Question 1. [7 MARKS]

For each of the code fragments below, state whether they are valid Java code, and if not, why. Write your answer to the right of the code fragment in question. An ellipsis (...) means irrelevant code has been omitted.

Part (a) [1 MARK]

```
abstract class A {
    public void a() {
        ...
    }
}

class B extends A {
    public void a() {
        super.a();
    }
}
```

Valid

Part (b) [1 MARK]

```
abstract class A {
    public abstract void a();
}

class B extends A{
    public void a() {
        super.a();
    }
}
```

Invalid: you cannot call super.a() because the superclass method is abstract.

Part (c) [1 MARK]

```
interface A { ... }

class B {
    public void a(A x) { ... }
}
```

Valid

Part (d) [2 MARKS]

```
class A extends Exception { ... }

class B extends A { ... }
```


CONT'D...

```
public class C {
    public void a() throws A {
        throw new B();
    }
}
```

Valid

Part (e) [2 MARKS]

```
public class D {
    public class C {
        public int x;
    }

    public static void main(String[] args) {
        C[] a = new C[10];
        for (int i = 0; i < a.length; i++) {
            a[i].x = i;
        }
    }
}
```

Array was created but no elements of the array refer to objects of class C.

CONT'D...

Question 2. [8 MARKS]

Part (a) [5 MARKS]

Each row in the table below gives first a regular expression, and secondly a string. Fill in the entry in the third column: either the part of the string that the regular expression matches, or "NO MATCH" if there is no match.

You can assume that no regular expression or string has any white space before the first visible character or after the last. If your answer includes blank characters, please show clearly where they are.

REGEX	STRING	MATCH
`a*bc`	`bcaaabc`	`bc`
`a*bc$`	`bcaaabc`	`aaabc`
`cb*a*bc$`	`bcaaabc`	`caaabc`
`^.*,.*,.*,.*$`	`Hi, mom, I'm home, for good, now`	(whole string)
`\s.*\s`	`Please wipe your feet`	`wipe your` (including blanks before

Part (b) [3 MARKS]

People have honorific titles like "Mrs.", "Ms.", "Mr.", and "Sir". Write a regular expression that matches any one of these four titles. Shorter solutions are better.

`(Mr?s?\.`Sir)—

Question 3. [12 MARKS]

Part (a) [2 MARKS]

Briefly describe the difference between a Product Backlog and a Sprint Backlog.

Part (b) [10 MARKS] Circle `True` or `False` as appropriate.

Don't guess: if you circle the correct answer, you earn a mark. If you circle the wrong answer, you lose a mark. If you do not circle either answer, there is no mark. (The minimum mark on this question is 0.)

Scrum is an agile process with a strong emphasis on up-front, detailed design related to software development. True False

False
Scrum is an agile process with a strong emphasis on managerial guidelines for software development. True False

True
In Scrum, it's important that the client understand what each team member is doing. True False

False
In Scrum, the client assigns difficulty estimates to each item on the Product Backlog. True False

False
In Scrum, the Scrum Master assigns difficulty estimates to each item on the Sprint Backlog. True False

False
The Product Backlog acts as a contract between the client and the team, and everyone involved in the project must agree to any proposed changes. True False

False
Daily Scrum meetings are intended to solve detailed technical problems. True False

False
The Sprint Backlog is a list of everything that needs to be done during the sprint, and should never change throughout the sprint. True False

False
At the beginning of each sprint, the Product Owner assigns tasks to the team members. True False

False
At the end of each sprint, the team posts the new version of the program on an open-source repository like SourceForge or Google Code so that it can be beta tested. True False

False True False

Note to markers: -1 for each wrong answer. Note that students can get a zero even if they get five correct.

CONT'D...

