CIS 1200 Final Exam December 17, 2024

Benjamin C. Pierce and Swapneel Sheth, instructors

SOLUTIONS

1. **OCaml Concepts** (11 points total)

Indicate whether the following statements are true or false.

(a) True ☐ False ☒
(1 points) In OCaml, if `s = t` returns **`true`**, then `s == t` is guaranteed to return **`true`**.

(b) True ☐ False ☒
(1 points) In OCaml, if `s == t` returns **`true`**, then `s = t` is guaranteed to return **`true`**.
False because = will loop on cyclic structures.

(c) True ☐ False ☒
(1 points) In OCaml, if `x` is a variable of any type, `Some x == Some x` will always return **`true`**.

(d) True ☐ False ☒
(1 points) In our OCaml ASM, the local variables of a recursive function are stored on the heap, whereas those of a non-recursive function are stored on the stack.

(e) True ☐ False ☒
(1 points) In the OCaml ASM, a closure is used to save a copy of all mutable variables on the heap so that they can be restored later if an exception is thrown.

(f) True ☐ False ☒
(1 points) In OCaml, all infinite loops will eventually trigger a `Stack_overflow` runtime error.

(g) True ☒ False ☐
(1 points) In OCaml, if the heap ever contains a *cycle* (where following one or more pointers brings us back to where we started from), then the program that created this heap must involve mutable state.

(h) True ☐ False ☒
(1 points) One advantage of the imperative programming style compared to functional programming with no mutable references is that reasoning about the imperative style relies on a simpler formulation of the ASM.

(i) True ☐ False ☒
(1 points) The higher-order `transform` function in OCaml is more fundamental than `fold`, in the sense that any computation that can be expressed as a call to `fold` can instead be expressed as a call to `transform`.

(j) True ☒ False ☐

(2 points) The following OCaml function is tail recursive:

```
let rec lookup (x: 'a) (t: 'a tree) : bool =
  begin match t with
  | Empty -> false
  | Node (lt, v, rt) ->
         if v = x then true
         else if x < v then lookup x lt
         else lookup x rt
  end
```

2. **OCaml Lists, Trees, and Recursion** (19 points total)

Consider this list function:

```
let rec foo (n: int) (lst: int list) : bool list =
  begin match lst with
    | [] -> []
    | x::xs -> (x > n) :: (foo n xs)
  end
let ans = foo 3 [2;3;4]
```

2.1 (2 points) What is the value computed for `ans` in the code above?

```
ans = [false;false;true]
```

2.2 (3 points) Recall the definition of the list function `transform`.

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

Which of the following correctly implements the function `foo` using `transform`? (Mark all that apply.)

- ☐
```
let foo (n: int) (lst: int list) : bool list =
  transform (fun x -> (x > n) :: xs) lst
```

- ☐
```
let rec foo (n: int) (lst: int list) : bool list =
  transform (fun x -> foo n xs) lst
```

- ☒
```
let foo (n: int) (lst: int list) : bool list =
  transform (fun x -> x > n) lst
```

- ☐
```
let foo (n: int) (lst: int list) : bool list =
  transform (fun x -> x > n)
```

Consider this list function:

```
let rec m (lst: bool list) : int =
  begin match lst with
    | [] -> 0
    | x::xs -> 2*(m xs) + (if x then 1 else 0)
  end
let ans = m [true; false; true]
```

2.3 (2 points) What is the value computed for `ans` in the code above?

ans = 5

2.4 (3 points) Recall the definition of the list function `fold`.

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

Which of the following correctly implements the function `m` using `fold`? (Mark all that apply.)

