University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

SEMESTER I EXAMINATION – 2012/2013

COMP 20010

Data Structures & Algorithms I

Prof. A. Mille

Mr. J. Dunnion

Dr Eleni Mangina

Time allowed: 2 hours

Instructions for candidates

Answer **any** two questions. All questions carry equal marks (50).

Use of calculators is prohibited.

Instructions for invigilators

Use of calculators is prohibited.

1. Answer parts (a) to (d). **50 marks in total**

(a) (15 marks)

(i) Briefly define the following variable modifiers in terms of scope (or visibility) of instance variables: public, protected, private, final. **(4 marks)**

Solution: (1 Mark each)
Public: Anyone can access public instance variables
Protected: Only methods of the same package or of its subclasses can access protected instance variables
Private: Only methods of the same class (not methods of the subclass) can access private instance variables.
Final: A final instance variable is one that must be assigned an initial value, and then can never be assigned a new value after that.

(ii) What is the output of the following program? Why? **(5 marks)**

```
public class AverageCalculator
{
                public static void main(String[] args)
                 {
                   int age1 = 18;
                    int age2 = 35;
                    int age3 = 50;
                    int age4 = 44;
                    double averageAge = (age1 + age2 + age3 + age4) / 4;
                    System.out.println(averageAge);
                  }
}
```

Answer:

36.0

That is not the real average (it should be `36.75`). The incorrect result is obtained because integer division is being used. Floating-point division should be used instead.

(iii) An int variable is four bytes in length. In the wrapper class Integer, there is a constant called Integer.SIZE which represents the number of bits the variable contains. The program below prints this constant. Modify the program so that it prints the sizes of each of the primitive numeric types. Try to include all eight primitive types. One class does not have a SIZE constant. Which class is it? **(6 marks)**

```
public class PrimitiveSizes
{
               public static void main(String[] args)
               {
                System.out.println("An int variable is " + Integer.SIZE + " bits.");
                }
}
```

Answer:

```
public class PrimitiveSizes
{   public static void main(String[] args)
  { System.out.println("An int variable is " + Integer.SIZE + " bits.");
    System.out.println("A byte variable is " + Byte.SIZE + " bits.");
    System.out.println("A short variable is " + Short.SIZE + " bits.");
    System.out.println("A long variable is " + Long.SIZE + " bits.");
    System.out.println("A double variable is " + Double.SIZE + " bits.");
    System.out.println("A float variable is " + Float.SIZE + " bits.");
    System.out.println("A char variable is " + Character.SIZE + " bits.");
  }
}
```

The boolean does not have a SIZE constant.

(b) (15 marks)

(i) Write a program that reads in the names and scores of students and then computes and displays the names of the students with the highest and lowest scores. **(5 marks)**

A simple method of carrying out this task would be to have two parallel arrays:

```
String[] names;
int[] scores;
```

However, you should avoid parallel arrays in your solution.
First, write a class `Student` to solve the parallel array problem. A student should have a name and a score.

Answer:

```
public class Student
{
  private String name;
  private int score;
  public Student(String n, int s)
  {
    name = n;
    score = s;
  }
  public String getName()
  {
    return name;
  }
  public int getScore()
  {
    return score;
  }
}
```

(ii) Listed below is a bad implementation of the `StudentScores` class that uses two parallel arrays. Modify it to eliminate the use of parallel arrays. Use an array list of students in your solution. Your improved `StudentScores` class should use the `Student` class that you created previously.**(10 marks)**

```
public class BadStudentScores
{
  private final int MAX_STUDENTS = 100;
  private String[] names;
  private int[] scores;
  private int numStudents;

  public BadStudentScores()
  {
    scores = new int[MAX_STUDENTS];
    names = new String[MAX_STUDENTS];
    numStudents = 0;
  }

  public void add(String name, int score)
  {
    if (numStudents >= MAX_STUDENTS)
      return; // Not enough space to add new student score
    names[numStudents] = name;
    scores[numStudents] = score;
    numStudents++;
  }

  public String getHighest()
  {
    if (numStudents == 0)
      return null;
    int highest = 0;
    for (int i = 1; i < numStudents; i++)
      if (scores[i] > scores[highest])
        highest = i;
    return names[highest];
  }

  public String getLowest()
  {
    if (numStudents == 0)
      return null;
    int lowest = 0;
    for (int i = 1; i < numStudents; i++)
      if (scores[i] < scores[lowest])
        lowest = i;
    return names[lowest];
  }
}
```

Answer:

```
import java.util.ArrayList;

public class StudentScores
{
   private ArrayList<Student> scores;
   public StudentScores()
   {
      scores = new ArrayList<Student>();
   }
   public void add(String name, int score)
   {
      scores.add(new Student(name, score));
   }

   public Student getHighest()
   {
      if (scores.size() == 0)
         return null;
      Student highest = scores.get(0);
      for (int i = 1; i < scores.size(); i++)
         if (scores.get(i).getScore() > highest.getScore())
            highest = scores.get(i);
      return highest;
   }

   public Student getLowest()
   {
      if (scores.size() == 0)
         return null;
      Student lowest = scores.get(0);
      for (int i = 1; i < scores.size(); i++)
         if (scores.get(i).getScore() < lowest.getScore())
            lowest = scores.get(i);
      return lowest;
   }
}
```

(c) We often need to convert a value from a `String` to a primitive data type. For example, the program below converts the `String` `"12345"` to an `int` using the static method `parseInt` that is contained in the wrapper class `Integer`. Complete the program below by filling in the code specified in the embedded comments. **(10 marks)**

```
public class StringConversion
{
  public static void main(String[] args)
   {
      String value1 = "12345";
      int intValue = Integer.parseInt(value1);
      System.out.println("intValue = " + intValue);

      String value2 = "12.345";
      // Convert value2 to a double here
      // Print the converted value

      String value3 = "87654";
      // Convert value3 to a long here
      // Print the converted value

      String value4 = "321";
      // Convert value4 to a short here
      // Print the converted value

      String value5 = "-28";
      // Convert value5 to a byte here
      // Print the converted value

      String value6 = "6";
     // Convert value6 to a char here.  (Hint: See Advanced Topic 3.5)
      // Print the converted value

      String value7 = "true";
      // Convert value7 to a boolean here.  (Hint: Check the API for the
      // Boolean wrapper class.  Which method returns a Boolean?)
      // Print the converted value

      String value8 = "-45.237";
      // Convert value7 to a float here
      // Print the converted value
   }
}
```

