1 Proctortrack

The technical requirements for exams.

1. Reliable internet connection with sufficient capacity to support video proctoring
2. Need a video face scan and show photo ID
3. Exam room may also be scanned
4. Computer screen recorded
5. Your actions recorded
6. Software is used to restrict access to other computer applications

2 Object Oriented Programming

2.1 Java Intro

Why programming language:

1. Computers only understand binary (0/1) code
2. Human readable: Python, Java, etc.
3. Compiler: A program, translates programming language(human readable) to machine code (0/1). Or high level language to low level language

Java basic:

1. Java program: Must be stored in .java files
2. Java compiler: Does Not produce machine code. But translates Java program into Java bytecode (an intermediate language)
3. Java bytecode: .class files
4. Java interpreter(virtual machine): execute java bytecode
5. Commands: *javac*(compiler) and *java*(interpreter)

2.2 Objects and Classes

Classes: template, pattern, model, definition.
See Listing 1

Objects: program modules

1. Objects have
   - Properties: data (*attributes, fields, instance variables*)
   - Behaviours: actions (*(instance) methods*)
2. Every object belongs to a specific class

Keywords you need to know:

1. *constructor*: called automatically when an object is created, initialize the attributes, must be the same name as the class
2. *new*:create new objects

3. *this*: See Figure 9.2
   Listing 2 illustrates that scope of variables. Explain why only one student is printed.
4. *equals* method: two objects are the same only when all the instance variables are the same (need to know the difference with ==).
5. javadoc
6. *final*: constant, can never be changed once defined
7. array: index starts at 0
8. overloading: same method, different behavior
9. Pseudocode
10. formal parameter: in method definition (firstName)
11. actual parameter: when invoke a method (Alice)
12. Modularity: cut big problem into small ones (modules)
13. Encapsulation: put the data and operations together
14. *static*: shared among all the objects of the same class(See Figure 9.1)
    static methods can only reference static variables!!
    Listing 3 uses incorrectly static variable. Explain why when printing the list of names, only one name is displayed.

2.3 Memory

Java data types:

1. primitive
   - boolean, char, byte, short, int, long, float, double
   - a variable stores a value
2. non-primitive
   - classes
   - a variable stores a address
   - reference variable: the name of a non-primitive variable (Person p)
3. Comparing: See Figures 9.3 9.4 9.5 9.6 9.7

3 Debugging

1. Compilation errors (syntax errors):
   a) language error
   b) found by the compiler
   c) program with compilation errors cannot be run
   d) errors and warnings
   e) forgetting a semicolon, closing bracket, redecalring

   See Listing 4.
2. Runtime errors:
   a) Program runs but gets exceptions
   b) See more in Section Exceptions

See Listing 5.

3. Logic errors:Program runs but results are not correct
Listing 6 looks for a value in an array

Debugging Strategies:

1. Trace by hand
2. Add main method
3. Add print statements
4. Use a debugger

4 EXCEPTIONS

1. Exception: an abnormal or erroneous situation at runtime
2. Exceptions can be thrown by Java VM or programs
3. throw, catch, re-throw, try-catch, finally
4. checked and unchecked exceptions

See Listing 7.

4.1 Java Exceptions

Java exceptions are objects:

1. ArithmeticException: ex. divided by 0
2. IndexOutOfBoundsException
3. IOException: file does not exist
4. NullPointerException

See the example in Listing 8. What's the output of the program?
What if we change the *ArrayIndexOutOfBoundsException* into *NullPointerException*?

5 INHERITANCE

5.1 Inheritance Terminology

1. subclass or child: is the derived new class, inherits the attributes and methods of
2. superclass or parent
3. *extends*: used to make a subclass
4. Visibility:
   - public: children classes can access them (except constructor)
   - private:children classes cannot access them (can use getters and setters)
   - protected: accessed by class in the same package or and subclass
5. *super*: refer to its parent class
   - The first line of a child's constructor should be: super(...);
   - access parent class methods or variables

6. *is a*: object of a subclass is a instance of superclass
7. override: child defines a method with the same signature as the parent, this is overriding
   Method with final keyword cannot be overridden
8. A variable of superclass type may reference and object of a subclass type, (but not the other way around!!!)
   Because a Square is a Rectangle, but a Rectangle is not a Square
9. Polymorphism:
   - Dynamic binding: which method to use decided not until run time!
   - Casting: Let compiler know what this object is but does not change the object type
   - instanceof: is an operator, return true or false
   - The Object class: any class extends Object
     - toString(): class name@value
     - equals(): compare address instead of data!

See Listing 9 10 11.

6 Linked List

1. One item is linked to other items
2. Single linked list: each item points to the next one
3. Double linked list: each item points to the next and the previous one
4. Items do not have to be in consecutive memory
   So can insert and delete without shifting data (advantage over array)
5. Nodes:
   - Data
   - pointer or pointers
6. Operations: insert, delete
   See Listing 12 13

7 Stack

ADT:

1. An abstract data type is a data type that is not programming-language specific
2. We describe abstract data types by the public methods that they provide
3. We do not specify the specific implementations for an abstract data type

Collections:

1. We want to group together items into a conceptual unit
2. Linear: We can tell the predecessor and successor (stack, queue, list)
3. Non-Linear: We can't (tree, graph)

Stack:

1. Linear collection

2. A stack of plate: you put on the top and take a plate from the top
3. All the activity happens at the top of the stack
4. Push: add to the top
5. Pop: remove from the top
6. Peek: look at the top
7. Applications:Reversing things (assignment 2),Postfix expressions, Undo ,Back button in browser
8. Implementations
   Array See Listing 14
   Linked List See Listing 15