☐
```
let m (lst: bool list) : int =
  fold (fun x acc -> 2*x + (if acc then 1 else 0)) 0 lst
```

☒
```
let m (lst: bool list) : int =
  fold (fun x acc -> 2*acc + (if x then 1 else 0)) 0 lst
```

☐
```
let m (lst: bool list) : int =
  fold (fun x acc -> if x then 1 else 0) (2*acc) lst
```

☐
```
let rec m (lst: bool list) : int =
  fold (fun x acc -> 2*(m acc) + (if x then 1 else 0)) 0 lst
```

Recall the definition of the type of generic binary trees:

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

2.5 (4 points) The following function, called `tree_fold` is the binary tree analogue of `fold`: it abstracts the recursion pattern into a generic function. We have left off the type annotations for the `combine` and `base` parameters—fill in those blanks so that they are consistent with the given code.

```
let rec tree_fold (combine: 'a -> 'b -> 'b -> 'b)
                  (base: 'b)
                  (t: 'a tree) : 'b =
  begin match t with
    | Empty -> base
    | Node(lt, x, rt) -> combine
                           x
                           (tree_fold combine base lt)
                           (tree_fold combine base rt)
  end
```

Consider this tree function:

```
let rec f (t: int tree) : int =
  begin match t with
    | Empty -> 0
    | Node(lt, x, rt) -> 0 + x + (f rt)
  end
let leaf3 = Node(Empty, 3, Empty)
let leaf4 = Node(Empty, 4, Empty)
let ans = f (Node(leaf3, 3, leaf4))
```

2.6 (2 points) What is the value computed for `ans` in the code above?

ans = 7

2.7 (3 points) Which of the following correctly implements the function `f` using `tree_fold`? (Mark all that apply.)

☐
```
let rec f (t: int tree) : int =
  tree_fold (fun lt x rt -> 0 + x + (f rt)) 0 t
```

☐
```
let f (t: int tree) : int =
  tree_fold (fun lacc x racc -> 0 + racc) 0 t
```

☒
```
let f (t: int tree) : int =
  tree_fold (fun x lacc racc -> 0 + x + racc) 0 t
```

☐
```
let f (t: int tree) : int =
  tree_fold (fun lacc x racc -> 0 + x + racc) 0 t
```

3. **Java Concepts** (8 points total)

Indicate whether the following statements are true or false.

(a) True ☐ False ☒
(1 points) In Java, a **`static`** method gets passed an implicit **`this`** parameter.

(b) True ☒ False ☐
(1 points) In Java, the `@Override` annotation prevents accidental overloading of a method.

(c) True ☒ False ☐
(1 points) In Java, it is possible to **`catch`** an unchecked exception (such as a `NullPointerException`) using a **`try`**-**`catch`** block.

(d) True ☐ False ☒
(1 points) In Java, if `A` is a subtype of `B`, then `Set<A>` is also a subtype of `Set<B>`.

(e) True ☒ False ☐
(1 points) In Java, `String` objects are immutable. Once they're created, their size and contents cannot be changed.

(f) True ☐ False ☒
(1 points) In the Java ASM, references are pointers to objects stored in the heap or on the stack.

(g) True ☒ False ☐
(1 points) The dynamic class of an object is always a subtype of the static type of any expression whose evaluation yields this object.

(h) True ☒ False ☐
(1 points) The variables `p` and `q` are aliases when the following program's execution reaches the line marked "HERE". (Assume that `ColoredPoint` is a subclass of `Point`.)

```
Point p = new Point(1, 2);
Point q = new ColoredPoint(1, 2, Red);
p = q;
// HERE
```

4. **Java Collections** (10 points total)

You are designing data structures to store the information needed for a Recording Studio. Choose the most appropriate data structure to keep track of the information below.

4.1 (2.5 points) Which data structure would be best to keep track of the names of all the instruments available for use at a recording studio so that we can quickly check if a particular instrument is available? (Select one.)

☒ `TreeSet<String>`

☐ `LinkedList<String>`

☐ `TreeMap<Integer, String>`

4.2 (2.5 points) Which data structure would be best to keep track of all the albums recorded by a specific artist? Each album should be accessible by name (you can assume that the name is unique) and all its songs should be stored so that they can be played in order. (Select one.)

☐ `TreeSet<String>`

☐ `LinkedList<String>`

☐ `LinkedList<TreeMap<String, String>>`

☒ `TreeMap<String, LinkedList<String>>`

☐ `TreeSet<LinkedList<String>>`

☐ `TreeSet<TreeMap<String, String>>`

4.3 (2.5 points) Each chord in a guitar chord chart can contain multiple notes. In a given chord, each note is a unique string (`"A"`, `"B#"`, `"Cb"`, etc.), and their order doesn't matter. Which data structure would be best to store a song broken down by chords, with each chord storing its constituent notes? (Select one.)

