Diagnostic Exam Answers and Explanations

Part I (Multiple-Choice) Answers and Explanations

Bullets mark each step in the process of arriving at the correct solution.

1. The answer is B.
   - When we first enter the loop, val = 13 and i = 2. The `if` condition is looking for even numbers. Since val + i is odd, we skip the `if` clause. Increment i to 3. i < 7 so we continue.
   - val = 13, i = 3. This time val + i is even, so add 3 to val. val = 16, increment i to 4, i < 7 so we continue.
   - val = 16, i = 4. val + i is even, add 3 to val. val = 19, increment i to 5, i < 7, continue.
   - val = 19, i = 5. val + i is even, add 3 to val. val = 22, increment i to 6, i < 7, continue.
   - val = 22, i = 6. val + i is even, add 3 to val. val = 25, increment i to 7. This time, when we check the loop condition, i is too big, so we exit the loop.
   - val = 25 and that is what is returned.

2. The answer is E.
   - The first `if` condition evaluates to (2 < 5 && 13 < 5), which is false, so we skip to the else clause.
   - The `else` clause has its own `if` statement. The condition evaluates to (2 == 2 && 13 < 2), which is false, so we skip to the `else` clause.
   - result = 2 + 13 = 15, which is what is printed.

3. The answer is D.
   - The `for-each` loop can be read like this: for each integer in numbers, which I am going to call n.
   - The `if` clause is executed only when the element is > 8, so the loop adds all the elements greater than 8 to the sum variable.
   - The elements greater than 8 are 13, 12, and 10, so sum = 35, and that is what the method returns.

4. The answer is C.
   - Option I is correct. The reference variable type `Planet` matches the `Planet` object being instantiated, and the `Planet` class contains a no-argument constructor.
   - Option II is correct. The reference variable type `Planet` is a superclass of the `DwarfPlanet` object being instantiated, and the `DwarfPlanet` class contains a constructor that takes one `String` parameter.
   - Option III is not correct. Although the reference variable type `Planet` is a superclass of the `DwarfPlanet` object being instantiated, the `DwarfPlanet` class does not contain a no-argument constructor. Unlike a method, the constructor of the parent class is not inherited.

5. The answer is A.

- Let's picture the contents of our `ArrayList` in a table. After the 3 adds, we have:

0	1	2
Vampire	Werewolf	Ghost

Setting 0 to Zombie gives us:

0	1	2
Zombie	Werewolf	Ghost

Adding Mummy at position 2 pushes Ghost over one position:

0	1	2	3
Zombie	Werewolf	Mummy	Ghost

Witch gets added at the end:

0	1	2	3	4
Zombie	Werewolf	Mummy	Ghost	Witch

After the remove at index 3, Ghost goes away and Witch shifts over one:

0	1	2	3
Zombie	Werewolf	Mummy	Witch

6. The correct answer is E.

- Options A and B are incorrect. The inner column loop is not necessary and includes more values than along the diagonal to be included in the sum.
- Option C is incorrect. If the grid has odd dimensions, the center value will be counted twice.
- Option D is incorrect. If the grid has even dimensions, one value in the main diagonal will be subtracted out of the sum at the end.
- Option E works correctly. If the grid has odd dimensions, the center value will only be counted once.

7. The answer is C.

- Option A will not compile, because acres is a private instance variable.
- Option B will not compile, because the name of the object is park, not central.
- Option C is correct. The method is called properly and it returns a `boolean`.
- Option D will not compile, because `getAcres` does not take a parameter.
- Option E will not compile, because `hasPlayground` is not a public `boolean` variable.

8. The answer is B.

- The `for` loop goes through the entire string, looking at each character individually. If the character is an `m`, the substrings add the section of the string *before* the `m` to the section of the string *after* the `m`, leaving out the `m`.
- The result is that all the `m`'s are removed from the string, and the rest of the string is untouched.