Question 4. [20 MARKS]

Read the next page first. This only comes first for layout reasons.

```
/** A document item with possibly many children, which are also DocItems. */
abstract class DocItem {

    /** This DocItem's children. */
    List<DocItem> children = new ArrayList<DocItem>();

    /** Append i to this DocItem's children. */
    void add(DocItem i) {
        this.children.add(i);
    }

    public Iterable<DocItem> getChildren() {
        return children;
    }
}

// Structure classes.

class DocRoot extends DocItem {}
class DocHead extends DocItem {}
class DocSubHead extends DocItem {}
class DocList extends DocItem {}
class DocListItem extends DocItem {}

// Text classes.

abstract class DocText extends DocItem {

    private String text;

    DocText(String s) {
        this.text = s;
    }

    public String getText() {
        return text;
    }
}

class DocPlainText extends DocText {
    DocPlainText(String s) {
        super(s);
    }
}

class DocBoldText extends DocText {
    DocBoldText(String s) {
        super(s);
    }
}
```

```
// Build a document tree and visit it.

DocRoot doc = new DocRoot();

DocHead h1 = new DocHead();
h1.add(new DocPlainText("first header\n"));

DocSubHead h2 = new DocSubHead();
h2.add(new DocPlainText("second header\n"));

DocList dl = new DocList();

DocText dt1 = new DocPlainText("item 1\n");
DocListItem dli1 = new DocListItem();
dli1.add(dt1);

DocListItem dli2 = new DocListItem();
DocText dt2 = new DocPlainText("item 2 ");
DocText db = new DocBoldText("moo");
DocText i3 = new DocPlainText(" gah\n");
dli2.add(dt2);
dli2.add(db);
dli2.add(i3);
dl.add(dli1);
dl.add(dli2);

DocText plain = new DocPlainText("plain\n");

doc.add(h1);
doc.add(h2);
doc.add(dl);
doc.add(plain);

WikiPrintVisitor pv = new WikiPrintVisitor();
doc.accept(pv);
```


CONT'D...

Consider this structured text, which uses two kinds of headings, a list, plain text, and bold text:

first header
second header

- item 1
- item 2 **moo** gah

plain

The previous page includes code that builds the following tree that represents the structured text:

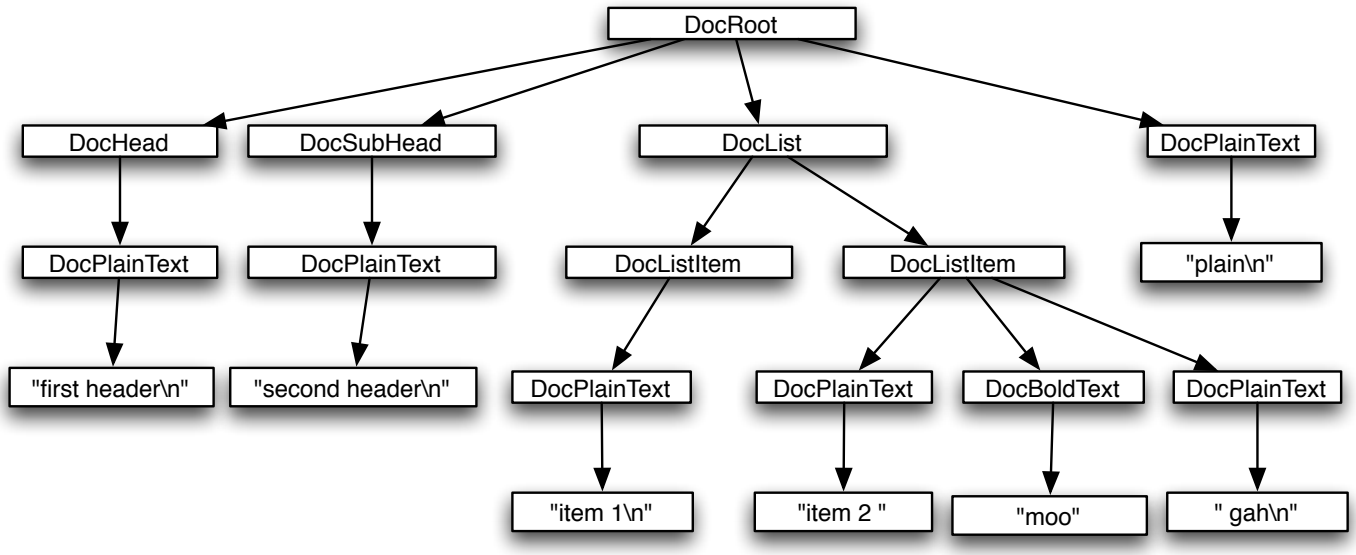

Figure 1: Document structure: the boxes are kinds of `DocItem` objects, except for the leaves, which are Java `String`s.