Ans:

```
public class StringConversion
{
  public static void main(String[] args)
  {
    String value1 = "12345";
    int intValue = Integer.parseInt(value1);
    System.out.println("intValue = " + intValue);
```

```
String value2 = "12.345";
// Convert value2 to a double here
double doubleValue = Double.parseDouble(value2);
// Print the converted value
System.out.println("doubleValue = " + doubleValue);

String value3 = "87654";
// Convert value3 to a long here
long longValue = Long.parseLong(value3);
// Print the converted value
System.out.println("longValue = " + longValue);

String value4 = "321";
// Convert value4 to a short here
short shortValue = Short.parseShort(value4);
// Print the converted value
System.out.println("shortValue = " + shortValue);

String value5 = "-28";
// Convert value5 to a byte here
byte byteValue = Byte.parseByte(value5);
// Print the converted value
System.out.println("byteValue = " + byteValue);

String value6 = "6";
// Convert value6 to a char here.
char charValue = value6.charAt(0);
// Print the converted value
System.out.println("charValue = " + charValue);

String value7 = "true";
// Convert value7 to a boolean here.
Boolean booleanValue = Boolean.valueOf(value7);
// Print the converted value
System.out.println("booleanValue = " + booleanValue);

String value8 = "-45.237";
// Convert value7 to a float here
float floatValue = Float.parseFloat(value8);
// Print the converted value
```

```
    System.out.println("floatValue = " + floatValue);
  }
}
```

(d) (10 marks)

(i) Which values of `year` cause the following loop to terminate with a correct answer? **(5 marks)**

```
/**
  Counts the number of years from a year input by the user
  until the year 3000.
*/
public class CountYears
{
  public static void main(String[] args)
  {
    int millennium = 3000;
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter the current year: ");

    int year = in.nextInt();
    int nyear = year;

    while (nyear != millennium)
    {
      nyear++;
    }

    System.out.println("Another " + (nyear - year) + " years to the millennium.");
  }
}
```

Answer:
Any value less than or equal to 3000 (even negative numbers). For a value x, greater than 3000, the loop iterates through all positive `int` values greater than x before overflowing the size of an `int` variable and becoming negative. It then iterates through all negative int values, zero, and then all positive `int` values less than 3000. In this case the computed answer is incorrect.

(ii) Convert this `while` loop to a `for` loop **(5 marks)**

```
/**
 Program to compute the first integral power to which 2 can be
raised that is greater than that multiple of a given integer.
*/
public class CountPowerOf2
{
  public static void main(String[] args)
  {
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter a number, 0 to quit: ");
    int n = in.nextInt();
    int i = 1;
            while (n * n > Math.pow(2, i))
```

```
                    {
                            i++;
                     }
                System.out.println("2 raised to " + i
                + " is the first power of two greater than " + n + " squared");
                    }
            }
```

Answer:

```
/**
   Program to compute the first integral power to which 2 can be
   raised that is greater than that multiple of a given integer.
*/
public class CountPowerOf2
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Please enter a number, 0 to quit: ");
      int n = in.nextInt();

      int i;
      for (i = 1; n * n > Math.pow(2, i); i++)
         ;

      System.out.println("2 raised to " + i
            + " is the first power of two greater than " + n + " squared");
   }
   }
}
```

2. Answer parts (a) to (d). 50 marks in total

(a) (5 marks)

Consider the following problem: A company allows its employees to check out certain items, such as handheld computers and music players, to gain personal experience with them. Popular items can be reserved on a first come/first served basis. A reservation list is kept for each item. There is a fine for overdue items.

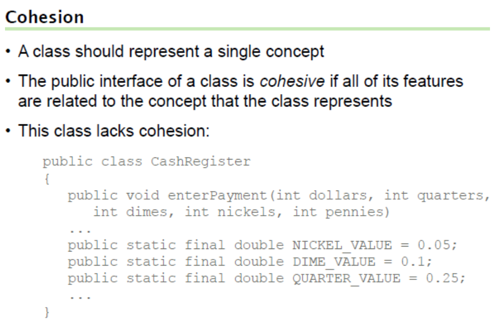
Your development team's task is to write a software program that allows the stockroom clerk to check out items and check them back in, reserve items, notify

employees when a reserved item has been returned, produce reports of overdue items, and track payment of fines.
Based on the rule of thumb for finding classes, what classes would you choose to implement this program?

Answer:
Item: describes an item that can be reserved
Employee: describes an employee
ReservationSystem: handles reservations and waiting lists, produces reports, keeps a list of fines to be able to track them, allows items to be checked out and checked in, notifies employees when a reserved item has been returned
Reservation: describes a current reservation (item, employee that holds it, date)
WaitingList: keeps an ordered list of employees waiting for an item
Fine: describes an overdue fine (employee, date)

(b) (15 marks)
Briefly define the following terms that are commonly used while designing classes and provide a simple example for each term: Coupling between classes, Immutable Classes, Side effect of a method, Static method, Static Variable

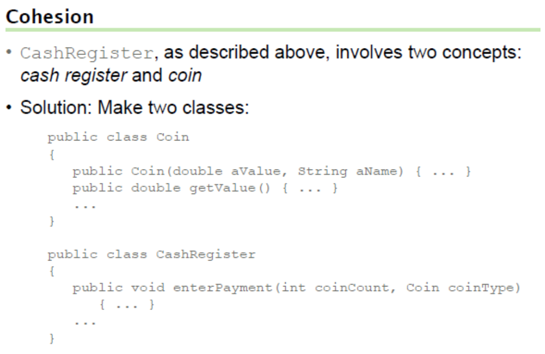
Solution:
(3 Marks Coupling)



Cohesion

- A class should represent a single concept
- The public interface of a class is *cohesive* if all of its features are related to the concept that the class represents
- This class lacks cohesion:

```
public class CashRegister
{
   public void enterPayment(int dollars, int quarters,
      int dimes, int nickels, int pennies)
   ...
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   ...
}
```



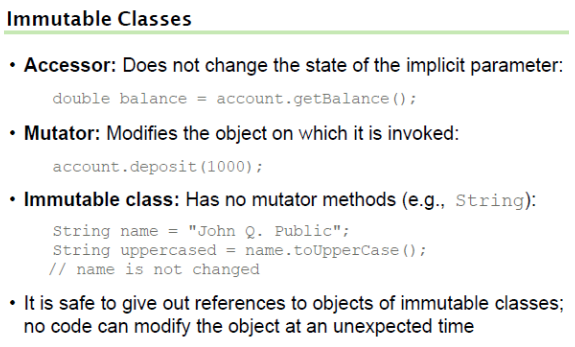
Cohesion

- `CashRegister`, as described above, involves two concepts: *cash register* and *coin*
- Solution: Make two classes:

```
public class Coin
{
   public Coin(double aValue, String aName) { ... }
   public double getValue() { ... }
   ...
}

public class CashRegister
{
   public void enterPayment(int coinCount, Coin coinType)
      { ... }
   ...
}
```

