# The University of British Columbia

## CPSC 210

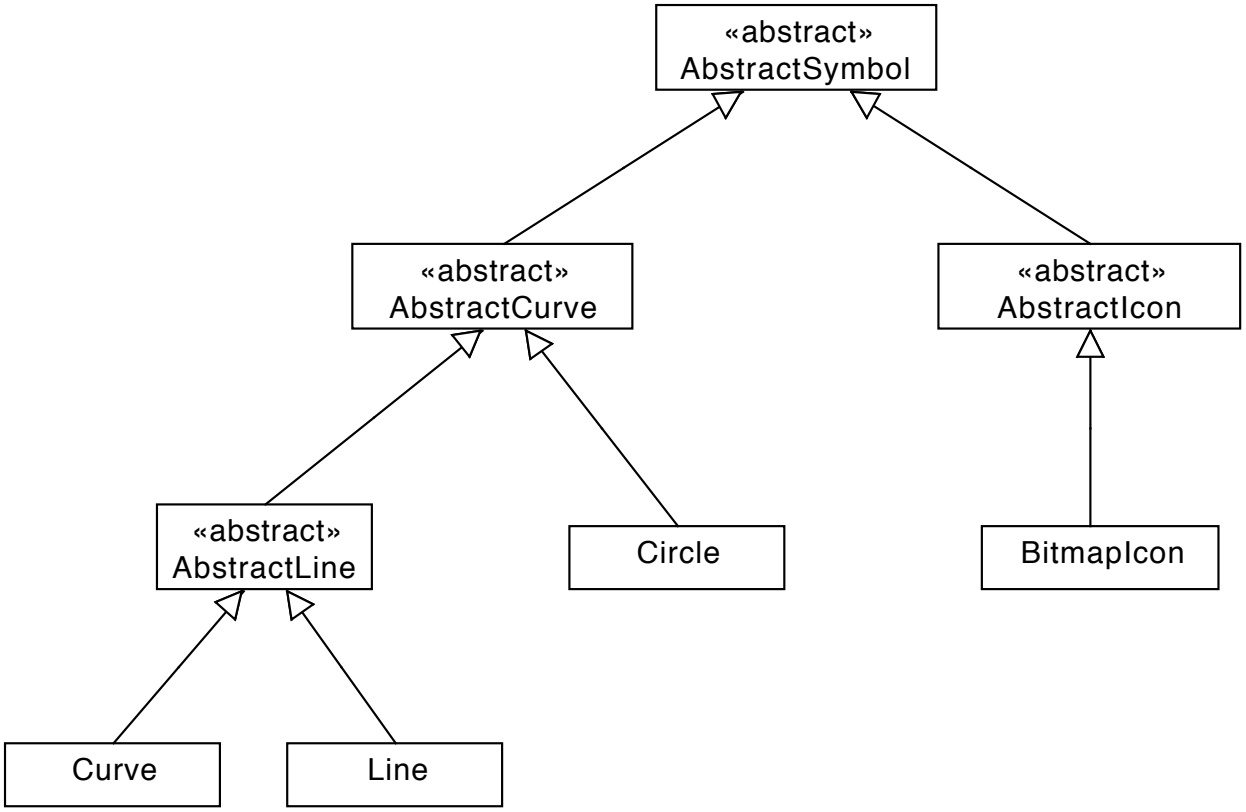
### Sample Midterm 1 Exam Questions (SAMPLE SOLUTIONS)

**Please** don't look at these solutions until you have put significant effort into coming up with your own. Doing so could lead to a false sense of security, as understanding a given solution is *much* easier than constructing your own. You need to practice doing what the exam will ask you to do!

