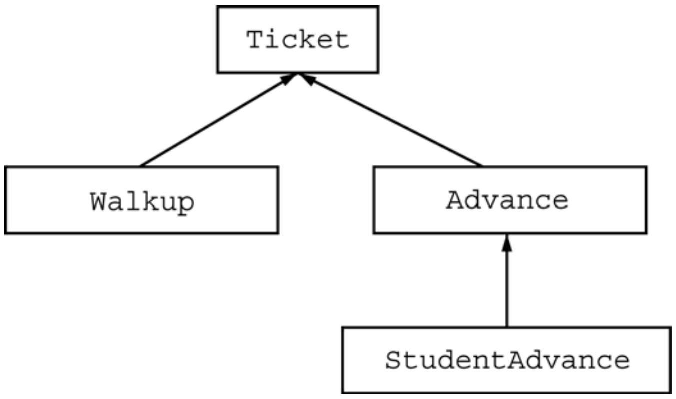
2. A set of classes is used to handle the different ticket types for a theater. The class hierarchy is shown in the following diagram.



All tickets have a serial number and a price. The class `Ticket` is specified as an `abstract` class as shown in the following declaration.

```
public abstract class Ticket
{
  private int serialNumber;    // unique ticket id number


  public Ticket()
  {  serialNumber = getNextSerialNumber();  }


  // returns the price for this ticket
  public abstract double getPrice();


  // returns a string with information about the ticket
  public String toString()
  {
    return "Number: " + serialNumber + "\nPrice: " + getPrice();
  }


  // returns a new unique serial number
  private static int getNextSerialNumber()
  {  /* implementation not shown */  }
}
```

GO ON TO THE NEXT PAGE.

Each ticket has a unique serial number that is assigned when the ticket is constructed. For all ticket classes, the `toString` method returns a string containing the information for that ticket. Three additional classes are used to represent the different types of tickets and are described in the table below.

Class	Description	Sample `toString` Output

`Walkup`	These tickets are purchased on the day of the event and cost 50 dollars.	`Number: 712`
		`Price: 50`

`Advance`	Tickets purchased ten or more days in advance cost 30 dollars. Tickets purchased fewer than ten days in advance cost 40 dollars.	`Number: 357`
		`Price: 40`

`StudentAdvance`	These tickets are a type of `Advance` ticket that costs half of what that `Advance` ticket would normally cost.	`Number: 134`
		`Price: 15`
		`(student ID required)`

Using the class hierarchy and specifications given above, you will write complete class declarations for the `Advance` and `StudentAdvance` classes.

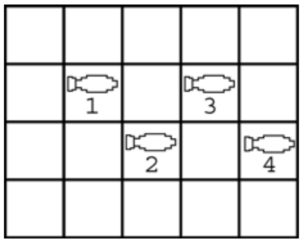
(a) Write the complete class declaration for the class `Advance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. Tickets purchased ten or more days in advance cost $30; tickets purchased nine or fewer days in advance cost $40.

(b) Write the complete class declaration for the class `StudentAdvance`. Include all necessary instance variables and implementations of its constructor and method(s). The constructor should take a parameter that indicates the number of days in advance that this ticket is being purchased. The `toString` method should include a notation that a student ID is required for this ticket. A `StudentAdvance` ticket costs half of what that `Advance` ticket would normally cost. If the pricing scheme for `Advance` tickets changes, the `StudentAdvance` price should continue to be computed correctly with no code modifications to the `StudentAdvance` class.

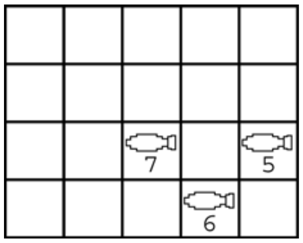
GO ON TO THE NEXT PAGE.

3. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

   Consider defining a new type of fish called `ZigZagFish`, which moves in a zigzag pattern. The first time a `ZigZagFish` moves, it will move to the location forward and to the right if that cell is empty. This is illustrated in the figure below as the move from position 1 to position 2. In each subsequent move, the `ZigZagFish` will attempt to move to the forward diagonal location on the side opposite its previous move (the second move will be to the left forward diagonal, the third move will be to the right forward diagonal, and so on). When the `ZigZagFish` has successfully moved, its direction does not change. If the `ZigZagFish` is unable to move, it stays in the same location but reverses its direction. After reversing its direction, the next time the `ZigZagFish` moves, it will attempt to move in the same diagonal direction as it tried before reversing. The diagrams below show the path followed by a single `ZigZagFish` object as a result of multiple moves.



   Now consider what happens when the `ZigZagFish` attempts to move forward diagonally to the <u>left</u> from position 4. This move is blocked, and consequently the `ZigZagFish` stays in the same location but reverses its direction. This is illustrated as position 5 in the diagram below. From position 5, the `ZigZagFish` moves forward diagonally to the <u>left</u>, and from position 6, it moves forward diagonally to the <u>right</u> to position 7.



GO ON TO THE NEXT PAGE.

The `ZigZagFish` class is defined by extending the `Fish` class and overriding the `move` and `nextLocation` methods. Because a `ZigZagFish` alternates its pattern of movement, a private instance variable `willZigRight` keeps track of the direction of the next movement.

The partial declaration for class `ZigZagFish` is shown below.

```
public class ZigZagFish extends Fish
{
  private boolean willZigRight;
    // true indicates the next move should be forward to the right
    // false indicates the next move should be forward to the left

  public ZigZagFish(Environment env, Location loc)
  {
    super(env, loc);
    willZigRight = true; // direction of the next move
  }

  // returns the forward diagonal cell to the left or right of this fish
  // (depending on willZigRight) if that cell is empty;
  // otherwise, returns this fish's current location
  // postcondition: the state of this ZigZagFish is unchanged
  protected Location nextLocation()
  {  /* to be implemented in part (a) */  }

  // moves this ZigZagFish diagonally (as specified in nextLocation) if
  // possible; otherwise, reverses direction without moving;
  // after a diagonal move, willZigRight is updated
  protected void move()
  {  /* to be implemented in part (b) */  }

  // other constructors, generateChild, and other methods not shown
}
```

GO ON TO THE NEXT PAGE.

