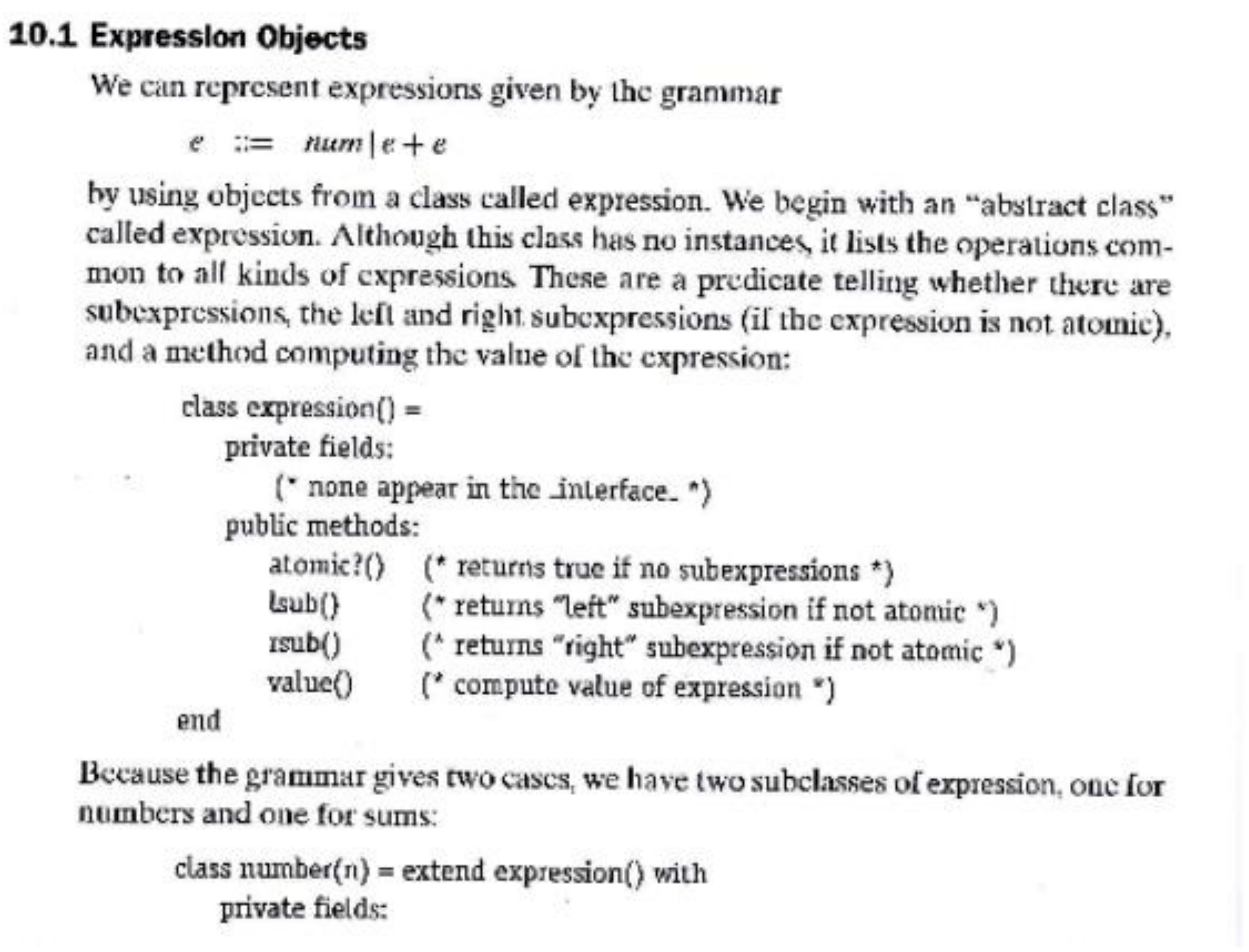
Problem 1



10.1 Expression Objects

We can represent expressions given by the grammar

$$e \quad ::= \quad num \mid e + e$$

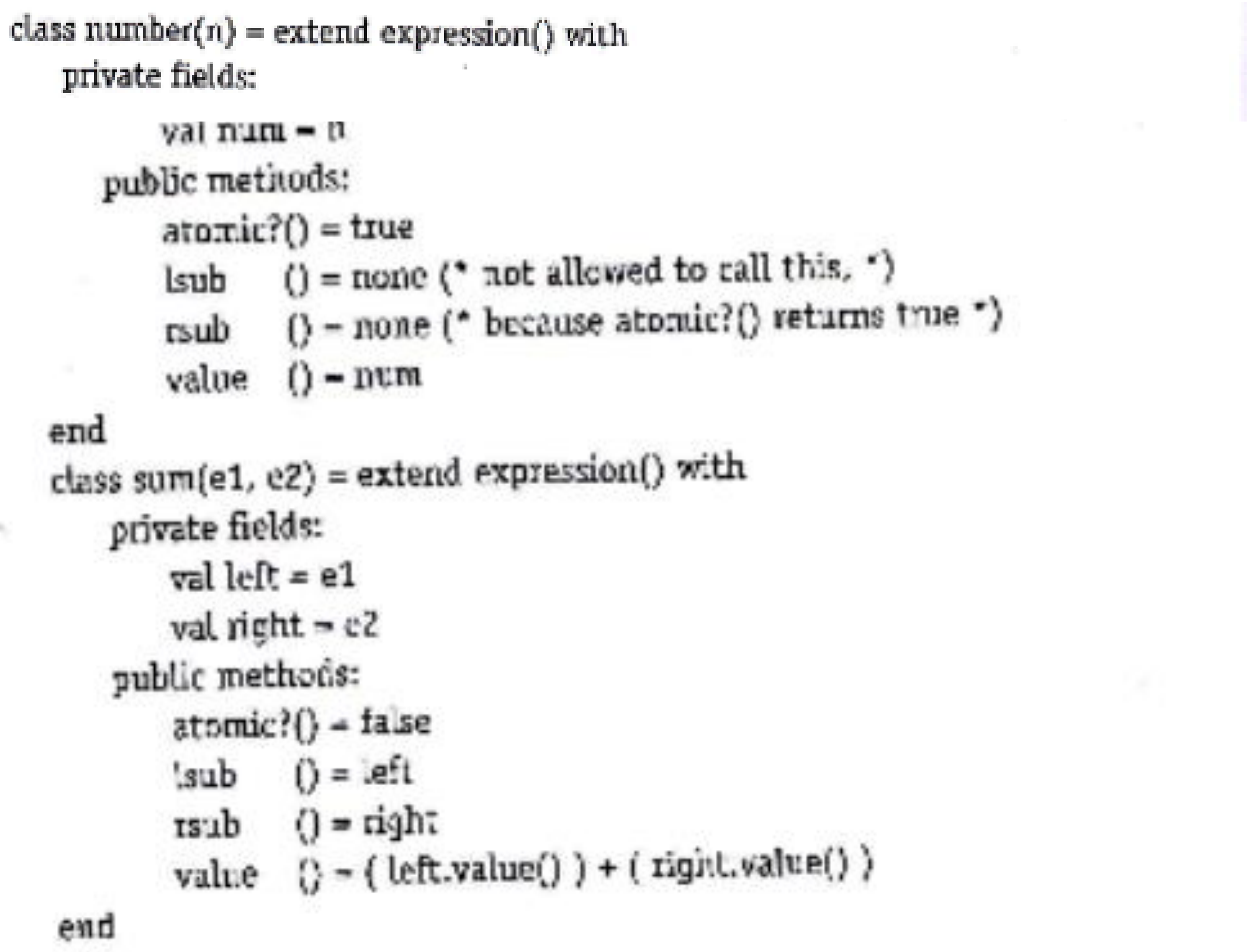
by using objects from a class called expression. We begin with an "abstract class" called expression. Although this class has no instances, it lists the operations common to all kinds of expressions. These are a predicate telling whether there are subexpressions, the left and right subexpressions (if the expression is not atomic), and a method computing the value of the expression:

```
class expression() =
    private fields:
        (* none appear in the _interface_ *)
    public methods:
        atomic?()   (* returns true if no subexpressions *)
        lsub()      (* returns "left" subexpression if not atomic *)
        rsub()      (* returns "right" subexpression if not atomic *)