☐ `TreeSet<TreeSet<String>>`

☐ `TreeSet<LinkedList<String>>`

☐ `TreeMap<TreeSet<String>, TreeSet<String>>`

☐ `TreeMap<LinkedList<String>, LinkedList<String>>`

☐ `TreeMap<LinkedList<String>, Integer>`

☒ `LinkedList<TreeSet<String>>`

4.4 (2.5 points) Which data structure would be best to keep track of the names of all the concert venues in each city and how many people each venue can fit? All concert venues within a city should be accessible via the city's name, and we should be able to add new venues to any given city. Additionally, we should be able to update the capacity of a given venue. (Select one.)

☐ TreeSet<TreeMap<String, Integer>>

☒ TreeMap<String, TreeMap<String, Integer>>

☐ TreeMap<String, TreeSet<String>>

☐ TreeSet<LinkedList<String>>

☐ LinkedList<TreeMap<String, Integer>>

☐ TreeMap<String, TreeSet<Integer>>

5. **Inheritance and Overriding** (14 points total)

Consider the Java class declarations shown in Appendix A.

For each code snippet below, indicate what string will get printed to the console, or mark "Ill typed" if the snippet has a type error.

5.1 (2 points)

```
B x = new B();
x.print1();
```

☒ `A's print1` ☐ `B's print2` ☐ `C's print1` ☐ Ill typed

5.2 (2 points)

```
A y = new B();
y.print1();
```

☒ `A's print1` ☐ `B's print2` ☐ `C's print1` ☐ Ill typed

5.3 (2 points)

```
A q = new B();
q.print2();
```

☐ `A's print1` ☐ `B's print2` ☐ `C's print1` ☒ Ill typed

5.4 (2 points)

```
C z = new C();
z.print1();
```

☐ `A's print1` ☐ `B's print2` ☒ `C's print1` ☐ Ill typed

5.5 (2 points)

```
A v = new B();
v.callPrint();
```

☐ `A's print1` ☒ `B's print2` ☐ `C's print1` ☐ Ill typed

5.6 (2 points)

```
C w = new C();
w.callPrint();
```

☐ `A's print1` ☐ `B's print2` ☒ `C's print1` ☐ Ill typed

5.7 (2 points)

```
A u = new C();
u.callPrint();
```

☐ `A's print1` ☐ `B's print2` ☒ `C's print1` ☐ Ill typed

6. **Java Exceptions** (13 points)

The code below defines three methods, `m1`, `m2`, and `m3`, that throw and catch exceptions `ExnA` and `ExnB` (two newly declared runtime exceptions that have no relationship with each other). If we start with a call to `m1()`, some of the calls to `System.out.println` will get executed, while others will not. Please mark the appropriate box next to each of these calls to indicate whether the corresponding string will or will not get printed (i.e., put an X inside the ☐ before either `Printed` or `Not printed`).

```
class ExnA extends RuntimeException { }
class ExnB extends RuntimeException { }

static void m1() {
    System.out.println("begin m1");                  // ☐ Printed ☐ Not printed
    try {
        System.out.println("calling m2");            // ☐ Printed ☐ Not printed
        m2();
        System.out.println("returned from m2");      // ☐ Printed ☐ Not printed
    } catch (ExnA e) {
        System.out.println("m1 caught ExnA");        // ☐ Printed ☐ Not printed
    } catch (ExnB e) {
        System.out.println("m1 caught ExnB");        // ☐ Printed ☐ Not printed
    }
    System.out.println("end m1");                    // ☐ Printed ☐ Not printed
}

static void m2() {
    System.out.println("begin m2");                  // ☐ Printed ☐ Not printed
    try {
        System.out.println("calling m3");            // ☐ Printed ☐ Not printed
        m3();
        System.out.println("returned from m3");      // ☐ Printed ☐ Not printed
    } catch (ExnA e) {
        System.out.println("m2 caught ExnA");        // ☐ Printed ☐ Not printed
        System.out.println("about to throw ExnB");   // ☐ Printed ☐ Not printed
        throw new ExnB();
    } catch (ExnB e) {
        System.out.println("m2 caught ExnB");        // ☐ Printed ☐ Not printed
    }
    System.out.println("end m2");                    // ☐ Printed ☐ Not printed
}

static void m3() {
    System.out.println("begin m3");                  // ☐ Printed ☐ Not printed
    try {
        System.out.println("about to throw ExnA");   // ☐ Printed ☐ Not printed
        throw new ExnA();
    } catch (ExnB e) {
        System.out.println("m3 caught ExnB");        // ☐ Printed ☐ Not printed
    }
    System.out.println("end m3");                    // ☐ Printed ☐ Not printed
}
```