**IMPORTANT:** Questions 1 to 3 apply to the `JDrawing` system provided in the specified repository.

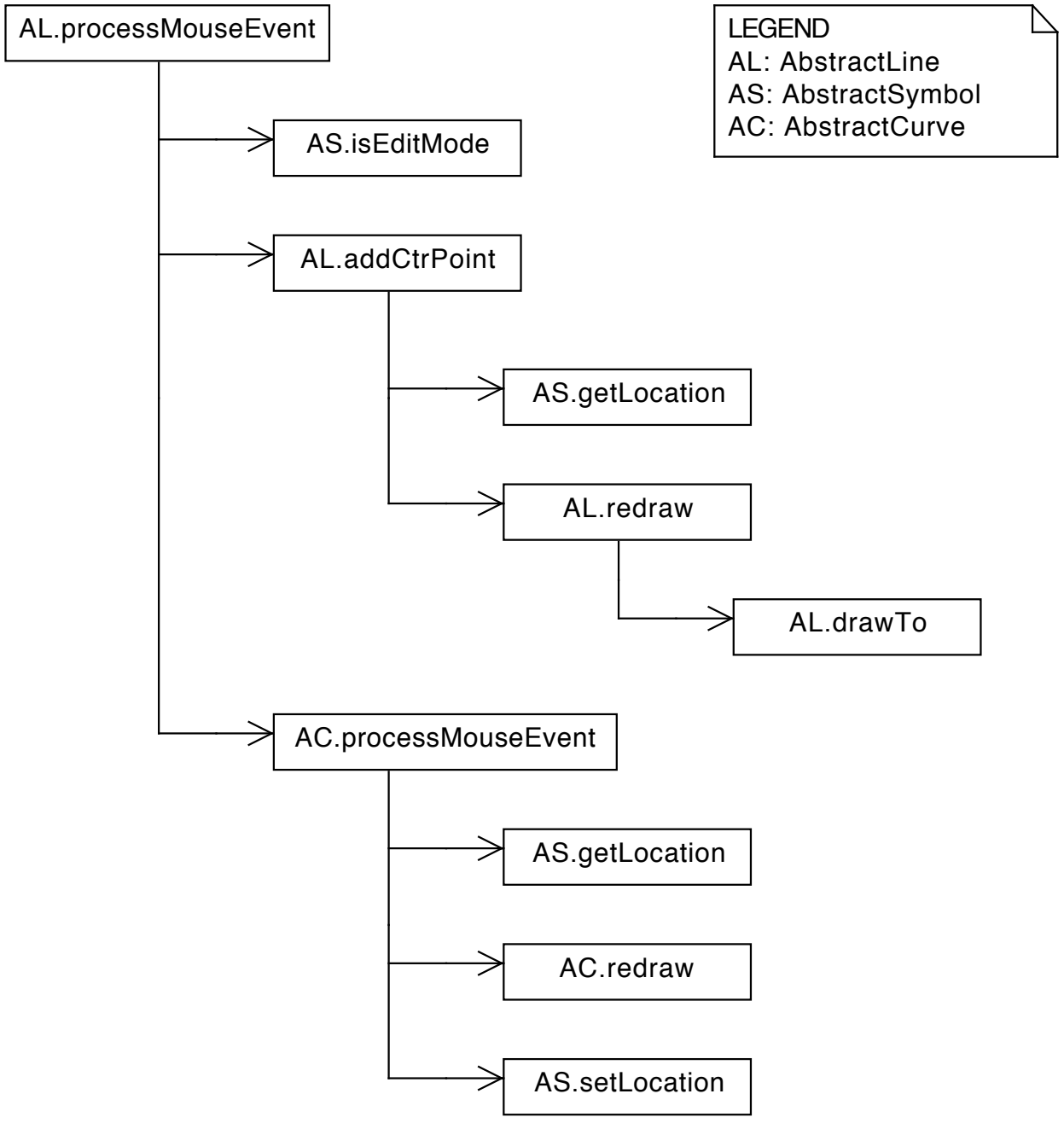
**Question 1. Type Hierarchy**

Draw a type hierarchy that includes all subtypes of AbstractSymbol declared in the `com.marinilli.draw` package. Do not include any class(es) or interface(s) declared in the Java library.



