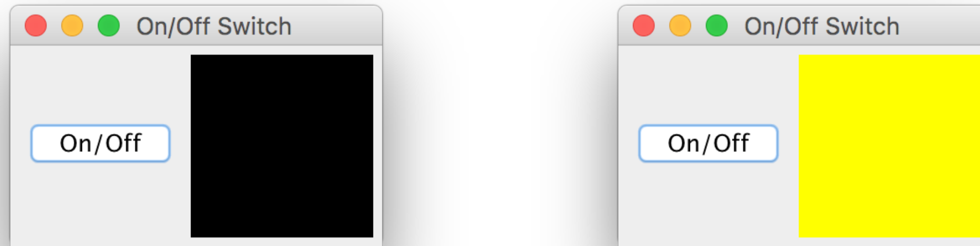
CIS 120 Final Exam 16 December 2015

SOLUTIONS

1. **Java Type System and Swing Programming** (22 points total)

   Consider the Java Swing program shown in Appendix A. It implements a variant of the lightbulb example covered in lecture, and it's two states are shown below.



   a. **(5 points)** Which of the following are *supertypes* of the `Lightbulb` class? Mark *all* correct answers.

      ☐ `OnOff` ☒ `JComponent` ☒ `Container` ☐ `JFrame` ☒ `Object`

   b. **(3 points)** True or False: __`False`__ Because of subtyping, we can change they type declaration on line 24 to the following (leaving the rest of the program untouched), without causing a compile-time error.

```
final Component frame = new JFrame("On/Off Switch");
```

The `Lightbulb` class is fine for this demo code, but suppose that we wanted to allow programmers to create lightbulbs with dimensions other than the hard-coded constant `100` used on lines 13 and 17.

   d. **(3 points)** True or False: __`False`__ It is possible to subclass `Lightbulb`, overriding only the `paintComponent` and `getPreferredSize` methods to create a new class that provides a different size bulb, while otherwise retaining the same functionality (i.e. the bulb still turns on and off when its `flip` is called). *Hint: consider the visibility of the* `isOn` *field.*

   e. **(5 points)** An alternative to inheritance and overriding is to refactor the `Lightbulb` class itself to provide more flexibility for its users. *Briefly* (!) describe how you could alter the `Lightbulb` class to allow `Lightbulb` clients to create bulbs of multiple different sizes, without introducing any new classes:

      ANSWER: Add a field (or fields) to store the size and a constructor that sets that field appropriately. Then modify the `paintComponent` and `getPreferredSize` to use the size field instead of the constant.

f. (3 points) There are five occurrences of the **`new`** keyword in the `run` method of the `OnOff` class. Circle the line number of the one that corresponds to the use of an *anonymous inner class*.

Line number: 24 26 29 32 [34]

g. (3 points) Note that none of the `OnOff` code directly invokes the `paintComponent` method of the `LightBulb` class. Which of the following explanations best describes why that is unnecessary? (Choose one)

☐ The `@Override` directive (on line 6) causes the `Lightbulb` class to overwrite the `JComponent`'s class table entry for `paintComponent` with its own code.

☐ The `Lightbulb` `paintComponent` method is called as a static proxy for a `JComponent` reference delegate object.

☒ The `Lightbulb` `paintComponent` method is called via dynamic dispatch from somewhere inside the Swing library, where the `bulb` object is treated as a `JComponent`.

☐ The `Lightbulb` `paintComponent` method is invoked by using `instanceOf` and a type cast operation from somewhere inside the Swing library.