9. The answer is D.

- This is a recursive method. Let's trace the calls. The parts in *italics* were filled in on the way back up. That is, the calls in the plain type were written top to bottom until the base case returned a value. Then the answers were filled in *bottom to top.*

```
loopy(12) = loopy(15) + 2 = 25 + 2 = 27, which gives us our final answer.
loopy(15) = loopy(18) + 2 = 23 + 2 = 25
loopy(18) = loopy(21) + 2 = 21 + 2 = 23
loopy(21) Base Case! return 21
```

10. The answer is B.

- The compiler looks at the left side of the equals sign and checks to be sure that whatever methods are called are available to variables of that type. Since `Letter` and `ALetter` both contain `toString` methods, the compiler is fine with the code. Option E is incorrect.
- The run-time environment looks at the right side of the equals sign and calls the version of the method that is appropriate for that type.
- `System.out.print(x)` results in an implicit call to the `ALetter toString` method, and prints "a".
- `System.out.print(y)` results in an implicit call to the `BLetter toString` method and prints "b".
- `System.out.print(z)` tries to make an implicit call to a `CapALetter toString` method, but there isn't one, so it moves up the hierarchy and calls the `toString` method of the ALetter class and prints "a".

11. The correct answer is B.

- On entry, word = "APCS" and index = 0.

  `word = word + its substring from 0 to 1, or "A". word = "APCSA".`
- Add 2 to index, index = 2. word.length() is 5, 2 < 5 so continue.

  `word = word + its substring from 2 to 3, or "C". word = "APCSAC".`
- Add 2 to index, index = 4. word.length() is 6, 4 < 6, so continue.

  `word = word + its substring from 4 to 5, or "A". word = "APCSACA".`
- Add 2 to index, index = 6. word.length() is 7, 6 < 7, so continue.

  `word = word + its substring from 6 to 7, or "A". word = "APCSACAA".`
- Add 2 to index, index = 8. wordlength() is 8, 8 is not < 8 so the loop exits.
- Notice that since word is altered in the loop, word.length() is different each time we evaluate it at the top of the loop. You can't just replace it with 4, the length of word at the beginning of the method. You must re-evaluate it each time through the loop.

12. The answer is C.

- Each time through the loop:
  - We are considering element n.
  - We calculate array[n] − array[n − 1] and compare it to array[n] − array[n + 1]. In other words, subtract the element *before* from the nth element, then subtract the element *after* from the nth element. If the *before* subtraction <= the *after* subtraction, print n.
  - Let's make a table. As a reminder, here is our array:

    `{-3, 0, 2, 4, 5, 9, 13, 1, 5}`

n	array[n] – array[n – 1]	array[n] – array[n + 1]	<= ?
1	0 − (-3) = 3	0 − 2 = -2	no
2	2 − 0 = 2	2 − 4 = -2	no
3	4 − 2 = 2	4 − 5 = -1	no
4	5 − 4 = 1	5 − 9 = -4	no
5	9 − 5 = 4	9 − 13 = -4	no
6	13 − 9 = 4	13 − 1 = 12	YES
7	1 − 13 = -12	1 − 5 = -4	YES

- Printing array[n] in the YES cases gives us 13 1. Remember that we are printing the *element,* not the *index.*

13. The answer is C.

- We want to negate our expression, so we are trying to solve this (note the added ! at the beginning).

```
!((n >= 4) || ((m == 5 || k < 2) && (n > 12)))
```

- DeMorgan's theorem tells us that we can distribute the !, but we must change AND to OR and OR to AND when we do that. Let's take it step by step.
- Distribute the ! to the 2 expressions around the || (which will change to &&).

```
!(n >= 4) && !((m == 5 || k < 2) && (n > 12)))
```

- !(n >=4) is the same as (n < 4).

```
(n < 4) && !((m == 5 || k < 2) && (n > 12)))
```

- Now let's distribute the ! to the expression around the second && (which becomes ||).

```
(n < 4) && !(m == 5 || k < 2) || !(n > 12)
```