        value()     (* compute value of expression *)
end
```

Because the grammar gives two cases, we have two subclasses of expression, one for numbers and one for sums:

```
class number(n) = extend expression() with
    private fields:
```

Problem 1



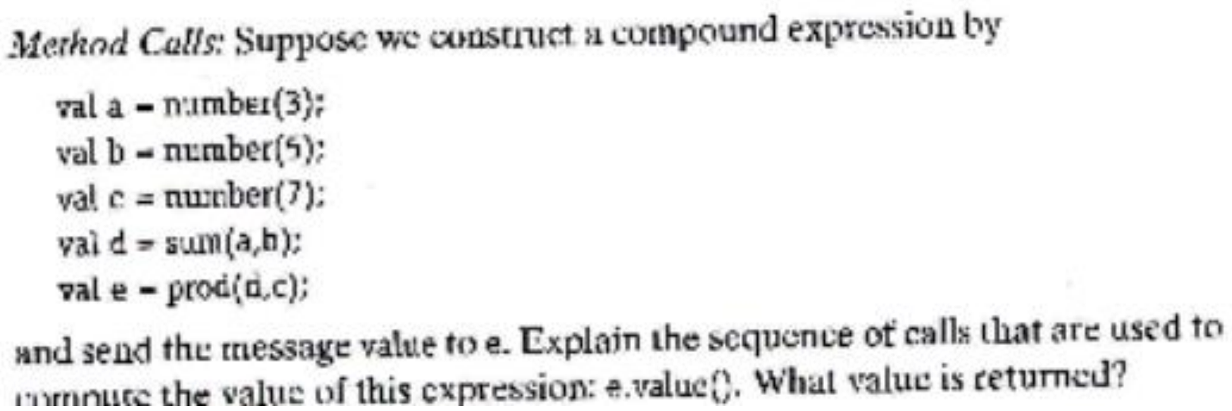
```
class number(n) = extend expression() with
    private fields:
            val num = n
        public methods:
            atomic?() = true
            lsub    () = none (* not allowed to call this, *)
            rsub    () = none (* because atomic?() returns true *)
            value   () = num
  end
  class sum(e1, e2) = extend expression() with
       private fields:
            val left = e1
            val right = e2
       public methods:
            atomic?() = false
            lsub    () = left
            rsub    () = right
            value   () = ( left.value() ) + ( right.value() )
  end
```