(2 Marks Immutable)



Immutable Classes

- **Accessor:** Does not change the state of the implicit parameter:

```
double balance = account.getBalance();
```

- **Mutator:** Modifies the object on which it is invoked:

```
account.deposit(1000);
```

- **Immutable class:** Has no mutator methods (e.g., `String`):

```
String name = "John Q. Public";
String uppercased = name.toUpperCase();
// name is not changed
```

- It is safe to give out references to objects of immutable classes; no code can modify the object at an unexpected time

Dependency

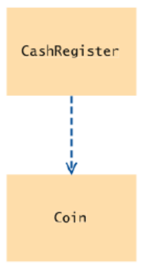
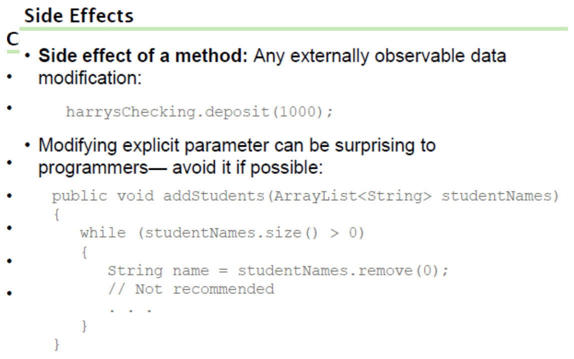


Figure 1
Dependency Relationship Between the CashRegister and Coin Classes

(3 Marks Side Effects)



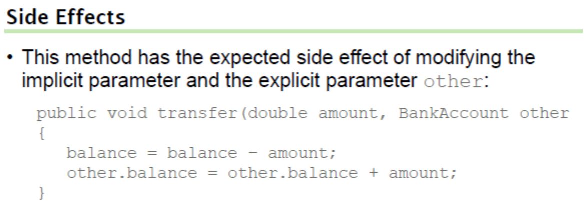
Side Effects

- **Side effect of a method:** Any externally observable data modification:

```
harrysChecking.deposit(1000);
```

- Modifying explicit parameter can be surprising to programmers— avoid it if possible:

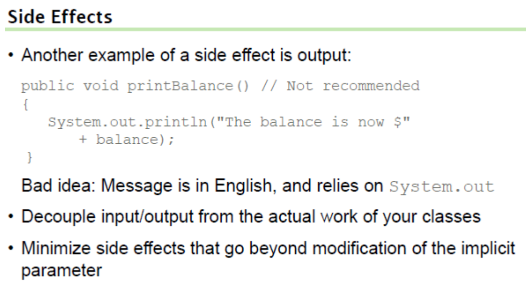
```
public void addStudents(ArrayList<String> studentNames)
{
   while (studentNames.size() > 0)
   {
      String name = studentNames.remove(0);
      // Not recommended
      . . .
   }
}
```



Side Effects

- This method has the expected side effect of modifying the implicit parameter and the explicit parameter `other`:

```
public void transfer(double amount, BankAccount other
{
   balance = balance - amount;
   other.balance = other.balance + amount;
}
```



Side Effects

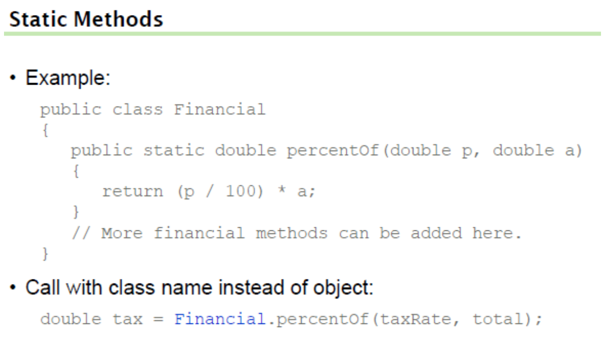
- Another example of a side effect is output:

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $"
      + balance);
}
```

Bad idea: Message is in English, and relies on `System.out`

- Decouple input/output from the actual work of your classes
- Minimize side effects that go beyond modification of the implicit parameter

(4 Marks: Static Methods)



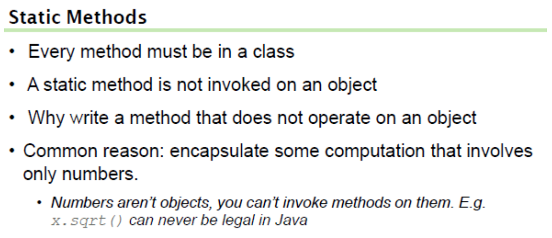
Static Methods

- Example:

```
public class Financial
{
   public static double percentOf(double p, double a)
   {
      return (p / 100) * a;
   }
   // More financial methods can be added here.
}
```

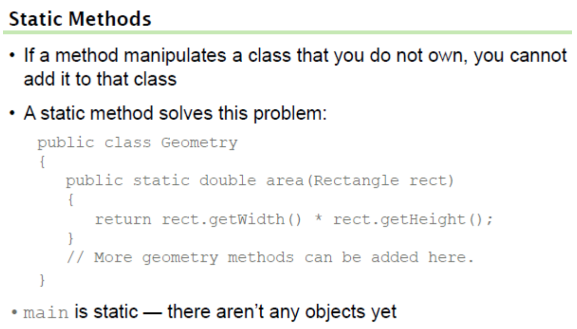
- Call with class name instead of object:

```
double tax = Financial.percentOf(taxRate, total);
```



Static Methods

- Every method must be in a class
- A static method is not invoked on an object
- Why write a method that does not operate on an object
- Common reason: encapsulate some computation that involves only numbers.
  - *Numbers aren't objects, you can't invoke methods on them. E.g. `x.sqrt()` can never be legal in Java*



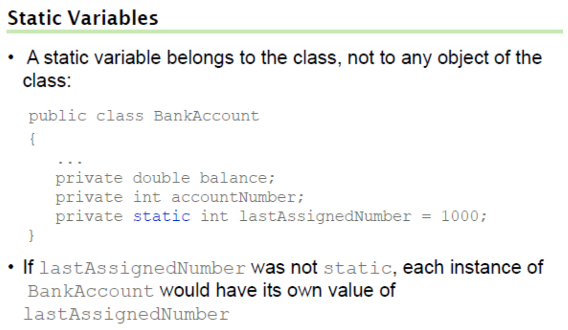
Static Methods

- If a method manipulates a class that you do not own, you cannot add it to that class
- A static method solves this problem:

```
public class Geometry
{
   public static double area(Rectangle rect)
   {
      return rect.getWidth() * rect.getHeight();
   }
   // More geometry methods can be added here.
}
```

- `main` is static — there aren't any objects yet

(3 Marks: Static Variables)



Static Variables

- A static variable belongs to the class, not to any object of the class:

```
public class BankAccount
{
   . . .
   private double balance;
   private int accountNumber;
   private static int lastAssignedNumber = 1000;
}
```

- If `lastAssignedNumber` was not `static`, each instance of `BankAccount` would have its own value of `lastAssignedNumber`

A Static Variable and Instance Variables

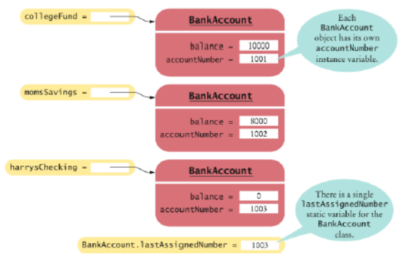

collegeFund =
BankAccount
balance = 10000
accountNumber = 1001
Each BankAccount object has its own accountNumber instance variable.
momsSavings =
BankAccount
balance = 8000
accountNumber = 1002
harrysChecking =
BankAccount
balance = 0
accountNumber = 1003
There is a single lastAssignedNumber static variable for the BankAccount class.
BankAccount.lastAssignedNumber = 1003

Figure 4
A Static Variable and Instance Variables

(c) (10 marks)

The following class has a method with a **side effect**:

```
/** A purse computes the total value of a collection of coins.*/
public class Purse
{           private double total;
             /**
            Constructs an empty purse.
            */
            public Purse()
            {
                        total = 0;
            }
             /**
            Adds a coin to the purse.
            @param aCoin the coin to add
            */
            public void add(Coin aCoin)
            {
                        total = total + aCoin.getValue();
                        System.out.println("The total is now " + total);
            }
             /**
            Gets the total value of the coins in the purse.
            @return the sum of all coin values
              */
              public double getTotal()
              {
                          return total;
              }
}
```

(i) Describe the side effect and explain why it is not desirable.

Answer:
The `add` method has a side effect: it prints the total (using `System.out.println`). It is not desirable because it makes two strong assumptions:
1) The message is in English—you assume that the user of your software reads English
2) You rely on `System.out`. A method that relies on `System.out` won't work in an embedded system, such as the computer inside an automatic teller machine.

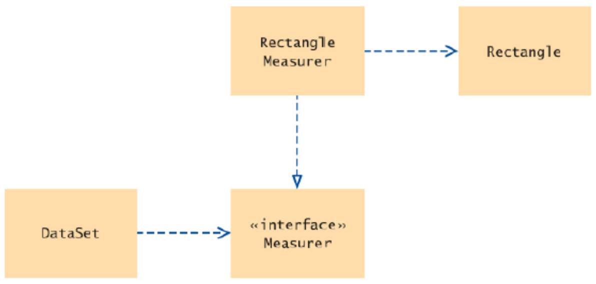
(ii) How would you eliminate the side effect in the `add` method?

Answer:
We could provide a method `printTotal` that prints the total to the screen. Or, even better, we could provide a method `toString` that returns a string representation of the

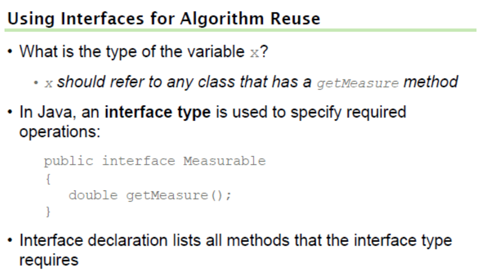
class (i.e. its total value in a string), so that the calling object can show it to the user in the most convenient way.

(d) (20 marks)

(i) Define the interface types in Java and denote the differences with classes.
(ii) Provide the general syntax of implementing an interface in Java and a simple paradigm.
(iii) Define the terms polymorphism and inner class in Java.
(iv) Given the following UML diagram comment on the types of classes and their relationships.



Answer:
((i) 5 Marks)

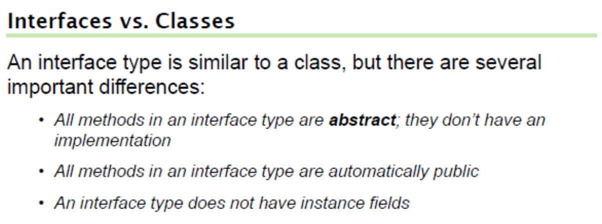


Using Interfaces for Algorithm Reuse

- What is the type of the variable `x`?
  - *`x` should refer to any class that has a `getMeasure` method*
- In Java, an **interface type** is used to specify required operations:

```
public interface Measurable
{
   double getMeasure();
}
```

- Interface declaration lists all methods that the interface type requires

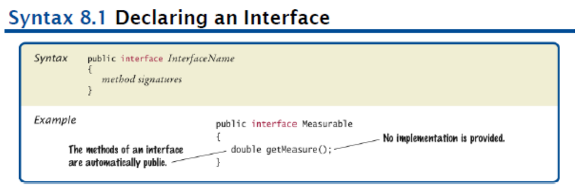


Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- *All methods in an interface type are **abstract**; they don't have an implementation*
- *All methods in an interface type are automatically public*
- *An interface type does not have instance fields*

((ii) 5 Marks)



Syntax 8.1 Declaring an Interface

Syntax

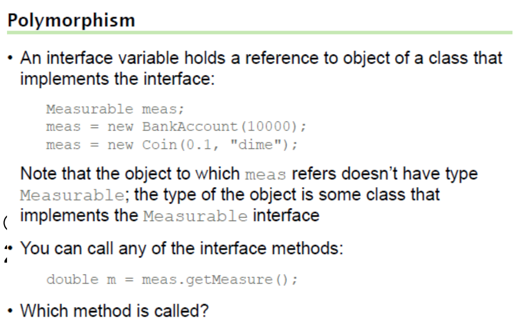
```
public interface InterfaceName
{
   method signatures
}
```

Example