- Fix the !(n > 12) because that's easy.

```
(n < 4) && !(m == 5 || k < 2) || (n <= 12)
```

- One to go! Distribute the ! around the || in parentheses (which becomes &&).

```
(n < 4) && !(m == 5) && !(k < 2) || (n <= 12)
```

- Simplify those last two simple expressions.

```
(n < 4) && (m != 5) && (k >= 2) || (n <= 12)
```

- A good way to double-check your solution is to assign values to m, n, and k and plug them in. If you do not think you can simplify the expression correctly, assigning values and plugging them in is another way to find the answer, though you may need to check several sets of values to be sure you've found the expression that works every time.

14. The answer is E.

- Consider Option A. Can an index be out of bounds? The largest values m and n will reach are 4 and 6, respectively. 4 + 6 = 10, and ray[10] is the 11th element in the array (because we start counting at 0). The array has a length of 11, so we will not index out of bounds.
- Look at the first for loop. The array is being filled with elements that are equal to twice their indices, so the contents of the array look like { 0, 2, 4, 6, 8, ... }.
- Look at the nested for loop. The outer loop will start at m = 0 and continue through m = 4. For each value of m, n will loop through 0, 2, 4, 6, because n is being incremented by 2. (It's easy to assume all loops use n++. Look!)
- We are looking for m + n > 8. Since the largest value for n is 6, that will not happen when m is 0, 1, or 2.
- The first time m + n is greater than 8 is when m = 3 and n = 6.
  ray[3 + 6] = ray[9] = 2 * 9 = 18. Print 18.
- The next time m + n is greater than 8 is when m = 4 and n = 6.
  ray[4 + 6] = ray[10] = 2 * 10 = 20. Print 20.
- Notice that we aren't printing any spaces, so the 1820 are printed right next to each other.

15. The answer is E.

- This is a recursive method. Let's trace the calls.

```
mystery("advanced placement", 9) = mystery("vanced placement", 10)
mystery("vanced placement", 10) = mystery("nced placement", 11)
mystery("nced placement", 11) = mystery("ed placement", 12)
mystery("ed placement", 12) = mystery ("placement", 13)
```

- By this point we should have noticed that we are getting farther and farther from the base case. What will happen when we run out of characters? Let's keep going and see.

```
mystery("placement", 13) = mystery("lacement", 14)
mystery("lacement", 14) = mystery("cement", 15)
mystery("cement", 15) = mystery("ment", 16)
mystery ("ment", 16)= mystery("nt", 17)
```

- `"nt".substring(2)` is, somewhat surprisingly, a valid expression. You are allowed to begin a substring just past the end of a string. This call will return an empty string.

```
mystery ("nt", 17) = mystery("", 18)
```

- This time code.substring(2) fails: `StringIndexOutOfBoundsException`.

16. The answer is A.

- This problem requires you to understand that primitives are passed by value and objects are passed by reference.
- When a primitive argument (or actual parameter) is passed to a method, its value is copied into the formal parameter. Changing the formal parameter inside the method will have no effect on the value of the variable passed in; `num` and `index` will not be changed by the method.
- When an object is passed to a method, its reference is copied into the formal parameter. The actual and formal parameters become aliases of each other; that is, they both point to the same object. Therefore, when the object is changed inside the method, those changes will be seen outside of the method. Changes to array nums inside the method will be seen in array val outside of the method.
- num and index remain equal to 10 and 3, respectively, but `val[3]` has been changed to 10.

17. The answer is B.

- The original code segment is a nested loop. The outer loop has an index, which will take on the values 2, 4, 6. For each of those values, the inner loop has an index, which will take on the values 30, 20, 10. The code segment will print:

```
(2+30) (2+20) (2+10) (4+30) (4+20) (4+10) (6+30) (6+20) (6+10)
= 32 22 12 34 24 14 36 26 16
```

We need to see which of I, II, III also produce that output.