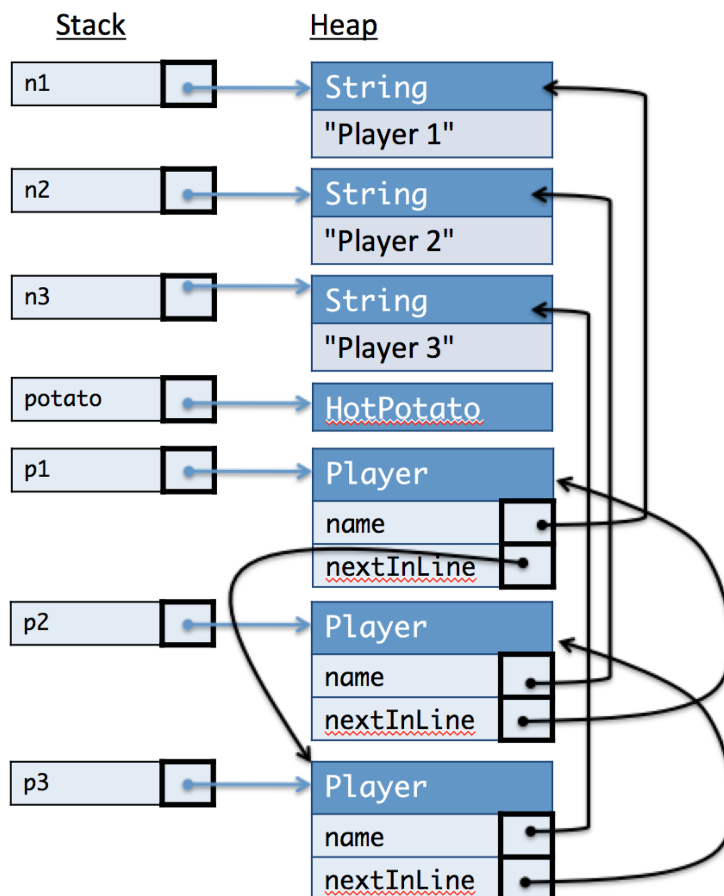
2. **Java Semantics: Exceptions and ASM** (18 points total)

This problem makes use of the class definitions for a game called "hot potato," which is provided for you in Appendix B.

Suppose the following code is run in the workspace of the Java Abstract Stack Machine.

```
String n1 = "Player 1";
String n2 = "Player 2";
String n3 = "Player 3";
HotPotato potato = new HotPotato();
Player p1 = new Player(n1, null);
//---------------START--------------------
Player p2 = new Player(n2, null);
Player p3 = new Player(n3, p2);
p1.nextInLine = p3;
p3.nextInLine.nextInLine = p1;
//---------------HERE---------------------
```

(12 points) The stack and heap diagram that corresponds to the point in the execution marked START is shown below. (We have omitted the fields of the `Potato` object for simplicity.) Extend this diagram so that it displays the stack and heap at the point when execution terminates. *Note: There is no need to show the workspace or the class table!*



(6 points) Now suppose we run the following code, starting from the state marked HERE (above).

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

What happens when this code is run? Circle the correct behavior from the choices below.

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

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

c. 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!
```

d. THIS ONE! 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!
```

e. 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!
```

f. 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.

3. Java Programming with Collections (46 points total)

A *bag* (sometimes also called a *multiset*) is an *unordered* collection of values that permits duplicates. Intuitively, a bag is a set that can contain multiple occurrences of the same element. For example, using set-like notation, we might write the bag containing two '1's and one '2' as $\{1, 1, 2\}$ or equivalently as $\{1, 2, 1\}$ or $\{2, 1, 1\}$.

In this problem, you will use the design process to implement (some parts of) a Java `Bag<E>` class.

Step 1 (Understand the problem) There is nothing to do for this step; your answers below will indicate your understanding.

Step 2 (Determine the interface) A `Bag<E>` object implements the `Collection<E>` interface, the relevant parts of which are given in Appendix C.

In addition to the standard collections methods, a bag also provides a method called `getCount(E e)`, which returns a non-negative integer indicating the number of times the element `e` occurs in the multiset.

Step 3 (Write test cases) Below are several example test cases for the `add`, `size`, and `contains` methods of the `Bag<E>` implementation. (No need to do anything but understand them.)

```
@Test
public void sizeEmpty() {
  Bag<Integer> b = new Bag<Integer>();
  assertEquals(0, b.size());
}

@Test
public void size1() {
   Bag<Integer> b = new Bag<Integer>();
   b.add(1);
   assertEquals(1, b.size());
}

@Test
public void size11() {
  Bag<Integer> b = new Bag<Integer>();
  b.add(1);
  b.add(1);
  assertEquals(2, b.size());
}

@Test
public void containsEmpty() {
  Bag<Integer> b = new Bag<Integer>();
  assertFalse(b.contains(1));
}