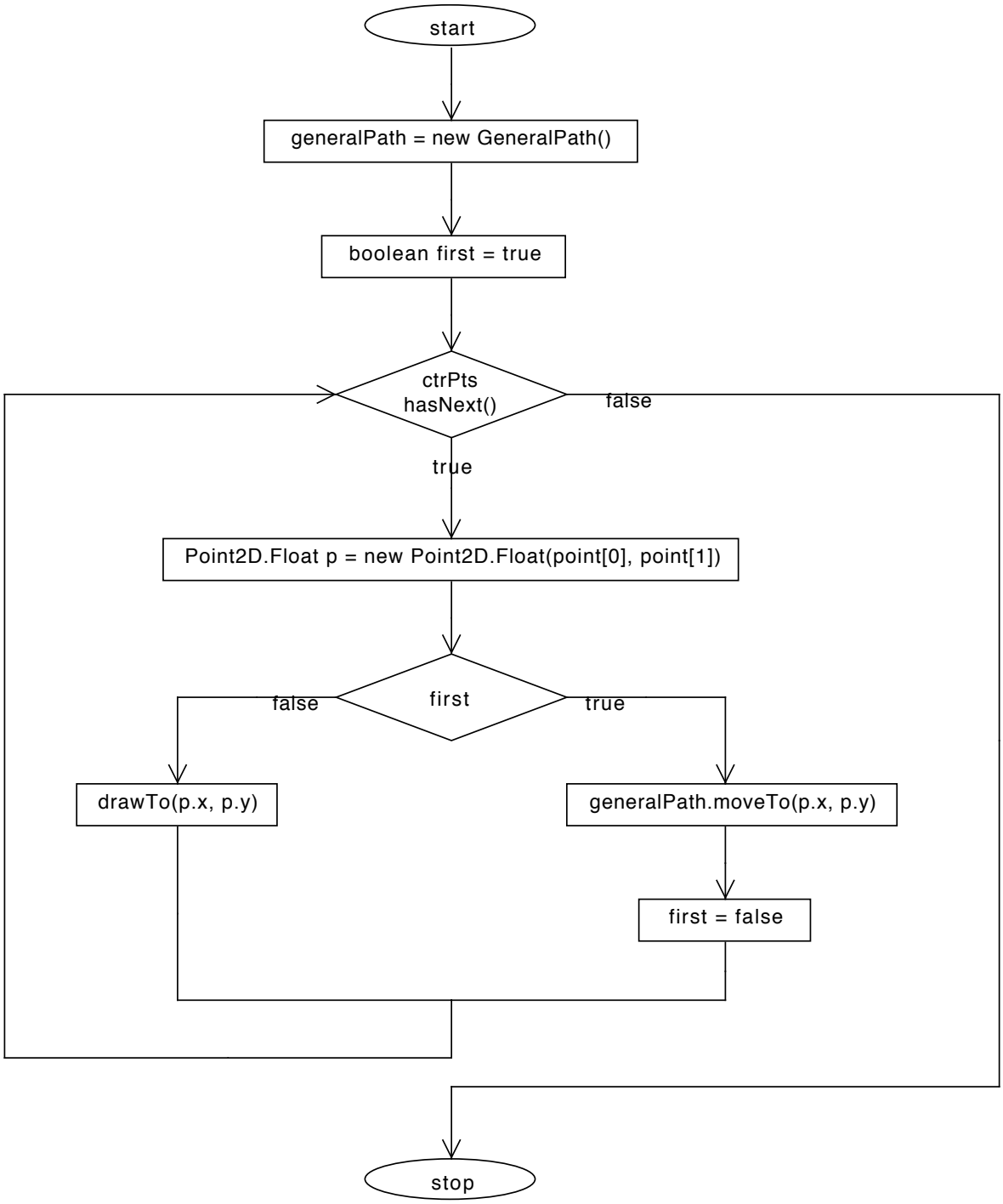
**Question 2. Inter-method Control Flow (Call Graph)**

Draw a call graph starting from the `processMouseEvent(MouseEvent me)` method defined in the `AbstractLine` class of the `com.marinilli.draw` package. Do not include calls to methods in any Java library. If you abbreviate any names, please provide a legend. You might want to rotate the page and draw your graph in landscape mode.