- Option I is a `for` loop. On entry, num = 32, count = 0, i = 0.
  - 32 is printed, num = 34, count % 3 = 0, so we execute the `if` clause and set count = 0 and num = 20.
  - Next time through the loop, 20 is printed . . . oops, no good! Eliminate option I.
- Option II is a `while` loop. On entry, num = 32.
  - 32 is printed, num = 22, if condition is false.
  - Next time through the loop, 22 is printed, num = 12, if condition is false.
  - 12 is printed, num = 2, if condition is true, num = 34.
  - 34 is printed, num = 24, if condition is false.
  - 24 is printed, num = 14, if condition is false.

- 14 is printed num = 4, if condition is true, num = 36.
- 36 is printed, num = 26, if condition is false.
- 26 is printed, num = 16, if condition is false.
- 16 is printed, num = 6, if condition is true, num = 38.
- Loop terminates – looks good! Option II is correct.

- Option III is a nested for loop.
  - The first time through the loops, h = 0, k = 30.
  - h + k = 30, so 30 is printed. That's incorrect. Eliminate option III.

18. The answer is A.

- This is one way of implementing the Insertion Sort algorithm. The interesting thing about this implementation is the use of a compound condition in the `for` loop. This condition says "Continue until you reach the beginning of the array OR until the element we are looking at is bigger than the key." When the `for` loop exits, the "key" is put into the open slot at position j. That's an Insertion Sort algorithm.
- Looking at this problem a different way:
  - It can't be B. If this were a search algorithm, there would have to be a parameter to tell us what element we are looking for, and that isn't the case.
  - It can't be D because Merge Sort is recursive and there's no recursion in this code segment.
  - It can't be E, because we know Sequential *Search* but there isn't a Sequential Sort.
  - That just leaves Selection Sort. Selection Sort has nested `for` loops, but the loops have no conditional exit. They always go to the end of the array. And on exit, elements are swapped. There's no swap code here.

19. The answer is D.

- The general form for generating a random number between *high* and *low* is

```
(int)(Math.random * (high - low + 1) + low
```

- high – low + 1 = 21, low = 13, so high = 33.
- The correct answer is integers between 13 and 33 inclusive.

20. The answer is D.

- Option A is incorrect. It uses array syntax rather than `ArrayList` syntax to retrieve the element.
- Option B is incorrect. It attempts to access the instance variable population directly, but population is private (as it should be).
- Option C is incorrect. First of all, the algorithm is wrong. It is only comparing the population in consecutive elements of the `ArrayList`, not in the `ArrayList` overall. In addition, it will end with an `IndexOutOfBoundsException`, because it uses (i + 1) as an index.
- Option D works correctly. It accesses the population using the getter method, and it compares the accessed population to the previous max.
- Option E is incorrect. If we began by setting `maxPop = Integer.MAX_VALUE`, then that is the value `maxPop` will have when the code segment completes.

Part II (Free-Response) Solutions

Please keep in mind that there are multiple ways to write the solution to a Free-response question, but the general and refined statements of the problem should be pretty much the same for everyone. Look at the algorithms and coded solutions, and determine if yours accomplishes the same task.

General penalties (assessed only once per problem):

-1 using a local variable without first declaring it
-1 returning a value from a void method or constructor
-1 accessing an array or `ArrayList` incorrectly
-1 overwriting information passed as a parameter
-1 including unnecessary code that causes a side effect like a compile error or console output

1. Complex Numbers

General Problem: Write a `ComplexNumber` class that will represent a complex number and allow the printing and addition of two `ComplexNumber` objects.

Refined Problem: Write a `ComplexNumber` class that includes:

- instance variables representing the real and imaginary components of the complex number.
- a valid constructor that takes two parameters, and a default constructor.
- a method for the addition of `ComplexNumbers` that takes two `ComplexNumber` objects as parameters and returns their sum in a `ComplexNumber` object.
- a method to build a string `ComplexNumber` object in (a + bi) form
- accessors (getters) for the instance variables.