@Test
public void contains1() {
  Bag<Integer> b = new Bag<Integer>();
  b.add(1);
  assertTrue(b.contains(1));
}
```

a. (5 points) Complete these three distinct tests cases for the `remove` method by filling in the blank with either `True` or `False` indicating whether the assertion should succeed or fail:

```
@Test
public void removeEmpty() {
  Bag<Integer> b = new Bag<Integer>();
  assertFalse(b.remove(1));
}
```

```
@Test
public void remove1() {
  Bag<Integer> b = new Bag<Integer>();
  b.add(1);
  assertTrue(b.remove(1));
  assertFalse(b.contains(1));
}

@Test
public void remove11() {
  Bag<Integer> b = new Bag<Integer>();
  b.add(1);
  b.add(1);
  assertTrue(b.remove(1));
  assertTrue(b.contains(1));
}
```

b. (4 points) Now complete this test case by filling in the blanks such that the test case should succeed:

```
@Test
public void countTest() {
  Bag<Integer> b = new Bag<Integer>();
  b.add(1);
  b.add(2);
  assertEquals(1, b.getCount(1));
  assertEquals(1, b.getCount(2));
  b.add(1);
  b.remove(3);
  assertEquals(2, b.getCount(1));
  assertEquals(3, b.size());
}
```

Step 4 (Implementation) To implement the `Bag<E>` class, we must decide how to represent the collection using other data structures, plus invariants. In our design, we will represent the internal state of the `Bag<E>` class using a `Map<E,Integer>` object. The idea is to associate with each element `e` a count of the number of times that `e` occurs in the bag. (Appendix D describes the `Map` interface.)

Let us introduce the notation $[k_1 \mapsto v_1 \ldots k_n \mapsto v_n]$ as shorthand for a `Map<K,V>` object `m` such that `m.get(`k_i`)` returns v_i. For example, we could write

$$[\texttt{"a"} \mapsto 2, \texttt{"b"} \mapsto 1]$$

for the `Map<String,Integer>` object `m` obtained by doing:

```
Map<String,Integer> m = new Map<String,Integer>();
m.put("a", 2);
m.put("b", 1);
```

This object `m` would be a suitable representation of the bag $\{\texttt{"a"}, \texttt{"a"}, \texttt{"b"}\}$, since it has two “a”s and one “b”.

In order to conveniently implement the `size()` method required by `Collection<E>`, we will also keep track of a `size` field as part of the bag implementation. We thus arrive at this partial implementation the bag class:

```
public class Bag<E> implements Collection<E> {
   private Map<E,Integer> bag; // representation
   private int size;       // number of elements

   public Bag() {
      bag = new TreeMap<E,Integer>();
      size = 0;
   }

   @Override
   public int size() {
      return size;
   }

   @Override
   public boolean isEmpty() {
      return size == 0;
   }

   // continued
```

(Note: There are no questions on this page.)

Of course, not *every* `bag` object of type `Map<E,Integer>` is a good representation; for example there should never be a negative number of elements in the bag, so we need an invariant to rule out such incorrect maps. Which invariant we choose will affect the difficulty of implementing the methods of the class.

Here are two correct invariants that we might use to rule out invalid maps.

INV1: If `bag.containsKey(e)` then `bag.get(e)` ≥ 0.

INV2: If `bag.containsKey(e)` then `bag.get(e)` > 0.

c. (2 points) Which of these invariants does the implementation of `contains` given below use?

```
@Override
public boolean contains(Object o) {
   return bag.containsKey(o);
}
```

☐ only INV1 ☒ only INV2 ☐ it works with both INV1 and INV2

d. (2 points) Which of these invariants does the implementation of `getCount` given below use?