```
public interface Measurable
{
   double getMeasure();
}
```

The methods of an interface are automatically public.
No implementation is provided.

((iii) 5 Marks)



Polymorphism

- An interface variable holds a reference to object of a class that implements the interface:

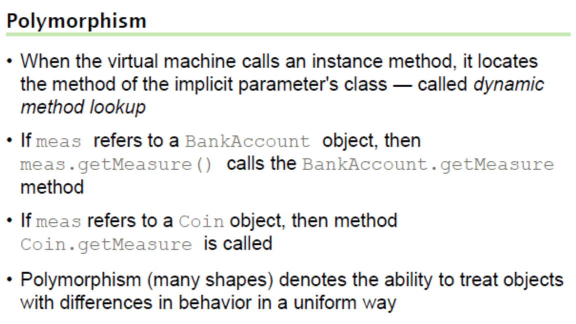
```
Measurable meas;
meas = new BankAccount(10000);
meas = new Coin(0.1, "dime");
```

  Note that the object to which `meas` refers doesn't have type `Measurable`; the type of the object is some class that implements the `Measurable` interface

- You can call any of the interface methods:

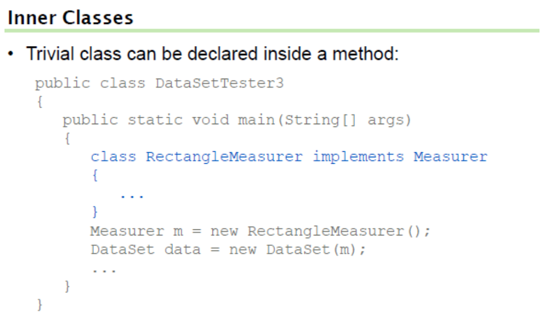
```
double m = meas.getMeasure();
```

- Which method is called?



Polymorphism

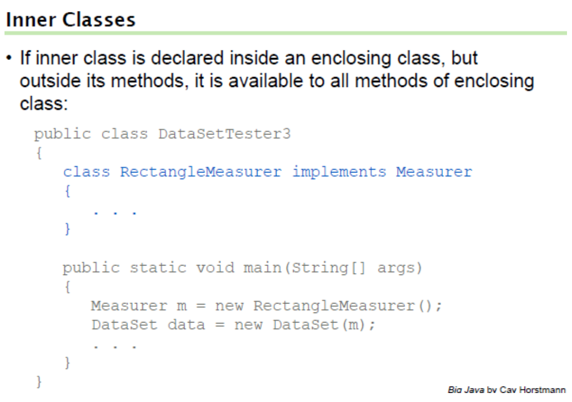
- When the virtual machine calls an instance method, it locates the method of the implicit parameter's class — called *dynamic method lookup*
- If `meas` refers to a `BankAccount` object, then `meas.getMeasure()` calls the `BankAccount.getMeasure` method
- If `meas` refers to a `Coin` object, then method `Coin.getMeasure` is called
- Polymorphism (many shapes) denotes the ability to treat objects with differences in behavior in a uniform way



Inner Classes

- Trivial class can be declared inside a method:

```
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         ...
      }
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m);
      ...
   }
}
```



Inner Classes

- If inner class is declared inside an enclosing class, but outside its methods, it is available to all methods of enclosing class:

```
public class DataSetTester3
{
   class RectangleMeasurer implements Measurer
   {
      . . .
   }

   public static void main(String[] args)
   {
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m);
      . . .
   }
}
```

Big Java by Cay Horstmann

((iv) 5 Marks)

UML Diagram of `Measurer` Interface and Related Classes

Note that the `Rectangle` class is decoupled from the `Measurer` interface

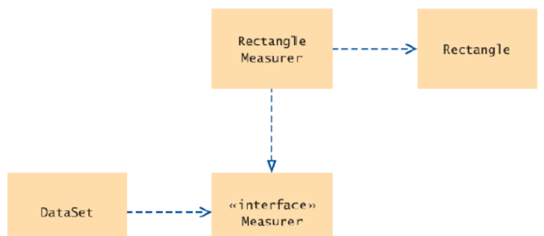
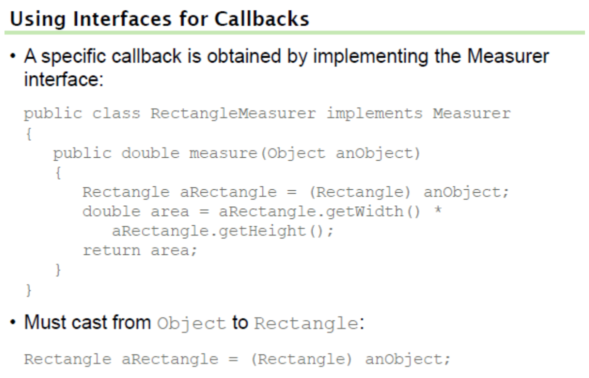


Figure 5 UML Diagram of the DataSet Class and the Measurer Interface



Using Interfaces for Callbacks

- A specific callback is obtained by implementing the Measurer interface:

```
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() *
         aRectangle.getHeight();
      return area;
   }
}
```

- Must cast from `Object` to `Rectangle`:

```
Rectangle aRectangle = (Rectangle) anObject;
```

3. Answer parts (a) to (c). **50 marks in total**

(a) (20 marks)

(i) A technique that can be used to describe the running time of an algorithm is known as *asymptotic notation*. One form of this technique is known as "Big-Oh" notation. Give the definition for this notation. **(3 marks)**

Solution:

The "Big-Oh" Notation (5 marks):

*Given functions **f(n)** and g(n), we say that **f(n)** is **O(g(n))** if and only if there are positive constants **c** and n_0 such that **f(n)**$\leq$ **c g(n)** for all $n \geq n_0$.*

(ii) In the following, use either a direct proof (by giving values for c and n_0 in the definition of "Big-Oh" notation) or cite one of the rules given in the textbook or in the lecture slides:

Show that if f(n) is O(g(n)) and d(n) is O(h(n)), then f(n)+d(n) is O(g(n)+h(n)). **(5 marks)**

Solution:

Recall the definition of the "Big-Oh" Notation: we need constants $c>0$ and $n_0 \geq 1$ such that $f(n) + d(n) \leq c(g(n) + h(n))$ for every integer $n \geq n_0$. $f(n)$ is $O(g(n))$ means that there exists $c_f > 0$ and an integer $n_{0f} \geq 1$ such that $f(n) \leq c_f\, g(n)$ for every $n \geq n_{0f}$. Similarly, $d(n)$ is $O(h(n))$ means that there exists $c_d > 0$ and an integer $n_{0d} \geq 1$ such that $d(n) \leq c_d\, g(n)$ for every $n \geq n_{0d}$. Let $n_0 = \max(n_{0f}, n_{0d})$, and $c = \max(c_f, c_d)$. So $f(n) + d(n) \leq c_f g(n) + c_d h(n) \leq c(g(n) + h(n))$ for $n \geq n_0$. Therefore $f(n)+d(n)$ is $O(g(n)+h(n))$.

(iii) In the following, use either a direct proof (by giving values for c and n_0 in the definition of big-Oh notation) or cite one of the rules given in the textbook or in the lecture slides:

Show that $3(n+1)^7 + 2n\log n$ is $O(n^7)$. **(5 marks)**

Solution:
$\log n$ is $O(n)$
$2n\log n$ is $O(2n^2)$
$3(n+1)^7$ is a polynomial of degree 7, therefore it is $O(n^7)$
$3(n+1)^7 + 2n\log n$ is $O(n^7+2n^2)$ [based on (a)(ii) above]
$3(n+1)^7 + 2n\log n$ is $O(n^7)$

(iv) Order the following functions by asymptotic growth rate:

$4n\log n + 2n$	2^{10}	$2^{\log n}$
$3n + 100\log n$	$4n$	2^n
$n^2 + 10n$	n^3	$n\log n$

(3 marks)

Solution:

$$2^{10}, 2^{\log n}, 3n + 100\log n, 4n, n\log n, 4n\log n + 2n, n^2 + 10n, n^3, 2^n$$

*(v)*Show that if d(n) is O(f(n)) and e(n) is O(g(n)), then the product d(n)e(n) is O(f(n)g(n)). **(4 marks)**

We have, by definition that $d(n) \leq c_1 f(n)$ for $n \geq n_1$, and $e(n) \leq c_2 g(n)$ for $n \geq n_2$. Thus, for $n \geq \max\{n_1, n_2\}$,

$$d(n)e(n) \leq c_1 f(n)e(n) \leq c_1 c_2 f(n)g(n).$$

(b) **(15 marks)**

(i) What is *pseudo code*? List the set of programming constructs that are typically associated with pseudo code.

(5 marks)

Solution:
A simple (non executable) language that allows more abstract descriptions of algorithms that is not intended to be as rigorous as a programming language. Syntax not precisely defined and it combines programming style concepts with mathematical notation and natural language.
Constructs:

*Operator support for expressions: + - * / % & | ! = < > <= >= <>.*
An assignment operator: ←.
If statements: if … then … [else…]

While loops: while … do …
For loops: for … to … do …
Repeat loops: repeat … until … (like do…while … loops in Java)
Array indexing: A[i]
Method Declarations: Algorithm algorithmName(param1, …)
Procedure Calls: algorithmName(param1,…)
Returns: return …

(ii) What does the following algorithm do? Analyze its worst-case running time and express it using the "Big-Oh" notation.

(4 marks)

```
Algorithm Foo (a, n):
    Input: two integers, a and n
    Output: ?
k ← 0
b ← 1
while k < n  do
    k ← k + 1
    b ← b * a
return  b
```

Solution:
The algorithm computes a^n. The running time of this algorithm is O(n) because:

- *The initial assignments take constant time*
- *Each iteration of the while loop takes constant time*
- *There are exactly n iterations*

(iii) What does the following algorithm do? Analyze its worst-case running time and express it using the "Big-Oh" notation.

(6 marks)

```
Algorithm Bar (a, n):
    Input: two integers, a and n
    Output: ?
k ← n
b ← 1
c ← a
while k > 0  do
    if k mod 2 = 0 then
            k ← k/2
            c ← c * c
    else
            k ← k – 1
            b ← b*c
return  b
```

Solution:
This algorithm computes a^n. Its running time is O(logn) for the following reasons:

- *The initialization and the **if** statement and its contents take constant time, so we need to figure out how many times the **while** loop gets called. Since k goes down (either gets halved or decremented by one) at each step, and it is equal to n initially, at worst the loop gets executed n times. But we can (and should) do better in our analysis. (2 marks)*
- *Note that if k is even, it gets halved, and if it is odd, it gets decremented, and halved in the next iteration. So at least every second iteration of the **while** loop halves k. One can halve a number n at most [logn] times before it becomes ≤ 1 (each time we halve a number we shift it right by*

one bit, and a number has [logn] bits). If we decrement a number in between halving it, we still get to halve no more than [logn] times. Since we can only decrement k in between two halving operations (unless n is odd or it is the last iteration) we get to do a decrementing iteration at most [logn] + 2 times. So we can have at most 2[logn] + 2 iterations. This is obviously O(logn). (4 marks)

(c) **(15 marks)**
Bill has an algorithm, find2D, to find an element x in an n x n array A. The algorithm find2D iterates over the rows of A, and calls the algorithm arrayFind, on each row, until x is found or it has searched all rows of A. What is the worst-case running time of find2D in terms of n? What is the worst-case running time of find2D in terms of N, where N is the total size of A? Would it be correct to say that find2D is a linear-time algorithm? Why or why not?