Algorithm:

- Declare two `double` instance variables, a and b.
- Write a constructor that assigns passed values to the instance variables.
- Write a default constructor that assigns the value 0 to the instance variables.
- Write an `add` method that correctly adds the real part and the imaginary part of the two passed `ComplexNumber` objects, and returns a new `ComplexNumber` object.
- Write a `toString` method that builds a string of the `ComplexNumber` object in (a + bi) form, and returns that string.
- Write the accessors (getters) for a and b.

Java Code:

```
public class ComplexNumber
{
    private double a, b;

    public ComplexNumber(double a, double b)
    {
        this.a = a;
        this.b = b;
    }

    public ComplexNumber()
    {
        a = 0;
        b = 0;
    }

    private double getReal()
    {
        return a;
    }

    private double getImaginary()
    {
        return b;
    }

    public ComplexNumber add(ComplexNumber a, ComplexNumber b)
    {
        double real = a.getReal() + b.getReal();
        double imaginary = a.getImaginary() + b.getImaginary();
        return new ComplexNumber(real, imaginary);
    }

    public String toString()
    {
        return "(" + a + " + " + b + "i");
    }

}
```

Common Errors:

- Not declaring the instance variables as `private`
- Not remembering to instantiate a new `ComplexNumber` object in the `add` method.
- Printing the `ComplexNumber` object in the `toString` method.

Scoring Guidelines:

+1 Declares two `private` double instance variables for the real part and imaginary part

+2 Implements the 2-parameter constructor

- **+1** Declares the header: `public ComplexNumber(double ___, double ___)`
- **+1** Uses the parameters to initialize instance variables

+1 Implements the default constructor

+2 `add` method

- **+1** Declares the header: `public ComplexNumber add (ComplexNumber ___, ComplexNumber ___)`
- **+1** Returns a new `ComplexNumber` representing the sum of the two parameters

+2 `toString` method

- **+1** Declares the header: `public String toString()`
- **+1** Returns the appropriate string representing the `ComplexNumber` object in (a + bi) form

+1 Declares accessor methods to access the real and imaginary part of a `ComplexNumber` object

2. Coin Collector

(a) General Problem: Complete the `CoinCollectionTools` class constructor.

Refined Problem: Instantiate the instance variable `coinBox` as a new array of the size specified by the parameters. Traverse the array filling every cell with a `Coin` object instantiated with country = the country parameter, year = 0, and coinType = 0.

Algorithm:

- Instantiate `coinBox` as a new array of `Coin` objects with dimensions `[rows][columns]`.
- Outer loop: traverse the rows of the array.
  Inner loop: traverse the columns of the array.
  - Instantiate a new `Coin` object, passing parameters `(country, 0, 0)`.

Java Code:

```
public CoinCollectionTools(String country, int rows, int columns)
{
   coinBox = new Coin[rows][columns];
   for (int row = 0; row < rows; row++)
      for (int col = 0; col < columns; col++)
         coinBox[row][col] = new Coin(country,0,0);
}
```

Common Errors:

- If you look above the constructor in the code for the class, you will see that the `coinBox` has been declared as an instance variable. If you write:

  ```
  int[][] coinBox = new int[rows][columns];
  ```

  then you are declaring a different array named `coinBox` that exists only within the constructor. The instance variable `coinBox` has not been instantiated.

Java Code Alternate Solution:
If you read the whole problem before starting to code, you might have noticed that the other two parts work in column-major order. You can write this in column-major order also. Since you are filling every cell with the same information, it doesn't make any difference.

```
public CoinCollectionTools(String country, int rows, int columns)
{
   coinBox = new Coin[rows][columns];
   for (int col = 0; col < columns; col++)
      for (int row = 0; row < rows; row++)
         coinBox[row][col] = new Coin(country,0,0);
}
```

(b) General Problem: Complete the `fillCoinBox` method.

Refined Problem: The parameter `myCoins` is an `ArrayList` of `Coin` objects in order by year minted. Assign them to the `coinBox` grid in column-major order.