**Question 3. Intra-method Control Flow (Flowchart)**

Draw a flowchart for the `redraw()` method defined in the `AbstractLine` class of the `com.marinilli.draw` package.



**Question 4. Unit Testing**

Consider the following specification for methods of the `Entity` class.

```
public class Entity {

   /**
    * Create a new Entity
    * EFFECTS: this.isForegroundColour is set to false
    */
   public Entity() {…}

   /**
    * Sets colour of entity.
    * REQUIRES: colourString is one of "red", "green" or "blue"
    * MODIFIES: this
    * EFFECTS: if this.isForegroundColour is true, sets foreground
    *          colour of this entity to colour specified by
    *          colourString; otherwise sets background colour of
    *          entity.
    */
   public setColour(String colourString) {…}

   /**
    * Sets a flag to indicate whether or not foreground colour
    * should be processed
    * MODIFIES: this
    * EFFECTS: this.isForegroundColour is set to isForeground
    */
   public void setIsForegroundColour(boolean isForeground) {…}

   /**
    * Get the current foreground colour of the Entity
    * EFFECTS: returns the foreground colour of the Entity
    */
   public String getForegroundColour() {…}

   /**
    * Get the current background colour of the Entity
    * EFFECTS: returns the background colour of the Entity
    */
   public String getBackgroundColour() {…}

}
```

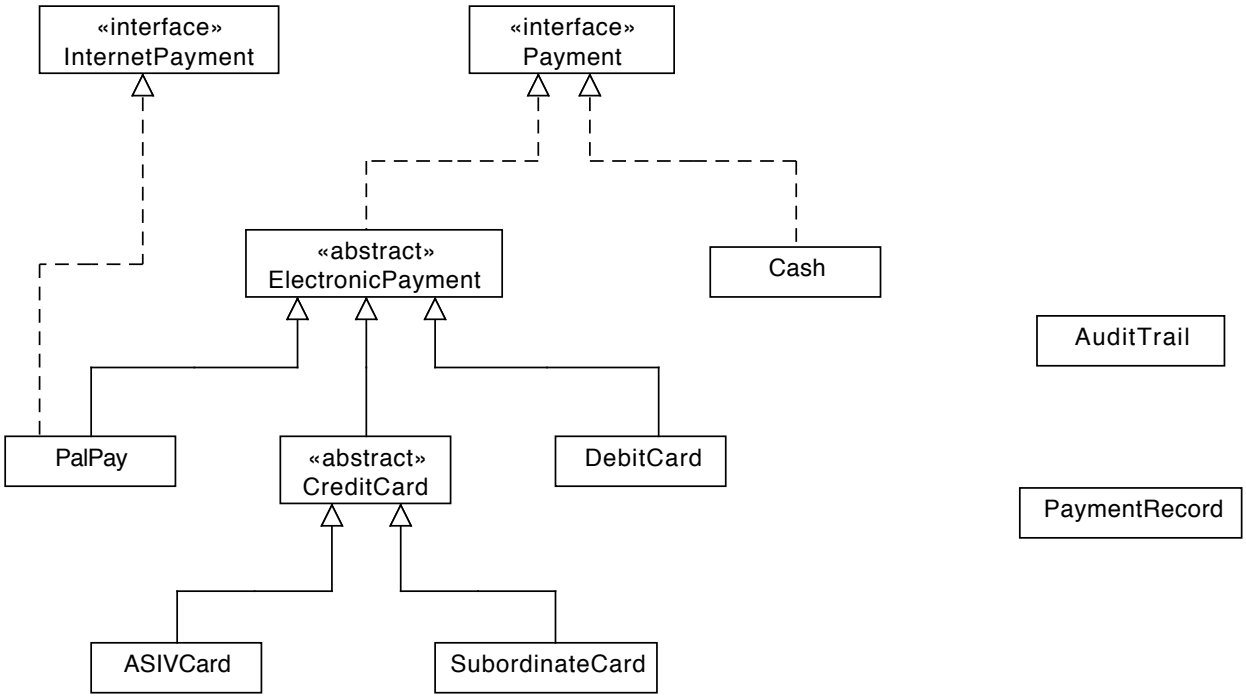
Provide the input and output for all test cases needed to thoroughly test the `setColour` method according to its specification. For each test case, you can assume a new `Entity` object, referred to through a variable named `anEntity`, is created at the start of the test case in a method annotated with `@Before`.

```
@Test
public void testSetForegroundRed() {
    anEntity.setIsForegroundColour(true);
    anEntity.setColour("red");
    assertEquals("red", anEntity.getForegroundColour());
}

@Test
public void testSetForegroundGreen() {
    anEntity.setIsForegroundColour(true);
    anEntity.setColour("green");
    assertEquals("green", anEntity.getForegroundColour());
}

@Test
public void testSetForegroundBlue() {
    anEntity.setIsForegroundColour(true);
    anEntity.setColour("blue");
    assertEquals("blue", anEntity.getForegroundColour());
}

@Test
public void testSetBackgroundRed() {
    anEntity.setIsForegroundColour(false);
    anEntity.setColour("red");
    assertEquals("red", anEntity.getBackgroundColour());
}

@Test
public void testSetBackgroundGreen() {
    anEntity.setIsForegroundColour(false);
    anEntity.setColour("green");
    assertEquals("green", anEntity.getBackgroundColour());
}

@Test
public void testSetBackgroundBlue() {
    anEntity.setIsForegroundColour(false);
    anEntity.setColour("blue");
    assertEquals("blue", anEntity.getBackgroundColour());
}
```