Problem 1 a)

- Extend the class hierarchy to include a class for product expressions e := … | e * e

```
class prod(e1,e2)=extend expression()
with
 private fields:
   val left = e1
   val right = e2
 public methods:
   atomic?() = false
   lsub()= left
   rsub()= right
   value()=(left.value()*right.value())
end
```

Problem 1b)



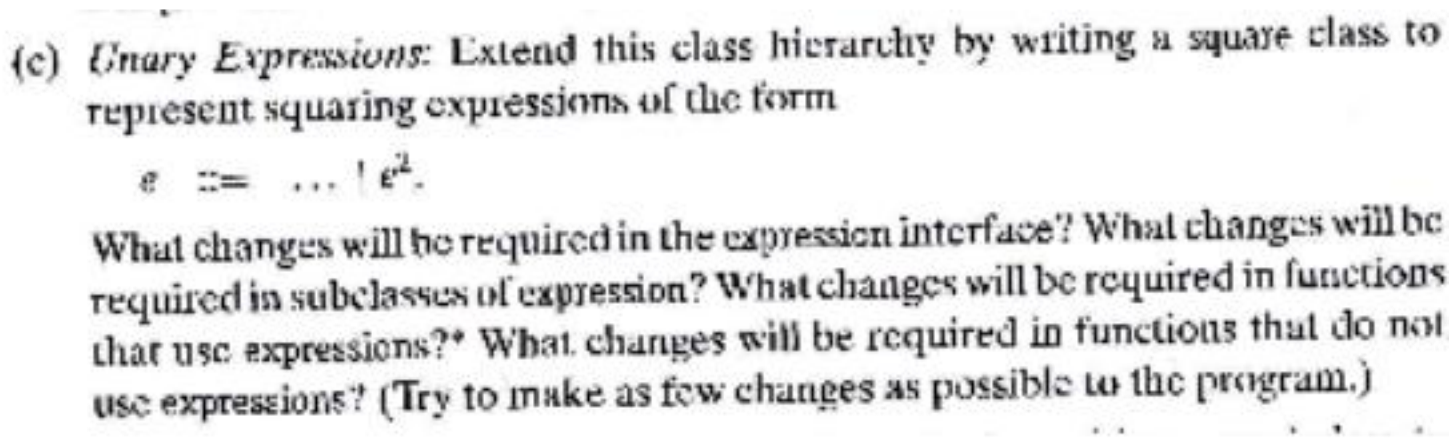
Method Calls: Suppose we construct a compound expression by

```
val a = number(3);
val b = number(5);
val c = number(7);
val d = sum(a,b);
val e = prod(d,c);
```

and send the message value to e. Explain the sequence of calls that are used to compute the value of this expression: e.value(). What value is returned?

e.value() =
e.left.value() * e.right.value() =
(d.left.value + d.right.value) * 7 =
(3 + 5) * 7 = 56

Problem 1 c)



(c) *Unary Expressions:* Extend this class hierarchy by writing a square class to represent squaring expressions of the form

$$e \;::=\; \ldots \mid e^2.$$

What changes will be required in the expression interface? What changes will be required in subclasses of expression? What changes will be required in functions that use expressions?* What changes will be required in functions that do not use expressions? (Try to make as few changes as possible to the program.)

A minimal extension of the expression abstract class uses the lsub() to represent the unary operand:

```
class expression()=
private fields:  (* none *)
  public methods:
    atomic?() (* returns true if an atomic expression *)
    unary?()  (* returns true if a unary expression *)
    lsub()    (* returns the left subexpression if not atomic,
                 or the operand of a unary expression *)
    rsub()    (* returns the left subexpression if not atomic,
                   but none in case of a unary expression *)
    value()
end
```

```
class square(e)= extend expression() with
  private fields:
    val sub = e
public methods:
    atomic?() = false
    unary?() = true
    lsub()= sub
    rsub()= none
    value()=(sub.value()*sub.value())
end
```

Problem 1c, alternative solution

Alternatively a separate field sub is introduced to keep this operand:

```
class expression()=
  private fields:
    (* none *)
  public methods:
    atomic?() (* returns true if a atomic expression *)
    unary?() (* returns true if a unary expression *)
    sub()    (* returns the operand of a unary expression *)
    lsub()   (* returns the left subexpression if not atomic,
             and not unary *)
    rsub()   (* returns the right subexpression if not atomic,
             and not unary *)
    value()
end
```

```
class square(e)=extend expression() with
  private fields:
    val sub = e
public methods:
    atomic?() = false
    unary?() = true
    sub()= sub
    lsub()= none
    rsub()= none
    value()=(sub.value()*sub.value())
end
```

Problem 2 a)

```
class Rect {
  Point ul;         // upper left corner
  Point lr;         // lower right corner

  void setUL(Point newUL){ this.ul = newUL;};
};

class ColorRect extends Rect {
  ColorPoint ul;
  ColorPoint lr;
}
```

We now have two variables with name ul, and two with named lr

What if Java instead would allow the original ul to be redefined to have type ColorPoint in ColoreRect, and correspondingly for lr. What type errors could occur?

The variables could be assigned values of the superclass Point, even though they were typed with ColorPoint → type error

Problem 2 b)

b) How would this be done if Java had virtual classes?