```
public int getCount(E e) {
   if (bag.containsKey(e)) {
      return bag.get(e);
   } else {
      return 0;
   }
}
```

☐ only INV1 ☐ only INV2 ☒ it works with both INV1 and INV2

e. (3 points) Now consider overriding the `equals` method for `Bag<E>` objects: two bags should be considered equal if, for every element `e` they contain the same number of occurrences of `e`. So $\{1, 1, 2\}$ and $\{1, 2, 1\}$ and $\{2, 1, 1\}$ are equivalent. One of the two invariants **INV1** or **INV2** make it much simpler to implement the `equals` method. Briefly (!) explain which one and why:

ANSWER: Invariant INV2 makes it easier because we can use the `Map` implementation of `equals`. The presence of "0" values in the map means two different maps with different keys might represent the same bag: e.g. $[$`"a"` $\mapsto 2,$ `"b"` $\mapsto 0]$ and $[$`"a"` $\mapsto 2,$ `"c"` $\mapsto 0]$ both represent the same bag $\{$`"a"`, `"a"`$\}$.

f. (2 points) We must also maintain the appropriate relationship between the `size` field and the `bag` map. Which of the following invariants correctly expresses that relationship? (Mark one)

☒ If `bag` is the map $[k_1 \mapsto v_1, \ldots, k_n \mapsto v_n]$ then `size` = $v_1 + ... + v_n$.

☐ If `bag` is the map $[k_1 \mapsto v_1, \ldots, k_n \mapsto v_n]$ then `size` = $k_1 + ... + k_n$.

Consider the following buggy(!), but almost correct, implementation of the `add` method.

```
@Override
public boolean add(E e) {
  if (bag.containsKey(e)) {
     Integer count = bag.get(e);
     bag.put(e, count+1);
  } else {
     bag.put(e,1);
     size++;
  }
  return true;
}
```

g. (3 points) One of the example test cases that we provided *fails* with this implementation of `add`. Which one? (The other methods are correct and their code is as shown earlier.)

`sizeEmpty`   `size1`   [`size11`]   `containsEmpty`   `contains1`

h. (3 points) In one sentence, describe how to fix the bug:

ANSWER: Increment the `size` field in both branches. (Or increment it outside of the conditional statement altogether.)

i. (12 points) Now implement the `remove` method, the Javadocs for which are:

> **`boolean remove(Object o)`**
> Removes a single instance of the specified element from this collection, if it is present. Returns true if this collection contained the specified element (or equivalently, if this collection changed as a result of the call) and false otherwise.

Use representation invariant **INV2** and the `size` invariant indicated above. Make sure that your implementation passes the test cases. We have reproduced the class declaration and fields here so that you can see them when writing this code. *Hint: pay careful attention to the types, you will need to use a type cast at some point.*

```
public class Bag<E> implements Collection<E> {
   private Map<E,Integer> bag; // representation
   private int size;       // number of elements


   @Override
   public boolean remove(Object o) {
      if (bag.containsKey(o)) {
         Integer cnt = bag.get(o);
         cnt = cnt - 1;
         if (cnt > 0) {
            bag.put((E)o, cnt);
         } else {
            bag.remove(o);
         }
         size = size - 1;
         return true;
      } else {
        return false;
      }
   }
```

j. (10 points) There are alternative designs that represent bags using other structures. Which of the following data structures could *also* be used to implement the internal state of a `Bag<E>` object? For each representation type, give an example of how to represent the instance for a `Bag<String>` value {`"a"`, `"a"`, `"b"`} or write "Not possible" if it can't be used. Note that the representation should work *generically* for any type `E`, not just when `E` happens to be `String`. For lists use OCaml notation like `[1;2;3]`, for sets use set notation like $\{1, 2, 3\}$, for maps use the notation described above. We have done the first one for you.

`Bag<E>` Repr. type	Representation of {`"a"`, `"a"`, `"b"`} in `Bag<String>`
`Map<E,Integer>`	[`"a"` $\mapsto 2$, `"b"` $\mapsto 1$]
`LinkedList<E>`	[`"a"`; `"a"`; `"b"`]
`LinkedList<Integer>`	Not Possible
`Set<E>`	Not Possible
`Map<Integer,E>`	Not Possible
`Map<Integer,Set<E>>`	[$1 \mapsto$ {`"b"`}, $2 \mapsto$ {`"a"`}]

4. OCaml Lists, Trees, and Recursion (22 points total)

(6 points) 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]
```

a. What is the value computed for `ans` in the code above?

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

b. 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`? (Choose one.)