8 TERMINOLOGY

8.1 Keyword Terminology

1. catch: defines how a particular kind of exception is handled
2. extends: a class is derived from an existing class
3. final: the value of the variable cannot be changed
4. finally: always executes after either a try or catch
5. implements: a class must provide method implementations to an interface
6. import: include classes from other libraries
7. interface: a class containing a collection of constants and abstract methods
8. new: call a class's constructor
9. null: a reference to no object
10. private: visible within the same class
11. protected: visible within the same class or subclass
12. public: visible from within and outside a class
13. super: used in a subclass to refer to the parent class
14. this: a reference to the current object (self)
15. throw: creates an exception
16. try: contains a group of statements that may thrown an exception

8.2 Concept Terminology

1. abstract data type: a data type whose values and operations are not inherently defined within a programming language
2. abstraction: (a.k.a. information hiding) making implementation details inaccessible; hiding "the right amount" of complexity
3. abstract class: a generic concept in a class
4. hierarchy; cannot be instantiated and usually contains one or more abstract methods
5. abstract method: a method that does not have an implementation
6. actual parameter: parameters sent into the method
7. alias: two variables that reference the same object
8. collection: an object that gathers and organizes other objects
9. constructor: special method that shares the same name as the class and initializes an object

10. data structure: the collection of programming constructs used to implement an abstract data type
11. dynamic binding: (a.k.a. late binding) binding a method invocation to a method definition at run time
12. encapsulation: variables contained within an object should only be accessible from within that object
13. error: generally represents an unrecoverable situation and should not be caught
14. exception: an object that defines an unusual or erroneous situation
15. exception propogation: when an exception isn't handled immediately, control returns to the calling method
16. formal parameter: parameters located inside the method
17. inheritance: creating a new class that is based on an existing class
18. instance variable: variable declared outside of methods in a class
19. interface: the public methods through which we can interact with an object
20. is-a relationship: the derived class should be a more specific version of the superclass
21. linked structure: a data structure that uses object reference variables to create links between objects
22. method overloading: multiple methods with the same name but different method signatures
23. method signature: the method name, number of parameters, types of those parameters, and ordering of those types of parameters
24. modularity: dividing a large program into small components; each module should perform one welldefined task
25. object reference: a variable whose value is a memory location; the memory location specifies where the object is located in memory
26. polymorphism: a reference variable that can refer to different types of objects at different points in time
27. scope: the part of a program in which a valid reference to a variable can be made
28. stack: a linear collection whose elements are added and removed from the top
29. static method: (a.k.a. class method) a method that can be called without first needing an object to be created; it is called from the class instead of the object, e.g. Math.Random()
30. static variable: (a.k.a class variable) a variable that is shared across all instances of a class; every object created with that class type shares the same static variables
31. subclass: (a.k.a. child class) the class that is based on an existing class (inherits from a superclass)
32. superclass: (a.k.a. parent class, base class) the class that is used to derive a new class

9 FIGURES AND LISTINGS

Listing 1: A class example

```java
/**
 * Class that represents a person with attributes name, email address
 * @author CS1027
 *
 */
public class Person {

        /* Attribute declarations (fields, instance variables)*/
        private String lastName;          // last name
        private String firstName;         // first name
        private String email;             // email address
        //private: other people can't access these variables


        //Constructor definitions
        /**
         * Constructor initializes the person's name and email address
         * Must be the same name as the class name
         */
        public Person(String firstName, String lastName, String email) {
                this.firstName = firstName;
                this.lastName = lastName;
                this.email = email;
        }

        //Methods definitions
        /**
         * getters
         * getName method returns the person's full name
         * @return first name followed by last name, blank separated
         */
        public String getName(){
                return firstName + " " + lastName;
        }

        /**
         * setters
         * setEmail method sets the person's email address
         * @param email
         */
        public void setEmail (String email) {
                this.email = email;
        }
}
```

Listing 2: Wrong scope

```
public class WrongClass {
    private static int numStudents = 1;
    private static String[] list;

    public WrongClass() {
        int numStudents = 10;
        list = new String[10];
        for (int i = 0; i < 10; ++i)
            list[i] = "Student " + i;
    }
    public static void main(String[] args) {
        // Print the list of students
        WrongClass c = new WrongClass();
        System.out.println("Students:");
        for (int i = 0; i < numStudents; ++i)
            System.out.println(list[i]);
    }
}
```

Listing 3: Static variables

```
public class StaticPerson {

  private static String name;

    public StaticPerson(String newName) {
        name = newName;
    }

    public  static String getName() {
        return name;
    }

    public static void main(String[] args) {
        name = "Joe Doe";

        StaticPerson[] list = new StaticPerson[10];

        list[0] = new StaticPerson("Jane Doe");
        list[1] = new StaticPerson("Jr Doe");
        list[2] = new StaticPerson(name);

        System.out.println("List of names:");
        System.out.println(list[0].getName());
```

```
                System.out.println(list[1].getName());
                System.out.println(list[2].getName());
        }

}
```

Listing 4: Code with compilation errors

```
public class CompilationErrors {

        private  int[] a;
        private  int length;

        public static void main(String[] args) {

                CompilationErrors c = new CompilationErrors();

                c.length = Integer.parseInt(args[0]);
                a = new int[length];
        }

        private void initialize() {
                int count;

                while (System.in.read() != -1)
                    ++count;
                System.out.println("Input has " + count + " chars.");
        }

        void printit() {
                int j;

                for (int i = 0; i < length; ++i)
                        System.out.println(a[i]);
        }
        System.out.println("done");

}
```