- A virtual class is an inner class that can be redefined in a subclass, much like a virtual method

```
class Point {
  int x, y;
  virtual class ThePointClass <: Point;

  bool equal(ThePointClass p) {
    return x == p.x && y == p.y;
  }
}

class ColorPoint extends Point {
  Color c;
  ThePointClass := ColorPoint;

  bool equal(ThisPointClass p) {
    return super.equal(p) && c == p.c;
} }
```

Syntax made-up!

Problem 2 c)

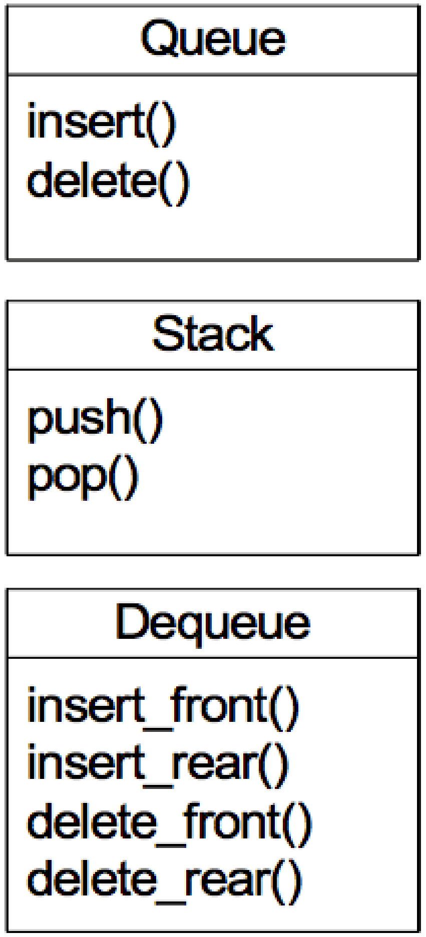
Virtual classes are not part of Java. Would a cast help, like in this redefinition of setUL in ColorRect:

```
class ColorRect extends Rect {

  void setUL(Point newUL){
    this.ul = (ColorPoint)newUL;
  }
}
```

No, this would not help much. Casting would make a runtime type check explicit, but would not guarantee type safety. Variables ul and lr would still be typed as Point and not ColorPoint.

Problem 3 a)

Is there an alternative to multiple inheritance if we want to *reuse only implementation, and do not care about subtyping?* Use the Stack and Queue, and try to implement Dequeue.



```
class Dequeue {
  Stack s = new Stack();
  Queue q = new Queue();
  Object[] r;
  Dequeue() { s.r = r; q.r = r; }

  void insert_front(Object o) { s.push(o); }
  void insert_rear(Object o) { q.insert(o); }
  void delete_front() { s.pop(); }
  void delete_rear() { q.delete(); }
}
```

Shared between Queue and Stack

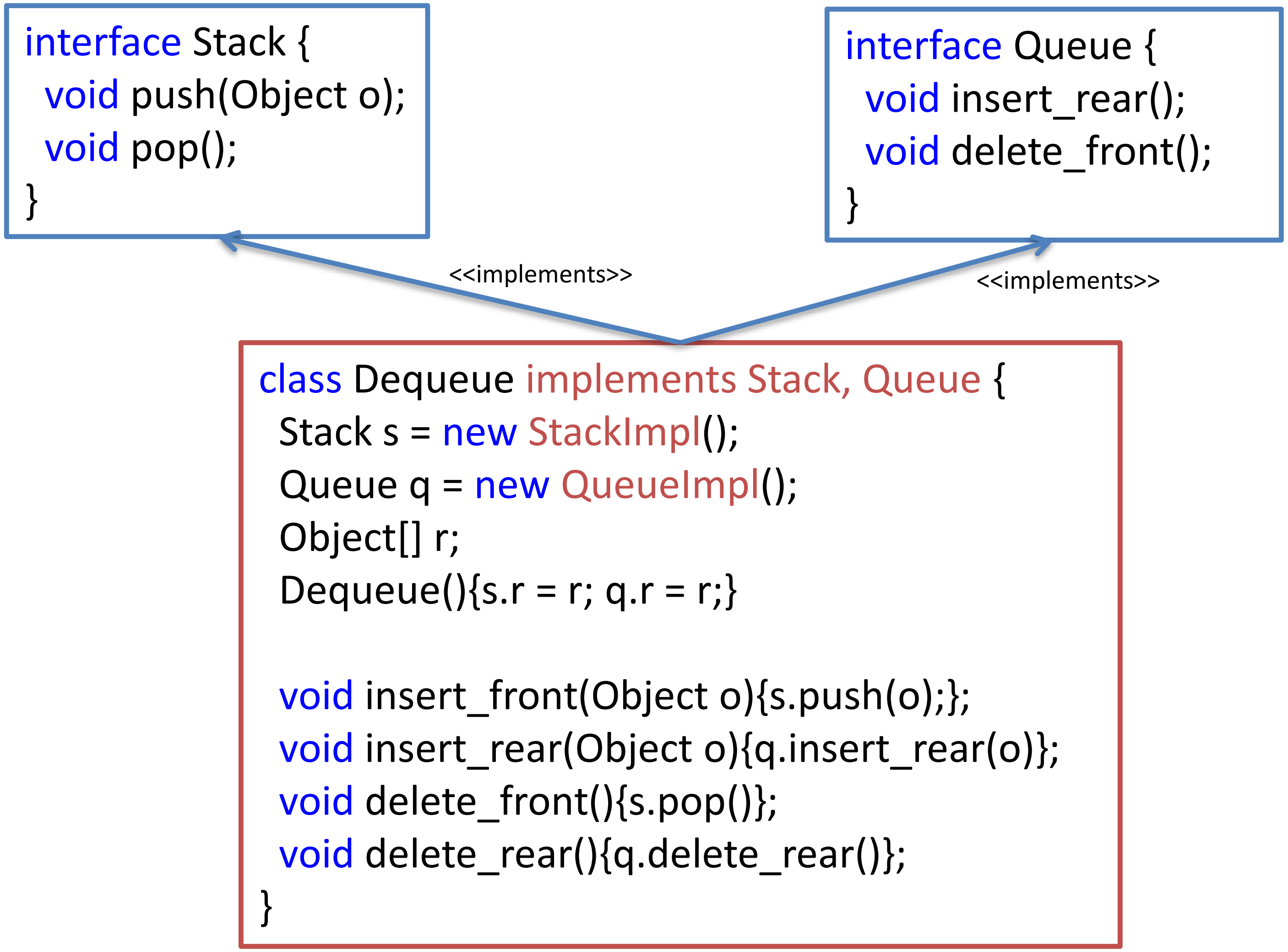
Problem 3 a2)