Answer:

```
begin m1
calling m2
begin m2
calling m3
begin m3
about to throw ExnA
m2 caught ExnA
about to throw ExnB
m1 caught ExnB
end m1
```

7. **Iterators** (27 points total)

In this problem, you will use the design process from class to implement a Java class called `BufferedIterator`. Read through Steps 1 and 2 below, then complete Steps 3 and 4.

Step 1: Understand the problem Recall that an iterator is an object that yields a sequence of elements. However, one issue with the iterator interface is that there is no way to peek at the next object without returning it.

For example, suppose one wanted a method that would advance an integer iterator so that it skips over all negative numbers. Although the following definition might seem reasonable, it has the wrong behavior. When given an iterator, it skips over any initial negative numbers produced by the iterator, but it *also* skips over the first non-negative number.

```
void skipNegativeWRONG(Iterator<Integer> it) {
        while (it.hasNext() && it.next() < 0) { }
}
```

A buffered iterator would solve this problem by being able to peek at the next number in the iteration, without advancing the iterator. That way, only the negative numbers can be skipped.

```
void skipNegative(BufferedIterator<Integer> it) {
        while (it.hasNext() && it.peek() < 0) {
            it.next();
        }
    }
```

Step 2: Design the interfaces The Javadocs for the `Iterator<E>` interface are given in Appendix B. In this problem you will develop a generic `BufferedIterator` class that implements this interface.

The constructor of this class should take another iterator as an argument and add "buffering", i.e. the ability to peek ahead to the next value, without advancing the iterator. The constructor of this class should have the following declaration.

```
public BufferedIterator(Iterator<E> i)
```

If `i`, the provided iterator, is `null`, the `BufferedIterator` constructor should throw an `IllegalArgumentException`.

The `peek` operation should have the same interface as the `next` method. The only difference is that it shouldn't advance the iterator when called. If there is no element to return from `peek`, then the iterator should throw a `NoSuchElementException`.

```
public E peek()
```

(There are no questions for you on this page.)

(a) (5 points) Suppose you are given an iterator for a list that contains only the value 1, and no other numbers. What test cases could you write for an instance of the `BufferedIterator` class constructed from this iterator?

Describe, in words, five different tests for such an instance, called `b`. You may assume that each test starts with a fresh definition of `b`. Your description of the test must be specific, describing either the outputs of methods in the `BufferedIterator` class or any exceptions that could be thrown. For example, one test case that you might include is "*two successive calls of* `b.peek()` *both return 1*."

You will be graded on the correctness and comprehensiveness of your test cases. We want **five good tests** in addition to the example above. Each test must be **non-overlapping**, which means it tests a different part of the functionality.

Possible answers:

1. *hasNext returns true*
2. *next returns the value 1*
3. *peek returns the value 1 (duplicates test above)*
4. *after a call to peek, hasNext returns true*
5. *after a call to peek, peek returns 1 (duplicates test above)*
6. *after a call to peek, next returns 1*
7. *after a call to next, hasNext returns false*
8. *after a call to next, peek throws NoSuchElement exception*
9. *after a call to next, next throws NoSuchElement exception*
10. *after a call to hasNext, peek returns 1 (hasNext doesn't advance)*
11. *after a call to hasNext, next returns 1*
12. *after a call to hasNext, hasNext returns true*

One point per test up to five points. Tests that duplicate the provided test case above don't receive credit. Tests that are for buffered iterators other than b above generally don't receive credit, and points are deducted for tests that talk about skipping negative numbers.

(b) (2 points) Choose one of your tests above (tell us which one by circling its number above) and complete the implementation below.