Questions 5 through 8 apply to the `PaymentSystem` provided in the specified repository.

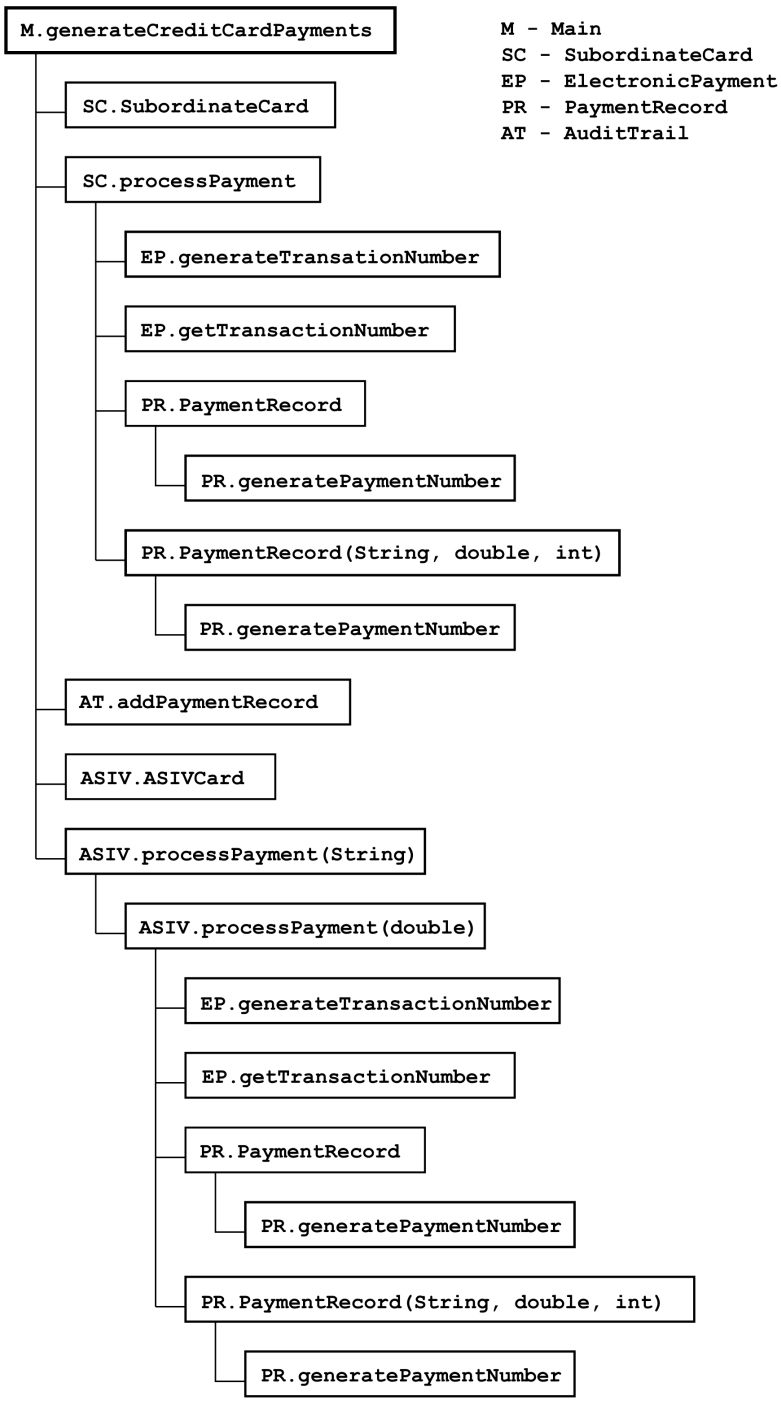
**Question 5. Type Hierarchy**

Draw the type hierarchy for all types declared in the `ca.ubc.cpsc210.payment.model` package. Use directional arrows to relate subtypes to supertypes in the drawing (i.e., lines between types should have an arrowhead only at one end; lines should go from the subtype to the supertype with the arrowhead at the supertype).



## Question 6. Call Graph

Draw a call graph starting from the `generateCreditCardPayments(AuditTrail auditTrail)` function defined in the `Main` class (Main.java). Stop following method calls for any method defined in a class outside of `ca.ubc.cpsc210.payment.model`. You might want to sketch the call graph on a scrap piece of paper before placing it on this sheet. You can also rotate the paper and write in landscape mode for more space. If you abbreviate any names, please provide a legend.




**Note:**

- if there are multiple calls to method B from method A, we record the call only once on the call graph

- to distinguish between calls to different versions of an overloaded method, we include parameter lists. For example:

`ASIV.processPayment(String)`

and

`ASIV.processPayment(double)`

represent calls to different methods in the same class.

**Question 7. Types.**

Consider the following code:

```
(1) Payment p;
(2) p = new DebitCard(3, 4);
(3) InternetPayment i = new PalPay();
```