Define Stack and Queue by means of an implementation by a Dequeue object d, and use this in the implementation of methods.

```
class Stack {
  Dequeue d = new Dequeue();
  void push(Object o){ d.insert_front(o); }
  void pop(){ d.delete_front(); }
}

class Queue {
  Dequeue d = new Dequeue();
  void insert_rear(){ d.insert_rear(o); }
  void delete_front(){ d.delete_front(); }
}
```

Problem 3 b)

What if we want to have a subtyping relationship, so that objects typed as Stack and Queue can hold objects of type Dequeue?



```
interface Stack {
  void push(Object o);
  void pop();
}
```

```
interface Queue {
  void insert_rear();
  void delete_front();
}
```

```
class Dequeue implements Stack, Queue {
  Stack s = new StackImpl();
  Queue q = new QueueImpl();
  Object[] r;
  Dequeue(){s.r = r; q.r = r;}

  void insert_front(Object o){s.push(o);};
  void insert_rear(Object o){q.insert_rear(o)};
  void delete_front(){s.pop()};
  void delete_rear(){q.delete_rear()};
}
```

Problem 4

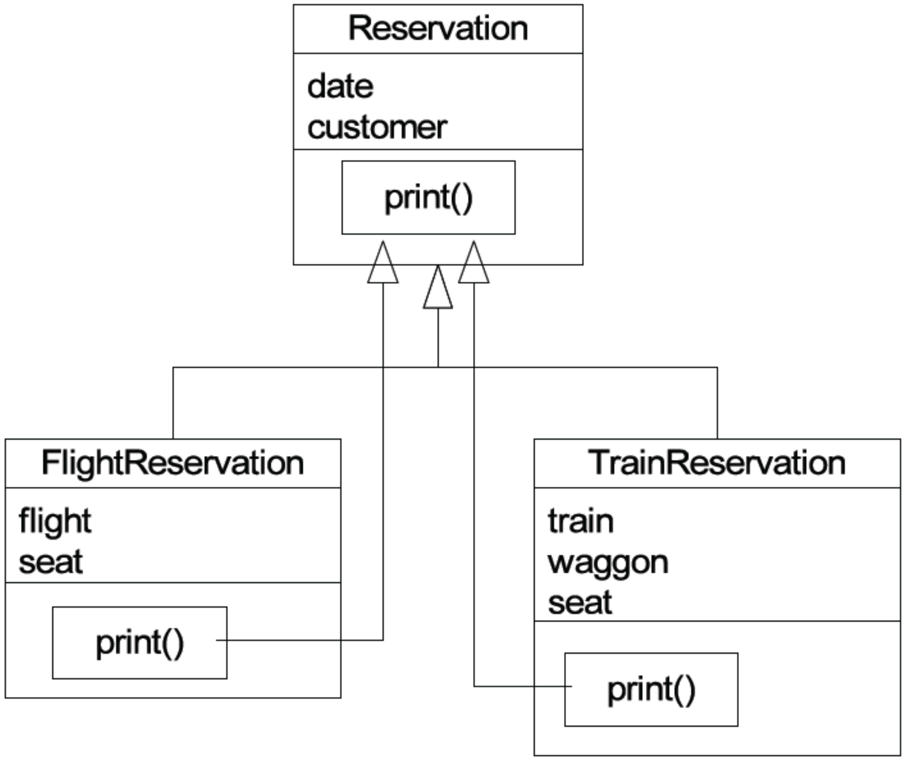
With this program, describe what happens on lines marked 1, 2 and 3

```
class TypeTest {
  C v = new C();
  void arrayProb(C[] anArray) {
    if (anArray.length >0)
      anArray[0] = v;                              // (2)   Runtime type error
  }

  static void main(string[] args) {
    TypeTest tt = new TypeTest();
    CSub[] paramArray = new CSub[10];
    tt.arrayProb(paramArray);                    // (1)     Ok!
    paramArray[0].methodOfCSubOnly();  // (3)     Will never be
  }                                                         reached!
}
```

Problem 5

Suppose that we have class Reservation with subclasses FlightReservation and TrainReservation as described in the foil set.