```
Algorithm arrayFind(x,A)
    Input: An element x and an n-element array, A
    Output: The index i such that x = A[i] or
            -1 if no element of A is equal to x
    i ← 0
    while i < n do
        if x = A[i] then
            return i
        else
            i ← i + 1
    return - 1
```

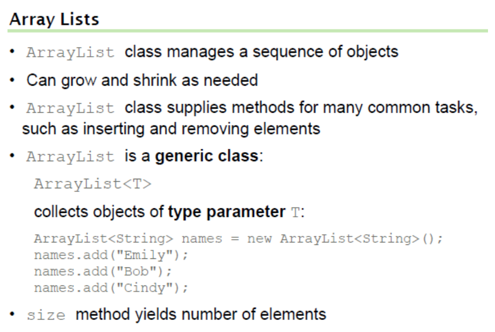
Solution:

The worst case running time of find2D is $O(n^2)$. This is seen by examining the worst case where the element x is the very last item in the $n \times n$ array to be examined. In this case, find2D calls the algorithm arrayFind n times. arrayFind will then have to search all n elements for each call until the final call when x is found. Therefore, n comparisons are done for each arrayFind call. Since arrayFind is called n times, we have $n * n$ operations, or an $O(n^2)$ running time. But the size, N, of A is n^2, so this is also $O(N)$-time algorithm. Thus, this is actually a linear-time algorithm, since its running time is equal to a linear function of the input size.

4. Answer parts (a) to (c). **50 marks in total**

(a) **(15 marks)**
(i) What is an ArrayList? List the methods commonly associated with the ArrayList.

(5 marks)



Array Lists

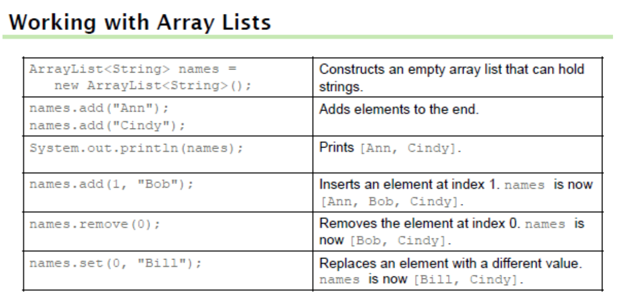
- `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- `ArrayList` is a **generic class**:

  `ArrayList<T>`

  collects objects of **type parameter** `T`:

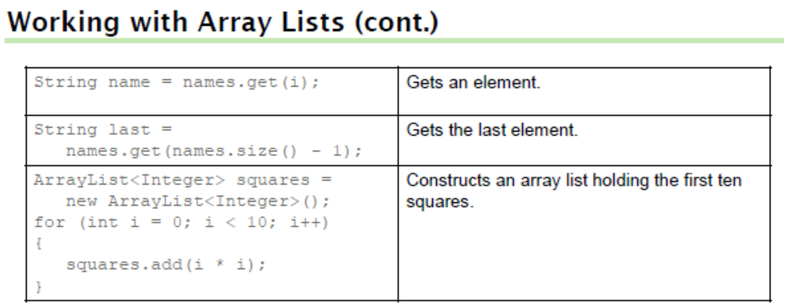
```
ArrayList<String> names = new ArrayList<String>();
names.add("Emily");
names.add("Bob");
names.add("Cindy");
```

- `size` method yields number of elements



Working with Array Lists

`ArrayList<String> names = new ArrayList<String>();`	Constructs an empty array list that can hold strings.
`names.add("Ann");` `names.add("Cindy");`	Adds elements to the end.
`System.out.println(names);`	Prints `[Ann, Cindy]`.
`names.add(1, "Bob");`	Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.
`names.remove(0);`	Removes the element at index 0. `names` is now `[Bob, Cindy]`.
`names.set(0, "Bill");`	Replaces an element with a different value. `names` is now `[Bill, Cindy]`.

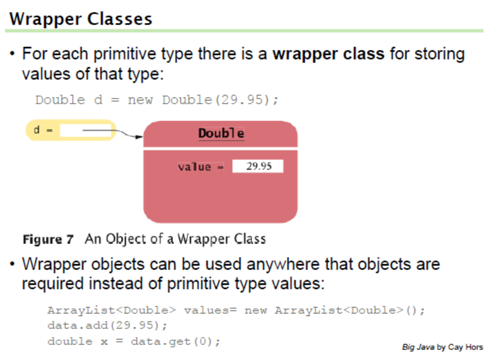


Working with Array Lists (cont.)

`String name = names.get(i);`	Gets an element.
`String last = names.get(names.size() - 1);`	Gets the last element.
`ArrayList<Integer> squares = new ArrayList<Integer>(); for (int i = 0; i < 10; i++) { squares.add(i * i); }`	Constructs an array list holding the first ten squares.

(ii) Briefly define the following terms: Wrapper class, autoboxing and enhanced "for" loop and provide examples for each one.

(10 marks)



Wrapper Classes

- For each primitive type there is a **wrapper class** for storing values of that type:

```
Double d = new Double(29.95);
```

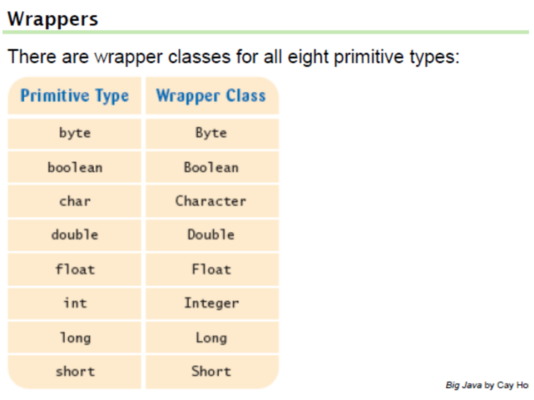
d =
Double
value = 29.95

Figure 7 An Object of a Wrapper Class

- Wrapper objects can be used anywhere that objects are required instead of primitive type values:

```
ArrayList<Double> values= new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
```

Big Java by Cay Hors

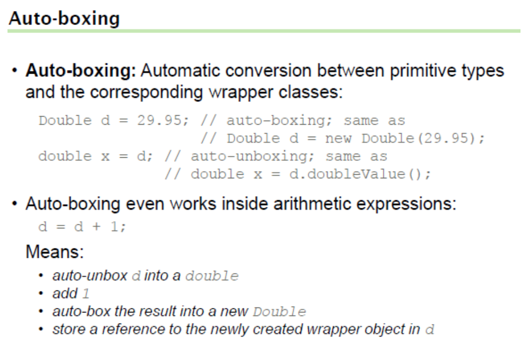


Wrappers

There are wrapper classes for all eight primitive types:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Big Java by Cay Ho



Auto-boxing

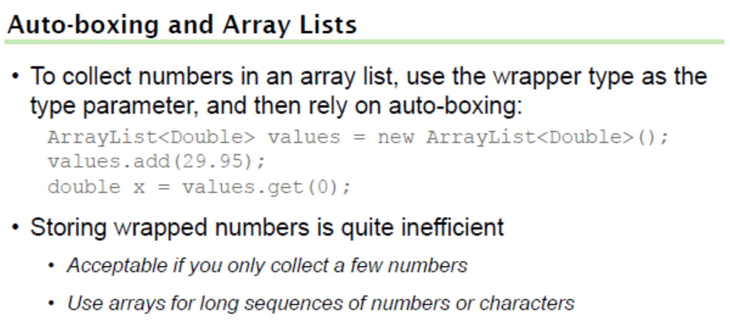
- **Auto-boxing:** Automatic conversion between primitive types and the corresponding wrapper classes:

```
Double d = 29.95; // auto-boxing; same as
                  // Double d = new Double(29.95);
double x = d; // auto-unboxing; same as
              // double x = d.doubleValue();
```

- Auto-boxing even works inside arithmetic expressions:

```
d = d + 1;
```

  Means:
  - *auto-unbox `d` into a `double`*
  - *add `1`*
  - *auto-box the result into a new `Double`*
  - *store a reference to the newly created wrapper object in `d`*

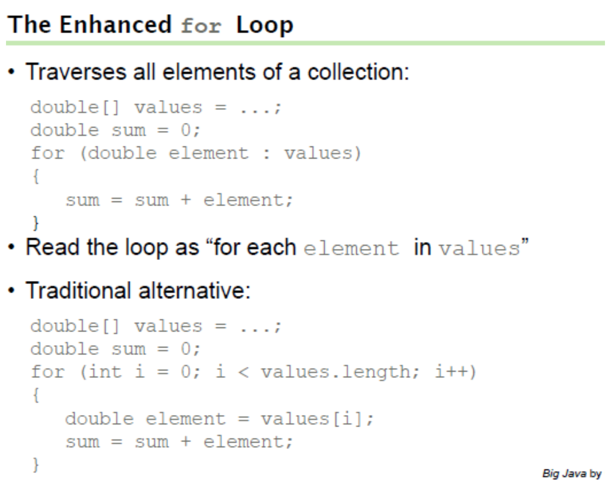


Auto-boxing and Array Lists

- To collect numbers in an array list, use the wrapper type as the type parameter, and then rely on auto-boxing:

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```

- Storing wrapped numbers is quite inefficient
  - *Acceptable if you only collect a few numbers*
  - *Use arrays for long sequences of numbers or characters*



The Enhanced `for` Loop

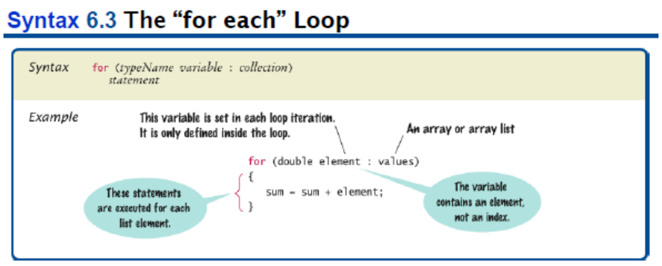
- Traverses all elements of a collection:

```
double[] values = ...;
double sum = 0;
for (double element : values)
{
   sum = sum + element;
}
```

- Read the loop as “for each `element` in `values`”
- Traditional alternative:

```
double[] values = ...;
double sum = 0;
for (int i = 0; i < values.length; i++)
{
   double element = values[i];
   sum = sum + element;
}
```

Big Java by



Syntax 6.3 The “for each” Loop

```
Syntax   for (typeName variable : collection)
            statement

Example
            for (double element : values)
            {
               sum = sum + element;
            }
```

This variable is set in each loop iteration. It is only defined inside the loop.
An array or array list
These statements are executed for each list element.
The variable contains an element, not an index.

(b) **(15 marks)**

(i) What is a *Stack*? List the operations commonly associated with the Stack Abstract Data Type (ADT).

(5 marks)

Solution:
Definition (1.5 marks):A stack is an example of a LIFO data structure. Items can be added at any time, but only the most recently added item can be removed.
Stack Operations (2.5 marks):
push(o) - Inserts object o onto top of stack
pop() - Removes the top object of stack and returns it; if the stack is empty, an error occurs
isEmpty() - Returns the number of objects in stack
size() - Return a boolean indicating if stack is empty.
top() - Return the top object of the stack, without removing it; if the stack is empty, an error occurs.

(ii) Describe the output of the following series of stack operations on a single, initially empty stack: *push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().*

(5 marks)

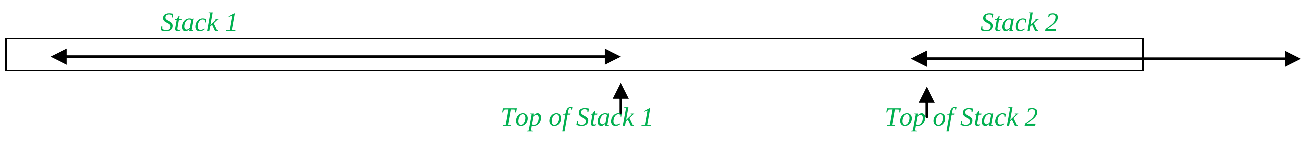
Solution (bottom of the stack is to the left):
5
5 3
5
5 2
5 2 8

5 2
5
5 9
5 9 1
5 9
5 9 7
5 9 7 6
5 9
5 9 4
5 9
5

(iii) Describe how to implement two stacks using one array. The total number of elements in both stacks is limited by the array length; all stack operations should run in O(1) time.

(5 marks)

Solution:
Let us make the stacks (S_1 and S_2) grow from the beginning and the end of the array (A) in opposite directions. Let the indices T_1 and T_2 represent the tops of S_1 and S_2 correspondingly. S_1 occupies places $A[0...T_1]$, while S_2 occupies places $A[T_2...(n-1)]$. The size of S_1 is T_1+1; the size of S2 is $n-T_2+1$. Stack S_1 grows right while stack S_2 grows left. Then we can perform all the stack operations in constant time similarly to how it is done in the basic array implementation of stack except for some modifications to account for the fact that the second stack is growing in a different direction. Also to check whether anyone of these stacks is full, we check if S_1.size() + S_2.size() $\geq$ n. In other words the stacks do not overlap if their total length does not exceed n.



(c) **(20 marks)**

(i) What is a *Queue*? List, giving a brief description for each one, the operations commonly associated with the Queue Abstract Data Type (ADT).

(5 marks).

Solution:
Definition (2 marks):
A queue is a FIFO data structure: items can be added at any time, but only the oldest item can be removed.
Queue Operations (3 marks):
enqueue(o) - Inserts object o onto the rear of the queue
dequeue() - Removes the object at the front of the queue
isEmpty() - Return a boolean indicating whether the queue is empty.
size() - Returns the number of objects in the queue
front() - Return the front object of the queue, without removing it

(ii) What is a *Deque*? Give the pseudo-code algorithm for insertion into the front of a Deque. **(5 marks)**

Solution:
Definition (2 marks):
A Deque is a First or Last In First or Last Out (FLI-FLO) ADT

- *Elements may be added to either the front or back of the deque*
- *Elements may be removed from either the front or the back of the dequeu*

Algorithm (4 marks):

```
Algorithm insertFirst(o):
    // Create the node, setting prev to be first, and next to
    // be the node that is next to first currently
    node := new Node(o, first, first.next)
    // set the prev of the node that is currently next to first to
    // be the new node
    first.next.prev := node
    // set the next of the first node to be the next node
    first.next := node
    size := size + 1;
```

(iii) Describe in pseudo code a linear-time algorithm for reversing a queue Q. To access the queue, you are only allowed to use the methods of queue ADT. *Hint: Consider using a Stack.* **(5 marks)**

Solution:
We empty queue Q into an initially empty stack S, and then empty S back into Q.

```
Algorithm ReverseQueue(Q)
    Input: queue Q
    Output: queue Q in reverse order
    S is an empty stack
    while (!Q.isEmpty()) do
        S.push(Q.dequeue()) do
    while(!S.isEmpty()) do
        Q.enqueue(S.pop))
```

(iv) Another variant of the basic Queue Abstract Data Type is a *Priority Queue*. Explain how this Priority Queue's work. **(5 marks)**

Solution:
Definition
A priority queue is an abstract data type for storing a collection of prioritized elements. It supports arbeitary element insertion but requires that elements be removed based on their priority. Specifically, the remove operation removes the element with the highest priority (which, by convention, is the lowest priority value). In a Priority Queue, elements are stored based on their "priority", and as such, no concept of position exists.