**i)** What is the actual type of the variable `p` at the statement numbered (2) after the statement executes?

**`DebitCard`**

**ii)** What is the apparent type of the variable `p` at the statement numbered (2) after the statement executes?

**`Payment`**

**iii)** What is the apparent type of the variable `i` at the statement numbered (3) after the statement executes?

**`InternetPayment`**

**iv)** What is the actual type of the variable `i` at the statement numbered (3) after the statement executes?

**`PalPay`**

**Question 8. Debugging.**

If you run the Main class as a Java application, the output will include the following:

```
Payment[ num=15, type=PalPay, amt=0.724302501394058, txNum=15]
Payment[ num=16, type=PalPay, amt=1.2554252514453499, txNum=16]
Payment[ num=-83, type=Cash, amt=0.0]
Payment[ num=-82, type=Cash, amt=0.3682269387159234]
```

Note that the last two lines of this output have a negative payment number, which is illegal according to the specification of the PaymentRecord data abstraction. Generate two hypotheses about what might be causing this error.

**Given that negative payment numbers appear to be generated only for `Cash` type payments, we generate the following hypotheses:**

**(1) payment numbers are generated incorrectly when the type is `Cash` payment**
**(2) payment numbers are printed incorrectly when the type is `Cash` payment**

**(Extra credit.) What is actually causing the error in the output shown above?**

**The second hypothesis above is correct. The error is in the `toString` method of the `PaymentRecord` class:**

```
if (typeOfPayment.equals("Cash"))
    repesentationAsString =
        repesentationAsString.concat(paymentNumber-100 + ", ");
```

**should be:**

```
if (typeOfPayment.equals("Cash"))
    repesentationAsString =
        repesentationAsString.concat(paymentNumber + ", ");
```