Listing 5: Code with runtime errors

```
public class RunTimeError {

        public static void main(String[] args) {
                int[] a = new int[10];
                for (int i = 0; i <= 10; ++i)
                        a[i] = i;

        for (int i = 0; i <= 10; ++i)
            System.out.println(a[i]);
    }

}
```

Listing 6: Code with logic errors

```
public class FindTest {

    public static void main(String[] args) {
        int[] items = {4, 5, 6, 7};
        // Store values 4, 5, 6, 7 in an array of length 4
        if (find(6, items, 4) == false) {
            System.out.println("Value 6 not found.");
            for (int i = 0; i < items.length; ++i)
                System.out.print(items[i] + " ");
            System.out.println("");
        }
        else System.out.println("Value 6 found");
    }

    public static boolean find(int value, int[] data, int numItems) {
        int index = 0;
        boolean flag = false;
        while (index < numItems) {
            if (value == data[index])
                flag = true;
            else flag = false;
            ++index;
        }
        return flag;
    }

}
```

Listing 7: Throw and deal with exceptions

```
//throw exceptions
public T pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException("Stack is empty");
    ...
}



//re−throw exceptions
```

```
private static void helper(ArrayStack<String> s)throws
EmptyStackException {
        s.pop();
        ...
}

//catch exceptions
//try-catch-finally
public static void main(){
        ArrayStack<String>s = ...;
        try {
                        helper(s);
                        ...
        }
        catch (EmptyStackException e) {
                        System.out.println(e.getMessage());
        }
        catch (SomeOtherException e) {
                        ...
        }
        finally{
                System.out.println("This will always be printed!\n");
        }
        ...
}
```

Listing 8: A tricky example

```
public void func(){
        int a[5];
        System.out.println("B");
        try{
                for(int i = 0; i <= 5; i ++){
                        a[i] = i;
                }
                System.out.println("C");
        }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("D");
        }finally{
                System.out.println("E");
        }
        System.out.println("F");
}

public static void main(){
        try{
                System.out.println("A");
                func();
                System.out.println("G");
        }catch(Exception e){
                System.out.println("H");
        }
}
```

Listing 9: Rectangle class

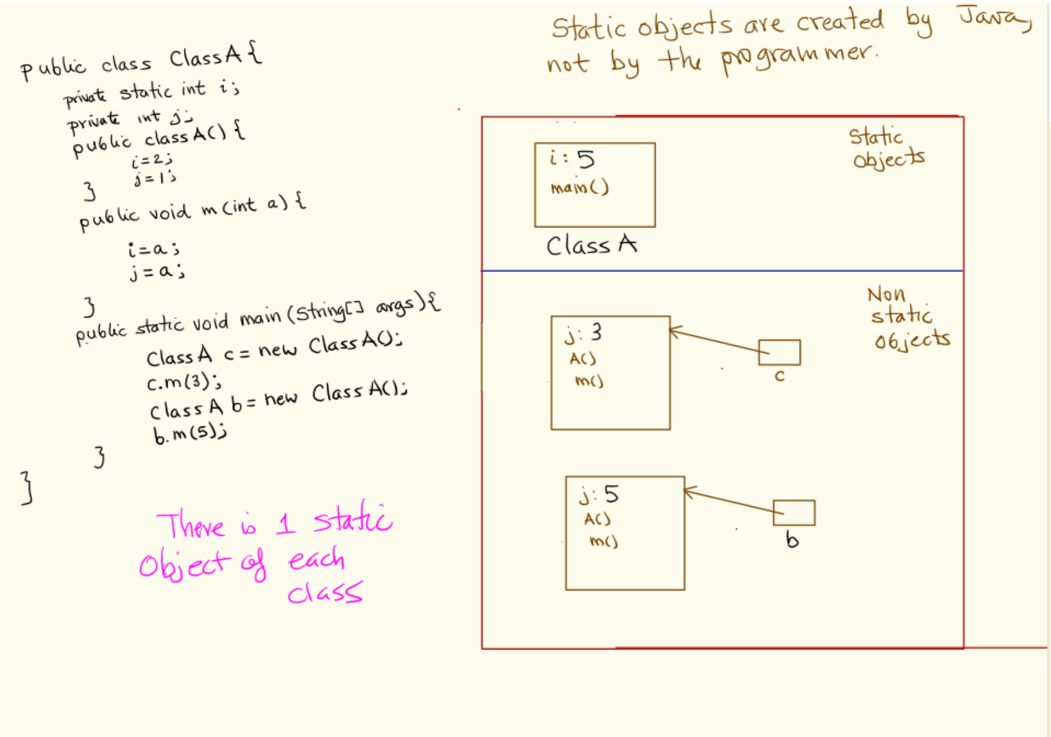
```
public class Rectangle {

  private int length;
  private int width;

  public Rectangle(int length, int width) {
    this.length = length;
    this.width = width;
  }
  public int getLength() {
    return length;
  }
  public int getWidth() {
    return width;
  }
  public void setLength(int length){
        this.length = length;
  }
  public int area() {
    return length*width;
  }
  public String toString() {
    return "Rectangle: " +
           "Length(" + length + ") " +
           "Width(" + width + ")";
  }
}
```

Listing 10: Sqyare class

```
public class Square extends Rectangle {

  // no new attributes need be introduced
  public Square(int s) {
    // call the 2 variable superclass constructor
    super(s, s);
  }

  public int getSide() {
    return getWidth();
  }

  public String toStringAsRectangel() {
    return super.toString();
  }

  public String toString() {
    return "Square: Side(" + getSide() + ")";
  }
}
```

