1. (a) import java.util.Vector;

/**
 * Trees with integer internal labels and lists of subtrees.
 * The {link TreeNode list of subtrees} is implemented as a linked list.
 * @author <a href="mailto:grant@liverpool.ac.uk">Grant Malcolm</a>
 * @version 1.0
 */
public class RoseTree
{
    /**
     * The internal label of the tree.
     */
    private int value;

    /**
     * A linked list of trees. Null for the empty list.
     */
    private TreeNode subtrees;

    /**
     * Creates a new <code>RoseTree</code> instance.
     * @param i an <code>int</code> value
     * @param f a <code>TreeNode</code> value
     */
    public RoseTree(int i, TreeNode f)
    {
        value = i;
        subtrees = f;
    }

    /**
     * Return a list of all the leaves in the tree.
     * A leaf is a tree with an empty list of subtrees.
     * @return a vector containing all the leaves of the tree
     */
    public Vector<Integer> getLeaves()
    {
        // add all the leaves to the empty list
        return addLeaves(new Vector<Integer>());
    }

    /**
     * Add all the leaves of the tree to a given vector.
     * @param v the vector to add the leaves to
     * @return the vector with all the leaves added
     */
private Vector<Integer> addLeaves(Vector<Integer> v)
{
    // add all leaves to the vector
    if (subtrees == null)
    {
        // then this tree is a leaf: add the internal label
        v.addElement(value);
    }
    else
    {
        // otherwise, get the leaves from all the subtrees
        TreeNode f = subtrees; // used to traverse the linked list of subtrees
        while (f != null) // not at the end of the list ...
        {
            // ... so add the leaves from the current node
            f.theTree.addLeaves(v);
            // move on to the next node
            f = f.next;
        }
        // all leaves have been added to v
        return v;
    }
}

/**
 * Minimal linked-list implementation of TreeNode.
 * As a linked list, a TreeNode is a node, whose fields store:
 * <ul>
 * <li> a RoseTree (the "value" at the node), and </li>
 * <li> the next TreeNode (a pointer to the next node in the list). </li>
 * </ul>
 */
class TreeNode
{
    /**
     * The tree at the current node.
     */
    RoseTree theTree;

    /**
     * The next node.
     */
    TreeNode next;

    /**
     * Constructor:
     * tree to store at current node, and the remainder of the list.
     */
    TreeNode(RoseTree t, TreeNode f)
    {
        theTree = t;
        next = f;
    }
}
(b) The if-clause in `addLeaves` adds the internal label to the list if the Rose-Tree is a leaf (the `subtrees` field is `null`), which corresponds to the equation

$$\text{eq} \quad \text{addLeaves}(\text{roseTree}(I, \text{null}), L) = \text{add}(L, I) .$$

The else-clause implements the ‘loop’ operation:

$$\text{var} \ F : \text{TreeNode} . \ (\text{error in the question: sorry!})$$

$$\text{cq} \quad \text{addLeaves}(\text{roseTree}(I, \_), L) = \text{loop}(F, L) \quad \text{if} \ F \neq \text{null} .$$

While the variable `f` is not `null`, the leaves of `f.theTree` are added to the vector, corresponding to the equation

$$\text{eq} \quad \text{loop}(\text{treeNode}(R, T), L) = \text{loop}(T, \text{addLeaves}(R, L)) .$$

Finally, when `f` is `null`, the vector is returned, corresponding to the equation

$$\text{eq} \quad \text{loop}(\text{null}, L) = L .$$

2.(a–c) import java.util.Vector;

```java
class Pair<A,B> {
    private A first;
    private B second;
    Pair(A a, B b) {
        first = a;
        second = b;
    }
    A getFirst() {
        return first;
    }
    B getSecond() {
        return second;
    }
}

interface PairFunction<A,B> {
    public Pair<A,B> compute(A a);
}

// bonus example
class DivByTwo implements PairFunction<Integer,Integer> {
    public Pair<Integer,Integer> compute(Integer i) {
        return new Pair<Integer,Integer>(i/2, i%2);
    }
}
```
public class PairFun
{
    static <A,B> PairFunction<A,Vector<B>> iterate(final PairFunction<A,B> f,
            final int n)
    {
        return new PairFunction<A,Vector<B>>()
            {
                public Pair<A,Vector<B>> compute(A a)
                {
                    Vector<B> values = new Vector<B>();
                    A val = a;
                    Pair<A,B> fVal;
                    int count = n;
                    while (count > 0)
                    {
                        fVal = f.compute(val);
                        val = fVal.getFirst();
                        values.addElement(fVal.getSecond());
                        count--;
                    }
                    return new Pair<A,Vector<B>>(val, values);
                }
            };
    }

    // bonus test
    public static void main(String[] args)
    {
        PairFunction<Integer,Vector<Integer>> digitize =
                iterate(new DivByTwo(), 4);
        Vector<Integer> digits = digitize.compute(13).getSecond();
        int len = digits.size();
        for ( ; len-- > 0; )
        {
            System.out.print(digits.elementAt(len));
        }
    }
}

(d) i. interface PairFunction
{
    public Pair compute(Object a);
}

ii. static PairFunction iterate(final PairFunction f, final int n)
{
    return new PairFunction()
    {
        public Pair compute(Object a)
        {
            Vector values = new Vector();
            Object val = a;
            Pair fVal;
            int count = n;
            };
}
while (count > 0)
{
    fVal = f.compute(val);
    val = fVal.getFirst();
    values.addElement(fVal.getSecond());
    count--;
}
return new Pair(val, values);
}

(note that the fields in the ‘erased’ version of Pair both have type Object).