For example:

```
@Test
public void testTwoPeeks(){
    List<Integer> list = new LinkedList<Integer>();
    list.add(1);
    BufferedIterator<Integer>bit =
            new BufferedIterator<Integer>(list.iterator());
    assertEquals(1,bit.peek());
    assertEquals(1,bit.peek());
}
```

No points deducted for failing to circle the number.

Step 4: Implementation (20 points)

Complete the implementation of the `BufferedIterator` class on the next page. We have provided you with the implementation of the constructor for this class. **Do not modify this definition.**

If you need more space for any of your answers, you may use the scratch space on page 20.

Note: You may assume that appropriate import statements bring `Iterator` and `NoSuchElementException` into scope; we omit them to save space. **You may not use any additional classes or libraries, nor add any import statements to your solution.**

Hint: You might want to think about test cases other than the ones you wrote for `b` on the previous problem. Are there any other iterators that you might need to consider?

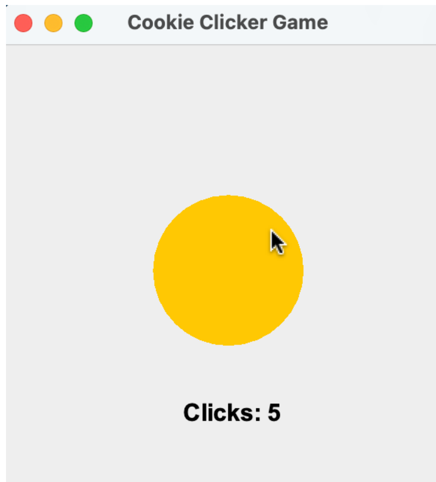
Hint: We have declared a helper method called `advance()` that you can use to move your iterator forward to the next result (if any). Complete this method as you see fit. This method is used in the definition of the `BufferedIterator` constructor that we have provided for you. You may also use `advance` in your new code, as appropriate.

(There is nothing for you to answer on this page.)

```java
public class BufferedIterator<E> implements Iterator<E> {
    private Iterator<E> it;
    private E nextElement;
    // Additional fields if needed:

    private boolean hasNext;

    public BufferedIterator(Iterator<E> i) {
        if (i == null) { throw new IllegalArgumentException(); }
        this.it = i;
        advance();
    }

    private void advance() {     // Complete:

        if (it.hasNext()) {
            hasNext = true;
            nextElement = it.next();
        } else {
            hasNext = false;
        }

    }

    public boolean hasNext() {     // Complete:

        return hasNext;

    }

    public E next() {     // Complete:

        if (hasNext()) {
            E n = nextElement;
            advance();
            return n;
        } else {
            throw new NoSuchElementException();
        }

    }

    public E peek() {      // Complete:

        if (hasNext) {
            return nextElement;
        } else {
            throw new NoSuchElementException();
        }

    }

}
```

8. **Java Swing Programming** (18 points total)

Appendix D shows code for an implementation of a “Cookie Clicker” game in Java. Each time the cookie is clicked, the counter is incremented and the updated value is shown. The image below shows the GUI after a few clicks.



The following questions test your understanding of both Java and Swing programming idioms:

$\boxed{8.1}$ (2 points) On line 28, calling **super**.`paintComponent(g)` invokes the constructor of the `JPanel` class.

True ☐ False ☒

$\boxed{8.2}$ (2 points) How many occurrences of the **new** keyword in the `CookiePanel` class correspond to anonymous inner classes? (Select one.)

☐ 0
☒ 1
☐ 2
☐ 3

8.3 (2 points) How will the program's behavior change if we delete the call to `repaint()` on line 15? Assume we do not resize the display window manually. (Select one.)

☐ Nothing at all will ever be displayed – just a blank window.

☐ The initial GUI will be displayed, but the next time the canvas is clicked, the screen will become blank.

☐ The initial GUI will be displayed. Clicking the cookie will *not* update the internal `clickCount` and the counter display will never change.

☒ The initial GUI will be displayed. Clicking the cookie will update the internal `clickCount`. The counter display may or may not change, depending on decisions Swing makes internally and other interactions with the display.

☐ No change in behavior.

8.4 (2 points) Which of the following statements are true? (Mark all that apply.)