Listing 11: Polymorphism

```
public class TestRectangle {

        public static void main(String[] args) {
                Rectangle r = new Rectangle(4,5);
                Square s = new Square(5);

                System.out.println("1." + r.toString());
                // which toString does it use? Rectangle

                System.out.println("2." + s.toString());
                // which toString does it use? Square

                Rectangle r2 = s;
                System.out.println("3." + r2.toString());
                // which toString does it use? Square

                Rectangle t = new Square(6);
                System.out.println("4." + t.toString());
                // which toString does it use? Square

                //Square s2 = new Rectangle(7,8);
                // why does compiler complain?

                r = new Square(5);
                System.out.println("5." + r.toString());
                // which toString does it use? Square

                System.out.println("6. width " + r.getWidth());
                // why is this OK?

                //System.out.println(r.getSide());
                // why does compiler complain?

                if (r instanceof Square)
                    System.out.println(( (Square)r ).getSide()); // this is OK
        }
}
```



```
public class ClassA {
    private static int i;
    private int j;
    public classA() {
            i=2;
            j=1;
    }
    public void m(int a) {

            i=a;
            j=a;

    }
    public static void main(String[] args){
            ClassA c = new ClassA();
            c.m(3);
            classA b = new ClassA();
            b.m(5);
    }
}
```

Figure 9.1: Static object

Listing 12: Singly Linked list

```
public class LinearNode<T>{
    private LinearNode<T> next;
    private T dataItem;

    public LinearNode(){
        next = null;
        dataItem = null;
    }

    public LinearNode (T elem){
        next = null;
        dataItem = elem;
    }
    ...
}

public class SinglyLinkedList<T> {

    private static LinearNode<T> front;
    ...
    public void insert(LinearNode<T> newNode, LinearNode<T> predecessor){
        if(predecessor == null){
            newNode.setNext(front);
            front = newNode;
        }
        else{
            LinearNode<T> succ = predecessor.getNext();
            newNode.setNext(succ);
            predecessor.setNext(newNode);
        }
    }
    public boolean delete (LinearNode<T> nodeToDelete) {
        LinearNode<T> current, predecessor;
        current = front;
        predecessor = null;
        while ((current != null) && (current != nodeToDelete)) {
            predecessor = current;
            current = current.getNext();
        }
        if (current == null) return false;
        else {
            if (predecessor != null)
                predecessor.setNext(current.getNext());
            else front = front.getNext();
            return true;
        }
    }
    ...

}
```

Listing 13: Doubly Linked list

```
public class LinearNode<T>{
    private LinearNode<T> next;
    private LinearNode<T> prev;
    private T dataItem;

    public LinearNode(){
        next = null;
        prev = null;
        dataItem = null;
    }

    public LinearNode (T elem){
        next = null;
        prev = null;
        dataItem = elem;
    }
    ...
}

public class DoublyLinkedList<T> {

    private static LinearNode<T> front;
    private static LinearNode<T> tail;
    ...
    public void insert(LinearNode<T> newNode, LinearNode<T> predecessor){
        if(predecessor == null){

            newNode.setNext(front);
            front.setPrev(newNode);
            newNode.setPrev(null);
            front = newNode;
        }
        else{
            LinearNode<T> succ = predecessor.getNext();
            newNode.setNext(succ);
            succ.setPrev(newNode);
            predecessor.setNext(newNode);
            newNode.setPrev(predecessor);
        }
    }
    public boolean delete (LinearNode<T> nodeToDelete) {
        LinearNode<T> current, predecessor;
        current = front;
        predecessor = null;
        while ((current != null) && (current != nodeToDelete)) {
            predecessor = current;
            current = current.getNext();
        }
        if (current == null) return false;
        else {
            if (predecessor != null){
                predecessor.setNext(current.getNext());
                current.getNext().setPrev(predecessor);
            }

            else front = front.getNext();
            return true;
        }
    }
    ...

}
```

Listing 14: Array stack

```
public class ArrayStack<T> implements StackADT<T>{
  private final int DEFAULT_CAPACITY = 100;
  private int top;
  private T[] stack;

  public ArrayStack(){
    top = 0;
    stack = (T[])(new Object[DEFAULT_CAPACITY]);
  }

  public ArrayStack (int initialCapacity){
    top = 0;
    stack = (T[])(new Object[initialCapacity]);
  }

  public void push (T element){
    if (size() == stack.length)
      expandCapacity();

    stack[top] = element;
    top++;
  }
  public T pop() throws EmptyCollectionException{
    if (isEmpty())
      throw new EmptyCollectionException("Stack");

    top--;
    T result = stack[top];
    stack[top] = null;

    return result;
  }

  public T peek() throws EmptyCollectionException{
    if (isEmpty())
      throw new EmptyCollectionException("Stack");

    return stack[top-1];
  }
  public boolean isEmpty(){
    return (top == 0);
  }
  public int size(){
    return top;
  }
  ...

}
```

Listing 15: Linked list stack

```
public class LinkedStack<T> implements StackADT<T> {
  private int count;
  private LinearNode<T> top;

  public LinkedStack() {
    count = 0;
    top = null;
  }

  public void push (T element) {
    LinearNode<T> temp = new LinearNode<T> (element);

    temp.setNext(top);
    top = temp;
    count++;
  }

  public T pop(){
    if (isEmpty())
      throw new EmptyCollectionException("Stack");

    T result = top.getElement();
    top = top.getNext();
    count--;

    return result;
  }

  public T peek()  {
    if (isEmpty())
      throw new EmptyCollectionException("Stack");

    return top.getElement();
  }
  ...

}
```