Algorithm:
- Create a count variable.
- Loop through all of the `Coin` objects in `myCoins` using variable count as the loop counter.
  - Update the row and column variables based on count.
  - Get the next `Coin` object from the `ArrayList`, and place it in the `coinBox` location specified by the row and column variables.
  - Increment the `count` variable.
- Return the completed `coinBox`.

Java Code:

```
public Coin[][] fillCoinBox(ArrayList<Coin> myCoins)
{
   int count = 0;
   int row;
   int column;
   while (count < myCoins.size())
   {
      row = count % coinBox.length;
      column = count / coinBox.length;
      coinBox[row][column] = myCoins.get(count);
      count++;
   }
   return coinBox;
}
```

Java Code Alternate Solution #1:

You may not have thought of using % and / to keep track of column and row. Here's another way to do it.

```
public Coin[][] fillCoinBox(ArrayList<Coin> myCoins)
{
   int count = 0;
   int row = 0;
   int column = 0;
   while (count < myCoins.size())
   {
      coinBox[row][column] = myCoins.get(count);
      if (row < coinBox.length - 1)
      {
         row++;
      }
      else
      {
         row = 0;
         column++;
      }
      count++;
   }
   return coinBox;
}
```

Java Code Alternate Solution #2:

This solution bases its loops on the grid, rather than the `ArrayList`.

```
public Coin[][] fillCoinBox(ArrayList<Coin> myCoins)
{
   int row = 0;
   int column = 0;
   int count = 0;

   // The first condition in the outer loop is not actually
   // necessary, because myCoins will run out before coinBox
   // goes out of bounds.
   while (column < coinBox[0].length && myCoins.size() > count)
   {
      while (row < coinBox.length && myCoins.size() > count)
      {
         coinBox[row][column] = myCoins.get(count);
         count++;
         row++;
      }
      row = 0;
      column++;
   }
   return coinBox;
}
```

Common Errors:

- Do not count on the fact that you can fill the entire grid. It is tempting to write nested `for` loops that traverse the whole grid, but if there are fewer `Coin` objects in the `ArrayList` than elements in the grid, your program will terminate with an `IndexOutOfBoundsException`.
- It is common to write the row and column loops in the wrong order. In column-major order, columns vary slower than rows (we do all the rows before we change columns), so the column loop is the outer loop. In row-major order, the row loop is the outer loop.
- Even though we are filling our array in column-major order, the syntax for specifying the element we want to fill is `coinBox[row][column]`, not the other way around.
- If you used remove instead of get when accessing the `Coin` objects in the `ArrayList`, you modified the list and that's not allowed. It's called *destruction of persistent data* and may be penalized.
- In the solutions that loop through the grid (Alternates #2 and #3), be careful not to use incorrect notation for the end conditions. In general, the number of rows is `arrayName.length` and the number of columns is `arrayName[row].length`. When processing in column-major order, the loop that varies the column is the outer loop. We cannot use the loop variable row as the array index when finding the length of a column, because it does not exist outside of the inner loop. Since this is not a ragged array, it is safe to use `[0]` as our index.

(c) **General Problem:** Complete the `fillCoinTypeList` method.

Refined Problem: Given a `coinBox` as created by part (a) and filled in part (b), create a list that contains `Coins` in order by coin type (1–6). If `Coins` are retrieved from the `coinBox` in column-major order, they will already be in order by year.

Algorithm:

- Loop 1: Complete the inner loops 6 times, once for each coin type 1–6.
- Loop 2: Loop through the columns.
- Loop 3: Loop through the rows.
- If the `Coin` object at the row-column location specified by the loop counters of loops 2 and 3 matches the `coinType` specified from the loop counter of loop 1:
  - Add the `Coin` object to the `ArrayList`.
- Return the `ArrayList` of `Coin` objects.