Part (a) [3 MARKS]

A `DocText` object contains a `String`, and `DocText` objects are not supposed to have any other kind of child. Therefore, the inherited list `children` is not used in any of the `DocText` subclasses. Because of this, it doesn't make sense to allow code to call method `add` on `DocText` objects without giving some kind of warning. Write code for this: override the `DocText add` and `getChildren` methods so that they do something sensible:

The rest of this question concerns the implementation of the Visitor pattern.

Part (b) [1 MARK]

You've probably noticed by now that the code creates a `WikiPrintVisitor` that is applied to a document. In preparation for making the code work, write the `Visitable` interface as found in the Visitor design pattern:

CONT'D...

Part (c) [2 MARKS]

The `DocItem` class needs to change so that it can be visited in the Visitor pattern. Edit the code for class `DocItem` on the first page of this question to allow for this. (In Part (g), you will write the Visitor interface.)

Part (d) [2 MARKS] The `DocBoldText` class now needs a new method. Write it here:

Part (e) [1 MARK] The `DocPlainText` class also needs a new method. Write it here:

Part (f) [3 MARKS] The `DocList` class also needs a new method. (Think of the children!) Write it here:

Part (g) [3 MARKS]

The `Visitor` interface for a tree of `DocItem`s needs to have one method for each kind of item. Write the `Visitor` interface:

Part (h) [5 MARKS]

Throughout your project you used a wiki to update your progress and describe your work. Assume we have a wiki that uses only the following simplified wiki syntax:

- A sequence of `#`'s marks the level of a heading
- An unordered list uses `*` as a prefix. (There are no nested lists.)
- Text wrapped in `**`'s (two asterisks on either side of text) indicates bold.

Here is the document from Figure 1 as it might appear in a wiki using our simplified wiki syntax:

```
#first header
##second header
*item 1
*item 2 **moo** gah
plain
```

On the following page, write a `WikiPrintVisitor` class that implements your `Visitor` interface. Only write code that deals with `DocList`, `DocListItem`, `DocPlainText`, and `DocBoldText`; don't bother with code for `DocRoot`, `DocHead`, and `DocSubHead`.

Use `System.out.print` instead of `System.out.println`; the newlines are already in the `String`s.

Again, you do *not* need to implement all the Visitor methods: only write the methods that pertain to `DocList`, `DocListItem`, `DocPlainText`, and `DocBoldText`.

CONT'D...

Write your `WikiPrintVisitor` class here.

Question 5. [15 MARKS]

Here are the Javadoc and method signatures for a `Part` class. A `Part` models a component of a machine or other manufactured device, and might be built from sub-parts.

```
/**
 * A Part is uniquely identified by its part number. The part number is
 * consistently named "partNum" in method signatures.
 *
 * A Part may be built up from other Parts. Parts may be sorted on the basis
 * of their partNums.
 */
class Part implements Comparable<Part> {

    /**
     * Create a Part with given part number, name and cost. The cost is
     * additional to the cost of any other Parts used in manufacturing this
     * Part. If the partNum already exists, throws DuplicatePartNumException.
     *
     * @param partNum   The part number of this Part.
     * @param name      The name of this Part.
     * @param cost      The cost of this Part.
     */
    public Part(int partNum, String name, double cost) {...}

    /**
     * Compare this Part with other Part on the basis of their partNums.
     *
     * @param other The other Part.
     */
    public int compareTo(Part other) {...}

    /**
     * Recognize that this Part requires howMany copies of other Part as
     * components; if other is the same kind of Part as this Part, throw
     * a RecursiveManufacturingException; if howMany is not positive,
     * throw BadCountException.
     *
     * @param other     The other Part to be used in manufacturing this Part.
     * @param howMany   The number of other Parts needed in this Part.
     */
    public void builtFrom(Part other, int howMany) {...}

    /**
     * Return the cost of manufacturing this Part, which must be the cost
     * of this Part, provided in the constructor, plus the cost of any
     * other Parts used in manufacturing it. The caller is resonsible for
     * ensuring that this Part is not directly or indirectly included as a
     * component of itself.
     *
     * @return  double  The total cost of manufacturing this Part.
     */
    public double getTotalCost() {...}
}
```