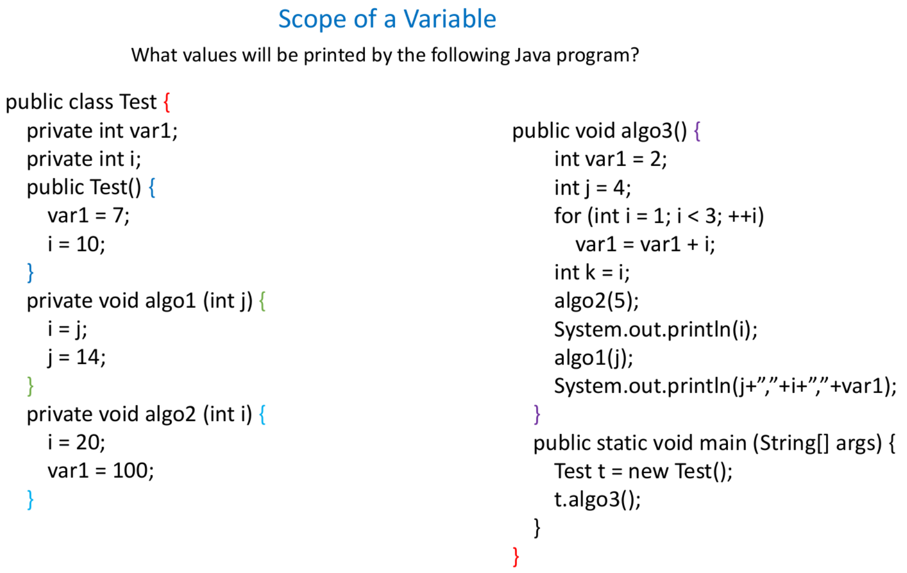
Listing 16: Circular array queue

```
/**
 * CircularArrayQueue represents an array implementation of a queue in which the
 * indexes for the front and rear of the queue circle back to 0 when they reach
 * the end of the array.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @author CS1027
 */

public class CircularArrayQueue<T> implements QueueADT<T> {
        private final int DEFAULT_CAPACITY = 100;
        private int front; // Index of the first data item in the queue
        private int rear; // Index of the last data item in the queue
        private int count; // Number of data items in the queue
        private T[] queue;

        /**
         * Creates an empty queue using an array of the default capacity.
         */
        public CircularArrayQueue() {
                front = 1;
                rear = 0;
                count = 0;
                queue = (T[]) (new Object[DEFAULT_CAPACITY]);
        }

        /**
         * Creates an empty queue using the specified capacity.
         *
         * @param initialCapacity the integer representation of the initial size of the
         *                        circular array queue
         */
        public CircularArrayQueue(int initialCapacity) {
                front = 1;
                rear = 0;
                count = 0;
                queue = ((T[]) (new Object[initialCapacity]));
        }

        /**
         * Adds the specified element to the rear of this queue, expanding the capacity
         * of the queue array if necessary.
         *
         * @param element the element to add to the rear of the queue
         */
        public void enqueue(T element) {
                if (size() == queue.length)
                        expandCapacity();

                rear = rear + 1;
                queue[rear] = element;
                count++;
        }

        /**
         * Removes the element at the front of this queue and returns a reference to it.
         * Throws an EmptyCollectionException if the queue is empty.
         *
         * @return the reference to the element at the front of the queue that was
         *         removed
         * @throws EmptyCollectionException if an empty collections exception occurs
         */
        public T dequeue() throws EmptyCollectionException {
                if (isEmpty())
                        throw new EmptyCollectionException("queue");

                T result = queue[front];
                queue[front] = null;
                front = front + 1;
                if (front > queue.length)
                        front = 0;
                count--;

                return result;
        }

        /**
         * Returns a reference to the element at the front of this queue. The element is
         * not removed from the queue. Throws an EmptyCollectionException if the queue
         * is empty.
         *
         * @return a reference to the first element in the queue
         * @throws EmptyCollectionException if an empty collections exception occurs
         */
        public T first() throws EmptyCollectionException {
                // left as programming project
        }

        /**
```

```
	 * Returns true if this queue is empty and false otherwise.
	 *
	 * @return returns true if this queue is empty and false if otherwise
	 */
	public boolean isEmpty() {
		return count == 0;
	}

	/**
	 * Returns the number of elements currently in this queue.
	 *
	 * @return the integer representation of the size of this queue
	 */
	public int size() {
		return count;
	}

	/**
	 * Returns a string representation of this queue.
	 *
	 * @return the string representation of this queue
	 */
	public String toString() {
		String result = "QUEUE: ";
		return result;
	}

	/**
	 * Creates a new array to store the contents of this queue with twice the
	 * capacity of the old one.
	 */
	public void expandCapacity() {
		T[] larger = new T[queue.length * 2];

		for (int i = 0; i < count; i++)
			larger[i] = queue[i];

		rear = i;
		queue = larger;
	}
}
```