Java Code:

```
public ArrayList<Coin> fillCoinTypeList()
{
   ArrayList<Coin> myCoins = new ArrayList<Coin>();
   for (int type = 1; type <= 6; type++)
      for (int col = 0; col < coinBox[0].length; col++)
         for (int row = 0; row < coinBox.length; row++)
            if (coinBox[row][col].getCoinType() == type)
               myCoins.add(coinBox[row][col]);
   return myCoins;
}
```

Common Errors:

- You should not create a new `Coin` object to add to the `myCoins` list. The `Coin` object already exists in the array.
- Do not worry about the default `Coins` added in the constructor. Since they have a `coinType` of 0, they will be ignored.

Scoring Guidelines:

Part (a): CoinCollectionTools constructor **2 points**

+1 Traverses the array using a nested loop. No bounds errors, no missed elements

+1 Instantiates new `Coin` objects for each cell in the array

Part (b): fillCoinBox **4 points**

+1 Accesses every element in `myCoins`. No bounds errors, no missed elements

+1 Accesses all appropriate elements of `coinBox`. No bounds errors, no missed elements

+1 Accesses elements of `coinBox` in column-major order

+1 Returns correctly filled `coinBox`

Part (c): fillCoinTypeList **3 points**

+1 Loops through `coinBox` at least once in column major order. No bounds errors, no missed elements

+1 Locates `Coins` of type 1, followed by types 2–6 in the correct order

+1 Instantiates, fills, and returns a correct `ArrayList<Coin>`

Sample Driver:

There are many ways to write these methods. Maybe yours is a bit different from our sample solutions and you are not sure if it works. Here is a sample driver program. Running it will let you see if your code works, and will help you debug it if it does not.

`CopyCoinCollectionToolsDriver` into your IDE along with the complete `Coin` and `CoinCollectionTools` classes (including your solutions). You will also need to add this import statement as the first line in your `CoinCollectionTools` class: `import java.util.ArrayList;`

```
import java.util.ArrayList;

public class CoinCollectionToolsDriver {

   public static void main(String[] args) {
      CoinCollectionTools tools = new CoinCollectionTools("USA", 3, 4);
      int[] years = { 1920, 1930, 1940, 1940, 1950, 1950,
            1950, 1960, 1970, 1980, 1990 };
      int[] types = { 1, 2, 4, 1, 2, 3, 3, 4, 4, 2, 4 };
      String[] typeNames = { "", "penny", "nickel", "dime", "quarter" };
      Coin[][] coinBox = new Coin[3][4];
      ArrayList<Coin> coins = new ArrayList<Coin>();

      for (int i = 0; i < years.length; i++)
         coins.add(new Coin("USA", years[i], types[i]));

      System.out.println("fillCoinBox test\nExpecting:");
      System.out.println("penny 1920\tpenny 1940\tdime 1950\tnickel 1980"
            + "\nnickel 1930\tnickel 1950\tquarter 1960\tquarter 1990"
            + "\nquarter 1940\tdime 1950\tquarter 1970\t0");

      System.out.println("\nYour answer:");
      coinBox = tools.fillCoinBox(coins);

      for (int row = 0; row < coinBox.length; row++) {
         for (int col = 0; col < coinBox[row].length; col++)
            System.out.print(typeNames[coinBox[row][col].getCoinType()] + " "
                  + coinBox[row][col].getYear() + "\t");
         System.out.println();
      }

      System.out.println("\nfillCoinTypeList test\nExpecting:");
      System.out.println("penny 1920, penny 1940, nickel 1930, "
            + "nickel 1950, nickel 1980, dime 1950, "
            + "\ndime 1950, quarter 1940, quarter 1960, "
            + "quarter 1970, quarter 1990,");
      System.out.println("\nYour answer:");
      coins = tools.fillCoinTypeList();
      for (int i = 0; i < coins.size(); i++) {
         System.out.print(typeNames[coins.get(i).getCoinType()]
            + " " + coins.get(i).getYear()+ ", ");
         if (i == coins.size() / 2)
            System.out.println();
      }
   }
}
```