36             status.setText("Player 1's Turn");
37         } else {
38             status.setText("Player 2's Turn");
39         }
40     }
41
42     @Override
43     public void paintComponent(Graphics g) {
44         super.paintComponent(g);
45
46         for (int i = 1; i < 5; i++) {
47             g.drawLine(i * 100, 0, i * 100, BOARD_WIDTH);
48             g.drawLine(0, i * 100, BOARD_WIDTH, i * 100);
49         }
50
51         for (int i = 0; i < SIZE; i++) {
```

```
52              for (int j = 0; j < SIZE; j++) {
53                  int state = c4.getCell(j, i);
54                  if (state == 1) {
55                      g.setColor(Color.LIGHT_GRAY);
56                      g.fillOval(30 + 100 * j, 30 + 100 * i, 40, 40);
57                  } else if (state == 2) {
58                      g.setColor(Color.DARK_GRAY);
59                      g.fillOval(30 + 100 * j, 30 + 100 * i, 40, 40);
60                  }
61              }
62          }
63      }
64
65      @Override
66      public Dimension getPreferredSize() {
67          return new Dimension(BOARD_WIDTH, BOARD_HEIGHT);
68      }
69  }
70
71  public class GameRunner {
72
73      public static void main(String[] args) {
74          SwingUtilities.invokeLater(() -> {
75                      final JFrame frame = new JFrame("Connect Four");
76                      frame.setLocation(300, 300);
77
78                      final JPanel status_panel = new JPanel();
79                      frame.add(status_panel, BorderLayout.SOUTH);
80                      final JLabel status = new JLabel("Setting up...");
81                      status_panel.add(status);
82
83                      final GameBoard board = new GameBoard(status);
84                      frame.add(board, BorderLayout.CENTER);
85
86                      final JPanel control_panel = new JPanel();
87                      frame.add(control_panel, BorderLayout.NORTH);
88
89                      final JButton reset = new JButton("Reset");
90                      reset.addActionListener(e -> board.reset());
91                      control_panel.add(reset);
92
93                      frame.pack();
94                      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
95                      frame.setVisible(true);
96                      board.reset();
97                  }
98          );
99      }
100 }
```

4 Java Code for “Hot Potato”

```
1  class HotPotato extends Exception {
2  }
3
4  class Player {
5      public static Player thrower = null;
6      Player nextInLine;
7      String name;
8
9      Player(String name, Player target) {
10         this.name = name;
11         this.nextInLine = target;
12     }
13
14     void getsIt(HotPotato potato) throws HotPotato {
15         System.out.println(name + " gets the potato.");
16         if (this == thrower) {
17             System.out.println(name + " throws the potato.");
18             throw potato;
19         } else {
20             try {
21                 System.out.println(name + " passes the potato.");
22                 nextInLine.getsIt(potato);
23             } catch (HotPotato p) {
24                 System.out.println(name + " catches the potato and throws it.");
25                 throw p;
26             }
27         }
28     }
29 }
```

5 Java Code for `NumberValidator` and `PrimeNumberValidator`

```
public interface NumberValidator {
    public boolean isValidNumber(int x);
}

public class PrimeNumberValidator implements NumberValidator {

    @Override
    public boolean isValidNumber(int x) {
        if (x == 0 || x == 1) return false; // 0 and 1 aren't prime

        // check for factors starting at 2 (they must be <= sqrt(x) )
        for (int factor = 2; factor * factor <= x; factor++) {
            if (x % factor == 0) return false; // x has a factor
        }
        return true; // x has no factors
    }
}
```

6 JavaDocs

class Integer **implements** Comparable<Integer>

The Integer class wraps a value of the primitive type **int** in an object. An object of type Integer contains a single field whose type is **int**. The Java compiler will transparently insert calls intValue and valueOf to convert between **int** literals and Integer objects as needed.

---

int intValue()

Returns the value of this Integer as an **int**.

---

static Integer parseInt(String s)

- **Returns:** the integer value represented by the argument s in decimal.
- **Throws:** NumberFormatException - if the string does not contain a parsable integer.

---

static Integer valueOf(**int** i)

Returns the Integer object corresponding to **int** i.

---

int compareTo(Integer anotherInteger)

Compares two integers numerically. x.compareTo(y) returns x - y.

interface `Iterator<E>`

---

boolean `hasNext()`

Returns **true** if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

- **Returns:** **true** if the iteration has more elements

`E next()`

Returns the next element in the iteration.

- **Returns:** the next element in the iteration
- **Throws:** `NoSuchElementException` - if the iteration has no more elements