Listing 17: Linked Queue

```
/**
 * LinkedQueue represents a linked implementation of a queue.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

public class LinkedQueue<T> implements QueueADT<T>
{
   private int count;
   private LinearNode<T> front, rear;

   /**
    * Creates an empty queue.
    */
   public LinkedQueue()
   {
      count = 0;
      front = rear = null;
   }

   /**
    * Adds the specified element to the rear of this queue.
    *
    * @param element  the element to be added to the rear of this queue
    */
   public void enqueue (T element)
   {
      LinearNode<T> node = new LinearNode<T>(element);

      if (isEmpty())
         front = node;
      else
         rear.setNext (node);

      rear = node;
      count++;
   }

   /**
    * Removes the element at the front of this queue and returns a
    * reference to it. Throws an EmptyCollectionException if the
    * queue is empty.
    *
    * @return                           the element at the front of this queue
    * @throws EmptyCollectionException  if an empty collection exception occurs
    */
   public T dequeue() throws EmptyCollectionException
   {
        if (isEmpty())
            throw new EmptyCollectionException ("queue");

        T result = front.getElement();
        front = front.getNext();
        count--;

        if (isEmpty())
            rear = null;

        return result;
    }

    /**
     * Returns a reference to the element at the front of this queue.
     * The element is not removed from the queue.  Throws an
     * EmptyCollectionException if the queue is empty.
     *
     * @return                                a reference to the first element in
     *                                        this queue
     * @throws EmptyCollectionsException      if an empty collection exception occurs
     */
    public T first() throws EmptyCollectionException
    {
        // left as programming project
    }

    /**
     * Returns true if this queue is empty and false otherwise.
     *
     * @return  true if this queue is empty and false if otherwise
     */
    public boolean isEmpty()
    {
         // left as programming project
    }

    /**
     * Returns the number of elements currently in this queue.
     *
     * @return  the integer representation of the size of this queue
     */
    public int size()
    {
         // left as programming project
    }

    /**
     * Returns a string representation of this queue.
     *
     * @return  the string representation of this queue
     */
    public String toString()
    {
        // left as programming project
    }
}
```



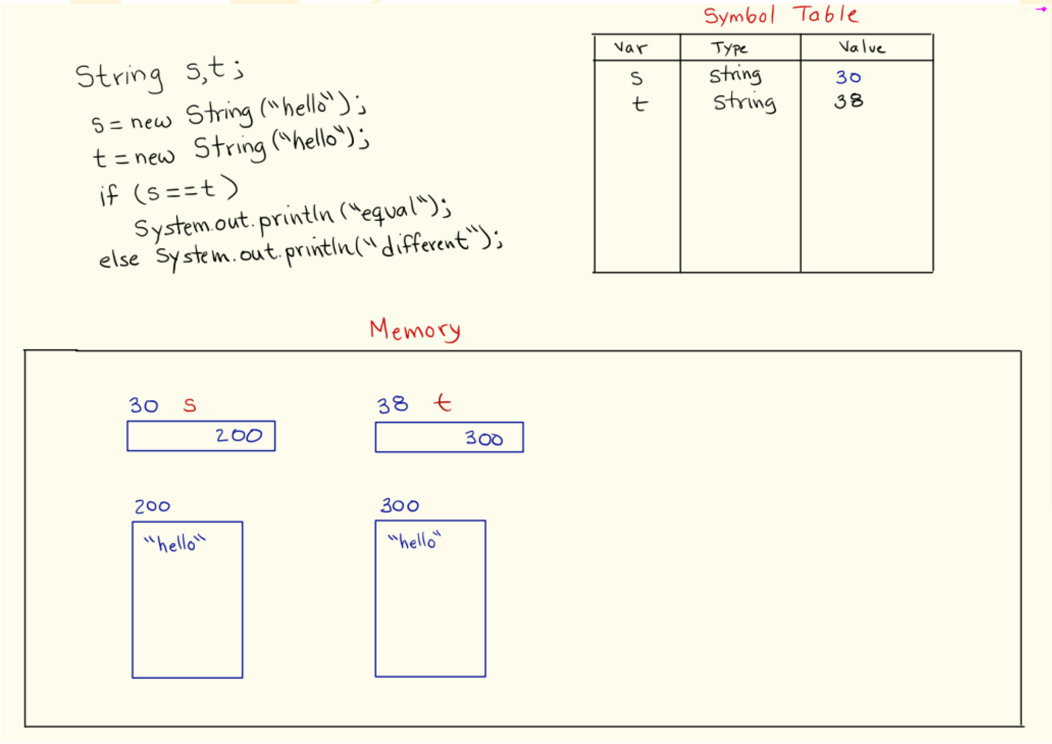
Scope of a Variable

What values will be printed by the following Java program?

```
public class Test {
   private int var1;
   private int i;
   public Test() {
      var1 = 7;
      i = 10;
   }
   private void algo1 (int j) {
      i = j;
      j = 14;
   }
   private void algo2 (int i) {
      i = 20;
      var1 = 100;
   }
```

```
public void algo3() {
      int var1 = 2;
      int j = 4;
      for (int i = 1; i < 3; ++i)
         var1 = var1 + i;
      int k = i;
      algo2(5);
      System.out.println(i);
      algo1(j);
      System.out.println(j+","+i+","+var1);
   }
   public static void main (String[] args) {
      Test t = new Test();
      t.algo3();
   }
}
```