CONT'D...

Complete the `Part` class so that with your code included, it meets the specifications stated in the Javadoc.

Your mark will be better if you use standard classes from the Java API rather than inventing your own solutions. Pay careful attention to how you define and use exceptions.

You will need to add instance variables, methods and classes. You may also need to modify the existing Javadoc; however, you may not *contradict* the existing descriptions. Try to avoid re-writing the existing Javadoc and code, unless that is easier than stating changes.

```
import java.util.*; // not required for full marks

// Note to marker: this code compiles, at any rate.

class Part implements Comparable<Part> {

    private static Set<Integer> partNums = new HashSet<Integer>(); // ADDED
    private Map<Part, Integer> components = new HashMap<Part, Integer>(); // ADDED
    private int partNum; // ADDED
    private String name; // ADDED
    private double cost; // ADDED


    /**
     * Create a Part with given part number, name and cost. The cost is
     * additional to the cost of any other Parts used in manufacturing this
     * Part. If the partNum already exists, throws DuplicatePartNumException.
     *
     * @param partNum   The part number of this Part.
     * @param name      The name of this Part.
     * @param cost      The cost of this Part.
     * @throws DuplicatPartNumException If partNum is not new to the system.
     */
    public Part(int partNum, String name, double cost)
            throws DuplicatePartNumException {
        if (partNums.contains(partNum)) // or whatever
            throw new DuplicatePartNumException();
        this.partNum = partNum;
        this.name = name;
        this.cost = cost;
    }

    /**
     * Compare this Part with other Part on the basis of their partNums.
     *
     * @param other The other Part.
     */
    public int compareTo(Part other) {
        return this.partNum - other.partNum;
```


CONT'D...

```
}

/**
 * Recognize that this Part requires howMany copies of other Part as
 * components; if howMany is not positive, throw BadCountException;
 * if other already added to this Part's components, throw
 * DuplicatePartNumException.
 *
 * @param other     The other Part to be used in manufacturing this Part.
 * @param howMany   The number of other Parts needed in this Part.
 * @throws DuplicatePartNumException    If this Part already uses other.
 * @throws BadCountException            If howMany is less than 1.
 */
public void builtFrom(Part other, int howMany)
        throws DuplicatePartNumException, BadCountException {
    if (components.containsKey(other))
        throw new DuplicatePartNumException();
    if (howMany < 1)
        throw new BadCountException();
    components.put(other, howMany);
}

/**
 * Return true or false according to other is a Part with the same partNum
 * as this Part.
 */
public boolean equals(Object other) {
    if (! (other instanceof Part))
        return false;
    else if (other == null)
        return false;
    else
        return this.partNum == ((Part) other).partNum;
}

/**
 * Return the cost of manufacturing this Part, which must be the cost
 * of this Part, provided in the constructor, plus the cost of any
 * other Parts used in manufacturing it. The caller is resonsible for
 * ensuring that this Part is not directly or indirectly included as a
 * component of itself.
 *
 * @return  double  The total cost of manufacturing this Part.
 */
public double getTotalCost() {
    double result = cost;
    for (Part component : components.keySet())
        result += components.get(component) * component.getTotalCost();
```


CONT'D...

```
        return result;
    }
}

class DuplicatePartNumException extends Exception {} // OK to make it internal
class BadCountException extends Exception {}
```


CONT'D...