It is desirable to have specific collections of reservations that cater for subclasses for new kinds of reservations (e.g. for space travels). How would you make a print method that prints all elements of such a collection, using the generics mechanisms of Java?



```
void print(Collection<? extends Reservation > reservations) { ... }
```

Problem 6

Consider the following Java sketch:

```
interface cowboy {void draw(); ...}
interface shape {void draw(); ...}
class LuckyLuke implements cowboy, shape {...}
```

Is this an example of structural (sub)typing, given the fact that Java may very well get the same method from different interfaces, but still only provide one implementation?

No, this is not an example of structural subtyping. Type checking is still done using the *names* cowboy and shape.

Problem 7

The FlightReservation class we have seen a couple of times has a Flight attribute. We assume that this is a reference to an object of class Flight. The Flight object represents the actual flight reserved. In the flight table of SAS we have entries for e.g. SK451 (Oslo to Copenhagen). Suppose that we would like to represent such an entry by means of a FlightType object. Class FlightType would therefore have attributes that are common to all SK451 Flights, like source, destination, scheduled departure time (8.20), scheduled flying time (1.10), scheduled arrival time, etc.

SK451 takes place every day (or almost), so a reservation system would need to have one Flight object for each actual flight. These Flight objects will have a representation of seats (free, occupied), and for other reasons one may imagine that they will also have actual departure time, actual flight time and delay (departure and arrival delay).

It is perfectly possible to do this without inner classes, but if you should exploit inner classes, how would this be done. Of special interest are of course the functions computing the departure and arrival delays.

Feel free to be inspired by the slide on inner classes exemplified by class Apartment, specially the fact the attribute height of Apartment is visible in the inner classes. Attributes like scheduled departureTime and arrivalTime should be attributes of the outer class FlightType.

Problem 7 – slide from lecture

```
class Apartment {
   Height height;
   Kitchen theKitchen = new Kitchen();
   class ApartmentBathroom extends Bathroom {... height ...}
   ApartmentBathroom Bathroom_1 = new ApartmentBathroom ();
   ApartmentBathroom Bathroom_2 = new ApartmentBathroom ();
   Bedroom theBedroom = new Bedroom ();
   FamilyRoom theFamilyRoom = new FamilyRoom ();
   . . .
   Person Owner;
   Address theAddress = new Address()
}
```

Problem 7

```
class FlightType {
  City source, destination;
  TimeOfDay departureTime, arrivalTime;
  Duration flyingTime;

  class Flight {
    Seat[] seats;
    TimeOfDay actualDepartureTime, actualArrivalTime;
    Duration ActualFlyingTime;

    Duration departureDelay(){
      return actualDepartureTime – departureTime;
    };
    Duration arrivalDelay(){
      return actualArrivalTime – arrivalTime;
    };  } };

class TimeTable {
  FlightType SK451 = new FlightType(Oslo, Copenhagen, 8.20);  // provided a corresponding constructor
};

TimeTableWithReservations ttwr = new TimeTable {
  SK451.Flight[365] SK451flights; // note: an array of an inner class Flight in an object SK451 of class
FlightType
...
}
```

INF3110 Group 2

Exam 2013 solutions and hints

But first, an example of compile-time and run-time type checking

> *Static type-checking is the process of verifying the type safety of a program based on analysis of a program's source code.*

> *Dynamic type-checking is the process of verifying the type safety of a program at runtime*

So, static is compile-time and dynamic is run-time. In this example (Java) will try to figure out the types of all «elements», in an effort to prevent bad things from happening.

Imagine we have the following code.
What would be returned by «new B().me()»?
An A object? A B object?

```
class A {
        A me() {
                return this;
        }

        public void doA() {
                System.out.println("Do A");
        }
}

class B extends A {
        public void doB() {
                System.out.println("Do B");
        }
}
```

The compiler only «sees» an A object being returned, but the object is actually a B object.

Are there any potential problems with this?

Since the compiler thinks it’s an A object, and B extends A, inheriting methods, this line is legal: new B().me().doA();

But it’s actually a B object, so:
new B().me().doB(); should be legal too?

But it’s not. The compiler can’t figure out that this should be legal. So we have to cast the object to a B object:
((B) new B().me().doB();

The compiler then «trusts» the programmer and lets it pass without it being able to verify that is actually is ok.

Full example can be found at http://www.programcreek.com/2011/12/an-example-of-java-static-type-checking/

1a

The code below is a program in this language. It is not intended to be complete, and it includes only things that are required in order to answer this question.

