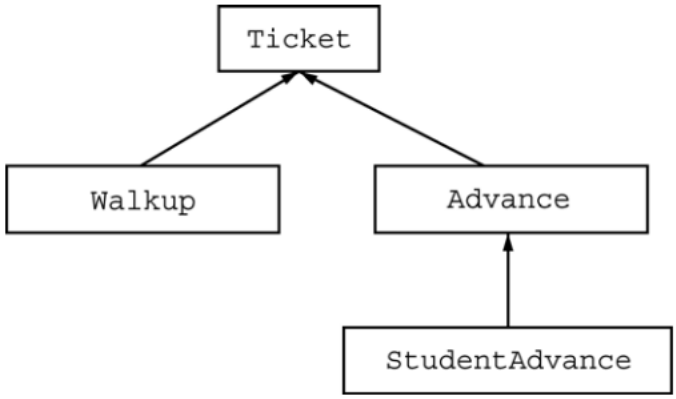
Name _____ Period_____

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 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 */
    }
}
```

Each ticket has a unique seral 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 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 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 variable 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.

```
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;
  }
}
```

/4

(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.

```
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)";
  }
}
```

/6

2. The StringChecker interface describes classes that check if strings are valid, according to some criterion.

```
public interface StringChecker {
    /**Returns true if str is vvalid */
    boolean isValid(String str);
}
```

The CodeWordChecker is a StringChecker. A CodeWordChecker object can be constructed with three parameters: two integers and a string. The first two parameters specify the minimum and maximum code word lengths, respectively, and the third parameter specifies a string that must not occur in the code word. A `CodeWordChecker` object can also be constructed with a single parameter that specifies a string that must not occur in the code word; in this case the minimum and maximum lengths will default to 6 and 20, respectively.

The following examples illustrate the behavior of `CodeWordChecker` objects.

Example 1

```
StringChecker sc1 = new CodeWordChecker(5, 8, "$");
```

Valid code words have 5 to 8 characters and must not include the string "$".

Method call	Return value	Explanation
`sc1.isValid("happy")`	`true`	The code word is valid.
`sc1.isValid("happy$")`	`false`	The code word contains `"$"`.
`sc1.isValid("Code")`	`false`	The code word is too short.
`sc1.isValid("happyCode")`	`false`	The code word is too long.

Example 2

```
StringChecker sc2 = new CodeWordChecker("pass");
```

Valid code words must not include the string "pass". Because the bounds are not specified, the length bounds are 6 and 20, inclusive.

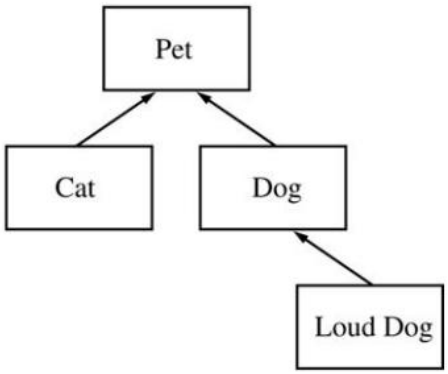
Method call	Return value	Explanation
`sc2.isValid("MyPass")`	`true`	The code word is valid.
`sc2.isValid("Mypassport")`	`false`	The code word contains `"pass"`.
`sc2.isValid("happy")`	`false`	The code word is too short.
`sc2.isValid("1,000,000,000,000,000")`	`false`	The code word is too long.

Write the complete `CodeWordChecker` class. Your implementation must meet all specifications and conform to all examples.

```
public class CodeWordChecker implements StringChecker
{
   private int minLength;
   private int maxLength;
   private String notAllowed;

   public CodeWordChecker(int minLen, int maxLen, String symbol)
   {
      minLength = minLen;
      maxLength = maxLen;
      notAllowed = symbol;
   }

   public CodeWordChecker(String symbol)
   {
      minLength = 6;
      maxLength = 20;
      notAllowed = symbol;
   }

   public boolean isValid(String str)
   {
      return str.length() >= minLength && str.length() <= maxLength &&
             str.indexOf(notAllowed) == -1;
   }

}
```

/9

3. Consider the hierarchy of classes shown in the following diagram,



Note that `Cat` "is-a" `Pet`, a `Dog` "is-a" `Pet`, and a `LoudDog` "is-a" `Dog`.

The class `Pet` is specified as an abstract class as shown in the following declaration. Each `Pet` has a name that is specific when it is constructed.

```
public abstract class Pet {
    private String myName;

    public Pet(String name){
        myName = name;
    }

    public String getName(){
        return myName;
    }

    public abstract String speak();
}
```

The subclass `Dog` has the partial class declaration shown below.

```
public class Dog extends Pet{

    public Dog(String name){
        /* implementation not shown */
    }

    public String speak(){
        /* implementation not shown */
    }
}
```

(a) Given the class hierarchy shown above, write a complete class declaration for the class `Cat`, including implementations of its constructor and method(s). The `Cat` method speak returns "meow" when it is invoked.

```
public class Cat extends Pet {
 public Cat(String name) {
   super(name);
 }
 public String speak() {
   return "meow";
 }
}
```

/2

(b) Assume that class `Dog` has been declared as shown at the beginning of the question. If the String *dog-sound* is returned by the `Dog` method speak, then the `LoudDog` method speak returns a String containing *dog-sound* repeated two times.

Given the class hierarchy shown previously, write a complete class declaration for the class `LoudDog`, including implementations of its constructor and method(s).

```
public class LoudDog extends Dog {
 public LoudDog(String name) {
   super(name);
 }
 public String speak() {
   return super.speak() + super.speak();
 }
}
```

/3

(c) Consider the following partial declaration of class Kennel.

```
public class Kennel {

    private Pet pets[] = new Pet[5];

    public void allSpeak(){
        /* To be implemented in this part */
    }

}
```

Write the Kennel method `allSpeak`. For each `Pet` in the kennel, all `Speak` prints a line with the name of the `Pet` followed by the result of a call to its `speak` method.

In writing `allSpeak`, you may use any of the methods defined for any of the classes specified for this problem. Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

```
public void allSpeak(){
    /* To be implemented in this part */
    for(Pet p:pets){
        p.getName();
        p.speak();
    }
}
```

/4