☒ `CookiePanel` is a subtype of `JPanel` and can itself contain other components.

☒ `JFrame` is a container component that can contain multiple components.

☒ The `CookiePanel` is added to the `JFrame`.

☐ The `JFrame` is added to the `CookiePanel`.

8.5 (2 points) What would happen if we changed 250 to 150 on line 35—i.e., changed the whole line to `g.drawString("Clicks: "+ clickCount, 120, 150)`? (Select one.)

(Hint: The second and third arguments to the `drawString()` method are the `x` and `y` coordinates, respectively, for where the string should be drawn.)

☐ The cookie would be drawn on top of the counter.

☒ The counter would be drawn on top of the cookie.

☐ The counter would appear to the right of the cookie.

☐ The counter would appear to the left of the cookie.

8.6 (2 points) What would happen if we declared the `clickCount` variable in `CookiePanel` as **`static`**? (Select one.)

☐ The counter will start at the previous value of `clickCount` every time the game restarts.

☐ The counter will increment for each click, as before, but repainting will stop working.

☐ The program will not compile.

☒ No change in behavior.

8.7 (6 points) Modify the `CookiePanel` class to add a "Reset" button that resets the click count to 0. Write a code snippet below to add the button and handle the reset action. Assume your code is inserted after line 18.

The methods of the `JButton` class are summarized in Appendix C.

```
JButton resetButton = new JButton("Reset");
add(resetButton);
resetButton.addActionListener(e -> {
    clickCount = 0;
    repaint();
});

// OR

JButton resetButton = new JButton("Reset");
add(resetButton);
resetButton.addMouseListener(new MouseAdapter() {
     @Override
     public void mouseClicked(MouseEvent e) {
            clickCount = 0;
            repaint();
     }
});
```

Scratch Space

Use this page for work that you do not want us to grade. If you run out of space elsewhere in the exam and you ***do*** *want to put something here that we should grade, make sure to put a clear note in the normal answer space for the problem in question.*

Appendices

Section A - Java Code for Inheritance and Overriding
Section B - JavaDocs for `Iterator`
Section C - JavaDocs for `JButton`
Section D - Java Code for “Cookie Clicker”

A Java classes for Inheritance and Overriding

```java
class A {
    public void print1() {
        System.out.println("A’s print1");
    }

    public void callPrint() {
        print1();
    }
}

class B extends A {
    public void print2() {
        System.out.println("B’s print2");
    }

    @Override
    public void callPrint() {
        print2();
    }
}

class C extends A {
    @Override
    public void print1() {
        System.out.println("C’s print1");
    }
}
```

B Java `Iterator<E>` interface

```
interface Iterator<E> {

    boolean hasNext()
    // Returns true if the iteration has more elements.

    E next()
    // Returns the next element in the iteration.

    ...
}
```

C Java `JButton` methods

```
class JButton {
    public JButton () {
        ...
    }

    public JButton (String text) {
        ...
    }

    public void addActionListener (ActionListener l) {
        ...
    }

    public void addMouseListener (MouseListener l) {
        ...
    }

    public void addKeyListener (KeyListener l) {
        ...
    }

    ...
}
```

D Java Code for “Cookie Clicker”

```
class CookiePanel extends JPanel {
    private int clickCount = 0;

    public CookiePanel() {
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                // Check if the click is within the "cookie" bounds
                int cookieX = 100;
                int cookieY = 100;
                int cookieSize = 100;
                if (e.getX() >= cookieX && e.getX() <= cookieX + cookieSize &&
                        e.getY() >= cookieY && e.getY() <= cookieY + cookieSize) {
                    clickCount++;
                    repaint();
                }
            }
        });
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(300, 300);
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // Draw the "cookie"
        g.setColor(Color.ORANGE);
        g.fillOval(100, 100, 100, 100);
        // Draw the click counter
        g.setColor(Color.BLACK);
        g.setFont(new Font("Arial", Font.BOLD, 16));
        g.drawString("Clicks: " + clickCount, 120, 250);
    }
}

public class GameRunner {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> {
            JFrame f = new JFrame("Cookie Clicker Game");
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            f.add(new CookiePanel());
            f.pack();
            f.setVisible(true);
        });
    }
}
```