Figure 9.2: The scope of variables

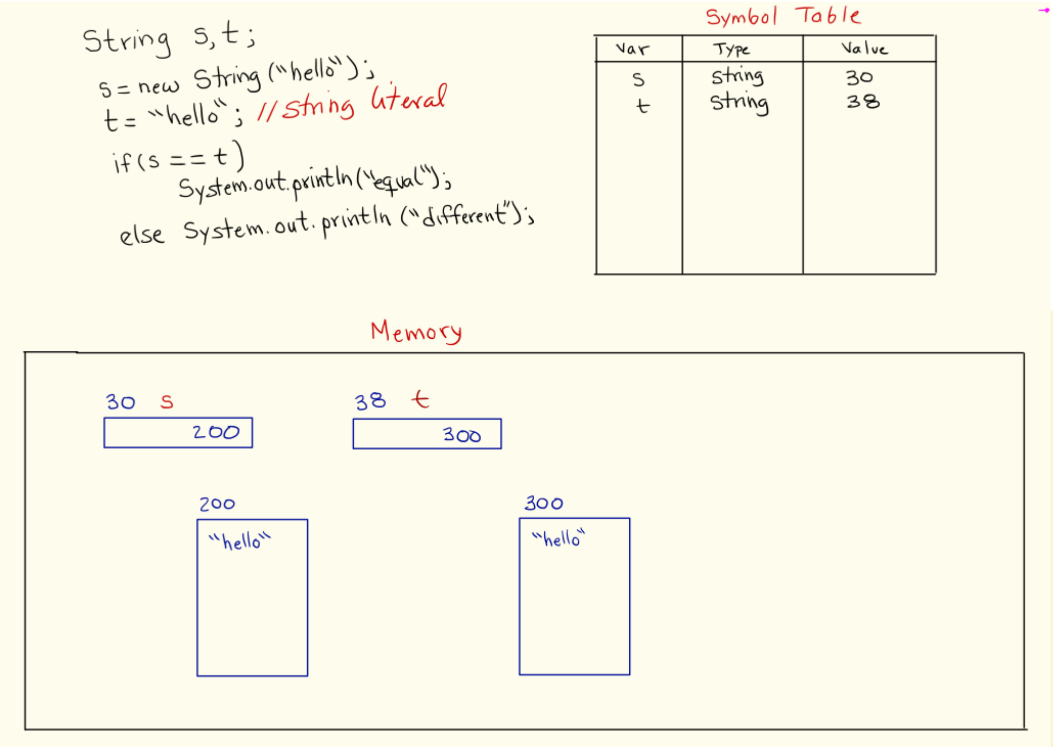


```
String s,t;
 s = new String ("hello");
 t = new String ("hello");
 if (s==t)
    System.out.println ("equal");
 else System.out.println("different");
```

Symbol Table

Var	Type	Value
s	String	30
t	String	38

Figure 9.3: Memory1

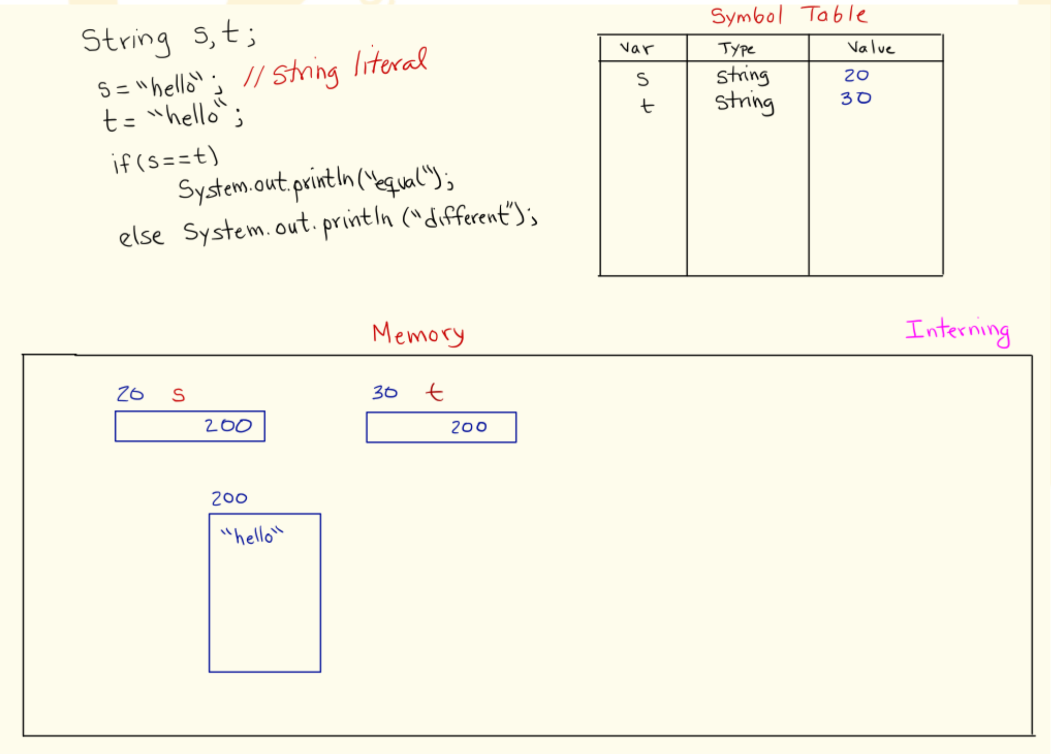


```
String s,t;
 s = new String ("hello");
 t = "hello"; //string literal
 if(s == t)
     System.out.println("equal");
 else System.out.println("different");
```

