The College Board

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the College Board is composed of more than 5,700 schools, colleges, universities and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,800 colleges through major programs and services in college readiness, college admission, guidance, assessment, financial aid and enrollment. Among its widely recognized programs are the SAT®, the PSAT/NMSQT®, the Advanced Placement Program® (AP®), SpringBoard® and ACCUPLACER®. The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities and concerns.

© 2011 The College Board. College Board, ACCUPLACER, Advanced Placement Program, AP, AP Central, SAT, SpringBoard and the acorn logo are registered trademarks of the College Board. Admitted Class Evaluation Service is a trademark owned by the College Board. PSAT/NMSQT is a registered trademark of the College Board and National Merit Scholarship Corporation. All other products and services may be trademarks of their respective owners. Permission to use copyrighted College Board materials may be requested online at: www.collegeboard.com/inquiry/cbpermit.html.

Visit the College Board on the Web: www.collegeboard.org.
AP Central is the official online home for the AP Program: apcentral.collegeboard.com.
Apply the question-specific rubric first; the question-specific rubric always takes precedence.

Penalties: The penalty categorization below is for cases not covered by the question-specific rubric. Points can only be deducted in a part of the question that has earned credit via the question-specific rubric, and no section may have a negative point total. A given penalty can be assessed only once in a question, even if it occurs on different parts of that question. A maximum of 3 penalty points may be assessed over the entire question.

Nonpenalized Errors
- spelling/case discrepancies if no ambiguity*
- local variable not declared if other variables are declared in some part
- use of keyword as identifier
- [] vs. () vs. <>
- = instead of == (and vice versa)
- length/size confusion for array, String, and ArrayList, with or without ()
- private qualifier on local variable
- extraneous code with no side effect; e.g., precondition check
- common mathematical symbols for operators (x • ÷ < > ≤ ≥)
- missing {} where indentation clearly conveys intent and {} used elsewhere
- default constructor called without parens; e.g., new Critter;
- missing ( ) on parameter-less method call
- missing ( ) around if/while conditions
- missing ; when majority are present
- missing public on class or constructor header
- extraneous [] when referencing entire array
- [i, j] instead of [i][j]
- extraneous size in array declaration, e.g., int[size] nums = new int[size];

* Spelling and case discrepancies for identifiers fall under the “nonpenalized” category only if the correction can be unambiguously inferred from context; for example, “ArayList” instead of “ArrayList”. As a counterexample, note that if a student declares “Bug bug;” then uses “Bug.move()” instead of “bug.move()”, the context does not allow for the reader to assume the object instead of the class.

Minor Errors (½ point)
- confused identifier (e.g., len for length or left() for getLeft())
- local variables used but none declared
- missing new in constructor call
- modifying a constant (final)
- use of equals or compareTo method on primitives, e.g., int x; ...
- x.equals(val)
- array/collection access confusion ([] get)
- assignment dyslexia, e.g., x + 3 = y; for y = x + 3;
- super(method()) instead of super.method()
- formal parameter syntax (with type) in method call, e.g., a = method(int x)
- missing public from method header when required
- "false"/"true" or 0/1 for boolean values
- "null" for null

Major Errors (1 point)
- extraneous code that causes side effect; e.g., information written to output
- interface or class name instead of variable identifier; e.g., Bug.move() instead of aBug.move()
- aMethod(obj) instead of obj.aMethod()
- attempt to use private data or method when not accessible
- destruction of persistent data (e.g., changing value referenced by parameter)
- use of class name in place of super in constructor or method call
- void method (or constructor) returns a value

Applying Minor Penalties (½ point):
A minor infraction that occurs exactly once when the same concept is correct two or more times is regarded as an oversight and not penalized. A minor penalty must be assessed if the item is the only instance, one of two, or occurs two or more times.
Question 1: Sound

**Part (a)**  limitAmplitude  4½ points

*Intent:* Change elements of samples that exceed ±limit; return number of changes made

+3 Identify elements of samples to be modified and modify as required
+1 Consider elements of samples
+½ Accesses more than one element of samples
+½ Accesses every element of samples (no bounds errors)
+2 Identify and change elements of samples
+½ Compares an element of samples with limit
+½ Changes at least one element to limit or −limit
+1 Changes all and only elements that exceed ±limit to limit or −limit appropriately

+1½ Calculate and return number of changed elements of samples
+1 Initializes and updates a counter to achieve correct number of changed samples
+½ Returns value of an updated counter (requires array access)

**Part (b)**  trimSilenceFromBeginning  4½ points

*Intent:* Remove leading elements of samples that have value of 0, potentially resulting in array of different length

+1½ Identify leading-zero-valued elements of samples
+½ Accesses every leading-zero element of samples
+½ Compares 0 and an element of samples
+½ Compares 0 and multiple elements of samples

+1 Create array of proper length
+½ Determines correct number of elements to be in resulting array
+½ Creates new array of determined length

+2 Remove silence values from samples
+½ Copies some values other than leading-zero values
+1 Copies all and only values other than leading-zero values, preserving original order
+½ Modifies instance variable samples to reference newly created array