(a) Override the `nextLocation` method for the `ZigZagFish` class. The `nextLocation` method returns the cell diagonally forward to the right, the cell diagonally forward to the left, or the cell that the `ZigZagFish` currently occupies, according to the description given at the beginning of this question.

In writing `nextLocation`, you may use any of the accessible methods of the classes in the case study.

Complete method `nextLocation` below.

```
// returns the forward diagonal cell to the left or right of this fish
// (depending on willZigRight) if that cell is empty;
// otherwise, returns this fish's current location
// postcondition: the state of this ZigZagFish is unchanged
protected Location nextLocation()
```

(b) Override the `move` method for the `ZigZagFish` class. This method should change the location and direction of the fish as needed, according to the rules of movement described at the beginning of the question. In addition, the state of the fish must be updated.

In writing `move`, you may call `nextLocation`. Assume that `nextLocation` works as specified, regardless of what you wrote in part (a). You may also use any of the accessible methods of the classes in the case study.

Complete method `move` below.

```
// moves this ZigZagFish diagonally (as specified in nextLocation) if
// possible; otherwise, reverses direction without moving;
// after a diagonal move, willZigRight is updated
protected void move()
```

GO ON TO THE NEXT PAGE.

4. Consider a grade-averaging scheme in which the final average of a student's scores is computed differently from the traditional average if the scores have "improved." Scores have improved if each score is greater than or equal to the previous score. The final average of the scores is computed as follows.

   A student has `n` scores indexed from 0 to `n-1`. If the scores have improved, only those scores with indexes greater than or equal to `n/2` are averaged. If the scores have not improved, all the scores are averaged.

   The following table shows several lists of scores and how they would be averaged using the scheme described above.

Student Scores	Improved?	Final Average
50, 50, 20, 80, 53	No	(50 + 50 + 20 + 80 + 53) / 5.0 = 50.6
20, 50, 50, 53, 80	Yes	(50 + 53 + 80) / 3.0 = 61.0
20, 50, 50, 80	Yes	(50 + 80) / 2.0 = 65.0

   Consider the following incomplete `StudentRecord` class declaration. Each `StudentRecord` object stores a list of that student's scores and contains methods to compute that student's final average.

```
public class StudentRecord
{
  private int[] scores; // contains scores.length values
                        // scores.length > 1

  // constructors and other data fields not shown


  // returns the average (arithmetic mean) of the values in scores
  // whose subscripts are between first and last, inclusive
  // precondition:  0 <= first <= last < scores.length
  private double average(int first, int last)
  {  /* to be implemented in part (a) */  }


  // returns true if each successive value in scores is greater
  // than or equal to the previous value;
  // otherwise, returns false
  private boolean hasImproved()
  {  /* to be implemented in part (b) */  }


  // if the values in scores have improved, returns the average
  // of the elements in scores with indexes greater than or equal
  // to scores.length/2;
  // otherwise, returns the average of all of the values in scores
  public double finalAverage()
  {  /* to be implemented in part (c) */  }

}
```

GO ON TO THE NEXT PAGE.

AP® COMPUTER SCIENCE A
2005 SCORING GUIDELINES

2005 A Question 2: Ticket Sales

Part A: `Advance` **3 1/2 points**

+1/2 `class Advance extends Ticket` (no abstract)

+1/2 private data field (either days or price)

+1 1/2 constructor
- **+1/2** correct header
- **+1** correctly assign data field(s) (lose if reference to super's private data)

+1 `getPrice`
- **+1/2** correct header (must be `public` & `double`, no `abstract`, no parameters)
- **+1/2** return correct price

Part B: `StudentAdvance` **5 1/2 points**

+1/2 `class StudentAdvance extends Advance`

+1 1/2 constructor
- **+1/2** correct header
- **+1/2** attempt to call `super`
- **+1/2** correct call to `super`

+2 `getPrice`
- **+1/2** correct header (must be `public` & `double`, no `abstract`, no parameters)
- **+1** call `super.getPrice()`
- **+1/2** calculate and return correct price

+1 1/2 `toString`
- **+1/2** call `super.toString()`
- **+1** return `string` with correct phrase concatenated (lose this with a reference to super class's private data)

Usage: -1/2 in part A if `super()` appears in the constructor and it is not the first statement executed.

Workshop Exam Materials
Canonical Solutions
2005 AP® Computer Science A

Question 2

PART A:

```
public class Advance extends Ticket
{
  private int daysInAdvance;

  public Advance(int numDays)
  {
    super();
    daysInAdvance = numDays;
  }

  public double getPrice()
  {
    if (daysInAdvance >= 10)
    {
      return 30.0;
    }
    else
    {
      return 40.0;
    }
  }
}
```

OR

```
public class Advance extends Ticket
{
  private double price;

  public Advance(int numDays)
  {
    super();
    if (numDays >= 10)
    {
      price = 30.0;
    }
    else
    {
      price = 40.0;
    }
  }

  public double getPrice()
  {
    return price;
  }
}
```

PART B:

```
public class StudentAdvance extends Advance
{
  public StudentAdvance(int numDays)
  {
    super(numDays);
  }

  public double getPrice()
  {
    return super.getPrice()/2;
  }

  public String toString()
  {
    return super.toString() + "\n(student ID required)";
  }
}
```