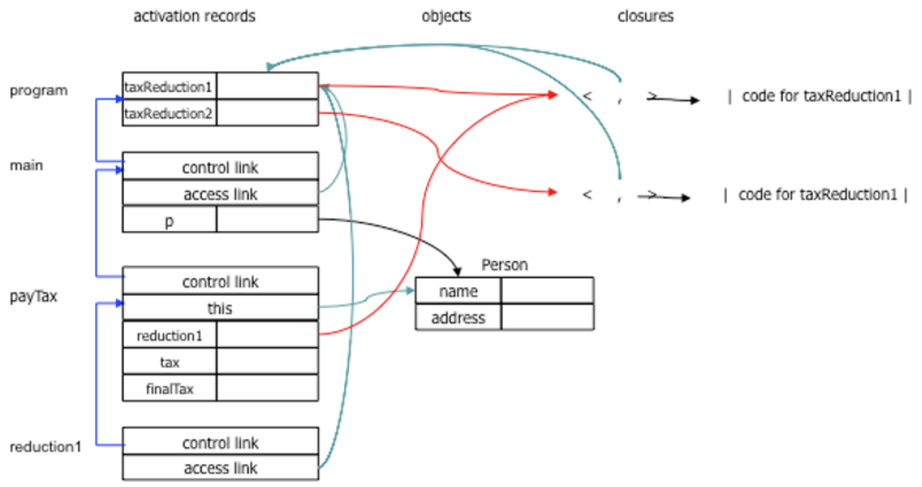
```
{
  int taxReduction1(String address) {...}
  int taxReduction2(String address) {...}

  interface Addressable {
    String address();
    void printLabel();
  }
  interface Taxable {
    int income()
    void payTax(int reduction(String));
  }
  class Person implements Addressable, Taxable {
    String name;
    String address;
    String name(){return name;}
    String address(){return address;}

    void printLabel(){...}

    void payTax(int reduction(String)){
      int tax, finalTax;
      ... // compute tax
      finalTax = tax - reduction(address());
      ... // pay finalTax
    }
    public void Person(String pname) {name=pname;}
  }

  void main() {
    Person p = new Person("Birger");
    // setting the address and income of p
    p.printLabel();
    ...
    p.payTax(taxReduction1);
    ...
  }
}
```

Draw the run-time stack and the `Person` object as they are while executing the call `reduction(address())` in `payTax`. Include all links between the activation records, the variables and parameters in the activation records, and illustrate/explain how both the formal and actual parameter to `payTax` are handled. You may assume that the static link for a method activation record is the object that has the method.

So we're to draw the run-time stack while executing reduction(address())

Yes, we did this exact exercise a couple of weeks ago

So, how do we start?



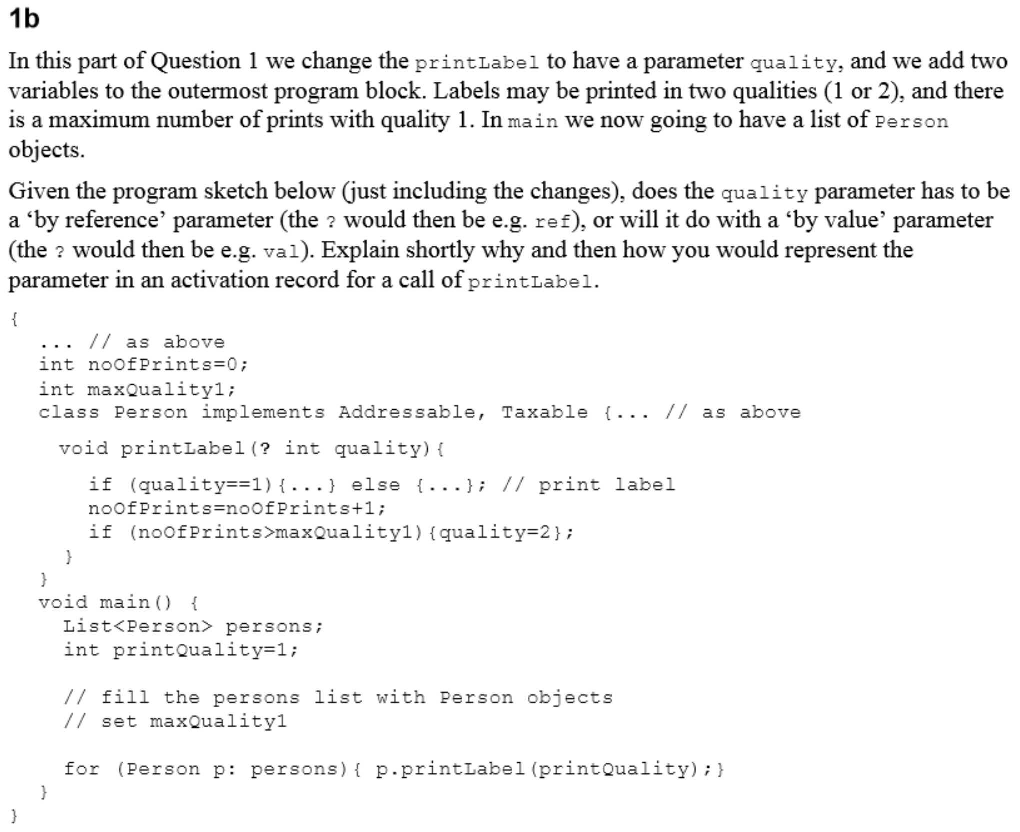
1b

In this part of Question 1 we change the `printLabel` to have a parameter `quality`, and we add two variables to the outermost program block. Labels may be printed in two qualities (1 or 2), and there is a maximum number of prints with quality 1. In `main` we now going to have a list of `Person` objects.