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

☐
```
let 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)
```

(6 points) 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]
```

a. What is the value computed for `ans` in the code above?

Answer: `ans = 5`

b. 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`? (Choose one.)

☐
```
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 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
```

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

(6 points) Consider this tree function:

```
let rec f (t: int tree) : int =
  begin match t with
    | Empty -> 0
    | Node(lt, x, rt) -> (f lt) + x + (f rt)
  end
let leaf = Node(Empty, 2, Empty)
let ans = f (Node(leaf, 3, leaf))
```

i. What is the value computed for `ans` in the code above?
Answer: `ans = 7`

ii. Which of the following correctly implements the function `f` using `tree_fold`? (Choose one.)

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

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

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

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

5. Mutable data structures in OCaml (12 points) (3 points each)

Recall the OCaml type of records with a single mutable component

```
type 'a ref = { mutable contents : 'a }
```

Can you make the following OCaml expressions evaluate to 42? Circle the correct answer.

a.
```
let x = { contents = 0 } in
let f () : int = x.contents in
???
f ()
```

What line could replace ??? to make this expression evaluate to 42?

i. `let x = 42 in`

ii. `let x = { contents = 42 } in`

iii. `x.contents <- 42;`

iv. `x.contents.contents <- 42;`

v. This expression cannot be made to evaluate to 42

b.
```
let x = { contents = 0 } in
let f () : int = x.contents in
let x = { contents = x } in (* <– Note the two occurrences of x *)
???
f ()
```

What line could replace ??? to make this expression evaluate to 42?

i. `let x = 42 in`

ii. `let x = { contents = 42 } in`

iii. `x.contents <- 42;`

iv. `x.contents.contents <- 42;`

v. This expression cannot be made to evaluate to 42

c.
```
type obj = { f : int -> int }

let new_obj (k:int) : obj =
  let x = { contents = k } in
  { f = fun (y:int) ->
        let z = x.contents in
         x.contents <- y;
         z }
let ans =
  let o = new_obj 3 in
  ???
  o.f 27
```

What line could replace ??? to make the `ans` expression evaluate to 42?

- **i.** `let x = 42 in`
- **ii.** `let p = new_obj 42 in`
- **iii.** `x.contents <- 42;`
- **iv.** `let q = o.f 42 in` (boxed)
- **v.** This expression cannot be made to evaluate to 42

d.
```
type obj = { f : int -> int }
let new_obj (k:int) : obj =
  let x = { contents = k } in
  { f = fun (y:int) ->
       x.contents + y }
let ans =
  let o = new_obj 6 in
  let z = ??? in
  o.f 27
```

What could replace ??? to make the `ans` expression evaluate to 42?

- **i.** `()`
- **ii.** `42`
- **iii.** `x.contents <- 15`
- **iv.** `o.f 15`
- **v.** This expression cannot be made to evaluate to 42 (boxed)

A Java GUI Lightbulb Program

This code is for use with problem 1. Appendix E describes relevant parts of the Swing libraries.

```
1   // imports omitted to save space (this code compiles)
2
3   class LightBulb extends JComponent {
4       private boolean isOn = false;
5
6       @Override
7       public void paintComponent(Graphics gc) {
8           if (isOn) {
9               gc.setColor(Color.YELLOW);
10          } else {
11              gc.setColor(Color.BLACK);
12          }
13          gc.fillRect(0, 0, 100, 100);
14      }
15
16      @Override
17      public Dimension getPreferredSize() { return new Dimension(100,100); }
18
19      public void flip() { isOn = !isOn; }
20  }
21
22  class OnOff implements Runnable {
23    public void run() {
24       JFrame frame = new JFrame("On/Off Switch");
25
26       JPanel panel = new JPanel();
27       frame.add(panel);
28
29       JButton button = new JButton("On/Off");
30       panel.add(button);
31
32       final LightBulb bulb = new LightBulb();
33       panel.add(bulb);
34       button.addActionListener(new ActionListener() {
35              @Override
36              public void actionPerformed(ActionEvent e) {
37                  bulb.flip();
38                  bulb.repaint();
39              }
40       });
41       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
42       frame.pack();
43       frame.setVisible(true);
44    }
45
46    public static void main(String[] args) {
47       SwingUtilities.invokeLater(new OnOff());
48    }
49  }
```

B Java "Hot Potato" Program

This code is provided for use with problem 2.