Symbol Table

Var	Type	Value
s	string	30
t	string	38

Figure 9.4: Memory1

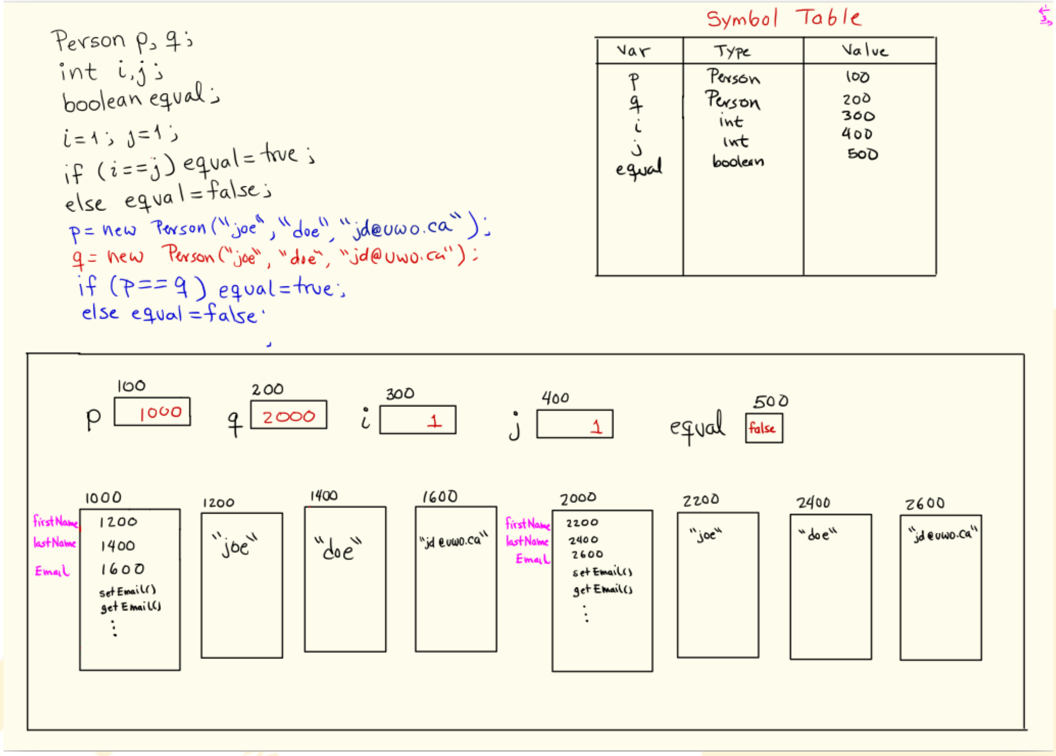


```
String s,t;
 s = "hello"; //string literal
 t = "hello";
 if(s==t)
     System.out.println("equal");
 else System.out.println("different");
```

Symbol Table

Var	Type	Value
s	string	20
t	string	30

Figure 9.5: Memory1

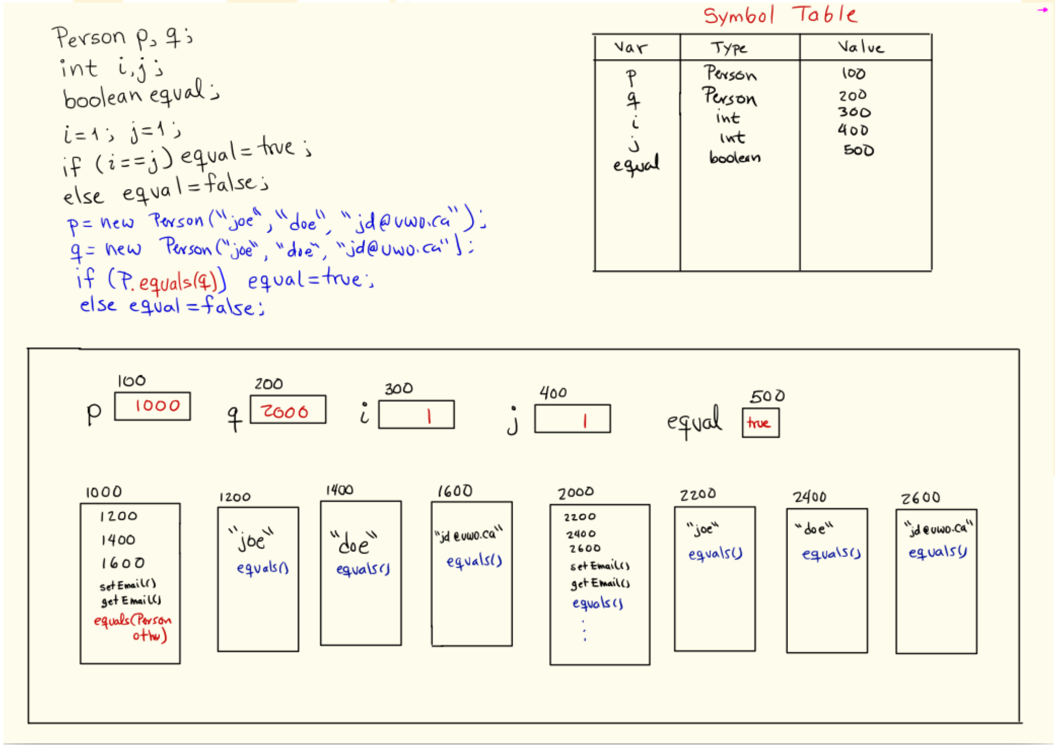


```
Person p, q;
int i,j;
boolean equal;
i=1; j=1;
if (i==j) equal=true;
else equal=false;
p= new Person("joe","doe","jd@uwo.ca");
q= new Person("joe", "doe", "jd@uwo.ca");
if (p==q) equal=true;
else equal=false;
```

Symbol Table

Var	Type	Value
p	Person	100
q	Person	200
i	int	300
j	int	400
equal	boolean	500

Figure 9.6: Memory1



```
Person p, q;
int i,j;
boolean equal;
i=1; j=1;
if (i==j) equal=true;
else equal=false;
p= new Person("joe","doe", "jd@uwo.ca");
q= new Person("joe", "doe", "jd@uwo.ca");
if (P.equals(q)) equal=true;
else equal=false;
```

Symbol Table

Var	Type	Value
p	Person	100
q	Person	200
i	int	300
j	int	400
equal	boolean	500

Figure 9.7: Memory1