**Question-Specific Penalties**

−1 Array identifier confusion (e.g., value instead of samples)
−½ Array/collection modifier confusion (e.g., using set)
Question 1: Sound

Part (a):

public int limitAmplitude(int limit) {
    int numChanged = 0;
    for (int i = 0; i < this.samples.length; i++) {
        if (this.samples[i] < -limit) {
            this.samples[i] = -limit;
            numChanged++;
        }
        if (this.samples[i] > limit) {
            this.samples[i] = limit;
            numChanged++;
        }
    }
    return numChanged;
}

Part (b):

public void trimSilenceFromBeginning() {
    int i = 0;
    while (this.samples[i] == 0) {
        i++;
    }
    int[] newSamples = new int[this.samples.length - i];
    for (int j = 0; j < newSamples.length; j++) {
        newSamples[j] = this.samples[j+i];
    }
    this.samples = newSamples;
}
# Question 2: Attractive Critter (GridWorld)

<table>
<thead>
<tr>
<th>Class: AttractiveCritter</th>
<th>9 points</th>
</tr>
</thead>
</table>

**Intent:** Define extension to Critter class that relocates all other actors closer to itself

- **+1** Properly formed class header for AttractiveCritter that extends Critter class.

- **+2½** Override Critter methods and maintain all postconditions
  - **+1** Overrides at least one method of Critter and satisfies all postconditions *(point not awarded if also overrides act method)*
  - **+½** Overrides getActors
  - **+1** Overrides processActors

- **+5½** Move other actors in grid to be closer to self
  - **+1** Considers all other actors in grid
  - **+½** Checks for an empty movement destination
  - **+1½** Moves an actor
    - **+½** Moves at least one other actor to different location in grid
    - **+1** Moves another actor and guards against inappropriate self-movement
  - **+1½** Determines correct direction and location
    - **+½** Determines correct direction toward self for at least one other actor
    - **+1** Determines adjacent location to at least one other actor *(point awarded only if calculated direction is used as parameter)*
  - **+1** Moves all other actors to calculated destinations

**Question-Specific Penalties**

- **−1** Inappropriate state change in world (Grid, Actor,...)
Question 2: Attractive Critter (GridWorld)

Solution that checks for self in `getActors`

```java
public class AttractiveCritter extends Critter {
    public class AttractiveCritter extends Critter {
        public ArrayList<Actor> getActors() {
            ArrayList<Actor> actors = new ArrayList<Actor>();
            for (Location loc : getGrid().getOccupiedLocations()) {
                if (!loc.equals(this.getLocation())) {
                    actors.add(getGrid().get(loc));
                }
            }
            return actors;
        }

        public void processActors(ArrayList<Actor> actors) {
            for (Actor a : actors) {
                int direction = (a.getLocation()).getDirectionToward(this.getLocation());
                Location newLoc = (a.getLocation()).getAdjacentLocation(direction);
                if (getGrid().get(newLoc) == null) {
                    a.moveTo(newLoc);
                }
            }
        }
    }
}
```

Solution that checks for self in `processActors`

```java
public class AttractiveCritter extends Critter {
    public ArrayList<Actor> getActors() {
            ArrayList<Actor> actors = new ArrayList<Actor>();
            for (Location loc : getGrid().getOccupiedLocations()) {
                actors.add(getGrid().get(loc));
            }
            return actors;
        }

        public void processActors(ArrayList<Actor> actors) {
            for (Actor a : actors) {
                if (a != this) {
                    int direction = (a.getLocation()).getDirectionToward(this.getLocation());
                    Location newLoc = (a.getLocation()).getAdjacentLocation(direction);
                    if (getGrid().get(newLoc) == null) {
                        a.moveTo(newLoc);
                    }
                }
            }
        }
    }
}
```
Question 3: Fuel Depot

Part (a)  nextTankToFill  5 points

**Intent:** Return index of tank with minimum level (\(\leq\) threshold)

+4  Determine minimum element of tanks that is \(\leq\) threshold, if any
    +1½  Consider fuel levels of elements of tanks
    +½  Accesses fuel level of an element of tanks
    +½  Accesses at least one element of tanks in context of repetition (iteration/recursion)
    +½  Accesses every element of tanks at least once

+2½  Identify minimum element of tanks that is \(\leq\) threshold
    +½  Compares fuel levels from at least two elements of tanks
    +½  Implements algorithm to find minimum
    +½  Identifies tank (object or index) holding identified minimum
    +½  Determines element identified as minimum fuel level that is also \(\leq\) threshold

+1  Return the index of the element satisfying the conditions, or the current index if no element does so
    +½  Returns index of element identified as satisfying threshold & minimum conditions*
    +½  Returns filler.getCurrentIndex() when no element satisfies conditions*

*Note: Point is not awarded if wrong data type is returned.