```
class HotPotato extends Throwable {
}

class Player {
   public static Player thrower = null;

   Player nextInLine;
   String name;

   Player(String name, Player target) {
      this.name = name;
      this.nextInLine = target;
   }

   void getsIt(HotPotato potato) throws HotPotato {
      System.out.println(name + " gets the potato.");
      if (this == thrower) {
         System.out.println(name + " throws the potato.");
         throw potato;
      } else {
         try {
            System.out.println(name + " passes the potato.");
            nextInLine.getsIt(potato);
         } catch (HotPotato p) {
            System.out.println(name + " catches the potato and throws it.");
            throw p;
         }
      }
   }
}
```

C Excerpt from the Collections Framework (Lists and Sets)

```
interface Collection<E> extends Iterable<E> {
  public boolean add(E o);
  // Ensures that this collection contains the specified element
  // Returns true if this collection changed as a result of the call.
  // (Returns false if this collection does not permit duplicates
  // and already contains the specified element.)

  public boolean contains(Object o);
  // Returns true if this collection contains the specified element.

  public int size();
  // Returns the number of elements in this collection.

  public boolean remove(Object o);
  // Removes a single instance of the specified element from this
  //  collection, if it is present. Returns true if this collection
  // contained the specified element (or equivalently, if this collection
  // changed as a result of the call) and false otherwise.

  // (Other methods omitted.)
}
```

D Excerpt from the Collections Framework (Maps)

```
interface Map<K,V> {


  public V get(Object key)
  // Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
  // More formally, if this map contains a mapping from a key k to a value v
  // such that (key==null ? k==null : key.equals(k)), then this method returns
  // v; otherwise it returns null. (There can be at most one such mapping.)

  public V put(K key, V value)
  // Associates the specified value with the specified key in this map

  public V remove(Object key)
  // Removes the mapping for a key from this map if it is present.
  // Returns the value to which this map previously associated the
  // key, or null if the map contained no mapping for the key.

  public int size()
  // Returns the number of key-value mappings in this map.

  public boolean isEmpty()
  // Returns true if this map contains no key-value mappings.

  public Set<K> keySet()
  // Returns a Set view of the keys contained in this map. The set is backed
  // by the map, so changes to the map are reflected in the set, and
  // vice-versa.

  public boolean containsKey(Object key)
  // Returns true if this map contains a mapping for the specified key.

}


class TreeMap<K,V> implements Map<K,V> {

  public TreeMap()
  // constructor
  // Constructs a new, empty tree map, using the natural ordering of its keys.

  // ... methods specified by interface
}
```

E Excerpt from the Swing Library

```
interface MouseListener extends EventListener {
  public void mouseClicked(MouseEvent e)
  public void mouseEntered(MouseEvent e)
  public void mouseExited(MouseEvent e)
  public void mousePressed(MouseEvent e)
  public void mouseReleased(MouseEvent e)
}

interface MouseMotionListener {
  public void mouseDragged(MouseEvent e)
  public void mouseMoved(MouseEvent e)
}

public class MouseAdapter implements MouseListener, MouseMotionListener {
  // methods specified by interfaces
}

public class Component {
  public void addMouseListener(MouseListener l)
  // Adds the specified mouse listener to receive mouse events from this component.

  public void addMouseMotionListener(MouseMotionListener l)
  // Adds the specified mouse motion listener to receive mouse motion events from this component.

  public Dimension getPreferredSize()
  // Returns the preferred size of this container.

  public void repaint()
}

public class Container extends Component {

  public Component add(Component comp)
  // Appends the specified component to the end of this container.
}

public class JComponent extends Container {

  public Dimension getPreferredSize()
  // If the preferredSize has been set to a non-null value just returns it.

  protected void paintComponent(Graphics g)

}
```

```
class JLabel extends JComponent {
  public JLabel(String text)
  // Constructor: Creates a JLabel instance with the specified text.

  public String getText()
  // Returns the text string that the label displays.

  public void setText(String text)
  // Defines the single line of text this component will display.
}

class JPanel extends JComponent {
  public JPanel()
  // Constructor: Creates a new JPanel with a double buffer and a flow layout.

  public void add(JComponent c)
  // Appends the specified component to the end of this container.

  public void setLayout(LayoutManager mgr)
  // Sets the layout manager for this container.

}

class JButton extends JComponent {

   public JButton(String text)
   // Constructor: Creates a button with text.

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

interface ActionListener {
   public void actionPerformed(ActionEvent e)
   // Invoked when an action occurs
}
```