3. (a) The method-call stack tracks the nesting of method-calls when the interpreter executes code. Whenever a method is called, that method’s name is pushed onto the stack; once the method is fully evaluated, its name is popped off the stack.

(b) Execution begins with the main method: main is pushed onto the empty method call stack. The Stack constructor is pushed onto the stack when the stack instance is created (method-call stack is: Stack(init) main), then popped off when the constructor call is complete. Then push is pushed onto the stack, while the code in the push method is evaluated, then popped off again. When remove is called, the method-call stack is (remove main), and the code in that method is executed. First iteration of the loop calls pop (method-call stack is pop remove main), which makes the Stack instance empty (i.e., its top field is 0). On the next iteration, pop is called again (method-call stack is pop remove main), and when the expression values[--top] is evaluated, an ArrayIndexOutOfBoundsException is thrown. This isn’t caught in remove or the main method, so the interpreter reports the exception and halts.

(c) Unchecked exceptions are subclasses of RuntimeException, and correspond to semantic errors (bugs); checked exceptions are all other subclasses of Exception, and the Java compiler will enforce that methods that might throw such exceptions either catch the exceptions, or advertise in the method signature that the method throws them.

(d) class EmptyStackException extends Exception{ }
    ...
    public int pop() throws EmptyStackException
    {
        if (top <= 0) throw new EmptyStackException();
        else return values[--top];
    }

    The main method and remove must either catch or advertise EmptyStackExceptions.

(e) Yes. It is true on initialisation, and all public methods either preserve this property or throw an EmptyStackException (the value of top is only decreased in pop and remove, where an exception is thrown if this would make the value less than 0).
4. (a) Interference is when two or more threads cause data corruption in a shared resource.

(b) One thread might evaluate the test (stack.isEmpty()) and get result false (if one value is in the stack); it’s possible the time-slicer will halt that thread’s execution before it starts to evaluate the else-clause; the other thread might then pop the stack, making it empty, and it’s possible the time-slicer will then recommence execution of the first thread while the stack is still empty, causing an EmptyStackException to be thrown.

(c) (this should have been (b)!) 

(d) The keyword synchronized causes the Java interpreter to place a monitor on the synchronized code.Notionally, a ‘key’ is associated with the code (which is called a critical section). Before a thread can execute that code, it must obtain the key, which it keeps until execution of the critical section terminates, whereupon the thread returns the key (thus allowing other threads to execute the critical section). If the key associated with the critical section is unique, this ensures mutual exclusion of threads in the critical section.

(e) One key is associated with the synchronized block; for synchronized methods, each instance of the class has one key that applies to all synchronized methods in the class.

(f) Synchronize the if-then-else command in the run method, using a static field in the ThreadTest class as a key-keeper.

5. (a) An abstract data type is a language-independent specification of an abstract set of data values, together with a number of operations on those values.

(b) mod SORTED is

    protecting INT .

    sort SortedList .

    op nil : -> SortedList .
    op add : Int SortedList -> SortedList .
    op insert : Int SortedList -> SortedList .
    op merge : SortedList SortedList -> SortedList .

    vars I J : Int .
    var L : SortedList .

    eq insert(I, nil) = add(I, L) .
    cq insert(I, add(J, L)) = add(I, add(J, L)) if I <= J .
    cq insert(I, add(J, L)) = add(J, insert(I, L)) if I > J .

    eq merge(nil, L) = L .
    eq merge(L, nil) = L .
   cq merge(add(I, L), add(J, L')) = add(I, merge(L, add(J, L'))) if I <= J .
    cq merge(add(I, L), add(J, L')) = add(J, merge(add(I, L), L')) if I > J .

*** or:
eq merge(nil, L') = L' .

eq merge(add(I, L), L') = insert(I, merge(L, L')) .

endm

(c) insert(2, insert(1, nil))

= (eq insert(I, nil) = add(I, nil), with var I matching 1)

insert(2, add(1, nil))

= (eq insert(1, add(J,L)) = add(J, insert(I,L)) if I > J, with I matching 2 and J matching 1)

add(1, insert(2, nil))

= (1st equation, again)

add(1, add(2, nil))

BONUS REVISION QUESTIONS!

1. Implement Sorted Lists in Java.

2. What is the main class invariant?