**Question 9. Specification**

Suppose you are designing a new data type to represent a fare box on a bus. The fare box accepts pre-paid tickets and cash (in the form of coins only). When a ticket is inserted into the machine, the value of the ticket is read and that amount is added to the total fare collected. The amount of the fare is deducted from the ticket. When coins are inserted, their value is added to the total fare collected. Write the specification for the `payByTicket` and `payByCash` methods:

```
public class FareBox {
    private int totalFareCollected;   // in cents

    // Modifies: this, t
    // Effects: value of ticket is added to total fare collected
    //          and value is deducted from t
    public void payByTicket(Ticket t) {
        …
    }

    // Modifies: this
    // Effects: value of coins is added to total fare collected
    public void payByCash(int coinValue) {
        …
    }
}
```

**Note:** it would not be unreasonable to put a requires clause on `payByTicket` to indicate that the ticket is valid – in other words, that it hasn't already been used.

**Question 10. Data Abstraction:** The `SimGame` project contains the partial specification and implementation for a `SimPet` class, along with associated unit tests. The `SimPet` represents a pet in a simulated world. Each pet has a location in the two-dimensional world and an energy level. A pet can be pointing in one of only four directions: North, South, East or West. We assume that the pet's location is specified using integer coordinates. In this question, we do not concern ourselves with the size of the world – so we don't worry about pets walking off the edge.

We want to be able to feed the pet and specify the number of units of energy it eats, assumed to be an integer value. We also want to be able to move the pet one unit in whatever direction it is currently pointing. *Each time the pet moves, it consumes one unit of energy*. We also want to be able to rotate the pet left or right by 90 degrees so that it can move in different directions. When a pet rotates, it does not consume any energy. If the pet's energy level drops to zero, it dies.

In this question, you can write your code in Eclipse but you **must** copy it on to this exam paper **before** the end of the exam – there is no electronic submission! Note that it is not necessary to copy the comment statements.

**a)** Write the implementation of the `SimPet` constructor. Run the JUnit tests provided in `ca.ubc.cpsc210.simgame.test.TestSimPet` and ensure that `testConstructor` passes.

```
public SimPet(int x, int y, int initialEnergy) {
     this.x = x;
     this.y = y;
     this.energy = initialEnergy;
     this.direction = 0;
     this.hasHadShots = false;
}
```

**b)** Write the implementation of the `SimPet.move` method. Run the JUnit tests provided and ensure that they all pass.

```
public void move() {
     if (energy > 0) {
          if (direction == 0)
               x = x + 1;
          else if (direction == 1)
               y = y + 1;
          else if (direction == 2)
               x = x - 1;
          else if (direction == 3)
               y = y - 1;
          energy = energy - ENERGY_TO_MOVE;
     }
}
```

**c)** Now suppose we want to add a method that will give a pet its shots. Write the specification for a method `SimPet.giveShots` and include a stub for this method. Assume that a pet can be given its shots only if it has an energy level of at least 5 and hasn't already had its shots. Note that a pet does not consume any energy when it is given its shots. Write your specification in such a way that there is no *requires* clause.

```
// REQUIRES:
// MODIFIES:  this
// EFFECTS:   if this pet has energy level at least 5 and it has
//            not had its shots, then record that the pet has now
//            had its shots
public void giveShots() {
    // stub
}
```

**d)** In this part of the question, we ask you to demonstrate how to *use* a data abstraction. Write code that will create a new `SimPet` object located at the origin with 10 units of energy. Your code must then rotate the `SimPet` so that it is pointing west and move it forward 5 steps. Finally, declare a variable of an appropriate type and assign to it the amount of energy that your pet has remaining after rotating and moving.

```
public void doStuff() {
     SimPet p = new SimPet(0, 0, 10);
     p.rotateLeft();
     p.rotateLeft();
     for (int i = 0; i < 5; i++) {
          p.move();
     }
     int remainingEnergy = p.getEnergy();
}
```

**Question 11. Data Abstraction:**

The `ca.ubc.cs.cpsc210.kafe.CoffeeCard` class in the `KafeCompany` project contains a partial specification for a data type that represents a loyalty card for the Kafe company. A coffee card can be loaded with credits that can be used to purchase drinks at Kafe stores. Every time a drink is purchased, a bean is added to the card. For every 9 beans earned, a free drink is added to the card. To be purchased, some drinks require more credits than others. However, only one bean is earned per drink purchase, regardless of the number of credits required to purchase the drink.