Given the program sketch below (just including the changes), does the `quality` parameter has to be a 'by reference' parameter (the `?` would then be e.g. `ref`), or will it do with a 'by value' parameter (the `?` would then be e.g. `val`). Explain shortly why and then how you would represent the parameter in an activation record for a call of `printLabel`.

```
{
  ... // as above
  int noOfPrints=0;
  int maxQuality1;
  class Person implements Addressable, Taxable {... // as above

    void printLabel(? int quality){

      if (quality==1){...} else {...}; // print label
      noOfPrints=noOfPrints+1;
      if (noOfPrints>maxQuality1){quality=2};
    }
  }
  void main() {
    List<Person> persons;
    int printQuality=1;

    // fill the persons list with Person objects
    // set maxQuality1

    for (Person p: persons){ p.printLabel(printQuality);}
  }
}
```

So, we want to figure out if quality should be called with value or by reference



1b

In this part of Question 1 we change the `printLabel` to have a parameter `quality`, and we add two variables to the outermost program block. Labels may be printed in two qualities (1 or 2), and there is a maximum number of prints with quality 1. In `main` we now going to have a list of `Person` objects.

Given the program sketch below (just including the changes), does the `quality` parameter has to be a 'by reference' parameter (the `?` would then be e.g. `ref`), or will it do with a 'by value' parameter (the `?` would then be e.g. `val`). Explain shortly why and then how you would represent the parameter in an activation record for a call of `printLabel`.

```
{
  ... // as above
  int noOfPrints=0;
  int maxQuality1;
  class Person implements Addressable, Taxable {... // as above

    void printLabel(? int quality){

      if (quality==1){...} else {...}; // print label
      noOfPrints=noOfPrints+1;
      if (noOfPrints>maxQuality1){quality=2};
    }
  }
  void main() {
    List<Person> persons;
    int printQuality=1;

    // fill the persons list with Person objects
    // set maxQuality1

    for (Person p: persons){ p.printLabel(printQuality);}
  }
}
```

So, we want to figure out
if quality should be called
with value or by reference

What would the different effects be?

So in conclusion?

By reference so the printLabel
method has access to change
the value in printQuality

1c

In this part we add two methods to class `Person`, and define `VisitingPerson` as a subclass of `Person` in the same scope as `Person`. In this language methods are by default virtual, and redefinitions must have the same signature (no variance) as the virtual method in the superclass. The method `sameAddress` is thus redefined in `VisitingPerson`:

```
class Person implements Addressable, Taxable {
  // as above
  boolean sameAddress(Person p) {
    return ((address() = p.address());
  }
  boolean samePerson(Person p) {
    return (name = p.name()& address() = p.address());
  }
}

class VisitingPerson extends Person {
  String visitingAddress;
  String visitingAddress(){return visitingAddress;}

  boolean sameAddress(Person p) {
    return ((address() = p.address())
        | (visitingAddress() = p.visitingAddress()));
  }

}
```

A person does not define a «visitingAddress» method

1. What kind of compile-time type error will this code give, and where?

2. How would you change the program to avoid this type error?

Can avoid the type error by casting p to VisitingPerson

But is this actually safe? Why or why not?

1d

In this part we try to turn the interfaces into classes in order to be able to specify the behaviour of `printLabel`. Class `Person` is now defined to have variables with types `Addressable` and `Taxable` instead of implementing the corresponding interfaces. We just consider the details of `Addressable`, and for this part of Question 1 we drop the parameter to `printLabel`. We assume that the `addr` variable of `Person` is visible, so that the `main` method can now directly call `printLabel` of a `Person p` by the call '`p.addr.printLabel()`'. The address of a `Person p` will similarly be accessible by `p.addr.address()`.

```
{
  // as above

  class Addressable {
    String address;
    String address(){return address;}
    void printLabel(){System.out.println(address());}
  }
  class Taxable {
    void payTax(int reduction(String)){...}
  }
  class Person {
    String name;
    String address;
    String name(){return name;}
    String address(){return address;}

    public Addressable addr = new Addressable();
    public Taxable tax = new Taxable();

    public void Person(String pname, paddress) {
      name=pname; addr.address=paddress;
    }
  }
  void main() {
    Person p = new Person("Birger", "Røahagan 33A");
    p.addr.printLabel();
    ...
    p.tax.payTax(taxReduction1);
    ...
  }
}
```

The problem with this solution is that `printLabel` will only print the `address` of a `Person`, while we would like it to print both `name` and `address` of a `Person`. It is not possible to define `printLabel` within class `Addressable` so that it prints the `name`, as `name` is not visible from class `Addressable`.

Sketch a solution, given that `printLabel` still has to be defined in `Addressable` (as above), and that `main` has the call `p.addr.printLabel()`. The solution must print the `name`. You may assume that a method `m` that is redefined in a subclass may call the `m` of the superclass by `super.m()`, and that the language supports anonymous classes.

So, we want to print the name
as well when calling upon printLabel.
At the same time we wish to preserve
the call, so it should still be p.addr.printLabel();

The given hints here are what you may assume.
So we have calls to superclasses using super.x()
and we have support for anonymous classes.

The solution given is this, but someone more awake
than me pointed me to a problem with this solution

```
class Person {
  String name;
  String name(){return name;}

  Addressable addr = new Addressable(){
    System.out.println(name());super.printLabel();
  }
```

So this would probably make more sense

```
class Person {
  String name;
  String name() { return name; }

  Addressable addr = new Addressable() {
    void printLabel() {
      System.out.println(name());
      super.printLabel();
    }
  }
}
```

If you get something similar and
you are complety stuck, work
around the problem. It's better
to solve the problem is a «bad»
way than do nothing. Any ideas
on how to work around this problem?