Part (b)  moveToLocation  4 points

**Intent:** Move robot to given tank location

+2  Ensure robot is pointing in direction of tank to be filled
    +½  Determines direction filler is currently facing
    +½  Changes filler’s direction for some condition
    +1  Establishes filler’s direction as appropriate for all conditions

+2  Place robot at specified location
    +½  Invokes moveForward method with a parameter
    +½  Invokes moveForward method with a verified non-zero parameter
    +1  Invokes filler.moveForward method with a correctly computed parameter
Question 3: Fuel Depot

Part (a):

```java
public int nextTankToFill(int threshold) {
    int minLevel = this.tanks.get(0).getFuelLevel();
    int minTankIndex = 0;
    for (int i = 1; i < this.tanks.size(); i++) {
        if (this.tanks.get(i).getFuelLevel() < minLevel) {
            minLevel = this.tanks.get(i).getFuelLevel();
            minTankIndex = i;
        }
    }
    if (minLevel <= threshold) {
        return minTankIndex;
    } else {
        return this.filler.getCurrentIndex();
    }
}
```

// Alternative solution

```java
public int nextTankToFillA(int threshold) {
    int minTankIndex = this.filler.getCurrentIndex();
    for (int i = 0; i < this.tanks.size(); i++) {
        if (this.tanks.get(i).getFuelLevel() <= threshold &&
            this.tanks.get(i).getFuelLevel() <
            this.tanks.get(minTankIndex).getFuelLevel()) {
            minTankIndex = i;
        }
    }
    return minTankIndex;
}
```

Part (b):

```java
public void moveToLocation(int locIndex) {
    if (this.filler.getCurrentIndex() > locIndex) {
        if (this.filler.isFacingRight()) {
            this.filler.changeDirection();
        }
        this.filler.moveForward(this.filler.getCurrentIndex() - locIndex);
    }
    if (this.filler.getCurrentIndex() < locIndex) {
        if (!this.filler.isFacingRight()) {
            this.filler.changeDirection();
        }
        this.filler.moveForward(locIndex - this.filler.getCurrentIndex());
    }
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.
Question 4: Cipher

Part (a) fillBlock 3½ points

**Intent:** Fill letterBlock in row-major order from parameter; pad block or truncate string as needed

- **+½** Copies at least one substring from parameter to letterBlock
- **+½** Completely fills letterBlock from parameter if possible
  - *(no bounds errors in letterBlock or parameter)*
- **+1** Results in a distribution of all consecutive one-character substrings from parameter to letterBlock *(ignores surplus characters)*
- **+½** Copies these one-character substrings from parameter to letterBlock in such a way that the result is in row-major order
- **+1** Pads letterBlock with "A" if and only if parameter is shorter than block size

Part (b) encryptMessage 5½ points

**Intent:** Return encrypted string created by repeatedly invoking fillBlock and encryptBlock on substrings of parameter and concatenating the results

- **+2** Partition parameter
  - **+½** Returns the empty string if the parameter is the empty string
  - **+½** Creates substrings of parameter that progress through the parameter string *(can overlap or skip)*
  - **+1** Processes every character in parameter exactly once *(no bounds errors)*
- **+3** Fill and encrypt a block, concatenate results
  - **+½** Invokes fillBlock with parameter or substring of parameter
  - **+½** Invokes fillBlock on more than one substring of parameter
  - **+½** Invokes encryptBlock after each invocation of fillBlock
  - **+1** Concatenates encrypted substrings of parameter
  - **+1** Builds complete, encrypted message
- **+½** Return resulting built string

**Question-Specific Penalties**

- **−1½** Use of identifier with no apparent resemblance to letterBlock for two-dimensional array
Question 4: Cipher

Part (a):

private void fillBlock(String str) {
    int pos = 0;
    for (int r = 0; r < this.numRows; r++) {
        for (int c = 0; c < this.numCols; c++) {
            if (pos < str.length()) {
                this.letterBlock[r][c] = str.substring(pos, pos+1);
                pos++;
            } else {
                this.letterBlock[r][c] = "A";
            }
        }
    }
}

// Alternative solution

private void fillBlock(String str) {
    for (int r = 0; r < this.numRows; r++) {
        for (int c = 0; c < this.numCols; c++) {
            if (str.length() > (c + (r * this.numCols))) {
                this.letterBlock[r][c] = str.substring(c + r * this.numCols, 1 + c + r * this.numCols);
            } else {
                this.letterBlock[r][c] = "A";
            }
        }
    }
}
Question 4: Cipher (continued)

Part (b):

```java
public String encryptMessage(String message) {
    String encryptedMessage = "";
    int chunkSize = this.numRows * this.numCols;
    while (message.length() > 0) {
        if (chunkSize > message.length()) {
            chunkSize = message.length();
        }
        fillBlock(message);
        encryptedMessage += encryptBlock();
        message = message.substring(chunkSize);
    }
    return encryptedMessage;
}
```

// Alternative solution

```java
public String encryptMessage(String message) {
    if (message.length() == 0) return "";
    fillBlock(message);
    if (message.length() <= this.numRows * this.numCols) {
        return encryptBlock();
    }
    return (encryptBlock() +
            encryptMessage(message.substring(this.numRows * this.numCols)));
}
```