Study the provided code for the `CoffeeCard` class carefully before continuing.

**a)** Suppose the topUp method has the following implementation rather than the one provided in the CoffeeCard class checked out of the repository.

```
public void topUp(int numCredits) {
      if (numCredits > 0)
      credits += numCredits;
}
```

Write the specification for the version shown above.

```
// top up credits
// MODIFIES: this
// EFFECTS: adds numCredits to number of credits on card
// only if numCredits > 0
```

**b)** Design jUnit tests for the `CoffeeCard.useFreeDrink` method – be sure to study the specification for this method carefully. Don't worry if your Java syntax isn't perfect but note that it may help you to examine the tests provided in the `CoffeeCardTests` class. You must assume that the method `CoffeeCardTests.runBefore` runs before each of your tests. If you are unsure about your syntax, include comments to explain what you are trying to do. Note that there's more space for your answer to this question on the following page. You may use Eclipse to develop your solution but you must make a copy of your work onto the exam paper *before the end of the exam*.

```
@Test
public void testUseFreeDrinkNoneAvailable() {
      assertFalse(card.useFreeDrink());
}
```

```
@Test
public void testUseFreeDrinkWhenAvailable() {
      // add credits to purchase enough drinks to earn a free one
      card.topUp(CoffeeCard.BEANS_PER_FREE_DRINK);

      // buy enough drinks to earn a free one
      for (int i = 0; i < CoffeeCard.BEANS_PER_FREE_DRINK - 1; i++) {
            assertTrue(card.purchaseDrink(1));
      }

      assertEquals(1, card.getFreeDrinks());   // not strictly needed
      assertTrue(card.useFreeDrink());
      assertEquals(0, card.getFreeDrinks());
}
```

## Question 12. Data Abstraction

Write an implementation for the `CoffeeCard.purchaseDrink` method. Note that some tests are provided for you in the `CoffeeCardTests` class but do not assume that these tests will catch every possible bug in your code. You may use Eclipse to develop your solution but you must make a copy of your work onto the exam paper *before the end of the exam*. It is not necessary to copy the provided documentation/comments.

```
public boolean purchaseDrink(int numCredits) {
      if (credits < numCredits)
            return false;

      credits -= numCredits;
      beans++;

      if (beans >= BEANS_PER_FREE_DRINK) {
            freeDrinks++;
            beans = 0;
      }

      return true;
}
```

**Question 13: Debugging**

The class `ca.ubc.cs.cpsc210.tests.ContactTests` contains three unit tests for the `Contact` class in the `ca.ubc.cs.cpsc210.addressbook` package. Run these tests and notice that all of them fail. Note that each test identifies a single software bug in the code. **For each test**:

- write the name of the test
- indicate how would fix the software bug identified by that test by writing a correct implementation of the method that contains the bug. Note that it is not necessary to copy the method's documentation (comment statements).

Note that the problem might be with the test rather than the method it is testing. In this case, you should re-write the test. You may use Eclipse to develop your solution but you must make a copy of your work onto the exam paper *before the end of the exam*.

**testOneParamConstructor – bug is in the test**

```
public void testOneParamConstructor() {
      Contact c = new Contact("Joey");
      assertEquals("Joey", c.getName());
}
```

**testTwoParamConstructor - bug is in the constructor**

```
public Contact(String name, String eMailAddress) {
      super(name);
      this.eMailAddress = eMailAddress;
}
```

**testGetAddressList – bug is in the `getAddressList` method**

```
public List<String> getAddressList() {
      List<String> al = new LinkedList<String();
      al.add(eMailAddress);
      return al;
}
```